A Doctor Thesis

博士論文

Design and Implementation of Hardware Accelerators with Multi-Level Parallelization and Application-Oriented Data Layout

（マルチレベル並列化とアプリケーション指向データレイアウトを用いるハードウェアアクセラレータの設計と実装）

by

Kenichi Koizumi

小泉賢一

Submitted to
the Graduate School of the University of Tokyo
on December 8, 2017
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Information Science and Technology
in Creative Informatics

Thesis Supervisor: Mary Inaba　稲葉真理
Associate Professor of Creative Informatics

ABSTRACT

In this thesis, we describe the design and implementation of high-performance hardware accelerators and propose a design methodology. In software programming, architecture of the processor, functions of operating system, and optimization of compiler support acceleration. However, in hardware design, there is no support as there is in software programming; thus, hardware engineers must optimize designs on their own. Engineers have greater flexibility when designing hardware; however, a design methodology for high-performance accelerators has not been established. We have established a design methodology through the design and implementation of accelerators using field-programmable gate arrays (FPGAs). To explain the proposed methodology, we select three accelerators and describe three targeted studies.

The first application is improving the throughput performance of TCP communication in long-distance fat-pipe networks (LFNs). It is known that TCP communication on LFNs is difficult, and obtaining good performance in parallel TCP communication is more difficult than with single TCP. To address this problem, we propose a hardware solution that balances streams. We implement the merging stream harmonizer (MSH) on MaSTER-1, our programmable network testbed. We can utilize 99.6% of 10-Gbps LAN PHY bandwidth in pseudo LFNs and 87.0% of 9.2-Gbps WAN PHY bandwidth in real LFNs. Note that different sections of LFNs have various bandwidths, and packet loss often occurs when the total input bandwidth is greater than that of the output. Using MaSTER-1, we analyzed the buffer effects of such switches and the relationship between round-trip time (RTT) and buffer size.

The second study involved the computer Go game. A Monte Carlo tree search method that involves Monte Carlo simulations has been developed to find the best next move in the Go game. The method increases the strength of the Computer-Go program. The effectiveness of this method depends on the number of simulations. Unfortunately, FPGA-based acceleration was difficult because, in this context, resource consumption tends to be high. FPGA-based acceleration was feasible for a 9×9 grid board; however, it was not feasible for a 19×19 grid board. We propose a triple line-based playout for Go (TLPG) hardware algorithm. By reproducing global information redundantly, the TLPG algorithm generates simulations using only local operations, which helps in realizing compact hardware logic implementations. We implemented TLPG in MaSTER-1. The results indicate that the TLPG algorithm can perform 40,649 playouts per second for a 9×9 grid board and 4,668 playouts per second for a 19×19 grid board.

The third study involved skyline computation, which is a method to extract interesting entries from a large population with multiple attributes. When the population changes dynamically, calculating a sequence of skyline sets is referred to as continuous skyline computation. Previous methods that employ divide and conquer and geometric algorithms are not robust in higher dimensional space. We propose the balanced jointed rooted tree (BJR-tree), which can represent a dominance relation as an arc. In addition, tree traversal at a deep position can be delayed to reduce unnecessary calculations. We also propose the low-latency skyline computation accelerator (LSCA) as a hardware algorithm. The LSCA parallelizes dominance relation calculations and evaluates postponed calculations during idle states. We implemented the LSCA on an FPGA and evaluated our software and hardware implementations. BJR-tree is approximately up to 70 times faster than LookOut on synthetic datasets. In addition, the LSCA is approximately 2.5 to 4.4 and 1.7 to 35 times faster than an Intel CPU running software implementations on synthetic and real-world datasets, respectively.

The proposed methodology, established through our studies including the above three, indicates the design flow for multi-level parallelization and application-oriented data layout. Note that our perspective relative to hardware design is not considered in current behavioral synthesis technology. In another three calculations, we compared the performance of a circuit generated automatically using a behavioral synthesis tool and a circuit designed based on the proposed methodology. The results show that the design based on the proposed methodology is more efficient than the behavior-based design. We expect that the proposed design methodology will provide a guideline for hardware designers and will be incorporated into behavioral synthesis in the future. Thus, the proposed design methodology is expected to contribute to an effective and efficient accelerator design.

論文要旨

本研究では、高性能なハードウェアアクセラレータの設計および実装と、その方法論について提案する。ソフトウェアプログラミングでは、プロセッサのアーキテクチャ、オペレーティングシステム、コンパイラ最適化の支援によって実行を高速化できるが、ハードウェア設計ではそのような支援を受けることが難しく、高い自由度のある設計を自ら最適化しなければならない。それにもかかわらず、いまだに高性能なアクセラレータを設計するための方法論は確立していない。我々は FPGA を用いたアクセラレータの設計を通して、その方法論を確立した。その説明のため、我々はこれまで実装した中の三つのアクセラレータを取り上げ、その研究内容を述べる。

一つ目の研究は、広帯域高遅延ネットワーク（LFN）上の TCP 通信の高速化である。一般に LFN 上の単一 TCP 通信による高速データ転送は難しいが、並列 TCP 通信ではより困難となる。そこで我々は並列通信を調停するハードウェア機構 Merging Stream Harmonizer（MSH）を提案する。MSH を我々のネットワーク実験用 FPGA 基板 MaSTER-1 に実装し適用することで、擬似遅延環境の 10Gbps 回線の帯域の 99.6%、現実の 9.2Gbps 回線の帯域の 87.0%を利用するデータ転送を実現した。また LFN の合流地点では、入力帯域幅の合計が出力の合計を超えると輻輳によるパケット損失が発生しうる。我々は回線の往復遅延時間と合流地点のスイッチの持つバッファの関係を MaSTER-1 を使用して解析した。

二つ目はコンピュータ囲碁に関する研究である。より強い一手を求めて提案された、モンテカルロ法を用いるゲーム木探索手法では、プレイアウトと呼ばれるシミュレーションの回数を増やすほど精度が向上する。元々、プレイアウト処理のハードウェアは回路資源の消費量が多く、FPGA を用いた高速化が難しかった。そこで、我々はハードウェアアルゴリズム Triple Line-based Playout for Go（TLPG）を提案する。TLPG では、大域的な情報を冗長に保存し局所的な計算でプレイアウト処理を行うことで、回路資源の消費量を削減する。MaSTER-1 に実装することで、9 路盤において毎秒 40,649 回、19 路盤において毎秒 4,668 回プレイアウトを生成するアクセラレータを実現した。

三つ目は動的に変化する多次元ベクトル集合から逐次的にパレート最適解を求める計算（連続スカイライン計算）の高速化である。これまでの分割統治法や空間的情報を使う手法は高次元データに適していなかった。我々の提案する、エントリ間の関係を有向辺で表現する新たな木構造 BJR 木では、深い頂点の探索を遅延させることで計算量を削減する。また、BJR 木を用いたハードウェアアルゴリズム Low-latency Skyline Computation Accelerator（LSCA）は、エントリ関係の判定処理を並列化し、待機時間に深く木探索を行うことで高速化する。BJR 木は既存の LookOut アルゴリズムより約 70 倍高速であった。また LSCA を FPGA に実装した結果、ソフトウェア実装と比較して、乱数生成データに対し 2.5 倍から 4.4 倍、実データに対し 1.7〜35 倍高速であった。

以上を含むアクセラレータの設計を通して我々が確立した設計の方法論では、複数の粒度で計算を並列化するために並列性の分類とパターン化を行い、アプリケーション指向のデータレイアウトを行う。我々は、具体的な計算処理において、動作合成で生成された回路と我々の方法論に基づいて設計された回路を比較し、我々の方法論の効果を示した。この方法論は、ハードウェア設計者のための指針となるだけでなく、今後より広く利用されていく動作合成技術に将来取り込まれることにより、高性能なアクセラレータの設計に大きく寄与することが期待される。

Acknowledgements

First and foremost, I would like to express my most profound appreciation to my PhD thesis supervisor, Professor Mary Inaba, who is an associate professor, and master thesis supervisor, Dr. Kei Hiraki, a professor emeritus at the University of Tokyo. They guided me throughout my research career and I would not have been able to complete my doctoral course without their support and guidance. Assoc. Prof. Inaba taught me the practical presentation of research, and she mentally supported me during my research life at the university. Prof. Hiraki taught me the importance of the progressive research approach and explained how the continuous accumulation of knowledge from a wide range of fields and not just that limited to own research target is important.

I would like to thank the chair, Professor Masayuki Inaba, and members of my doctoral committee, Professor Reiji Suda, Professor Takeo Igarashi, Professor Kenji Yamanishi, and Senior Lecturer Hideki Nakayama for their essential comments to improve this doctoral thesis. I carried out more in-depth discussions on hardware design methodology. I highly appreciate the time they took and the effort they spent to examine this thesis.

I appreciate the advice and support that I received from Dr. Yutaka Sugawara (IBM Thomas J. Watson Research Center). He introduced me to the exciting FPGA-based design, which led to me designing and implementing circuits on FPGAs. The FPGA-based testbed MaSTER-1 that I have developed under his guidance was used for conducting research on high-speed network communication and acceleration of Monte Carlo Go. I appreciate the support received from Professor Peter D. Eades (The University of Sydney). I enhanced the proposed tree algorithm, refined explanation of our proposals, and my work on the continuous skyline computation became more persuasive through discussion with him. Moreover, because of his helpful guidance, I found the way to over the path to promote my research projects of designing and implementing hardware circuit to a design methodology.

I am deeply grateful to Professor Akira Kato, Mr. Katsuyuki Hasebe, Mr. Takeshi Yoshino, Dr. Hirochika Asai, Mr. Naoki Tanida, Mr. Hisanobu Tomari, Mr. Junji Tamatsukuri (associate professor), Dr. Seiichi Yamamoto (assistant professor), Mr. Hiroshi Tezuka, Mr. Junichiro Shitami (assistant professor), and Mr. Goki Honjo (assistant professor) for their support on network experiments. They supported me in many aspects, including coordination with the agencies and organizations responsible for network paths, management of the experiment schedule, configuring settings for network routers and switches, installation of experimental equipment, and debugging of packet loss and faults in networks. I am indebted to Mr. Kazunori Konishi and Mr. Kees Neggers for their supporting the use of international network circuit. Domestic and intercontinental network experiments were supported by collaborators from many countries, and all names cannot be mentioned here. However, I would like to thank all collaborators of our network experiments.

I am thankful to Dr. Kazuki Yoshizoe, Dr. Takefumi Miyoshi, and Dr. Yasuo Ishii for their technical comments. Discussions with them regarding Computer Go contributed to the research on the acceleration of Monte Carlo Go. They have detailed knowledge of the Go game and hardware circuit design, and the ideas of the proposed hardware algorithm suitable for the rules of Go game were obtained via collaboration with them.

I am indebted to Dr. Yosuke Nakamura and Mr. Kaneharu Nishino for their helpful comments on the design methodology for a high-performance FPGA-based accelerator. I would like to thank all past and present members of Hiraki laboratory and Inaba laboratory, who supported my work and gave me useful suggestions. I would like to acknowledge the laboratory staff, Ms. Naoko Motohashi for always supporting my research activities. I would like to thank all professors of Department of Creative Informatics for valuable comments from a broad perspective in the seminar.

Finally, I would like to thank my parents for their continuous support.

Contents

List of Figures

List of Tables

Chapter 1

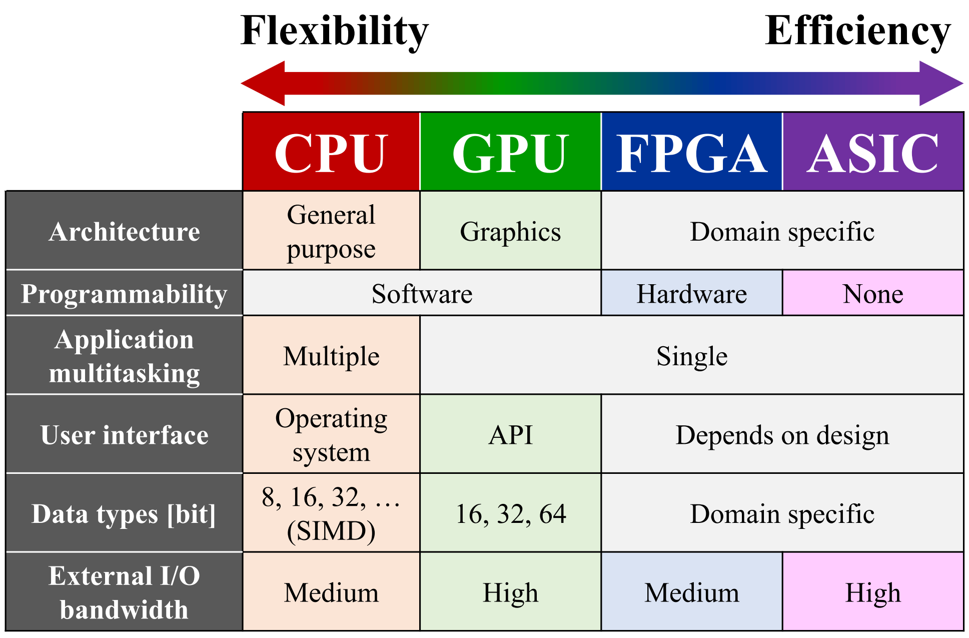
Introduction

1.1 Research Background

Recently, it has been observed that computer research and development focuses on high performance and low power consumption. CPU performance has improved by increasing the clock frequency and by introducing multi-core technologies based on the progress of semiconductor technology and microarchitecture design. However, recently, performance saturation has been observed. High-performance computing targets a wide range of applications, such as scientific computation, artificial intelligence (AI), machine learning, big data, and high-speed computer networks. These applications require significant computational resources to increase the accuracy of their results and increase the size of problems. From the above consideration, methods to accelerate large-scale computations are neccesary to overcome the saturation of CPU performance.

CPU architecture is designed in a manner so as to achieve high performance for a wide range of applications. Therefore, CPUs are not the most suitable solution for specific applications. CPUs are required to run multiple applications simultaneously under the management of operating systems. Then, several CPUs have functions for context switching at the hardware level. Therefore, accelerators to reduce the execution time and power consumption of specific large-scale applications have been extensively studied, e.g., accelerators that employ graphic processing units (GPUs), field-programmable gate array (FPGAs), and application-specific integrated circuits (ASICs). These four computing approaches, i.e., CPU and GPU-, FPGA-, and ASIC-based accelerators, are shown in Figure 1.1. Note that each approach has different characteristics. The use of CPUs is the most versatile and flexible approach. Originally, GPUs are hardware-based accelerators that targeted graphics processing; however, GPU architecture has been generalized to include software programming environments. Thus, GPUs can be employed to speed up several calculations at a low cost. GPUs have software programmability and make up the second most flexible approach. An FPGA is an LSI whose internal circuits can be reconfigured dynamically. FPGAs that have hardware programmability constitute the second most efficient approach. Further, ASICs target specific applications. ASICs have the lowest programmability; however, they are the most efficient. Accelerators that employ GPUs, FPGAs, and ASICs have been studied, and various commercial accelerators to obtain high-performance computing systems for an extensive range of applications have been developed.

In ASICs, execution circuits, data paths, and the memory structure are optimized for specific target applications. However, such hardware optimization incurs high development costs and requires significant time. The clock frequency


Flexibility
Efficiency

	CPU	GPU	FPGA	ASIC
Architecture	General purpose	Graphics	Domain specific	
Programmability	Software		Hardware	None
Application multitasking	Multiple	Single		
User interface	Operating system	API	Depends on design	
Data types [bit]	8, 16, 32, … (SIMD)	16, 32, 64	Domain specific	
External I/O bandwidth	Medium	High	Medium	High

Figure 1.1: Computing approaches; general-purpose CPUs, and GPU-, FPGA-, and ASIC-based accelerators

of ASICs can be increased because of optimizations in chip manufacturing processes; thus, such devices can provide the highest efficiency and performance. However, ASICs have several disadvantages: unfortunate design errors result in disastrous development delays, and when target algorithms change or improve, ASIC chips, which are expensive to manufacture, become obsolete, and modifying the design of manufactured chips is costly. ASICs are designed for specific applications, and it is assumed that the ongoing demand for such chips will be sufficient to ensure profitability. Given the time require to design and manufacture ASICs, the development cycles of competitive devices, such as CPUs, must also be considered, i.e., a newly released ASIC may be inferior in performance to current or next-generation CPUs. Thus, ideally, accelerators should have short development periods. As alternative approaches, software-programmable GPUs and hardware-programmable FPGAs can be employed to compensate against the disadvantages of ASICs.

Originally, ASICs were used to increase the speed of 3D graphics processing. As graphics processing requirements become increasingly diversified, more flexible devices were developed. While data paths and the memory hierarchy were maintained specialized for the graphics processing, execution circuits were generalized and software programmability is introduced. For example, a shader with programmable color rendering circuits is referred to as a *programmable shader*. In addition to graphics processing, GPUs can achieve high computational performance for various operations, e.g., matrix operations. Technology to employ GPUs for purposes other than graphics processing is referred to as general-purpose computing on GPUs (GPGPUs). Initially, GPUs were used for scientific computations; they are now used in a wide range of fields, such as finance, deep learning, climate simulation, multimedia, and entertainment. More than 40% of the 48 systems with accelerators that were ranked within the Top 100 in the Top500 List employ GPUs [128]. Typically, the GPU architecture includes many floating-point and integer arithmetic units. Registers, a scheduler, and cache memory are associated with each group of several arithmetic and floating-point units. Unlike CPUs, GPUs do not have an operating system, and GPU computations are initiated by requests from a host server. The GPU usage has increased dramatically because of the demands of deep learning. In 2017, in response to the emerging deep learning requirements, NVIDIA announced the Volta microarchitecture. The Volta microarchitecture includes Tensor Cores, which are circuit elements dedicated to perform 4×4 matrix multiplication and addition. Typically, GPUs have large memory bandwidths and are useful for applications wherein the data paths, memory hierarchy, and specialized execution units of the GPU are suitable. However, there are also applications wherein GPUs are not useful, e.g., tree search that includes many conditional branches. It is difficult to optimize power consumption for each algorithm in a GPU, and their performance and efficiency are limited.

In FPGAs, both logic and wiring can be reconfigured. Originally, FPGAs were based on programmable logic devices (PLD). Motorola's XC157, which appeared in 1968 [58], had 12 gates and could be configured only once. Lattice Semiconductor's generic array logic (GAL), which was developed in the 1980s, had memory for the configuration and it was reconfigurable. In 1985, Xilinx released its first FPGA device, i.e., the XC2064, which had 800 gates [104]. Initially, FPGAs were primarily used to realize easy firmware updating of hardware requiring real-time processing, such as the control units of instruments for computer networks and experiments in physics. Since the 2000s, owing to developments in semiconductor technology and development environments, FPGAs have been used as acceleration devices to replace ASICs in commercial products and research instruments. Currently, attention is focused on the use of the FPGAs in AI. As mentioned previously, ASIC hardware designs are not changeable, and the scope of software programmability tends to be narrow in the process of pursuing high performance. For example, a software programming framework CUDA is developed by NVIDIA for GPGPUs; however, engineers cannot change GPU hardware designs. On the other hand, FPGAs are programmable at a hardware logic level, and it differs from CPUs, GPUs, and ASICs. Therefore, FPGA designers can freely determine the scope of software programmability. In addition, the FPGA logic circuits can be modified on the fly. The FPGA approach has two primary advantages, i.e., specialization for individual application, which is also achievable by ASICs, and hardware-level programmability, which is not achievable by ASICs. FPGAs have several disadvantages. Although recent FPGA devices have high-speed transceivers, the I/O bandwidth for external devices is inferior to that of an ASIC. In addition, for the same circuit, the clock frequency and degree of gate integration are lower than those of ASICs.

The CPU pursues flexibility through a general-purpose hardware design and software programmability, and the GPU demonstrates high performance locally for graphics and AI processing, including matrix operations. However, domains where GPUs are effective are limited. Since the ASIC is designed based on the execution unit, data paths, and memory hierarchy optimized for a target application, it achieves the highest performance and efficiency. Similarly, FPGAs can be optimized for target applications. For example, data types are fixed in CPUs and GPUs; however, data types can be adjusted to the required bit width in ASICs and FPGAs. Furthermore, only ASIC- and FPGA-based approaches can exploit application-oriented circuit designs.

Figure 1.2 shows the design difficulties relative to software- and FPGA-based approaches. In the software-based approach, the hardware design is fixed, and the operating system is often off-the-shelf. Note that software programmers primarily design applications and libraries. On the other hand, in FPGA-based

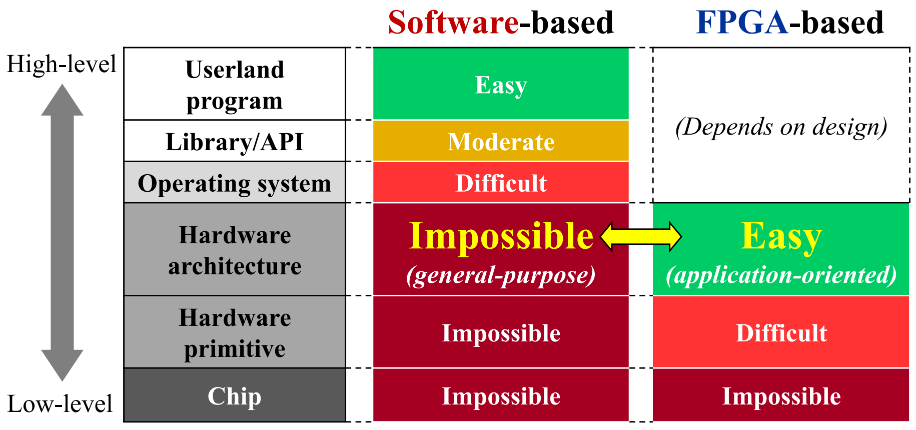


Figure 1.2: Design difficulties for developers. In the software-based approach, software programmers can design only userland programs and libraries easily. It is impossible for the programmers to design the hardware architecture and optimize it for the target application. On the other hand, hardware designers can easily design all components other than FPGA chip and hardware primitives including the hard-macros. The hardware designer may create a software program that communicates with the FPGA device. Its difficulty depends on the design of the FPGA device, but it is often a simple program for controlling the FPGA.

accelerators, only the logical block arrangement of the FPGA and the circuit of the built-in hard macro are fixed. Hardware engineers can freely select the architectural design of the hardware, which includes the data paths connecting execution units and memory hierarchy configurations. In addition, hardware engineers determine the design of the software controller for FPGAs. Thus, the design of FPGA-based accelerators is more flexible than software programming. In this thesis, we discuss a methodology for balanced design with high degrees of freedom through selected three studies.

The microarchitecture of a computer system comprises components for instruction interpretation, execution, and interconnection (Figure 1.3). Note that architecture designers must pursue high performance in all components. Many engineers have researched and developed microarchitecture designs, and a number of thoughtful designs have been proposed for each of the three components. The performance of general-purpose processors has been improved by increasing the clock frequency because of the progress of semiconductor technology and by improving the microarchitecture. We can give examples for the instruction interpretation component, such as the superscalar architecture [124], the branch prediction algorithm [90, 65], out-of-order execution [127], multi-core architecture, and simultaneous multi-threading [56]. We can also provide examples for the interconnection component, such as the three-dimensional torus interconnect in Cray T3D [70] and the Tofu six-dimensional torus interconnects in the RIKEN K computer [1]. Finally, we can give two examples for the execution component, i.e., high-throughput floating-point units [129, 132] and a high-throughput matrix operation unit with a combination of the recursive Strassen algorithm [114] and a specialized small matrix operation circuit (e.g., the NVIDIA Tensor Core). The execution component has been solely improved for general-purpose computations to maintain processor versatility. However, a design methodology for the execution component specialized for specific applications has not been discussed systematically; therefore, engineers tend to design the execution component for each application with significant cost and time.

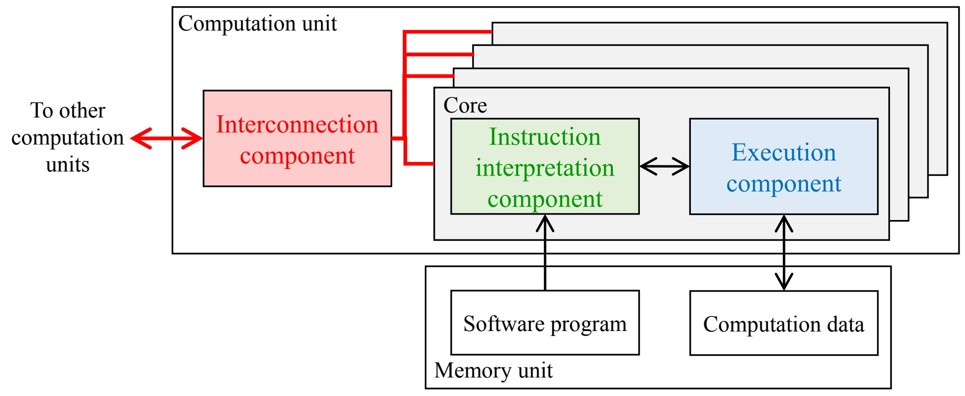

Computation unit
To other computation units
Interconnection component
Core
Instruction interpretation component
Execution component
Software program
Computation data
Memory unit

Figure 1.3: Three microarchitecture components of the computation unit. The instruction interpretation component is a circuit for interpreting programmable software code loaded from memory to the device and controlling execution components. The execution component is a circuit for computing target calculations. The interconnection component consists of data paths that connect cores and interfaces to other computation units.

1.2 Research Overview

We have developed a series of FPGA-based accelerators. When designing those accelerators, we found important keys for high-performance accelerators, and we have established a design methodology based on these points. We select and describe the following typical three studies and accelerators that are suitable for explaining the proposed design methodology: (1) high-throughput TCP communication with parallel streams in computer networks, (2) speedup of playout generation for a computer Go algorithm, and (3) acceleration of continuous skyline computations.

We discuss accelerators for high-speed data processing in computer networks in Chapter 3. When realizing network data processing at wire-speed throughput using a software approach, we often face a lack of computing power with the CPUs that are available at the same age. Furthermore, new network functions are proposed and implemented to improve the performance and network functionality, and they are evaluated 'on-site'. Some functions will be maintained, but the others will be obsolete. New functions must be implemented in network equipment in a short period. When vendors manufacture specialized ASICs, they cannot quickly release new versions of chips and earn high profits. Note that the CPU and ASIC approaches do not work effectively in the domain of high-speed computer networks. Thus, historically, FPGAs have been installed in network instruments, such as the switching facilities of a network service provider and network interfaces in edge servers. Recently, the I/O function of FPGAs has been enhanced, and FPGA devices have implemented high-speed transceivers. As a result, we can handle the wire-speed network streams of state-of-the-art standards directly within an FPGA. The FPGA-based approach has become a central method in network device development. The TCP is a data transfer protocol that guarantees data arrive at the target destination without loss. The TCP is currently used as a standard protocol on the Internet; however, with the widespread use of long-distance fat-pipe networks (LFNs), we have not been able to realize high-throughput data transfer in TCP communication. It is well known that throughput performance decreases because of instantaneous increases in Internet traffic for a short period, and this is referred to as the burst traffic problem.

Thus, we addressed the burst traffic problem in LFNs. Since the RTT of LFNs is large, the burst problem becomes more serious. Note that improving the TCP and its congestion control algorithm are insufficient to solve this problem. We must analyze the behavior of network streams and directly control the packets. For example, the time available to handle a single 64-byte packet in 10-Gbps networks is no more than approximately 50 nanoseconds, which means that it is difficult to analyze and control streams using software-based approaches. Moreover, no current hardware can control streams at these speeds. Therefore, we developed the MaSTER-1 FPGA-based network testbed and analyzed network streams in various networks. We can produce our testbed at low cost in short development time using FPGAs. By capturing and analyzing network streams in real-world networks, we can observe interesting situations that cannot be reproduced in a pseudo network environment simplified and realized by a network simulator. Analysis with dedicated testbeds having high-throughput data transmission capability is key to perform high-speed communication. We propose the MSH packet control mechanism in Chapter 3. We introduced the MSH to the networks and evaluated the performance improvement of TCP communication.

We also discuss high-speed Monte Carlo simulation for a computer game AI in Chapter 4. Accelerators have been actively utilized in computer game AI. For example, IBM's Deep Blue, which defeated the Chess world champion, Garry Kasparov, in 1997, had 512 very-large-scale integration (VLSI) accelerators specialized in Chess [24]. Google's Deep Mind AlphaGo, which won the top world Go game player in 2016, employed four tensor processing units (TPU), which are ASICs specialized to perform neural network computations. Application-oriented hardware-based accelerators are highly effective in the field of computer game AI. With Go, the introduction of the Monte Carlo tree search method [35] in 2008 improved the strength of computer Go players. In Monte Carlo tree search, rather than constructing an evaluation function for the state of the game board, random moves are repeated until the game ends (this is referred to as a *playout*). Since playout generation requires very large computing resources and gameplay has a time limit, we must speed up this computation.

With Reversi and Chess, the games are compatible with 64-bit processors because the size of their game boards is small (64 squares). However, Go's board has 361 squares, and playout generation comprises complicated bit operations to judge stone death based on the rules of the game. In addition, the computations contain practically no floating-point operations. Thus, it is difficult to accelerate playout generation with GPUs. However, since the bit width can be fitted to the game board size and a circuit can be optimized for the rules of Go, hardware-based approaches, such as FPGAs and ASICs, are very suitable for playout generation. We used FPGAs to construct an acceleration system at low cost and short development time. FPGAs are not suitable for tree searches because the tree search requires many random memory accesses and conditional branches; thus, it is not a good approach to implement the entire Monte Carlo Go AI on an FPGA. This weakness is solved by dividing the Monte Carlo tree search into simple tree search and playout generation tasks and by executing each task in the software and FPGA hardware, respectively. However, it is difficult to design a high-performance circuit for playout generation due to Go's rules. We describe our hardware design for playout generation in Chapter 4.

We discuss high-speed continuous skyline computation relative to accelerators in Chapter 5. A skyline of a set of multi-dimensional vectors is a subset of multi-dimensional vectors such that no vector having a smaller value than themselves exists in any dimension. Intuitively, entries superior to a skyline entry in all dimensions do not exist in the set. Computing a skyline in d-dimensional space is the same as finding the Pareto optimal solutions of a multi-objective optimization problem of d functions. Skyline is both similar to and different from the convex hull, and generally, time complexity of skyline computation is less than that of the convex hull. However, as the number of dimensions increases, the time required for the skyline computation increases significantly. Recently, incremental skyline computation for dynamically changing the vector sets has been focused. The computation is referred to as continuous skyline computation. In real-time applications, it is necessary to compute the skyline of a changing set immediately. Therefore, acceleration of continuous skyline computation is more important than that of the static skyline computation.

We propose the BJR-tree, a software algorithm for high-speed continuous skyline computation. The design and implementation of an efficient online algorithm for a time-varying entry set are the keys to low-latency query processing in continuous skyline computations. In the BJR-tree algorithm, vector entries are managed by a unique tree structure to delay calculations of entries that have less potential to join the skyline in the future. Then, we propose the LSCA, a BJR-tree-based hardware algorithm, and implement it in an FPGA. We describe our hardware design for continuous skyline computations in Chapter 5.

FPGAs can be employed to develop application-specific accelerators that can perform highly parallelized computations based on low-latency and real-time operations. Differing from software programming languages, in the hardware description language (HDL), the structure of register transfer level (RTL) circuits, including execution units, data paths, and the memory hierarchy, is described directly. Recently, a behavioral synthesis technology for converting the "behavior" of algorithms, described using a high-level language, into a "structure" of the circuits, described using an HDL, has been developed. However, converting the imperative descriptions of a target function to logical structures is difficult, and generation of a high-performance HDL code for acceleration has not been achieved.

We reviewed the implementations of our FPGA-based accelerators and established a design methodology in Chapter 6. We describe a design flow to represent abstract algorithms as a logic circuit. The core concepts of the proposed design methodology are multi-level parallelization and an application-oriented data layout. We describe principles applicable to logic design of steps of the design flow. Additionally, to demonstrate how the proposed methodology can be employed with two core concepts, we describe the detailed hardware algorithm of accelerators implemented in our three typical studies, which are mentioned in the previous chapters. We compare the performance of behavior-based and HDL-based design approaches to evaluate the effectiveness of HDL-based approach using our design methodology.

1.3 Organization of the Thesis

The remainder of this thesis is organized as follows. In Chapter 2, we describe processors and accelerators and review the research background of accelerators in high-performance computing. We introduce large-scale GPU-, FPGA-, and ASIC-based acceleration systems as existing notable studies. In addition, we review fundamental parallelism concept and available memory hierarchy, which are central elements in acceleration. In Chapters 3, 4, and 5, we describe our three studies in detail. In Chapter 3, we describe the performance improvement of parallel TCP communication in LFNs. In Chapter 4, we address the acceleration of the Monte Carlo tree search algorithm in a computer Go player AI, and in Chapter 5, we focus on the acceleration of continuous skyline computations. In each study, our approach is based on an efficient architecture and an algorithm for an FPGA-based accelerator. In Chapter 6, we describe the proposed design methodology for realizing high-performance FPGA-based accelerators. We clarify the difficulties associated with hardware design and briefly review previous hardware design methods. Then, we describe the proposed design methodology and demonstrate its application. We consider additional applications and compare a hardware design based on the proposed methodology and a behavior-based hardware design relative to the performance of the applications. Source codes for the behavior-based design are described in Appendix A. Additionally, we clarify the issues to be addressed in future hardware designs. Conclusions and suggestions for future work are in Chapter 7.

Chapter 2

FPGA-based Acceleration

2.1 Background

Currently, research and development are focused on developing high-performance computing and reducing power consumption requirements. High-performance computing targets a wide range of domains, such as scientific computation, artificial intelligence, machine learning, big data, and computer networks, which require vast computational resources to increase computational accuracy and scale. In addition, other domains, such as financial transactions, require very low-latency query processing. Traditionally, the performance of a general-purpose computer has depended on the central processing unit (CPU), and CPU performance has been improved by increasing the clock frequency and introducing multi-core and multi-thread technologies. However, CPUs are designed to ensure performance in general-purpose applications and are not optimized for specific applications.

Various accelerators, e.g., GPUs, FPGAs, and ASICs, have been proposed to reduce the execution time and power consumption of specific large-scale computations. Note that CPUs have greater generality and flexibility than GPUs, FPGAs, and ASICs. On the other hand, ASICs demonstrate higher performance capability and efficiency than CPUs, GPUs, and FPGAs.

GPUs were originally designed for image processing as a simple computer component. However, application programming interfaces (APIs) such as the NVIDIA CUDA has made it possible for end users to run their software on GPUs, and many GPU products specifically intended for high-performance computing are available from various vendors. Similar to CPUs, end users cannot modify GPU hardware logic; however, both CPUs and GPUs can be programmed at a software level. A GPU can be considered a processor dedicated to the code with API. GPUs are used to accelerate the performance of supercomputers. GPUs can realize high performance and efficiency on applications that consist of matrix operations and on calculations in which similar operations can be highly parallelized. However, GPUs are not suitable for computations that involve complicated conditional branches. ASICs differ from CPUs and GPUs in that they are dedicated large-scale integration (LSI) chips designed for specific applications in consideration of performance and power consumption. However, developing ASICs requires significant time, and to be profitable, chips must be mass-produced. Changing ASIC designs at an advanced production stage incurs additional cost. FPGAs are programmable LSIs. As mentioned previously, CPU, GPU, and ASIC hardware designs are fixed for the programmers and the end users; however, FPGAs can be programmed at a hardware logic level. Unlike ASICs, we can modify hardware logic circuits on FPGAs on the fly. The FPGA approach has two advantages.

One is specialization for applications, which is achieved by ASICs, and the other is hardware-level programmability, which is not achieved by ASICs.

With CPUs, software programmers have greater flexibility because they are general-purpose devices that can be programmed to target a wide range of applications. On the other hand, the ASIC approach attempts to improve performance and efficiency using specialized and unmodifiable hardware devices. Although the CPU approach is superior in terms of cost, its performance is limited. Recently, methods that employ GPU-, FPGA-, and ASIC-based accelerators have become common in large-scale cluster computing systems.

2.2 Field-Programmable Gate Array

FPGAs are reconfigurable logic devices, i.e., the logic circuits in an FPGA are programmable. The FPGA architecture (Figure 2.1) includes basic Logical Blocks (LBs) that realize combinational and sequential circuits. Routing channels are interconnection resources that connect LBs, and the connection relationships are configured using Switching Blocks (SBs). User configuration data specify truth tables in LBs and routing information in SBs. Similar to other LSI devices, FPGAs have external I/O pins that connect internal routing resources via I/O Blocks (IOBs). Users can configure I/O pins to conform to the electrical requirements of the various devices connected to an FPGA.

Currently, Look Up Table (LUT) method is used for realizing LBs in many types of FPGAs. In addition to LBs, FPGAs employ Digital Signal Processors (DSPs), Random Access Memory (RAM), Serializers/Deserializers (SerDes), Delayed Locked Loops (DLLs), Phase Locked Loops (PLLs), and other circuit blocks that realize I/O interfaces for various protocols, such as Ethernet. The DSP block consists of a floating-point adder and multiplier. Using DSPs has advantages relative to resource consumption and logic delay time compared to using LBs to implement floating-point units. Note that specialized circuit blocks are referred to as *hard-macros*. In FPGAs, hard-macros perform various functions, such as Ethernet Media Access Control (MAC)/Physical Coding Sublayer (PCS), exter-

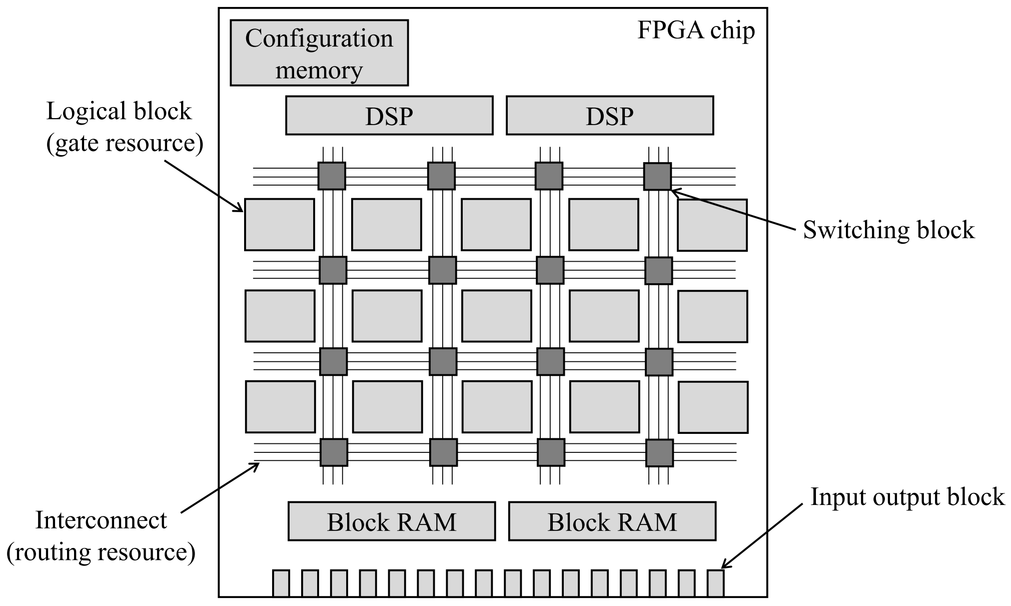


Figure 2.1: FPGA architecture

nal DRAM control, PCI Express interfaces, and analog/digital (A/D) converters. The corresponding hard-macro blocks are instantiated when a circuit includes specialized functional blocks, and routing resources are automatically assigned based on the circuit design. The placement of logical block instances and routing information are referred to as the FPGA *firmware*. After downloading firmware to an FPGA device, the internal circuits are determined, and the FPGA initiates the designed operation. Many FPGA boards implement non-volatile flash memory connected to configuration pins of the FPGA. When the FPGA and board are powered up, the firmware in flash memory is downloaded automatically. FPGA functions can be changed during operation by downloading different firmware.

FPGAs were initially developed in the 1970s as a simple reconfigurable device. In the 2010s, 20-nanometer semiconductor-technology-based FPGAs with 50 million ASIC gates became available. Currently, the major FPGA device manufacturers are Xilinx, Intel, Lattice, and Microsemi. Recently, Xilinx and Intel have deployed System-on-Chip (SoC) FPGA devices integrated with embedded processors, such as ARM CPUs. In addition, highly integrated FPGAs that utilize three-dimensional stacking technology are currently being developed, and the resource capacity of FPGAs is increasing. Thus, FPGAs are expected to be employed more extensively in the future.

Compared to CPU-, GPU-, and ASIC-based approaches, the advantages of the FPGA approach are as follows.

1. **Low-latency interface**

   In CPUs, data I/O is managed by hardware, the operating system, and software. Some devices connected to a CPU require driver software, and I/O throughput and latency strongly depend on the quality of the device driver. In addition, relative to the I/O of the interface between a CPU and GPU, the CPU and operating system manage PCI Express data transmission, which can incur performance bottlenecks. On the other hand, data I/O is managed by the userland logic on FPGAs, and logic latency depends on the user's design. In other words, the FPGA I/O interface is not influenced by vendor software, such as device drivers.

2. **Real-time processing**

   In CPUs, the operating system manages process scheduling and resource sharing. If competing resource demands occur, a time-critical process may be halted. It is well known that operating system context switching and interruptions cause fatal delays for real-time applications, and these delays cannot be controlled by software programmers. An operating system is not essential in an FPGA because processes are executed in real time and are performed as designed.

3. **Parallelization**

   In a non-superscalar processor, only one instruction can be processed per single clock cycle on a single core. The number of CPU cores and threads is predetermined; therefore, the degree of parallelism that can be achieved through software design is limited. On the other hand, hardware designers can implement circuits with high degrees of parallelism if sufficient logic resources are available. Thus, FPGAs outperform CPUs operating at higher clock frequencies.

4. **Flexible data structure**

In CPUs and GPUs, the bit widths of internal registers, instructions, and alignments are predetermined. For example, in a 64-bit processor, even if the range of values handled in computations is limited, the circuits will be driven for 64-bit processing. Note that the bit width can vary in FPGAs. In other words, circuit bit width can be determined relative to the scale of the target application. Therefore, the FPGA approach can achieve high power efficiency.

5. **Low development cost and fast development speed**

   Differing from FPGA hardware development, ASIC development must include a chip manufacturing phase. Thus, if the chip is not mass-produced, the FPGA approach can reduce development costs and time. Furthermore, FPGA circuits can be changed easily; thus, the FPGA approach reduces the cost of specification and requirement changes. Note that FPGAs are typically used for the evaluation of prototype design in the development of the ASIC.

The disadvantages of the FPGA approach are as follows:

1. **Low operating frequency**

   In FPGAs, the resources of the LBs and routing wires are prearranged and fixed before the configuration. FPGA firmware consists of instantiation information relative to prearranged resources. FPGA circuits tend to have longer logical gate and wiring delays than ASIC circuits where placement and routing are optimized for specific applications. Therefore, the clock frequency of an FPGA is typically less than that of CPUs and ASICs.

2. **Limited logic resources**

   In a given FPGA device, the number of available LBs is fixed; thus, circuits must be designed in consideration of the capacity of the given FPGA device. With the same semiconductor technology and chip size, the number of gates on an FPGA tends to be less than that of an ASIC.

3. **Power consumption**

   The circuits of an ASIC are optimized to reduce power consumption. However, with FPGAs, circuit power consumption strongly depends on optimizations determined by synthesis tools.

Note that the first and second disadvantages are critical issues that directly affect FPGA-based acceleration.

2.3 Related Work

2.3.1 High-performance Computing Accelerators

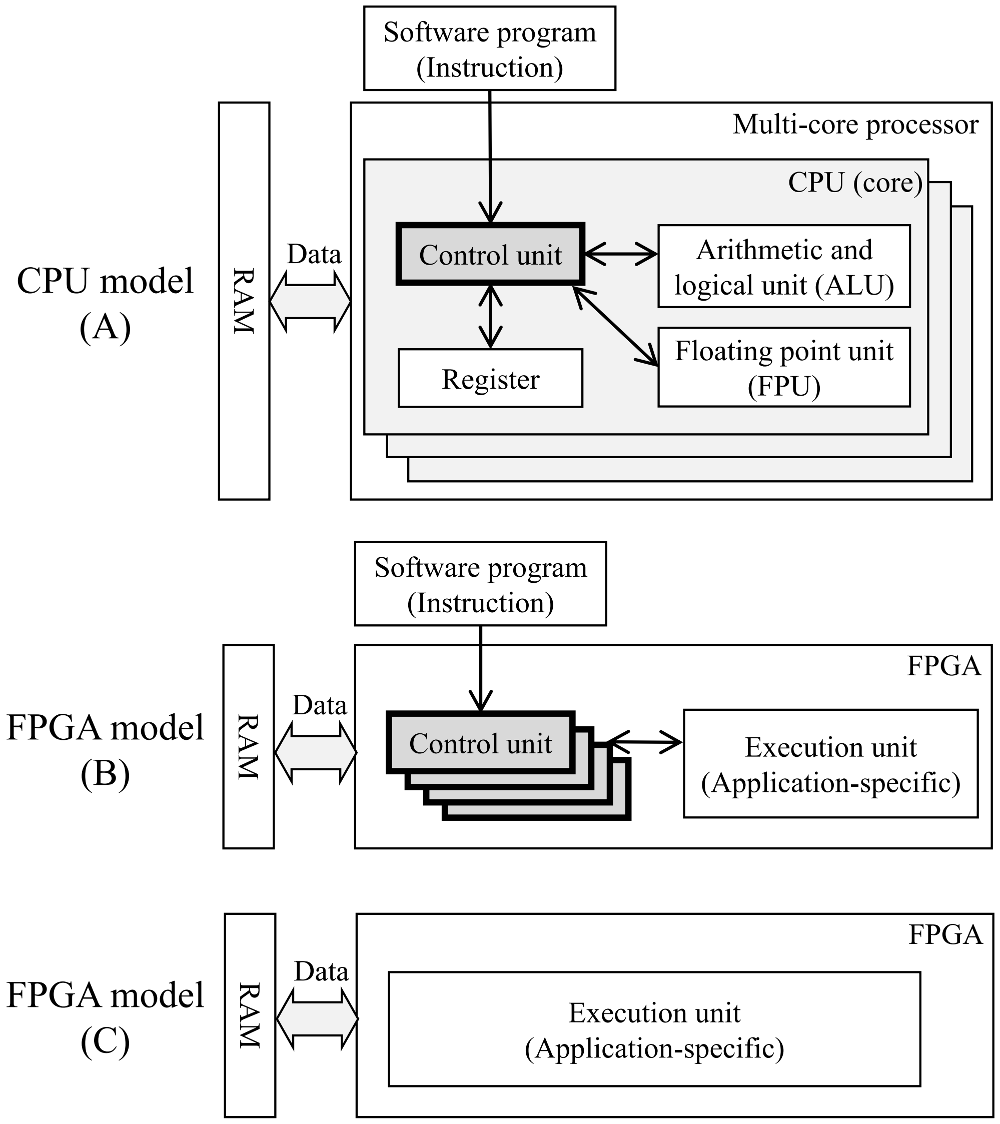
Acceleration technologies using GPUs, FPGAs, and ASICs are well established, and many accelerators, all of which attempt to achieve high performance and high efficiency, have been studied and developed for various research and commercial fields.

GPUs are commonly used in computing systems to speed up applications, such as finance, deep learning, climate simulation, multimedia, and entertainment applications. As mentioned previously, GPUs are effective for matrix calculations, particularly dense matrix calculations. However, the GPU approach is not suitable for tree searches that include many conditional branches. Therefore, the range of application domains for which GPU-based acceleration is effective is limited.

Recently, many FPGA-based accelerators have been developed. For example, Microsoft's Catapult [102] and BrainWave [32] are FPGA-based accelerators for search engines and the deep learning, respectively. The BrainWave system includes Intel Arria 10 FPGAs or Intel Stratix 10 FPGAs, which implement execution units for deep neural networks (DNNs). Users can run a Long Short-Term Memory network, which is a type of Recurrent Neural Network, using the BrainWave system. The Baidu XPU [96], which is an FPGA-based accelerator for DNNs, can also be utilized for search engine services. IBM and Xilinx [13] have developed a system that uses a POWER8 processor and FPGA chips to accelerate Memcached, which is an open-source distributed memory caching system, used to speed up database access. IBM and Xilinx have developed an FPGA-based implementation of Memcached. In addition, Netcope has released a network interface card that implements an FPGA for network traffic monitoring [101]. This product provides packet capturing capability at 100 gigabits per second and can be used for various other network applications. In addition, the GRAPE project has developed cluster systems, e.g., GRAPE-8 [88], that use FPGAs and ASICs for large-scale simulation of the N-body problem. A GRAPE-8 computation board includes two GRAPE-8 ASICs and a single FPGA that controls the PCI Express interface between the computation board and a host. Intel's Nervana Neural Network Processor [93] and Google's Tensor Processing Unit [66] employ ASICs dedicated to machine learning. In the FPGA and ASIC approaches, hardware designers can employ application-specific execution units and data structures. Therefore, FPGA- and ASIC-based accelerators are active in various application domains.

2.3.2 FPGA-based Accelerators

A CPU and two FPGA-based accelerator implementations are shown in Figure 2.2. In the CPU implementation (top), a general multi-core processor consists of a control unit (CU), registers, arithmetic and logical units (ALUs), and floating-point units (FPUs). The CU interprets software read from external memory. As shown in Figure 2.2, FPGA-based accelerators can be roughly divided into two types, i.e., models (B) and (C). In a type (B) FPGA-based accelerator, computations for which hardware acceleration is effective are implemented as an execution unit with RTL coding. The type (B) accelerator has a hard or soft processor core. Here, the FPGA receives input data for computations and program to operate the processor. Note that the program is coded based on the instruction set specification of the processor core. Note that the type (B) model is similar to the GPU-based approach. However, it differs from a GPU-based accelerator in that the dedicated hardware-based execution unit is not limited to matrix operations. FPGA designers can develop application-specific execution units. Here a hard processor core, e.g., the ARM processor, is a built-in processor and is implemented on the FPGA in advance. The hard processor core uses a prescribed instruction set, and the I/O bandwidth of the processor is fixed. On the other hand, a soft processor core is implemented using LB resources. Thus, engineers can design application-specific instruction sets and I/O circuits. Note that the type (C) accelerator does not have a CU, and only logical resources are used for execution units. In addition, the type (C) accelerator does not instantiate a processor core. The FPGA only receives input data required for


Software program (Instruction)
Multi-core processor
CPU (core)
CPU model (A)
RAM
Data
Control unit
Arithmetic and logical unit (ALU)
Register
Floating point unit (FPU)
Software program (Instruction)
FPGA
FPGA model (B)
RAM
Data
Control unit
Execution unit (Application-specific)
FPGA
FPGA model (C)
RAM
Data
Execution unit (Application-specific)
Figure 2.2: CPU (A) and FPGA (B, C) models. CPU (A) and FPGA (B) have CUs that interpret software programs. FPGA (C) only has an application-specific execution unit.

computations. Note that the execution unit tends to be complicated because it includes control circuits; thus, development costs increase. However, overall performance and efficiency tend to improve compared to the type (B) accelerator. Note that, for both types of accelerators, it is important to identify and accelerate costly computations that influence performance.

To date, many FPGA-based accelerators have been studied. For example, the Convey HC-1 [20], shown in Figure 2.3, is a cluster system that uses 14 Xilinx Virtex-5 FPGAs. This system targets a wide range of applications, such as oil, gas, finance, and scientific computations. The unique feature of this system is the cache-coherent shared virtual memory that can be accessed by both the host processor and the FPGAs. Note that the FPGAs interact with a large amount of data in this shared memory. The vendor claims that this system can achieve high performance for applications that require large memory bandwidth. However, finding applications in which the host and FPGAs share a large amount of data and FPGA-based acceleration works effectively was difficult. In Figure

2.3, four FPGAs (far right) accelerate computation. The other FPGAs function as interfaces to the CPU and memory. The utilization efficiency of the FPGA is low because the ratio of the circuit size occupied by the execution unit is low. Note that this system is a type (C) accelerator (Figure 2.2).

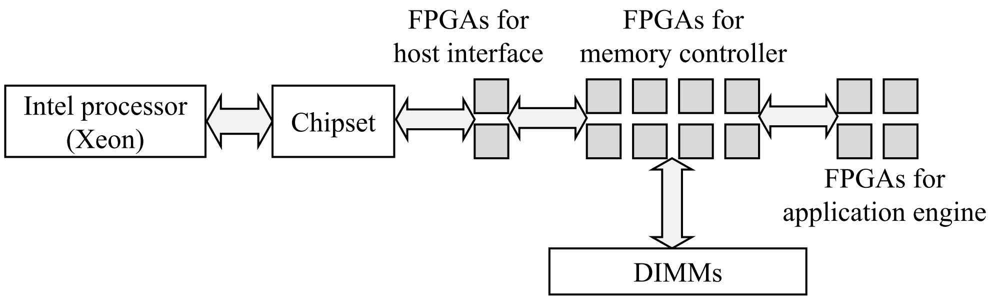


Figure 2.3: Convey HC-1 system diagram

Pico Computing's SC5 [59] is a cluster system that uses 48 Xilinx Kintex-7 FPGAs (Figure 2.4). The system aims code-breaking and can break a code encrypted using the 56-bit Data Encryption Standard (DES) in 20 hours and the cipher used in Wi-Fi Protected Access (WPA) in 50 seconds. Code-breaking is suitable for FPGA-based acceleration because this application employs ciphertext-block-level parallelism. In addition, in these applications, a very small amount of data is transferred from/to the FPGA devices. A small amount of data is only transferred at the beginning and end of the calculation, and no data are transferred during calculation. In other words, this system cannot achieve high performance in applications that do not have simple data parallelism and include heavy memory access. Note that this system is also a type (C) accelerator (Figure 2.2).

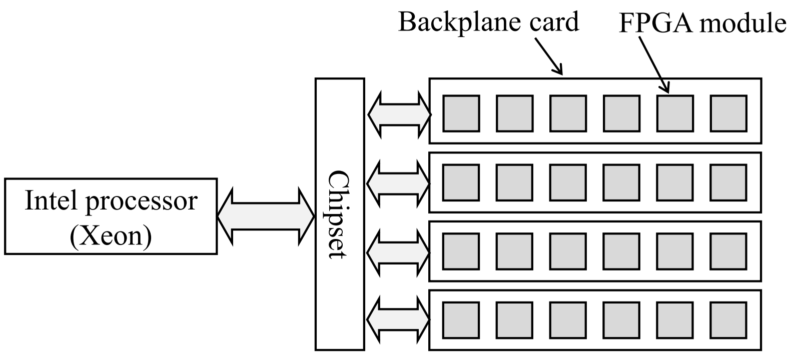


Figure 2.4: System diagram of Pico Computing's SC5

Baidu's XPU [96], which is a type (B) accelerator (Figure 2.2), uses a single Xilinx Ultrascale VU 9P FPGA (Figure 2.5). An accelerator implemented using the XPU platform is referred to as a Software Defined Accelerator (SDA). Here, the SDA includes 256 small processing cores and customized application-specific logic circuits. The processor developed by Baidu is a software processor core. Many processor cores are arranged on the FPGA such that the I/O bandwidth of processors does not become a performance bottleneck. In five benchmark applications, i.e., three simple arithmetic computations and two computer vision computations, the performance results of this system were comparable to or worse than that of the Intel Xeon E5 processor. One reason for the low performance is that the FPGA memory bandwidth was insufficient. Another reason is that the internal clock frequency of the FPGA is 600 MHz, which is low as the clock frequency of the CU.

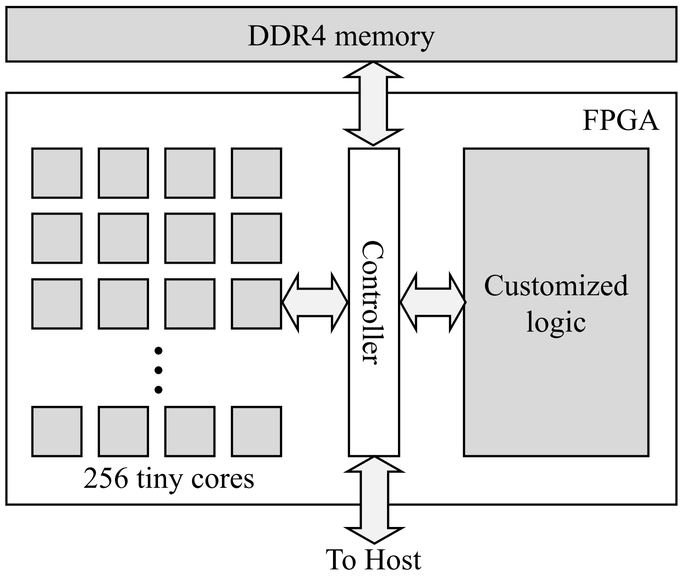


Figure 2.5: Baidu XPU system diagram

The Microsoft Catapult [102] is a cluster system that uses 48 Intel Stratix V D5 FPGAs. Figure 2.6 shows a system diagram of the Microsoft Catapult. The entire system occupies the space of a single server rack, and this system was originally developed for Internet Search Engine query processing (i.e., Microsoft Bing); however, it can also be used for other applications, including machine learning and scientific calculations. In this system, only two FPGA chips can be mounted per single rack unit. Compared to the physical size of this system, the number of FPGA chips is small; thus, its integration density is low.

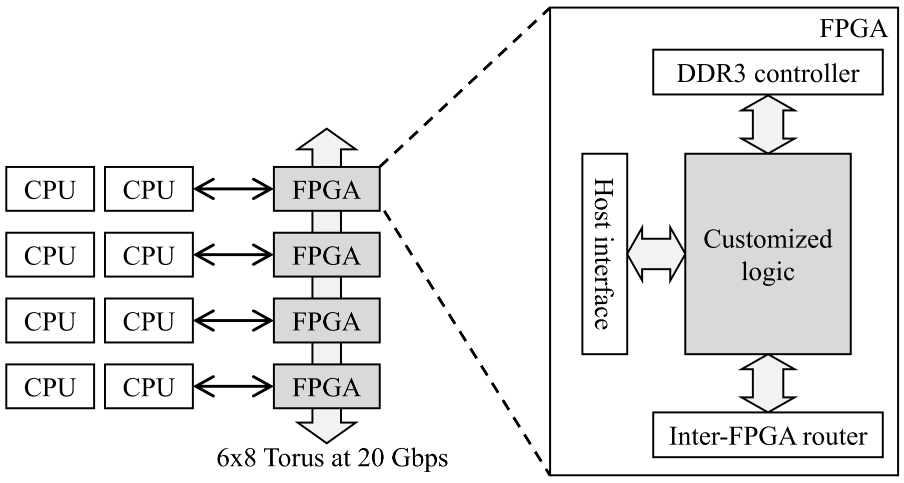


Figure 2.6: Microsoft Catapult system diagram

2.3.3 Design Guidelines

Until now, guidelines for designing hardware including FPGAs have been discussed in many publications. The circuit design guidelines for hardware engineers described in [89, 131, 135] are basic and can be automated using modern logical synthesis tools. Design guidelines described in [2] only include entry-level advice relative to logic delay reduction. Other studies [46, 130] describe how to write a recommended HDL code; however, they did not consider hardware algorithm design. The design guidelines described in another study [111] focus on coding rules for efficient development. Other design guidelines described in [140, 72, 133] target high-level languages for behavioral synthesis; thus, applying these guidelines to the HDL-based approach is difficult. A methodology described in [79] is specialized in Network on Chip, and does not target the circuit design of algorithm of other fields. Other studies [86, 12] describe methodologies specialized in the design for dynamic reconfiguration. Methodologies described in [29, 91] target the design for small-scale control circuits and industrial control systems, and they do not take into consideration performance. [126] is designed to improve tolerance to differential power analysis, and [17] focuses on design methodology for enhancing error tolerance. A methodology described in [136] is aimed at reducing power consumption, but our study is aimed at improving performance. As just described, design methodologies for high-performance hardware logic have not been established.

2.4 Computational Parallelisms

Computations include those that can and cannot be executed in parallel. Parallelizable computations are classified into three types of parallelism relative to how calculations are executed in parallel.

The first type of parallelism is data parallelism, where each element in a structured data, such as array data, is processed in the same manner. Generally, processing units arranged in parallel (corresponding to *cores* or *threads*) employ the same mechanism and receive different data. We take matrix multiplication as an example of computation with data parallelism. Matrix multiplication includes calculations of a scalar product of a row component and a column component of two matrices. The scalar product calculations are parallelizable, and the calculations for each matrix element are the same.

The second type of parallelism is task parallelism, where multiple processes with no dependency relationships are executed in parallel. Note that, in this case, the data inputs to the processing units arranged in parallel are not always the same. In addition, the internal logic circuits in the processing unit are not always the same. We take the Monte Carlo method as an example of computations with task parallelism. In the Monte Carlo method, an approximated value is computed by repeating calculations with random numbers. Here, each input datum is generated by a random number generator and has a different content. Since the calculations have no co-dependencies, they can be parallelized.

The third type of parallelism is pipeline parallelism. Pipeline parallelism means that a process can be parallelized by dividing a large process sequence into multiple steps. Here, a divided state is called a *pipeline stage*, and the stages are connected via memory called *pipeline registers*. The intermediate computational results in each state are stored in subsequently connected pipeline registers, and the next stage loads and uses data stored in the registers. In pipelined logic circuits, when a given calculation is executed in a stage, the next calculation can be executed in the preceding stage. In non-pipelined logic circuits, the next calculation cannot begin unless the previous calculation has completed. As a result, pipelining increases the degree of parallelism and the utilization efficiency of the logic unit.

Figure 2.7 shows an example timing chart of instruction processing of general processors. Here, timing charts (A) and (B) are for non-pipelined and pipelined processors, respectively. The processing sequence is divided into five stages. Generally, pipelining increases logic delay summation of all stages slightly because of pipeline register insertions. On the other hand, due to pipelining, the logic delay of a single clock cycle becomes approximately one-fifth times and we can increase the clock frequency at most about five times by pipelining. Therefore, the number of calculations per unit time (*computational throughput*) becomes at most five times. The maximum clock frequency is determined by the logic delay time of the longest path (i.e., the *critical path*). The clock frequency is the greatest when a process is divided equally in the delay time. More unbalanced the logic delay lengths of five stages are, the smaller the clock frequency will be. Note that pipeline registers should be inserted at appropriate positions to maximize pipeline throughput.

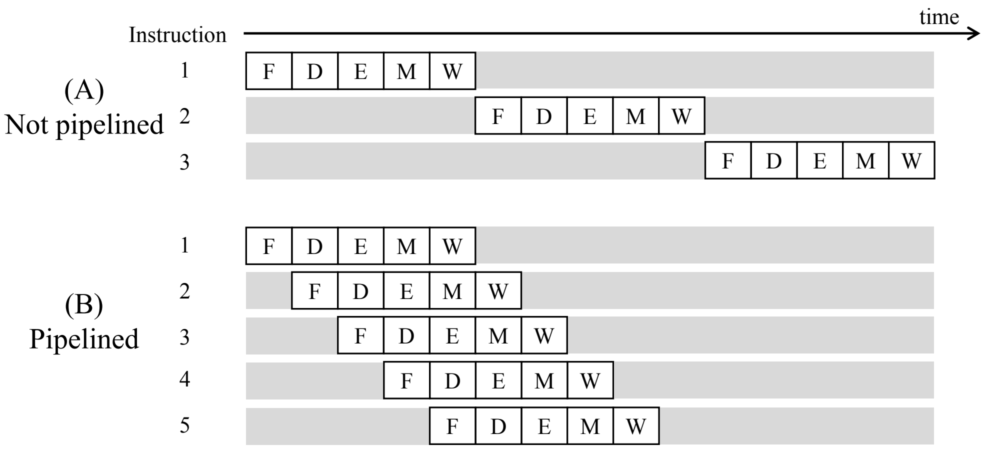


Figure 2.7: Timing chart of general processors: (A) non-pipelined and (B) pipelined architecture

These three parallelism mechanisms have been discussed individually; however, domain-specific operations should be parallelized simultaneously at multiple levels to design an efficient FPGA-based accelerator.

2.5 Memory Hierarchy

In various architecture, the major cause of performance limitation is a lack of memory bandwidth, and memory bandwidth bottlenecks prevent acceleration. Note that the memory hierarchy determines the upper bound of an architecture's performance.

There are four types of memory associated with a CPU, i.e., registers, caches, main memory, and storage devices. Registers and caches are on-chip memory devices inside the CPU, and main memory uses external volatile memory devices mounted on the motherboard. Hard disk drives and solid-state drives (SSDs) are examples of non-volatile external storage devices that are also connected to the motherboard. Note that registers and caches have a small capacity but offer low access latency. On the other hand, the access latency of external storages is large; however, they provide a large capacity. When software engineers write code that runs on the CPU, they cannot directly control which data are stored in cache, i.e., cache replacement policies are predetermined in the CPU design phase. Note that data layouts are not optimized for individual applications in the CPU approach.

The memory environment of a GPU is more complicated than that of a CPU. A GPU board has on-chip memory blocks (i.e., registers and shared memory), and off-chip memory devices (i.e., local memory, constant memory, global memory, and texture memory). Note that software programmers can control the data layout in the CUDA framework. In other words, the computing performance of the GPU approach strongly depends on the programmers' data layout for this memory hierarchy.

In addition, with FPGAs, the data layout affects performance significantly, and there are multiple types of memory blocks and devices for FPGAs. Early FPGA devices comprised only LUTs and Flip-Flops. Note that Flip-Flops are also called registers. There was no choice but to store data in registers or external memory devices in early FPGAs. In addition, there was a large gap relative to capacity and access latency between a register and an external memory, and this gap caused a performance bottleneck in FPGA-based accelerators. Built-in memory blocks are installed on FPGAs to resolve this gap problem in the FPGA memory hierarchy. Current FPGA devices provide two types of built-in memory blocks, i.e., Block RAM (BRAM) and Distributed RAM. Block RAM is a hard-macro, and the dedicated hardware logic for Block RAM is prearranged on the FPGA device. A truth table of combinational circuits realized by the LUT is configured into Synchronous RAM (SRAM) in the LUT. An FPGA designer can use this small RAM as a general memory block, and this scheme is called *distributed RAM*. Distributed RAM is often used as Read-only Memory for a table of fixed coefficients in mathematical calculations. Note that the available Block RAM resources increase with increased resources on FPGAs. We can reduce the logical block consumption using Block RAM rather than distributed RAM. As long as the wiring delay is less than the clock period, we can use as much Block RAM as desired. External memory, such as SRAM, Dynamic RAM (DRAM), and Reduced Latency DRAM (RLDRAM), are directly connected to the FPGA, and the I/O pins of the FPGA can be driven with appropriate electrical characteristics for each memory device. When we use external storage, such as an SSD, we implement a storage controller on the FPGA. Many intellectual properties (IPs) for popular interfaces and controllers, such as DRAM, have already been provided by FPGA vendors and third-party companies. Memory blocks and devices have various capacities, e.g., the capacity of off-chip memory devices is approximately 10 to 100 gigabits. The capacity of registers on an FPGA is limited to approximately 100 kilobits because of feasible logic and wiring arrangements. With distributed RAM, the limitation is approximately 1 to 10 megabits, and with Block RAM, the limitation is approximately 10 to 100 megabits. For each application, FPGA designers must consider effective memory partitioning (how to use memory) and data layout (which data are stored in which memory) solutions.

Chapter 3

Performance Improvement of Parallel TCP Streams on Long-distance Fat-pipe Networks

3.1 Introduction

With the rapid progress of network technology such as optical fiber and network switches, high speed networks are spreading all over the world. Difficulties about TCP communication on LFNs are well known. But combination of various improvement such as congestion control algorithms, pacing technologies and various techniques for reducing CPU loads almost perfectly solved these problems of single TCP stream communication on 10Gbps LFN. On the other hand, it is still difficult to get good performance while using parallel TCP streams. Parallel TCP streams are commonly used in the field of high performance applications especially for data transfer between clusters. Each stream tends to behave differently, even if all settings and environments are same, and some streams suffer self-made congestion.

This chapter shows how to make parallel streams balanced with each other, which results in the improvement of total system performance. First, we show the problem. Then we introduce the hardware MaSTER-1, a programmable testbed with five FPGAs and five 10GbE port to handle parallel streams. We implement merging stream harmonizer (MSH) on MaSTER-1, which merges input streams into a sequence of packet by a specified policy with packet level granularity. By using MSH, parallel TCP streams from end-node hosts of the cluster can be controlled without modification of the end-nodes. We evaluate MSH using four parallel TCP streams on LFN, both pseudo-LFN environment with network emulator and Japan-US real LFN. We can utilize 99.6% of 10Gbps LAN PHY bandwidth on pseudo LFNs and 87.0% of 9.2Gbps WAN PHY bandwidth on real LFNs.

On LFN, it is often observed that some sections have smaller bandwidth than other sections, which we call "*path-bottleneck*". On LFN, especially on the inter-continent LFN, this path-bottleneck often occupies very long section, such as over-sea network, which often has less bandwidth because under-sea fiber construction is more difficult. The path-bottleneck is critical, even when the difference of bandwidth is rather small, such as WAN-PHY LAN-PHY difference which is less than 10 %, and packet losses seem to occur mainly on the switches on the entrance of the path-bottleneck, whose input bandwidth is larger than that of output. We call such switch *"path-bottleneck switch"*. To avoid packet losses on the path-bottleneck switch, we applied pacing at the sender host to slowdown the throughput less than path-bottleneck [142]. On the other hand, it is also observed that if the path is not so long, packet loss rarely occurs even in the situation that we cannot utilize the flow control mechanism of the switch to temporally stop the stream and control bandwidth.

This study focuses on the effect of the buffer of the path-bottleneck switch. Our goal is to find the method to avoid unnecessary packet losses caused by insufficient amount of buffers at the path-bottleneck switch on the network path. For this purpose, this study clarifies the relation between RTT, maximum bandwidth of the network, necessary buffer size of the path-bottleneck switch and TCP congestion control algorithm. Previously, the necessary size of the buffer at path-bottleneck switch is assumed to be proportional to the size of inflight data. However, our measured data shows different relation between them, e.g. low bandwidth TCP stream requires larger buffer size. To analyze the effect of the buffer on the path bottleneck switch, we use the MaSTER-1. And, we make MaSTER-1 take a role of the path-bottleneck switch. We analyze the relationship between buffer size and round-trip time (RTT), with several TCP congestion control algorithms. In this chapter, we briefly introduce TCP congestion control. Then, using this MaSTER-1, we show experimental result of the effect of the buffer of the path-bottleneck switch on LFN, analyzing the relationship between RTT and buffer size, with several TCP congestion control algorithms. We show that if path-bottleneck switch has huge size of buffer, we can avoid packet losses to some extent.

In Section 3.2, we overview the feature of LFN and parallel TCP streams and their problems. In Section 3.3, we introduce our network testbed MaSTER-1. In Section 3.4, we explain the MSH. In Section 3.5, we show experiments and show the results of the parallel TCP communications. In Section 3.6, we describe the relationship between TCP window size and RTT, and show the problem. In Section 3.7, we show experimental result of path-bottleneck switch. In Section 3.8, we discuss the results of experiments. In Section 3.9, we discuss related work. In Section 3.10, we conclude and describe our future work.

3.2 Parallel TCP Streams on LFN

3.2.1 Bursty Behavior of TCP

TCP/IP is a standard protocol for reliable data transfer. For reliability, TCP uses ACK; a sender keeps data for re-transmission until ACK returns. Data, which is sent but not ACKed, is called "*in-flight*" data, and its maximum size is called "*window size*". Data transfer rate of TCP is roughly window size/RTT. But, microscopically, streams behave differently. Using a packet logger, traffic analysis precise enhancement engine (TAPEE) [143], we observe streams with the packet level granularity. Figure 3.1 shows an example of single stream TCP data transfer on pseudo-LFN of 500ms RTT. First, a sender host transmits data to the network with the speed of network interface, that is 10Gbps, until the size of cwnd data is transferred, and then it almost stops transmission till next RTT starts. This bursty behavior causes unnecessary packet losses that result in bad performance. Pacing of the stream at a sender host is effective for a single stream [142, 67].

3.2.2 Parallel TCP Streams

When we use parallel streams, the situation becomes more complicated and difficult. Figure 3.2 and 3.3 shows the throughputs of two streams and four streams on 500ms pseudo LFN, respectively. In Figure 3.2, stream1 gets larger throughput from the first beginning, and after packet losses, stream2 fails to gain through-

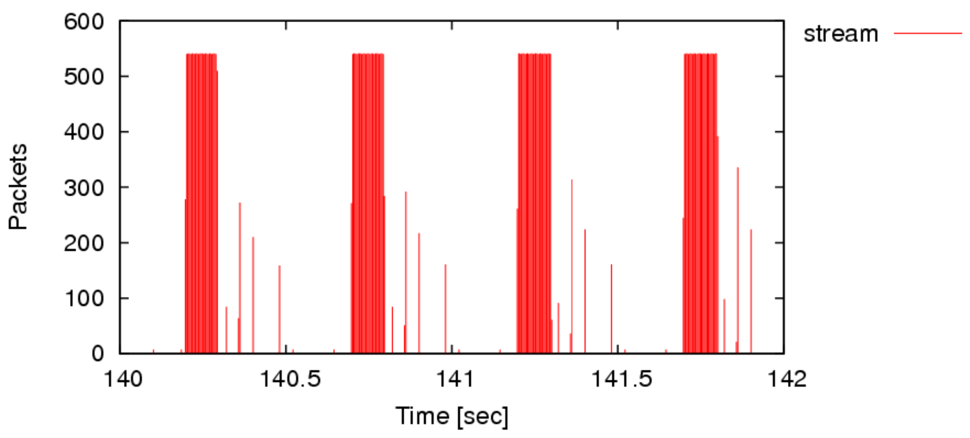


Figure 3.1: Packet bursts in large RTT (500ms) network (pkts/4ms)

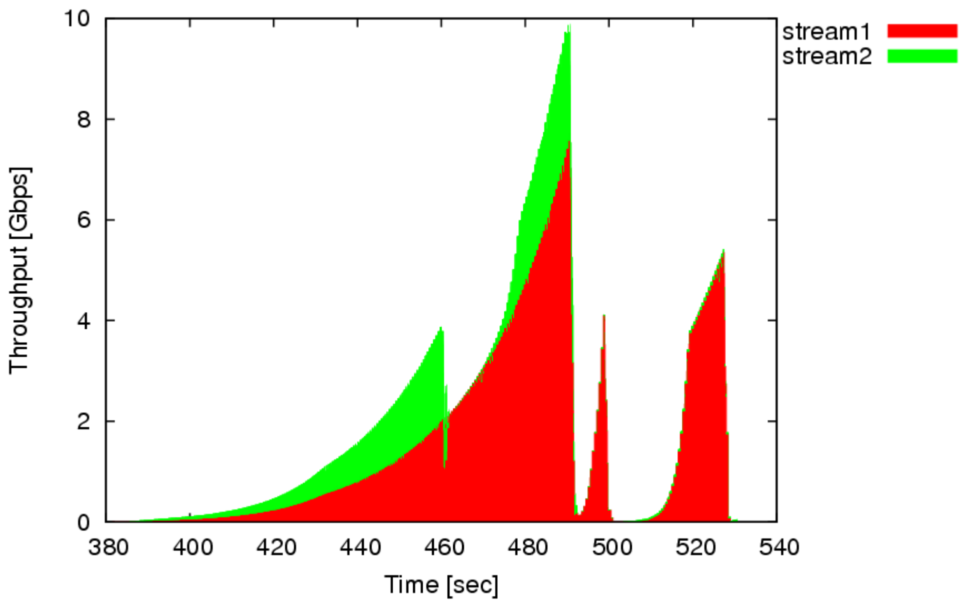


Figure 3.2: Throughputs of 2 streams on pseudo LFNs (RTT 500ms)

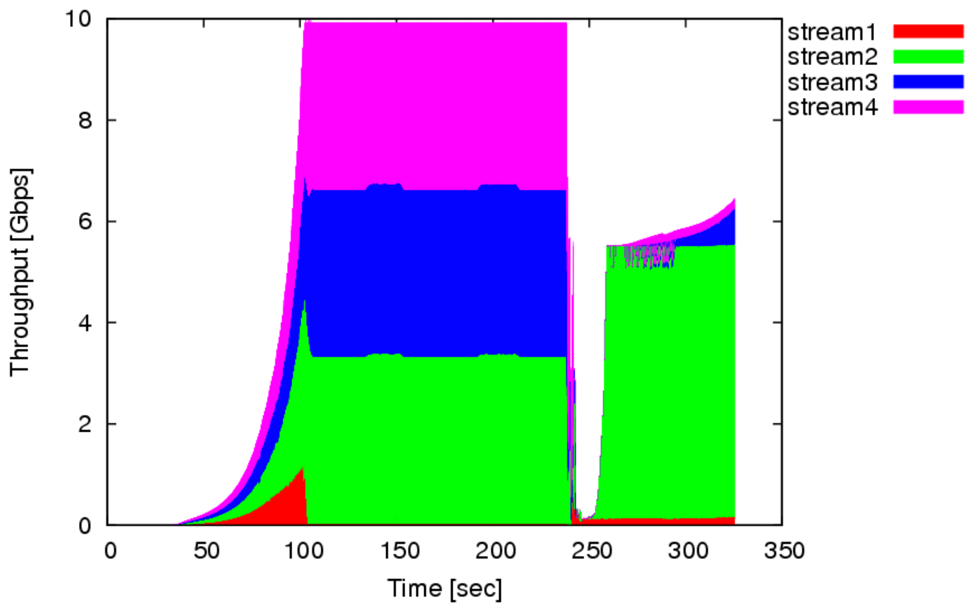


Figure 3.3: Throughputs of 4 streams on pseudo LFNs (RTT 500ms)

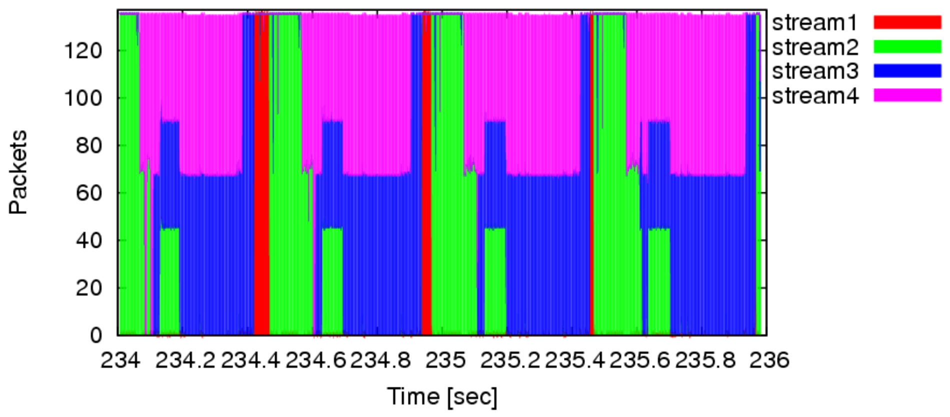


Figure 3.4: Pkts/msec of 4 streams on pseudo LFN, magnified (RTT 500ms)

put. In Figure 3.3, stream1 fails to get throughput, and after packet losses occur, stream2 uses about half of bandwidth, and other three streams fail. Figure 3.4 shows the period from 234 sec to 236 sec of Figure 3.3. We can observe that similar packet patterns of 4 streams repeat in every RTT. Total throughput is almost 10Gbps and packet losses of stream1 are just going to occur. Thus, parallel streams have two more additional difficulties compared to single stream. (1) Performance of each stream may differ to each other although the conditions of these streams are exactly same. Even though network bandwidth is fully utilized, this unbalance may decrease system total performance because the worst stream tends to become a bottleneck of performance in many applications. (2) Bursts of each stream are piled, as the result, self-made congestion may occur more easily, and even if the network is not congested, packet losses may easily occur.

We used Chelsio S310E-SR 10GbE NIC [34] which prepares two functions of controlling data transmission rate. One is IPG control to limit maximum transmission rate of the NIC. IPG control is precise and effective, but it can be applied only for the stream faster than 9Gbps. The other is "`pktsched`", which is not so precise but can be applied to any transmission rate. Using these functions, the parallel TCP streams can be controlled individually.

Figure 3.5 and Figure 3.6 show the throughput of 4 parallel TCP streams communication with sender's NIC packet pacing in the network that has 500ms RTT. Figure 3.5 is the case that NICs pace the throughput to 1.0Gbps per host. Figure 3.6 is the case that NICs pace the throughput to 2.0Gbps per host. When we pace each stream to 1.0Gbps, data transfer goes without packet losses, but, when we set to 1.5Gbps and 2.0Gbps, packet losses occur. Both in Figure 3.5 and Figure 3.6, throughputs are balanced and equalized, but, because of the periodical microscopic burst, when we set the throughput of each stream to more than 1.5Gbps, packet losses occur, caused by the pile of microscopic burst. Hence, we want to control throughput in finer grain, which hardware support is necessary.

3.3 MaSTER-1

3.3.1 Overview

We try to make the parallel TCP communication stable by placing a hardware at the merge point of the streams. We have to prepare a hardware which can buffer the streams, enable us to observe the phenomena in it, and handle the multiple TCP streams quickly. For handling the one 10GbE stream, the one packet must

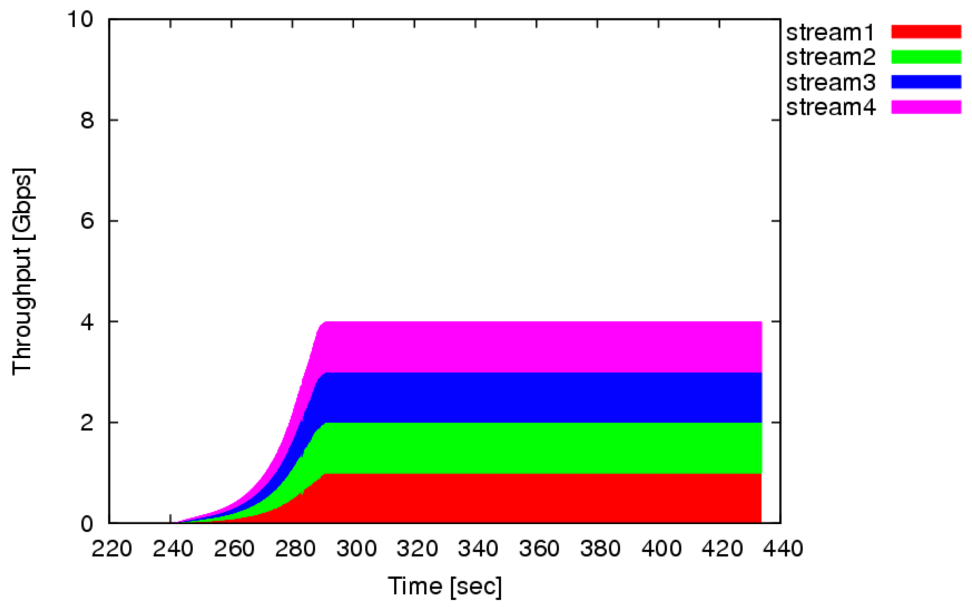


Figure 3.5: 4 streams with NIC 1.0Gbps pacing on pseudo LFNs (RTT 500ms)

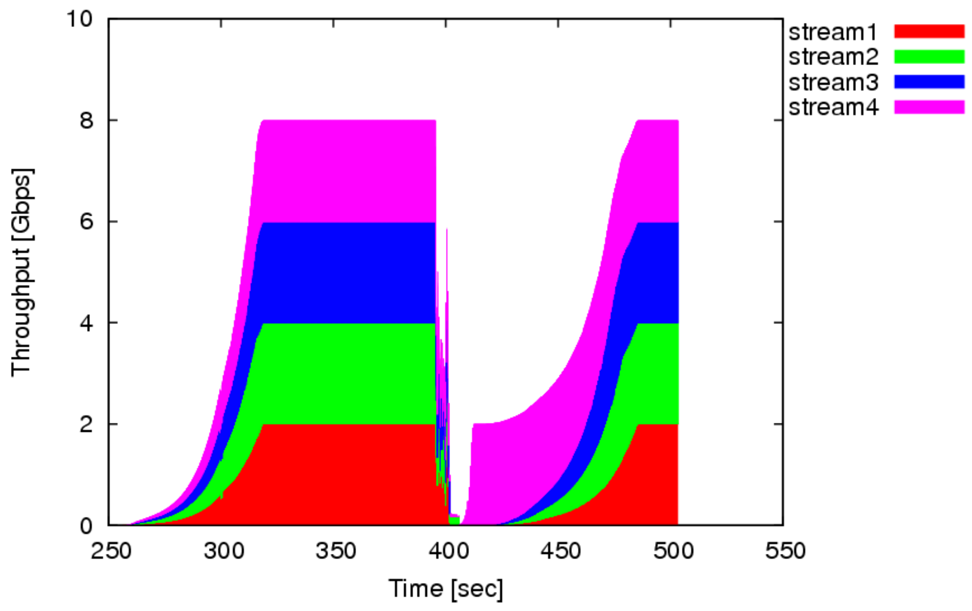


Figure 3.6: 4 streams with NIC 2.0Gbps pacing on pseudo LFNs (RTT 500ms)

be operated within 7.3usec. This cannot be achieved by software in any way. It is clear that the solution by not software but hardware is needed.

We developed MaSTER-1 for what we want to experiment with (Figure 3.7). The MaSTER-1 is a hardware testbed for wire-rate parallel TCP streams processing. We can implement our mechanism on the MaSTER-1 and experiment the parallel TCP communication. The MaSTER-1 has three features for general-purpose properties; 10GbE port, large size buffer, and programmability.

1. **Five 10GbE port**

   The MaSTER-1 has five 10GbE ports physically, and they are connected to the neighbor hosts or switches with optical fibers. Each port can receive and send the packets at a wire rate.

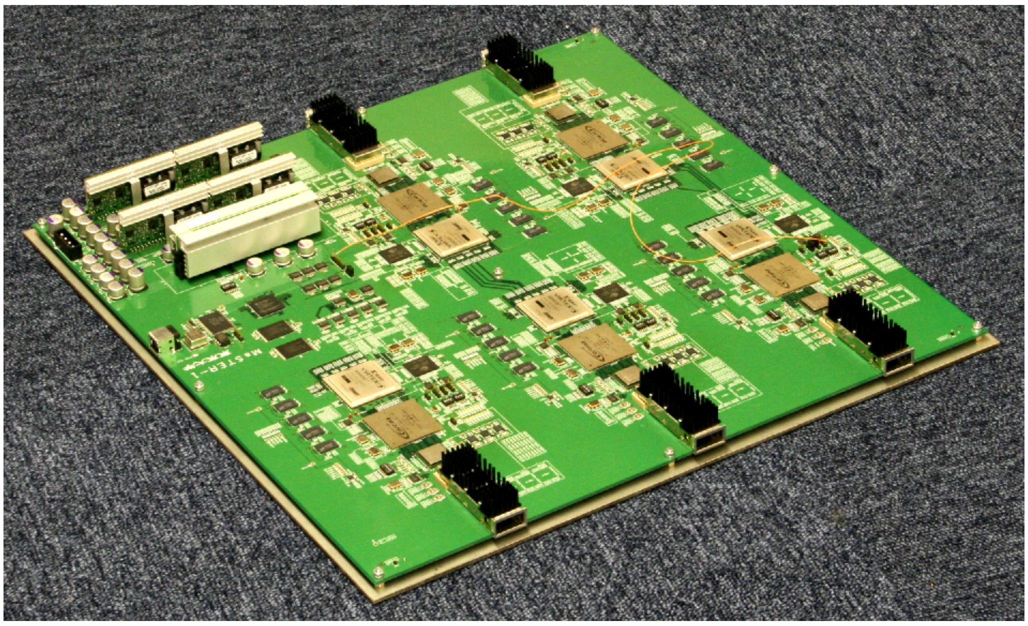


Figure 3.7: Network testbed MaSTER-1

2. **Large Buffers**

   The MaSTER-1 has Large buffers for storing received packets temporary for each port. Because this hardware will be placed in the merging point of streams, there is a possibility that it keeps large amount of data before it merges the streams.

3. **Programmability**

   The MaSTER-1 will be able to be used for various applications by reconfiguring the programmable chips. Programmable chips can be used for forwarding packets from one port to another port and the various experiments will be conducted. Those applications are described in the section of future work. And these chips are needed to operate at high frequency because they must handle the streams at packet-level and pace packets precisely. We used an FPGA which is one of the programmable chips.

On the cost front, a high-capacity programmable chip which can handle multiple streams by itself must have many logic elements and IO interfaces and it is expensive. Instead, we used programmable chips with intermediate capacity and distribute the tasks of packets handling over them. Because the ports have identical specifications, we can design a hardware and produce firmware for chips easily.

3.3.2 Hardware Design

Figure 3.8 shows a block diagram of MaSTER-1. The MaSTER-1 has five 10GBASE-SRLR ports. They are fully inter-connected with eight 3.25GHz high-speed one-way signal transmission lines. With 8b10b encoding, the aggregate two-way bandwidth is 10Gbps per pair of ports. Figure 3.9 shows the components of each port. They consist of XFP module [33], AMCC S19237 Serdes, Intel IXF18103 10 Gigabit Ethernet LAN or WAN PHY Chip, 266MHz DDR2 SDRAM, and FPGA (Xilinx Virtex-5 XC5VFX70T [141]). The interface between XFP module and S19237 is 10GHz differential high Speed CML signals, that of

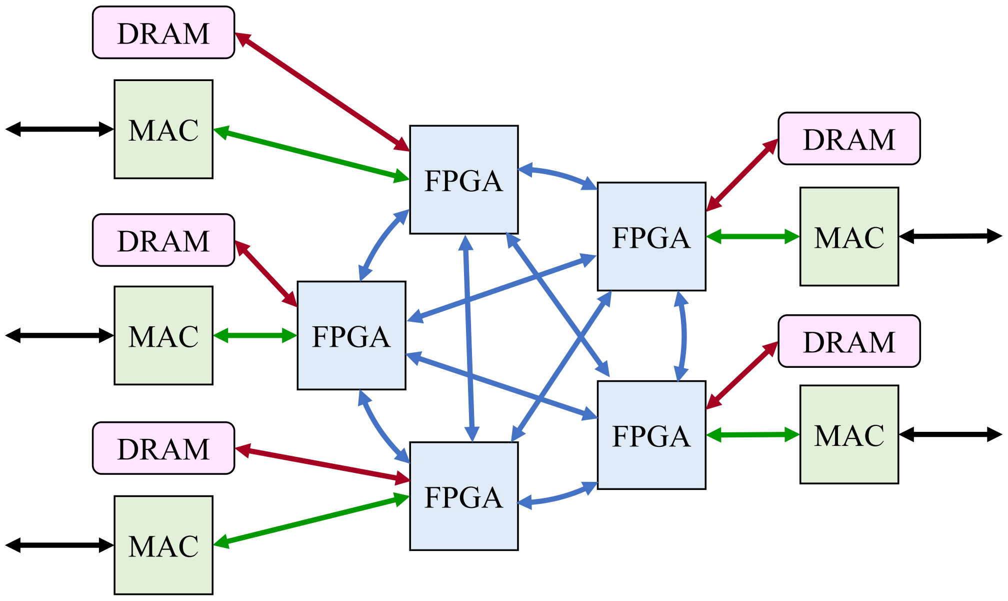


Figure 3.8: Block diagram of MaSTER-1

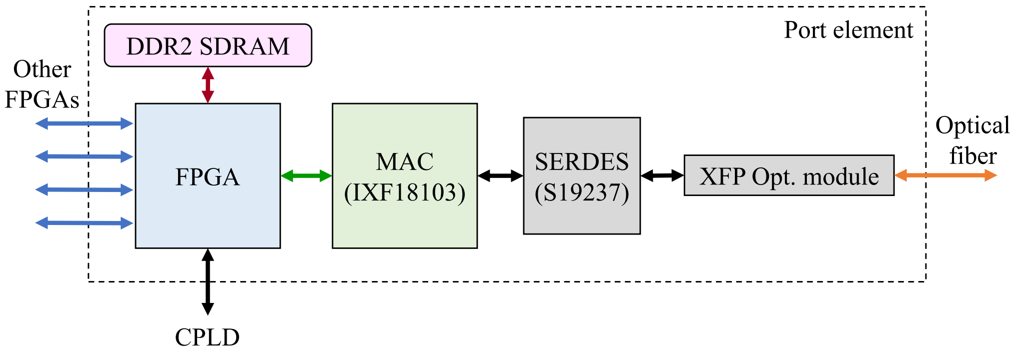


Figure 3.9: Block diagram of one port in MaSTER-1

S19237 and IXF18103 is XSBI, and that of FPGA and IXF18013 is SPI-4 Phase 2 [139]. Either connection has more than 10Gbps bandwidth and avoids becoming a bottleneck. The FPGA is connected to 537MB DDR2 SDRAM, which has enough bandwidth (34Gbps) to read and write at 10Gbps simultaneously. It is useful for implementing FIFO buffers for packet pacing, delay emulation and so on. Traffic flow in the MaSTER-1 is as follows. When a packet is received by one port, it is processed by XFP module, S19237, IXF18103, FPGA in that order. Then the packets are forwarded to a FPGA in the other ports. And the packets go to another 10GbE port in that reverse order. The packets are processed by FPGAs twice, before and after merging. This flow is indicated by the arrows in Figure 3.8 and 3.9.

The size of memories is estimated as follows. We want to process TCP streams. In the TCP protocol, the data that has been sent but not yet acknowledged is called in-flight data. Sender must not send data with a sequence number higher than the sum of highest acknowledged sequence number and the minimum of the congestion window and receiver window size [87]. This congestion window control mechanism shows that the buffer size of each port should be bigger than the in-flight data size of a host. When the throughput of a port is b bps and RTT is t_r sec, the size of in-flight data is approximately bt_r bit. If

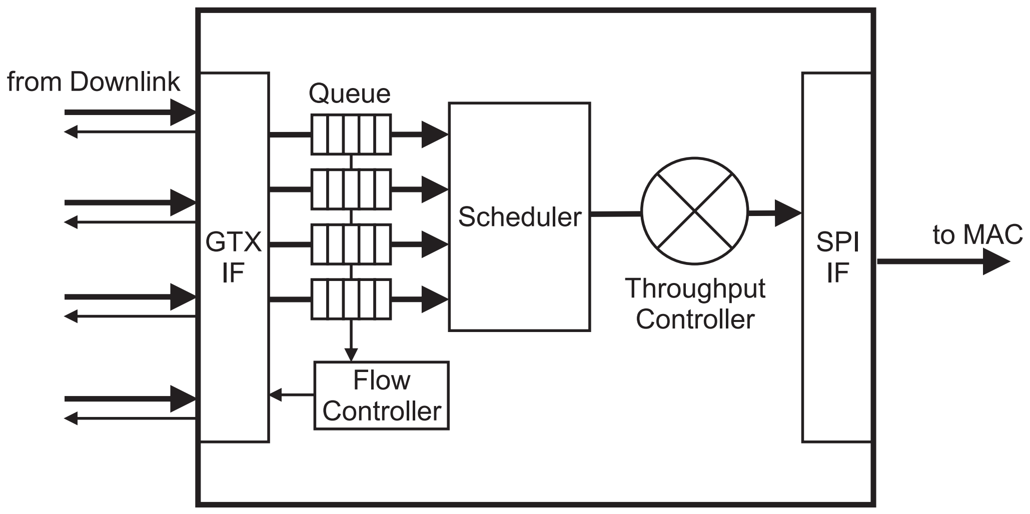


Figure 3.10: MSH uplink logic diagram in MaSTER-1

$b = 2.5G$ and $t_r = 0.5$, the size of memories is needed to be larger than 1.25×10^9 bits. The MaSTER-1 has enough capacity.

MaSTER-1 has one USB 2.0 interface connected to a CPLD (Xilinx CoolRunner-II XC2C512). The CPLD is connected to five FPGAs and several chips. User can communicate with FPGAs and other chips by control host through the CPLD. They can be done hot reset and initialized their parameters. By reading and writing registers of FPGAs, user can configure functions of the packet pacing rate and so on. These components are implemented in a board, which can be mounted in a 19-inch rack chassis.

3.4 MSH: Merging Stream Harmonizer

We want to merge parallel TCP streams fairly without packet losses, eliminate bursts and regulate the throughput of streams. We propose merging stream harmonizer (MSH) which merges multiple streams and balances among them. To realize this mechanism, we use the network testbed MaSTER-1. We can implement our mechanism on the MaSTER-1 and performance experiments on the parallel TCP communication. MSH buffers the streams which come into the MaSTER-1, the packets of multiple streams are selected by round-robin algorithm and go out from MaSTER-1 with the specified transmission rate. MSH is substituted for the switch which merges multiple streams. General Layer-2 switches have many functions that include traffic controlling (stream merger and distribution). We focus on the multiple streams merging and the switching capacity. We realize the functions of merging and scheduling on MaSTER-1.

We implement the packet forwarding circuit of MSH on FPGAs in the MaSTER-1. In MSH packets are processed by FPGAs twice, uplink FPGA and downlink FPGA. Figure 3.10 and Figure 3.11 show the logic diagram of uplink port FPGA and downlink port FPGA, respectively. The parallel streams must be processed at wire rate speed from input port to output port. Traffic flow in the MSH is as follows. When packets are received by each downlink port XFP module (optical port), they are stored to the main FIFO memory through the SPI Interface (SPI IF). The packets are immediately loaded from the memory and forwarded to the uplink port FPGA through a GTX Interface as shown in Figure 3.11. The main FIFO is implemented by using the components of DDR2 SDRAMs and Xil-

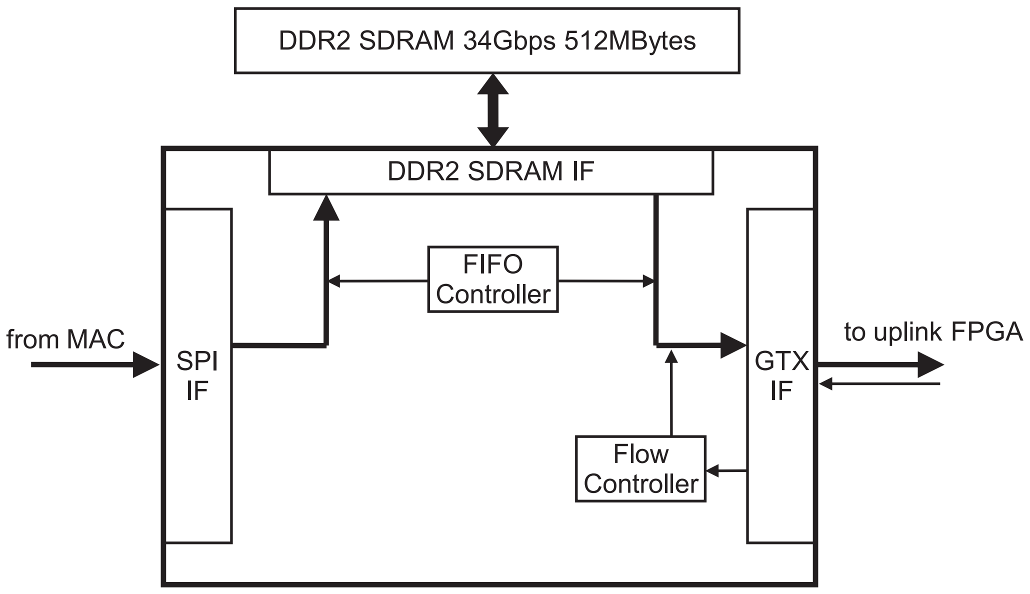


Figure 3.11: MSH downlink logic diagram in MaSTER-1

inx's IP core [97]. The GTX Interface (GTX IF) is implemented by using GTX Transceivers, which are the resource elements of the Virtex-5. Four downlink FPGAs forward their packets to their corresponding queues in uplink FPGA as shown in Figure 3.10. The uplink FPGA loads packets from these queues, merges them by "*Scheduler*" to fix the order of the packets in the scheduling policy of round-robin. Thus, *Scheduler* loads a packet from the queues in order of queue1, 2, 3, 4, 1, 2, 3,... And it proceeds pacing by "*Throughput Controller*", then, forwards them to the IXF18103 (Intel 10Gigabit Ethernet PHY Chip) through the SPI Interface towards the uplink port XFP. The "*Flow Controller*" monitors the usage of the queues, and if the queues are going to overflow, it sends the signals expeditiously to the downlink port through GTX Interface. When the flow controller in the downlink FPGA receives this pressing signal, it stops reading data from the main FIFO that consists of DDR2 SDRAM.

3.5 Performance Evaluation on Pseudo and Real LFNs

We arrange both pseudo and real LFNs. We transfer data over the network with parallel TCP streams and observe the throughput. We evaluate the effect for the throughput stability of parallel TCP streams by using our mechanism on LFNs. We compare the case when the MSH is used for merging as an intermediate node in the network and when the MSH is not used and traditional 10 Gigabit Ethernet switch is used as ever.

3.5.1 Experimental Environment

We use an Anue H-Series Network Emulator [4] to emulate the LFN. It is settled in the network path in place of real LFNs and inserts delay (up to about 800ms) to both directions. We can configure delay time independently of each direction. Real LFNs have many features, long-distance (large delay), intermediate switches, path bottlenecks and cross traffics, etc. To observe the pure effects of the network delay, we use this network emulator. We use eight servers for experiments of data transmission. Four sender hosts and four receiver hosts have identical specification of the hardware and operating system as shown in Figure

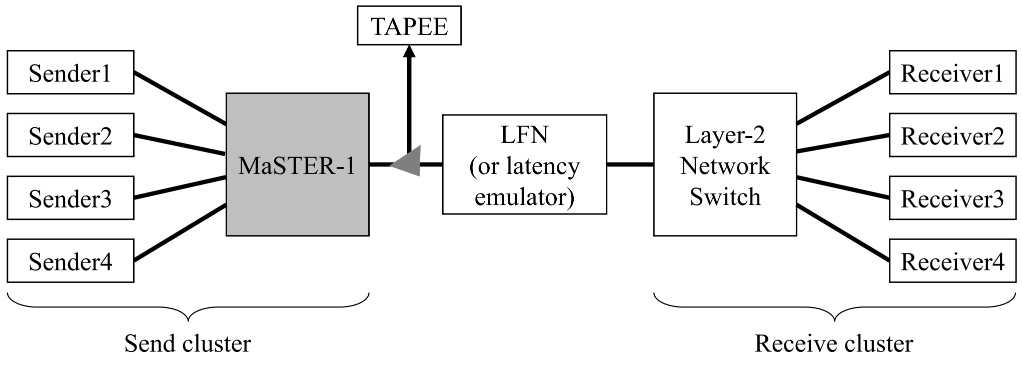


Figure 3.12: Network diagram of sender and receiver clusters

3.12. They consist of two Intel Dual-Core Xeon 5160 (3.00GHz, 4MB L2 cache) processors, Supermicro X7DB8, DDR2 SDRAM (total 4GB), Chelsio Communications S310E-SR 10 Gigabit Ethernet network interface card (PCI Express Gen1 x8) and Linux 2.6.18 Kernel. We adopt BIC-TCP as a TCP congestion control algorithm. We run Iperf 2.0.2 [61] simultaneously at sender hosts. Table 3.1 shows the parameter of Iperf.

We use TAPEE to observe the streams. TAPEE is settled at the point where we want to observe the packets. TAPEE receives the bidirectional streams which are copied by an optical tap and forwards only the headers of them with TAPEE's precise time stamp to logging hosts. Using this instrument, we can perform a precise measurement of the streams' throughput. We use Fujitsu XG800 switches [44] for two purposes. One is to compare with the MSH. We make them merge the streams and evaluate the throughput stability of them. The other is to distribute the streams from one 10GbE to four 10GbE and to merge ACK packets. The load of these tasks is lighter than that of merging DATA packets and the traditional switches have enough potential to do them.

Table 3.1: The Parameter of Iperf

Senders	-c *rec_host* -i 1 -t 600 -l 192k -w 600M -M 9150
Receivers	-s -i 1 -w 600M -l 192k

3.5.2 Experimental Results in Emulated LFN

Using Anue network emulator, we use pseudo LFN whose RTT is 500ms. A pair of sender and receiver generates one stream and total 2 or 4 streams are established. We settle MSH instead of sender edge switch, which merges the packet in the round-robin manner at MSH. Figure 3.13 and Figure 3.14 show the throughput of two streams and four streams, respectively. The streams are equalized and no packet loss occurs. After 400 sec in Figure 3.14, total throughput of four streams is 9.88Gbps, which is 99.6% of TCP payload bandwidth in LAN PHY network. Figure 3.15 and Figure 3.16 show the magnified 2 seconds of Figure 3.14; 412 to 414 sec, and 420 to 422 sec. Around 412 sec, total throughput reaches to the maximum, but, at this point, we can still see periodical pattern on the graph. But, after 6 sec, after 420 sec, the bandwidth is shared equally to four streams, and the throughput of each stream is very stable.

We observe one of streams precisely to analyze how MSH works. As shown in Figure 3.17, we put two logging tools TAPEE; we settle one at the point between

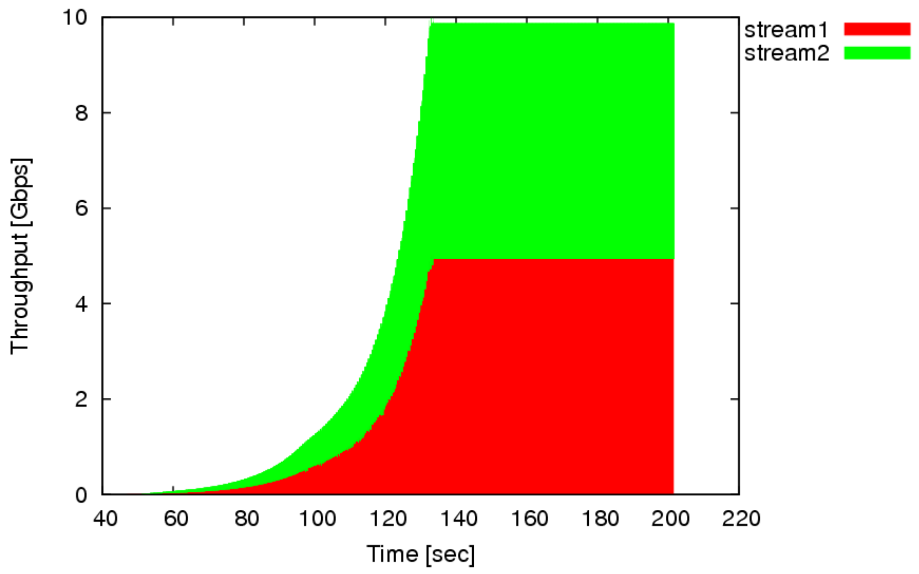


Figure 3.13: Throughputs of 2 streams on pseudo LFNs with MSH (RTT 500ms)

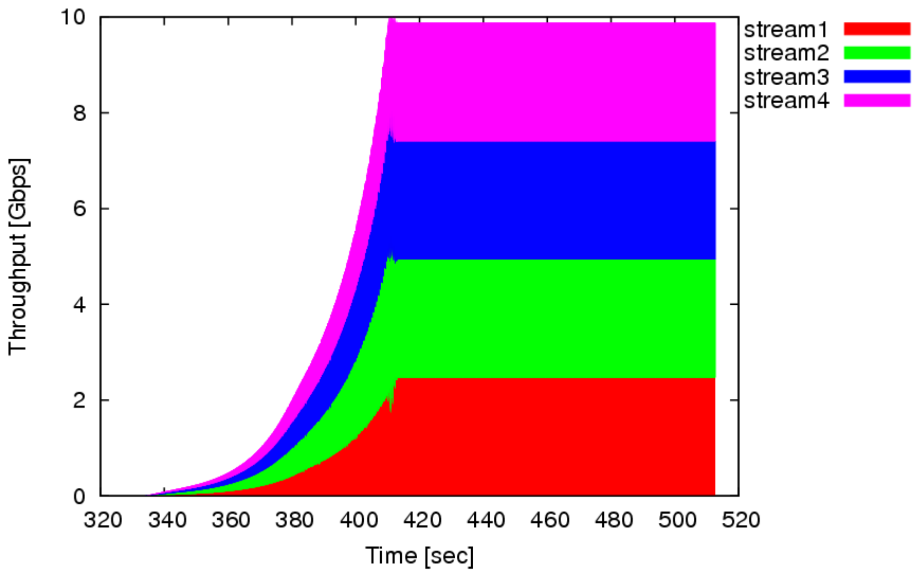


Figure 3.14: Throughputs of 4 streams on pseudo LFNs with MSH (RTT 500ms)

one port of MaSTER-1 and a sender host, and at the point between uplink port of MaSTER-1 and network emulator. We compare these two points packet loggings, just before MSH and just after MSH. Figure 3.18 and Figure 3.19 shows the change of packet behavior caused by MSH, which are drawn by two TAPEEs simultaneously. One TAPEE (Figure 3.18) is placed between a sender host and the MaSTER-1 and another (Figure 3.19) is placed between the MaSTER-1 and LFNs. In these graphs, the green fine plots are 1msec moving average of the throughput and the blue plots are 1sec moving average. Until the throughput reaches 2.47Gbps, the MSH does not pacing the stream. After the throughput reaches 2.47Gbps, the MSH paces the stream to 2.47Gbps. In this time, with the observation of 1sec moving average, the throughput of the stream which the sender NIC generates looks 2.47Gbps. However, the stream has fine bursts from

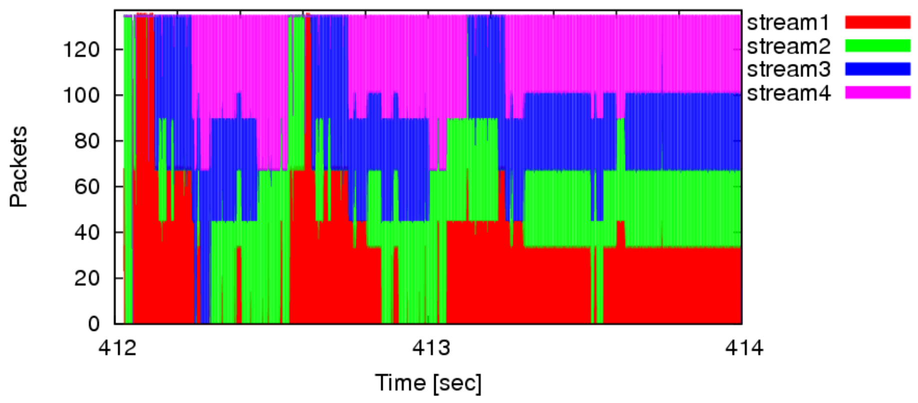


Figure 3.15: 4 streams in balancing phase on pseudo LFNs with MSH, magnified (RTT 500ms)

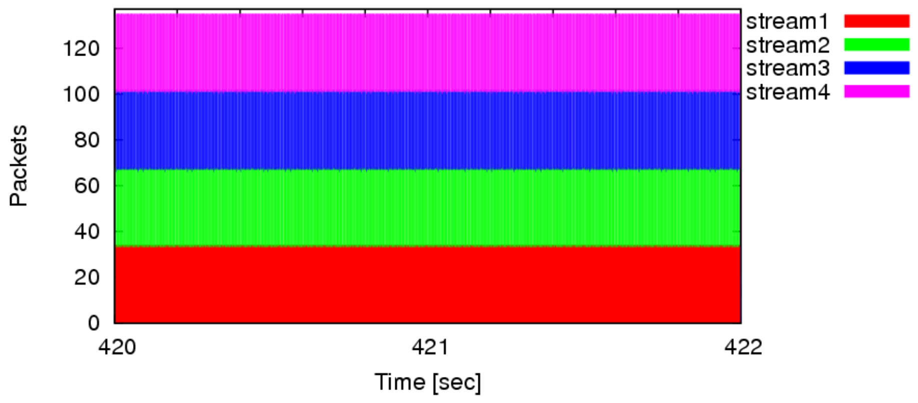


Figure 3.16: 4 streams in stable phase on pseudo LFNs with MSH, magnified (RTT 500ms)

2Gbps to 3Gbps with the observation of 1msec moving average. The stream which has passed through the MSH has no burst and both of 1msec and 1sec moving average of throughput are exactly 2.47Gbps. The total throughput averages for 10 minutes are shown in Figure 3.20. These indicate that the throughput of the parallel TCP streams is improved by the MSH.

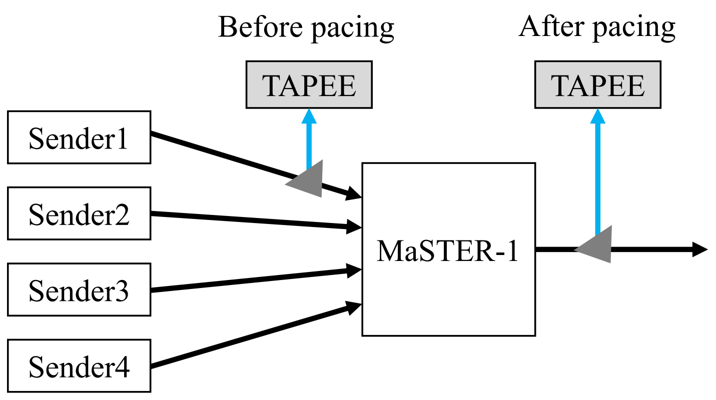


Figure 3.17: Observing points of TAPEE

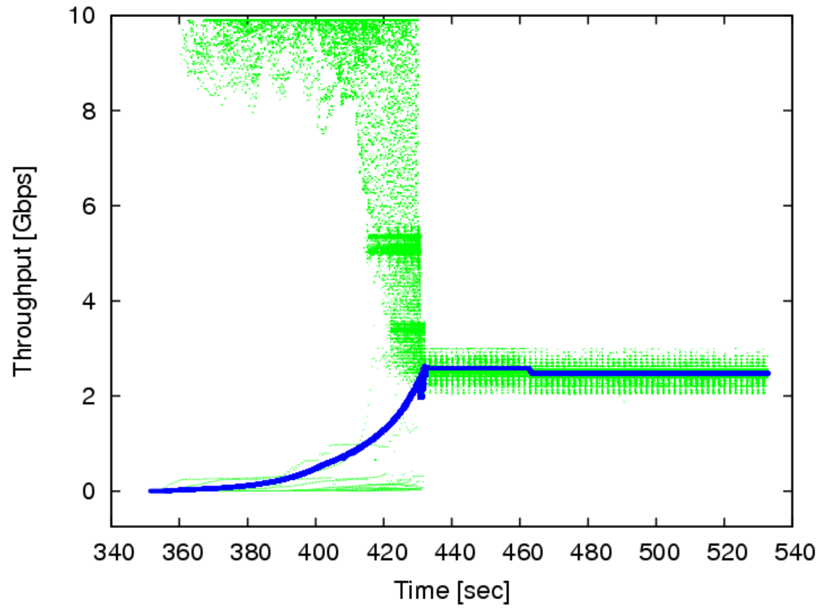


Figure 3.18: 1 of 4 streams on pseudo LFNs with MSH (RTT 500ms) BEFORE pacing

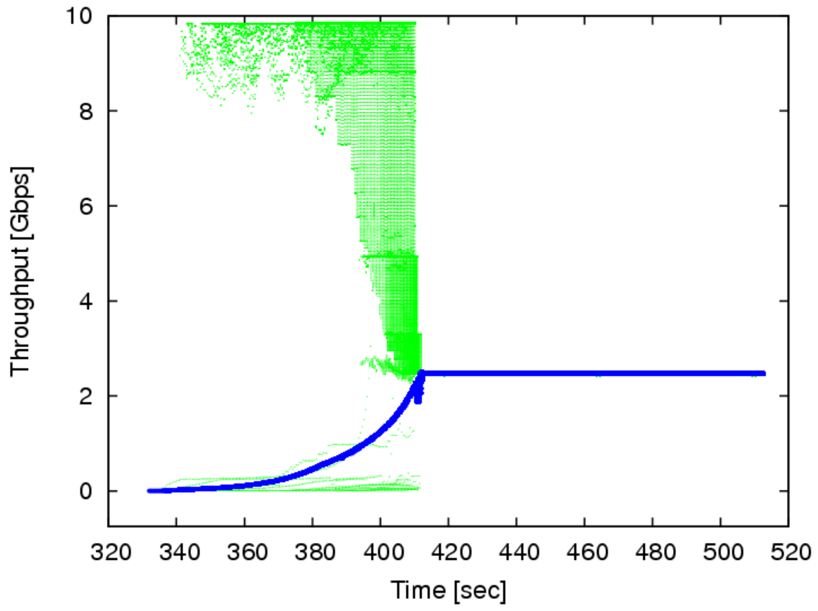


Figure 3.19: 1 of 4 streams on pseudo LFNs with MSH (RTT 500ms) AFTER pacing

3.5.3 Experimental Results in Real LFN

The real LFN is shown in Figure 3.21. These paths consist of some WAN PHY switches and IEEAF intercontinental line over Pacific Ocean. Packets are sent from the send cluster in Tokyo, pass through several switches, turn around at the switch in Seattle, go back on same paths and arrive to the receive cluster in Tokyo. These paths have approximately 96ms delay and its RTT is 192ms. Figure 3.22 shows the throughputs of four TCP streams with MSH on real LFNs. With 8.0Gbps pacing of MSH, the four streams reach 2.0Gbps each. Because these 4 connections start at different times, stream2,4 reach 2.0Gbps earlier than

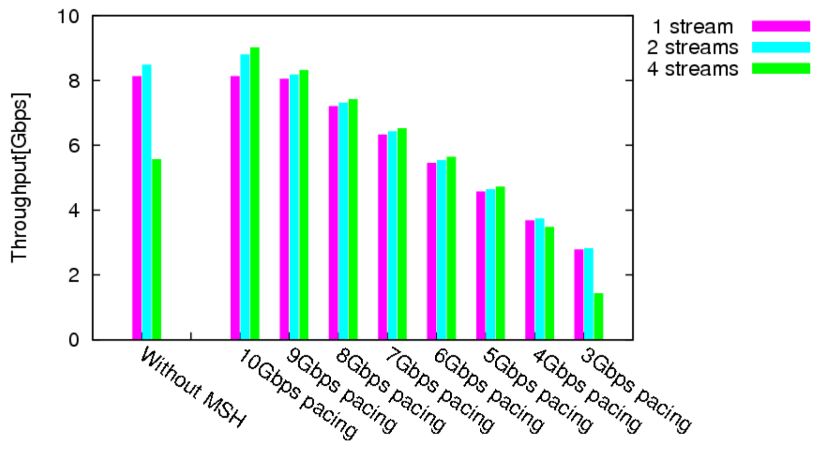


Figure 3.20: The total throughput averages on pseudo LFNs (RTT 500ms)

stream1,3. MSH presses the throughput of stresm2,4 (over 2.0Gbps) to protect bandwidth for stream1,3 and balance all streams. In this process, no packet loss occurs. But surprisingly when the MaSTER-1 limits the packets to 4.00Gbps at its uplink port, the throughput of four streams become unstable and many packet losses occur.

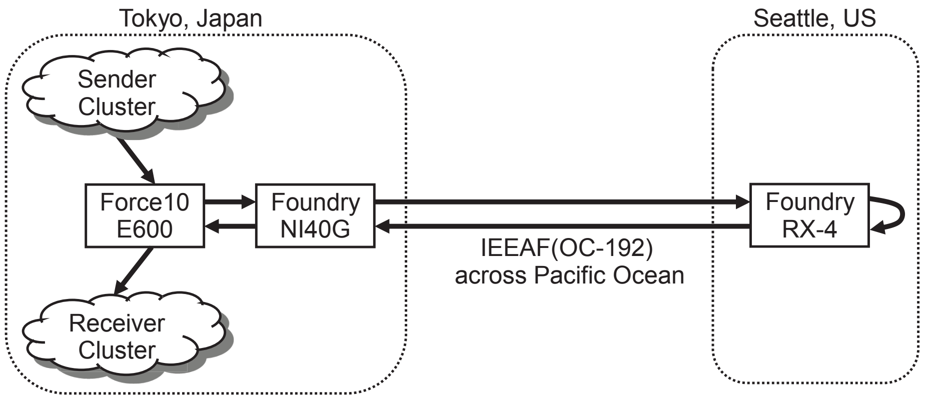


Figure 3.21: Real LFNs diagram

3.6 Relationship between TCP Window Size and RTT

Long distance data transfer needs large window size, since transfer rate is inverse proportional to RTT. In addition, speed of growth of the window size is proportional to RTT, since it takes RTT time till ACK returns. And when packet loss occurs, it takes longer time for the sender to know the occurrence of the packet loss, and to recover the throughput. Many congestion control algorithms have been proposed to improve TCP performance on LFN; such as Fast TCP, Scalable TCP, BIC-TCP, and CUBIC. Commonly, the maximum window size is set in advance, and size of the window grows until either it reaches this maximum size or a packet loss occurs. There is correlation between the size of RTT and the communication distance of the real network. When we proceed experiments for parallel TCP streams on real LFN, we face strange phenomena.

Figure 3.23 and Figure 3.22 show the experiment of data transfer from four senders to four receivers between Tokyo – Seattle – Tokyo round-trip path whose RTT is 193ms. The original aim of the experiment is to balance four TCP

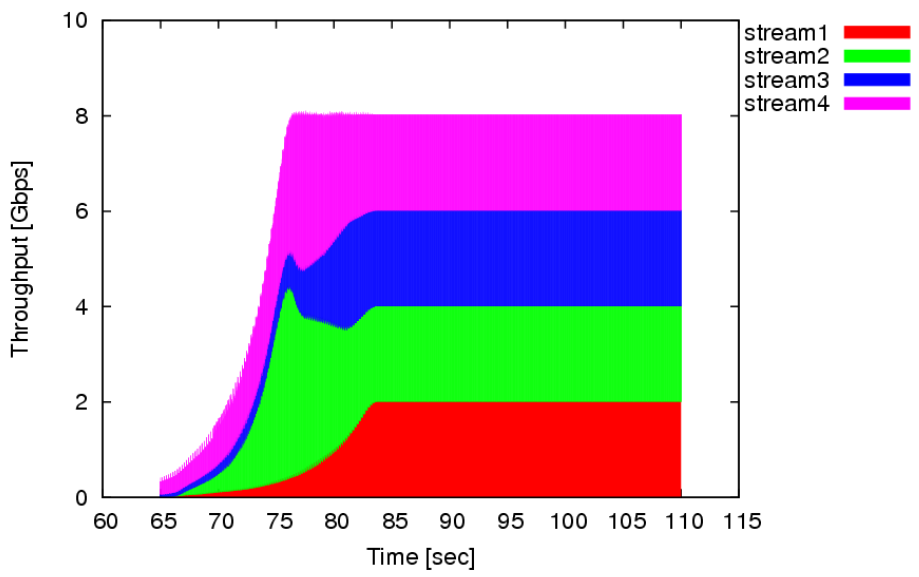


Figure 3.22: 4 streams on real LFNs with MSH (8.0Gbps pacing)

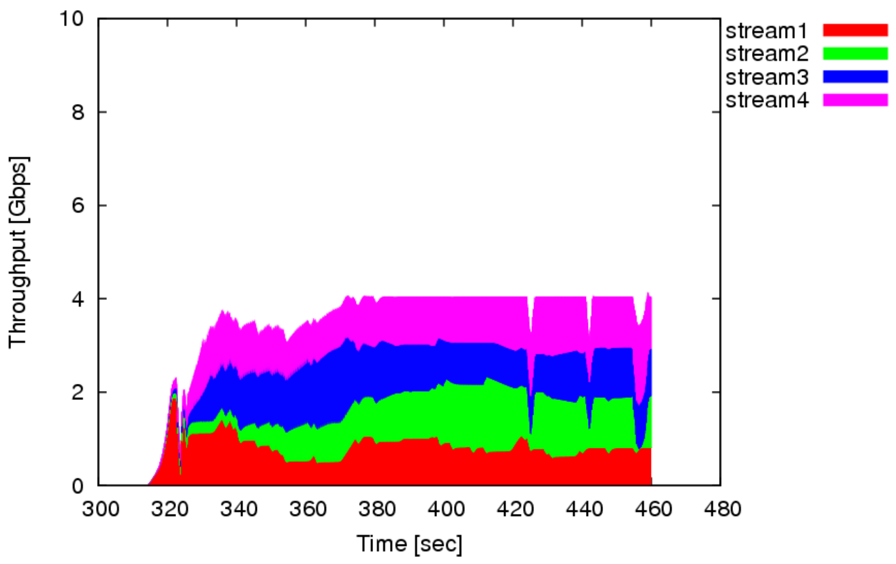


Figure 3.23: 4 of 1Gbps streams in real LFN

streams using our network testbed MaSTER-1, which is described in next section. In Figure 3.23 and Figure 3.22, the total throughput of output streams is limited to 4Gbps and 8Gbps. Our network testbed tries to make stream share the bandwidth equally, and the resulting throughput of each stream is 1Gbps and 2 Gbps, respectively. But, with a careful look, we notice that surprisingly when the MaSTER-1 paces the packets to 4Gbps at its output port, the throughput of four streams become unstable and many packet losses occurred. On the other hand, when the MaSTER-1 paces the packets to 8Gbps, all streams are stable and equally share the bandwidth. This is the opposite result of the experiment, where we send each sender host desirable throughput rate, 1 Gbps and 2Gbps, respectively. When we set sender host throughput, each 1Gbps and total 4Gbps is stable, but, each 2Gbps and total 8Gbps is unstable and result in a lot of packet losses, possibly because in total 8Gbps case, margin to the WAN PHY bandwidth (9.2Gbps) is much smaller. In the real world, it often happens that

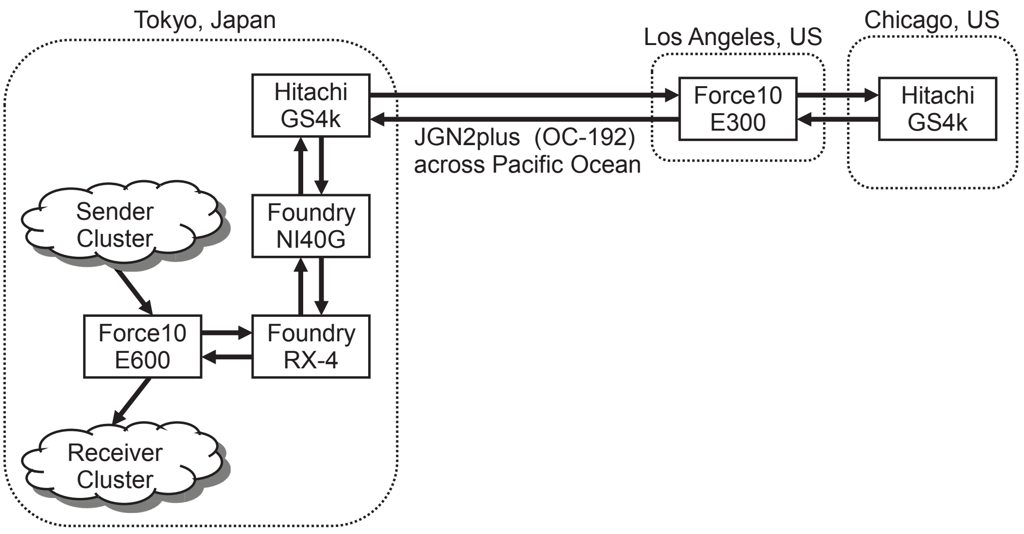


Figure 3.24: Network diagram of round-trip path between Tokyo and Chicago

the bandwidth of output port of the switch is limited. So, we investigate the buffer utilization of the switch and throughput.

3.7 The Behavior of Path-bottleneck Switch

We arrange the pseudo network and the real network paths as LFNs. The network diagram is shown in Figure 3.12. As for the real network, we use Tokyo to Tokyo via Chicago networks as shown in Figure 3.24.

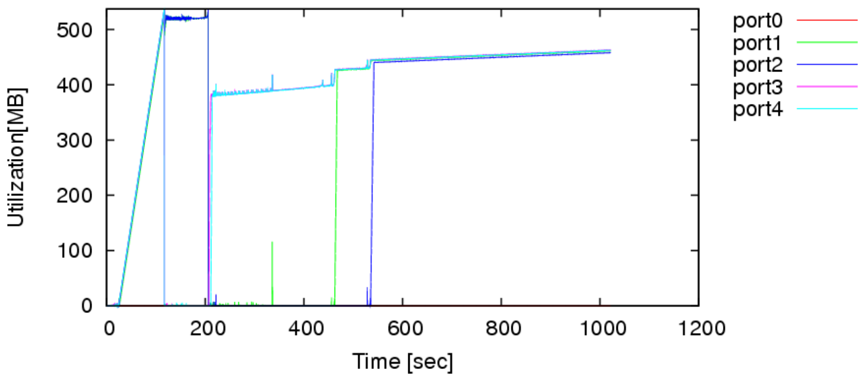


Figure 3.25: Buffer flooding of 4Gbps pacing for 4 streams

First, we show the utilization of the buffer on MaSTER-1 in Figure 3.25 and Figure 3.26, for the case of Figure 3.23 and Figure 3.22, respectively; the former limits output throughput to 4Gbps, and the buffer reaches the ceiling, and this buffer shortage may cause its unstableness. On the other hand, the latter has sufficient buffer, which limits output throughput to 8Gbps and is stable. For 340msec, 500msec, and 700msec RTT, we changed the bandwidth of output port, and we plot the relationship between buffer utilization and throughput for each RTT in Figure 3.28. This plot can be on the line of

$$Buffer_{util} = Const. - RTT \times Throughput \tag{3.1}$$

where $Buffer_{util}$ is the used size of the buffer. Figure 3.28 suggests that the

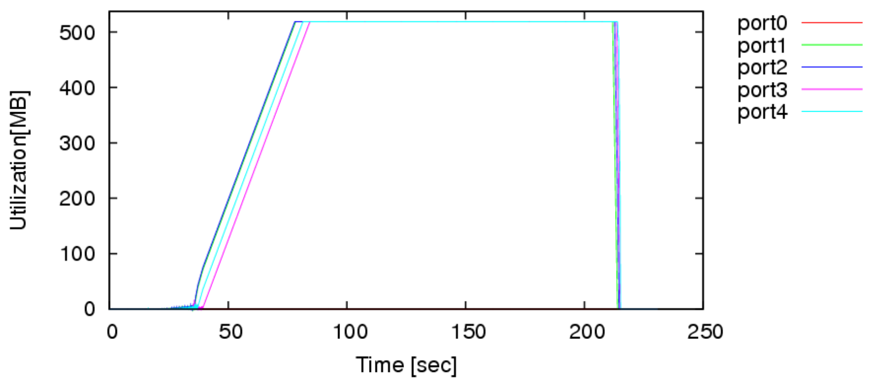


Figure 3.26: Buffer utilization of 8Gbps pacing for 4 streams (BIC-TCP)

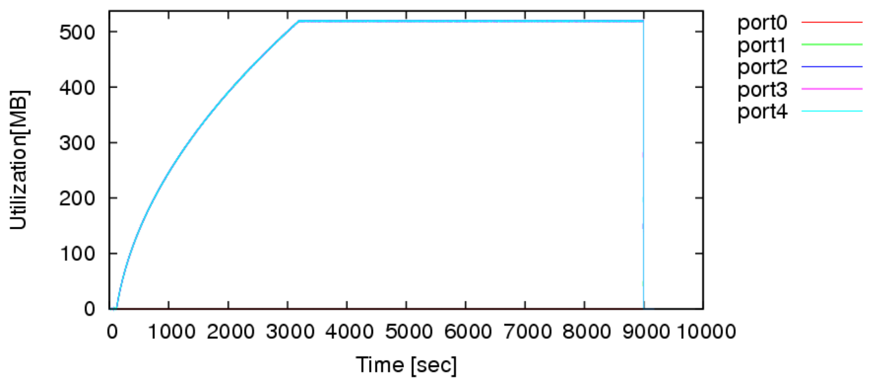


Figure 3.27: Buffer utilization of 8Gbps pacing for 4 streams (highspeed TCP)

point over the upper limit of the buffer size, communication may become unstable. We also compare the congestion control algorithms. Figure 3.26 and Figure 3.27 shows the utilization of BIC-TCP, and highspeed TCP, respectively.

3.8 Discussion

From the observation in previous sections, the followings seem to occur.

1. $cwnd$ increases while no packet loss occurs.

2. At time t_1 in Figure 3.29, $\frac{cwnd}{RTT_{orig}}$ exceeds the bandwidth of the path bottleneck, where RTT_{orig} is the RTT of the whole path.

3. At time t_2 in Figure 3.29, $cwnd$ reaches the max value, which is set by default or in advance.

4. In the period from t_1 to t_2, with the speed equal to the difference of input throughput and output throughput, the packets are stored in the buffer of MaSTER-1.

5. Since the data is stored, hence stops in buffer, RTT becomes longer, which is denoted RTT_{ext}.

6. RTT_{ext} and throughput become equilibrium, and buffer utilization becomes constant.

And, we get the followings:

$$cwnd_{max} = Buffer_{util} + Thrpt_{target} \times RTT_{orig} \quad (3.2)$$

$$RTT_{ext} = \frac{cwnd_{max}}{Thrpt_{target}} \quad (3.3)$$

where let $cwnd_{max}$ denotes the max $cwnd$ of the sender host, $Buffer_{util}$ denotes the utilization of the buffer, $Thrpt_{target}$ denotes the throughput of the port, RTT_{orig} denotes the RTT of the network path, and RTT_{ext} denotes the addition of RTT_{orig} to period of time that data is staying in the buffer, respectively.

3.9 Related Work

FPGA-based network testbeds for 10GbE have include TGNLE-1 [116], XGE-ProtoDevel [123], GtrcNET-10p3 [74], Force10 P-Series [41] and HP ProCurve [38]. TGNLE-1 has two 10GbE ports and RX side of one port is connected with TX side of another port through a FPGA, and can operate the streams at wire rate. TGNLE-1 can pace the streams but cannot merge the parallel streams from multiple 10GbE ports. XGE-ProtoDevel is a FPGA-based network testbed. It is supposed that this is placed between a NIC and a network switch. It is designed for the long-time packet analysis and the emulation of packet loss and error. It is not dedicated to operate the parallel TCP streams. GtrcNET-10p3 has three 10GbE ports and they are managed by one FPGA. It is used for packet capturing, packet generating, latency insertion and bandwidth control, etc. No research of it includes the parallel TCP streams merging. Force10 P-Series has two 10GbE XFP ports. Force10 P-Series is a FPGA-based hardware and the FPGA is used for only analyzing the packets. HP ProCurve 9400sl has four 10GbE ports, which

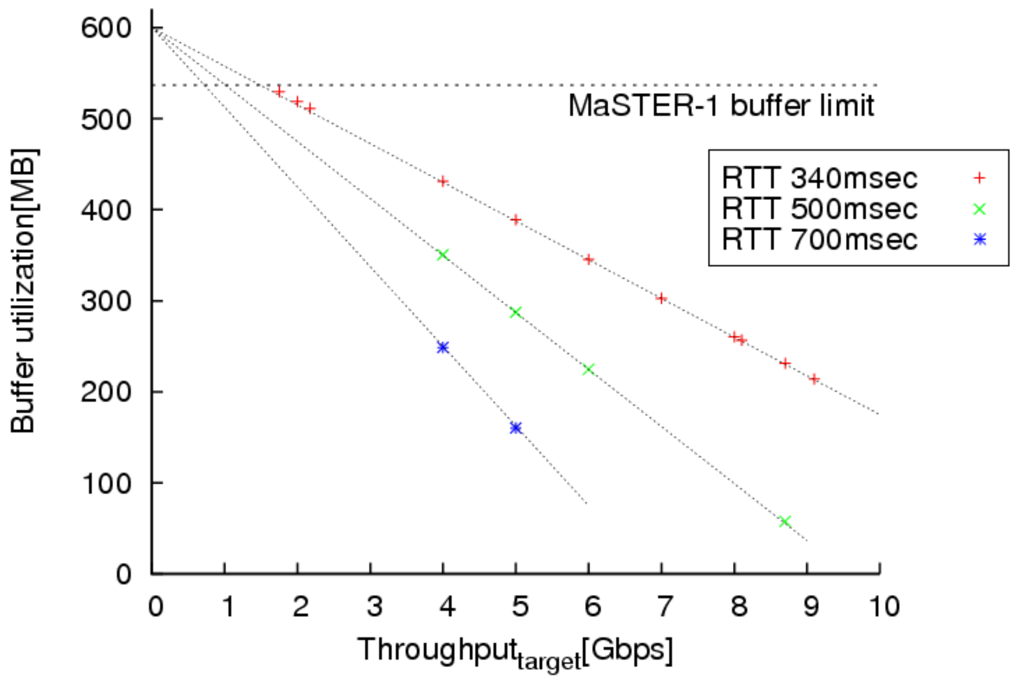


Figure 3.28: Buffer utilization and throughput of one port (the maximum value of congestion window size is 600 MB)

have a FPGA for each. But there is no research and evaluation for the parallel TCP streams.

Other researches of the parallel TCP streams include [67], [120] and Stream Harmonizer [117, 118]. In [67], the communication performance was improved by fixing TCP stacks in hosts. In [120], the end hosts have 1GbE interface and the packets that sender transmit are paced by software. It is difficult to process 10GbE streams by software. The Stream Harmonizer [117, 118] is implemented on the FPGAs of TGNLE-1. One of the ports of the Stream Harmonizer is connected to the end switch and the other port is connected to the LFNs. Parallel TCP streams in a 10GbE line are disassembled to an individual stream by the Stream Harmonizer. By suppressing a high-throughput stream, most of TCP streams are balanced. FT-Box that is also implemented on TGNLE-1 is proposed in [119]. It schedules the parallel TCP streams with several algorithms; SHIFT, DIV, and SHUFFLE, etc. But these two mechanisms can be placed in only the outside of the end switches. Therefore, the packet loss in them cannot be worked around. On the other hand, our MaSTER-1 has five 10GbE XFP ports and the potential to analyze and scheduling the parallel TCP streams. Users can use the MaSTER-1 as a network switch and internal logic in the switch can be programmed at their choice.

Researches of the implementation on FPGAs for network switching include [3]. This shows that FPGAs are less power-efficient than dedicated ASICs but that the leakage power can be reduced. It describes the possibility of the routing switch by using FPGAs. Network switches for 10GbE are, for example, Optixia [62], SmartBits [112], BigIron RX Series [21], Force10 E-Series [41], etc. They are chassis, which are able to have up to dozens or several hundred 10GbE ports. But the switching is mostly done by hardware with specific circuits. So, we cannot reconfigurable the internal scheduling algorithms in them. Similarly, Fujitsu MB86C69RBC is 10Gbps 26-port Ethernet Switch Chip [43], but it is not programmable.

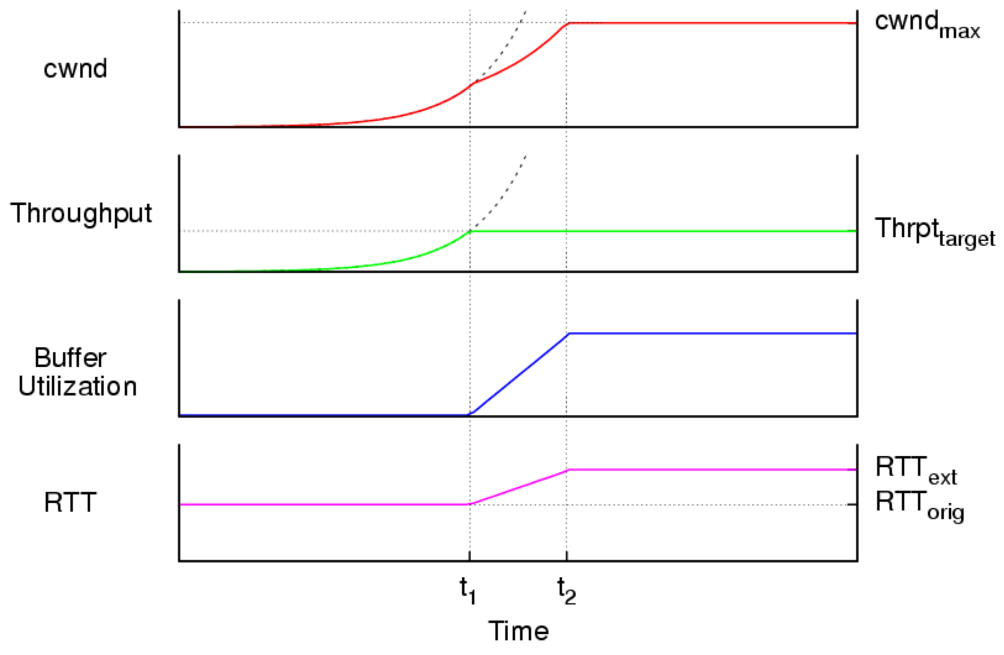


Figure 3.29: Transition of congestion window size (cwnd) and RTT with MaSTER-1 (BIC-TCP). Finally, the TCP reaches a stable state.

3.10 Conclusion

In this chapter, we propose the MSH, a hardware mechanism that merges the parallel TCP streams and paces each stream to improve communication performance of parallel TCP. We implemented MSH on the FPGA-based network testbed MaSTER-1. The MSH can buffer the packet bursts and schedule the packets to avoid unnecessary packet losses due to collision of packet bursts. We evaluated the communication performances of the parallel TCP streams with and without MSH. As a result, the throughput of the parallel TCP streams communication was more stable when we use the MSH on MaSTER-1 than the traditional switch in the pseudo LFN. In the real LFN, four TCP streams filled in 8.0Gbps totally on the 9.2Gbps bandwidth with 8.0Gbps packet pacing. Parallel TCP streams could be transferred stably by using MSH. Results of experiments show MSH is effective to stabilize parallel TCP communication. Our mechanism is a stand-alone solution for the parallel TCP streams and the MaSTER-1 has more potential to solve other problems of the communication performance in LFNs than our previous approach, two-port network harmonizer [118]. The proposed MSH can be applied to a cluster to another cluster communication with 10GbE NICs. This situation will be common in near future.

Our future work includes improvement of the implemented scheduling algorithm on the MaSTER-1. Currently we use the round-robin algorithm to decide the priorities of the streams. This provides packet-level fairness but fails to guarantee a fair allocation of bandwidth when the streams have varied size of the packets. The packet pacing with byte-level analyses is needed for it. Another future work is use of MaSTER-1 for harmonizing parallel TCP in the presence of dynamically changing cross traffic. Detailed instrumentation of cross traffic and feedback mechanism to packet scheduler is necessary.

Also, in this chapter, we introduce the importance of a path-bottleneck switch on very long-distance communication with TCP protocol. We investigate the effect of the buffer of the path-bottleneck switch on LFN, analyzing the relationship between RTT and buffer size, with several TCP congestion control algorithms. Behavior of bandwidth of the TCP communication differs by the TCP congestion control algorithm. The main contribution of this study is to show the relation between RTT, maximum bandwidth of the network, necessary buffer size of the path-bottleneck switch and TCP congestion control algorithm through detailed experiments using pseudo LFN and real LFN. Inverse-proportional relation between bandwidth and the size of the buffer e.g. low bandwidth TCP stream requires larger buffer size. We constructed MaSTER-1, a FPGA based network testbed to measure effect of buffer size at the path-bottleneck switch. We showed that we can avoid packet losses to some extent when path-bottleneck switch has huge size of buffer. Loss-based TCP algorithms show better and stable performance than delay-based algorithms when the size of the buffer at path-bottleneck switch is large enough to avoid packet losses. We also showed the guideline to set appropriate buffer size at the path-bottleneck switch. However, all the current network switch cannot have the necessary buffer size. To avoid this problem with existing switch, it is necessary to insert large buffer by external hardware such as MaSTER-1.

As a future work, we will investigate more detailed behavior of the path-bottleneck switch with wide range of TCP congestion control algorithm. The key issue is the management of the size of buffer by the cooperation of cwnd, advertise window size management and the utilization of buffer.

Chapter 4

Accelerators for Playout Generation of Monte Carlo Go

4.1 Introduction

4.1.1 Go – A Board Game for Two Players

Go is a popular board game for two players. It originated in ancient China more than 2,500 years ago, and it spread to Korea, Japan, and subsequently, to all parts of the world. Each player is given a set of either black or white stones. Given an empty line-grid, players alternately place their stone on a vacant grid point to enclose and enlarge their own territory. The player's territory comprises sets of grid points that are surrounded by his or her own stones. The goal in Go is to capture a larger territory than the opponent. We shall briefly explain the rules and terms. The board is called a *Go board*, and a square grid with an odd number of lines is printed on this board. A standard Go board has 19 $\times$ 19 grid lines, whereas smaller boards with 13 $\times$ 13 and 9 $\times$ 9 grid lines can also be used for shorter games. Each grid point of the Go board is in one of three states of occupancy—*black stone*, *white stone*, or *vacant.* The state of the Go board, or the collection of all states of all grid points at a particular time, is called a *position.* Normally, the game starts with an empty position. Each player places his or her own stone on a vacant grid point. A single turn for one player is called a *move.* Only when two grid points are horizontally or vertically adjacent, we say that the two grid points are *adjacent.* When two stones of the same color are adjacent, they are defined as being *connected.* A stone connected to one of a group of connected stones is also defined as being connected. A stone or a set of connected stones is called a *block.* A vacant point adjacent to a stone of a block is known as a *liberty* of the block.

A block dies when it loses its liberties, i.e., all adjacent points of the block are covered by stones of the opposite color. When a block dies, all stones of the dead block are removed from the Go board. In Figure 4.1(left), a block consists of two white stones "P" and is killed by a move "black Q"; the white stones are going to be removed. A player is prohibited from placing a stone in such a manner that his or her own block dies (*suicide*). In Figure 4.1(center), a move "white R" is a suicide move and is thus prohibited. It is also prohibited to repeatedly kill the opponent's blocks by alternately placing a stone at the same point (*ko*). Figure 4.1(right) shows an example of ko; the repetition of continuous moves of "white S" and "black T" is prohibited. There are several varieties of local rules on prohibiting ko.

Basically, players alternately make moves, however, if a player cannot find an appropriate move, he or she may *pass* the turn. The game is over when both

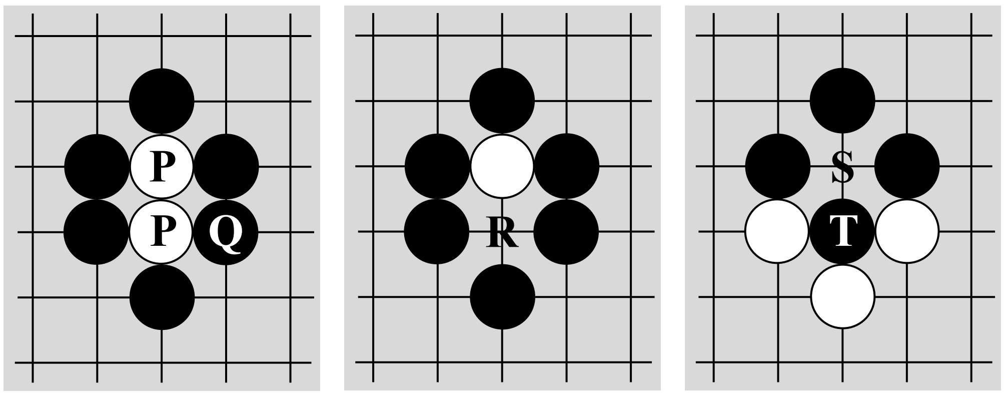


Figure 4.1: Rules of Go game: (left) When "black (Q)" is employed, a white block (P) is dead and is going to be removed. (center) Suicide move "white (R)" is prohibited. (right) Ko move "white (S)" just after "black (T)", which has killed the previous "white (S)", is prohibited.

players agree to terminate the game or both players cannot find a move and pass, i.e., they have reached a *terminal position.* There are several variants of rules on how to evaluate the territory size or how to prohibit ko. In this study, we use the so-called Chinese rule, which is simpler and is a standard rule for Computer-Go.

4.1.2 Computer Go

From the very beginning of computers, many people have attempted to develop programs that can be used to play board games such as Othello, chess, Shogi, and Go. Currently, Computer-Othello and Computer-Chess can beat human world champions, and Computer-Shogi can beat amateur champions. On the other hand, until very recently, Computer-Go was much weaker than ordinary human players.

In general, these two-player board games are represented by rooted game trees. A game tree is a directed graph whose nodes are positions of the game, edges are moves of each player, and leaves are the terminal positions where victory or loss is determined. A game starts at the root node. As each player makes a move, the current position changes to one of the children nodes. When the current position reaches a leaf, the game is over. At each node, the objective of each player is to determine the optimum move, which is likely to lead to the best leaf. In other words, one player attempts to select a move that maximizes the value of the position, while the opponent attempts to minimize the value of the position; this is called the *mini-max* search. However, the size of the tree is combinatorially large, and the complete analysis of the tree is impossible. Therefore, in many cases, the evaluation function for each position is developed beforehand. While the game proceeds, a player evaluates the candidate positions after several alternate moves by using an evaluation function and selects a move that would lead to the best position. However, it is believed that it is difficult to create an appropriate evaluation function for Go. In addition, the search space of Go is larger than that for chess and Shogi. Therefore, Computer-Go is difficult.

4.1.3 Monte Carlo Go

Since the creation of the evaluation function for Go is quite difficult, the use of Monte Carlo simulation is proposed in [22]. Monte Carlo simulation can be used to solve problems experimentally by generating a large number of samples using random numbers. In this case, the result of Monte Carlo simulation is used instead of the evaluation function to determine the best move. Monte Carlo simulation starts from a node to be evaluated and the random selection of a child node is repeated until it reaches a leaf of the tree, i.e., the game terminates. We call this single simulation from a node to a leaf a *playout.* In Monte Carlo Go, the best candidate is selected using the result of playout repetition.

Monte Carlo Go did not become popular initially, mainly because it was not strong and the machine power was not sufficient. However, two big improvements were proposed: (1) using the win-loss ratio for evaluation, instead of the difference between the territory sizes at the terminal node and (2) applying randomized tree-search techniques such as UCT [73], an extension of the UCB1 algorithm, to the tree structure; UCB1 is a probabilistic algorithm to maximize the total reward in the multi-armed bandit framework by using past selections and results [5]. This is called Monte Carlo Tree search. Currently, the Monte Carlo Go method is used in many famous and strong Computer-Go programs [35, 47].

To make Monte Carlo Go strong, a huge number of playouts are required. Further, as the size of the Go board grows, the desirable number of playouts increases because of its combinatorial nature. Since the generation of playouts has naive parallelism, several methods for the acceleration of generating playouts have been proposed. For example, MoGo uses Huygens, an IBM Power 575 Hydro-Cluster system, and it competed well with a professional human Go player on a 19 $\times$ 19 Go board [81].

4.1.4 Our Proposal for Acceleration Using FPGA

We propose a triple line-based playout for Go (TLPG): a fast and compact algorithm for an FPGA to accelerate the generation of playouts, which can be used for a 19 $\times$ 19 Go board. For generating a single playout, the following operations are required: (1) enumeration of all possible moves, (2) random selection of a move, and (3) update of the position. Both for the enumeration and the update, information on a block is used. Some blocks may cover a large part of the Go board, and global operation may be required for a single move. However, if complete information on the Go board is stored in registers, the circuit size becomes large and parallelization becomes difficult. The key idea in TLPG is as follows; to execute operations locally, global information is reproduced at local points and stored with redundancy. This enables pipelining and keeps the circuit size small so that we can operate several copies of circuits in one FPGA in parallel.

This chapter is organized as follows. In Section 4.2, we describe the preliminary experiment on how the number of playouts affects the strength. In Section 4.3, we describe our proposal, TLPG, and in Section 4.4, we present our evaluation. In Section 4.5, we describe related studies, and in Section 4.6, we present our conclusion.

4.2 Preliminary Experiments

When there exists a difference between the abilities of two players, to play an even game, the weaker player places handicap Go-stones at the initial position, and this number of handicap Go-stones is commonly used as a measure of the relative strength of Go. To evaluate the strength of Computer-Go, sometimes, matches between human Go players and Computer-Go as well as automatic matches between Computer-Go programs are carried out. Here, both the win-loss ratio and

Table 4.1: Parameters of GNU Go

Normal mode 9 × 9	–mode gtp –quiet –level 10 –never-resign –boardsize 9 –komi 7.5 –chinese-rules
Monte Carlo mode 9 × 9	–mode gtp –quiet –level 10 –never-resign –boardsize 9 –komi 7.5 –chinese-rules –monte-carlo –mc-games-per-level Number

the number of handicap Go-stones are used as its measures. GNU Go is a popular free software program for playing Go. According to their web page, "GNU Go may be 1-2 stones weaker than the top commercial Go programs" [48]. Since GNU Go is ported to many platforms, GNU Go is popularly used to measure the strength of Computer-Go.

Monte Carlo Go is also implemented for GNU Go as one of its possible strategies. Currently, it has only a 9 × 9 Monte Carlo Go mode. To see the effect of the number of playouts of Monte Carlo Go, we observe the correlation between the strength and the number of playouts. We carried out 200 matches between GNU Go in the normal mode and GNU Go in the Monte Carlo mode; in 100 matches, the first move is made in the normal mode, and in the remaining 100 matches, the first move is made in the Monte Carlo Go mode.

Table 4.1 lists the run-time option for GNU Go execution. Figure 4.2 shows the relationship between the number of playouts for each move (horizontal axis) and the win ratio of Monte Carlo mode (vertical axis). With the small number of playouts, i.e., less than 20,000, the win ratio shows an increasing trend. On the other hand, for around 100,000 playouts, this trend weakens, and for around 180,000 playouts, the win ratio may not increase even though the number of playouts increases.

Next, to observe the computational cost for generating playouts, we compare the time for playout generation and total time. Table 4.2 lists the specifications of the host machine. Figure 4.3 shows the playout generation time and the total time per playout required to make the n-th move. In the initial stage, the time for

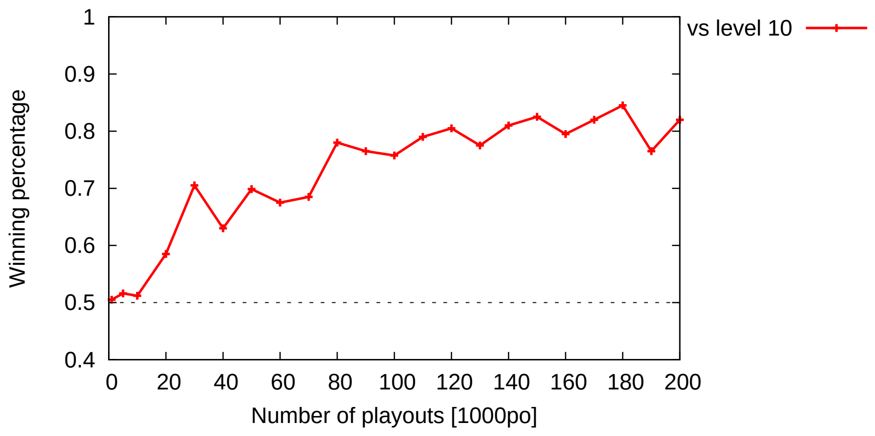


Figure 4.2: The relationship between number of playouts and win-ratio of Monte Carlo mode in the games of GNU Go (Monte Carlo mode) versus GNU Go (normal mode)

generating playouts is approximately 80% of the total time. In the final stage, although the search space becomes smaller, the time for generating playout is more than 50%. Further, for the 19 × 19 grid, this ratio of playout generation is likely to increase a lot.

Table 4.2: Specification of the host computer for experiments

Processor	Core i7 965 3.20GHz * 4CPU
Memory	6GB
Operating system	CentOS5.3 64 bit

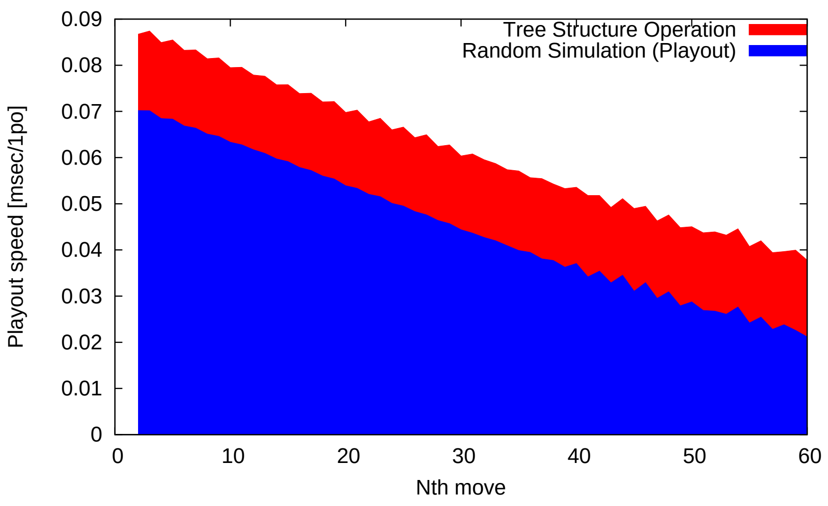


Figure 4.3: Playout speed in the n-th move in GNU Go. Monte Carlo tree search consists of tree operations and random simulations referred to as playouts. The Y-axis indicates the execution time of a single tree insertion (red) and a single playout (blue). In the initial stage, the time for generating playouts is approximately 80% of the total time. In the final stage, although the search space becomes smaller, the time for generating playout is more than 50%.

4.3 TLPG: Triple Line-based Playout for Go

4.3.1 Scope and Difficulty

As shown in the previous section, the number of playouts directly affects the strength of the Monte Carlo Go. In addition, the ratio of the time for playout generation to the total time is quite high. Most of the required operations for generating a playout are bit manipulations. This is quite different from the tree-search involving the use of UCT, where a probabilistic method is utilized and floating-point operations are required. In addition, since each playout is independently generated, the computation has naive parallelism. Thus, in this study, we concentrate on generating playouts; given a position, the win-loss result of Monte Carlo simulation is returned.

Since bit-manipulation plays a critical role, we design an accelerator for generating playouts using an FPGA. However, a naive implementation on an FPGA fails because the circuit size becomes large. As shown in Section 4.1, every stone shares its fate with the block to which it belongs, and the life and death of a block are determined by the number of liberties. However, it may be necessary

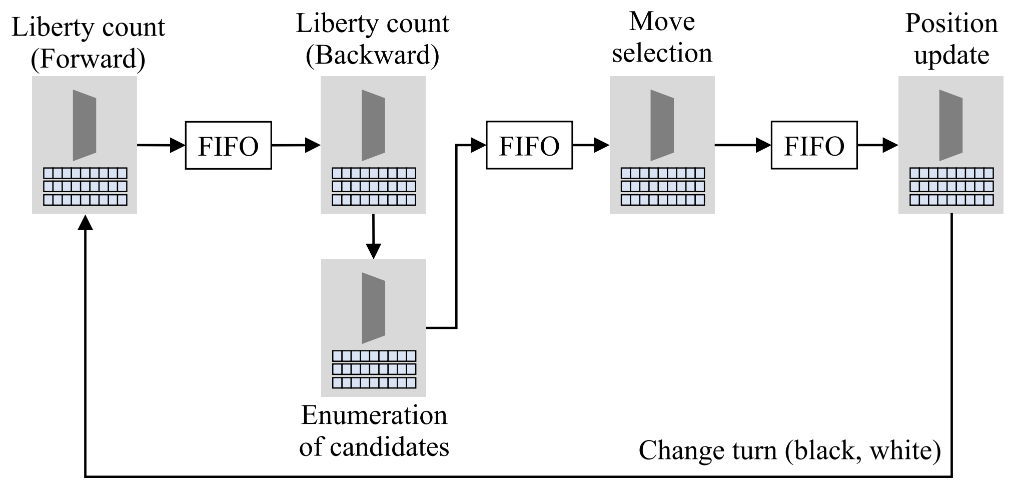


Figure 4.4: Logic diagram of TLPG

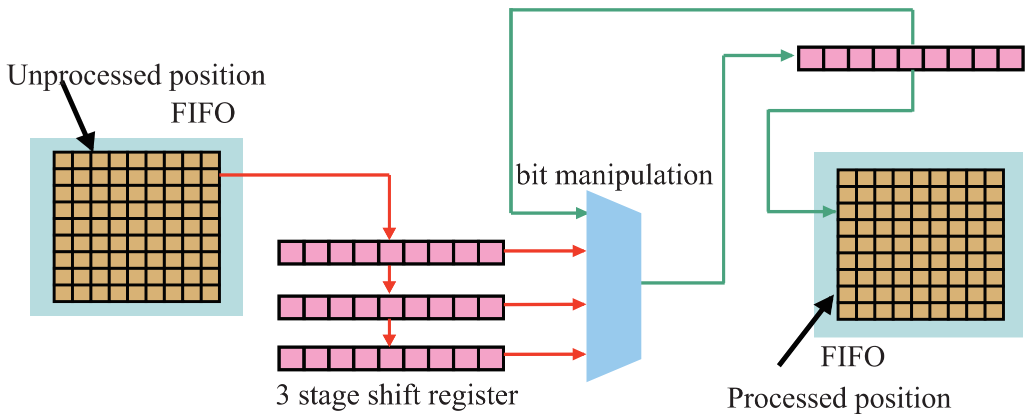


Figure 4.5: Concept of TLPG Logic

to check all the adjacent points of the block, which may cover a large part of the Go board. Thus, for one move, we may need global operation. However, if we store complete information on the position in registers, the circuit size becomes huge, and for the standard 19 × 19 Go board, this is not feasible.

4.3.2 Basic Idea of the TLPG

To keep the required hardware resources small, we count the number of liberties by local operation. The number of liberties is redundantly stored at all points of the block so that all operations can be locally performed. This enables pipelining. In addition, since the circuit size is small, we can operate several copies of circuits in one FPGA in parallel.

We propose triple line-based playout for Go (TLPG), a compact hardware algorithm to generate playouts. TLPG attempts to accelerate the generation of playouts by processing one line at a time; all points on a single line are treated in parallel, and columns are naturally pipelined. Since information on a block is reproduced at each point, it is sufficient to check four adjacent points. Thus, to operate on a single line, it is sufficient to check its adjacent lines, namely, the line above and the line below. TLPG performs all operations locally using three lines, which are stored in a three-stage shift register.

Figure 4.5 shows the concept of TLPG logic. Here, a single line from the

FIFO for the unprocessed position is retrieved and is input to a three-stage shift register. Bit manipulation on the middle line on the three-stage shift register is executed using the information on the other two lines, and the result is stored in the FIFO for the processed position. The required operations such as counting liberties, enumerating possible moves, selection of a move, and updating the position, are divided into multiple stages depending on whether the completion of the previous operation is required, and FIFOs are inserted between these stages (Figure 4.4). Here, all data for one playout are sent to the next FIFO after the current operation, which enables the generation of multiple playouts simultaneously, where the number of playouts that can be generated is the number of FIFOs since each playout occupies only one stage. Moreover, since TLPG avoids storing the complete Go board information on costly pipeline registers, the circuit size is small, which enables us to place multiple generators on a single FPGA and exploit its parallelism.

4.3.3 Algorithm Description

The generation of a single move for playout consists of the following three operations:

- Count the number of liberties of each block.
- Enumerate all possible moves.
- Select a move at random.
- Update the position.
- Change turn.

Each grid point is in one of three states {vacant, black, white}, and when a point is in either the black or white state, the block ID and liberty count of the block are stored. As for the liberty count, we categorize the blocks into three states {0, 1, 2 or more}; liberty count 0 means the block is dead, which is only used in operations of move selection and position update. When a block has one liberty, losing the liberty by a move may lead to the death of the block, and a check for suicide and ko is also required. When a block has two or more liberties, the block always survives for one move. Figure 4.6 shows the flowchart for generating a playout. We describe each operation.

1. **Count the number of liberties of each block**

   Since a block is continuous on the grid of the Go board, to count the number of liberties, TLPG scans lines in two directions: forward and backward. For each point of the target line, if a stone exists, it examines its block ID and assigns the liberty count of the same block on the line above to a block on target line. Then, it checks all four adjacent points, and if there exist several liberties, it increments the number of liberties till it becomes 2 and updates the liberty counts of all stones of the same block on the same line. Here, a double count is avoided by using a single local variable per block. TLPG scans forward, stores all processed data in FIFO, and then scans backward. Thus, TLPG obtains the correct liberty count state at all points of the block. Figure 4.7 shows the liberty count state of each point of a white block just after the forward scan (left) and after the whole operation (right).

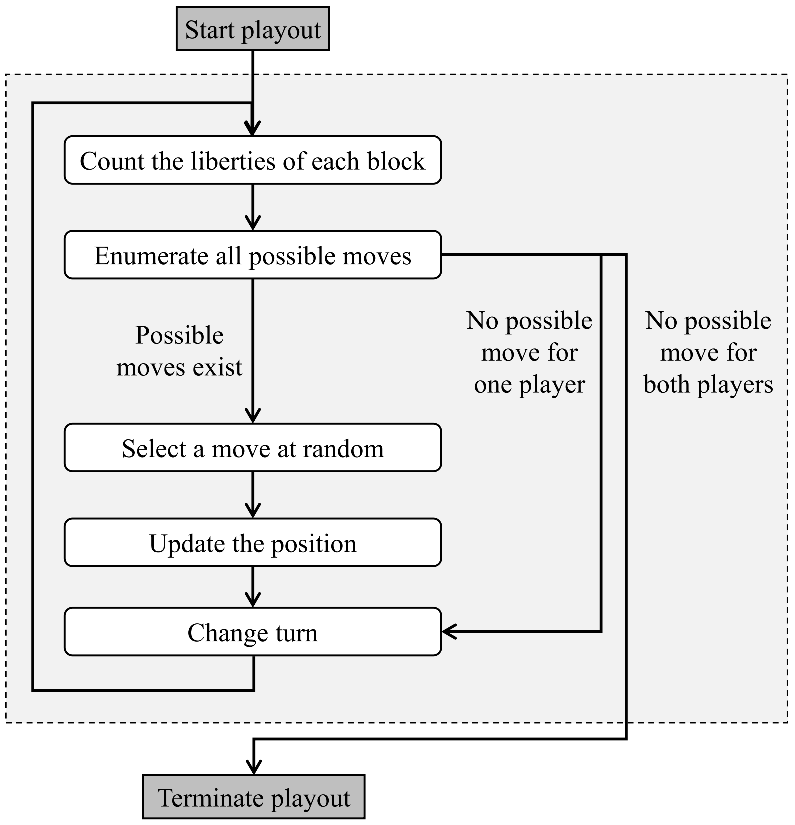

Start playout
Count the liberties of each block
Enumerate all possible moves
Possible moves exist
No possible move for one player
No possible move for both players
Select a move at random
Update the position
Change turn
Terminate playout

Figure 4.6: Flowchart for generating a playout

2. **Enumerate all possible moves**

   Check all the vacant points of the target line, and if the target point is neither suicide move nor ko move, it is a candidate. The conditions in a suicide move are; (a) all four adjacent points are not vacant, (b) all adjacent blocks of the opponent's color have two or more liberties, and (c) all adjacent blocks of the player's color have one liberty. As for ko, we relax the constrains only during playout; we prohibit the placement of a stone at the point where a single stone was just killed in the previous turn.

3. **Select a move at random**

   TLPG selects a move from all the possible moves with equal probability, checks the block ID and liberty count state of all adjacent blocks, and determines whether the block needs to be updated. Note that selection is performed using the total number of possible moves. Thus, selection starts after all the possible moves are stored in FIFO.

4. **Update the position**

   TLPG updates the target point and information on blocks, as obtained by the operation above. When a block is killed, TLPG removes all stones of the block, and when multiple blocks are merged into one block, it updates their block ID.

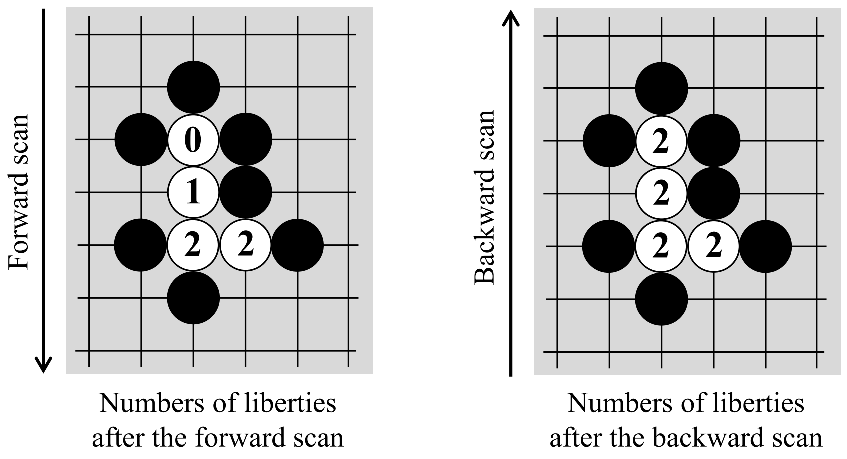

Forward scan
0
1
2
2
Numbers of liberties after the forward scan
Backward scan
2
2
2
2
Numbers of liberties after the backward scan

Figure 4.7: Calculate the numbers of liberties for a block with two-way scans

4.3.4 Detailed Data Structure

Table 4.3 lists the data structure. The state of each point is represented in 2 bits; {vacant, white, black}. The block ID is represented in 8 bits for a 9 × 9 grid and 10 bits for a 19 × 19 grid. The number of liberties is expressed in 2 bits; {0, 1, 2 or more}. Table 4.4 shows the clock counts for each process. In this table, the following stage does not start before all the data pass through the previous stage.

Table 4.3: Data structure in the TLPG

	Data size		
Data	9 × 9	19 × 19	Unit size
Turn	1 bit	1 bit	
Position state	18 bit	38 bit	2 bits per point
ID of the block	72 bit	190 bit	8 bits (9 × 9) or 10 bits (19 × 19)
Number of liberties	18 bit	38 bit	2 bits per point
Possible move or not	9 bit	19 bit	1 bit per point
Update position list	72 bit	88 bit	

4.4 Evaluation

We evaluate the TLPG on an FPGA board. We implement playout generators for the 9 × 9 grid and 19 × 19 grid. Xilinx Virtex-5 XC5VFX70T-1FF1136 is used as a target device. Our designs are synthesized with Xilinx ISE 11.2. We evaluate our designs in terms of (1) implementation, i.e., clock cycle time and circuit area, (2) speed, i.e., the cycle count for the generation of one playout, and (3) validity of the playout, i.e., whether TLPG logic generates playouts accurately.

4.4.1 Clock Cycle and Resource Consumption

First, we evaluate our design on the basis of the synthesis report. Table 4.5 lists the usage of resources and the clock cycle time for generators. The resources for

Table 4.4: Clock counts for each process in the TLPG

	Clocks	
	9×9	19×19
Liberty count (forward direction)	3	3
FIFO	11	21
Liberty count (backward direction)	3	3
Move possibility check	3	3
FIFO	11	21
Move selection	3	3
FIFO	11	21
Position update	3	3
Turn change	1	1
Total	49	79

each component of the generator are listed in Table 4.6 and Table 4.7.

Table 4.5: Resource consumption of the TLPG implementations

	9×9 (%)	19×19 (%)	capacity
Slice Registers	4268 (10)	11538 (26)	44800
Slice LUTs	3882 (9)	14545 (32)	44800
Block RAM	16 (11)	23 (16)	148
Critical path	9.572 ns (104.5 MHz)	15.972 ns (62.6 MHz)	

Table 4.6: Resources for 9×9 grid Go board

	Slice Register	Slice LUTs	BRAM
Liberty count	670	1592	0
Enumeration of the candidates	1063	718	0
Position update	328	356	0
FIFO between logic blocks	677	130	9

In obtaining these results, block RAMs are used for FIFO between the stages. The FIFO-based stage reduces the number of registers and the area of the generator. This area-effective design also helps realize reasonable clock cycle time for the 9×9 grid. However, the clock cycle time for the 19×19 grid is much slower than that for the 9×9 grid, which is because of the complexity of liberty check. The liberty check for the 19×19 grid uses around six times the LUTs as that for the 9×9 grid. Such a large combinational circuit increases the design complexity and degrades the clock cycle time.

Table 4.7: Resources for 19 × 19 grid Go board

	Slice Register	Slice LUTs	BRAM
Liberty count	1762	9345	0
Enumeration of the candidates	2551	1738	0
Position update	740	862	0
FIFO between logic blocks	1341	246	18

4.4.2 Playout Generation Speed

To evaluate the average playout generation speed, we estimate the number of playouts generated for 100 ms. The cycle counts are evaluated by performing a VHDL simulation. The actual playouts for a second are calculated from the circuit frequency in Table 4.5.

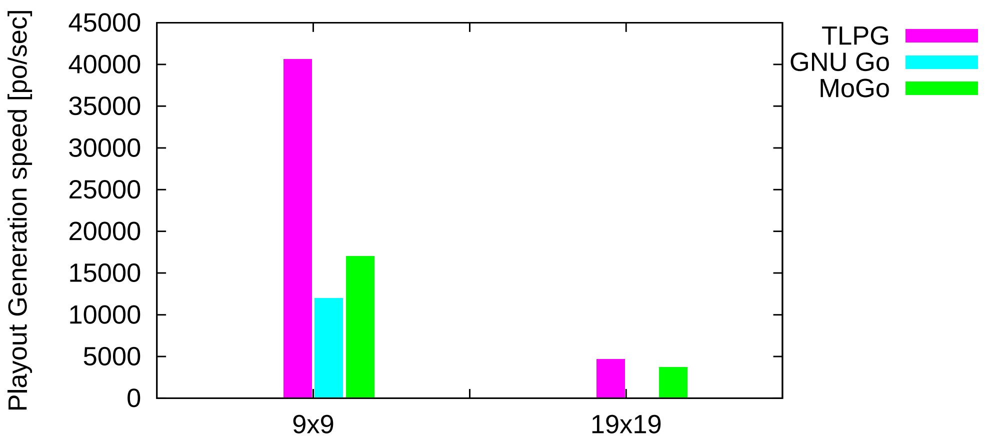


Figure 4.8: Playout generation speed of GNU Go, MoGo, and TLPG. 19 × 19 playout is unsupported in the GNU Go Monte Carlo mode.

The results are shown in Figure 4.8. TLPG generates 40,649 playouts per second for the 9 × 9 grid, and 4,668 playouts per second for the 19 × 19 grid. This playout generation speed for the 9 × 9 grid is two to three times faster than the speed of software implementation by the computer, as listed in Table 4.2 (single thread). However, the playout generation speed of TLPG for the 19 × 19 grid is approximately equal to the speed of software implementation. This is mainly because the critical path delay of TLPG logic for the 19 × 19 grid is larger than that of the logic for the 9 × 9 grid. TLPG can support three playouts simultaneously because one playout occupies only one operation stage at a time. Moreover, since our current implementation of TLPG requires one-tenth of the FPGA resources for the 9 × 9 grid and one-third for the 19 × 19 grid, we will be able to accelerate these speeds several times faster by duplicating the core circuit.

4.4.3 Validity of the Playout

We check validity of the playout of our generator in a Computer-Go game. We carry out matches between the program using our playout-generation algorithm and GNU Go. To play the real Computer-Go game, we implement a naive subtree expander instead of UCT, which expands its subtree under all possible nodes with equal probability.

We use our FPGA board MaSTER-1 to implement the TLPG logic. MaSTEr-1, which we originally designed as a testbed for network experiments, is characterized by the following; (1) five FPGAs (Xilinx Virtex-5 XC5VFX70T-1FF1136), (2) 537 MB DDR2 SDRAM components for each FPGA, (3) five 10GBASE-SR/LR optical ports, and (4) complete connections between all FPGAs by Rocket I/O (10 Gbps). In these experiments, we use one FPGA for playouts. In the FPGA, TLPG logic is operated at 104.2 MHz for 9 × 9 grid playouts.

Table 4.8: TLPG v.s. GNU Go (for Chinese-rule)

number of playouts for each move	win-loss-draw result
100	1-18-1
300	5-12-3
500	3-7-0

Since our sub-tree expander is inferior to the UCT method, we use six handicap stones. The results of the games (9 × 9 grid) are listed in Table 4.8. In this table, "N playouts" means that all candidates are simulated N times. We set komi to 0. The result for 500 playouts shows a higher winning ratio than the results for 100 and 300 playouts. We observe that as the number of playouts increases, the win-loss ratio is improved.

4.5 Related Work

In May 2009, MoGo on the Huygens IBM Power 575 Hydro-Cluster system defeated a human Go professional in an official match on a 19 × 19 grid board with a 9-stone handicap [81]. The key implication of this news is the Monte Carlo Go can defeat a professional player. It implies that Monte Carlo Go with the support of more computational power might defeat the grandchampion. To achieve high-performance computing, the acceleration of programming kernel is important. TLPG can also accelerate the playout speed, which is kernel of Monte Carlo Go.

4.6 Conclusion

Monte Carlo Go seems to be one of the promising methods to strengthen Computer-Go, which has shown good results for the 9 × 9 grid board, i.e., appears as strong as human Go players. However, performing Monte Carlo Go on a 19 × 19 grid board is difficult because its search space expands rapidly. In this study, we propose an algorithm for the generation of playouts, TLPG, which is suitable for FPGA implementation. Board data are divided into rows and treated row by row, which saves hardware resources, and FIFO is placed between operation stages. By implementing the TLPG on an FPGA, we generated 40,649 playouts/s on a 9 × 9 grid board and 4,668 playouts/s on a 19 × 19 board. In our study, we have accelerated Monte Carlo simulations for Go. However, other components such as UCT are also important for a good match. We would like to implement these components to create a much stronger Go program that defeats not only the strongest Go program but also the professional human champion.

Chapter 5

Accelerators for Continuous Skyline Computation

5.1 Introduction

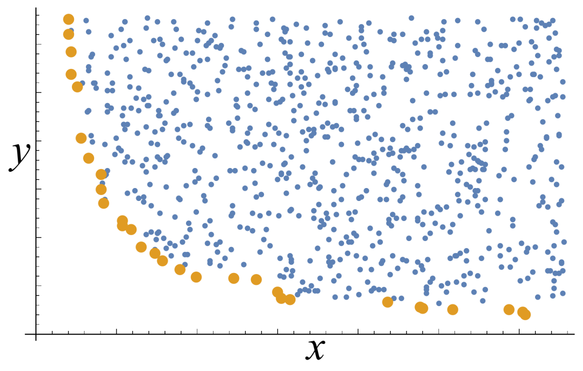
5.1.1 Skyline Computation

The skyline computation algorithm [18] is used for extracting the interesting entries from a database of multi-attribute entries. Skyline computation plays an important role in data-driven artificial intelligence, which extracts the extraordinary entries from data without requiring a model. This multi-objective optimization problem finds a Pareto (maxima) set of vectors [80], and various efficient algorithms (e.g., [82]) are studied. The computation can be stated as a geometric problem. Given two d-dimensional vectors $\boldsymbol{v} = (v_0, v_1, \ldots, v_{d-1})$ and $\boldsymbol{u} = (u_0, u_1, \ldots, u_{d-1})$, we define the *dominance relation* as follows: $\boldsymbol{v}$ *dominates* $\boldsymbol{u}$ if and only if $(\forall k \in \{0, 1, \ldots, d-1\}\ v_k \leq u_k) \bigwedge (\exists k \in \{0, 1, \ldots, d-1\}\ v_k < u_k)$. Given a set V of n points $\boldsymbol{v}_0, \boldsymbol{v}_1, \ldots, \boldsymbol{v}_{n-1}$ in the closed positive orthant of a d-dimensional space, $\boldsymbol{v} \in V$ is defined as a *skyline point* of V if there exists no $\boldsymbol{u} \in V$ such that $\boldsymbol{u}$ dominates $\boldsymbol{v}$. The set of all skyline points of V is called the *skyline* of V. The left and right panels of Figure 5.1 show examples of skylines in two- and three-dimensional space, respectively. As shown in Figure 5.5, the skyline cannot extract entries effectively from a high-dimensional space because the skyline ratio (ratio of skyline entries to whole entries) asymptotically approaches 1. The skyline ratio r is obtained as follows:

$$r = \frac{1}{n}\left(\sum_{k=1}^{n} (-1)^{k+1} \frac{\binom{n}{k}}{k^{d-1}}\right) \tag{5.1}$$

We experimentally confirmed that when r is fixed, the number of dimensions d is proportional to $\log n$. Therefore, we target mid-range-dimensional data even for larger n.

The task of computing the skyline of a *dynamic* set of points is known as the *continuous skyline computation* [92]. Real databases are updated frequently in streaming or other suitable environments. Streaming applications play a significant role in diverse fields such as fraud detection in financial trading, intrusion detection in computer networks, data processing, scheduling, and traffic control in sensor network applications [103]. The injection and ejection times of an entry into/from the database are called the *activation* and *deactivation* times of the entry, respectively. Let $act(\boldsymbol{v})$ and $deact(\boldsymbol{v})$ be the activation and deactivation times of an entry $\boldsymbol{v}$. At time t, define S_t as the skyline of



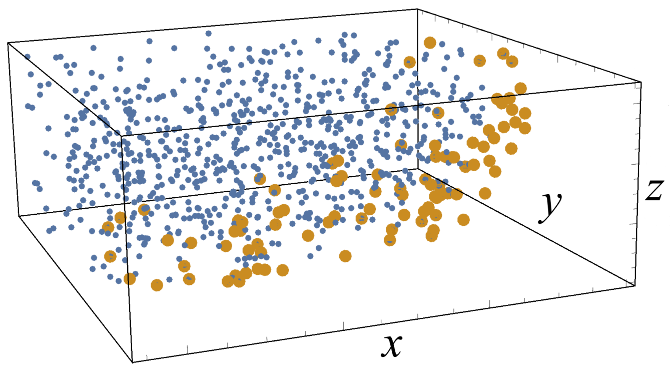


Figure 5.1: Examples of skylines in multi-dimensional spaces (top: two dimensions and bottom: three dimensions). Yellow and blue points denote the skyline and non-skyline points, respectively.

$V_t = \{\boldsymbol{v} \in V \mid act(\boldsymbol{v}) \leq t < deact(\boldsymbol{v})\}$. In a continuous skyline computation, S_t is updated from the points that are activated and deactivated at time t.

The continuous skyline computation is useful for removing non-skyline points in preprocessing for screening a large amount of data in real time. Our objective is to optimize the overall maintenance time of the continuous skyline computation. The continuous skyline computation is a difficult problem because the activation or deactivation of just one point can demote many existing skyline points from the skyline or promote many existing points to the skyline, which changes a large part of the skyline. Many previous studies have proposed data structures for managing point sets and fast algorithms for computing the skyline. These methods use spatial indexing trees, such as B-trees, R-trees, R*-trees [18, 76, 98], or quadtrees [92].

5.1.2 New Tree Structure for Skyline Computation

We employ the skyline potential of point $\boldsymbol{v}_A$ in the d-dimensional space defined by

$$1 - \sqrt[d]{\frac{|\{\boldsymbol{v} \in V | \boldsymbol{v} \text{ dominates } \boldsymbol{v}_A\}|}{n}} \tag{5.2}$$

A skyline potential of a skyline point is 1. Assuming that points are uniformly distributed, the number of points is proportional to the volume. Therefore, we use the d-th root of the dominant point ratio. A point that is dominated by many points has a low potential to become a skyline point. As the points in Figure 5.2 are activated and deactivated, the skyline potential of point A exceeds that of point B. Exploiting this difference in skyline potential, we can reduce the number of comparison operations in the skyline update.

A point dominated by many points is unlikely to become a future skyline point. Therefore, we can avoid unnecessary comparisons between the points. Existing algorithms can compute the skyline of low-dimensional of data (e.g., $d \lesssim 5$) at reasonably fast speed. However, because points are stored in the tree without considering their potential to become a skyline point, these algorithms perform redundant computations. Additionally, the number of spatial partitions in a spatial indexing tree exponentially increases with number of dimensions. Therefore, the points become sparsely distributed in the partitioned space, and the traversal performance of the tree decreases.

It is inefficient to store every dominance relation between pairs of active points in the continuous skyline computation in a simple $O(N^2)$-sized table. As the dominance relation is transitive, there are numerous dominance relations. However, many of these dominance relations (such as those between points with low potential of joining the skyline) are unlikely to be used in the skyline computation, so their calculations can be delayed. Furthermore, the number of recalculations can be reduced by storing the results of the previous activation and deactivation operations. We propose a *balanced jointed rooted tree (BJR-tree)* for continuous skyline computation. Each vertex and arc in the BJR-tree represent a point and a dominance relation respectively. As the dominance relation is transitive, a directed graph G that exactly expresses a set of points V is uniquely determined. This graph, called the *complete dominance graph*, is a transitive closure with $\Omega(n^2)$ arcs, whereas the BJR-tree contains $O(n)$ arcs. Our concept excludes arcs that are unnecessary for computing the skyline, and preserves arcs that may be utilized in future. The BJR-tree is a rooted spanning subtree of the complete dominance graph.

The higher computational speed of the BJR-tree algorithm over existing algorithms is attributed to two features: (1) appropriate hierarchical expression and (2) dimensionality independence. Regarding (1), the hierarchy in the BJR-tree reflects the points' potentials to join the skyline, namely, the *skyline potentials*. A point with higher potential is closer to the root than a point with lower potential. Existing spatial indexing methods (such as B-tree, R-tree, and quadtree), which express the clustering and proximity relationships, are unsuitable for skyline computation because the non-uniformity of the spatial distribution is not directly related to the dominance relationships. Regarding (2), BJR-tree algorithms are dimensionality-independent because they project a multidimensional space onto a simple graph using the dominance relations alone. In other words, the number of comparisons is independent of the dimensionality, so the skyline computation can be rapidly and continuously performed in any number of dimensions. Furthermore, the BJR-tree is not restricted to a set of points with a dominance relationship, but applies to any partial order. The BJR-tree is detailed in Section 5.4, and its complexities are discussed in Section 5.6.

We then propose a non-dominated relation cache (ND-cache), a data structure that accelerates the continuous skyline computation. This $O(n)$-sized structure essentially caches the non-dominated relationships (in contrast to the BJR-tree, which stores dominance relationships). The BJR-tree and ND-cache play complementary roles.

5.1.3 Serendipitous Searching Problem

Many classification methods for spatial data processing in super-high-dimensional spaces have been proposed in recent years. Clustering methods divide vectors into several groups, and outlier detection methods identify distinct entries such as

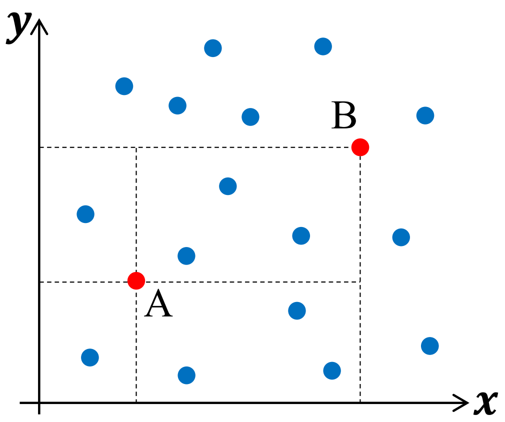


Figure 5.2: Points with high and low skyline potentials (Points A and B, respectively)

noise. The *serendipitous searching problem (SSP)* is a new cognitive problem that detects "boundary" entries; i.e., entries that locate geometrically near the rind (or envelope) of a population. The population may comprise multiple clusters or include outliers. The typical characteristics of the population are regarded as central entries (e.g., entries around the mean and median). Meanwhile, the extreme characteristics of the population are regarded as boundary entries. Very rare and valuable boundary entries can be encountered serendipitously.

We solve the SSP as a skyline computation problem, regarding the set of boundary entries as a set of skyline points. As the cultivation proceeds, the features of the boundary set gradually shift. Hence, the boundary group is maintained by an online algorithm that adds newly processed cells and removes earlier cells.

Our SSP application *Serendipiter* [52, 95, 55] is a fast cell sorter that discovers very rare cells with atypical ability from an enormous number of cells. Serendipiter includes six technologies at the single-cell level with the following functionalities: (1) cell stimulation, (2) speed control of the cells, (3) high-resolution cell measurements using multiple sensors, (4) identification of cells from multidimensional vectors of cell measurement information, (5) sorting of cells into wanted and unwanted groups, and (6) analysis of the wanted cells. A block diagram of Serendipiter is shown in Figure 5.3. The cells are identified from cell measurement information obtained by multiple sensor technologies, such as optical imaging and spectroscopy. The measurement and identification latencies must be below ten milliseconds. Existing cell sorters such as [8] process cells at sufficient speed, and microscopes enable the accurate analysis of cells. However, because these methods cannot simultaneously realize fast and accurate analysis, we have developed Serendipiter. The discovery of very rare cells (constituting one per trillion cells) in a realistic time is expected for efficient biofuel production by *Euglena* spp. and high-precision blood testing. Serendipiter, which combines an optical time-stretch quantitative phase microscope with a hydrodynamic-focusing microfluidic chip, accurately analyzes single cells and classifies 10,000 cells each second.

In biofuel production by *Euglena* spp., we require cells with a superior fat-producing ability (Super *Euglena*). However, because the measured static information relies on the lifecycle and photonic synthesis phase of the cells, it cannot directly determine the fat production potential of *Euglena* cells. Various features of *Euglena* cells can be generated by stimulating the cells and promoting muta-

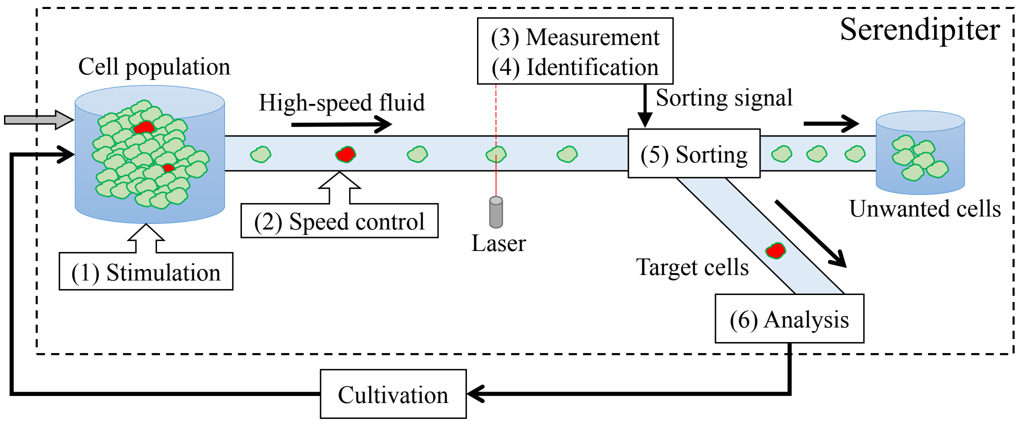


Figure 5.3: Flow diagram of Serendipiter

tion. The Serendipiter system selects and cultivates *Euglena* cells with extreme features. After several stimulation-cultivation cycles, we obtain a population of *Euglena* cells that are rarely found in ordinary cell populations. The efficiency of our proposed algorithms is verified on Serendipiter.

To determine the usefulness of the skyline cells extracted by BJR-tree, we must biochemically analyze the cells after repetitive cultivation and extraction. This study confirms that by appropriately selecting the features, we can stably isolate rare skyline cells. We also confirm the faster execution of BJR-tree than the existing algorithms, regardless of feature selection.

5.1.4 Hardware Implementation of Skyline Computation

We also propose the low-latency skyline computation accelerator (LSCA) with delayed JR-tree reshaping. The JR-tree is a variant of the BJR-tree and employs simple injection process instead of balancing injection; thus, an initially discovered child that dominates an injected node is traversed in the JR-tree. Methods for continuous skyline computation using cluster systems and hardware accelerators have been proposed in many previous studies, but it is difficult to efficiently parallelize this task because many entries must be handled but only a limited number of entries can be processed at the same time. The LSCA reduces the time required for entry deactivation in two ways: (1) it parallelizes the dominance relation calculations between a target entry and the skyline entries and (2) it delays the evaluation of postponed dominance relation calculations until the system is idle. In the LSCA, random-access memory (RAM) is implemented on the FPGA to manage the entries. Since the most frequently performed dominance relation comparisons involve current skyline entries, our method maintains all skyline entries using flip-flops on the FPGA and executes related comparison operations in parallel. This speeds up the comparison operations that have previously bottlenecked the continuous skyline computation. Linked list-based data structures are implemented in the FPGA's on-chip RAM, allowing us to achieve high-speed tree construction and traversal and compact implementation of the hardware logic.

In Section 5.2, we discuss related work. In Section 5.3, we discuss relevant concepts for static skylines. In Section 5.4, we describe our BJR-tree, and in Section 5.5, we introduce our ND-cache. In Section 5.6, we discuss the complexity. In Section 5.7, we describe experiments that compare the BJR-tree to other methods, and In Section 5.8, we discuss performances of skyline operation and other entry extraction methods and describe another application of our BJR-tree. In Section 5.9, we propose a hardware algorithm LSCA for FPGA-based continuous skyline computation. In Section 5.10, we describe experimental results about query processing latency, and in Section 5.11, we conclude this study.

5.2 Related Work

In a pioneering study, Kung et al. [80] extracted the maxima among a set of vectors. They also showed that for n vectors in d-dimensional space, the time complexity of this problem is $O(n \log_2 n)$ for $d = 2, 3$, and it is bounded between $\lceil \log_2 n! \rceil$ and $O(n \log_2^{d-2} n)$ for $d \geq 4$. The same problem has been applied to database applications, e.g., the skyline operator of Börzsönyi [18].

In static skyline computation, the input points are loaded and stored in data structures, which are accessed by a skyline computation algorithm. The block nested loop (BNL) [18], bitmap, and index [121] algorithms use no special data structures. A variant of the BNL algorithm, called the sort-filter-skyline (SFS) algorithm [31], presorts the points. The linear elimination sort for skyline (LESS) algorithm [49] improves the SFS algorithm. The sort and limit skyline algorithm (SaLSa) [6] presorts points by a monotonous limiting function. The authors of [18] proposed basic divide-and-conquer algorithms using a B-tree [7] or R-tree [53], and those of [76] proposed nearest-neighbor algorithms using an R*-tree [9]. The branch and bound skyline (BBS) algorithm [98] also uses an R*-tree. In the object-based space partitioning (OSP) algorithm [144], the partition trees store only the skyline points. BSkyTree [82] selects effective pivots for the space partitioning, which reduces the number of dominance relation computations. These methods, which construct spatial-indexing tree structures, initially build a tree from the input points, then traverse the tree to compute the skyline. These methods are optimized for static skyline computation and are unsuitable for updating the skyline.

Depending on the format of their temporal information, datasets in continuous skyline computation are classified into three types: count-based [85, 122], moving-object-based [57, 125, 83], and time-based [92]. Count-based datasets include an activation sequence with a fixed number of concurrently activated points. The complexity of the update operation for count-based datasets is discussed in [85]. In moving-object-based datasets, the points move around a multi-dimensional space and the skyline must be tracked throughout the simulation. In [83], the tracking is performed on an R*-tree. Our algorithm targets a time-based dataset with the time of activation (entry into the database) and deactivation (exit from the database).

In continuous skyline computation, the tree-building algorithm is replaced by injection and ejection algorithms. A variant of the bitmap algorithm was proposed in [42]. The Lookout algorithm [92] performs more effectively on a quadtree [39] than an R*-tree. The quadtree reportedly outperforms the R-tree in update-intensive applications [77], and is more effective in point indexing than the R*-tree [71]. The height of an n-leaf quadtree in d-dimensional space is $O(\log_d n)$. Closely placed points are stored in deep positions of the quadtree. Because the quadtree must be recursively traversed in skyline computation, the computational time is difficult to reduce. Recently, many parallelized and hardware-based methods have been proposed for this purpose. For instance, Lazy List and Lock-Free Parallel BNL [107] and APSkyline [84] have been proposed for a multi-core processor environment. The hardware-based methods include methods for GPUs [30, 14, 15] and FPGAs [137, 138].

The k-dominant skyline [27] is similar to the skyline, but contains more points under the relaxed dominance relation definition. The nearest-neighbor query [105] and convex hull [50] are well-known sampling problems with multidimensional vectors. The SR-tree [69], proposed for nearest-neighbor querying, is a structure that is a combination of an R*-tree and an SS-tree [134]. A method based on X-tree [11], a variant of R-tree, has been proposed for convex hull computation [16].

5.3 Preliminaries

In this section, we show our findings for a static skyline. We randomly generated synthetic datasets with multidimensional vectors. The specifications of our generated datasets are described in Section 5.7.

Figure 5.4 shows the change in the number of skyline points due to an increase in the number of points (called the "cardinality") in three different numbers of dimensions ($d = 2, 5, \text{and } 8$). In each dataset, points are activated cumulatively. When a point that dominates many other points is activated, the number of skyline points decreases. Such a decrease is often found in $d = 2$, but the number of skyline points continues to increase for $d = 5$ and 8.

We call the ratio of the number of skyline points to the total number of points the "skyline ratio". When the cardinality is 100k, the skyline ratio in a two-dimensional dataset is approximately 0.01%. On the other hand, the skyline ratio is approximately 1% in five dimensions and approximately 10% in eight dimensions. The skyline ratio varies with the number of dimensions; thus, the algorithms and tuning techniques optimized for low-dimensional datasets are miscast as a method for high-dimensional datasets.

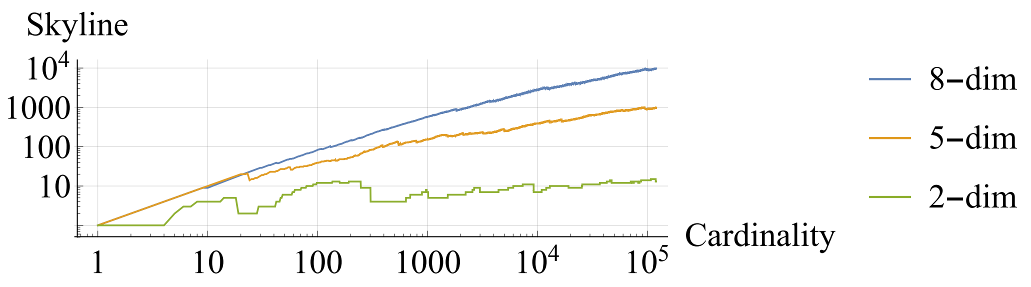


Figure 5.4: Growth of the number of skyline points with new point activations (independent distribution)

The skyline ratio in each number of dimensions is shown in Figure 5.5. We set the cardinality to 10k. We show three types of spatial distributions of the input vectors: anti-correlated (ANTI), independent (INDE), and correlated (CORR). In ANTI, nearly all points in the datasets of more than 10 dimensions belong to the skyline. In CORR, nearly all points in the datasets of more than 25 dimensions belong to the skyline. Let $A(n, d)$ be the average number of maximal vectors of n d-dimensional vectors, $A(n, d) = \frac{1}{n}A(n, d-1) + A(n-1, d)$ for $n, d \geq 2$, $A(1, d) = 1$ for $d \geq 1$, and $A(n, 1) = 1$ for $n \geq 1$ [10]. The purpose of the skyline operation is to extract interesting points. In such a situation, any skyline with a high ratio has no significance. In [28], a skyline frequency was proposed for this problem. The dominance relation of the k-dominant skyline [27] is defined as a partial order in k-dimensions ($k \leq d$). In this variant of the dominance relation, the number of cases in which a dominance relation is

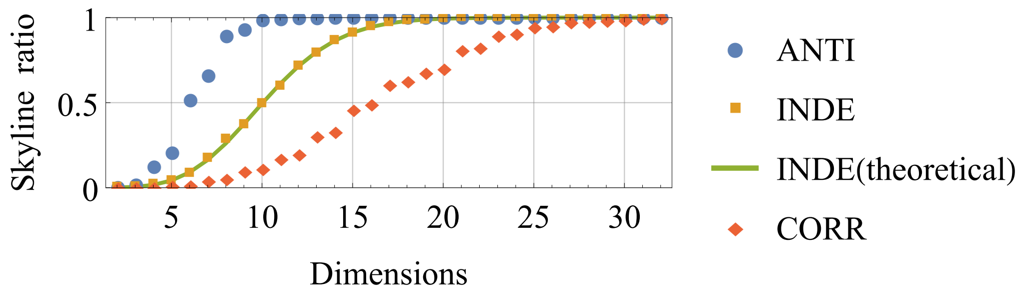


Figure 5.5: Skyline ratio versus number of dimensions for 10,000 randomly generated entries with three types of spatial distributions (anti-correlated, independent, and correlated)

established increases and the number of skyline points decreases. The continuous k-dominant skyline was studied in [75]. The acceleration of high-dimensional datasets is needed in these problems. Therefore, the targets of our work are the datasets with 2-25 dimensions.

We observed that the features of a dataset strongly depend on the spatial distribution of points and number of dimensions. The dominance relation is established easily when the number of dimensions is small but is more difficult to establish when the number of dimensions is large. Previous methods that use spatial indexing trees consider not the dominance relation but the spatial distribution of points in the tree construction phase. If the target dataset has a small number of dimensions and many dominated relationships, it is effective to manage points in geometrically hierarchic structures. However, when the dataset has a large number of dimensions and the dominated relationships sparsely exist, such structures are not effective for skyline computation. The spatial indexing trees are not suited for skyline computation, especially when the number of dimensions is large. Therefore, a data structure that can directly deal with dominance relations and omit the needless operations on points with a low skyline potential will accelerate skyline computation.

5.4 BJR-tree

5.4.1 Algorithm Description

In the BJR-tree concept, a point that is dominated by many other points is unlikely to become a skyline point in future, so its precise information need not be maintained. The BJR-tree is a rooted tree that expresses the dominance relations in a given set V, with n points in the closed positive orthant of a d-dimensional space. Each vertex represents a point and each arc (directed edge) represents a dominance relationship. The vertex set of a BJR-tree is $\{O\} \cup V$, where O is the origin of the d-dimensional space. Note that O dominates all n points in V, and its corresponding vertex is the root of the BJR-tree. A *complete dominance graph* is a directed acyclic graph in which the vertices and arcs correspond to the points $\{O\} \cup V$ and all dominance relations, respectively. A BJR-tree T is a spanning subtree of the complete dominance graph with the following properties: (1) If $\boldsymbol{v}_X$ is a skyline point, then T directly connects O to the corresponding vertex X, and (2) If $\boldsymbol{v}_X$ is not a skyline point, then T directly connects a non-origin point to the corresponding vertex X.

A given set of points in a multi-dimensional space admits multiple BJR-trees. If an arc extends from A to B, then point $\boldsymbol{v}_A$ corresponding to A dominates

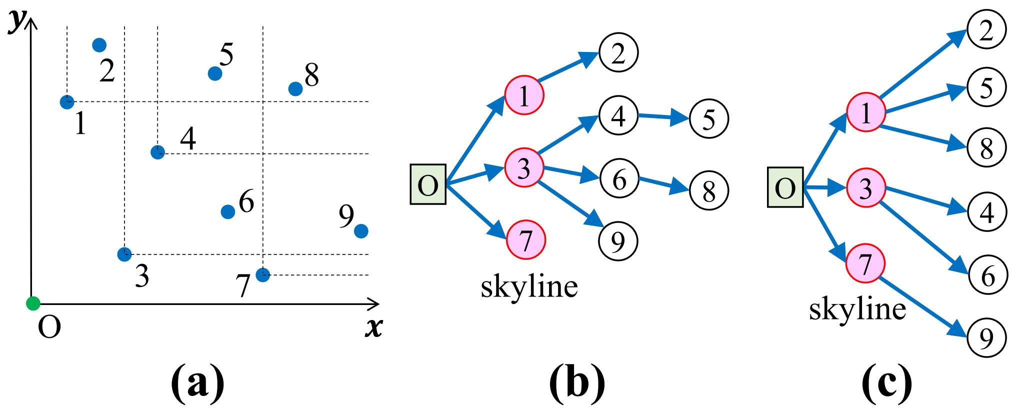


Figure 5.6: Examples of BJR-trees (b,c) constructed from nine points (a) represented by two-dimensional vectors

point $\boldsymbol{v}_B$ corresponding to B. However, even if $\boldsymbol{v}_A$ dominates $\boldsymbol{v}_B$, T does not necessarily connect A to B by one arc. A point belongs to the skyline if and only if the corresponding vertex is a child of O.

Figure 5.6 shows two examples of BJR-trees constructed from nine points in a two-dimensional space. One of these BJR-trees is sufficient to determine the skyline.

To output the skyline, we need to enumerate the children of O. There is no need to traverse the tree. The BJR-tree offers two advantages in continuous skyline computation: (1) the small size of the BJR-tree and (2) the applicability of BJR-tree to partially ordered datasets as well as multi-dimensional datasets. Regarding (1), the tree contains $O(n)$ edges, whereas the complete dominance graph contains $\Omega(n^2)$ edges. Regarding (2), BJR-tree accesses the values of each dimension when computing the dominance relation, and its concept is applicable to any dataset with a transitive partial order. In contrast, methods based on spatial indexing trees evaluate the dominance relations between a point and the maximum or minimum corners of the partitioned regions, using the Manhattan distance between the points. In methods that select the dimensions for presorting, the constructed trees are of limited applicability in solving independent multi-objective optimization problems. A point corresponding to a vertex in a deeper layer of the BJR-tree tends to have a low skyline potential, whereas a point corresponding to a vertex near O tends to have a high skyline potential.

Continuous skyline computation activates and deactivates the entries over time. The BJR-tree dynamically injects new vertices and ejects existing vertices. The BJR-tree is built by repeating the injection and ejection operations, which are implemented by Algorithms 1 and 2, respectively.

The BJR-tree maintains the dominated relations discovered in the injection and ejection operations as arcs. This information may be utilized when the skyline potential of the points increases because of the ejection operations. This information is used when related points approach the root under subsequent ejection operations. When executing multiple ejection operations in a single time-step, it is helpful to collectively remove points and inject the children into their parents.

We now show a running example of the injection and ejection operations. Suppose that the BJR-tree in Figure 5.6 (b) is constructed from the dataset

Algorithm 1 Injection (base)

```
1: procedure inject($r$: root, $v$: new vertex)
2: $C \leftarrow$ children of $r$;
3: for all $c \in C$ do
4:     if $c$ dominates $v$ then
5:         inject($c, v$);
6:         return
7: set $v$ to $r$'s child;
8: for all $c \in C$ do
9:     if $v$ dominates $c$ then
10:        move $c$ to $v$'s child;
```

Algorithm 2 Ejection

```
1: procedure eject($v$: ejected vertex $\neq O$)
2: $p \leftarrow$ parent of $v$;
3: $C \leftarrow$ children of $v$;
4: remove $v$ from $p$;
5: for all $c \in C$ do
6:     inject($p, c$);
```

shown in Figure 5.6 (a), and that point 3 is to be deactivated. First, node 3 is removed from the tree. Second, the children of node 3 (nodes 4, 6, and 9) are injected into the root node (the parent of node 3). Node 4 neither dominates nor is dominated by node 1 or 7; consequently, node 4 becomes a child of the root node. Node 6 neither dominates nor is dominated by node 1, 4, or 7, so becomes a child of the root node. Finally, node 9 is dominated only by node 7, which has no children; therefore, node 9 becomes a child of node 7. Figure 5.7 (b) shows the new BJR-tree constructed after ejecting node 3 from the BJR-tree in Figure 5.6 (b).

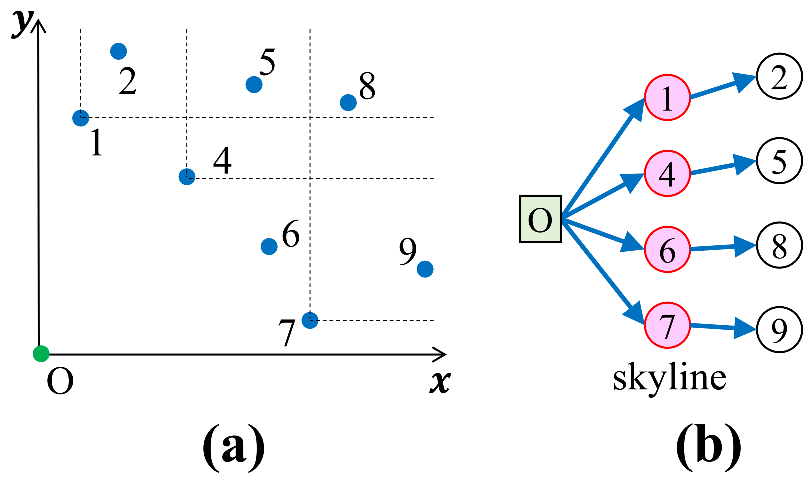


Figure 5.7: Points in two-dimensional space (a) and a BJR-tree (b) derived from the dataset shown in Figure 5.6 after ejecting point (node) 3

5.4.2 Lazy Strategy

The `inject()` and `eject()` procedures (in Algorithms 1 and 2 respectively) generate one possible BJR-tree representing the current set of points. The generated tree depends on the child-selection order in the first iteration of Algorithm 1. For example, multiple BJR-trees can be generated from the complete dominance graph in Figure 5.8 (a), which is transitively closed. Note that graph (b) of Figure 5.8 is the transitive reduction of graph (a) and is not a BJR-tree. Clearly, the rooted tree generated by `inject()` without `eject()` (see Figure 5.8 (c)) is a spanning subtree of the transitive reduction.

Algorithm 1 increases the depth of the tree. A shallow vertex in the BJR-tree corresponds to a point with high skyline potential. Such points must be processed immediately. Conversely, points far from the origin (i.e., deeper vertices) can be computed later because their results may not be used until the end of the computation. However, in tree (c) of Figure 5.8, the dominance relation between B and E is computed even though both points are more than two hops from A.

To delay such operations, we substitute Algorithm 1 with Algorithm 3. A tree built by Algorithm 3 is displayed in Figure 5.8 (d). This tree is a subtree of the transitive closure and satisfies the conditions of a BJR-tree. Rather than increasing the tree height, Algorithm 3 increases the child number of a parent. By using appropriate thresholds on the dataset, we can balance the number of children and the depth of each vertex. Tree (d) is low-height but unbalanced. In contrast, tree (c) has the smallest maximum number of children among the trees in Figure 5.8 and is properly balanced, but has a large height. Figure 5.8 (e) shows another tree built by Algorithm 3, which is a compromise between trees (c) and (d). Building the appropriate type of BJR-tree equates to a lazy evaluation problem. The timing of the lazy evaluation is determined by imposing a limited depth, as shown in Section 5.6.

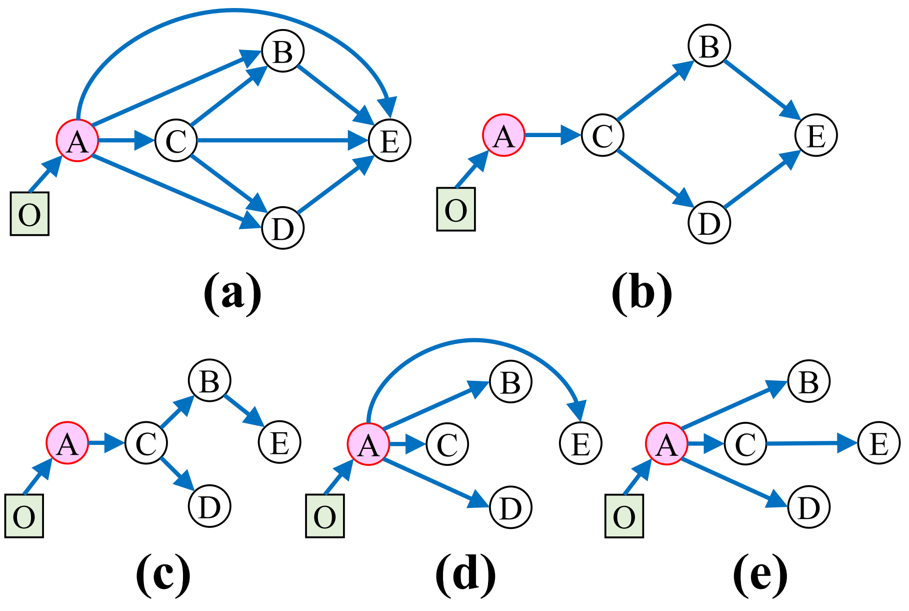


Figure 5.8: Examples of rooted trees derived from transitive closure (a). The reachabilities of the rooted trees differ from those of the original transitive closure and its transitive reduction (b). Tree (c) was generated by Algorithm 1, and trees (d, e) were generated by Algorithm 3.

5.4.3 Balancing Injection

In the previous subsection, we discussed the balance between the tree height and number of children. Here, we balance the numbers of descendants of vertices at the same depth. When multiple vertices dominate an injected vertex, Algorithm 1 selects and traverses from the first-found vertex, which unbalances the BJR-tree. When vertex X is injected to vertex Y and Y has children Z and W who dominate X, either Z or W is chosen, whichever equalizes (as far as possible) the number of descendants of Z and W. This concept, which underlies Algorithm 3, balances the number of descendants of each vertex. The number of descendants of vertex x is denoted $Desc(x)$. In the ejection process (Algorithm 2), the number of calls of the injection function equals the number of children of the ejected vertex. A single ejection operation invokes multiple injection operations. A BJR-tree is balanced directly in each invoked injection operation. Therefore, the BJR-tree remains balanced through the entire ejection operation, and the ejection process yields a balanced BJR-tree.

Algorithm 3 Injection (lazy evaluation and tree balancing)

```
1: $d$: the depth at which a vertex is injected lazily
2: **procedure** inject($r$: root, $v$: new vertex)
3: **if** $r = O$ or $Depth(r) < d$ **then**
4:     $C \leftarrow$ children of $r$;
5:     $g \leftarrow +\infty$;
6:     $t \leftarrow$ null;
7:     **for all** $c \in C$ **do**
8:         **if** $c$ dominates $v$ and $Desc(c) < g$ **then**
9:             $t \leftarrow c$;
10:            $g \leftarrow Desc(c)$;
11:    **if** $t \neq$ null **then**
12:        inject($t, v$);
13:        **return**
14: set $v$ to $r$'s child;
15: **if** $r = O$ or $Depth(r) < d$ **then**
16:    **for all** $c \in C$ **do**
17:        **if** $v$ dominates $c$ **then**
18:            move $c$ to $v$'s child;
```

5.5 ND-cache

In the datasets considered here, once a set of skyline points p has been dominated by a set of newly activated points q, some of the p are restored as skyline points by deactivating q. In such datasets, we can reduce the recalculation by saving information about the former dominance relation calculations. Because non-dominated relations (except those of the skyline) are not stored in the BJR-tree, many recalculations are required.

To solve this problem, we propose a non-dominated relation cache (ND-cache) that stores recently computed skylines. The ND-cache comprises an array C of length N (where N is the total number of given points). Each value C_i is updated at time t_1 if and only if point e_i is a skyline point at t_1. At time t_2, if C_a equals C_b, then e_a cannot dominate e_b and e_b cannot dominate e_a (we call this case a "cache hit."). If C_a and C_b are unequal, the standard comparison is computed.

As an example, Tables 5.1 and 5.2 show the history of the skyline and the contents of ND-cache at time T_3 of Table 5.1, respectively. As e_2 and e_3 are

Table 5.1: History of the skyline

Time	T_0	T_1	T_2	T_3
Skyline	$\{e_0, e_1\}$	$\{e_0, e_1, e_2, e_3\}$	$\{e_5\}$	$\{e_0, e_1, e_6\}$

Table 5.2: Contents of ND-cache at time T_3. The timestamp at which entry e_x last belonged to the skyline before T_3 is stored as a value at index x.

Index	0	1	2	3	4	5	6
Value	T_3	T_3	T_1	T_1	null	T_2	T_3

equal in the skyline cache, both points simultaneously belong to the skyline, and neither dominates the other. Table 5.3 compares the complexities of ND-cache, a non-cached method, and a method storing all former comparison results. The spatial complexity of the method that caches all results is $O(N^2)$, versus $O(N)$ for the ND-cache. The time complexity of a hit is $O(1)$ in a fixed-dimensional space.

We consider the situation in which the skyline S composed of s points is completely dominated by a newly activated point, and one new skyline point is deactivated immediately. The ejection operation invokes $s(s-1)$ comparison operations. However, all points in S have the same value in the ND-cache. The ND-cache guarantees that the results of all $s(s-1)$ comparison operations are non-dominated.

Table 5.3: Complexity of the `dominates`() function

Complexity	No cache	Cache all	ND-cache
Time	$O(d)$	$O(1)$ when hit	$O(1)$ when hit
Space	None	$O(N^2)$	$O(N)$

The ND-cache stores information about former dominance relation calculations in a one-dimensional array instead of a set. This feature realizes the following advantages: (1) the later skyline is preferentially stored; (2) the old skyline, which has dropped into a deeper position, is maintained and not overwritten; and (3) the ND-cache is easily implemented.

The dominance relation includes both dominated and non-dominated relations. The BJR-tree stores the dominated relations and the ND-cache stores the non-dominated relations between two skyline points. The BJR-tree and the ND-cache complement each other. However, the ND-cache is applicable not only to BJR-trees but also to other continuous skyline computation algorithms.

5.6 Complexity Analysis

We first describe the spatial complexity of our BJR-tree. Each vertex in the BJR-tree contains the information of its parent and a variable-length list of its children. The spatial cost of the fixed information is $O(1)$ per vertex; thus, the total cost is $O(n)$, where n is the number of points that are active at the same time. Therefore, the spatial complexity of the BJR-tree is $O(n)$.

We now compute the complexities of the existing algorithms and our algorithm on a 2-dimensional dataset. The time complexity is the number of comparison operations of the dominance relation (which constitute the most expensive part of the algorithm). The average number of skyline points in a set of n independent d-dimensional vectors $A(n, d)$ is given by [23]

$$\frac{1}{(d-1)!}\left(\sum_{k=1}^{n}\frac{1}{k}\right)^{d-1} \le A(n,d) \le \left(\sum_{k=1}^{n}\frac{1}{k}\right)^{d-1} \tag{5.3}$$

When $d = 2$, we have $A(n, 2) = H(n)$, where $H(n)$ denotes the n-th harmonic number. In the left panel of Figure 5.9, the number of children of the root is the same as the number of skyline points $H(n)$. Because the numbers of children in the BJR-tree are balanced, the average number of descendants of the vertices at depth 1 (e.g., vertex P), denoted by $D_1(n)$, is $\frac{n}{H(n)}$. Therefore, the number of children of a depth-1 vertex, $C_1(n)$, is $A(D_1, 2) = H\left(\frac{n}{H(n)}\right)$. These quantities meet the following criteria.

$$D_0(n) = n, \quad C_0(n) = H(n) \tag{5.4}$$

$$C_i = H(D_i), \quad D_k \prod_{i=0}^{k-1} C_i = n \tag{5.5}$$

We first describe the complexity of a single injection, which causes a recursive traversal of the BJR-tree. The dominance relations between the children of the root and the injected vertex are calculated at depth k with complexity $O(C_{k-1}(n))$. Therefore, the complexity of a single injection is

$$\begin{aligned}\sum_{i=0}^{d} C_i \quad &= C_0(n) + C_1(n) + \cdots + C_d(n) && (5.6)\\ &= O(\log n) + O(\log n - \log\log n) + \cdots && (5.7)\\ &= O(\log n) && (5.8)\end{aligned}$$

We next show the time complexity of a single ejection. When a vertex R at depth k is ejected (Figure 5.9, right), all of R's children (numbering $C_k(n)$) are injected into R's parent Q. If the depth of R equals or exceeds L, no dominance relation is calculated by the lazy evaluation; otherwise, the complexity of each injection is $O(\log D_{k-1})$. The time complexity of a single ejection without lazy evaluation is

$$\begin{aligned}C_k \cdot O\left(\log D_{k-1}\right) \quad &= O\left(H(D_k)\log D_{k-1}\right) && (5.9)\\ &= O\left(\log D_k \log D_{k-1}\right) && (5.10)\\ &= O\left(\log^2 n\right) && (5.11)\end{aligned}$$

In the BBS method [98], the average complexity of both injection and ejection is $O(\log^2 n)$. In LookOut [92], the average complexities of injection and ejection are $O(\log n)$ and $O(\log^2 n)$, respectively, if the ejection of the LookOut consists only of the computation of dominance relations between the vertices. Actually, when traversing an R*-tree or quadtree, LookOut additionally needs the comparison operations between the ejected point and the maximum corner of the regions expanded in a heap. The worst-case cost of these operations is $O(n)$. Therefore, the BJR-tree incurs lower injection and ejection costs than BBS and

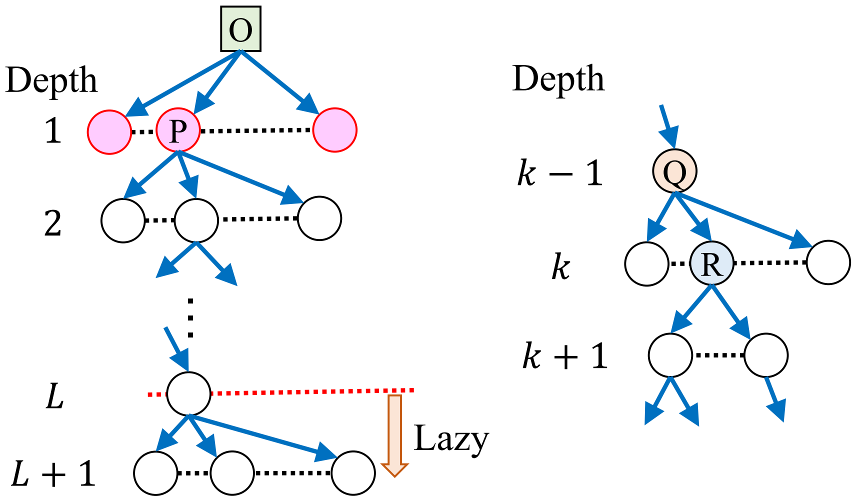


Figure 5.9: An example of BJR-trees built by the lazy algorithm at depths below L (left). When a vertex R is ejected, its children are moved to below vertex Q (right).

LookOut, and more rapidly ejects the vertices deeper than L. Asymptotically, the complexity of the BJR-tree is the same as or better than those of previous methods.

In the worst-case injection scenario, when all active points belong to the skyline, the complexity of single injection into a BJR-tree is $O(n)$. In the worst-case ejection scenario, when the ejected point is the only skyline point and all others belong to the new skyline, the complexity is $O(n^2)$. The worst-case complexities of BJR-tree match that of LookOut.

5.7 Experimental Evaluation of BJR-tree

We now compare our proposed algorithm with two existing algorithms using two types of datasets: randomly-generated synthetic datasets and datasets of real-world cell measurements. Datasets for continuous skyline computation contain both spatial and temporal information, i.e., the vector values of the points and their activation/deactivation times, respectively.

5.7.1 Synthetic Datasets

We generated three types of random synthetic dataset, characterizing the spatial information by the correlation strengths of the spatial distributions. In a **correlated dataset (CORR)**, any point A that dominates another point B in one dimension tends to dominate B in other dimensions. All pairs of criteria for these points are strongly positively correlated. In an **independent dataset (INDE)**, each multi-dimensional vector is randomly determined and there are no correlations among dimensions. In an **anti-correlated dataset (ANTI)**, any point A that dominates another point B in one dimension tends to be dominated by B in the next dimension. In this dataset, the $2k+1$-th and $2(k+1)$-th axes are strongly negatively correlated. CORR datasets are easily handled because of their low skyline ratio, whereas ANTI datasets are well known to be difficult to handle [108]. The spatial features of static skyline computations are categorized in [18]. In the synthetic datasets of this study, the activation time $act(\boldsymbol{v})$ and

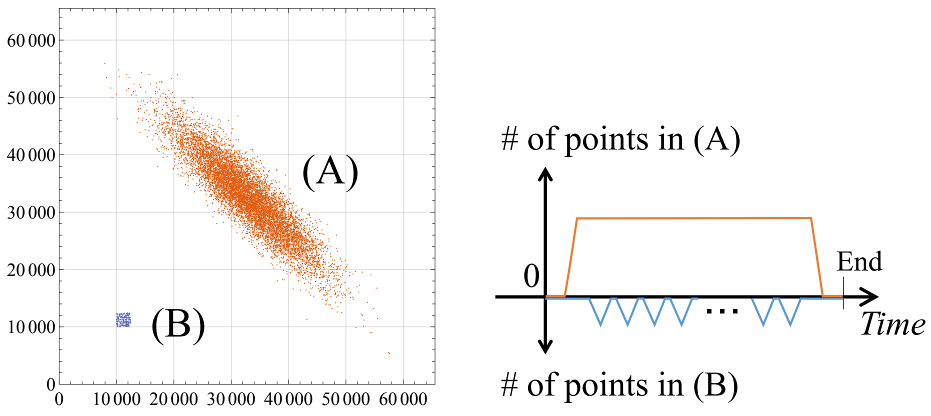


Figure 5.10: Model of CONC. Set (A), with a strongly anti-correlated spatial distribution, is activated at the start and deactivated at the end. The points in set (B), which dominate most of the points in (A), are repeatedly activated and deactivated. At each activation (deactivation), some of the points in (A) are purged from (readmitted to) the skyline. These operations require heavy computational effort.

lifetime $(deact(\boldsymbol{v}) - act(\boldsymbol{v}))$ of each point $\boldsymbol{v}$ were randomly selected.

The synthetic datasets were generated from simple parameters. To provide a pathological benchmark, we created additional synthetic datasets with a drastically changing skyline. This dataset, called the concealed-skyline dataset (CONC), is shown in Figure 5.10. Previous skyline points repeatedly leave and then return to the skyline, forcing the same dominance relations to be repeatedly recalculated. This dataset consists of a set of anti-correlated points (A) and multiple sets of independent points near the origin (B). When only (A) is activated, the skyline is a subset of (A), but when a set in (B) is activated, most of the skyline points in (A) are dominated. Then, when the set in (B) is deactivated, the points that were previously part of the skyline return to it. Such a dataset tends to require longer processing time than other datasets consisting of similar numbers of points.

The generated datasets contained up to 320,000 points in spaces of dimensions 2 to 25, and their parameters are shown in Table 5.4. For all criteria, smaller values are better, i.e., points with smaller values dominate those with larger ones.

5.7.2 Real Cell Measurement Datasets

We then extracted features from cell images taken under a fluorescence microscope and created real-world datasets by using the feature vectors. The latencies of the proposed and existing methods were compared on these real-world datasets. Three types of real-world datasets are described in detail below.

Human Protein Cell Images

High-resolution images were taken from the CYTO 2017 Image Analysis Challenge [36]. These images, stored in the Human Protein Atlas database, were taken by a Leica SP5 confocal fluorescence microscope. They show immunostained human proteins in four fluorescence channels (see Figure 5.11): (a) DAPI for the nucleus (blue), (b) antibody-based staining of microtubules (red), (c) endoplasmic reticulum (yellow), and (d) protein of interest (green). Each image was tagged with 19 types of labels, 13 indicating the protein location and six indicating rare events in the cells. For example, images (a), (b), and (c) in Figure 5.12 were tagged with the protein-location labels "Mitocondria," "Cytosol," and "Plasma membrane," respectively. Images (d), (e), and (f) were tagged with the rare-event labels "Cytokinetic bridge," "Focal adhesion sites," and "Nuclear speckles" respectively. The cell positions in the images are synchronized among the channels.

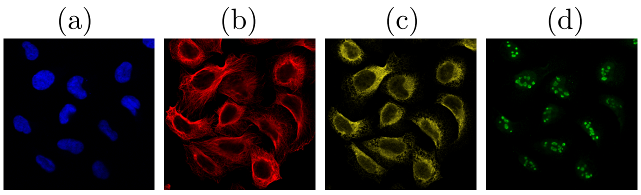


Figure 5.11: (a–d) Human protein cell images from the CYTO 2017 Image Analysis Challenge. Each image is captured in four channels: (a) DAPI for the nucleus, (b) antibody-based staining of microtubules, (c) endoplasmic reticulum, and (d) protein of interest.

We first identified the nucleus region in the DAPI nuclear staining images. Second, we identified the entire cell region using the antibody-based microtubule staining images. Third, we identified the cytoplasm region by subtracting the nucleus region from the whole-cell region. In this way, we found 19,495 cells in 2,538 microscopic images. For each cell, we calculated a 1,014-dimensional feature vector including 48 area-shape features, 60 intensity features, 18 location features, and 212 texture features. We then constructed pipelines and analyzed these cells using the Cell Profiler [25]. We selected seven labels, namely, "Euglena," "Mitochondria," "Nucleoli," "Cytosol," "Nucleus," "Aggresome," "Cytokinetic bridge," and "Focal adhesion sites," and individually assigned them to seven datasets, each consisting of 19,495 three-dimensional entries with two-, three-, or four-dimensional vectors. The condition positive rate in each dataset ranged from approximately 4.9% to 54.1%. All skyline entries in each dataset were tagged with each label (true condition). Thus, the skyline computation can identify the required entries in these biologically important datasets, confirming the suitability of these datasets for the performance evaluation. To create datasets

Table 5.4: Synthetic dataset parameters

Cardinality	10,000 to 320,000
Dimensionality	2, 3, 4, $\cdots$, 25
Spatial distribution	Anti-correlated ($r = -0.9$), Independent ($r = -0.0$), Correlated ($r = +0.9$), Concealed
Temporal distribution	Independent, Concealed
Spatial value	[0, 65535]
Temporal value	up to 49999
Lifetime	1,000 to 1999

for continuous skyline computation, we allocated uniformly distributed lifetimes to the entries. The maximum time step was set to 10,000.

Euglena Cell Images

We also used low-resolution cell images taken under a fluorescence microscope in flow cytometry (our Serendipiter [52]). The cells were *Euglena* cells, which are expected to be highly effective for biofuel production. The cells were captured in one transmitted light and two fluorescence channels (Figure 5.13(a)). The images show lipid and chlorophyll staining of cells. The cell positions in the images are synchronized among the channels, and there is one *Euglena* cell per image. Each cell was tagged with one of two labels depending on the cell's environment: nitrogen deficient ("N-def") or nitrogen sufficient ("N-suf"). High-efficiency biofuel production will require the extraction and cultivation of *Euglena* cells that are lipid-rich in nitrogen deficient environments.

The goals of Serendipiter are simultaneous high image quality and throughput of cell flow. As the Serendipiter microscopes are still being developed, the images are noisy with low resolution, and difficult to analyze in Cell Profiler. Instead, we analyzed the images ourselves. First, we identified the *Euglena* cell region and constructed mask images from the transmitted light images (Figure 5.13(b)). In this way, we found 1,072 *Euglena* cells in 1,072 microscopic images. Second, we calculated a 47-dimensional feature vector for each cell, including 23 area-shape features from the mask images and 24 intensity features from the

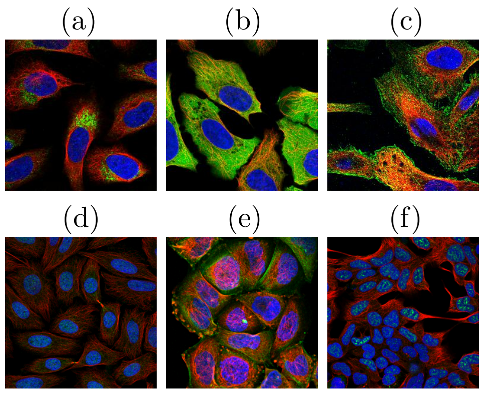


Figure 5.12: Examples of the tagged cell image: (a) "Mitochondria," (b) "Cytosol," (c) "Plasma membrane," (d) "Cytokinetic bridge," (e) "Focal adhesion sites," and (f) "Nuclear speckles." In (a–c), the labels indicate the protein location; in (d–f), they indicate rare events in the cells.

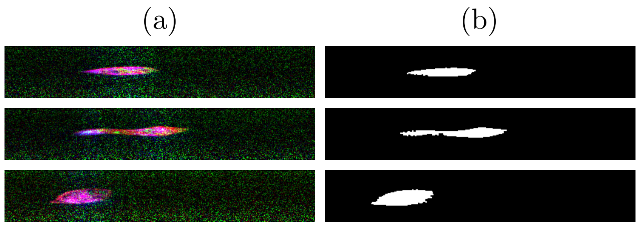


Figure 5.13: (a) Images of *Euglena* cells under a fluorescence microscope in flow cytometry (Serendipiter). Each image is captured in three channels: transmitted light, lipid fluorescence, and chlorophyll fluorescence. (b) Mask images of the *Euglena* cell generated from the transmitted light channel.

three-channel images. We then selected the features “IntensityMinumum” and “IntensityMean” in the transmitted light channel and “IntensityStandardDeviation” in the chlorophyll channel. Finally, we obtained the EUGLENA dataset consisting of 1,072 three-dimensional entries. Approximately 7.4% of the entries in EUGLENA, and all of the skyline entries in EUGLENA, were tagged with “N-def.” That is, the skyline computation extracted the rare “N-def” entries with 100% precision. To create datasets for continuous skyline computation, we allocated uniformly distributed lifetimes to the entries.

Blood Cell Images

We also generated datasets of blood cell measurements. The blood cell images contained aggregated platelets, single platelets, and white blood cells. In total, 4992 cell images were captured by optofluidic time-stretch quantitative phase microscopy and the cell features were extracted from the images by Cell Profiler [25]. Cell Profiler is an open-source software that analyzes cell images for biologists. The cell information was provided by the authors of [64]. Figure 5.14 shows the heat map of the pairwise correlation matrix of 41 features (14 area-shape features, 13 intensity features, and 14 texture features) extracted from the blood cells. Ten of the area-shape features were strongly correlated with the “Area” feature (enclosed in the upper-left black box in Figure 5.14). The intensity and texture features were also strongly correlated. As poorly correlated features in datasets are preferred for experiments, we eliminated the strongly correlated features and selected four weakly correlated features: the “Area” feature, one of the three area-shape features (“Compactness,” “Eccentricity,” or “Orientation”), one of the 13 intensity features, and one of the 14 texture features. Thereby, we created 546 types of datasets, each containing 4992 four-dimensional vectors. Figure 5.15 shows the cumulative distribution function of the maximum skyline ratio in these datasets. The skyline ratio was below 7% in all datasets, and below 1% in 20% of the datasets. Therefore, these datasets were pragmatic.

5.7.3 Experimental Setup

We implemented the BJR-tree on a single Intel-based computer. Although the processor was multi-cored, we executed and benchmarked all implementations in a single thread. The specifications are detailed in Table 5.5.

Table 5.5: Specifications of our platform

Processor	Intel Core i7 6700K @ 4.0 GHz
Motherboard	ASUS Z170M-PLUS
Main Memory	4 CORSAIR 8GB DDR4-2133
Operating System	CentOS Linux 7.1.1503 x86_64
C/C++ Compiler	GNU GCC 4.8.3

Our algorithm was competed against three existing algorithms. **Continuous BNL (cBNL)** and **Continuous BBS (cBBS)** extend the static skyline computation algorithms BNL [18] and BBS [98], respectively, to continuous skyline computation. **LookOut** is an algorithm developed for continuous skyline computation [92]. All algorithms were implemented in C/C++. In all implementations, we employed the same interface functions to read the input datasets

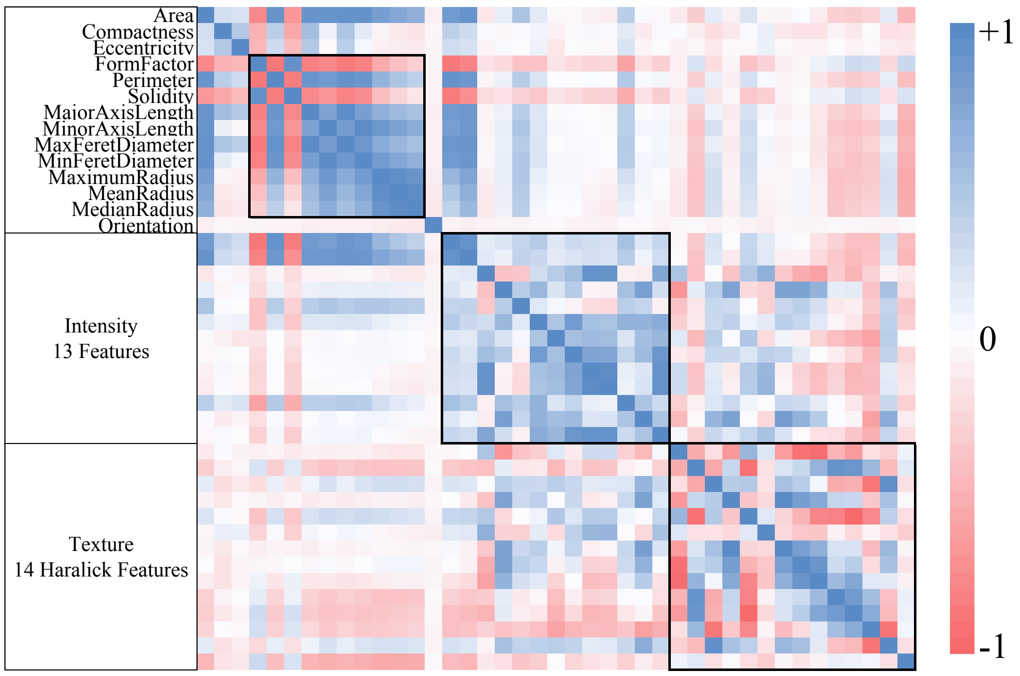


Figure 5.14: Heat map displaying the correlations between 41 features obtained by Cell Profiler (blue and red indicate positive and negative correlations, respectively). The black boxes enclose the strongly correlated feature groups.

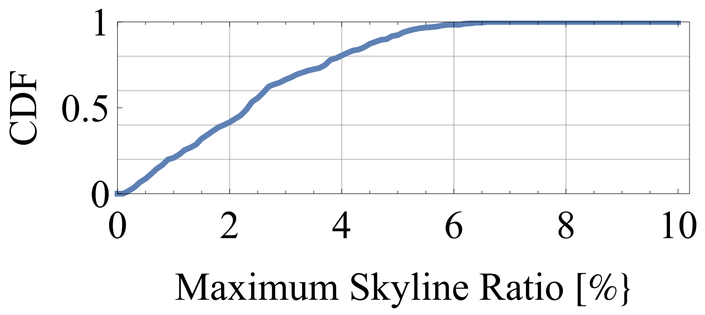


Figure 5.15: Cumulative distribution function of the maximum skyline ratio in 546 real datasets. Each dataset contains 4992 four-dimensional vectors.

and output the skylines, and the same comparison functions for the dominance relations. The cBBS and LookOut algorithms were implemented on quadtrees, which accommodate a maximum of q points in their leaf regions.

5.7.4 Tree Analysis

In Algorithm 3, we included a depth parameter L that controls the timing of the lazy evaluation and balances the tree. The relationship between execution time and L for different datasets is shown in Figure 5.16. The cases of $L \to \inf$ and $L = 1$ are referred to as the *no-lazy* and *full-lazy* policies, respectively. The best L depended on the dataset, and equaled 1 in the two-dimensional datasets. L exerted little influence on the execution time in the other datasets. Therefore, we adopted the full-lazy policy.

We now show the features of the constructed quadtrees and BJR-trees. Figure 5.17 shows the relationship between the cardinality and heights of the quadtrees

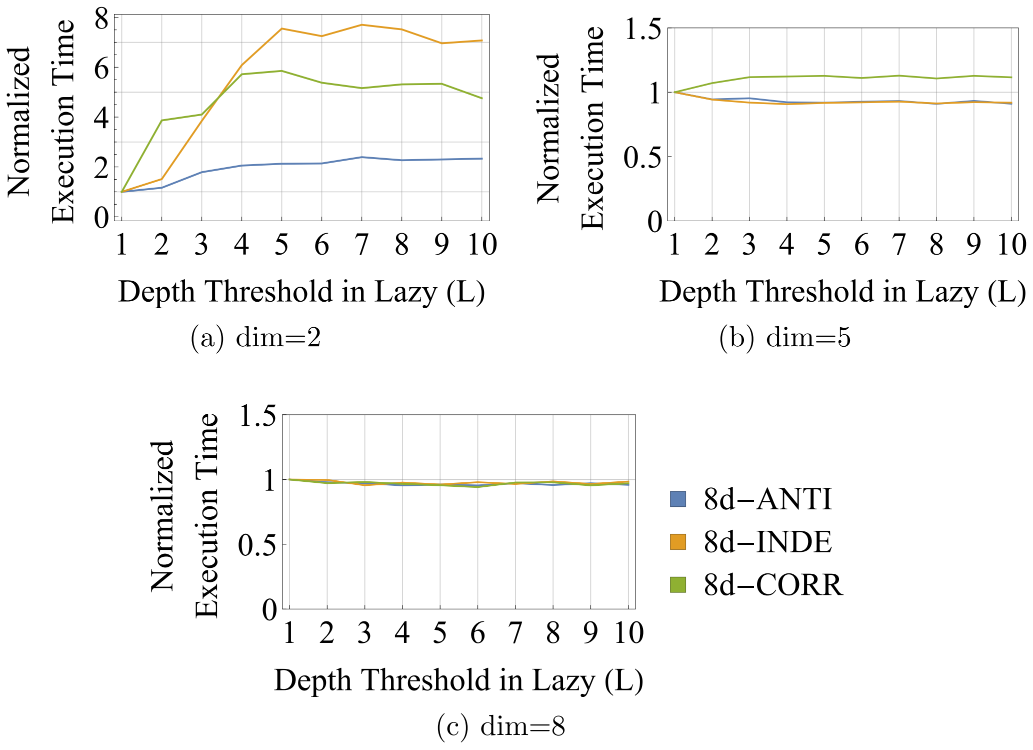
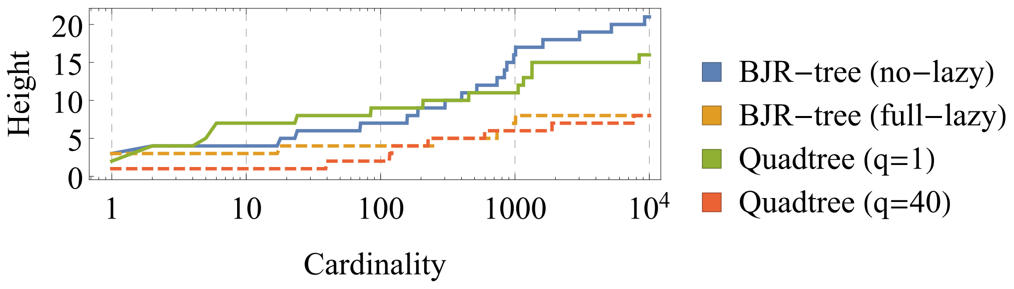


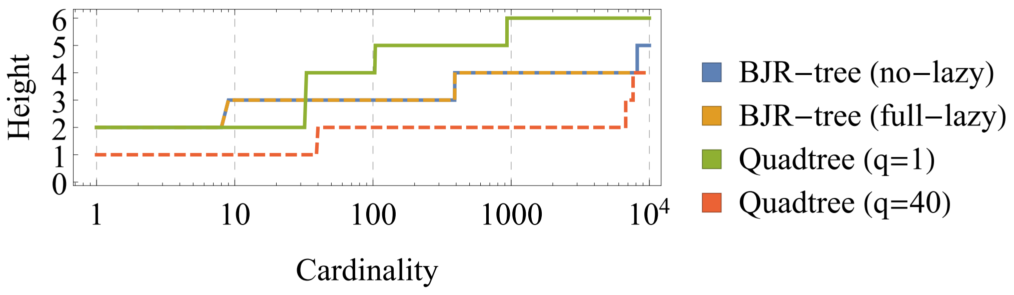
Figure 5.16: Execution time (normalized such that $y = 1$ when $L = 1$) versus depth threshold L in the BJR-tree algorithm for different datasets with 320k cardinality: ANTI (blue), INDE (orange), CORR (green).

and BJR-trees in independent datasets with two and eight dimensions. The trees were of four types; a BJR-tree with a full-lazy policy (blue), a BJR-tree with a no-lazy policy (yellow), a quadtree with $q = 1$ (green), and a quadtree with $q = 40$ (red). The large-q quadtree was shallower than the small-q quadtree, but the local skylines needed to be computed in each leaf region. Meanwhile, the height of the BJR-tree was lowered by the lazy evaluation. As shown in Figure 5.17 (bottom), in the eight-dimensional dataset, the heights of all four trees were very similar. The number of spatial partitions in the spatial indexing tree exponentially increased with increasing number of dimensions. Establishing a dominated relation between two independent points is more difficult in a high-dimensional space than in a low-dimensional space, which explains the lack of height difference in the eight-dimensional dataset. Therefore, the tree structure largely depends on the number of dimensions. It is valuable for low-dimensional datasets to lower the height of the trees.

Panel (a) of Figure 5.18 shows the skyline potential of the two-dimensional vectors, and panel (b) displays the depths of the vertices corresponding to each vector in the BJR-tree expressing the same two-dimensional vectors. In each distribution, the BJR-tree expressed the skyline potential of each vector as a vertex depth in the tree. Moreover, the skyline potential and vertex depth were highly consistent near the origin (in the high skyline potential area). The BJR-tree stores the skyline potentials as the vertex positions.



(a) dim=2



(b) dim=8

Figure 5.17: Heights of the BJR-trees and quadtrees constructed from up to 10,000 points. The green quadtree has a single point per leaf, and the red quadtree accommodates up to 40 points per leaf.

5.7.5 Results for Synthetic Datasets

We evaluated the execution times of our proposed algorithm and other existing continuous skyline computation algorithms on the synthetic datasets, excluding the times consumed by initialization, memory-loading of the points from the input file, and storing the skylines from the array to an output file.

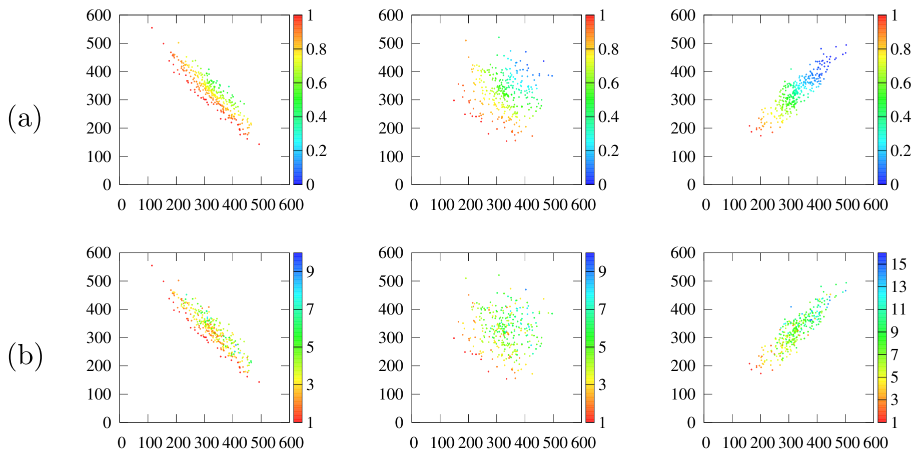


Figure 5.18: (a) Skyline potentials of the points and (b) BJR-tree depth positions of the corresponding vertices (left: ANTI, center: INDE, and right: CORR)

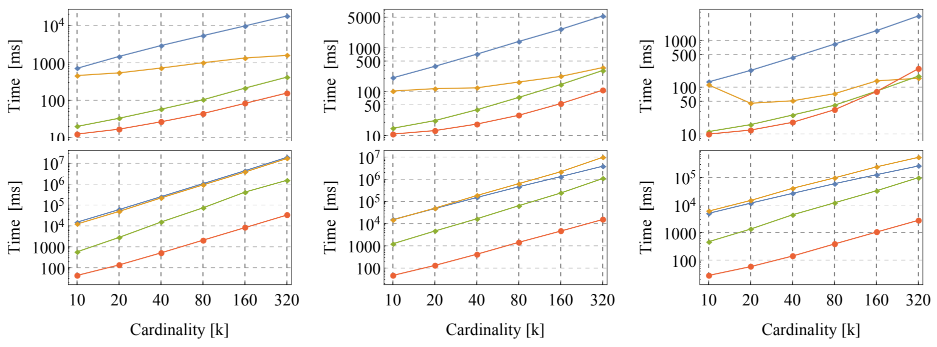


Figure 5.19: Execution times versus dataset cardinality (left: ANTI, center: INDE, and right: CORR/upper: 2D and lower: 8D) in different algorithms: cBNL (blue), cBBS (yellow), LookOut (green), and BJR-tree (red). All vertical axes denote execution time in milliseconds.

Cardinality

The execution times are plotted as functions of cardinality in Figure 5.19. In this comparison, the dimensions are two or eight. On almost all datasets, our BJR-tree outperformed the existing algorithms; the only exception was the CORR dataset with 320,000 two-dimensional points, on which cBBS and Lookout performed 1.6 times faster than BJR-tree. On the INDE dataset with 320,000 eight-dimensional points, BJR-tree executed more than 70 times faster than LookOut.

Dimensionality

Figure 5.20 shows the relationships between the number of dimensions and the execution time at cardinalities of 10,000 and 100,000. The execution time only slightly varied with number of dimensions because cBNL does not use a tree structure and BJR-tree is dimensionally independent. The execution time of BJR-tree mostly depended on the skyline ratio (see Figure 5.5). At lower dimensions, the execution times of cBBS and LookOut also depended mainly on the skyline ratio, but a dimensional effect emerged at up to 14 dimensions.

On the CORR dataset with 25-dimensional points, the BJR-tree outperformed LookOut by a factor of approximately 570. For example, the skyline ratio of the eight-dimensional CORR was approximately 0.3. On the INDE dataset with 100,000 eight-dimensional points, BJR-tree was approximately 43 times faster than LookOut.

Concealed-skyline Dataset

Processing CONC datasets is time-intensive because the ejection operations incur a heavy computational load. Our ND-cache resolves this problem. Figure 5.21 compares the execution times of BJR-trees with and without an ND-cache. The x-axis indicates the number of anti-correlated points (set (A) in Figure 5.10). The anti-correlated points were dominated by 200 temporal clusters (set (B) in Figure 5.10), each with five points. The ND-cache reduced the CONC execution time without significantly increasing the execution time of the INDE datasets. In the CORR dataset with eight-dimensional points, the BJR-tree executed approximately 2.3 times faster with the ND-cache than without the cache. The ND-cache achieved a hit ratio of 50–70% in five- and eight-dimensional CONC datasets (see Figure 5.22).

5.7.6 Results for Real-world Cell Measurement Datasets

Finally, we competed our proposed algorithm against three existing algorithms for continuous skyline computation on real-world datasets. Figure 5.23 shows the speed-up ratios calculated from the execution times of the existing method implementations (cBNL, cBBS, and LookOut), and our BJR-tree implementation for eight datasets. On average, the BJR-tree reduced the execution time by 97.7% from that of cBNL, 94.4% from that of cBBS, and 63.9% from that of LookOut. On the Euglena dataset and Mitochondria datasets, BJR-tree executed 2.1 and 9.7 times faster than LookOut, respectively.

Figure 5.24 shows the execution times of the algorithms on real-world datasets of blood cells. Plotted are the cumulative distribution functions of 546 blood cell datasets. In real-world datasets, the algorithm performance depends on the feature selection. The average execution times of BJR-tree and LookOut were 14.2 and 33.8 microseconds, respectively. In almost all datasets, the execution time of BJR-tree was below 20 milliseconds. In contrast, LookOut required over 100 milliseconds in 17 datasets, and cBNL and cBBS required over one second in almost all datasets. Therefore, in both synthesized and real-world datasets, the execution time of BJR-tree surpasses the execution times of existing algorithms.

5.8 Discussion

In this section, we discuss performances of skyline operation and other entry extraction methods on real-world applications. We also describe another application of the proposed BJR-tree.

Gating is a simple method for extracting the target cell entries in cell information analysis. This statistical method is widely used in cell analysis. After selecting two or more cell measurement metrics, the measurement data are projected to a space with two or more dimensions. The minimum and maximum thresholds in each dimension are then set, and the subspace enclosed by the thresholds is obtained. By filtering the entries outside the enclosed subspace, the purity of the measurement data is improved. However, gating is problematic for two reasons. First, the metrics must be appropriately selected to isolate sets of

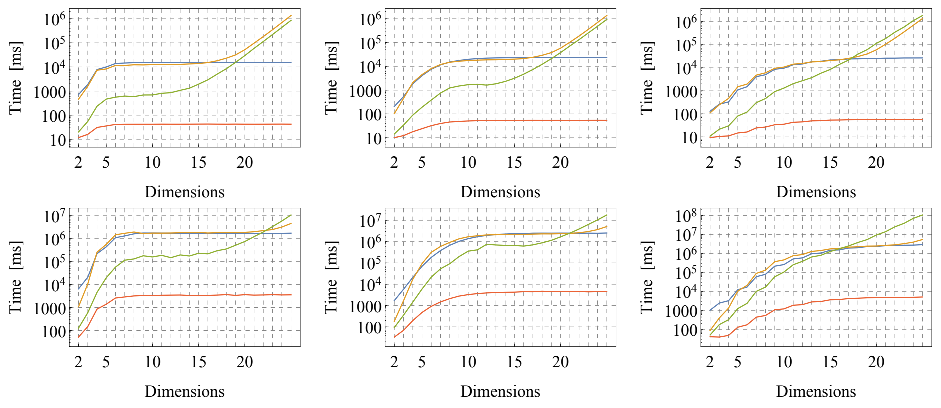


Figure 5.20: Execution time versus number of dimensions (left: ANTI, center: INDE, and right: CORR/upper: 10k and lower: 100k) in different algorithms: cBNL (blue), cBBS (yellow), LookOut (green), and BJR-tree (red). All vertical axes denote execution time in milliseconds.

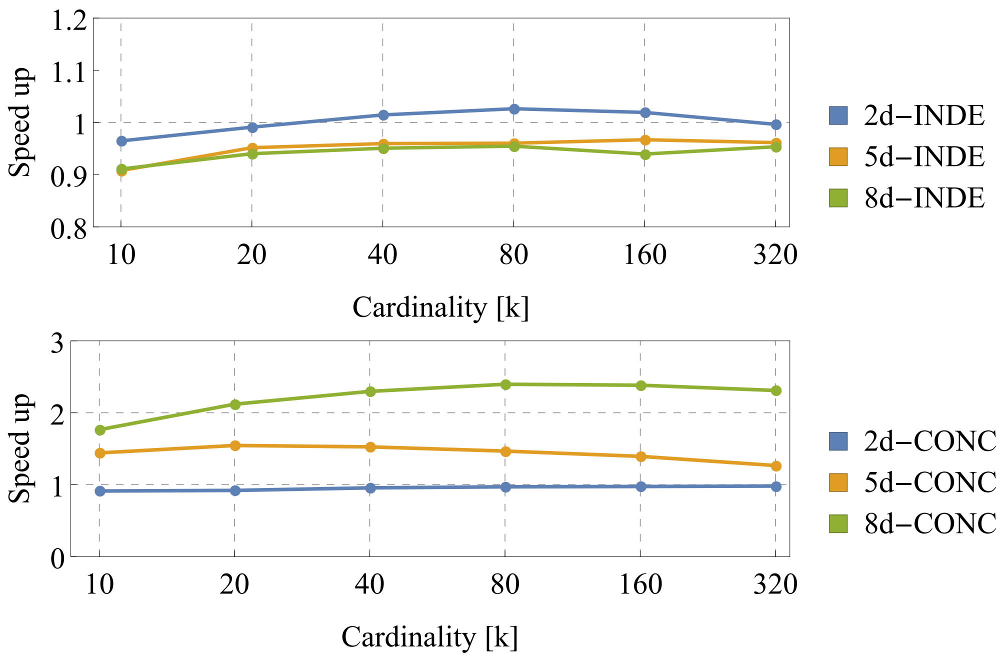


Figure 5.21: Speed-up ratio of BJR-tree with an ND-cache

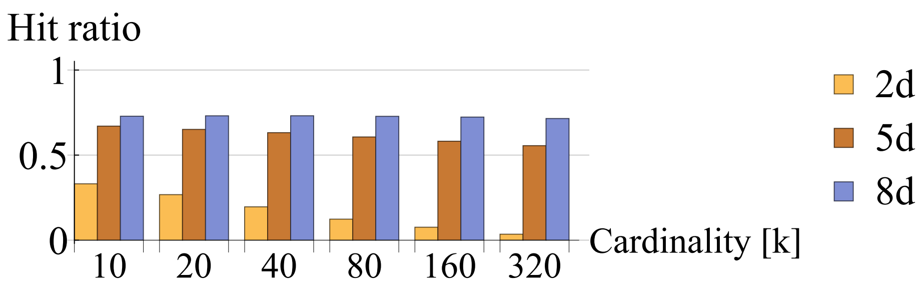


Figure 5.22: Hit ratios of the ND-cache in CONC

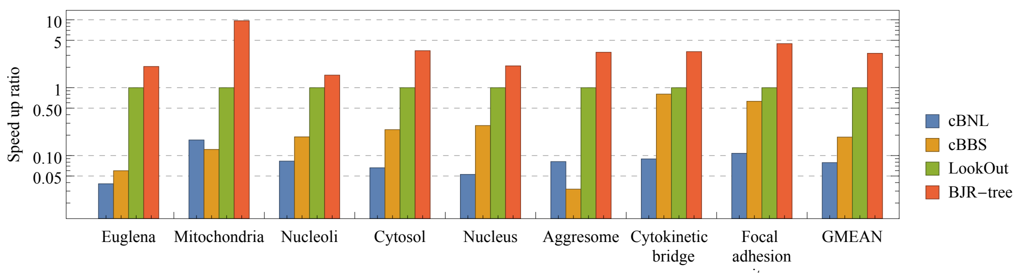


Figure 5.23: Speed-up ratios in the existing methods (cBNL, cBBS, LookOut), and our BJR-tree implementation for eight datasets. In each dataset, the speed-up ratio (y-axis) of the four methods are normalized such that the LookOut value is 1.

cells. Unless we know the metrics that can divide cell clusters, gating is inapplicable to cell identification. Another problem is the low precision of gating. Gating extracts the large-scale populations in low-dimensional spaces. The aim of the gating method is to improve the purity of a relatively large-scale population. Therefore, the gating method is not suitable to improve the precision of extracting extremely rare cells. For example, in the datasets of the CYTO 2017 Cell Image Analysis Challenge, the proteins in cells tagged with “Cytosol” and “Endoplasmic reticulum” occupy intracellular regions outside the cell nucleus. Therefore, in the cell images, the channel intensity of the protein is high in the cytosol and endoplasmic reticulum, respectively. Meanwhile, the proteins in cells tagged with “Nuclear membrane” reside at the boundary between the cell nucleus and the surrounding region. In these cell images, the channel intensity of the protein is high at the outer perimeter of the cell nucleus. Many tagged cells can be extracted by combining metrics that appropriately describe the cell characteristics. However, this rough approach is unsuitable for extracting cells with rare labels. Thus, gating is an ineffective method for extracting rare cells in the Serendipiter.

Clustering methods, including gating, are employed in the extraction of large-scale populations. In the data analysis domains, researchers have applied outlier detection methods to rare-entry extraction. Outlier detection is also termed anomaly detection. In several applications of Serendipiter, obtaining large amounts of measured samples tagged with a rare label and creating a model from the extracted data are difficult tasks. To alleviate these difficulties, we often employ unsupervised machine learning in the Serendipiter. Typical unsupervised outlier detection algorithms are based on the local outlier factor (LOF) [19]. In addition, a method based on the one-class support vector machine (OC-SVM) [106] is also used. In high-dimensional spaces, the entries are very sparsely distributed owing to the curse of dimensionality, so the outliers are difficult to detect. Consequently, the LOF-based and other distance-based outlier detection methods are unsuitable for high-precision rare-entry extraction in high-dimensional space. This difficulty in high-dimensional space has been overcome by the angle-based outlier detection (ABOD) method [78]. ABOD and other angle-based methods assume that the variance of angles among vectors of an outlier entry to other entries is low even in high-dimensional space. Table 5.6 lists the prevalence and precision performances of the existing outlier detection methods on the CYTO 2017 datasets. We used scikit-learn Python library [100]. We have listed the true contamination value of the rare label for each outlier detection method. Neither outlier detection method could achieve a high precision score.

Outlier detection methods are effective because they require no pre-selection

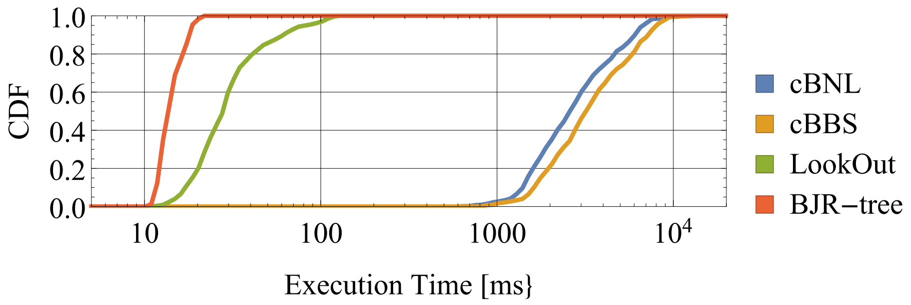


Figure 5.24: Cumulative distribution function of the execution times in real-world blood cell datasets

of the dimension. However, in real-time operations, we have to reduce the number of dimensions and implement a high-speed identification algorithm. The cells to be discovered are neither outliers nor anomaly cells, but are regular cells with extreme characteristics or rare labels. Therefore, outlier detection methods are inappropriate for our purpose. Instead, our problem can potentially be solved by skyline computation, which was originally proposed for extracting interesting entries. We searched appropriate metrics for extraction with the skyline computation on the CYTO 2017 datasets and obtained sets of metrics that extract 2–11 true positive cells from populations with prevalences from 0.02 to 0.1. Rare-labeled cells on the Serendipiter are expected to be poorly identified by unsupervised machine learning with no dimensional reduction. Furthermore, the appropriateness of an identification algorithm depends on the cell type. In this study, we discussed the effectiveness of the combination of skyline computation and appropriate dimension reduction. We also showed the robustness of our algorithm to the low-latency real-time query processing of continuous skyline computation.

The BJR-tree is robust in high-dimensional spaces. As the number of dimensions increases, the skyline ratio approaches 1, and we cannot obtain an effective subset of entries by using the skyline computation. Even in datasets with same cardinality, the skyline ratio varies greatly because of correlation coefficient of the entry distribution. The skyline ratio of a dataset with positive or negative correlations is higher and lower, respectively, than that of an uncorrelated dataset. Nonetheless, as shown in Figure 5.5, in synthetic datasets with 25 or more dimensions and practical correlation values, almost all entries join the skyline. In this study, we showed that BJR-tree operates faster than the existing algorithms in datasets with up to 25 dimensions. In both the experimental results and theoretical analysis of computational complexity, it is showed that the execution time of BJR-tree in high-dimensional datasets strongly depended on the number of skyline entries. Thus, even in real-world datasets, the number of skyline entries and the execution time of BJR-tree can be controlled by adjusting the window size of the computational target entries.

The BJR-tree operates efficiently in continuous skyline computation. We now present another application other than continuous skyline computation of the BJR-tree. Because the skyline ratio is uniquely determined relative to a dataset. Therefore, skyline computation cannot be used for extracting an arbitrarily sized subset of entries. In addition, the skyline ratio largely depends on the distribution of entries in a multi-dimensional space. Thus, we cannot stably extract an entry

Table 5.6: Prevalence and precision performances of existing outlier detection methods: One-Class Support Vector Machine (OC-SVM), Local Outlier Factor (LOF), Angle-based Outlier Detection (ABOD). We created a dataset from 19495 cell information with 1035 attributes computed by the Cell Profiler [25].

Rare label	Prevalence	OC-SVM	LOF	ABOD
Aggresome	0.049	0.023	0.067	0.072
Cytokinetic bridge	0.112	0.426	0.133	0.125
Focal adhesion sites	0.126	0.119	0.212	0.225
Microtubule organizing center	0.003	0.006	0.000	0.000
Nuclear speckles	0.005	0.009	0.000	0.011
Nucleoli fibrillar center	0.007	0.013	0.008	0.007

subset of the expected size. Previously, entries surrounding the skyline have been extracted by methods such as skyband [99] and a distance-based method [115], which use geometric information (e.g., the norm). As mentioned above, when the number of dimensions is high, the entries are sparsely distributed and cannot be successfully extracted. The concept of k-dominant skyline [26] also provides another extraction method with dimension selection. However, it is necessary to compute a skyline in each combination of dimensions. It appreciably increases the computational time. To address this problem, we can employ the skyline potential concept. All entries in a dataset can be ranked based on a value of their skyline potential. Because the time complexity of obtaining the exact values of skyline potential of all entries is $O(N^2)$, we can instead construct a BJR-tree and roughly rank the entries based on the depths of their corresponding nodes. Thus, only by a single construction of the BJR-tree, we can obtain entry subsets of various sizes. This application exemplifies how entry extraction methods based on the BJR-tree can compensate the weak points of skyline operation.

5.9 LSCA: Low-Latency Skyline Computation Accelerator

5.9.1 Basic Idea of the LSCA

We propose the LSCA, a hardware algorithm for high-speed continuous skyline computation and implement it on an FPGA. We delay non-urgent dominance relation comparisons under the concept of the BJR-tree algorithm. However, many comparisons related to deactivation of skyline entries cannot be delayed. These comparisons are the main part of the BJR-tree update process. Since there is no dependency between these operations, LSCA parallelizes them. In the BJR-tree algorithm, we are free to choose a strategy for delaying comparison operations. Preliminary experiments show that we should maximally delay operations in most kinds of datasets. Therefore, The LSCA adopts a maximally-lazy strategy where only one level of the BJR-tree is traversed. Because the dominance relation comparisons are executed in parallel, the time complexity of vertex injection is $O(S)$ and that of vertex ejection is $O(C)$. In the injection process, the time complexity when the newly-activated entry belongs to the skyline is $O(S)$, but becomes $O(1)$ when the entry does not belong to the skyline. To avoid complication of circuits, we employ the JR-tree instead of the BJR-tree; thus the LSCA is based on the JR-tree. Figure 5.25 shows the logic diagram of the LSCA, and we will now describe the algorithms used for each of its part in detail.

5.9.2 Tree-Structure Memory

Figure 5.26 shows an example JR-tree and the corresponding tree-structure memory for LSCA. The tree traversal and update (i.e., vertex addition, deletion, and move) are implemented as read and write operation of this memory. In this memory, the children of an entry are stored in a linked list, and the entries are indexed by their ID numbers. Address k in the tree structure stores the following IDs: (1) the ID of the parent of entry k, (2) the ID of the next child after entry k in the linked list of children, and (3) the ID of the first child of entry k, namely the head node of the linked list. When entry k belongs to the skyline, its parent is the root of the tree. When entry k is the last child, which is the tail node in the linked list, the next child is null. When entry k has no children, the first child is null. For example, in the tree in the left of Figure 5.26, the parent of vertex B is vertex A, the next child for B is C, and the first child of B is D. By using this

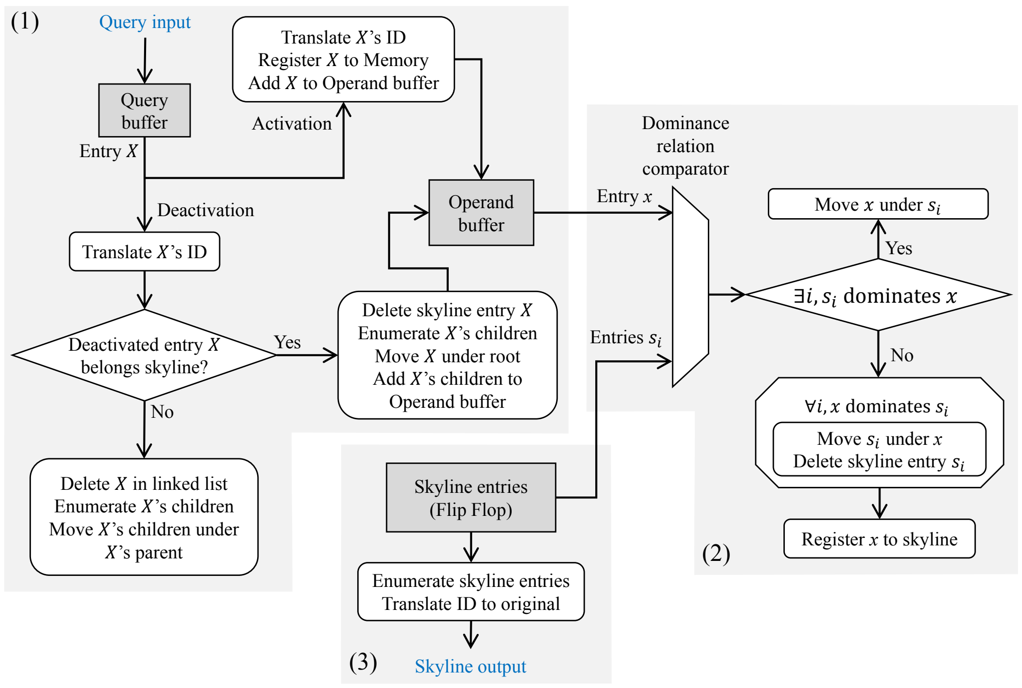
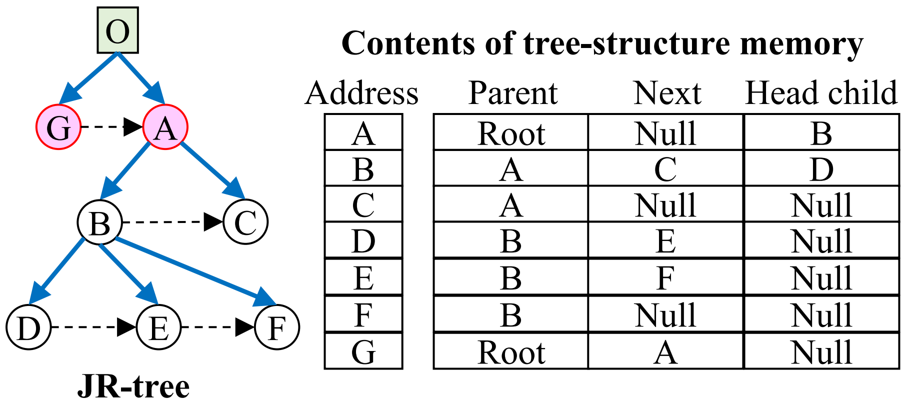


Figure 5.25: Logic diagram for the LSCA, which mainly consists of (1) query dispatcher, (2) skyline comparator, and (3) skyline enumerator.



Address	Parent	Next	Head child
A	Root	Null	B
B	A	C	D
C	A	Null	Null
D	B	E	Null
E	B	F	Null
F	B	Null	Null
G	Root	A	Null

Figure 5.26: Example contents of the tree-structure memory. A JR-tree (left) is expressed in the memory such as a table (right).

structure, the JR-tree can be fully built, traversed, and updated efficiently.

The tree-structure memory is accessed by several different parts of the control logic, so these read and write requests are properly serialized. In between the activation and deactivation queries handled by real-time streaming applications, the bus of the tree-structure memory and the dominance relation comparator buses are not used. The LSCA uses this time to eagerly re-evaluate non-skyline entries to potentially reduce the number of children of each parent. This background evaluation makes it possible to speed up ejection operations, whose processing time is proportional to the number of the children.

5.9.3 Query Dispatch

Figure 5.25(1) shows the logic of query dispatcher. First, queries are stored in the query first-in, first-out (FIFO) buffer. Activation and deactivation queries are processed differently. For an activation query, a vector representing the activation entry is stored in the ID/vector memory, and the corresponding vertex information is stored in the tree-structure memory. For a deactivation query, the process varies depending on whether or not the deactivated entry belongs to the skyline.

Let X be the deleted vertex. If X does not belong to the skyline, the LSCA first traverses the linked list of children of X's parent to find the vertices adjacent to X in the tree-structure memory. After X has been found, it is removed from the linked list. The LSCA also enumerates all of X's children and moves them under X's parent, resulting in X's children becoming children of X's parent. These operations can be executed in parallel as long as there are no tree-structure memory access conflicts.

If X does belong to the skyline, it is first removed from the set of skyline entries. Then, to check whether X's children belong to the skyline, the children are stored in the FIFO buffer used for adding a new entry, i.e., the operand buffer. Since entry addition requests derived from skyline entry ejection are merged in the same buffer with the request of new entry injection, a dominance relation comparator unit is utilized more efficiently.

5.9.4 Dominance Relation Comparison

Figure 5.25(2) shows the logic of dominance relation comparator. The LSCA stores the vectors representing the skyline entries in the flip-flops so that the dominance relation comparisons can be parallelized. Entries stored in the operand buffer have to be checked whether or not they join the skyline. Each entry x is compared to all existing skyline entries in a fixed number of clock cycles as shown in Figure 5.27. Based on these results, when x is dominated by a skyline entry, the tree-structure memory is updated to make x a child of that skyline entry. When no skyline entries dominate x, x becomes a skyline entry and all other skyline entries dominated by x are removed from the skyline. In this case, both the tree-structure memory and the skyline entry flip-flops are updated.

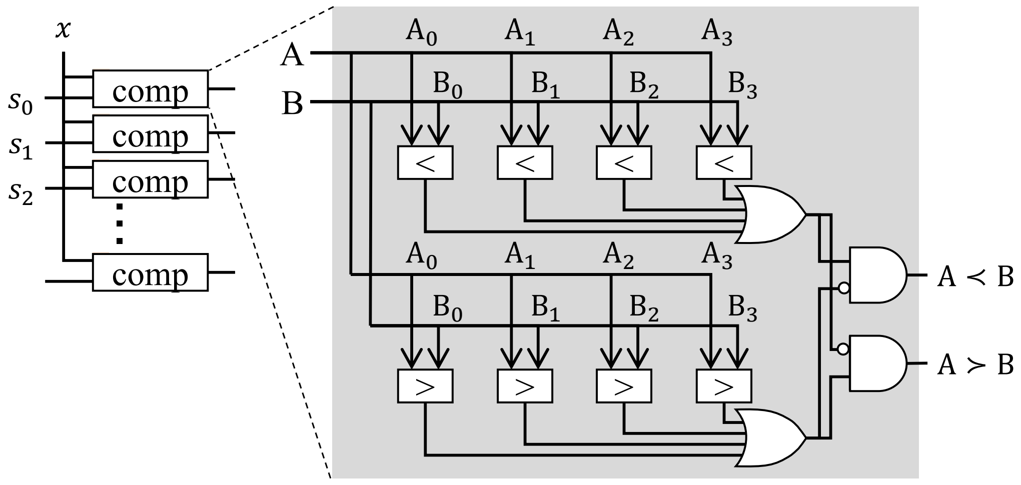


Figure 5.27: Logic diagram of dominance relation comparators for dataset with four-dimensional vectors. Each dominance relation comparator includes "less-than" and "greater-than" comparators.

5.9.5 Skyline Enumeration

With a JR-tree, we do not need to traverse the tree to output the skyline entries. Instead, we just have to enumerate the root's children. In the LSCA, since the skyline is stored in the flip-flops, no RAM access occurs, Figure 5.25(3) shows the logic of skyline enumerator. so the use of flip-flops for the skyline entries leads to faster calculations. On the other hand, the flip-flops consume a lot of FPGA slice resources. However, since the goal of skyline computation is to find rare and interesting entries, in real-world datasets, the number of skyline entries tends to be small. Therefore, it is feasible to design the logic to limit the number of skyline entries.

The ID numbers are never reused. If we store vectors in the ID/vector memory using the original entries' ID numbers, the memory used for deactivated entries cannot be reused. In the LSCA, we use a hash table and a shorter ID number that is called an SID. The hash table is updated whenever an entry is activated or deactivated. When outputting the skyline entries, the SID numbers are converted back to the original ID numbers using the hash table. First, the LSCA calculates a new SID number corresponding to the entry for an activation query using the FNV-1 algorithm. If a collision occurs when registering the calculated SID entry in the hash table, a new address is found using *open addressing* strategy with *linear probing*. In the main LSCA logic, the SID numbers are used as the entries' ID numbers.

5.9.6 Delayed JR-tree Reshaping

Continuous skyline queries consist of the activation or deactivation of entries. The JR-tree can adjust the tree shape it maintains by changing its lazy or eager evaluation strategy. When implementing the LSCA on an FPGA, a maximally lazy strategy is used to reduce the query processing time and simplify the logic design. However, this means that the numbers of children in the JR-tree tend to be larger. When a non-skyline entry is deactivated, dominance relation comparisons are not executed due to the maximally lazy strategy, resulting in the query processing time increasing little. On the other hand, when a skyline entry with children is deactivated, the dominance relation comparisons are executed due to the enforced eager strategy, which means that the execution time depends strongly on the number of children the ejected vertex has.

To solve the issue of the processing time for deactivation of skyline entry, we propose the new calculation method, delayed JR-tree reshaping (DJR). The DJR reshapes the JR-tree in the waiting time before the query arrives. In a region of the JR-tree which is built with the lazy strategy, it is not guaranteed that children of a parent do not dominate each other. For example, in Figure 5.28, since vertex P and Q are children of the root, it is guaranteed that they do not dominate each other. On the other hand, children of the P may dominate each other and children of the Q may dominate each other. The DJR re-traverses the JR-tree with eager strategy and calculates dominance relation and move dominated vertices under dominant vertex. The DJR reduces the number of children and query processing time of deactivation queries. The DJR is equivalent to bring forward the comparison operations that likely to occur in future deactivation of skyline entries.

In realistic continuous skyline applications, there will be some spare time between the completion of one query and the arrival of the next query. The DJR is optimized for continuous skyline computation and it improves logic element utilization efficiency. Traversal of the DJR can be aborted at any time, and it can use any spare time to traverse and reshape the entire JR-tree. Since the time taken to deactivate a skyline entry depends on the number of children, not the number of descendants, reshaping at depth two, corresponding to the root's grandchildren, has the greatest impact.

5.10 Experimental Evaluation

In this section, we show evaluation results for LSCA.

5.10.1 Experimental Setup

We compared our proposed LSCA method, which was implemented on an FPGA platform, with two existing software algorithms implemented on a computer. We implemented two LSCA versions, one with DJR and one without, and compared their results. For the LSCA, we used a Xilinx Ultrascale FPGA VCU108 Evaluation Kit equipped with a Virtex UltraScale XCVU095-2FFVA2104E FPGA. Although this platform includes DDR4 SDRAMs, the LSCA did not use it to achieve low-latency performance. We used the Xilinx Vivado System Edition 2016.4 software for synthesis, implementation, and simulation. We measured the total execution time from the start of each input query until all skyline entries had been output for each time step. We implemented the existing algorithms on a single Intel-based computer using the x86_64 architecture. The processor has multiple processing cores, but LookOut is not capable of multithreading. Although JR-tree can process multiple queries in parallel, it cannot parallelize a single query process. Thus JR-tree is not capable of multithreading in actual continuous skyline application. Therefore, we executed and benchmarked all implementations using a single thread. The computer's specifications are shown in Table 5.7. For fairness, since the FPGA was produced using a 20-nm semiconductor fabrication technology, we used an Intel Core i7 3770 processor based on

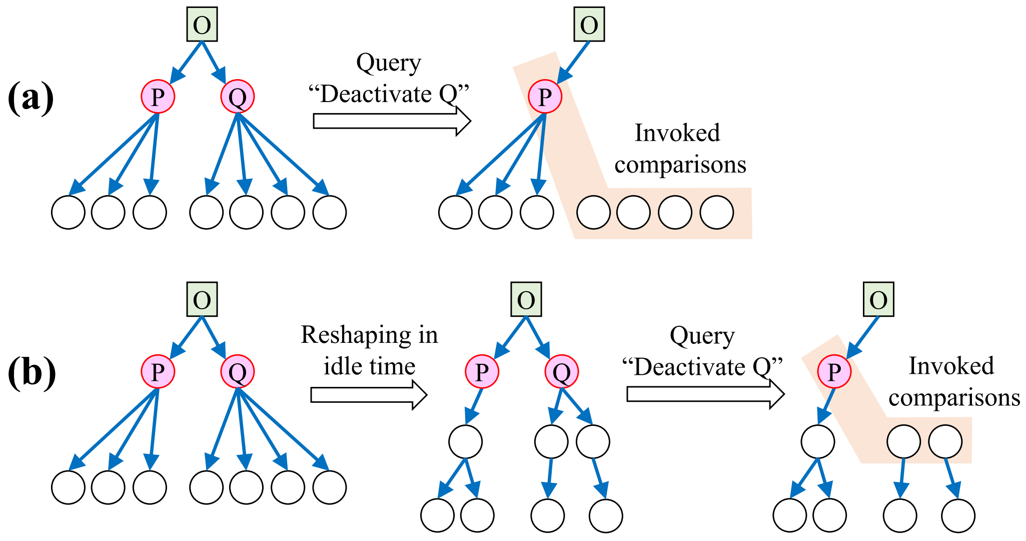


Figure 5.28: (a) Example of deactivation query processing without DJR. Because the vertex Q has four children, the deactivation process involves dominance relation comparisons between five entries. (b) Example of deactivation query processing with DJR. The JR-tree was reshaped before the query arrived, reducing Q's number of children to two. Now, the deactivation process only involves dominance relation comparisons between three entries.

a 22-nm semiconductor technology. We measured the execution time from the time when all queries had been written to the main memory to the time when all skyline entries had been written to the main memory, thus excluding the time taken for initialization, loading the entries from the input file into memory, and storing the array of skylines to an output file.

We compared our algorithm with two existing algorithms that have been previously proposed for continuous skyline computation: LookOut [92] and the JR-tree. All algorithms were implemented in C/C++ and employed the same interface functions to read the input datasets and output the skylines, as well as the same comparison functions for the dominance relations. The LookOut algorithm was implemented using a quadtree, which accommodates a maximum of q points in its leaf regions.

5.10.2 Performance

We evaluated our proposed algorithms against two existing software algorithms for continuous skyline computation using both synthetic and real-world datasets. Figure 5.29 shows the normalized execution times for the existing software implementations, LookOut and JR-tree, and our hardware implementations, LSCA with and without DJR, for ten datasets. LSCA without DJR reduced the latencies by 46% from LookOut and 30% from JR-tree on average. LSCA with DJR reduced the latencies by 80% from LookOut and 71% from JR-tree on average. For all synthetic datasets, LSCA with DJR was the fastest method. Particularly, for the five-dimensional CORR datasets, LSCA was 3.7x faster than the software JR-tree implementation. As the distribution's correlation coefficient increased, LSCA with DJR became faster, while LSCA became slower. This was because LSCA uses a maximally lazy strategy and does not calculate all the dominance relations, leading to increasing numbers of children in the JR-tree. The DJR deals with this weak point of the LSCA. JR-tree executes faster than LookOut algorithms on higher-dimensional datasets. For all the concealed datasets (CONC), LSCA with DJR was also the fastest method. For the two-dimensional CONC dataset, DJR made little difference to the execution time. If the two-dimensional entries shown in Figure 5.10 are evaluated with eager strategy, a deep JR-tree will be built and, once built, this structure is not changed by DJR re-traversal. For the EUGLENA real-world dataset, LSCA without DJR was 13x faster than the software JR-tree implementation, and DJR improved the performance by a further 2.7x faster than LSCA. For the CYTOSOL dataset, LSCA without DJR was slower than JR-tree, but LSCA with DJR was 1.7x faster than JR-tree.

Figure 5.30 shows the latency improvements when DJR was used for the synthetic datasets. For example, the bottom left subfigure shows that when activation and deactivation queries were processed using LSCA for the two-dimensional

Table 5.7: Computer platform specifications

Processor	Intel Core i7 3770 @ 3.4 GHz
Motherboard	Gigabyte Z77MX-D3H TH
Main Memory	2x Corsair 4GB DDR3-1333
Operating System	CentOS Linux 7.3.1611 x86_64
C/C++ Compiler	GNU GCC 4.8.5

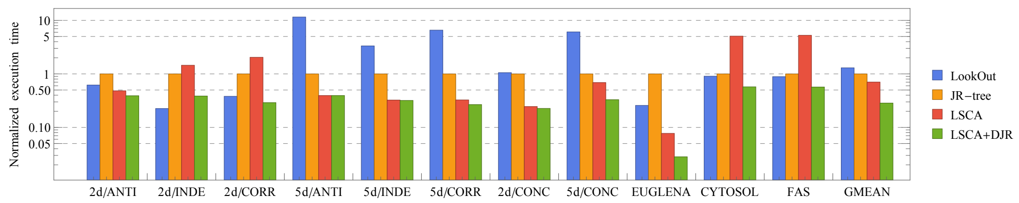


Figure 5.29: Comparison of the normalized execution times for the existing software implementations, LookOut and JR-tree, and our hardware implementations, LSCA with and without DJR, for ten types of datasets. For each dataset, the execution times (y-axis) of the four methods are normalized such that the value of the JR-tree is 1.

CORR dataset over 10,000 time steps, approximately 50% of the queries were processed within 500 clocks (2.5 microseconds). For the two-dimensional INDE and CORR datasets, DJR significantly improved the latency. On the other hand, the latency only slightly improved or the two-dimensional ANTI and five-dimensional CORR datasets, and hardly at all for the five-dimensional ANTI and INDE datasets. This was because, in datasets with high numbers of dimensions or small correlation coefficients, entries are less likely to dominate each other, decreasing the number of dominance relations, which means that DJR does not significantly change the shape of the JR-tree. Figure 5.31 shows the latency improvements due to DJR for the real-world datasets. For all real-world datasets, LSCA with DJR was the fastest method. For the EUGLENA dataset, there was no latency improvement for approximately 90% of the queries, although the latencies of the remaining 10% of queries improved. For the CYTOSOL dataset, there was no latency improvement for approximately 10% of the queries, but the latencies of the remaining 90% of queries improved. These results show that DJR can also improve latency in real-world datasets.

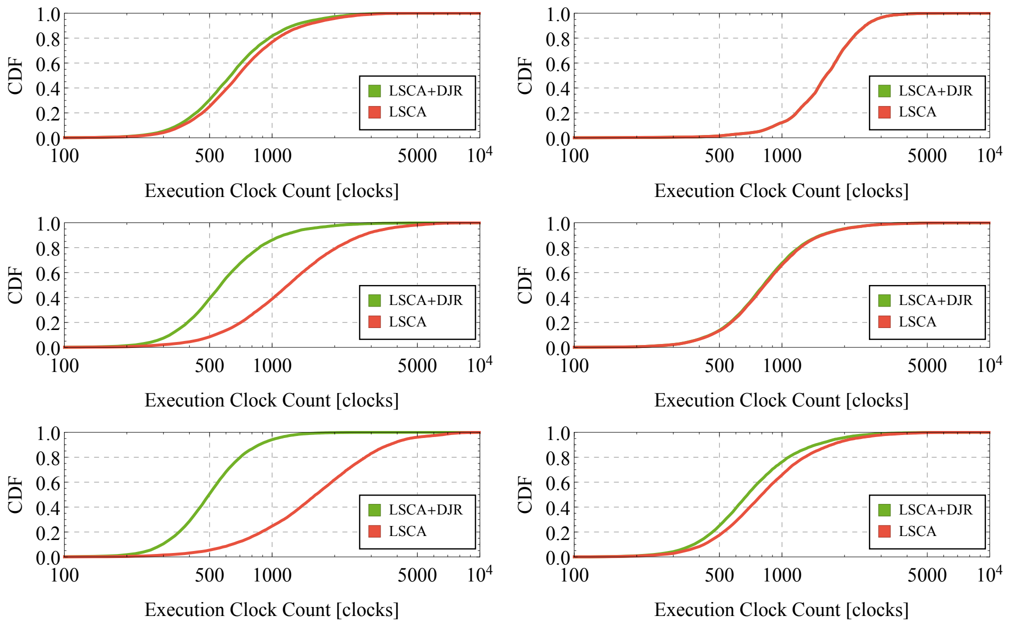
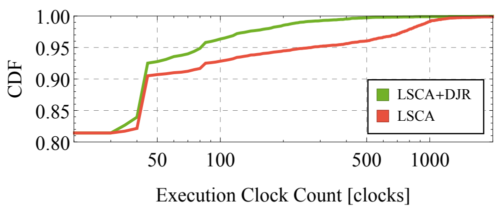


Figure 5.30: Execution clock counts for the ANTI (top), INDE (center), and CORR (bottom) synthetic datasets, plotted as cumulative distributions over 10,000 time steps. The entry vector dimensions were two (left) and five (right). The LSCA was implemented on an FPGA operating at 200 MHz.



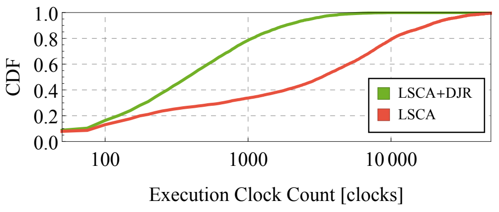


Figure 5.31: Execution clock counts for the EUGLENA (left) and CYTOSOL (right) real-world datasets, plotted as cumulative distributions over 10,000 time steps. The LSCA was implemented on an FPGA operating at 200 MHz.

5.10.3 Resource Consumption

The tree-structure memory is designed to be allocated to Block RAMs. The FIFO buffer that stores the queries and entries waiting to be processed, is designed to be allocated to FPGA hard macros. All the intellectual property cores that we used are built-in resources provided by Xilinx. The size of the LSCA logic is determined by three parameters: the maximum number of skyline entries, the maximum number of entries that can be active at the same time, and the entry vector dimension. Most of the allocated slices are used for comparison operation logic for the dominance relations. The slice (LUT and flip-flop) usage depends on the maximum number of skyline entries. Figure 5.32 shows resource consumption of the LSCA designed for different maximum number of skyline entries. The slice usage is proportional to the maximum number of skyline entries. Table 5.8 shows the slice usage of each unit in a logic that can handle five-dimensional datasets with a maximum of 256 skyline entries and a maximum of 4,096 active entries. Approximately 88% of used slice LUTs are allocated to the logic for the dominance relation comparator. Block RAM consumption mainly depends on the maximum number of simultaneously active entries and the number of entry dimensions. When processing larger numbers of active entries, we can instead use high-capacity RAM outside of the FPGA (e.g., SRAM or DRAM). The critical path is in the logic for managing the values of the skyline entry vectors.

Table 5.8: LSCA with DJR slice usage on a Xilinx Virtex Ultrascale XCVU095

	LUT	FF
Skyline	52459	38767
Comparator	46157 (88.0%)	25327 (65.3%)
Enumerator	1703 (3.24%)	3372 (8.70%)
DJR	305 (0.58%)	615 (1.59%)

5.10.4 Energy Consumption

One of the advantages of using an FPGA is its low energy consumption. Table 5.9 shows the power consumption of the logic circuits with three different sets of parameters. These results are derived from the power consumption reports produced by the Xilinx Vivado synthesis software, which gives the static power consumption of the device and the dynamic power consumption of the synthesized logic. The actual measured idle power consumption of the target

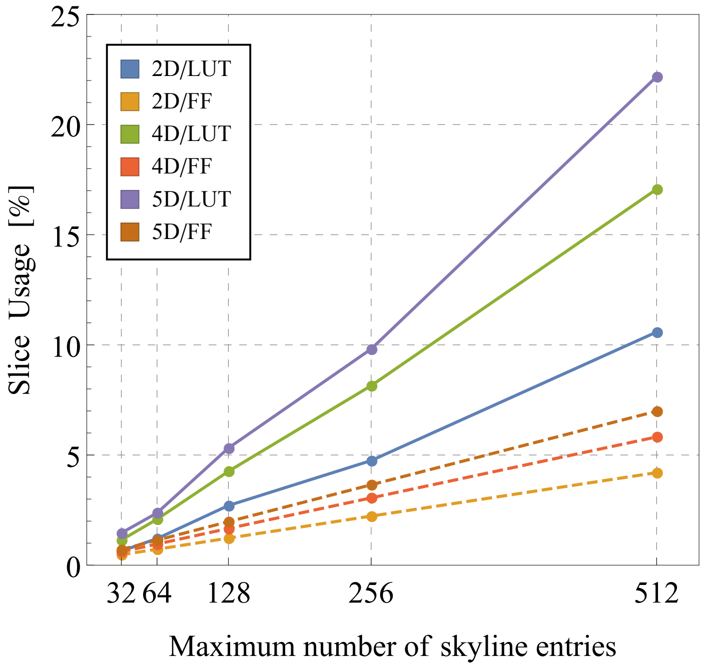


Figure 5.32: Slice usage vs maximum number of skyline entries of the LSCA

Xilinx VCU108 FPGA platform was approximately 18 W. As shown in Table 5.7, the computer used for the software implementations was equipped with an Ivy-Bridge processor and its TDP is 77 W. For comparison, we measured the power consumption of the computer while running the existing software algorithms with a power meter, showing that it consumed 43 W. Thus, the power consumption of the FPGA platform was half that of the processor, and that of the FPGA chip was a tenth or less. Therefore, for solving problems that do not require floating-point arithmetic operation, such as skyline computation using an algorithm with hardware-optimized tree search (e.g., JR-tree), FPGAs offer substantial advantages.

Table 5.9: Power consumption of a Xilinx Virtex Ultrascale XCVU095

	Logic A	Logic B	Logic C
Dimensions	2	4	5
Max. skyline entries	128	256	256
Max. active entries	1024	1024	4096
Dynamic	0.402	0.740	1.254
Skyline comparator	0.120	0.414	0.826
Clocks	0.072	0.137	0.161
Signals	0.072	0.208	0.474
Block RAM	0.066	0.079	0.158
MMCM	0.114	0.114	0.114
Device Static	1.378	1.386	1.400
Total On-Chip Power	1.780	2.127	2.654

5.11 Conclusion

The skyline computation algorithm is used for extracting the interesting entries from a database of multi-attribute entries. The task of computing a *dynamic* set of points is known as the continuous skyline computation. The continuous skyline computation is useful for removing the non-skyline points in preprocessing for screening a large amount of data in real time. We proposed the BJR-tree structure for continuous skyline computation. The fast speed of the BJR-tree is conferred by an appropriate hierarchical expression and dimensionality independence. To handle artificial datasets with temporal features requiring many computations, we proposed an ND-cache mechanism. The BJR-tree and ND-cache store the dominated relations and the important non-dominated relations, respectively.

We competed our proposed algorithm against the extended BNL and BBS (for continuous skyline computation), and LookOut. We used datasets with randomly-generated vectors and real datasets of blood cell measurements. On the synthetic datasets, the BJR-tree computed the continuous skylines approximately 3–70 times faster than LookOut. On real-world datasets, the BJR-tree is approximately 2.4–3.2 times faster than LookOut.

Our main contributions are as follows. (a) We proposed a new tree structure and a new cache mechanism for continuous skyline computation. Our approach reduces the number of avoidable comparisons and stores the previously calculated results. (b) Combined with the ND-cache, our proposed algorithm speeds up the continuous skyline computation, as confirmed in comparisons with existing algorithms. (c) In terms of execution time, BJR-tree outperforms LookOut on real-world medium-dimensional datasets extracted from cells.

To determine the usefulness of the skyline cells extracted by BJR-tree, we must biochemically analyze the cells after repetitive cultivation and extraction. This study confirmed that by appropriately selecting the features, we can stably isolate rare skyline cells.

Although skyline computation is a significant task for AI computation, little attention has been paid to the hardware-based acceleration of the skyline problem. Continuous skyline computation is a useful preprocessing step to help screen large amounts of data in real time by removing non-skyline entries. In this chapter, we have proposed an efficient FPGA-based approach to accelerate the JR-tree algorithm, LSCA. The LSCA minimizes the skyline computation time and realizes high-throughput continuous skyline query processing, and parallelize continuous skyline computation while reducing energy consumption. The evaluation results show that the LSCA was 5.5x and 3.1x faster than a software implementation of the LookOut and JR-tree algorithms for synthetic datasets on average, respectively. For two real-world datasets, the LSCA was approximately 2.8x and 4.8x faster than LookOut and JR-tree, respectively. The experimental results show that our algorithm not only accelerated the computation but also reduced power consumption. LSCA with DJR reduced the latencies of skyline query processing by 80% from LookOut and 71% from JR-tree on average.

Our main contributions are as follows. (a) We have proposed a new hardware-based algorithm (LSCA) that exploits the intrinsic parallelism of continuous skyline computation. (b) Evaluation shows our proposed LSCA implementation reduces both execution time and energy consumption from existing optimized software implementations. (c) We have implemented a classifier integrated into an actual image-activated cell sorter system for biological and medical
application fields.

Chapter 6

Design Methodology for High-performance FPGA-based Accelerators

6.1 Introduction

In software design, automatic optimization has been realized at the practical level through cooperation of the CPU, operating system, and compiler. Software programmers do not have to be highly aware of parallelization, and with automatic parallelization of compilers, for-loop statements are automatically executed across multiple threads. The data layout of the execution of a software program is controlled automatically by register renaming of the compiler, virtual memory management of the operating system, and the CPU cache mechanisms. Therefore, programmers do not have to be aware of where calculated data are stored and how the data are read.

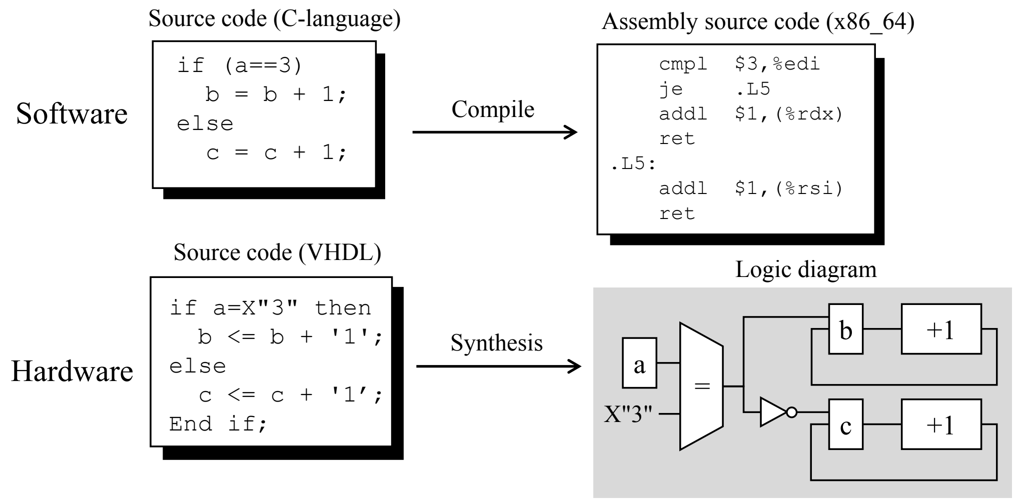
In hardware design, designers must optimize parallelization and the data layout without assistance from the compiler, operating system, and CPU. Instruction sequences arranged in the execution order and control syntax, which determine the additional execution order, are described in the software code. On the other hand, the static structures of circuits are described in the hardware code. Programming with the API framework for a GPU can be considered to be a continuation of the programming in CPU. Software programmers can implement GPU-based applications by acquiring several extended specifications. However, hardware design is not a continuation of software design, and it is more challenging to learn hardware design as compared to the software design (i.e., CPUs and GPUs).

It is widely known that hardware-based accelerators, particularly FPGAs, are more efficient at high-performance computing than CPUs and GPUs. Therefore, the requirements for software programmers to design high-performance FPGA-based accelerators have increased significantly. Behavioral synthesis technology has appeared as a support tool for software programmers to design hardware. Behavioral synthesis technology converts a code written by software programmers into a hardware RTL code. This technology is expected to enable software programmers to design high-performance FPGA-based accelerators efficiently; however, the performance of the generated designs was not as high as expected. Compiling and assembling a software program involve converting behavioral description to behavioral description. On the other hand, behavioral synthesis converts a behavioral description to a circuit structure. This is the reason why it is difficult to generate high-performance hardware designs. Current behavioral synthesis technology is insufficient in automatic optimization. Skilled hardware designers can make an efficient design using the HDL coding.

Owing to the progress in hardware development environments, hardware design has become much easier. However, designing a high performance FPGA-based accelerator remains difficult. Methodologies that would enable less experienced hardware designers to acquire an ability to design circuits that are equivalent to those developed by experienced designers have not been discussed sufficiently in previous studies. In this chapter, we discuss the important hardware design elements obtained from our studies. The keys are multi-level parallelism and an application-oriented data layout, which are described in the form of a design methodology.

6.2 Hardware Design Difficulties

Development models, as well as the code contents, differ significantly between software and hardware development. Example software and hardware development models are shown in Figure 6.1. In the software code, instructions to be executed on the CPU are described in order. Even if the processor has an out-of-order execution function, the result is the same as when the code is executed as described. In most software development projects, a processing order flowchart is created, and the code is written based on this flowchart. Even if the code is compiled and converted to assembly or machine languages, the order of processing remains the same. On the other hand, hardware designers often create an initial rough block diagram of circuits and timing charts and write an RTL HDL code based on these block diagrams. Current hardware development primarily uses the HDL-based approach, which is employed in the development model shown in Figure 6.1. The RTL HDL code describes the "structure" of the circuits rather than the processing procedures. Even if the RTL HDL code is synthesized into a netlist code describing schematic information, the netlist represents the "structure" of gate-level logical connections. In hardware design, the structure corresponds to the CPU architecture in software design. Hardware design is difficult because elements in the structure have high degrees of freedom, such as parallelization of the execution unit, data paths, and data layout in the memory hierarchy. Among the various elements, parallelization and data layout are observed to considerably affect the performance. A design methodology to determine such elements is required to design high-performance accelerators.



```
if (a==3)
  b = b + 1;
else
  c = c + 1;
```

```
    cmpl  $3,%edi
    je    .L5
    addl  $1,(%rdx)
    ret
.L5:
    addl  $1,(%rsi)
    ret
```

```
if a=X"3" then
  b <= b + '1';
else
  c <= c + '1';
End if;
```

Figure 6.1: Software and hardware development models. Behaviors are described in the software model, and structures are described in the hardware model.

Here, we consider two computations and compare the software and hardware development. The first computation is a population counting (popcount) function, i.e., a bit summation function, that returns the number of "1" digits in binary number representation. Algorithm 4 shows the pseudocode of a naive popcount function that includes for-loop statements and conditional branches. The pseudocode for another well-known popcount algorithm is shown in Algorithm 5. Here, the returned values for all input values are stored in a read-only array in advance. This code invokes memory access and does not include for-loop statements. Figure 6.2(a) shows the optimized hardware logic of the popcount function. Here, each bit of an input value is connected directly to adders, and the circuit outputs a bit summation. Here, 1-bit adders, which are also referred to as half adders (HA), are placed in the first stage of the logic shown in Figure 6.2(a). The logic design is a structural representation of how logical gates and logical modules are connected. Algorithm 6 shows the pseudocode for a faster popcount function. Here, the pseudocode consists of bit-operation instructions. A sufficiently optimized code that contains this code may be an approximate representation of the structure. The circuit shown in Figure 6.2(b) is a hardware implementation of Algorithm 6. Note that this circuit does not obtain performance that is as high as the circuit shown in Figure 6.2(a). Using general logical synthesis technology, it is possible to optimize circuit (b) to circuit (a). Several processors have native instructions for very simple calculations, which is equivalent to a hardware approach. For example, Intel's IA-32 architecture implements an embedded function `popcnt()` to perform the popcount. Here, we introduce the dominance relation calculation with four-dimensional vectors, which is a more complicated computation than the popcount function. Algorithm 7 shows the pseudocode of the dominance relation calculation function. Note that the code includes for-loop statements and conditional branches. Figure 6.3 shows a manually optimized circuit. In more complicated applications, such as the dominance relation calculation, even though the behavioral description is complicated, the structural description may be simplified through optimization. Behavioral synthesis converts software code to hardware circuits. With the current behavioral synthesis technique, generating a hardware circuit, such as that shown in Figure 6.2(a), using a software code, such as algorithms 4 and 5, is difficult. However, it is relatively easy to generate the hardware circuit shown in Figure 6.2(b) from a software code, such as Algorithm 6. Although it is possible to convert a behavioral description that is approximately close to the structural description into an actual structural description, it is challenging to convert a general behavioral description into a structural description.

Algorithm 4 Popcount function 1

```
1: procedure popcnt1(u_int8 $a$)
2: while $a \neq 0$ do
3:     if ($a$ & 0x1) $> 0$ then
4:         $c = c + 1$
5:     $a = a >> 1$
6: return $c$
```

Algorithm 5 Popcount function 2

```
1: procedure popcnt2(u_int8 $a$)
2: mem[256]= $\{0, 1, 1, 2, 1, \cdots, 8\}$
3: return mem[$a$]
```

Algorithm 6 Popcount function 3

```
1: procedure popcnt3(u_int8 $a$)
2: $a = (a$ & 0x55$) + (a >> 1$ & 0x55$)$
3: $a = (a$ & 0x33$) + (a >> 2$ & 0x33$)$
4: $a = (a$ & 0x0f$) + (a >> 4$ & 0x0f$)$
5: return $a$
```

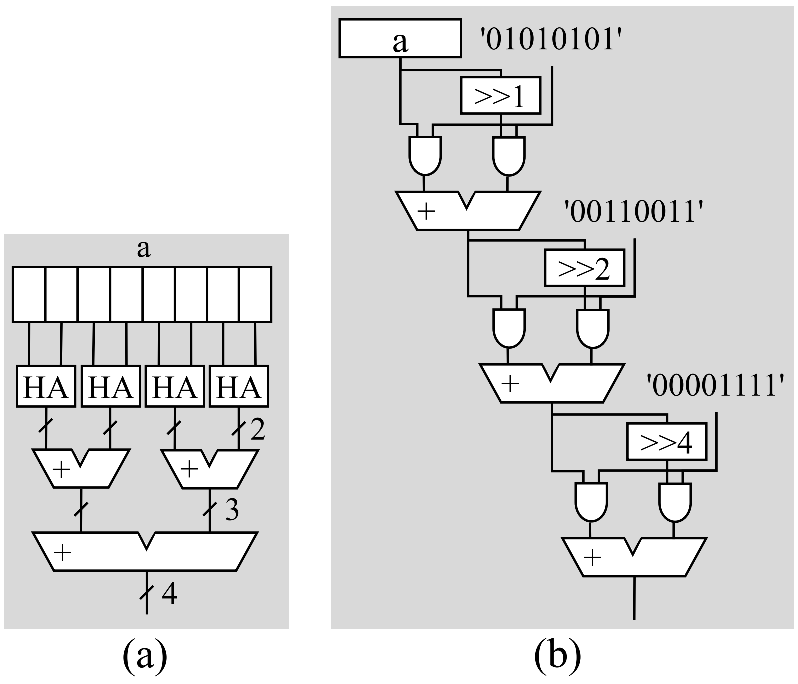


Figure 6.2: Popcount function logic diagrams: (a) optimized logic and (b) manually converted logic from Algorithm 6.

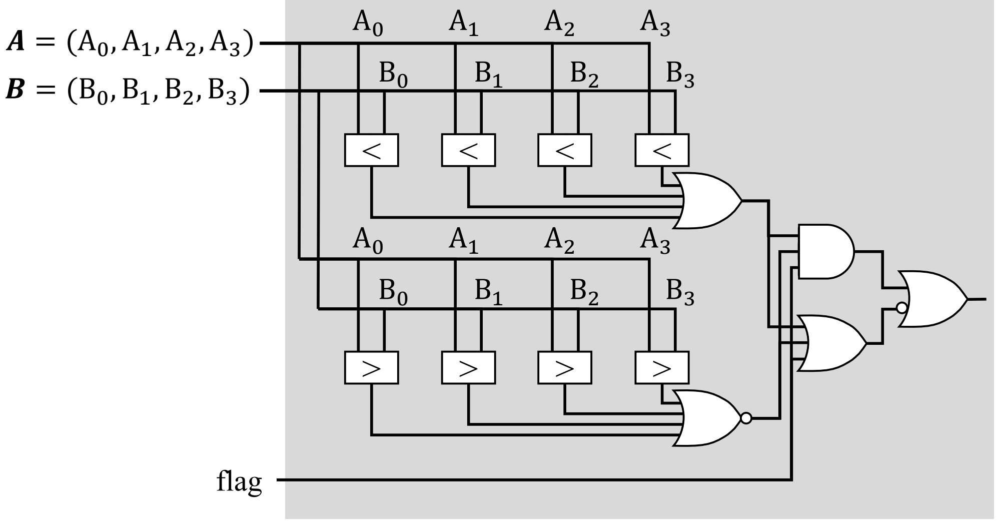


Figure 6.3: Dominance relation function logic diagram

Algorithm 7 Dominance relation function

```
1: procedure domrel1(int $a$[4], int $b$[4], int $flag$)
2: $c = 0$
3: if $flag = 1$ then
4:    for $0 \leq d < 4$ do
5:       if $a$[d] $>$ $b$[d] then
6:          return 0
7:       else if $a$[d] $<$ $b$[d] then
8:          $flag = 1$
9: else
10:    for $0 \leq d < 4$ do
11:       if $a$[d] $<$ $b$[d] then
12:          return 0
13:       else if $a$[d] $>$ $b$[d] then
14:          $flag = 1$
15: return $flag$
```

6.3 Previous FPGA Design Methods

6.3.1 Schematic-based Design

Previously, various logic design methods have been employed. In the 1960s, engineers employed a *schematic-based design* [89], i.e., logic diagrams for electronic circuits were drawn manually. Logical optimization, simulation, and verification have also been performed manually. Currently, a computer-aided design (CAD) system can be used to create logic diagrams. However, even if CAD systems are used, logic gates are positioned and wirings are drawn in the logic diagrams manually. Currently, HDL-based design and behavior-based design methods, which are described in sections 6.3.2 and 6.3.3, have become common. However, schematic-based design is still employed because it realizes circuits with the optimal performance and efficiency. However, the schematic-based design method requires a long development period; thus, for large-scale projects, using schematic-based design for an entire circuit is unrealistic. In situations where a critical part of a design that is developed using other design methods is optimized locally, the schematic-based design remains a realistic approach.

6.3.2 HDL-based Design

Designing large-scale logic within a realistic timeframe using the schematic-based design method is difficult. However, the HDL-based design method, which significantly reduces the development time, has been introduced. Unlike software programming languages, such as C and C++, HDLs, such as Verilog-HDL and VHDL, describe hardware logic. Low-level circuit elements, such as logical gates and adders, are described as abstract grammar representations. Representing the input and output of registers from the viewpoint of data flow is referred to as an RTL description. With HDL-based circuit design, the RTL HDL code is written manually. Initially, Verilog-HDL and VHDL considered simulation and verification rather than synthesis. Note that advances in synthesis tools have achieved a certain level of performance. However, HDL-based design still requires longer development time than software programming.

Figure 6.4 shows development flows of software and HDL-based FPGA circuit design. The feasibility analysis of software development is made easier by the processor, operating system, and register mapping of the compiler. Therefore, software programmers rarely perform taxing feasibility analyses and tend to concentrate on algorithms that have to be implemented.

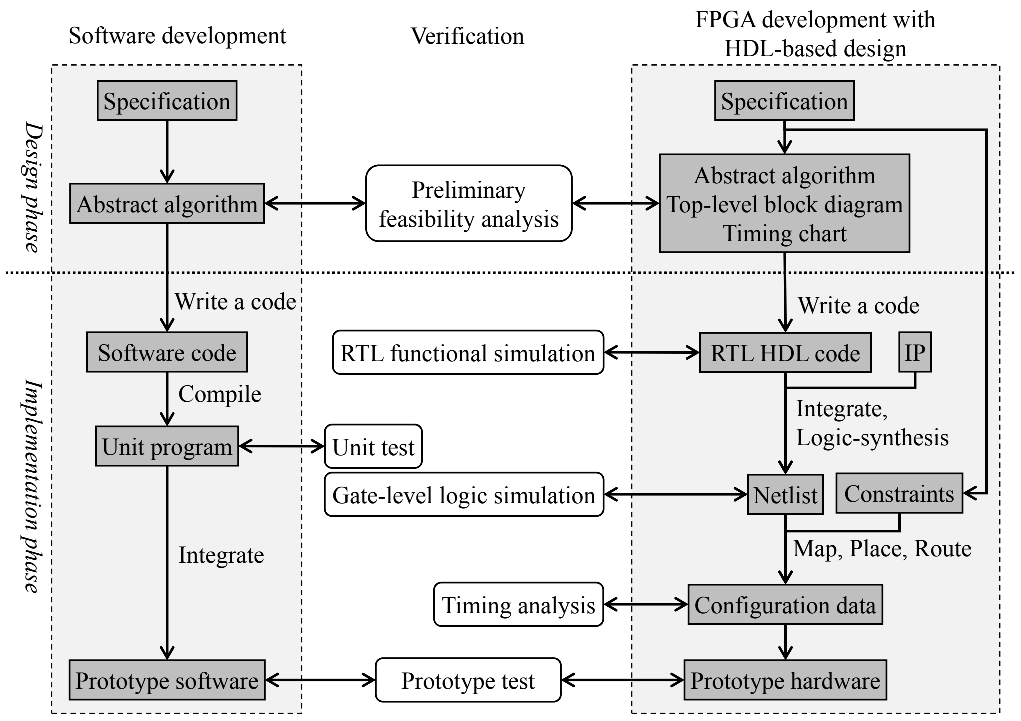


Figure 6.4: Development flows of software and HDL-based FPGA circuit design

6.3.3 Behavior-based Design

Mainstream hardware design has shifted from classical schematic-based design to HDL-based design. However, as mentioned previously, hardware design still consumes significant development time. Recently, design methods with more abstract behavioral descriptions have become available. Target users of such methods are software programmers who intend to design hardware circuits. In behavior-based design, a code written in an extended HDL or an extended high-level language, such as SystemC [60], is converted to an RTL HDL code or gate-level netlist code. Here, extended C or other high-level languages are referred to as C-like languages. Note that we will describe the examples of the C-like language later in this section. This conversion process is referred to as a behavioral synthesis or high-level synthesis (HLS).

First, the C-like code is analyzed and converted to an intermediate form with dataflow graphs and state machines. Then, after optimization, the RTL HDL code is generated.

Many extended high-level languages for behavior-based design have been proposed and standardized. For example, the Unified Design Language for Integrated Circuit (UDL/I) [68] was proposed by Japan Electronic Industry Development Association (JEIDA) in 1990. Unfortunately, UDL/I is not widely used outside of Japan. In 2000, Superlog, which combined Verilog-HDL and the C language, was introduced by Co-Design Automation (CDA) [40]. SpecC, which is a descriptive language extended from the C language, was proposed in 2001 [45, 37]. In 2005, Accellera proposed SystemVerilog which is an extension of Verilog-HDL.

Note that SystemVerilog has been standardized as IEEE 1800-2005 [51]. In SystemVerilog, the verification functions were enhanced; however, the behavioral design environment did not receive considerable improvements, and it maintains a similar functionality to that of Verilog-HDL. In VHDL-200X, an improved version of VHDL, specifications were improved to comply with IEEE 1076-2008. Bluespec SystemVerilog (BSV), which is based on Haskell, was proposed in 2003 [94]. In BSV, type verification is performed during the compile process. OpenCL [113], which is an extension of the C language, was introduced in 2008. Libraries and device drivers for I/Os, such as memory, PCI Express, and DMA are available in OpenCL. Note that the OpenCL code can be adapted to a new FPGA board by recompiling the code. The OpenCL code does not depend on the memory capacity, memory type, or capabilities of the hard-macros (e.g., DSP) of the target FPGA board. OpenCL provides a device-independent development environment, and its use is currently expanding. In 2011, SystemC, which is an extension of C++, was standardized as IEEE 1666, and users can employ SystemC as class libraries in C++. Owing to the development of these high-level languages, it became possible to verify designs in a short time using behavior-level simulation. Furthermore, it has become possible to develop cooperating software and hardware seamlessly. These extended languages differ only relative to the given language specifications and verification functions. The languages are similar in that a behavioral description written in a C-like language is converted to a circuit design. SystemC and SystemVerilog include various specification extensions for more abstract behavior descriptions. Basically, in behavioral synthesis, local variables, global variables, array variables, and functions are converted to registers, internal RAM, FIFOs, and circuit modules, respectively. Efficiency of the HDL code that is generated by behavioral synthesis is strongly dependent not only on the language extensions but also on the synthesis algorithm of each behavioral synthesis tool.

Figure 6.5 shows the developmental flow of the behavior-based design. In HDL-based design, the HDL code is written manually. In contrast, in behavior-based design, the HDL code is generated from the C-like code using a synthesis tool. In behavior-based design, directives are sometimes inserted manually into the C-like code. Here, directives are used as hints to create circuits that the designers provide to the synthesis tool.

Here, we describe the Xilinx Vivado Design Suite as an example of a consistent development environment, which covers various elements including synthesis tools, verification tools, IP cores, chip devices, and evaluation boards. The Xilinx Vivado Design Suite consists of Vivado, a logical synthesis tool, and Vivado HLS, a behavioral synthesis tool. Vivado logical synthesis tool converts an RTL HDL code to a netlist code. Note that Vivado HLS supports C, C++, SystemC, and OpenCL.

We illustrate the advantages of the behavior-based design. Behavioral synthesis tools can generate a high-performance HDL code for simple calculations. In addition, verification is fast and easy. HDL-based design requires a considerably long time for coding, simulation, and debugging processes. In contrast, behavior-based design reduces the development time of each process, thereby reducing the overall development time [54]. Functional behavior simulation is much faster than logical gate-level simulation, and behavioral synthesis tools can automatically divide an algorithm into multiple stages to generate a pipelined circuit. Here, users must use a directive statement to designate which calculation should be pipelined.

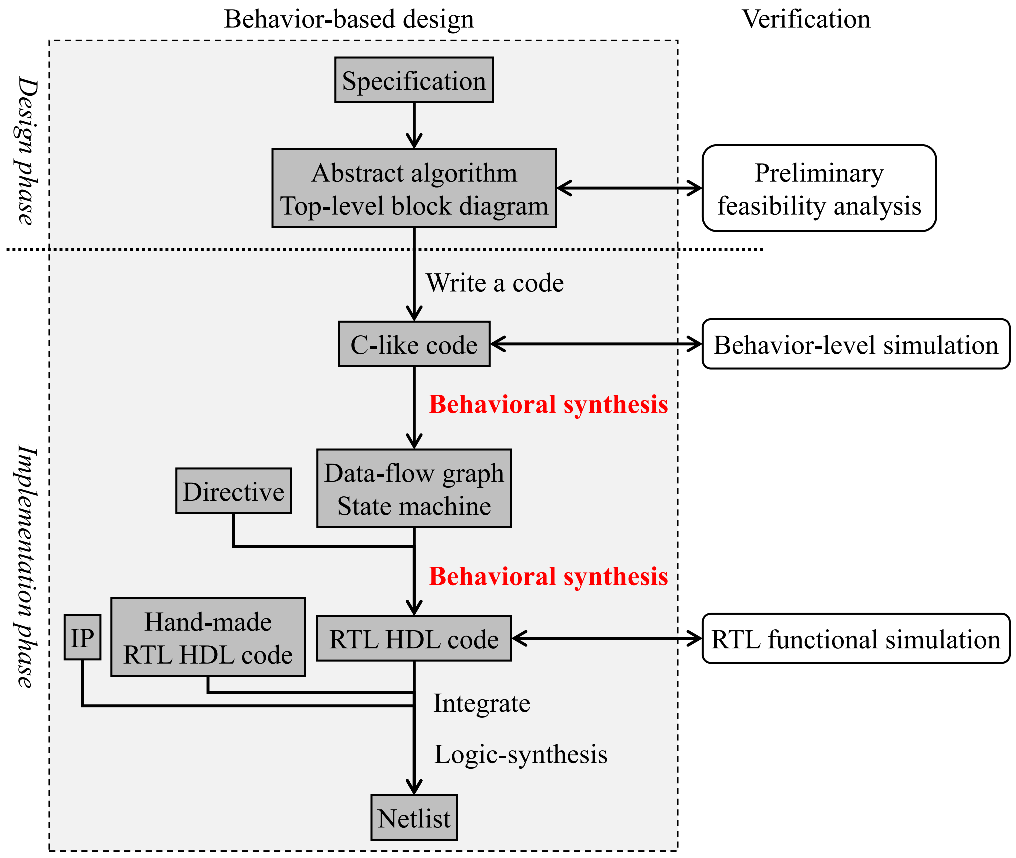


Figure 6.5: Development flow of a behavior-based design for an FPGA

6.3.4 Limitations of Behavior-based Design

Although abstract behavioral descriptions can be written in C-like language, there are many restrictions. Such restrictions vary according to the tool employed. Major restrictions are listed in the following.

1. **Recursive function calls are prohibited**

   In contrast, the C language and many other high-level languages permit the use of recursive functions. In the C-like code, the function corresponds to a module in the circuit. Note that the number of times a recursive function will be called can only be determined at runtime. Since the number of module layers cannot be determined statically, the behavioral synthesis tool cannot generate an exact circuit.

2. **Multiple indirections are prohibited**

   The C language and many other languages permit multiple indirections. A multiple indirection means that one pointer variable indicates the pointer of another pointer variable. Variable values are stored in the registers or on-chip RAMs. With multiple indirection, if a value of the pointer variable is changed, the register or RAM to be accessed will also change. Note that the register or RAM to be accessed can only be identified during runtime. Therefore, the behavioral synthesis tool cannot route the data paths to registers and RAM statically. Similarly, a structure definition in C-like code can have fixed-length arrays as members but cannot have pointers to arrays or structure variables.

3. **Arrays whose length is not determined at compile time are prohibited**

   The C language and many other languages provide memory allocation functions for an array of dynamic length, such as the `malloc()` and `calloc()` functions. On the other hand, for functions to be synthesized, dynamically allocated arrays cannot be accessed because the array may be reallocated with a different length in using a reallocation function, such as `realloc()`. A fixed-length multidimensional array can be synthesized; however, variable-length multidimensional arrays with multiple memory allocations and indirections cannot be synthesized. The length of a local array variable in stack memory must also be fixed. Thus, the definition and use of arrays whose length is not determined at compile time are prohibited. In other words, we can write `int *a` as a *call by reference* of a single variable in a list of the function's parameters; however, we cannot write `int *a` or `int a[]` as a *call by reference* of an array. When a function with an input array variable is defined, the length must be added like `int a[10]`. For example, several behavioral synthesis environments for an SDSoC-based design provide specific alternative functions for dynamic memory allocation.

4. **Functions with subloops that cannot be unrolled cannot be pipelined**

   A loop statement whose number of iterations cannot be determined at compile time cannot be unrolled at synthesis time. Functions including such loops can be synthesized; however, they cannot be converted to a pipelined circuit.

5. **Other coding recommendations**

   For example, global variables in C-like code can be synthesized; however, this is not recommended because using a global variable in a wide area of a circuit leads to severe timing constraints. There are many other coding patterns that should be avoided even though they are synthesizable.

A code that is written for a software compiler may not be synthesized as it is. For example, C code must be rewritten as restricted C-like code, and this is not a simple code conversion and may require reconstitution of the software algorithm.

6.3.5 Problems of Behavior-based Design

In addition to the aforementioned restrictions, the behavior-based design approach has various problems, which are discussed in the following.

1. **Designers must insert directives for optimization.**

   An operable HDL code can be obtained if the C-like code satisfies the above restrictions. However, manual optimization is required to obtain a high hardware efficiency. One advantage of FPGAs is that resource and power consumption can be reduced by decreasing the bit width based on the precision and value range required by the target application. Here, bit width is indicated by an inserted directive. However, it is difficult to precisely indicate the optimization policies using such directives. For example, when pipelining a function, it is difficult to indicate how the function is to be divided into the pipeline stages. Note that determining which functions should be pipelined also requires a directive.

2. **The accuracy and limitation of behavior simulation and verification are problematic.**

   Functional behavior simulation is fast; however, certain coding bugs in error handling and boundary condition cannot be found by behavior simulation and verification techniques. Therefore, logical gate-level simulation and verification must be performed after synthesis. In addition to the above logical bugs, bugs related to the performance issues should also be eliminated. Since the HDL code generated by behavioral synthesis is difficult to understand, it is also difficult for designers to see the overall structure of a generated circuit. A designer can only verify the generated circuit using a black box process, and it is difficult to modify the HDL code directly.

3. **Directions of circuit specification**

   Although a target clock frequency can be specified, precise control of the logic delay in a generated circuit is difficult. In addition, it is difficult to control resource consumption. There is a trade-off between the maximum operating clock frequency and resource consumption. Resource consumption should be reduced while satisfying the required operating performance. However, it is difficult to control the trade-off during the behavioral synthesis.

Currently, behavioral synthesis is widely used in hardware development; however, it remains "rough-around-the-edges," and it is difficult to generate an HDL code for high-performance circuits from a C-like code unless it is simple. Note that OpenCL supports both GPUs and FPGAs, and although OpenCL generates a code optimized for a GPU, it is difficult to generate such code for an FPGA [63]. Software programmers must be aware of the circuit that will be generated while writing the C-like code. Currently, the main purpose of behavior-based design is to create IP of primitive functions. It is still not possible to completely replace the HDL-based design approach. Additionally, if an accelerator cannot provide better performance and efficiency than that of a CPU, there is no objective to produce or use an accelerator. The software approach for CPUs has shorter development time, easier maintenance, and lower cost. In software languages, the use of low-speed language such as scripting languages (e.g., Python and Ruby) is still meaningful because the scripting languages have higher portability and lower development costs than the C language. As mentioned previously, the hardware approach incurs greater costs than the software approach; thus, the hardware approach must yield considerable performance improvements. It is reasonable to say that the software programmers will be able to understand various features of code that is suitable for the hardware during the process of writing a restricted C-like code. The knowledge that is obtained in such a way is useful to design high-performance accelerators.

6.3.6 Hybrid Approach of Software and RTL

We can implement a general-purpose processor on an FPGA. A processor implemented by logical block resources is referred to as a soft processor core, and a built-in processor on the FPGA device is called a hard processor core. A soft processor core can also be instantiated on FPGAs that do not have a built-in processor. Note that the hard processor core does not consume logical block resources. Currently, Xilinx and Intel have released FPGA devices with ARM's processor as the hard processor core. A hybrid approach is also used with FPGAs that have a processor core. In the initial development stage, the entire software code is compiled and executed on the processor core of the FPGA. Here, the functional behavior simulation of such an accelerator is easy to debug. Then, a part of the primitive functions is converted to HDL code by manual coding or behavioral synthesis. When a converted function is called in the processor core, the processor core drives the corresponding circuit rather than calling the software function. This approach is more efficient than when an external CPU device drives the acceleration circuit on the FPGA. In this approach, calculations that are not suitable for FPGAs (e.g., floating-point operations) are performed by the processor core. Here, the performance of the converted or written HDL code dominates the overall performance. In the latest version of OpenCL, we can directly invoke the highly optimized IP and RTL HDL design of other hardware designers. Also in the hybrid approach, it is important for realizing a high-performance logic to generate or write an optimized HDL code.

6.4 Proposed Design Methodologies

6.4.1 Overview

Logical synthesis has become highly technical, and simple design guidelines for small-scale circuits have been studied and incorporated into logical synthesis tools. Therefore, the need for precise manual optimization has been reduced. If we only write the RTL HDL code, the logical synthesis tool optimizes the logic delay and resource consumption and generates netlists. Thus, the HDL-based design approach realizes high-performance FPGA-based accelerators. The performance of the manual RTL HDL code strongly depends on the hardware designers. Not all designers can design a high-performance accelerator. The behavioral synthesis environment supports the software programmers in designing circuits in high-level languages. However, the practicality of behavioral synthesis technology is still far from that of manual RTL design. In particular, it is a challenging task to detect parallelisms automatically, generate parallelized circuits, and determine the data layout in the memory hierarchy automatically. Currently, these tasks rely on manual insertion of directives.

A simple design pattern containing pipelining and optimization techniques for simple circuits has been discussed in previous studies [89, 131, 2, 135]. Unfortunately, a unified design methodology for high-performance accelerators for real-world applications has not yet been established. Such a methodology is essential to perform behavioral synthesis; however, it is absent from the current behavioral synthesis flow.

In this chapter, we propose a design methodology for high-performance FPGA-based accelerators using the HDL-based design approach. The proposed design methodology is expected to be useful for hardware designers to systematically design logic circuits. Simply developing an accelerator that works at least is insufficient to elicit the entire capability of the FPGA device. Engineers must pursue high parallelism and an optimal data layout in the memory hierarchy. Nevertheless, the importance of these two concepts has not been emphasized in previous studies.

Here, we describe the first concept, which we refer to as multi-level parallelism. Parallelizable calculations have various parallelization granularities. The entire computation runtime is reduced by parallelizing at multiple granularity-scales simultaneously. Therefore, we should parallelize calculations with as many granularities as possible. The parallelisms are divided into three types, i.e., data, task, and pipeline parallelism. Data parallelism indicates that the same processing can be executed in parallel for each element in a data array. Task parallelism means that different processes without dependencies can be executed in parallel. Pipeline parallelism means that the same processes can be executed in parallel by separating the process into multiple steps and executing the processes such that the same steps are not executed simultaneously. We can create a circuit pattern to realize each type of parallelism. Note that these parallelism techniques have only been discussed independently in the context of classifying a software code. However, in the design of FPGA-based logic circuits, it is important to parallelize application-specific computations simultaneously at multiple granularities.

Here, we describe the second concept, which we refer to as an application-oriented data layout. There are many types of on-chip memory blocks on an FPGA and off-chip memory devices around the FPGA. Specifying a data layout indicates determining the location at which data should be stored from across various memories. Figure 6.6 shows the specifications of typical types of memory relative to FPGAs. Here, distributed RAM and block RAM are the on-chip memory blocks of an FPGA. Static random-access memory (SRAM), reduced latency dynamic random-access memory (RLDRAM), and synchronous dynamic random-access memory (SDRAM) are memory devices that can be accessed using circuits on the FPGA. DDR4 SDRAM is an available state-of-the-art type of SDRAM. External large-capacity storage devices, such as solid-state drives (SSD), may be connected to the FPGA board via a cable or socket. Note that these memory and storage devices have various capacities. The capacity of off-chip devices is approximately 10 to 100 gigabits, and the overall capacity of registers implemented with flip-flops is limited to approximately 100 kilobits because of the resource capacity and timing constraints. The capacity of distributed RAM implemented with LUTs is approximately 1 to 10 megabits, and the capacity of block RAM is approximately 10 to 100 megabits in currently available FPGAs. Note that memory capacity and access latency should be considered when determining the data layout. The data layout is an important factor that determines the upper performance bound of an FPGA-based accelerator. If we begin development with

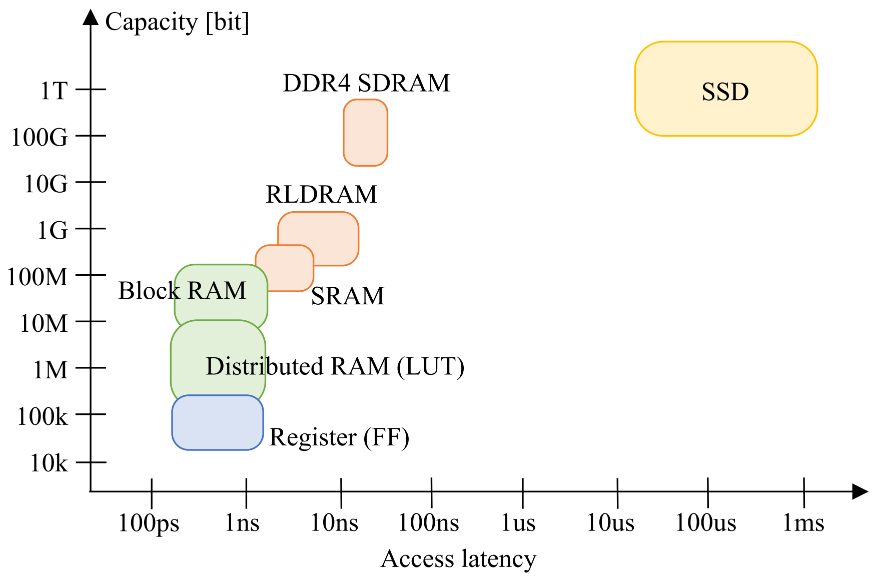


Figure 6.6: Capacity and latency specifications of memory blocks on FPGAs and memory devices (2017)

the design of an FPGA board, we can control the design of the memory hierarchy. Generally, the memory that is located near to the computing circuit and exhibits short access latency tends to have a small capacity. In contrast, the memory that is located far away from the computing circuit and exhibits long access latency tends to have a large capacity. There is a trade-off between memory capacity and access latency, i.e., in distributed or block RAM, read data become available in the next clock cycle of a read request. On the other hand, the access latency to off-chip memory is approximately a few tens of nanoseconds. Such long latency suspends the operation of the circuit. However, determining an effective and efficient data layout is not a simple task because it should be optimized based on the data size, data access pattern, and data access frequency of an actual test run of the target application. Note that the impractical data layouts should be redesigned. The data layout has already been automated in behavioral synthesis; however, analysis of the input code is insufficient, and an optimal data layout is not necessarily provided by the behavioral synthesis. Behavioral synthesis often rejects dynamic memory allocation in the input code, and synthesis is restricted to a code that contains only static memory allocation. The behavioral synthesis tool should use trace information for multiple input datasets to determine the optimal data layout. Currently, we must manually analyze a code to obtain a good data layout.

6.4.2 Design Flow

Figure 6.7 shows the design flow of the proposed methodology. This design flow is used in the steps from “specification” to “RTL HDL code” of the HDL-based design flow shown in Figure 6.4. In step 1, we first describe an abstract algorithm for the target application. This abstract algorithm contains the pseudocode of the primary computations. In step 2, we estimate the amount of hardware resources required by the abstract algorithm. Specifically, we estimate how much memory bandwidth and capacity each calculation will require. When the required memory size massively exceeds the available capacity, we cannot implement the abstract algorithm on the FPGA. In this case, we must return to step 1 to modify the abstract algorithm. In step 2, we can roughly estimate the theoretical performance limit of the accelerator to be implemented. In step 3, we enumerate candidates for parallelizable functions in the pseudocode of the algorithm. Note that several functions with nested structure can be parallelizable at multiple levels. Here, we enumerate parallelism at all levels. In step 4, we divide the parallelizable functions enumerated in step 3 into three types of parallelism (data, task, and pipeline parallelism). The classification principle is described in the next subsection. In step 5, we design circuits using patterned circuits for each type of parallelism. The abstract algorithm is also converted to a set of circuit modules in this step. Note that functions with multi-level parallelism are converted to nested circuit modules. By reviewing this logic diagram, we can understand the entire circuit. In step 6, we enumerate the data handled in each of the designed computation logic circuits. Here, we calculate and check the size and throughput of all data. Concretely, we clarify how much data the circuit designed in step 5 receives in a single clock period. In step 7, we determine memories on which data are stored based on the size and throughput of the data estimated in step 6. Note that data are not necessarily stored on a single type of memory. To facilitate adequate performance, the same data may be stored in multiple types of memory. When we are stuck on the data layout, we will cancel several parallelization components or return to step 1 to modify the abstract algorithm. The memory decision principle is described in the next subsection. In step 8, we write the RTL HDL code based on the circuits designed in the previous steps. At this point, since the data layout has been determined, here, the main task is designing data paths between the modules, including the memory access controllers.

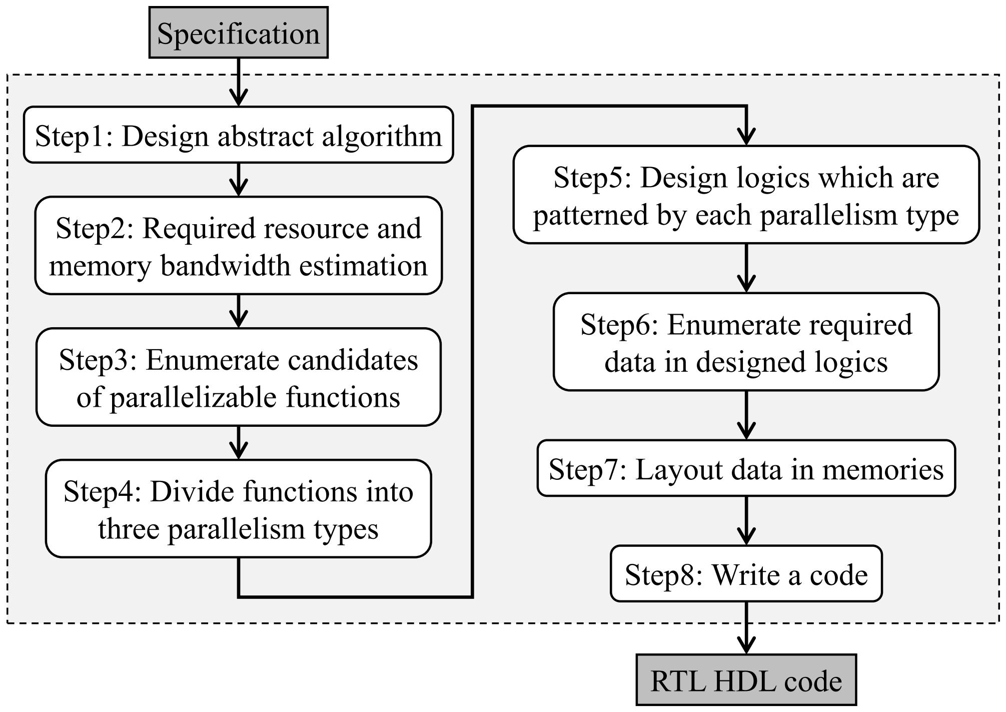

Specification
Step1: Design abstract algorithm
Step2: Required resource and memory bandwidth estimation
Step3: Enumerate candidates of parallelizable functions
Step4: Divide functions into three parallelism types
Step5: Design logics which are patterned by each parallelism type
Step6: Enumerate required data in designed logics
Step7: Layout data in memories
Step8: Write a code
RTL HDL code

Figure 6.7: Flow of the proposed design methodology

6.4.3 Design Principles

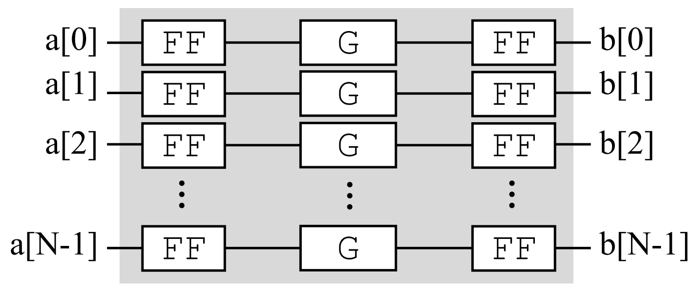
Here, we describe the design principles in the three steps of our design flow. In the following, we sequentially show the design principles required in steps 4, 5, and 7.

First, we discuss how to categorize the functions in step 4. When multiple data elements are processed in the same procedure, these elements can be processed in parallel using the same logic circuits. Note that such processing has data parallelism. For example, consider a circuit for the pseudocode shown in Figure 6.8(a). In this code, the data elements are given in the form of an array. We apply a function to each element of the input array and assign the returned values to another array. Such a computation is parallelizable at a level of the array index. Multiple processes without mutual dependency have task parallelism. Because of the existence of conditional branches, multiple processes can exhibit different sequences of instruction execution. Such processes exhibit task parallelism. Thus, we observe that multiple processes exhibit task parallelism even when different data are provided as input to the same algorithm. In processes with task parallelism, the input data are not necessarily given as an array at the same time. Note that the format of the input data and the timing when the input data are given may differ. For example, consider a circuit for the pseudocode shown in Figure 6.9(a). Here, two integer numbers `a` and `b` are given, and different functions `F()` and `G()` are applied to `a` and `b`. `H()` depends on the results of `F()` and `G()`, but `F()` and `G()` do not depend on each other. Thus, the processes of `F()`

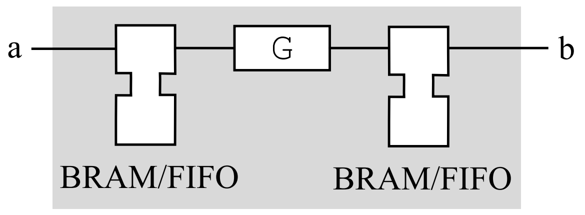
and G() can be executed in parallel. When a process can be divided into multiple steps, multiple executions of the process for different datasets can be parallelized. Such a process has pipeline parallelism (Section 2.4). For example, consider a circuit for the pseudocode shown in Figure 6.10(a). This code includes a product-sum operation. Since multiplication and addition are executed sequentially, these instructions can be divided into two pipeline stages. Pipelining significantly improves throughput in the execution of instruction sequences (e.g., on the CPU) and streaming applications (e.g., multimedia CODECs). As described above, we find parallelizable functions and classify them by the parallelism type in step 4.

```
function(int a[N], int b[N]) {
  for (i=0;i<N;i++)
    b[i]=G(a[i]);
}
```

(a) C-like code



(b) Logic diagram



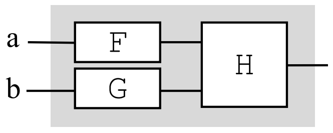
(c) Logic diagram

Figure 6.8: Design pattern for data parallelism. Here, C-like code (a) has data parallelism. Logic diagrams (b, c) are design patterns for code (a).

Second, we describe how we apply patterned logic circuits in step 5. We show an example of a circuit for functions with data parallelism in Figure 6.8(b). This is a circuit for the pseudocode shown in Figure 6.8(a). Here, in circuit (b), all input and output data are stored in flip-flops. Note that the flip-flops are directly connected to the input and output ports of the combinational circuit for function G(). Circuit (b) completes the processes for all data elements in a single clock cycle. However, a weak point of circuit (b) is that the consumption of flip-flop resources is proportional to the length of the array. We show another example circuit in Figure 6.8(c). Here, in circuit (c), the input data are read sequentially from a RAM device, and the calculation results are written sequentially to another RAM device; thus, the functions are not executed in parallel, and circuit (c) consumes memory blocks and few flip-flop resources. We show an example circuit

```
function(int a, int b) {
  return H(F(a),G(b));
}
```

(a) C-like code

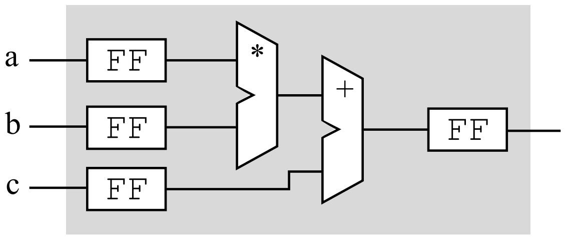


(b) Logic diagram

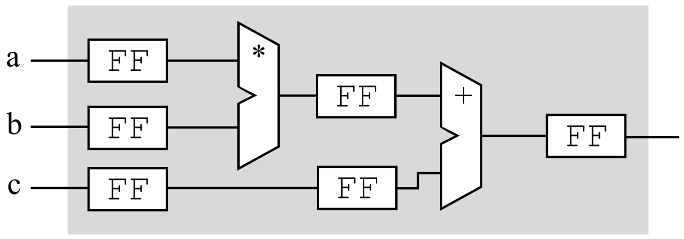
Figure 6.9: Design pattern for task parallelism. C-like code (a) has task parallelism. Logic diagram (b) is a design pattern for code (a).

```
function(int a, int b, int c) {
  return a*b+c;
}
```

(a) C-like code



(b) Logic diagram (Not pipelined)



(c) Logic diagram (Pipelined)

Figure 6.10: Design pattern for pipeline parallelism. C-like code (a) has pipeline parallelism. Logic diagram (b) is a design pattern for code (a) without pipeline parallelization, whereas logic diagram (c) is a design pattern for code (a) with pipeline parallelization.

for functions with task parallelism in Figure 6.9(b). This is a circuit for the pseudocode shown in Figure 6.9(a). Here, the circuits of F() and G() are arranged in parallel, and their output ports are connected to the input ports of the circuit of H(). Here, the total logic delay is observed to be the delay of circuit H() plus the delay of the slower of F() or G(). Note that task parallelism differs from data parallelism in that the granularity of the synchronization of the parallelized circuits is coarse and different processes are performed in the same clock cycle. We show an example circuit for functions with pipeline parallelism in Figure 6.10(c). This is a circuit for the pseudocode shown in Figure 6.10(a). Here, by comparing circuit (c) with the not-pipelined circuit (b) in Figure 6.10, it can be seen that flip-flops (i.e., pipeline registers) are inserted between the multiplier and adder in this case. We show the timing chart of circuits (b) and (c) in Figure 6.11. In pipelined circuit (c), the multiplier calculates the next data, and the adder calculates the previous data. In this case, pipelining doubles the calculation throughput. As described above, we create concrete parallelized circuits for the software functions in step 5. Note that the hardware circuits are patterned after each type of parallelism and that we can arrange the patterned circuits and connect their input and output ports.

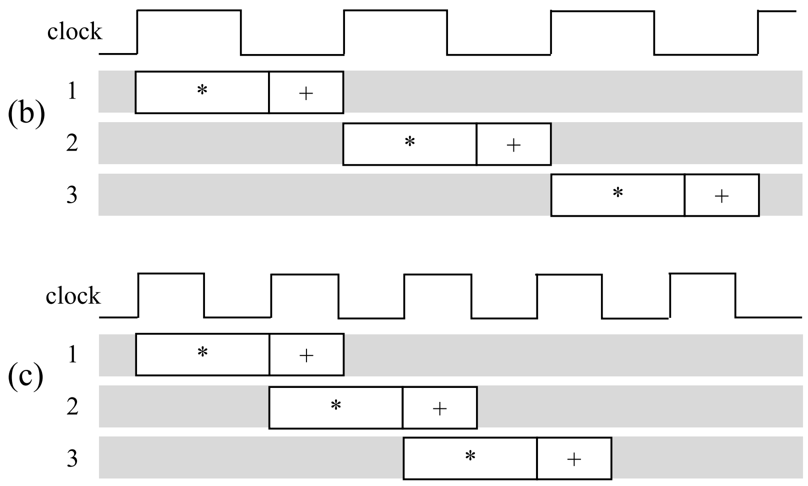


Figure 6.11: Timing chart comparison of the logic circuits in Figure 6.10(b, c)

Third, we show how to form the data layout in step 7. We enumerate the flip-flops and built-in memory blocks that were actually used in the circuits which were designed in the previous steps. Then, we consider the memory access pattern of each circuit based on its structure. If large amounts of data are accessed simultaneously in the same clock cycle, we place the data in flip-flops. If the data are accessed in sequential order or streaming manner, we place the data in FIFO blocks. If the data are accessed in a discontinuous order, we place the data in block RAM. If a significant memory capacity is not required and a smaller logical delay is required, we place the data in a distributed RAM. If the data size is beyond the capacity of on-chip memory blocks, we place the data in an external memory device, such as SRAM and DRAM. Table 6.1 depicts the access pattern, latency, and capacity characteristics of the available memory blocks and devices. For example, in the pseudocode shown in Figure 6.8(a), if the input array length is large and comparable to the available capacity of the flip-flop resources, the circuit shown in Figure 6.8(b) would be unrealistic. To achieve a more realistic circuit with less flip-flop consumption (Figure 6.8(c)), we must modify the data layout and change the patterned circuit. When using external memory, we access it by indicating a target address location; thus, we must implement a memory controller with an address management circuit. Since we have already obtained concrete circuits during the preceding steps, we can analyze the data size, access pattern, and access frequency more clearly than the estimation in step 2. In step 7, as described above, we ensure the data layout via clearer feasibility analysis.

Table 6.1: Features of the memory blocks and devices

Memory	**Access pattern**	**Latency**	**Capacity**
Flip-flop	Simultaneous	Short	Small
Distributed RAM	Discontinuous		
Block RAM			Medium
FIFO	Sequential		
Off-chip memory	Any	Long	Large

6.5 Examples: Applications of the Proposed Design Methodology

To explain the proposed design methodology in a concrete manner, we depict the design policies for FPGA-based accelerators that were implemented in our three typical studies, which are described in Chapters 3, 4, and 5. The well-considered hardware design compensates the shortcoming of the FPGA that the maximum operating clock frequency is low, and we can develop high-performance accelerators that outperform the CPU-based approach Relative to multi-level parallelization and the application-oriented data layout, we discuss the relationship between each study and the proposed design methodology in this section.

6.5.1 Accelerator for Chapter 3

In Chapter 3, we implemented an FPGA-based accelerator that increases the speed of the switching process of network traffic. Note that data transfer in computer networks should be processed in real time with low delay. An application that requires real-time processing and low latency is a representative example in which FPGAs are effective. First, we discuss parallelism. Packets of multiple streams from the downlink port FPGAs are received and scheduled in the uplink port FPGA of the merging stream harmonizer (MSH). These processes are parallelized at the port level in the uplink port FPGA, and the processes have task parallelism. Multiple stream processing is also load-balanced among four downlink port FPGAs. Second, we discuss the application-oriented data layout. In the MSH, several buffers are implemented on the FPGAs for temporarily storing the packets. The packet buffer with the greatest capacity is a FIFO that is implemented using an external DRAM. Figure 6.12 shows the logic diagram of the DRAM FIFO and neighboring small FIFOs of the MSH. To store 10-Gbps wire-speed traffic for 0.5 seconds, a packet buffer of greater than approximately 0.6 gigabytes is necessary. Such a large-capacity buffer cannot be implemented using the on-chip memory blocks of an FPGA; thus, external memory devices are required. The memory access pattern when buffering traffic data is sequential for both reading and writing data. Therefore, we can place traffic data in DRAM, which has long latency. The access latency of DRAM is mitigated by a small packet buffer implemented using on-chip block RAM. In this example, same data are stored in multiple memory types. Note that all of the DDR memory capacity in each port of the MSH is allocated to a single packet buffer, and there is no memory partitioning mechanism. This point simplifies and speeds up the circuit.

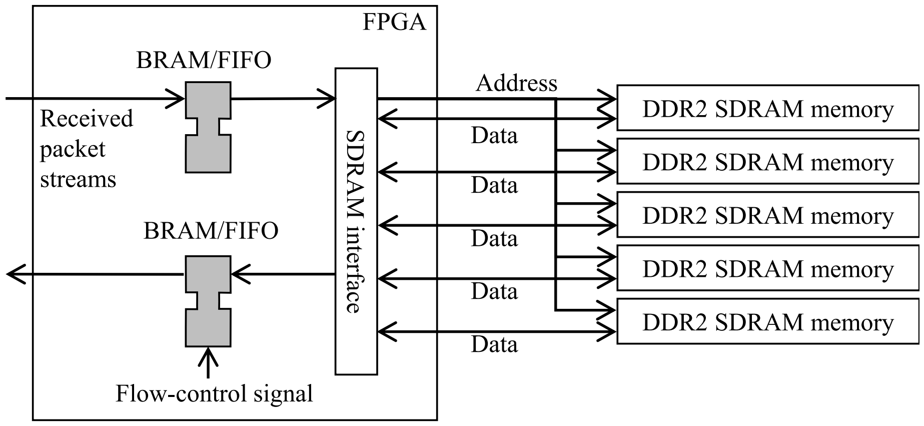


Figure 6.12: Logic diagram of a packet buffer in the downlink port FPGA

6.5.2 Accelerator for Chapter 4

In Chapter 4, we proposed an algorithm to scan a computer Go board with three-row-based processing. This algorithm is a good example of a hardware design based on the proposed design methodology. First, we discuss parallelism. The Triple Line based Playout for Go (TLPG) algorithm consists of multiple steps. Note that the board is scanned once in each step, and the logic of each step is connected via a FIFO for storing the board information in our hardware implementation. The FIFOs are similar to the pipeline registers in a general pipeline architecture, and we refer to the FIFOs as *pipeline FIFOs.* The information of one board moves from a single circuit for one step to another circuit for the next step. Thus, at some point, different playout generations are processed by each step circuit. We refer to this parallelism as *step-level pipeline parallelism.* Calculations for liberties and updates for stone colors are done for a single row of the board, and we refer to this parallelism as *row-level pipeline parallelism.* Calculations for a single row (nine or 19 stones) are performed in a single clock cycle, and we refer to this parallelism as *stone-level data parallelism.* The playout generation is a Monte Carlo computation, which has obvious task parallelism. As shown in Figure 6.13, the playout generation of the Monte Carlo tree search for Go has multiple parallelisms and can be parallelized at multiple levels. Each parallelism and logic design is patterned in our design principles. Here, we discuss the application-oriented data layout. In software implementation, information about the game board, including stone color and liberty number, is stored in array variables. This means that the information is stored in the main memory (DRAM). Software programmers can entrust optimizations, including copying the information to other memory regions such as cache and registers, to the compiler and CPU architecture. When the board information is stored in DRAM while using an FPGA-based approach, the FPGA-based accelerator loses ground to the CPU in terms of performance. On the other hand, an FPGA design that stores all the board information in Flip-Flops consumes many logical blocks, and it is difficult to synthesize such a design that satisfies both the timing constraints and resource limitations. As a result, the maximum operating clock frequency of the design becomes low. Therefore, in the TLPG, the board information is read from a FIFO line by line, and new board information is stored to a FIFO line by line. Since the FIFOs are implemented with built-in block RAM, consumption of logical block resources is reduced. Multi-level pipelining increases the consumption of pipeline registers. However, owing to the TLPG algorithm, the flip-flop consumption is proportional to the length of the side rather than the area of the board. The TLPG algorithm efficiently reduces flip-flop consumption. Thus, we have realized a high-performance FPGA-based accelerator with an effective data layout, including efficient use of pipeline registers and FIFOs.

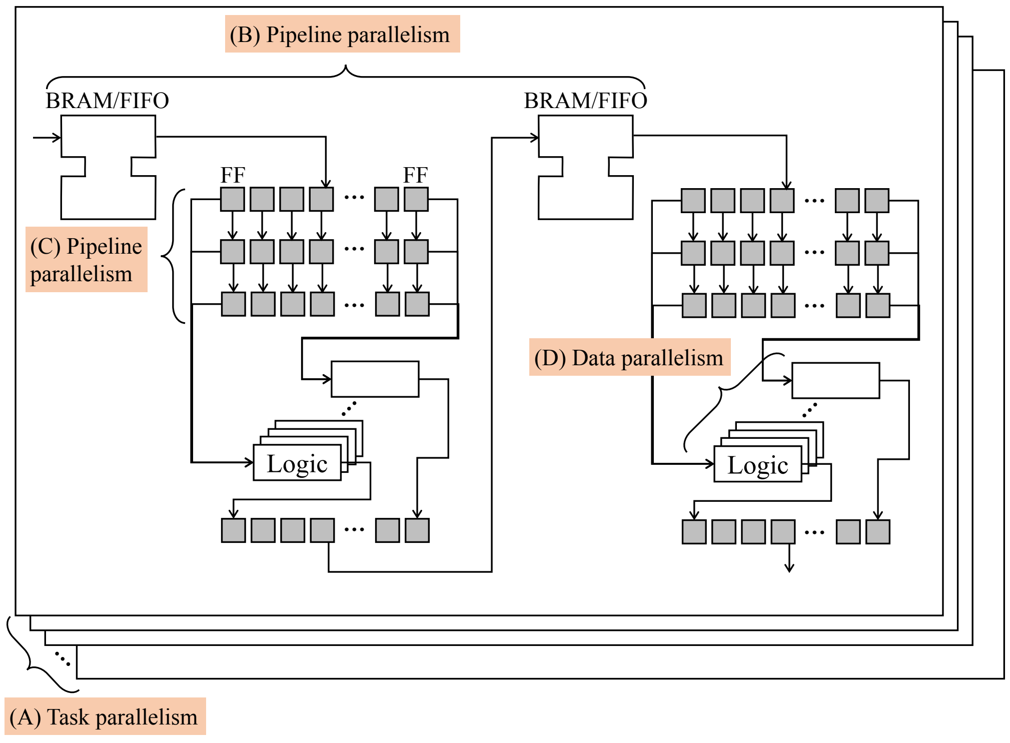


Figure 6.13: Logic diagram of the TLPG algorithm. (A) Monte Carlo playout generation has playout-level task parallelism. (B) The processing steps of the TLPG algorithm contain step-level pipeline parallelism. (C) Simple board processing has row-level pipeline parallelism. (D) Single row processing has stone-level data parallelism.

6.5.3 Accelerator for Chapter 5

Generally, since hardware approaches are weak relative to tree search, it is not a good idea to traverse a deep tree using the FPGA logic. In Chapter 5, we designed the JR-tree algorithm to avoid deep tree traversals. Recall that nodes corresponding to skyline points are always located at a depth of one in the JR-tree. Therefore, when enumerating skyline points, it is unnecessary to traverse nodes that are beyond a depth of one. Furthermore, we introduced a delay evaluation to the tree update process. When injecting a new node, we can terminate the traversal beyond a depth of one. First, we discuss parallelism. The most primitive calculation in the skyline computation is the dominance relation calculation, which has dimension-level data parallelism. Note that dominance relation calculations between the latest skyline points and a newly evaluated point are performed frequently, these calculations have data parallelism, and this parallelization gives a higher priority to accelerate the overall computation. Note that multiple tree update jobs can be processed simultaneously without resource conflict if the subtrees do not overlap. We can parallelize the tree update by dividing the process into multiple stages (i.e., pipeline parallelism). When a skyline node is ejected, a node move occurs once for each child of an ejected node, i.e., the ejection of a node with many children is very heavy, and pipelining is effective at speeding up this process. Our low-latency skyline computation accelerator (LSCA) exploits the benefits of multi-level parallelization. Here, we discuss the application-oriented data layout. As mentioned previously, the dominance relation calculations between the latest skyline points and a newly evaluated point have a higher parallelization priority. To realize this data parallelization, the vectors of all skyline points must be stored in registers and be available in the same clock cycle. Here, the data layout policy is determined based on the trade-off between flip-flop consumption and computation speed. On the other hand, the vectors of many other non-skyline points can be stored in a RAM with sufficient capacity. The data layout in this accelerator also follows our principle of placing frequently used data in registers. Note that continuous skyline computations can be accelerated by multi-level parallelization and the application-oriented data layout.

6.6 Performance Evaluation

In this section, we quantitatively evaluate accelerators based on the proposed design methodology. We compare the performance of the manual HDL code based on our design methodology and the auto-generated HDL code from the C-like code and directives generated using behavioral synthesis tools. The C-like code is optimized for a general processor, and they are rewritten to satisfy the constraints of the behavioral synthesis tools. The directives are manually inserted to provide various hints to the behavioral synthesis tools. We used Xilinx Vivado System Edition and Xilinx Vivado HLS to perform logical and behavioral synthesis, respectively. The accelerators were evaluated based on two metrics: (1) the number of clock cycles in the execution time for the same datasets in the logic simulator and (2) resource consumption. Figure 6.14 shows the evaluation flow. The target applications are the dominance relation calculation, decoding of a BCH code, and JR-tree based skyline algorithm. The target device is Xilinx Virtex UltraScale XCVU095-2FFVA2104E FPGA.

6.6.1 Dominance Relation Calculation

The evaluation of dominance relation is considered to be a primitive calculation in skyline computation. This calculation is performed many times; therefore, it is very important to accelerate it. Since this calculation consists of comparison instructions rather than arithmetic instructions, the FPU is not required even if the vector value is a floating-point number. Therefore, this calculation is suitable for an FPGA. The source code is shown in Appendix A.1. Since the HDL code generated by Vivado HLS has low readability, it is difficult for engineers to understand, analyze, and modify the generated circuits. Table 6.2 shows the benchmark results. Here, we used a set of all pairs of 100 pseudorandom vectors as a dataset. The benchmark results show that we can obtain a high-performance HDL code using the behavioral synthesis technique if a target computation is simple, such as the dominance relation calculation. Note that there is not much difference between the performance of the manually designed circuit and that

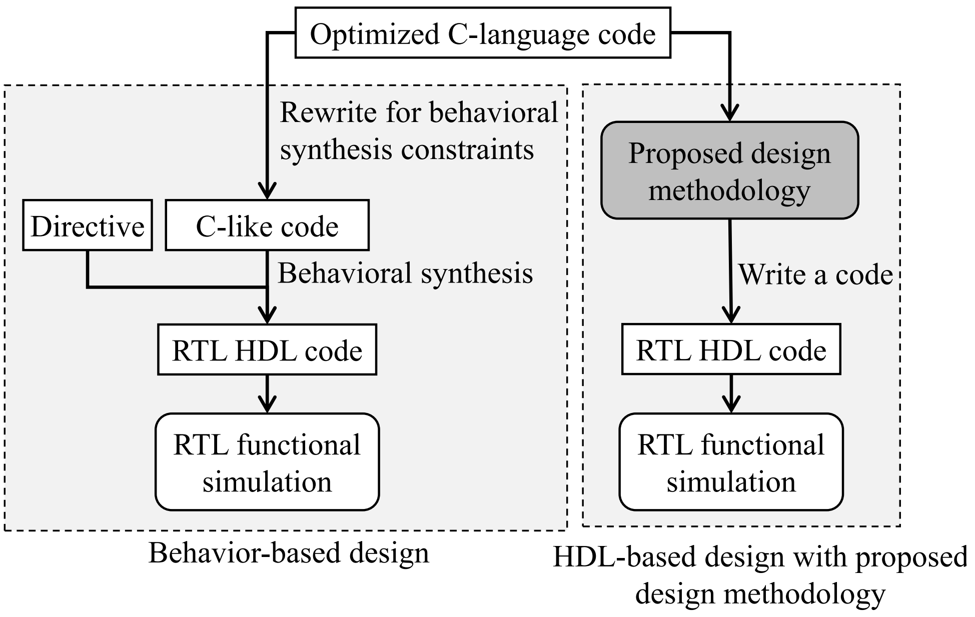

Optimized C-language code
Rewrite for behavioral synthesis constraints
Directive
C-like code
Behavioral synthesis
RTL HDL code
RTL functional simulation
Behavior-based design
Proposed design methodology
Write a code
RTL HDL code
RTL functional simulation
HDL-based design with proposed design methodology

Figure 6.14: Evaluation flow of the design methodology

of the automatically generated circuit because the synthesis tool can perform dimension-level loop-unrolling successfully.

Table 6.2: Synthesis and performance results of the dominance relation calculation

Approach	HLS		HDL (proposed)
Runtime [us]	624	250	250
Latency [clk]	2	3	1
Interval [clk]	1	1	1
Clock [MHz]	200	500	500
FF	243	184	160
LUT	596	429	83

6.6.2 BCH Decoding

For the error correction technique in a communication channel with errors, coding and decoding are important tasks that require high-throughput performance and low-latency processing. An encoder and a decoder are often implemented using hardware to achieve high-performance operation and low-power consumption. The Bose-Chaudhuri-Hocquenghem (BCH) code is a coding scheme that uses operations on the Galois field. The source code is shown in Appendix A.2. Table 6.3 depicts the benchmark results that were obtained using the decoder of the BCH code. The BCH code that was targeted by the implemented decoders employs a 15-bit code length, which includes seven information bits and eight check bits. This BCH code can correct two error bits per a single codeword. We generated 1000 codewords with 0- to 2-bit errors as a dataset. We inputted the dataset to the decoder and measured the runtime to decode all the codewords.

The latency number of the design generated by the HLS is greater than that of the manually coded HDL. This is because the HLS could not generate an efficient circuit to perform the arithmetic operations on the Galois field $GF(2^4)$, which is required by the decoder of the target BCH code. On the other hand, due to the appropriate insertion of the pipeline registers and placement of circuits for the parallelizable operations on the Galois field, the execution of the manually coded design is approximately 60 times faster than that of the HLS-based design.

Table 6.3: Synthesis and performance results of the BCH decoder

Approach	HLS	HDL (proposed)	
Runtime [us]	178	4	3
Latency [clk]	235	1	3
Interval [clk]	30	1	1
Clock [MHz]	170	250	375
Block RAM	1	0	0
Distributed RAM	0	1	0
FF	9873	47	62
LUT	10146	168	196

6.6.3 JR-tree Algorithm

We evaluated the JR-tree algorithm as a large-scale complicated application. We used the C and HDL code implemented and optimized in Chapter 5. The C code is optimized for a general processor. Note that it is difficult to auto-optimize a code as is in behavioral synthesis. Therefore, we rewrote the C code and manually inserted directives. The source code is shown in Appendix A.3. For details about the JR-tree algorithm, see sections 5.4 and 5.9. For the manually written logic circuits of the LSCA algorithm, see Section 5.9. Table 6.4 shows the benchmark results. The design of the manually coded HDL code was observed to be 2.3 times faster than that of the code generated by Vivado HLS because the loop unrolling failed, pipelined circuits could not be implemented, and the memory data layout was not optimized for the target application. Furthermore, the auto-generated design was able to utilize logic resources only at approximately 30% of that of the manually written design. In behavioral synthesis, we have no influence on the selection of a tradeoff between performance and resource consumption. These results demonstrate that it is difficult to design accelerators that are significantly faster than software implementations by behavioral synthesis and that we can design high-performance accelerators with manual HDL coding based on our design methodology.

6.7 Discussion

In this chapter, we have summarized the key points for the design of high-speed accelerators obtained by our studies as a design methodology. The proposed design methodology provides more in-depth guidelines to engineers who have learned basic HDL grammar. In recent years, with the introduction of behavioral

Table 6.4: Synthesis and performance results of the JR-tree algorithm

Approach	HLS	HDL (proposed)
Runtime [us]	1524	649
Clock [MHz]	200	200
Block RAM	2	14
DSP	2	0
FF	3655	10284
LUT	4186	11201

synthesis technology, the development period has been reduced only relative to verification processes. However, replacing a manual logic design with automated code conversion using behavioral synthesis tools has not been realized. It is expected that the proposed design methodology and design principles will be incorporated into a flowchart of the existing behavioral synthesis technology and tools. By improving the behavioral synthesis environments using the proposed design methodology, many users, including software programmers, will be able to design high-performance hardware accelerators more easily.

As mentioned in Section 1.1, the microarchitectural design of a computer system, such as general processors and accelerators, can be divided into (1) instruction interpretation, (2) execution, and (3) interconnection components (Figure 1.3). (1) **The instruction interpretation component** is a circuit that interprets a programmable software code loaded from memory to the device and that controls the execution components. In a general processor, the instruction interpretation component analyzes the dependencies of an instruction sequence and controls the procession of the pipeline stages. In a dedicated accelerator, a soft/hard processor core corresponds to this interpretation component. Note that an accelerator that does not include a processor core does not have an interpretation component. (2) **The execution component** comprises a circuit for computing the target calculations. In a general processor, this component includes an arithmetic and logical unit (ALU) and a floating point-unit (FPU), whereas in a dedicated accelerator, this component includes a main circuit for the target algorithm. (3) **The interconnection component** consists of data paths that connect logic modules and a circuit to control the data flow. This component includes a connection between the interpretation and execution components, a connection between the cores in a multicore processor, and an off-chip connection between devices.

Note that these three components compete for logical and wiring resources. In the development of general processors, researchers and developers have proposed many conventional methods for each component, and a combination of efficient individual technologies has led to greater processor performance and efficiency. However, relative to the development of specialized hardware accelerators, such versatile methods and approaches have not been developed. In this chapter, we have established a design methodology for the execution component.

6.8 Summary

The structural design in hardware development has a high degree of freedom compared to that of software development. In this chapter, we have established a design methodology for the HDL-based design method. In the proposed methodology, schemes for parallelizing the computations at multiple levels are patterned. This methodology provides a feasible data layout method for a given FPGA device environment. In this chapter, we have shown how two concepts, i.e., multi-level parallelization and the application-oriented data layout, were utilized in our studies (Chapters 3, 4, and 5). Both a single move in computer Go and one query in continuous skyline computations are operations that can drastically change the internal state of the calculations, and such operations are difficult to accelerate using an FPGA. The data layout policy of using suitable memory blocks and devices based on the data access pattern and frequency greatly contributes to the performance improvement of the accelerators. However, this knowledge is not sufficiently considered in the code conversion process of the currently available behavioral synthesis tool. By incorporating the proposed design methodology, it is expected that code conversion will improve such that high-performance RTL designs can be generated. Our evaluations illustrate that the performance of the manually written HDL code, which was based on the proposed design methodology, is better than that of the HDL code generated by the behavioral synthesis tool. Note that this behavioral synthesis technology is currently on a learning curve.

The proposed methodology helps designers who have understood the grammar of the HDL and the basic features of the FPGA to understand important concepts that can help in designing high-performance accelerators. The proposed methodology suggests various points that decide the performance of an accelerator in order to achieve a performance that exceeds the performance of the software implementation on a CPU. The proposed methodology complements the gap between gate-level design guidelines and design guidelines for an abstract algorithm (including pseudo-code). In recent years, a System on Chip (SoC) FPGA, which contains processor cores, has been in extensive use. In design using the SoC FPGA, the proposed methodology also exhibits its effect in implementing a circuit of the function that affects the performance.

Chapter 7

Conclusions

In this chapter, we review our studies, which are based on FPGA-based accelerators, and our design methodology. We also discuss future research prospects based on our contributions.

7.1 Summary of Contributions

In Chapter 3, we described our research into improving the performance of parallel TCP streams. We proposed the merging stream harmonizer (MSH) hardware mechanism, which directly merges parallel TCP streams. Merging parallel TCP streams without packet loss improves the performance of TCP communication. In addition, we designed and produced the MaSTER-1 FPGA-based network testbed and implemented the MSH on MaSTER-1. The MSH mitigates traffic bursts in long-distance networks. Network congestion caused by bursty traffic results in unnecessary packet loss. However, packet scheduling of the MSH avoids such packet losses. We evaluated the performance of the MSH, and the results demonstrate that it increases the throughput performance of parallel TCP communications in pseudo high-latency high-bandwidth networks. The MSH achieved parallel TCP streams that equally occupy 10-Gbps wire-speed network bandwidth. In real-world long-distance fat-pipe networks with 9.2 Gbps bandwidth, the MSH realized four TCP streams that equally occupy 8.0 Gbps bandwidth with a packet pacing function with an 8.0 Gbps limitation. The MSH can realize stable data transmission using parallel TCP streams. Note that hardware-level packet scheduling techniques are essential in higher-bandwidth computer networks. Many FPGA-based WAN accelerators have been studied to increase the throughput of data transmission. However, such existing studies did not comply with the specifications of TCP, UDP, or other protocol standards and required dedicated software and hardware inserted at the network endpoints. Communication over proprietary protocols that do not consider RTT fairness or TCP friendliness can reduce the performance of background traffic. Our approach is TCP-compliant and maintains fairness and friendliness. In the future, our method is expected to become an essential solution for inter-cluster communication that generates large-scale end-to-end traffic.

In addition, we also described performance analysis of the TCP, which is the most widely and commonly used network communication protocol. We analyzed the relationship between RTT and the buffer size of the path-bottleneck switch. We clarified the importance of a large buffer at the merge point of long-distance fat-pipe networks, and we investigated the influence of packet buffer size on the performance of parallel TCP streams. As a result, it became apparent that the narrower the bandwidth of an uplink port of path-bottleneck switch compared to the total bandwidth of the end hosts, a larger packet buffer is required by the path-bottleneck switch to prevent packet loss. We demonstrated that loss-based TCP congestion control algorithms can avoid packet loss at the path-bottleneck switch if the buffer size is sufficient. We also showed equations to calculate the buffer amount required to avoid packet loss. In computer networks, intermediate switches installed at a high-bandwidth network merge point must have a large buffer, similar to that in MaSTER-1.

Our analysis was realized using a hand-made testbed with FPGAs, which can perform real-time processing and have low development costs. Stream processing at each port was distributed to each dedicated FPGA, and packet scheduling at an uplink port was parallelized on a single FPGA. Packet datagrams handled by the MSH are accessed sequentially, and they require a large amount of memory. As a result, we arranged datagrams in DRAM by considering the number of ports and required buffer size.

In Chapter 4, we described our study into speeding up the Monte Carlo Go playout generation. Monte Carlo tree search is used in a stronger state-of-the-art computer Go player. A game tree for a Go game has a large search space; thus, it has been difficult to traverse such a tree and implement a player on hardware. A playout requires the processing of many moves based on the complicated rules of Go. In this study, we proposed a hardware algorithm for playout generation. We simplified the complicated rules of Go by focusing on the relationships among adjacent stones, and we focused on four adjacent neighbor stones and parallelized the calculations for a row of stones. Here, the key idea is to maintain information about three rows of stones and process a single row in a single cycle. Note that a naive playout generation implementation requires a very wide memory bandwidth. We stored information about the three rows, which comprise one ongoing row and two adjacent rows, in registers and stored the complete board information in FIFOs implemented by RAM. This application-oriented data layout led to more efficient memory usage and realized acceleration of the playout generation. The TLPG algorithm is easy to implement on FPGA. Our FPGA implementation of the TLPG achieved a playout generation speed of 40,649 playouts per second on a 9×9 grid board and 4,668 playouts per second on a 19×19 grid board.

In the TLPG algorithm, the playout, scanning the entire board, and updating a single row of stones were parallelized. Note that these processes have different types of parallelism, such as data, task, and pipeline parallelism. Here, we realized high parallelism by employing logic circuits suitable for each parallelism. Stone and liberty information of three rows used in the calculation is stored in the registers. On the other hand, entire board information is stored in the FIFOs. These techniques realized high parallelism of the TLPG algorithm.

In Chapter 5, we described our research into speeding up a continuous skyline computation. We accelerated the computation of skylines, which represent a set of points in multi-dimensional space, where points are activated and deactivated dynamically. We discussed the reduction of potentially needless comparison operations as a difficult point in continuous skyline computations. We proposed the BJR-tree to speed up the continuous skyline computation by storing the results of previous comparison operations. The BJR-tree is a new tree structure that can intuitively express dominance relations and provides updating algorithms that schedule comparisons to reduce unnecessary operations. Furthermore, we proposed the ND-cache mechanism, which efficiently saves the results of previous comparisons. The results of a performance evaluation demonstrate that the BJR-tree can reduce execution time compared to the state-of-the-art continuous skyline computation algorithm. The results also showed that ND-cache can reduce the number of invoked comparison operations on pathological datasets and generates little overhead with synthesized datasets. The BJR-tree software implementation overcame the implementation of existing software algorithms. Furthermore, the BJR-tree (JR-tree) was designed and optimized for an FPGA implementation. An FPGA implementation of the LSCA hardware algorithm based on the JR-tree reduced the execution time of the continuous skyline computation significantly.

In the LSCA, a single dominance relation operation, the computation of all dominance relations between the current skyline points and a newly injected point, and the injection and ejection operations of the JR-tree were parallelized. These computations have data and pipeline parallelism. Here, we also realized high parallelism by employing logic circuits suitable for each parallelism. We stored information about skyline points, which are repeatedly used for dominance relation calculations, in registers, and we stored the information of all points in Block RAM. When a skyline point is deactivated, the dominance relation calculations between the child nodes of an ejected node and the rest of the skyline nodes are invoked. These heavy calculations were parallelizable using an application-oriented data layout.

In this thesis, we have demonstrated that multi-level parallelization and an application-oriented data layout are essential for high-performance logic design through these typical studies. A single move in the Monte Carlo Playout Generation and a single query in the continuous skyline computation have the potential to dynamically change the internal states of the computation (e.g., the values of variables or registers). Thus, these applications have been considered difficult for FPGA-based acceleration. We improved the performance of these applications using a data layout based on the data access pattern and frequency. We focused on multi-level parallelization (to parallelize computations at multiple granularities with multiple types of parallelism) and the application-oriented data layout (to select a memory type to store the internal data in execution of applications) for the logic design of the FPGA-based acceleration. Our design methodology is a guideline for designing a high-performance FPGA-based accelerator and provides a method to enhance behavioral synthesis technologies.

7.2 Future Outlook

The TCP is widely used as a communication protocol that guarantees reliable data transmission. TCP functions include session management, a TCP state machine, acknowledgment packets, and a sequence number in the TCP header section of packets. However, owing to the popularization of long-distance high-bandwidth networks, TCP features, which were advantageous when the protocol was initially designed, have been revealed as disadvantages. Although many alternative protocols have been proposed, a definitive next-generation protocol has not been determined yet. Extended TCP specifications have been proposed and standardized to extend the life of the TCP. In this study, we demonstrated that the TCP can be employed to utilize network bandwidth efficiently even in long-distance fat-pipe networks (LFNs). We implemented an FPGA-based accelerator and evaluated its performance improvement. Issues that remain relative to the use of the TCP in LFNs are that the fact that congestion control algorithms are not suitable to such networks and that the protocol specification limits the bandwidth delay product (BDP) to 1 gigabytes. We have already begun analyzing buffering and scheduling in path-bottleneck switches using six well-known congestion control algorithms, i.e., BIC-TCP, CUBIC-TCP, TCP Reno, high-speed

TCP, H-TCP, and TCP Westwood. Furthermore, we have proposed proprietary extensions for TCP to resolve the BDP limitation problem and evaluated the throughput performance of our modified TCP stack implementation in real-world long-distance networks. It is expected that this will provide helpful insights for the design of a next-generation data transfer protocol.

Recently, Alpha Go, a computer-based Go player, has defeated a professional human Go player [109]. Hardware acceleration has played an important role in Alpha Go's increasing effectiveness [110]. In addition, hardware acceleration is receiving increasing attention in various AI applications. Many AI applications, such as convolutional neural networks, require an extraordinary amount of computing resources. We have accelerated heavy computation of playout generation using a hardware algorithm adapted to complicated Go rules. Application-oriented FPGA-based accelerators are expected to achieve high performance and efficiency in other AI fields.

Skyline computation was initially proposed to extract interesting entries from a large database. We introduced fast cell identification and selection on flow cytometry as a new application of continuous skyline computation. A future biological analysis will clarify the effectiveness of cells extracted by skyline computation. Since our flow cytometry, i.e., the Serendipiter, targets a wide variety of cells, various other schemes are extraction method candidates. For other extraction schemes, FPGA-based acceleration will be a practical approach for the low-latency cell identification system in Serendipiter.

In this research, we compared the performance of accelerators which are manually implemented and accelerators implemented with behavioral synthesis for evaluation of the proposed design methodology. Additionally, we will be able to verify the effectiveness of the proposed methodology in actual educational sites. We show an example of a verification method as follows. We divide students of courses for mastering hardware design in higher education facility into two groups randomly. They all have learned the HDL grammars and basic coding techniques. We introduce the proposed methodology to students belonging to one group, and textbooks including existing design guidelines to students belonging to the other group. Then, we give the students of the two groups the same task of implementing applications like those shown in Section 6.6. We will be able to the effectiveness of the proposed methodology in the implementation of high-performance accelerators by comparing the number of the clock cycles of latency and interval and runtime of implemented accelerators. In this evaluation method, because a circuit of the accelerators designed by the students is evident and analyzable, we can obtain statistical information such as what type of parallelized circuit was designed, what kind of memory was utilized, and what kind of data was laid out. Such information is useful for analyzing the effectiveness of parallelization and data layout.

Microarchitecture comprises instruction interpretation, execution, and interconnection components (Figure 1.3). In a CPU example, instruction interpretation, the execution, and interconnection components correspond to a control unit (CU), an ALU/FPU, and data buses, respectively. In this thesis, we have presented a design methodology for the execution component. To design a large-scale massively parallelized accelerator, it is essential to establish design methodologies for the instruction interpretation and interconnection components of application-specific circuits. However, this remains an unresolved problem. As discussed in Chapter 2, there are hardware accelerators with and without instruction interpretation components implemented by processor IP cores. It is difficult to determine the following two points; (1) whether an accelerator should have an instruction interpretation component and (2) how design the instruction interpretation component should have if the accelerator should have it. An effective design methodology for the instruction interpretation component would contribute to resolving such uncertainties.

The clock frequency of FPGAs tends to be less than that of CPUs and ASICs. In addition, there is a trade-off between memory capacity and access latency in the memory environment of FPGA devices. These disadvantages and difficulties have been bottlenecks for FPGA-based accelerators to achieve high performance. However, new memory technologies for FPGAs are expected in the future. For example, Ultra RAM is an on-chip memory block that complements the gap between capacity and latency relative to Block RAM and external memory chips. The Bandwidth Engine and Hybrid Memory Cube (HMC) are off-chip memory devices that have serial I/O interfaces, and High Bandwidth Memory 2 (HBM2) has a wide parallel I/O interface. By increasing the degree of integration of FPGAs, we expect greater freedom relative to multi-level parallelization. With memory device diversification, we also expect greater design freedom relative to data layouts. Beyond the specific fields discussed in the thesis, FPGA-based accelerators are expected to be used in various other applications in which memory bandwidth is a performance bottleneck. Thus, we consider that, in conjunction with the increasing popularization of FPGA-based accelerators, the contributions of this thesis will form a basis for future logic design methods.

References

[1] Yuichiro Ajima, Shinji Sumimoto, and Toshiyuki Shimizu. Tofu: A 6D Mesh/Torus Interconnect for Exascale Computers. *Computer*, 42(11):36–40, November 2009.

[2] Doug Amos, Austin Lesea, and Ren Richter. *FPGA-based Prototyping Methodology Manual: Best Practices in Design-for-Prototyping*. Synopsys Press, USA, 2011.

[3] J. H. Anderson and F. N. Najm. A novel low-power FPGA routing switch. In *Proceedings of the IEEE 2004 Custom Integrated Circuits Conference (IEEE Cat. No.04CH37571)*, pages 719–722, Oct 2004.

[4] Anue Systems (ixia). Network Emulator. `https://www.ixiacom.com/products/network-emulator-ii`.

[5] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multi-armed bandit problem. *Machine Learning*, 47:235–256, 2002.

[6] Ilaria Bartolini, Paolo Ciaccia, and Marco Patella. Efficient Sort-based Skyline Evaluation. *ACM Transactions on Database Systems (TODS)*, 33(4):31:1–31:49, December 2008.

[7] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, Sep 1972.

[8] BD Biosciences. Cell Sorters. `http://www.bdbiosciences.com/us/instruments/research/cell-sorters/c/744762`.

[9] Norbert Beckmann, Hans Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, SIGMOD '90, pages 322–331, New York, NY, USA, 1990. ACM.

[10] Jon Louis Bentley, Hsiang Tsung Kung, Mario Schkolnick, and Clark D Thompson. On the average number of maxima in a set of vectors and applications. *Journal of the ACM (JACM)*, 25(4):536–543, 1978.

[11] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. The X-tree: An Index Structure for High-Dimensional Data. In *Proceedings of the 22th International Conference on Very Large Data Bases*, VLDB '96, pages 28–39, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

[12] Florent Berthelot, Fabienne Nouvel, and Dominique Houzet. Partial and Dynamic Reconfiguration of FPGAs: A Top Down Design Methodology for an Automatic Implementation. In *Proceedings of the 20th International*

Conference on Parallel and Distributed Processing, IPDPS'06, pages 207–, Washington, DC, USA, 2006. IEEE Computer Society.

[13] Michaela Blott, Ling Liu, Kimon Karras, and Kees Vissers. Scaling out to a Single-node 80Gbps Memcached Server with 40Terabytes of Memory. In *Proceedings of the 7th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'15, pages 8–8, Berkeley, CA, USA, 2015. USENIX Association.

[14] Kenneth S. Bøgh, Ira Assent, and Matteo Magnani. Efficient GPU-based Skyline Computation. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware*, DaMoN '13, pages 5:1–5:6, New York, NY, USA, 2013. ACM.

[15] Kenneth S. Bøgh, Sean Chester, and Ira Assent. Work-efficient Parallel Skyline Computation for the GPU. *Proceedings of the VLDB Endowment*, 8(9):962–973, May 2015.

[16] Christian Böhm and Hans-Peter Kriegel. Determining the convex hull in large multidimensional databases. In *Data Warehousing and Knowledge Discovery*, pages 294–306. Springer, 2001.

[17] Cristiana Bolchini, Antonio Miele, and Chiara Sandionigi. A Novel Design Methodology for Implementing Reliability-Aware Systems on SRAM-Based FPGAs. *IEEE Transactions on Computers*, 60(12):1744–1758, Dec 2011.

[18] S. Börzsönyi, D. Kossmann, and K. Stocker. The Skyline operator. In *Proceedings 17th International Conference on Data Engineering*, pages 421–430, 2001.

[19] Markus M. Breunig, Hans-Peter Kriegel, Raymond T. Ng, and Jörg Sander. LOF: Identifying Density-based Local Outliers. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, pages 93–104, New York, NY, USA, 2000. ACM.

[20] T. M. Brewer. Instruction Set Innovations for the Convey HC-1 Computer. *IEEE Micro*, 30(2):70–79, March 2010.

[21] Brocade. BigIron RX Series. `http://www.foundrynet.com/products/`.

[22] Bernd Brügmann. Monte Carlo Go, 1993.

[23] Christian Buchta. On the average number of maxima in a set of vectors. *Information Processing Letters*, 33(2):63 – 65, 1989.

[24] Murray Campbell, A. Joseph Hoane, Jr., and Feng-hsiung Hsu. Deep Blue. *Artif. Intell.*, 134(1-2):57–83, January 2002.

[25] Anne E. Carpenter, Thouis R. Jones, Michael R. Lamprecht, Colin Clarke, In Han Kang, Ola Friman, David A. Guertin, Joo Han Chang, Robert A. Lindquist, Jason Moffat, Polina Golland, and David M. Sabatini. CellProfiler: image analysis software for identifying and quantifying cell phenotypes. *Genome Biology*, 7(10):R100, Oct 2006.

[26] Chee-Yong Chan, H. V. Jagadish, Kian-Lee Tan, Anthony K. H. Tung, and Zhenjie Zhang. Finding K-dominant Skylines in High Dimensional Space. In *Proceedings of the 2006 ACM SIGMOD International Conference on*

Management of Data, SIGMOD '06, pages 503–514, New York, NY, USA, 2006. ACM.

[27] Chee-Yong Chan, HV Jagadish, Kian-Lee Tan, Anthony KH Tung, and Zhenjie Zhang. Finding k-dominant skylines in high dimensional space. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 503–514. ACM, 2006.

[28] Chee-Yong Chan, HV Jagadish, Kian-Lee Tan, Anthony KH Tung, and Zhenjie Zhang. On high dimensional skylines. In *Advances in Database Technology-EDBT 2006*, pages 478–495. Springer, 2006.

[29] L. Charaabi, E. Monmasson, and I. Slama-Belkhodja. Presentation of an efficient design methodology for FPGA implementation of control systems. Application to the design of an antiwindup PI controller. In *IEEE 2002 28th Annual Conference of the Industrial Electronics Society. IECON 02*, volume 3, pages 1942–1947, Nov 2002.

[30] Wonik Choi, Ling Liu, and Boseon Yu. Multi-criteria decision making with skyline computation. In *Information Reuse and Integration (IRI), 2012 IEEE 13th International Conference on*, pages 316–323. IEEE, 2012.

[31] Jan Chomicki, Parke Godfrey, Jarek Gryz, and Dongming Liang. Skyline with presorting. In *Proceedings of 19th International Conference on Data Engineering*, pages 717–719. IEEE, 2003.

[32] Eric Chung et al. Accelerating Persistent Neural Networks at Datacenter Scale. In *Hot Chips 2017*, 2017.

[33] SFF Committee et al. 10 Gigabit Small Form Factor Pluggable Module. *INF-8077i, Revision*, 4, 2005.

[34] Chelsio Communications. High Performance 10GbE Storage Accelerator. `http://www.chelsio.com/`, 2007.

[35] Rémi Coulom. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In *Proceedings of the 5th International Conference on Computers and Games*, CG'06, pages 72–83, Berlin, Heidelberg, 2007. Springer-Verlag.

[36] CYTO. CYTO2017 Image Analysis Challenge. `http://cytoconference.org/2017/Home.aspx`, 2017.

[37] Rainer Dömer, Jianwen Zhu, and Daniel D Gajski. The SpecC Language Reference Manual. In *SpecC Technology Open Consortium.* Citeseer, 1998.

[38] Hewlett Packard Enterprise. ProCurve. `https://www.hpe.com/jp/ja/networking.html`.

[39] Raphael A. Finkel and Jon Louis Bentley. Quad trees a data structure for retrieval on composite keys. *Acta informatica*, 4(1):1–9, 1974.

[40] Peter L. Flake and Simon J. Davidmann. Superlog, a Unified Design Language for System-on-chip. In *Proceedings of the 2000 Asia and South Pacific Design Automation Conference*, ASP-DAC '00, pages 583–586, New York, NY, USA, 2000. ACM.

[41] FORCE10 (DELL). Networking Devices. http://www.dell.com/en-us/work/shop/cty/sc/networking-products.

[42] Katerina Fotiadou and Evaggelia Pitoura. BITPEER: continuous subspace skyline computation with distributed bitmap indexes. In *Proceedings of the 2008 international workshop on Data management in peer-to-peer systems*, pages 35–42. ACM, 2008.

[43] Fujitsu. Electronic Devices. http://www.fujitsu.com/us/products/devices/.

[44] Fujitsu. XG800 Ethernet Switch. http://www.fujitsu.com/us/about/resources/news/press-releases/2004/fla-20041108-1.html, 2004.

[45] Daniel D Gajski, Jianwen Zhu, Rainer Dömer, Andreas Gerstlauer, and Shuqing Zhao. *SpecC: Specification language and methodology*. Springer Science & Business Media, 2012.

[46] Philippe Garrault and Brian Philofsky. *HDL Coding Practices to Accelerate Design Performance*. Xilinx, 2006.

[47] Sylvain Gelly and Yizao Wang. Exploration exploitation in Go: UCT for Monte-Carlo Go. In *NIPS: Neural Information Processing Systems Conference On-line trading of Exploration and Exploitation Workshop*, Canada, December 2006.

[48] GNU Project. GNU Go. http://www.gnu.org/software/gnugo/.

[49] Parke Godfrey, Ryan Shipley, and Jarek Gryz. Algorithms and analyses for maximal vector computation. *The VLDB Journal-The International Journal on Very Large Data Bases*, 16(1):5–28, 2007.

[50] Ronald L. Graham. An efficient algorith for determining the convex hull of a finite planar set. *Information processing letters*, 1(4):132–133, 1972.

[51] IEEE System Verilog Working Group et al. IEEE Standard for SystemVerilog C Unified Hardware Design, Specification, and Verification (IEEE Std 1800–2005), 2005.

[52] Baoshan Guo, Cheng Lei, Hirofumi Kobayashi, Takuro Ito, Yaxiaer Yalikun, Yiyue Jiang, Yo Tanaka, Yasuyuki Ozeki, and Keisuke Goda. High-throughput, label-free, single-cell, microalgal lipid screening by machine-learning-equipped optofluidic time-stretch quantitative phase microscopy. *Cytometry Part A*, 91(5):494–502, 2017.

[53] Antonin Guttman. R-trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, pages 47–57, New York, NY, USA, 1984. ACM.

[54] K. Hill, S. Craciun, A. George, and H. Lam. Comparative analysis of OpenCL vs. HDL with image-processing kernels on Stratix-V FPGA. In *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 189–193, July 2015.

[55] K. Hiraki, M. Inaba, H. Tezuka, H. Tomari, K. Koizumi, and S. Kondo. All-IP-Ethernet architecture for real-time sensor-fusion processing. In *Proc. SPIE, High-Speed Biomedical Imaging and Spectroscopy: Toward Big Data Instrumentation and Management*, volume 9720, page 97200D, 2016.

[56] Hiroaki Hirata, Kozo Kimura, Satoshi Nagamine, Yoshiyuki Mochizuki, Akio Nishimura, Yoshimori Nakase, and Teiji Nishizawa. An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ISCA '92, pages 136–145, New York, NY, USA, 1992. ACM.

[57] Zhiyong Huang, Hua Lu, Beng Chin Ooi, and A. K. H. Tung. Continuous Skyline Queries for Moving Objects. *IEEE Transactions on Knowledge and Data Engineering*, 18(12):1645–1658, Dec 2006.

[58] Motorola Semiconductor Products Inc. *The Semiconductor Data Book*. Motorola Incorporated, Semiconductor products division, 1966.

[59] Pico Computing Inc. Pico Computing SC5 Interconnect Architecture. Technical report, Pico Computing Inc., 2014.

[60] Open SystemC Initiative et al. IEEE standard SystemC language reference manual (IEEE1666–2005). *IEEE Computer Society*, pages 1666–2005, 2006.

[61] Iperf.fr. iPerf - The ultimate speed test tool for TCP, UDP and SCTP. `https://iperf.fr/`.

[62] IXIA. Optixia. `http://ixiacom.com/`.

[63] Q. Jia and H. Zhou. Tuning Stencil codes in OpenCL for FPGAs. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pages 249–256, Oct 2016.

[64] Yiyue Jiang, Cheng Lei, Atsushi Yasumoto, Hirofumi Kobayashi, Yuri Aisaka, Takuro Ito, Baoshan Guo, Nao Nitta, Natsumaro Kutsuna, Yasuyuki Ozeki, et al. Label-free detection of aggregated platelets in blood by machine-learning-aided optofluidic time-stretch microscopy. *Lab on a Chip*, 17(14):2426–2434, 2017.

[65] Daniel A. Jiménez and Calvin Lin. Dynamic Branch Prediction with Perceptrons. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, HPCA '01, pages 197–, Washington, DC, USA, 2001. IEEE Computer Society.

[66] Norman P. Jouppi et al. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 1–12, New York, NY, USA, 2017. ACM.

[67] Hiroyuki Kamezawa, Makoto Nakamura, Junji Tamatsukuri, Nao Aoshima, Mary Inaba, and Kei Hiraki. Inter-Layer Coordination for Parallel TCP Streams on Long Fat Pipe Networks. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, SC '04, pages 24–, Washington, DC, USA, 2004. IEEE Computer Society.

[68] O. Karatsu. UDL/I standardization effort another approach to HDL standard. In *Euro ASIC '91*, pages 388–393, May 1991.

[69] Norio Katayama and Shin'ichi Satoh. The SR-tree: An index structure for high-dimensional nearest neighbor queries. In *ACM SIGMOD Record*, volume 26(2), pages 369–380. ACM, 1997.

[70] R. E. Kessler and J. L. Schwarzmeier. Cray T3D: a new dimension for Cray Research. In *Digest of Papers. Compcon Spring*, pages 176–182, Feb 1993.

[71] You Jung Kim and Jignesh M Patel. Rethinking Choices for Multi-dimensional Point Indexing: Making the Case for the Often Ignored Quadtree. In *CIDR*, pages 281–291, 2007.

[72] Dirk Koch, Frank Hannig, and Daniel Ziener. *FPGAs for Software Programmers*. Springer Publishing Company, Incorporated, 1st edition, 2016.

[73] Levente Kocsis and Csaba Szepesvári. *Bandit Based Monte-Carlo Planning*, pages 282–293. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[74] Yuetsu Kodama, Tomohiro Kudoh, and Toshiyuki Shimizu. GtrcNET-10: Network test-bed supporting 10gbe – its configuration and preliminary evaluation. In *Summer United Workshops on Parallel, Distributed and Cooperative Processing (SWoPP2005) (in Japanese)*. IPSJ, Aug 2005.

[75] Maria Kontaki, Apostolos N Papadopoulos, and Yannis Manolopoulos. Continuous k-dominant skyline computation on multidimensional data streams. In *Proceedings of the 2008 ACM symposium on Applied computing*, pages 956–960. ACM, 2008.

[76] Donald Kossmann, Frank Ramsak, and Steffen Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 275–286. VLDB Endowment, 2002.

[77] Ravi Kanth V Kothuri, Siva Ravada, and Daniel Abugov. Quadtree and R-tree indexes in oracle spatial: a comparison using GIS data. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 546–557. ACM, 2002.

[78] Hans-Peter Kriegel, Matthias S hubert, and Arthur Zimek. Angle-based Outlier Detection in High-dimensional Data. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '08, pages 444–452, New York, NY, USA, 2008. ACM.

[79] S. Kumar, A. Jantsch, J. P. Soininen, M. Forsell, M. Millberg, J. Oberg, K. Tiensyrja, and A. Hemani. A network on chip architecture and design methodology. In *Proceedings IEEE Computer Society Annual Symposium on VLSI. New Paradigms for VLSI Systems Design. ISVLSI 2002*, pages 105–112, 2002.

[80] Hsiang Tsung Kung, Fabrizio Luccio, and Franco P Preparata. On finding the maxima of a set of vectors. *Journal of the ACM (JACM)*, 22(4):469–476, 1975.

[81] C. S. Lee, M. H. Wang, T. P. Hong, G. Chaslot, J. B. Hoock, A. Rimmel, O. Teytaud, and Y. H. Kuo. A novel ontology for computer go knowledge management. In *2009 IEEE International Conference on Fuzzy Systems*, pages 1056–1061, Aug 2009.

[82] Jongwuk Lee and Seung Won Hwang. BSkyTree: scalable skyline computation using a balanced pivot selection. In *Proceedings of the 13th International Conference on Extending Database Technology*, pages 195–206. ACM, 2010.

[83] Mu-Woong Lee and Seung-won Hwang. Continuous Skylining on Volatile Moving Data. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, ICDE '09, pages 1568–1575, Washington, DC, USA, 2009. IEEE Computer Society.

[84] Stian Liknes, Akrivi Vlachou, Christos Doulkeridis, and Kjetil Nørvåg. AP-Skyline: improved skyline computation for multicore architectures. In *Database Systems for Advanced Applications*, pages 312–326. Springer, 2014.

[85] Xuemin Lin, Yidong Yuan, Wei Wang, and Hongjun Lu. Stabbing the Sky: Efficient Skyline Computation over Sliding Windows. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE '05, pages 502–513, Washington, DC, USA, 2005. IEEE Computer Society.

[86] P. Lysaght, B. Blodget, J. Mason, J. Young, and B. Bridgford. Invited Paper: Enhanced Architectures, Design Methodologies and CAD Tools for Dynamic Reconfiguration of Xilinx FPGAs. In *2006 International Conference on Field Programmable Logic and Applications*, pages 1–6, Aug 2006.

[87] W. Stevens M. Allman, V. Paxson. RFC 2581 - TCP Congestion Control, Apr 1999.

[88] Junichiro Makino and Hiroshi Daisaka. GRAPE-8: An Accelerator for Gravitational N-body Simulation with 20.5Gflops/W Performance. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 104:1–104:10, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[89] Clive Maxfield. *The Design Warrior's Guide to FPGAs: Devices, Tools and Flows.* Newnes, Newton, MA, USA, 1st edition, 2004.

[90] Scott Mcfarling. Combining Branch Predictors. *WRL Technical Note, TN-36, Digital Equipment Corporation*, 1993.

[91] E. Monmasson and M. N. Cirstea. FPGA Design Methodology for Industrial Control Systems – A Review. *IEEE Transactions on Industrial Electronics*, 54(4):1824–1842, Aug 2007.

[92] M. Morse, J. M. Patel, and W. I. Grosky. Efficient Continuous Skyline Computation. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 108–108, April 2006.

[93] Intel Nervana. Inside Artificial Intelligence, Next-level computing powered by Intel Nervana. `https://www.intelnervana.com/`.

[94] R. Nikhil. Bluespec System Verilog: Efficient, Correct RTL from High Level Specifications. In *Proceedings of the Second ACM/IEEE International Conference on Formal Methods and Models for Co-Design*, MEMOCODE '04, pages 69–70, Washington, DC, USA, 2004. IEEE Computer Society.

[95] M. Oikawa, D. Hiyama, R. Hirayama, S. Hasegawa, Y. Endo, T. Sugie, N. Tsumura, M. Kuroshima, M. Maki, G. Okada, C. Lei, Y. Ozeki, K. Goda, and T. Shimobaba. A computational approach to real-time image processing for serial time-encoded amplified microscopy. In *Proc. SPIE, High-Speed Biomedical Imaging and Spectroscopy: Toward Big Data Instrumentation and Management*, volume 9720, page 97200E, 2016.

[96] Jian Ouyang et al. XPU: A programmable FPGA Accelerator for diverse workloads. In *Hot Chips 2017*, 2017.

[97] Karthi Palanisamy and Rich Chiu. *High-Performance DDR2 SDRAM Interface in Virtex-5 Devices.* Xilinx, 2010.

[98] Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. An optimal and progressive algorithm for skyline queries. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 467–478. ACM, 2003.

[99] Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. Progressive Skyline Computation in Database Systems. *ACM Transactions on Database Systems (TODS)*, 30(1):41–82, March 2005.

[100] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[101] V. Puš, P. Velan, L. Kekely, J. Kořenek, and P. Minařík. Hardware accelerated flow measurement of 100 Gb ethernet. In *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 1147–1148, May 2015.

[102] A. Putnam et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 13–24, June 2014.

[103] Pethuru Raj, Anupama Raman, Dhivya Nagaraj, and Siddhartha Duggirala. *High-Performance Big-Data Analytics: Computing Systems and Approaches.* Springer Publishing Company, Incorporated, 1st edition, 2015.

[104] Wim Roelandts. 15 years of innovation. *Xilinx Xcell*, 32, 1999.

[105] Nick Roussopoulos, Stephen Kelley, and Frédéric Vincent. Nearest neighbor queries. In *ACM sigmod record*, volume 24(2), pages 71–79. ACM, 1995.

[106] Bernhard Schölkopf, John C. Platt, John Shawe-Taylor, Alex J. Smola, and Robert C. Williamson. Estimating the Support of a High-Dimensional Distribution. *Neural Computation*, 13(7):1443–1471, 2001.

[107] Joachim Selke, Christoph Lofi, and Wolf-Tilo Balke. Highly scalable multiprocessing algorithms for preference-based database retrieval. In *Database Systems for Advanced Applications*, pages 246–260. Springer, 2010.

[108] Haichuan Shang and Masaru Kitsuregawa. Skyline operator on anti-correlated distributions. *Proceedings of the VLDB Endowment*, 6(9):649–660, 2013.

[109] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[110] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

[111] Philip Andrew Simpson. *FPGA Design: Best Practices for Team-based Reuse*. Springer Publishing Company, Incorporated, 2nd edition, 2015.

[112] Spirent Communications. SmartBits. `http://www.spirentfederal.com/ip/products/smartbits/datasheets/`.

[113] John E. Stone, David Gohara, and Guochun Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *IEEE Design & Test*, 12(3):66–73, May 2010.

[114] Volker Strassen. Gaussian Elimination is Not Optimal. *Numerische Mathematik*, 13(4):354–356, August 1969.

[115] Liang Su, Peng Zou, and Yan Jia. *Adaptive Mining the Approximate Skyline over Data Stream*, pages 742–745. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[116] Yutaka Sugawara, Mary Inaba, and Kei Hiraki. Implementation and Evaluation of Fine-grain Packet Interval Control. *IPSJ SIG Technical Reports (in Japanese)*, 2005(79):85–92, Aug 2005.

[117] Yutaka Sugawara, Mary Inaba, and Kei Hiraki. Flow Assignment Method for Parallel TCP Streams. *IPSJ SIG Technical Reports (in Japanese)*, Jul 2006.

[118] Yutaka Sugawara, Mary Inaba, and Kei Hiraki. Flow Balancing Hardware for Parallel TCP Streams on Long Fat Pipe Network. In *Proceedings of the Future Generation Communication and Networking - Volume 01*, FGCN '07, pages 391–396, Washington, DC, USA, 2007. IEEE Computer Society.

[119] Yutaka Sugawara, Takeshi Yoshino, Mary Inaba, and Kei Hiraki. Fine Tune for parallel TCP Streams on Long Fat-pipe Network using Hardware Engine. In *Proceedings of PFLDnet 2008 (Fifth International Workshop on Protocols for FAST Long-Distance Networks)*, 2008.

[120] Ryousei Takano, Tomohiro Kudoh, Yuetsu Kodama, Motohiko Matsuda, Hiroshi Tezuka, and Yutaka Ishikawa. Design and evaluation of precise software pacing mechanisms for fast long-distance networks. In *Workshop on Protocols for Fast Long-Distance Networks (PFLDnet)*, 2005.

[121] Kian Lee Tan, Pin Kwang Eng, Beng Chin Ooi, et al. Efficient progressive skyline computation. In *Proceedings of the 27th international conference on Very Large Data Bases*, volume 1, pages 301–310, 2001.

[122] Yufei Tao and Dimitris Papadias. Maintaining sliding window skylines on data streams. *IEEE Transactions on Knowledge and Data Engineering*, 18(3):377–391, March 2006.

[123] The Ministry of Education, Culture, Sports, Science and Technology (MEXT). XGE-ProtoDevel. `http://www.mext.go.jp/b_menu/shingi/gijyutu/gijyutu2/006/shiryo/08042313/001/001.pdf` (in Japanese).

[124] James E. Thornton. Parallel Operation in the Control Data 6600. In *Proceedings of the October 27-29, 1964, Fall Joint Computer Conference, Part II: Very High Speed Computer Systems*, AFIPS '64 (Fall, part II), pages 33–40, New York, NY, USA, 1965. ACM.

[125] Li Tian, Le Wang, Peng Zou, Yan Jia, and Aiping Li. Continuous monitoring of skyline query over highly dynamic moving objects. In *Proceedings of the 6th ACM international workshop on Data engineering for wireless and mobile access*, pages 59–66. ACM, 2007.

[126] Kris Tiri and Ingrid Verbauwhede. A Logic Level Design Methodology for a Secure DPA Resistant ASIC or FPGA Implementation. In *Proceedings of the Conference on Design, Automation and Test in Europe Conference and Exhibition*, volume 1 of *DATE '04*, pages 246–251, Washington, DC, USA, 2004. IEEE Computer Society.

[127] R. M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, 11(1):25–33, Jan 1967.

[128] TOP500.org. Top500 List for November 2017. `https://www.top500.org/lists/2017/11/`, 2017.

[129] Son Dao Trong, Martin Schmookler, Eric. M. Schwarz, and Michael Kroener. P6 Binary Floating-Point Unit. In *Proceedings of the 18th IEEE Symposium on Computer Arithmetic*, ARITH '07, pages 77–86, Washington, DC, USA, 2007. IEEE Computer Society.

[130] Cem Unsalan and Bora Tar. *Digital System Design with FPGA, Implementation Using Verilog and VHDL.* McGraw-Hill Education, 2017.

[131] Frank Vahid. *Digital Design with RTL Design, Verilog and VHDL.* Wiley Publishing, 2nd edition, 2010.

[132] S. Vangal, Y. Hoskote, D. Somasekhar, V. Erraguntla, J. Howard, G. Ruhl, V. Veeramachaneni, D. Finan, S. Mathew, and N. Borkar. A 5 GHz floating point multiply-accumulator in 90 nm dual V/sub T/CMOS. In *2003 IEEE International Solid-State Circuits Conference, 2003. Digest of Technical Papers. ISSCC.*, pages 334–497 vol.1, Feb 2003.

[133] Hasitha Muthumala Waidyasooriya, Masanori Hariyama, and Kunio Uchiyama. *Design of FPGA-Based Computing Systems with OpenCL.* Springer, 2017.

[134] David A. White and Ramesh Jain. Similarity Indexing with the SS-tree. In *Proceedings of the Twelfth International Conference on Data Engineering*, ICDE '96, pages 516–523, Washington, DC, USA, 1996. IEEE Computer Society.

[135] Peter R Wilson. *Design recipes for FPGAs: using Verilog and VHDL. Design recipes for field-programmable gate arrays; 2nd ed.* Newnes, London, 2016.

[136] F. G. Wolff, M. J. Knieser, D. J. Weyer, and C. A. Papachristou. High-level low power FPGA design methodology. In *Proceedings of the IEEE 2000 National Aerospace and Electronics Conference. NAECON 2000.*, pages 554–559, 2000.

[137] Louis Woods, Gustavo Alonso, and Jens Teubner. Parallel Computation of Skyline Queries. In *Proceedings of the 2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, FCCM '13, pages 1–8, Washington, DC, USA, 2013. IEEE Computer Society.

[138] Louis Woods, Gustavo Alonso, and Jens Teubner. Parallelizing Data Processing on FPGAs with Shifter Lists. *ACM Transactions on Reconfigurable Technology and Systems (TRETS) - Special Section on FPL 2013*, 8(2):7:1–7:22, March 2015.

[139] Xilinx. *LogiCORE IP SPI-4.2 v12.2, Product Specification*, 2012.

[140] Xilinx. *Introduction to FPGA Design with Vivado High-Level Synthesis*, 2013.

[141] Xilinx. *Virtex-5 FPGA Data Sheet: DC and Switching Characteristics, Product Specification*, 2016.

[142] Takeshi Yoshino, Yutaka Sugawara, Katsushi Inagami, Junji Tamatsukuri, Mary Inaba, and Kei Hiraki. Performance Optimization of TCP/IP over 10 Gigabit Ethernet by Precise Instrumentation. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 11:1–11:12, Piscataway, NJ, USA, 2008. IEEE Press.

[143] Takeshi Yoshino, Junji Tamatsukuri, Katsushi Inagami, Yutaka Sugawara, Mary Inaba, and Kei Hiraki. Analysis of 10 Gigabit Ethernet using Hardware Engine for Performance Tuning on Long Fat-pipe Network. In *Proceedings of PFLDnet 2007 (Fifth International Workshop on Protocols for FAST Long-Distance Networks)*, pages 43–48, Feb 2007.

[144] Shiming Zhang, Nikos Mamoulis, and David W Cheung. Scalable skyline computation using object-based space partitioning. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 483–494. ACM, 2009.

Appendix A

Source Code in Behavioral Synthesis

A.1 C-Like Code of Dominance Relation Calculation (Section 6.6.1)

Listing A.1: skydom.h

```
#ifndef __SKYDOM_H__
#define __SKYDOM_H__

#define DIMENSION 5
#define N 500

unsigned char dominates_array(unsigned short da[DIMENSION], unsigned short db[
      DIMENSION]);
void dominates_array_top_function(unsigned short da[DIMENSION], unsigned short db[
      DIMENSION], unsigned char *result);

#endif
```

Listing A.2: skydom.c

```
#include "skydom.h"

unsigned char dominates_array(unsigned short da[DIMENSION], unsigned short db[
      DIMENSION]) {
    int d;
    unsigned char anyBetter = 0;
    for (d = 0; d < DIMENSION; d++) {
        if (da[d] > db[d])
            return 0;
        else if (da[d] < db[d])
            anyBetter = 1;
    }
    return anyBetter;
}

void dominates_array_top_function(unsigned short da[DIMENSION], unsigned short db[
      DIMENSION], unsigned char *result) {
    *result = dominates_array(da, db);
}
```

Listing A.3: tb_main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include "skydom.h"

inline double get_dtime(void) {
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return ((double)(tv.tv_sec) + (double)(tv.tv_usec) * 0.001 * 0.001);
}

```

```c
int main() {
    int i, j, k;

    /*
     * load input data
     */
    unsigned short dataset[N][DIMENSION];
    char buf[1024];
    FILE *ifh;
    FILE *ofh;
    ifh = fopen("tv_large.input.txt", "r");
    if (ifh == NULL) {
        printf(" [Error] can't open input data file\n");
        return 1;
    }

    for (i = 0; i < N; i++) {
        for (j = 0; j < DIMENSION; j++) {
            int x;
            fscanf(ifh, "%d", &x);
            dataset[i][j] = x;
        }
        char temp[10000];
        fgets(temp, 10000, ifh);
    }
    fclose(ifh);

    /*
     * evaluation
     */
    for (i = 0; i < N-1; i++) {
        for (j = i+1; j < N; j++) {
            unsigned char result_sw;
            unsigned char result_hw;

            result_sw = dominates_array(dataset[i], dataset[j]);
            dominates_array_top_function(dataset[i], dataset[j], &result_hw);

            if (result_hw != result_sw) {
                printf(" [Error] simulation fault");
                printf(" [Error] i = %d, j = %d\n", i, j);
                printf(" [Error] result_hw = %d, result_sw = %d\n", result_hw, result_sw);
                for (k = 0; k < DIMENSION; k++) {
                    printf("%d vs %d\n", dataset[i][k], dataset[j][k]);
                }

                return 1;
            }
        }
    }
    return 0;
}
```

Listing A.4: directive.tcl

```tcl
#
# directives
#
set TopFunction dominates_array_top_function

set_directive_pipeline $TopFunction
set_directive_interface -mode ap_ctrl_hs $TopFunction
set_directive_interface -mode ap_hs -register $TopFunction da
set_directive_interface -mode ap_hs -register $TopFunction db
set_directive_interface -mode ap_vld -register $TopFunction result
set_directive_array_partition -type complete $TopFunction da
set_directive_array_partition -type complete $TopFunction db
```

A.2 C-Like Code of BCH Decoding (Section 6.6.2)

Listing A.5: bchdec.h

```c
#ifndef __BCHDEC_H__
#define __BCHDEC_H__

#include <stdio.h>

#define GF_ORDER 4
#define GF_N (2*2*2*2-1)
#define CODE_N 15
#define CODE_K 7

void conv_int_to_bitarray(unsigned char [], int);
int conv_bitarray_to_int(unsigned char []);
void bchdec(unsigned char [CODE_N], const unsigned char [CODE_N]);
int bchdec_int(int);

#endif
```

Listing A.6: bchdec.c

```c
#include "bchdec.h"

void conv_int_to_bitarray(unsigned char d[], int s) {
    int i;
    for (i = 0; i < CODE_N; i++)
        d[i] = (s >> (CODE_N-1-i)) & 0x1;
}

int conv_bitarray_to_int(unsigned char s[]) {
    int i;
    int ret = 0;
    for (i = 0; i < CODE_N; i++)
        ret |= s[i] << (CODE_N-1-i);
    return ret;
}

int gf2_4_f(int s) {
    s = (s + GF_N) % GF_N;
    int tbl[] = {
        0x1, 0x2, 0x4, 0x8,
        0x3, 0x6, 0xC, 0xB,
        0x5, 0xA, 0x7, 0xE,
        0xF, 0xD, 0x9};
    return tbl[s];
}

int gf2_4_b(int s) {
    int tbl[] = {
        -1, 0, 1, 4,
        2, 8, 5, 10,
        3, 14, 9, 7,
        6, 13, 11, 12};
    return tbl[s];
}

unsigned int gf2_4_inv(int s) {
    return gf2_4_f(gf2_4_b(1) - gf2_4_b(s));
}

unsigned int vec_mul(unsigned int src1, unsigned int src2) {
  if (src1 == 0x0) return 0x0;
  if (src2 == 0x0) return 0x0;
    return gf2_4_f((gf2_4_b(src1) + gf2_4_b(src2)) % 15);
}

void print_binary(unsigned int src1, unsigned int bit) {
  int i;
  printf("[");
  for (i = bit - 1; i >= 0; i--) {
    printf("%1d", (src1 >> i) & 0x1);
  }
  printf("]\n");
  return;
}

void bchdec(unsigned char dst[CODE_N], const unsigned char src[CODE_N]) {
    int i;

    unsigned int s1 = 0;
    unsigned int s2 = 0;
    unsigned int s3 = 0;
    unsigned int s4 = 0;
    for (i = 0; i < CODE_N; i++) {
        if (src[i]) {
            unsigned int idx = CODE_N - 1 - i;
            s1 = s1 ^ gf2_4_f(idx);
            s2 = s2 ^ vec_mul(gf2_4_f(idx), gf2_4_f(idx));
            s3 = s3 ^ vec_mul(gf2_4_f(idx), vec_mul(gf2_4_f(idx), gf2_4_f(idx)));
            s4 = s4 ^ vec_mul(gf2_4_f(idx), vec_mul(gf2_4_f(idx), vec_mul(gf2_4_f(idx), gf2_4_f(idx))));
        }
    }
    unsigned int d = vec_mul(s1, s3) ^ vec_mul(s2, s2);
    unsigned int l1 = 0x0;
    unsigned int l2 = 0x0;

    if (d != 0x0) {
        /* two-bit error */
        unsigned int l1_temp0 = vec_mul(vec_mul(s1, gf2_4_inv(s2)), s4) ^ s3;
        unsigned int l1_temp1 = vec_mul(vec_mul(s1, gf2_4_inv(s2)), s3) ^ s2;
        l1 = vec_mul(l1_temp0, gf2_4_inv(l1_temp1));
        l2 = vec_mul(vec_mul(s2, gf2_4_inv(s1)), l1) ^ vec_mul(s3, gf2_4_inv(s1));
    } else if (s1 != 0x0) {
        /* one-bit error */
        l1 = vec_mul(s2, gf2_4_inv(s1));
        l2 = 0x0;
    } else {
        /* no error */
    }

    unsigned int e[CODE_N];
        for (i = 0; i < CODE_N; i++) {
        e[i] = 0;
    }
    for (i = 0; i < CODE_N; i++) {
        if ((vec_mul(l2, vec_mul(gf2_4_f(i), gf2_4_f(i))) ^ vec_mul(l1, gf2_4_f(i)) ^ gf2_4_f( 0)) == 0x0) e[gf2_4_b(gf2_4_inv(gf2_4_f(i)))] = 1;
    }
    for (i = 0; i < CODE_N; i++) {
        int idx = CODE_N - 1 - i;
        dst[i] = src[i] ^ e[idx];
    }
}

int bchdec_int(int s_int) {
    unsigned char s_array[CODE_N];
    unsigned char d_array[CODE_N];
    conv_int_to_bitarray(s_array, s_int);
    bchdec(d_array, s_array);
    return conv_bitarray_to_int(d_array);
}
```

Listing A.7: tb_main.c

```c
#include <stdio.h>
#include <stdlib.h>

#include "bchdec.h"

int main() {
    int i;

    FILE *ifh;
    ifh = fopen("bench_in.txt", "r");
    if (ifh == NULL) {
        printf(" [Error] fopen()");
        return 1;
    }

    for(;;) {
        int sent_int;
        int recv_int;
        int scnt = fscanf(ifh, "%x %x", &sent_int, &recv_int);
        if (scnt < 0) {
            break;
        }
        unsigned char sent[CODE_N];
        unsigned char recv[CODE_N];
        unsigned char corr[CODE_N];
        conv_int_to_bitarray(sent, sent_int);
        conv_int_to_bitarray(recv, recv_int);
        conv_int_to_bitarray(corr, bchdec_int(sent_int));

        int errbit1 = 0;
        int errbit2 = 0;
        for (i = 0; i < CODE_N; i++) {
          if (sent[i] != recv[i])
                errbit1++;
          if (sent[i] != corr[i])
                errbit2++;
        }
        if (errbit2) {
            return EXIT_FAILURE;
        }
    }
    printf(" [Info] test pass!\n");
    return EXIT_SUCCESS;
}
```

Listing A.8: directive.tcl

```
#
# directives
#
set TopFunction bchdec

set_directive_pipeline $TopFunction
set_directive_interface -mode ap_ctrl_hs $TopFunction
set_directive_interface -mode ap_hs -register $TopFunction src
set_directive_interface -mode ap_vld -register $TopFunction dst
set_directive_array_partition -type complete $TopFunction src
set_directive_array_partition -type complete $TopFunction dst
```

A.3 C-Like Code of JR-tree Algorithm (Section 6.6.3)

This source code is compatible with the file format of the dataset provided in MEMOCODE 2015 Design Contest [1].

Listing A.9: parameter.h

```
#ifndef __PARAMETERS_H__
#define __PARAMETERS_H__

#define CARD_MAX 12000
#define MAX_SKYLINE_NUM 512
#define MAX_TIME_STEP_NUM 12000
#define OPERATION_DEFAULT 550
#define MAX_DIMENSION_NUM 8

#endif
```

Listing A.10: mylib.h

[1] http://www.ece.stonybrook.edu/~pmilder/memocode15/

```c
#ifndef __MYLIB_H__
#define __MYLIB_H__

#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <string.h>
#include <malloc.h>
#include <sys/time.h>

typedef unsigned long long int ULLI;

#define MAX(a, b) ((a) > (b) ? (a) : (b))
#define MIN(a, b) ((a) < (b) ? (a) : (b))
#define REP(i, n) for ((i) = 0; (i) < (n); (i)++)
#define REPREV(i, n) for ((i) = (n)-1; (i) >= 0; (i)--)
#define DEBUG(s) \
  do { \
    printf("%s\n", (s)); \
    fflush(stdout); \
  } while (0)

double get_dtime(void);
void fprintf_space(FILE *, int);
void chomp(char *);

#endif
```

Listing A.11: mylib.c

```c
#include "mylib.h"

double get_dtime(void) {
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return ((double)(tv.tv_sec) + (double)(tv.tv_usec) * 0.001 * 0.001);
}

void fprintf_space(FILE *stream, int len) {
    int i;
    REP(i, len) {
        fprintf(stream, " ");
    }
}

void chomp(char *s) {
    int len;
    len = strlen(s);
    if ((len > 0) && s[len-1] == '\n')
        s[len-1] = '\0';
    len = strlen(s);
    if ((len > 0) && s[len-1] == '\r')
        s[len-1] = '\0';
}
```

Listing A.12: skylib.h

```c
#ifndef __SKYLIB_H__
#define __SKYLIB_H__

#include "parameter.h"
#include "mylib.h"

typedef unsigned int TYPE;
typedef unsigned short TYPE_E;

#define DOMINATES dominates_array

typedef struct skyio_structure {
    int datasetSize;
    int dimensions;
    int maxSkylineElements;
    TYPE maxTimeSteps;
    char benchmarkName[256];

    TYPE_E **dataset;
    TYPE **datasetTimes;
    TYPE **skyline;
    TYPE *numSkyline;
} SKYIO_T;

unsigned char dominates_array(TYPE_E [], TYPE_E [], int);
void getSettings(SKYIO_T *, char **);
void initAllocation(SKYIO_T *);
void initValues(SKYIO_T *);
void outputAndCleanup(SKYIO_T *);
int outputVerification(SKYIO_T *);

#endif
```

Listing A.13: skylib.c

```
#include "skylib.h"

int cmpfunc(const void *a, const void *b) {
    return (*(TYPE *)a - *(TYPE *)b);
}

unsigned char dominates_array(TYPE_E da[MAX_DIMENSION_NUM], TYPE_E db[
     MAX_DIMENSION_NUM], int dim) {
    int d;
    unsigned char anyBetter = 0;
    for (d = 0; d < dim; d++) {
        if (da[d] > db[d])
            return 0;
        else if (da[d] < db[d])
            anyBetter = 1;
    }
    return anyBetter;
}

void getSettings(SKYIO_T *skyio_pack, char *argv[]) {
    char ifname[8192];
    sprintf(ifname, "%s.setup", argv[1]);
    FILE *f = fopen(ifname, "r");
    if (!f) {
        printf("Cannot open settings file %s\n", ifname);
        assert(0);
    }
    printf("Reading settings from %s.\n", ifname);

    fscanf(f, "%d", &(skyio_pack->datasetSize));
    fscanf(f, "%d", &(skyio_pack->dimensions));
    fscanf(f, "%d", &(skyio_pack->maxTimeSteps));
    fscanf(f, "%d", &(skyio_pack->maxSkylineElements));
    fscanf(f, "%s", skyio_pack->benchmarkName);
    strcpy(skyio_pack->benchmarkName, argv[1]);
    fclose(f);
}

void initAllocation(SKYIO_T *skyio_pack) {
    skyio_pack->dataset = (TYPE_E **)malloc(sizeof(*(skyio_pack->dataset)) * skyio_pack->
         datasetSize);
    skyio_pack->datasetTimes = (TYPE **) malloc(sizeof(*(skyio_pack->datasetTimes)) *
         skyio_pack->datasetSize);
    skyio_pack->skyline = (TYPE **) malloc(sizeof(*(skyio_pack->skyline)) * skyio_pack->
         maxTimeSteps);
    skyio_pack->numSkyline = (TYPE *) malloc(sizeof(TYPE) * skyio_pack->maxTimeSteps);

    if ((skyio_pack->dataset == NULL) ||
        (skyio_pack->datasetTimes == NULL) ||
        (skyio_pack->skyline == NULL) ||
        (skyio_pack->numSkyline == NULL))
        fprintf(stderr, " [Error] memory allocation, dataset, datasetTimes, skyline,
             numSkyline.\n");
```

```
}

void initValues(SKYIO_T *skyio_pack) {
    int i, j;
    TYPE ii;

    int datasetSize = skyio_pack->datasetSize;
    int dimensions = skyio_pack->dimensions;
    int maxTimeSteps = skyio_pack->maxTimeSteps;
    int maxSkylineElements = skyio_pack->maxSkylineElements;
    char *benchmarkName = skyio_pack->benchmarkName;
    TYPE_E **dataset = skyio_pack->dataset;
    TYPE **datasetTimes = skyio_pack->datasetTimes;
    TYPE **skyline = skyio_pack->skyline;
    TYPE *numSkyline = skyio_pack->numSkyline;

    if (dataset) {
        for (i = 0; i < datasetSize; i++) {
            dataset[i] = (TYPE_E *)malloc(sizeof(TYPE_E) * dimensions);
            if (dataset[i] == NULL)
                fprintf(stderr, " [Error] memory allocation, dataset[%d].\n", i);
        }
    }
    if (datasetTimes) {
        for (i = 0; i < datasetSize; i++) {
            datasetTimes[i] = (TYPE *)malloc(sizeof(TYPE) * 2);
            if (datasetTimes[i] == NULL)
                fprintf(stderr, " [Error] memory allocation, datasetTimes[%d].\n", i);
        }
    }
    if (skyline) {
        for (ii = 0; ii < maxTimeSteps; ii++) {
            skyline[ii] = (TYPE *)malloc(sizeof(TYPE) * maxSkylineElements);
            if (skyline[ii] == NULL)
                fprintf(stderr, " [Error] memory allocation, skyline[%d].\n", ii);
        }
    }

    char buf[256];
    sprintf(buf, "%s.input", benchmarkName);
    FILE *f = fopen(buf, "r");
    if (!f) {
        printf("Can't open file %s\n", buf);
        assert(0);
    }

    REP(i, datasetSize) {
        REP(j, dimensions) {
            int x;
            fscanf(f, "%d", &x);
            dataset[i][j] = (TYPE_E)x;
        }
        char temp[10000];
        fgets(temp, 10000, f);
    }
    fclose(f);

    sprintf(buf, "%s.times", benchmarkName);
    f = fopen(buf, "r");
    if (!f) {
        printf("Can't open file%s\n", buf);
        assert(0);
    }

    REP(i, datasetSize) {
        int x;
        fscanf(f, "%d", &x);
        datasetTimes[i][0] = (TYPE)x;
        fscanf(f, "%d", &x);
        datasetTimes[i][1] = (TYPE)x;
    }
    fclose(f);

    REP(ii, maxTimeSteps)
```

```
        numSkyline[ii] = 0;
}

void outputAndCleanup(SKYIO_T *skyio_pack) {
    int i;
    TYPE ii;
    TYPE l;
    FILE *f;

    int datasetSize = skyio_pack->datasetSize;
    int maxTimeSteps = skyio_pack->maxTimeSteps;
    char *benchmarkName = skyio_pack->benchmarkName;
    TYPE_E **dataset = skyio_pack->dataset;
    TYPE **datasetTimes = skyio_pack->datasetTimes;
    TYPE **skyline = skyio_pack->skyline;
    TYPE *numSkyline = skyio_pack->numSkyline;

    char buf[8192];
    sprintf(buf, "%s.out", benchmarkName);
    f = fopen(buf, "w");
    if (!f) {
        printf("Can't open file %s\n", buf);
        assert(0);
    }

    REP(ii, maxTimeSteps) {
        qsort(skyline[ii], numSkyline[ii], sizeof(TYPE), cmpfunc);
        REP(l, numSkyline[ii])
            fprintf(f, "%d ", skyline[ii][l]);
        fprintf(f, "\n");
    }
    fclose(f);

    REP(i, datasetSize)
        free(dataset[i]);
    free(dataset);

    REP(i, datasetSize)
        free(datasetTimes[i]);
    free(datasetTimes);

    REP(ii, maxTimeSteps)
        free(skyline[ii]);
    free(skyline);

    free(numSkyline);
}

int outputVerification(SKYIO_T *skyio_pack) {
    char ofname1[8192];
    char ofname2[8192];
    sprintf(ofname1, "%s.out", skyio_pack->benchmarkName);
    sprintf(ofname2, "%s.refout", skyio_pack->benchmarkName);
    FILE *ofh1 = fopen(ofname1, "rb");
    FILE *ofh2 = fopen(ofname2, "rb");
    if ((!ofh1) || (!ofh2)) {
        printf("Can't open out/refout files\n");
        assert(0);
    }

    char buf1[100000];
    char buf2[100000];
    while(!feof(ofh1) && !feof(ofh2)) {
        fgets(buf1, sizeof(buf1), ofh1);
        fgets(buf2, sizeof(buf2), ofh2);
        chomp(buf1);
        chomp(buf2);
        if (strcmp(buf1, buf2) != 0) {
            printf(" [Error] incorrect skyline: line mismatch.\n");
            printf("<%s> <%s>\n", buf1, buf2);
            return EXIT_FAILURE;
        }
    }
    if (feof(ofh1) != feof(ofh2)) {
```

```c
        printf(" [Error] incorrect skyline: short length.\n");
        return EXIT_FAILURE;
    }
    printf(" [Info] correct skylines.\n");
    return EXIT_SUCCESS;
}
```

Listing A.14: tree.h

```c
#ifndef __TREE_H__
#define __TREE_H__

#include "parameter.h"
#include "skylib.h"

typedef unsigned short TYPE_TS;

typedef struct memory_pack {
    int data_parent[CARD_MAX];
    int child_head[CARD_MAX];
    int child_next[CARD_MAX];
    int root_num;
    int root_child[MAX_SKYLINE_NUM+10];
    int root_child_reverse[CARD_MAX];
    int idmap[CARD_MAX];
    TYPE_E m_dataset[CARD_MAX][MAX_DIMENSION_NUM];
    int dimensions;
} MEMPACK;

void initAllocationMP(MEMPACK *, SKYIO_T *);
int reg_id(MEMPACK *, SKYIO_T *, int);
void proc_single_timestep(MEMPACK *, TYPE, int*, int*, TYPE *, TYPE *);

#endif
```

Listing A.15: tree.c

```c
#include "parameter.h"
#include "tree.h"

void initAllocationMP(MEMPACK *mp, SKYIO_T *skyio_pack) {
    mp->root_num = 0;
    mp->dimensions = skyio_pack->dimensions;
    int i;
    REP(i, CARD_MAX)
        mp->idmap[i] = -1;
}

int get_hash(int s) {
    int offset = 0x97cd;
    int prime = 0x0193;
    int x;
    x = offset ^ (s & 0xff);
    x = ((x & 0xffff) * prime) & 0xffff;
    x = x ^ ((s >> 8) & 0xff);
    x = ((x & 0xffff) * prime) & 0xffff;
    x = x % CARD_MAX;
    return x;
}

int reg_id(MEMPACK *mp, SKYIO_T *skyio_pack, int s) {
    int x = get_hash(s);
    for(;;) {
        if (mp->idmap[x] == -1) break;
        x = (x + 1) % CARD_MAX;
    }
    mp->idmap[x] = s;
    int i;
    REP(i, skyio_pack->dimensions) {
        mp->m_dataset[x][i] = skyio_pack->dataset[s][i];
    }
    return x;
```

```c
}

int unreg_id(MEMPACK *mp, int s) {
    int x = get_hash(s);
    for(;;) {
        if (mp->idmap[x] == s) break;
        x = (x + 1) % CARD_MAX;
    }
    mp->idmap[x] = -1;
    return x;
}

int remap_id(MEMPACK *mp, int x) {
    int s = mp->idmap[x];
    if (s == -1)
        printf(" [Error] invalid idmap value, idx=%d\n", x);
    return mp->idmap[x];
}

void add_child(MEMPACK *mp, int parent, int child) {
    mp->data_parent[child] = parent;
    mp->child_next[child] = mp->child_head[parent];
    mp->child_head[parent] = child;
}

void del_root_child(MEMPACK *mp, int child) {
    int tmp1 = mp->root_child_reverse[child];
    int tmp2 = mp->root_child[--mp->root_num];
    mp->root_child[tmp1] = tmp2;
    mp->root_child_reverse[tmp2] = tmp1;
}

void stree_root_add(MEMPACK *mp, int target, int move_only) {
    int num = mp->root_num;
    int i;
    if (!move_only) {
        mp->child_head[target] = -1;
    }
    mp->data_parent[target] = -1;
    REP (i, num) {
        if (DOMINATES(mp->m_dataset[mp->root_child[i]], mp->m_dataset[target], mp->
             dimensions)) {
            add_child(mp, mp->root_child[i], target);
            return;
        }
    }
    int candidate[MAX_SKYLINE_NUM];
    REP(i, num) {
        candidate[i] = mp->root_child[i];
    }

    REP (i, num) {
        if (DOMINATES(mp->m_dataset[target], mp->m_dataset[candidate[i]], mp->
             dimensions)) {
            del_root_child(mp, candidate[i]);
            add_child(mp, target, candidate[i]);
        }
    }
    mp->root_child[mp->root_num] = target;
    mp->root_child_reverse[target] = mp->root_num;
    mp->root_num++;
}

void stree_remove(MEMPACK *mp, int target) {
    int parent = mp->data_parent[target];
    if (parent == -1) {
        del_root_child(mp, target);
        int z = mp->child_head[target];
        for (;;) {
            if (z == -1) break;
            int next = mp->child_next[z];
            stree_root_add(mp, z, 1);
            z = next;
        }
    } else {
        int x = mp->child_head[parent];
        int z = mp->child_next[x];
        if (x == target) {
            mp->child_head[parent] = z;
        } else {
            for (;;) {
                int y = z;
                z = mp->child_next[y];
                if (y == -1) {
                    printf(" [Error] delete: node[%d] is not node[%d]'s child.", target, parent);
                    assert(0);
                }
                if (y == target) {
                    mp->child_next[x] = z;
                    break;
                }
                x = y;
            }
        }
        z = mp->child_head[target];
        for (;;) {
            if (z == -1) break;
            int next = mp->child_next[z];
            add_child(mp, parent, z);
            z = next;
        }
    }
}

void stree_dump(MEMPACK *mp, int target, int depth) {
    if (depth == 0) {
        int i;
        REP(i, mp->root_num) {
            stree_dump(mp, mp->root_child[i], depth + 1);
        }
    } else {
        fprintf_space(stdout, 2 * depth);
        fprintf(stdout, "%d\n", target);
        int z = mp->child_head[target];
        for (;;) {
            if (z == -1) break;
            stree_dump(mp, z, depth + 1);
            z = mp->child_next[z];
        }
    }
}

void proc_single_timestep(
    MEMPACK *mp, TYPE timestep,
    int addlist_t[OPERATION_DEFAULT], int remlist_t[OPERATION_DEFAULT],
    TYPE skyline[MAX_SKYLINE_NUM], TYPE *numSkyline) {

    int i;

    REP(i, addlist_t[0]) {
        stree_root_add(mp, addlist_t[i+1], 0);
    }
    REP(i, remlist_t[0]) {
        int x = remlist_t[i+1];
        x = unreg_id(mp, x);
        stree_remove(mp, x);
    }
    *numSkyline = mp->root_num;
    REP (i, mp->root_num)
        skyline[i] = remap_id(mp, mp->root_child[i]);

#ifdef __EAGER_SIMPLE__
    stree_eager_simple(mp);
#endif
#ifdef __EAGER_DOUBLE__
    stree_eager_double(mp);
#endif
```

```c
#ifdef __EAGER_FULL__
    stree_eager(mp);
#endif

}
```

Listing A.16: tb_main.c

```c
#include "mylib.h"
#include "skylib.h"
#include "tree.h"

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s setupfile's prefix\n", argv[0]);
    } else {
        SKYIO_T *skyio_pack = malloc(sizeof(SKYIO_T));
        getSettings(skyio_pack, argv);
        initAllocation(skyio_pack);
        initValues(skyio_pack);

        int **addlist = malloc(sizeof(*addlist) * (skyio_pack->maxTimeSteps+1));
        int **remlist = malloc(sizeof(*remlist) * (skyio_pack->maxTimeSteps+1));
        int time;
        REP(time, skyio_pack->maxTimeSteps+1) {
            addlist[time] = calloc(sizeof(int), OPERATION_DEFAULT);
            remlist[time] = calloc(sizeof(int), OPERATION_DEFAULT);
        }
        int i;
        REP(i, skyio_pack->datasetSize) {
            int addtime = skyio_pack->datasetTimes[i][0];
            int remtime = skyio_pack->datasetTimes[i][1];
            addlist[addtime][0]++;
            addlist[addtime][addlist[addtime][0]] = i;
            remlist[remtime][0]++;
            remlist[remtime][remlist[remtime][0]] = i;
        }

        MEMPACK *mp = malloc(sizeof(MEMPACK));
        initAllocationMP(mp, skyio_pack);

        TYPE t;
        REP(t, skyio_pack->maxTimeSteps) {
            int addlist_t[OPERATION_DEFAULT];
            int remlist_t[OPERATION_DEFAULT];
            memcpy(addlist_t, addlist[t], sizeof(int) * OPERATION_DEFAULT);
            memcpy(remlist_t, remlist[t], sizeof(int) * OPERATION_DEFAULT);

            int mapped_addlist_t[OPERATION_DEFAULT];
            mapped_addlist_t[0] = addlist_t[0];
            REP(i, addlist_t[0]) {
                int x = addlist_t[i+1];
                mapped_addlist_t[i+1] = reg_id(mp, skyio_pack, x);
            }

            TYPE skyline[MAX_SKYLINE_NUM];
            TYPE numSkyline;

            proc_single_timestep(mp, t, mapped_addlist_t, remlist_t, skyline, &numSkyline);

            skyio_pack->numSkyline[t] = numSkyline;
            memcpy(skyio_pack->skyline[t], skyline, sizeof(TYPE) * numSkyline);
        }

        free(mp);
        outputAndCleanup(skyio_pack);
        int return_status = outputVerification(skyio_pack);
        free(skyio_pack);

        return return_status;
    }
    return EXIT_FAILURE;
}
```

Listing A.17: directive.tcl

```
#
# directives
#
set TopFunction proc_single_timestep

set_directive_pipeline $TopFunction
set_directive_interface -mode ap_ctrl_hs $TopFunction
set_directive_interface -mode ap_none -register $TopFunction mp
set_directive_interface -mode ap_hs -register $TopFunction timestemp
set_directive_interface -mode ap_fifo -register $TopFunction skyline
set_directive_interface -mode ap_vld -register $TopFunction numSkyline
```