TOHOKU UNIVERSITY
Graduate School of Information Sciences

Latency-tolerant Vector Processor Architectures

(レイテンシ耐性を持つベクトルプロセッサアーキテクチャに関する研究)

A dissertation submitted for the degree of
Doctor of Philosophy (Information Sciences)

Department of Computer and Mathematical Sciences

by

Hikaru Takayashiki

January 10, 2023

Latency-tolerant Vector Processor Architectures

Hikaru Takayashiki

Abstract

A vector instruction set is now available for modern processors. This instruction set contributes to gaining high computing capability. As a result, the vector instruction set is essential for modern processors.

In combination with the high memory performance, vector processors that utilize a vector instruction set can achieve high sustained performance. By optimizing the architecture for vector instructions with a long vector length and designing the memory system to prioritize high memory bandwidth, vector processors can achieve a high performance in scientific and engineering applications. The increasing demand for higher accuracy and broader applicability in these fields has led to a growth in performance requirements, making vector processors a promising option for achieving high performance.

Modern vector processors have adopted four features different from the classical ones. First, modern vector processors have increased their computing capability by increasing the number of vector cores, even though historically the vector processors have focused on increasing the performance of one vector core. As this trend may continue in the future, the many-core technology will play an essential role in driving the growth of computing capability of vector processors. Second, modern vector processors employ an out-of-order execution mechanism for vector instructions. This allows vector processors to increase the utilization of vector functional units, as well as the ability to exploit instruction-level parallelism for higher sustained performance. Third, modern vector processors

have a virtual memory system. This system separates the memory space perceived by processes from the actual memory space, leading to improved memory usage efficiency and reduced programming workload for the user. Fourth, modern vector processors utilize multi-banked caches in order to maintain high sustained memory bandwidth. As the number of input/output pins that can be integrated onto a single chip is limited, further improving the off-chip memory bandwidth becomes difficult. Therefore, the cache is used to provide vector cores with reusable data at high bandwidth rather than relying on the off-chip memory.

Applications for the modern vector processors are actively developed to improve accuracy and expanded their scope. Recently, new applications such as graph processing and machine learning, which involve irregular memory accesses, have become popular workloads. Since the modern vector processors and their memory systems are optimized for high sustained performance in the case of continuous memory accesses, the processors may not perform at their best in the case of irregular ones. Furthermore, irregular memory accesses may cause cache misses due to the lack of locality for the data reference, making it difficult for vector processors to maintain high performance in applications with irregular memory accesses. One common aspect of these problems is latency, i.e., the time between issuing a vector instruction and its completion. The latency is a common issue that affects the performance of systems dealing with irregular memory accesses. It can arise due to delays in accessing the data or the overhead of managing the accesses. Reducing the latency is often a key factor in improving the performance of systems with irregular memory accesses.

It is generally said that the latency problem does not arise in vector processors because their vector processing mechanism can hide latency by pipelined

data accesses. However, in modern vector processors, the latency problem becomes a performance bottleneck due to the following four reasons. First, as semiconductor manufacturing technologies improve computing performance, the time required for each vector operation tends to become short. The latency hiding capability of vector processors is gradually decreasing. Second, the number of instructions that the processor can handle simultaneously may be insufficient to exploit instruction-level parallelism from the program. For example, irregular memory accesses can be handled by the vector instructions set using specialized instructions. This handling requires multiple instructions for one access, resulting in a longer latency than sequential accesses. Third, in applications with irregular memory accesses, data are not reused in the cache system properly. The irregular memory accesses may not have locality in data references, the cache system cannot fully be utilized. To solve the problems related to these reasons, this dissertation proposes four approaches.

First, an indirect memory access prefetcher for vector gather instructions is designed to reduce the latency of indirect memory accesses on modern vector processors. These accesses are implemented as vector gather instructions in the vector instruction set, which allows for the vectorization of irregular or sparse memory access sequences. When a vector gather instruction is executed, it loads data from the main memory based on index data that has been previously loaded. Because the values in the index data are unpredictable, the processor must wait for their arrivals, which can result in long memory access latencies that negatively impact performance even on vector processors. To solve this issue, the vector gather prefetcher employs a two-phase approach. In the first phase, the vector gather prefetcher loads the index data with assuming that the index data will be accessed sequentially. In the second phase, after

prefetching the index data is complete, the vector gather prefetcher predicts the addresses for the indirect memory accesses using the prefetched index data and attempts to prefetch the data at those addresses. This allows the prefetcher to hide the latency caused by indirect memory accesses.

The evaluation of the vector gather prefetcher is performed by using a simple kernel with two types of index data: sequential values and random values. The evaluation results show that the prefetching mechanism improves the performance of the sequential-indexed and random-indexed kernels by 2.2× and 1.2×, respectively.

Second, a criticality-aware out-of-order mechanism aims at hiding latency of instructions that cause stalling. The conventional out-of-order mechanism has a limitation in the number of instructions that can overtake other instructions. If many instructions with dependencies run out the out-of-order window, the processor stops issuing instructions. Thus, the criticality-aware out-of-order mechanism issues the instructions causing stalling on another execution path. This mechanism enables modern vector processors to exploit more instruction-level parallelism and conceal the latency of issuing instruction.

The evaluation of the criticality-aware out-of-order mechanism is performed by using memory-intensive applications such as numerical simulations and graph applications. The evaluation results show that the proposed mechanism achieves up to 80% performance improvements on several memory-intensive applications.

Third, a page-address coalescing method aims at reducing the translation time between virtual memory addresses and physical memory addresses for vector instructions. Since a vector instruction handles multiple elements in one instruction, the processor has to translate multiple addresses. Furthermore, in

the case of vector gather instructions that are responsible to indirect memory accesses, the processor has to translate all the addresses. This causes a long latency for address translations. The page-address coalescing method tries to deduplicate page addresses in a vector using vector arithmetic units already built in the processor.

In the evaluation of the page-address coalescing method, simulation experiments are conducted to evaluate the performance improvement on the numerical applications and the graph applications that contain many vector gather instructions. The evaluation results show that the proposed method can achieve a 2x performance improvement in numerical applications and 1.08x in graph applications.

Fourth, a skewed multi-banked cache is designed to reduce conflict misses that occur when multiple vector cores access the cache. While a cache with a high associativity can avoid conflict misses, implementing such a cache for a multi-banked configuration can be cost-prohibitive. Instead of increasing the associativity, the skewed multi-banked cache reduces the number of conflict misses by preventing data requests from using the same cache set. This dissertation also examines the use of several hashing functions and replacement policies in the skewed multi-banked cache. Eventually, the proposed cache employs odd-multiplier displacement hashing to effectively skew the data, and the static re-reference interval prediction policy for efficient replacement. This mechanism enables processors to achieve a high cache hit ratio and reduce latency.

In the evaluation of the skewed multi-banked cache, this dissertation discusses the cache hit rates by using stencil calculation kernels varying the number of vector cores sharing a cache and its associativity. In a 3D 7-point stencil kernel, the skewed multi-banked cache can approximately eliminate 70 % of the

conflict misses. In a 3D 13-point stencil kernel, the skewed multi-banked cache can approximately eliminate 90 % of the conflict misses.

In conclusion, this dissertation demonstrates that the proposed four approaches can solve the latency problems for modern vector processors. These approaches allow modern vector processors to realize high sustained performance in memory-intensive applications by making vector processors more tolerant to latency. These approaches to solve latency problems are valuable for architects to design vector processors in the future.

Contents

1 Introduction **1**

1.1 Introduction 1

1.2 Objective of the Dissertation 5

1.3 Organization of the Dissertation 7

2 Importance of latency-tolerance for vector processor architectures **9**

2.1 Introduction 9

2.2 Modern vector processors and their requirements 10

2.3 Latency problems of vector processor 15

2.3.1 Latency of preceding dependent instructions 15

2.3.2 Latency of vector instruction stalling 17

2.3.3 Latency of address translation 20

2.3.4 Latency of memory access 22

2.4 Latency-tolerant vector processor architectures 24

2.5 Related work 28

2.5.1 Studies of indirect memory accesses 28

2.5.2 Studies of instruction stalling 29

2.5.3 Studies of address translation 31

2.5.4 Studies of vector cache 32
2.6 Conclusions 34

3 Indirect memory access prefetcher for vector gather instructions 36
3.1 Introduction 36
3.2 Motivation 38
3.2.1 Indirect memory access prefetcher 38
3.2.2 Vector gather instruction 39
3.3 Vector gather prefetcher 41
3.3.1 Stream list vector prefetcher 42
3.3.2 Indirect vector prefetcher 43
3.4 Evaluations 44
3.4.1 Configurations of the vector processor 44
3.4.2 Benchmark kernels 44
3.4.3 Simulator 45
3.4.4 Performance evaluation 46
3.4.5 Parameter study 49
3.4.6 Effect of the number of cache blocks 51
3.5 Conclusions 56

4 Criticality-aware out-of-order mechanism for vector instructions 57
4.1 Introduction 57
4.2 Motivation 60
4.2.1 Runahead execution 60
4.2.2 Problems to apply precise runahead execution for vector processors 61
4.3 Criticality-aware out-of-order vector processor 64

4.3.1 Challenges 64
4.3.1.1 Consistency of the committing order 66
4.3.1.2 Consistency of the register renaming order 67
4.3.2 Mechanism overview 67
4.3.2.1 Decoded instruction queue (DIQ) 69
4.3.2.2 Stalling instruction cache (SIC) 70
4.3.2.3 Pending commit queue (PCQ) 70
4.3.2.4 Critical register aliasing table (CRAT) 71
4.3.3 Behavior of criticality-aware vector processing 71
4.3.3.1 Recognition of the stalling instruction chain in the normal mode 72
4.3.3.2 Enter the critical mode 72
4.3.3.3 Exit the critical mode 72
4.3.3.4 Pipeline behaviors in the critical mode 73
4.3.3.5 Contextualization of early-dispatched instruction in the normal mode 74
4.3.4 Memory disambiguation 76
4.4 Evaluation 77
4.4.1 Experimental setup 77
4.4.2 Applications 79
4.4.3 Assumptions 80
4.4.4 Area overheads 82
4.4.5 Performance 82
4.4.5.1 PolyBench case 83
4.4.5.2 Practical application kernels 85
4.4.5.3 Vector Graph Library 87

4.4.6 Analysis of criticality-aware executed instructions 89
4.4.7 Architectural sensitivity study 90
4.5 Conclusions 96

5 Page-address coalescing method of vector gather instructions 97
5.1 Introduction 97
5.2 Motivation 99
5.2.1 Vector Gather Instruction 99
5.2.2 Vector Gather Instruction with Virtual Memory 100
5.3 Page-Address Coalescing for Vector Gather Instruction 101
5.3.1 Idea to use existing vector arithmetic units for page-address coalescing 101
5.3.2 Procedure of page-address coalescing 103
5.3.3 Implementation 105
5.3.4 Requirements for vector arithmetic unit 106
5.3.5 Trade-off of the proposal 106
5.4 Evaluations 108
5.4.1 Experimental setup 108
5.4.2 Applications 110
5.4.3 Coalescing trial and performance 110
5.4.4 Discussion on performance improvement 112
5.4.5 TLB access reduction 113
5.5 Conclusions 115

6 Skewed multi-banked cache for many-core vector processors 116
6.1 Introduction 116
6.2 Motivation 118

6.2.1 Many-core Vector Processors with Multi-banked Shared Cache118
6.2.2 Conflict Misses on The Many-core Vector Processor 120
6.2.2.1 3D 7-point Stencil Calculation 120
6.2.2.2 Stencil Calculation with A Shared Cache 120
6.2.3 Preliminary Evaluation 123
6.3 Skewed Multi-banked Cache for Many-core Vector Processors . . . 126
6.3.1 Hashing Functions 128
6.3.1.1 XOR-based Hashing Function 128
6.3.1.2 Odd-multiplier Displacement Hashing Function . . 129
6.3.2 Replacement Policies 130
6.3.2.1 Not Recently Used Not Recently Written 130
6.3.2.2 Static Re-Reference Interval Prediction 130
6.4 Evaluation 133
6.4.1 Experimental Environment 133
6.4.2 Evaluation Results and Discussion 135
6.4.2.1 Cache Hit Rate 135
6.4.2.2 Performance 138
6.5 Conclusions 141

7 Conclusions **142**

Bibliography **148**

Acknowledgements **161**

List of Tables

2.1 The relation of this dissertation between chapters and two perspectives with four approaches. 25

3.1 Parameters of the evaluation. 45
3.2 The relation between stride or duplicate width and the number of cache blocks per vector gather instruction. 53

4.1 Baseline configuration for the out-of-order vector processor. 78
4.2 The Bytes per flop of PolyBench. 80
4.3 The Bytes per flop of the optimized version. 81
4.4 The overview of the practical application kernels. 81
4.5 The algorithms of Vector Graph Library. 82

5.1 Baseline configuration for the out-of-order vector processor. 109
5.2 The relationship between the number of coalescing trial and occupying cycles. 109
5.3 The overview of the numerical applications. 111
5.4 The algorithms of Vector Graph Library. 111

6.1 Configurations of the simulation 134

List of Figures

2.1 The overview of the modern vector processor. 11
2.2 The breakdown of latency defined in this dissertation. 16
2.3 Indirect memory accesses. 16
2.4 The roof-line model of the scale kernel with indirect memory accesses. 18
2.5 The number of instructions for each iteration in the HPC applications. 19
2.6 TLB bandwidth and room for performance improvement. 21
2.7 The cache hit rate when the number of cores sharing the same cache and the number of the associativity are changed 23

3.1 Vector gather prefetcher. 42
3.2 Roof-line model of a scale kernel in the case of the sequential index array. 47
3.3 Roof-line model of a scale kernel in the case of the random index array. 48
3.4 The effects of parameters, *prefetch distance* & *prefetch degree* in the case of the scale kernel with the sequential index array. 51
3.5 The effects of parameters, *prefetch distance* & *prefetch degree* in the case of the scale kernel with the random index array. 52

3.6 The performance improvement when the number of cache blocks per vector gather instruction is decreased from the sequential case. 54
3.7 The performance improvement when the number of cache blocks per vector gather instruction is increased beyond the sequential case. 55

4.1 The percentage of the instructions causing pipeline stalling and their dependencies. 62
4.2 The challenges to distinguish dependency where the normal mode and the critical mode. 65
4.3 The overview of the criticality-aware out-of-order vector processor. 69
4.4 The performance evaluation results on the PolyBench suite, normalized by the baseline out-of-order configuration. 84
4.5 The performance evaluation results on six benchmarks, normalized by the baseline out-of-order configuration. 86
4.6 The performance evaluation results on the vector graph library, normalized by the baseline out-of-order configuration. 88
4.7 Effect of the different size of the DIQ. 91
4.8 Effect of the different size of the PCQ. 91
4.9 Effect of the different size of the CRAT. 92
4.10 Effect of the different size of the SIC. 94
4.11 Effect of the different switching latency. 94
4.12 Effect of physical vector registers. 95

5.1 The percentage of pages per vector gather instruction. 102
5.2 The cycles usable for page-address coalescing. 103
5.3 The example of the proposed address coalescing method. 104

5.4 The performance results for numerical applications. 112
5.5 The performance results for graph applications. 113
5.6 The performance improvement for each application. 113
5.7 The number of addresses translated by TLB. 114

6.1 Example of M cores sharing the same cache in the N vector cores processor 119
6.2 The elements used in one calculation 121
6.3 The elements shared on the cache 122
6.4 The cache hit rate when the number of cores sharing the same cache and the number of the associativity are changed 125
6.5 The 2-way set-associative. 127
6.6 The 2-way skewed-associative. 127
6.7 The bit field of hashing function of the given address 129
6.8 Cache hit rate result with the same hashing function, oDisp, except set-associative cache 135
6.9 Cache hit rate result with the same replacement policy, SRRIP . . 136
6.10 Performance comparison of the conventional set-associative cache and the proposed skewed cache 139

List of Algorithms

1 An example code of indirect memory accesses. 17
2 The sequence of indirect memory accesses by vector gather instructions. 40
3 VLDscale 45
4 VGTscale 46
5 3D 7-point stencil calculation 121
6 The flow of NRUNRW policy. 131
7 The flow of SRRIP policy 132

List of Source Codes

5.1 Example of indirect memory access 99

Chapter 1

Introduction

1.1 Introduction

We live in a world surrounded by computers. When checking weather forecast, watching movies, listening music, or traveling somewhere, we use computers in our daily life. Computers have become an integral part of our daily life, and they provide for our every need. As several types of computers support our lives, computers have been demanded to realize further performance improvements to enrich our lives.

In particular, computers in high performance computing are used to conduct scientific and engineering simulations. These computers are extremely sensitive to performance improvements. As personal computers now possess the same level of power as computers in high performance of several decades ago, it is anticipated that the advances in computers for high performance computing will disseminate throughout all computers in the future. The development of computers in high performance computing is therefore vital for the overall advances of computers.

The core part of computers is called processors, and the processors are one

of the most important parts for performance of the computers. The performance of processors can be determined by computing capability and memory performance. The computing capability means how many calculations in a second the processor can execute. Thanks to improvements in semiconductor technologies, the computing capability has continued to improve over the past decade. However, due to the end of Dennard's scaling and the stagnation of clock frequency, processors are required to improve performance while reducing power consumption.

The vector instruction set is one of the promising solutions to satisfy this requirements. The vector instruction set enables processors to handle multiple elements by one instruction. Since a compiler guarantees that the elements in a vector are independent, the processors can leverage data-level parallelism by processing the data in parallel by hardware. Many applications can potentially benefit from the vector instruction set for performance and energy efficiency [1]. The vector instruction set has been adopted not only for traditional high-performance computing (HPC) processors [2, 3] but also for general-purpose processors [4, 5].

The memory performance has also continued to improve over the past decades to sustain the computing capability improvements. The memory performance can be explained by bandwidth and latency. The bandwidth refers to the amount of data transferred per second, while the latency represents the time between a request for data and the actual receipt of the data. Although there have been significant advancements in terms of memory bandwidth, including the development of 3D die-stacking technologies, memory latency has lagged behind the bandwidth improvement [6].

Thanks to the architecture specialized for vector instructions, the processors

employing vector instruction set, namely vector processors, achieve high computing capability. Especially in high performance computing, vector processors can achieve high sustained performances. By adopting long vector length to enhance computing capability and a memory system focusing on high memory bandwidth, vector processors well work for scientific and engineering applications traditionally known as memory-intensive applications.

The performance requirements of the memory-intensive applications are increasing as they are developed to improve accuracy and expand the scope of the memory-intensive applications. For example, new memory-intensive applications, such as graph processing and machine learning, which rely on advanced algorithms and complex data structures, have become popular workloads. As a result, the memory system can easily become a bottleneck for these new memory-intensive applications. Vector processors have been expected to provide high sustained performance even for these new memory-intensive applications.

Some of numerical simulations and new applications require irregular memory accesses, and the high memory bandwidth of vector processors may not be fully utilized. One reason of this problem is a long latency, i.e., the time between issuing an vector instruction and completing its execution, resulting in degradation of performance. Irregular memory accesses are composed of a sequence of memory access instructions depending on other memory access instructions. If the first memory access instruction is delayed, the execution of its dependent memory access instruction is also delayed. In addition, irregular memory accesses may cause cache misses because these accesses may not have locality in the data reference. Therefore, it is difficult for vector processors to achieve high sustained performance in applications with irregular memory accesses.

Vector processors have not been considered a latency problem because their vector processing mechanism can hide latency. However, it become a performance bottleneck due to the following four reasons.

First, as semiconductor manufacturing technologies improves computing capability, the time required for each vector operation tends to become short. The latency hiding capability of vector processors is gradually decreasing.

Second, the number of instructions that the processor can handle simultaneously may be insufficient to exploit instruction-level parallelism from the program. Irregular memory accesses can be handled by a vector instructions set using a specialized instruction, and this handling requires multiple instructions for one access. This fact may result in a longer latency than the case of sequential accesses.

Third, in applications with irregular memory accesses, data may not hit in the cache system properly. Because the irregular memory accesses may not have locality in data references, the cache system cannot fully be utilized.

Fourth, memory bandwidth and memory capacity have been improved year by year, however, the memory access latency has not been reduced much compared to the memory bandwidth and memory capacity.

Therefore, architectures that make vector processors latency-tolerant are demanded.

1.2 Objective of the Dissertation

This dissertation aims to establish the architectonic methodology for vector processors that enables high sustained performance in memory-intensive applications from the viewpoints of latency-tolerance. The latency tolerance to be discussed in this dissertation is defined as the ability to achieve a high sustained performance in the case of irregular memory access or indirect memory access in memory-intensive applications. The latency-tolerant architectures can contribute to the resolution of latency-related problems when designing the next-generation vector processors. To this end, the following research items are clarified in this dissertation.

The first item is to clarify which architectural elements of vector processors causing latency problems and to establish approaches to solve these latency problems. It is also explained that the impact of the latency on the sustained performance of vector processors is significant and should be resolved.

Based on the impact of latency on vector processors clarified in this dissertation, architectures to make vector processors latency-tolerant are proposed. The second item is to show that the architectural approaches for making vector processors latency-tolerant, which is stated in the first item, are evaluated in various memory-intensive applications. In particular, this dissertation shows four approaches. The first approach is a prefetch mechanism for indirect memory access through vector instructions. The second approach is an out-of-order execution mechanism according to the instruction latency of vector instructions. The third approach is a method for reducing conversion latency by optimizing address translation of vector instructions. The fourth approach is a mechanism for reducing memory access latency by reducing conflict misses in the cache mechanism for vector processors.

These architectural approaches enable vector processors to achieve high sustained performance in memory-intensive applications with irregular memory access or indirect memory access. These approaches expand the range of applications for vector processors, contributing to the advancement of science and technology.

1.3 Organization of the Dissertation

This dissertation is organized as follows. Chapter 1 describes the background and the objective of this dissertation. This chapter highlights the importance of addressing latency tolerance in vector processors to achieve the sustained performance.

Chapter 2 describes an architecture of vector processors assumed in this dissertation. The vector processors are crucial in high-performance computing (HPC) processors and general purpose processors, especially for memory-intensive applications such as scientific and engineering endeavors. However, the performance of vector processors can be compromised by the latency, i.e., the time between issuance of a vector instruction and its completion, especially for applications with irregular memory access patterns. Providing the definition of latency in the vector processors, this chapter discusses the sources of the latency that can be a bottleneck for the vector processors. Then, this chapter breaks down this long latency into four types.

Chapter 3 proposes a prefetching mechanism for vector gather instructions to hide the latency in the aspect of dependency among memory instructions. The proposed mechanism consists of two prefetchers. The first prefetcher loads sequential data that are indices for indirect memory accesses. Using the prefetched data, the second prefetcher calculates the addresses of indirect memory accesses and prefetches these data.

Chapter 4 proposes a criticality-aware out-of-order mechanism for the vector processors to hide the latency due to stalling. The out-of-order mechanism is crucial to hide latency by executing vector instructions in an out-of-order manner. The proposed mechanism identifies vector instructions that cause stalling

and tracks the dependencies of these instructions. Once the stalling vector instructions are identified, the mechanism issues them while the vector processors are stalling, allowing the processors to utilize unused resources.

Chapter 5 proposes an address coalescing method for vector gather instructions to reduce the latency in the aspect of address translation. The proposed method aims to reduce the latency of address translation by using the arithmetic vector units already built into the vector processors to deduplicate the virtual addresses of vector gather instructions. By deduplicating these addresses, the number of address translations is reduced, thereby reducing the number of cycles required for address translation.

Chapter 6 proposes a skewed cache mechanism to reduce the latency in the aspect of the conflict miss. Since the vector processors employ multiple-core architecture, the cache system receives many requests from multiple cores. This situation causes the way-conflict of the cache due to the shortage of associativity. The proposed mechanism reduces cache misses using the skewed associativity to avoid conflict misses.

Finally, Chapter 7 concludes this dissertation and discusses remaining issues as the future work.

Chapter 2

Importance of latency-tolerance for vector processor architectures

2.1 Introduction

This chapter discusses the importance of the latency-tolerant vector processor architectures. First, this chapter shows the vector processors assumed in this dissertation. The vector processors implement vector processing units and high memory bandwidth. Then, this chapter defines the latency that this dissertation addresses. From the definition of the latency, this chapter breaks down the latency into four latencies with architectural factors.

While vector processors are well-suited to handle regular memory access patterns that can be easily vectorized, they struggle to achieve high sustained performance in applications that require irregular memory accesses. This is because such accesses often consume resources for an unnecessarily long duration, resulting in a decrease in resource utilization.

2.2 Modern vector processors and their requirements

Since vector processors appeared in the early 1970s, vector processors have been used for scientific and engineering simulations in high performance computing [7]. The design of vector processors enables the processors to exploit data-level parallelism from programs. Successors to the original vector processors, namely modern vector processors, realize high sustained performance for several memory-intensive applications.

First, this section explains the features of modern vector processors that are inherited from traditional vector processors. This section then describes the points that modern vector processors have evolved from traditional vector processors. The modern vector processors inherit traditional vector processors, such as a powerful core with the capability of vector processing with long vector length, and high bandwidth memory.

Figure 2.1 depicts a modern vector processor assumed in this dissertation. The modern vector processor contains multiple vector cores. The vector core consists of a scalar processing unit and a vector processing unit. The scalar processing unit fetches and decodes all the instructions and is responsible for the execution of scalar instructions in the subsequent pipeline.

The vector processing unit consists of vector functional units and vector registers. The vector functional units execute vector instructions that perform arithmetic or logical operations on vectorized data. Multiple elements of vectorized data are handled by a single vector instruction. *Vector length* refers to the number of elements that can be processed by a single vector instruction. For example, the maximum vector length is 256 for double-precision floating-point

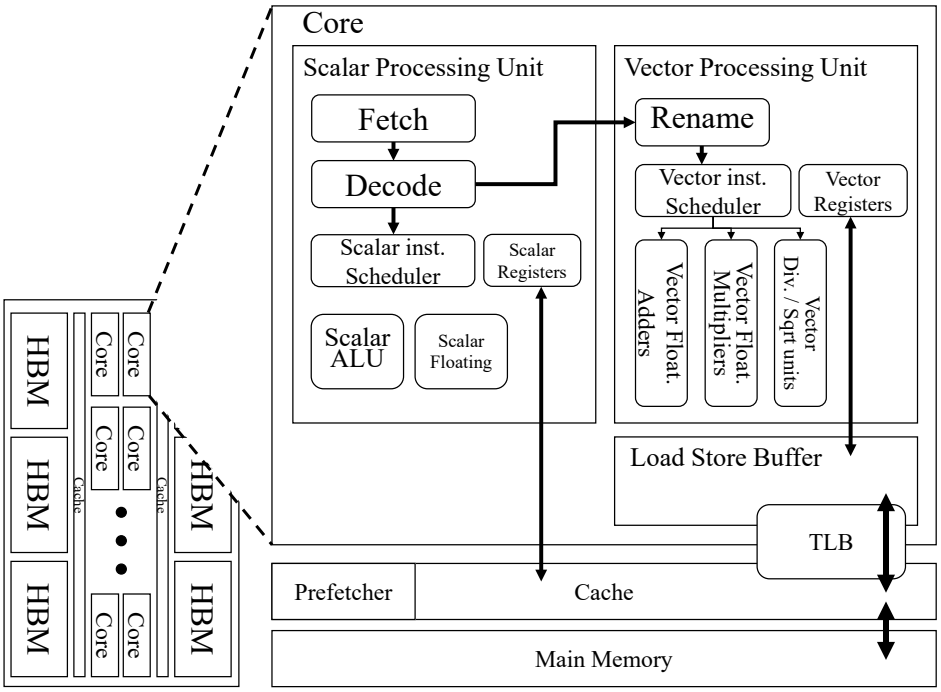


Figure 2.1: The overview of the modern vector processor.

elements in the NEC SX vector processor series, such as SX-ACE [8, 3] and SX-Aurora TSUBASA [2, 9, 10]. Compared to SIMD instructions of general-purpose processors [4, 5, 11, 12, 13, 14], vector instructions can process a vast amount of data with a single instruction.

Vector registers hold vector data used in vector instructions during execution, and vector functional units can access these data for calculations. These units contain multiple arithmetic pipelines for various operations such as addition, multiplication, division/square root, and logical operations. Vector mask registers represent the validity of the results of vector instruction execution, and they can be used to vectorize loops, including conditional branches and execute them as vector instructions.

As traditional vector processors realize high memory bandwidth, the modern vector processors also realize high memory bandwidth. To realize high sustained

memory bandwidth, the modern vector processors adopt high bandwidth memory (HBM). Compared to conventional memory such as DDR4, HBM can provide more bandwidth by stacking multiple memory dies as a package. HBM with up to eight stacked DRAM dies is connected to the memory controller via silicon interposer. The dies are connected vertically to each other inside the stack using through-silicon via (TSV) technologies. Since HBM has many memory channels and wider bus, its memory bandwidth is higher than conventional DDR memories [15]. The modern vector processors implement several packages of HBM. For example, SX-Aurora TSUBASA implements six HBM packages [16, 17].

Furthermore, modern vector processors have employed four architectural mechanisms, such as multiple vector cores, an out-of-order execution mechanism, virtual memory system, and cache memory, over the traditional design to satisfy the demand from applications.

First, the modern vector processors contain many vector cores allowing for more computational capacity, although the vector processor is composed of a small number of high performance vector cores, instead of implementing a large number of low performance cores in the case of a scalar general-purpose processor such as X86 processors. The modern vector processors achieve computational capacity by achieving process-level parallelism through multi-core technologies.

Second, the modern vector processors adopt an out-of-order execution mechanism. The vector processing unit schedules vector instructions in an out-of-order fashion. This out-of-order mechanism allows vector instructions to be executed as soon as they become available, hence enhancing performance by exploiting instruction-level parallelism.

The out-of-order execution mechanism is enabled by a larger number of physical vector registers than logical vector registers specified by the instruction set

architecture (ISA). The registers defined by the ISA are referred to as logical registers, whereas the registers that the processor really contains are referred to as physical registers in this dissertation. Since the vector processing unit renames vector registers from logical registers to physical registers, vector instructions can circumvent false dependency of vector registers.

Third, the modern vector processors support a virtual memory system, which decouples the process-visible memory space from the actual memory space. This functionality allows many programs to seamlessly utilize the memory space and decreases the programming burden for developers [7]. The process of converting addresses from the process-visible space to the actual memory space is called address translation. During address translation, a virtual memory address is divided into a virtual page number and a page offset, where the virtual page number will be translated to a physical page number.

In order to reduce the cost of address translation, a translation lookaside buffer (TLB) is used to cache the mapping information between virtual page numbers and physical page numbers. When a virtual page number is translated to a physical page number, the TLB is accessed in the load/store queue to translate the addresses for each vector memory instruction. This mechanism reduces the time required for subsequent translations of the same virtual page number.

Fourth, the modern vector processors adopt multi-banked caches to keep the high sustained memory bandwidth. With the improvement of the computing capability of vector cores, the demand for memory performance also increases to supply the data required by vector cores. However, the improvement of memory performance is relatively behind that of computing capability even with the

debut of HBM. Since it is difficult to further increase the off-chip memory capacity owing to the limited number of input/output ports on a single chip, on-chip caches feed vector cores with reusable data at high bandwidth. For example, the SX-9 and later vector processors in NEC SX series include on-chip multi-banked cache memories [18, 2, 19].

Modern vector processors are widely used for memory-intensive applications, such as scientific simulations and engineering applications, for example, tsunami inundation simulation [20], oil and gas applications [21], superconductivity simulations [22], drug design application [23], liquid crystal simulation [24], and analytical query processing [25]. Furthermore, the modern vector processors have been used in various range of emerging research and productions, such as simulated annealing applications [26, 27], graph applications [28, 29, 30], and machine learning applications [31]. The performance requirements of these applications are growing as these applications are eagerly developed to satisfy the demands of improving their accuracy and broadening the scope of applications.

2.3 Latency problems of vector processor

The term “latency” is used to express the delay, such as the memory access latency that is the time between sending a request and receiving data from memory. The memory access latency has a significant impact on the completion time of a vector instruction. The completion of the vector instruction has a significant impact on the execution of dependent vector instructions. Accordingly, the latency has a significant impact on the overall performance of vector processors.

To clearly discuss the impact on vector processor performance, this dissertation expands the concept of the latency in a broader sense that more directly affects sustained performance. Here, this dissertation defines “latency” as the time between the fetch of a vector instruction and the completion of the vector instruction.

The latency is the sum of a number of different sources of the latency within the architecture. From the definition of latency in this dissertation, this dissertation breaks down the latency into four types as shown in Figure 2.2.

2.3.1 Latency of preceding dependent instructions

The execution of some memory access instructions may be contingent upon completing another memory access instruction. This kind of memory access instruction cannot be executed until the result of the previous memory access instruction has been completed.

One example of instruction sequences having dependency is indirect memory accesses. Algorithm 1 represents a simple program including indirect memory accesses. In the indirect memory accesses, the index array determines the addresses where they access. In Algorithm 1, the elements in array L determine

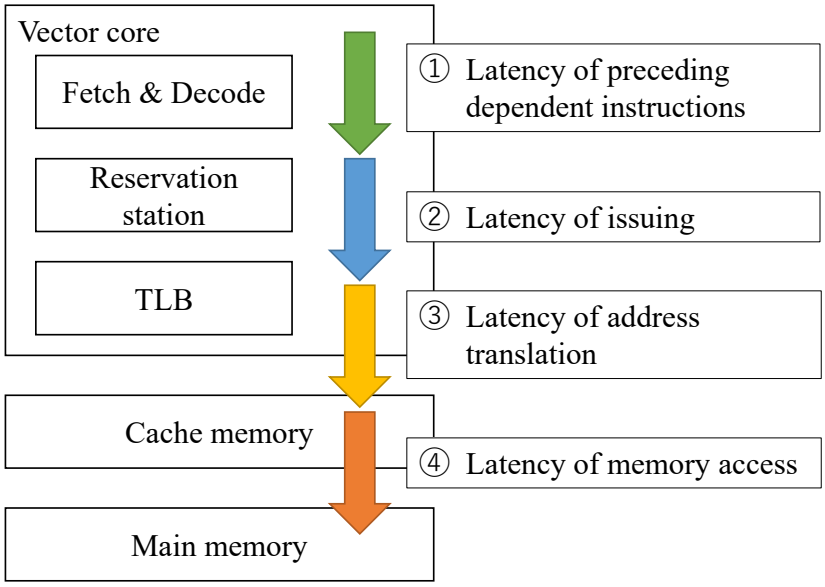


Figure 2.2: The breakdown of latency defined in this dissertation.

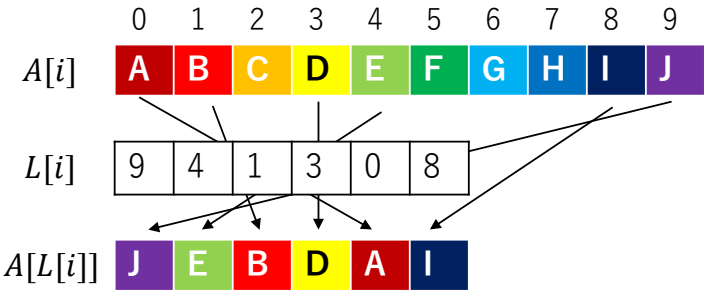


Figure 2.3: Indirect memory accesses.

how the elements in array A are accessed, as shown in Figure 2.3. The elements of array A are accessed indirectly through array L, for example $A[L[i]] = J, E, B$ while $A[i] = A, B, C$ for $i = 0, 1, 2$. In general, the values of array L are not consecutive, resulting in irregular access patterns for $A[L[i]]$.

Applications such as numerical calculations, machine learning, and graph algorithms often rely on indirect memory accesses. For instance, a sparse matrix with many zero elements is often compressed to reduce memory consumption, requiring indirect memory accesses to access each non-zero element. Similarly, machine learning and graph algorithms also use sparse matrices that require

Algorithm 1 An example code of indirect memory accesses.

1: **for** $i = 0, 1, ...N$ **do**
2: $\quad B[i] = s \times A[L[i]]$
3: **end for**

indirect memory accesses.

To examine the performance degradation by indirect memory accesses, this dissertation conducts preliminary evaluation. Figure 2.4 shows the preliminary evaluation of the scale kernel with/without indirect memory accesses, shown in the roof-line model [32]. The vertical axis shows the performance, and the horizontal axis shows operational intensity that is the amount of floating operations divided by the amount of memory accesses. *Scale* indicates the scale kernel as shown in Figure 2.3, and *Scale with indirect memory accesses* indicates the case of using indirect memory accesses. The index array contains continuous values. *Scale* obtains performance near the ceiling of the roof-line model that represents an upper bound of performance. However, the obtained performance of *Scale with indirect memory accesses* is far from the performance of *Scale*. This suggests that the latency of indirect memory accesses hinders the ability of the vector processor to utilize its memory bandwidth.

2.3.2 Latency of vector instruction stalling

Modern vector processors can execute vector instructions in an out-of-order fashion to enhance instruction execution efficiency. However, due to the limitation of hardware budget of the processor, there is a limit to the number of vector instructions that the processor can analyze for their readiness. If instructions take memory access latency, the execution of subsequent instructions will be

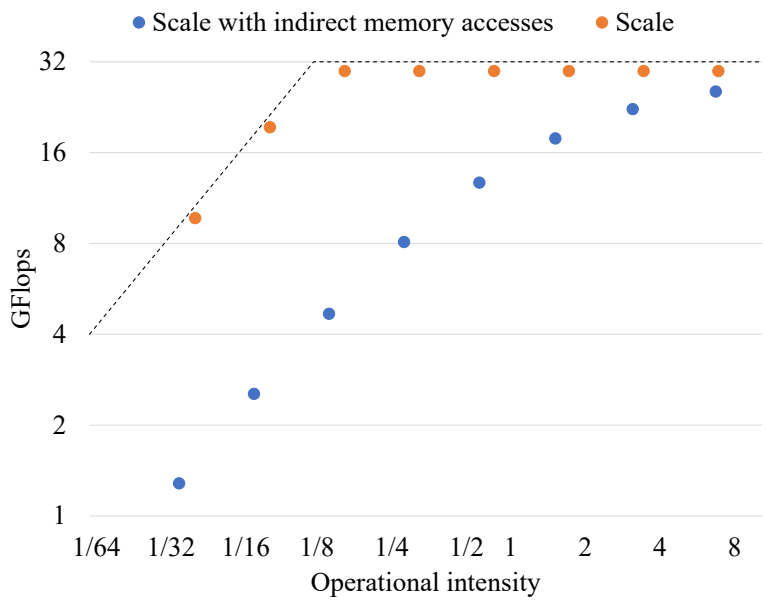


Figure 2.4: The roof-line model of the scale kernel with indirect memory accesses.

stalled.

Vector processors have traditionally been used to execute applications such as numerical simulations. These applications often involve repeating the same loop structure many times to compute large computational spaces or time steps to iteratively solve mathematical or physical equations. This means that the same vector instructions tend to be executed repeatedly, including those with a long access latency that may degrade performance.

Additionally, these applications have hundreds of instructions in one iteration. Figure 2.5 shows the number of instructions in one iteration on the application for vector processors. The vertical axis shows the number of vector instructions inside one iteration, and the horizontal axis shows applications. This chapter prepares six kinds of applications from different application domains [19, 2]. Each application has a single loop structure that iterates multiple times. These applications are compiled for the vector processors and optimized

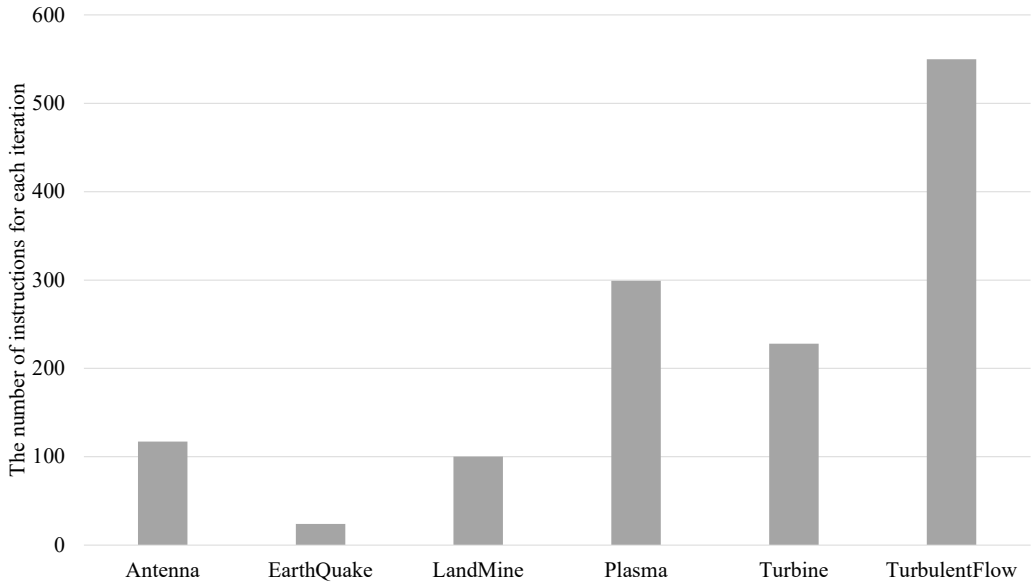


Figure 2.5: The number of instructions for each iteration in the HPC applications.

so that 99% part of the code can be vectorized. The detail of the evaluation environment will be shown in Section 5.4. Figure 2.5 shows that the number of instructions in one iteration varies depending on applications. The **TurbulentFlow** kernel especially has many instructions appearing in one iteration, making it easy to fill the out-of-order window with instructions of one iteration. The **Plasma** and the **Turbine** kernels also involve hundreds of instructions in one iteration. Note that in the cases of the **LandMine** and **EarthQuake** kernels, the number of instructions appearing in one iteration is small because they are stencil calculations without loop unrolling.

Since vector instructions for memory accesses appear frequently, a large number of vector instructions between each memory access should wait for their execution. This can hinder conventional out-of-order mechanisms from effectively paralleling these memory accesses. The processor may not have sufficient capacity to simultaneously handle the instruction-level parallelism present in the program, resulting in the latency as instructions must wait to be issued.

For example, the Fujitsu A64FX processor, which utilizes the vector instruction set to take advantage of data-level parallelism, has 79 reservation stations connected to its execution pipeline, enabling it to handle a maximum of 79 instructions simultaneously [33]. This limitation on the number of instructions that can be handled at once can prevent the processor from fully exploiting the instruction-level parallelism present in the applications that commonly run on vector processors.

The inability of the processor to exploit the ILP causes vector instructions waiting to be executed. This can result in latency. Therefore, there is potential to improve the performance of vector processors by enabling them to exploit the ILP across iterations of vector instructions.

2.3.3 Latency of address translation

Since vector instructions can handle multiple elements in a single instruction, all addresses of the instruction must be translated. This translation process may take a long time for the translation of all addresses, resulting in a long latency.

In the case of address translation for vector instructions, addresses should be translated as many as vector elements. Supposing that consecutive accesses can be expected, such as in the case of vector load instructions, the minimum number of required pages can be predicted from the range of virtual addresses.

On the other hand, in the case of indirect memory accesses, where memory accesses are not sequential, and the addresses are unpredictable, it is necessary to translate all the virtual page numbers in the addresses to physical page numbers. The number of virtual page numbers that need to be translated is proportional to the vector length. The vector processor with a long vector length

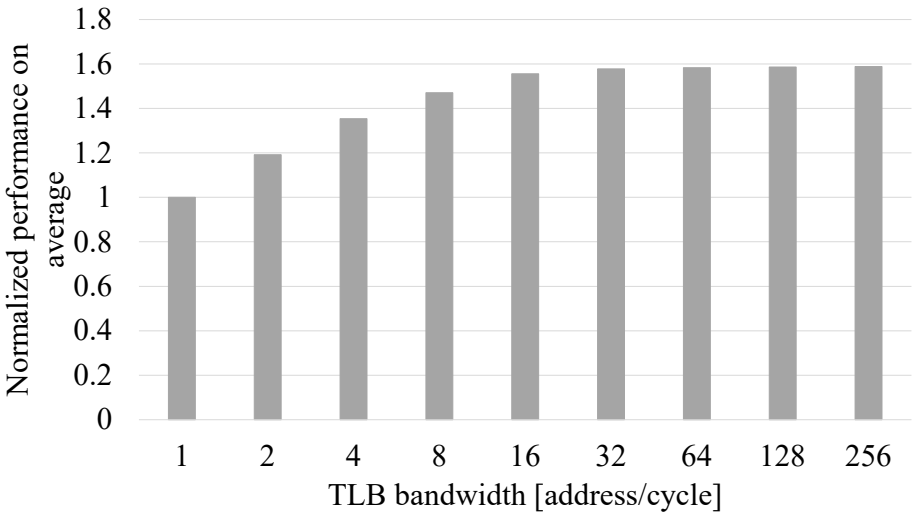


Figure 2.6: TLB bandwidth and room for performance improvement.

consumes a non-negligible number of cycles for the address translation.

To confirm this fact, the preliminary experiment is conducted. The vector processor with a long vector length executes several benchmarks, such as numerical computation and graph algorithms containing indirect memory accesses using a simulator. Figure 2.6 shows the impact of the TLB performance on the total performance in handling vector gather instructions. The vertical axis shows the average performance and the horizontal axis shows the throughput regarding the number of pages that the TLB can translate per cycle (TLB bandwidth). This figure indicates that the performance increases as the TLB bandwidth increases.

The TLB requires many cycles to translate all virtual addresses of vector instructions when the bandwidth is limited. As a result, the latency of vector instruction is affected more by the limited TLB bandwidth. Although increasing the bandwidth of the TLB is challenging, the time spent on the TLB must be minimized.

2.3.4 Latency of memory access

The vector processor is equipped with cache memories to realize high sustained bandwidth. The cache memory is also important for reducing the memory access latency. However, if the cache cannot cover the memory access pattern, the accesses become cache misses, resulting in a long latency. Furthermore, when the vector processor has multiple cores, the accesses that the cache receives become more complex.

To evaluate the effect of multiple accesses from multiple cores into the caches, the preliminary experiment is conducted. Figure 2.7 shows the results of the preliminary evaluation in a stencil calculation. The vertical axis indicates the cache hit rate, and the horizontal axis indicates the number of cores sharing one cache with an associativity of four. In Figure 2.7, the theoretical cache hit rate, which is calculated by the memory access pattern, increases as the number of cores sharing one cache increases. On the other hand, the cache hit rate of the set-associative cache with the LRU replacement policy becomes significantly low in the case of a large number of cores sharing one cache with an associativity of four. This is because, when the stencil calculation is executed in parallel, multiple cores simultaneously access the same set, resulting in conflict misses.

When the cache hit ratio is low, two issues may arise. First, the sustained memory bandwidth is reduced due to the failure to exploit data locality. Second, the memory access latency increases because all data must be transferred from the main memory. On the other hand, if the cache hit ratio becomes high, the processor not only benefits from high sustained memory bandwidth but also becomes more tolerant of latency.

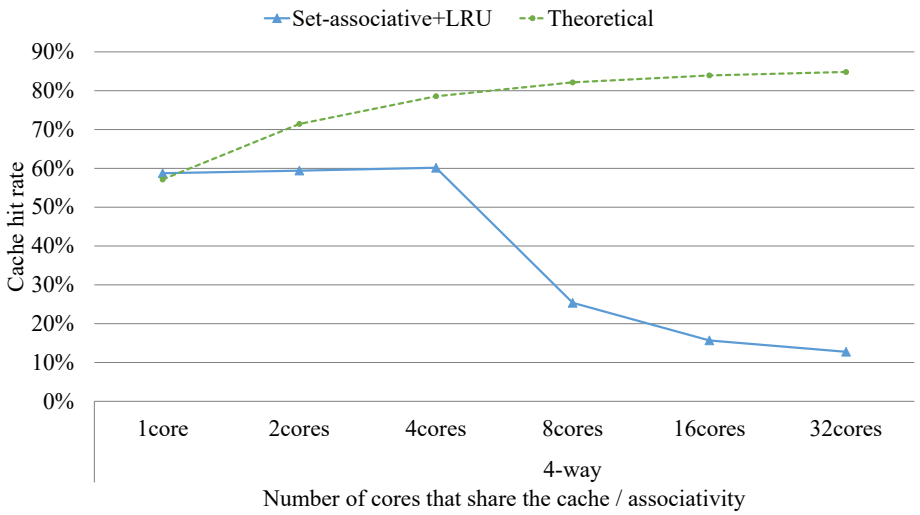


Figure 2.7: The cache hit rate when the number of cores sharing the same cache and the number of the associativity are changed

2.4 Latency-tolerant vector processor architectures

This dissertation aims to present architectural approaches to designing latency-tolerant vector processors. As discussed in Section 2.3, latency-tolerance is an important factor in achieving high sustained performance in memory-intensive applications on modern vector processors. Since vector processors have traditionally been designed to achieve throughput rather than the latency, it is necessary to develop latency-tolerant architectures to achieve high sustained performance.

This dissertation proposes four approaches from two perspectives as shown in Table 2.1. There are two perspectives that can be taken to solving latency problems in modern vector processors. The first perspective is to consider multiple ways to solve latency problems, as different mechanisms may be more effective in different scenarios. This includes both approaches to hide latency and reduce latency. The second perspective is the need to consider both the core architecture and memory architecture of modern vector processors, as both can significantly impact the overall performance of the processors.

For the first perspective, there are two approaches to solve latency problems discussed in Section 2.3 in modern vector processors. The first approach is to hide latency, which involves ways to continue executing instructions even in the presence of latencies. This can be achieved through various architectures such as an out-of-order execution, and speculative execution. The second approach is to reduce latency, which involves implementing features or taking procedures to minimize latency. This can be achieved through architectures such as improving cache utilization, and reducing the number of cycles required for address

Table 2.1: The relation of this dissertation between chapters and two perspectives with four approaches.

	Hiding latency	Reducing latency
Core architecture	An out-of-order mechanism in Chapter 4	A page-address coalescing method in Chapter 5
Memory architecture	A prefetching mechanism in Chapter 3	A skewed cache in Chapter 6

translations.

For the second perspective, there are two architectural approaches to solve latency problems in modern vector processors. The first approach is to optimize the core architecture of the processor, which involves designing the processor to handle latencies of as an out-of-order execution and a virtual memory. The second approach is to improve the memory architecture, which involves the design of the cache system and a prefetching mechanism to handle latencies.

From these perspectives, this dissertation proposes four approaches. First, this dissertation proposes a prefetch mechanism. This is the approach to hiding the latency of memory access instructions that depend on each other, such as indirect memory accesses, for the memory architecture. The idea is that a prefetcher can identify the index data needed for the indirect memory accesses and load it in advance, allowing the processor to overlap memory accesses and improve performance. However, prefetching also carries the risk of loading redundant data, which can reduce the overall benefit. Moreover, the vector instructions used in many memory-intensive applications handle multiple elements in a single instruction, and therefore, the negative performance impact of prefetching misses can be significant. To overcome this trade-off, it is necessary to design a prefetcher that is specialized for vector instructions and can effectively hide latency while minimizing the risk of loading redundant data.

Second, this dissertation investigates the use of speculative execution techniques. This is the approach to hide the latency of issuing instruction in vector processors, for the core architecture. While an out-of-order execution mechanism can be effective at hiding latency, it can also be resource-intensive and energy-consuming. Instead, this dissertation focuses on the runahead mechanism to hide latency at the time of instruction issue. The runahead mechanism is a form of speculative execution that exploits instruction-level parallelism beyond the scope of out-of-order execution, allowing the processor to hide latency and improve performance. However, because vector processors often have large amounts of data, this dissertation proposes the mechanisms to reduce the redundant bandwidth occupancy between vector cores and caches by utilizing the data discarded speculatively in runahead mechanisms.

Third, this dissertation proposes a method that deduplicates virtual addresses before address translations. This is the approach to reduce the latency of address translations in vector processors, for the core architecture. In vector processors, indirect memory accesses require multiple virtual address translations, which can consume significant TLB bandwidth. To solve this issue, an approach to use hardware coalescers has been proposed in GPU systems [34]. However, this approach can be resource-intensive and may not be feasible in systems with limited area budgets. The proposed method in this dissertation leverages the existing vector arithmetic units in the vector processor to perform address deduplication, eliminating the need for additional hardware and reducing the latency of address translations.

Finally, this dissertation tackles to reduce cache conflict misses in vector processors. This is the approach to reduce the latency of memory accesses, for the

memory architecture. One of the methods to reduce conflict misses is to increase the associativity; however, it is challenging to realize the higher associativity because of cost, power consumption, and area overheads on the chip in large-capacity and multi-banked caches. Therefore, this dissertation discusses another way to eliminate conflict misses instead of increasing the associativity.

In summary, the latency-tolerant vector processor architectures proposed in this dissertation can meet the following requirements.

- Minimizing performance degradation due to dependencies between vector memory instructions. Chapter 3 proposes indirect memory access prefetcher for vector gather instructions.
- Hiding latency by leveraging instruction-level parallelism. Chapter 4 proposes criticality-aware out-of-order mechanism for vector instructions.
- Avoiding performance degradation due to address translation of vector instructions. Chapter 5 proposes page-address coalescing method of vector gather instructions.
- Achieving high cache hit rate for vector instructions. Chapter 6 proposes skewed multi-banked cache for many-core vector processors.

2.5 Related work

This section introduces studies tackling to solve the factors discussed in Section 2.4 and addresses the approaches discussed in this dissertation among them.

2.5.1 Studies of indirect memory accesses

Several papers have addressed the problem of indirect memory access through prefetching. Ainsworth *et al.* [35] have developed a compiler pass that generates software prefetches for indirect memory accesses and evaluated it across various in-order and out-of-order architectures. Their implementation has shown average speedups between 1.1x and 2.7x for memory-intensive benchmarks. However, their work focuses on scalar processors and scalar instructions rather than vector instructions. It does not clarify the effects of their indirect memory access mechanism on vector gather instructions.

Al Farhan *et al.* [36] have proposed software optimizations, including software prefetching, for indirect memory accesses in the context of vector instructions. These optimizations exploit prefetching instructions provided by AVX-512, such as `VGATHERPF0DPD` and `VGATHERPF1DPD`, which utilize the vector units for non-blocking prefetching. In their approach, some gather prefetching instructions are performed before the data is required. The evaluation results show that these software optimizations achieve a 2.9x speedup on the Intel Knights Landing processor.

Ainsworth *et al.* [37] have proposed a programmable prefetcher for irregular memory access patterns, which provides more flexibility for prefetching beyond specific data structures. In addition, they have developed compiler techniques

to automatically assist the prefetcher by analyzing source code, enabling it to prefetch more complex access patterns, including indirect access patterns.

These studies indicate that prefetching benefits indirect memory accesses, although they have not clarified the effect on the vector instruction set. Therefore, this dissertation focuses on a hardware prefetcher for indirect memory accesses by vector gather instruction.

2.5.2 Studies of instruction stalling

Dundas *et al.* [38] have proposed runahead execution that speculatively executes future instructions to exploit ILP. Mutlu *et al.* [39] have introduced and evaluated the runahead execution for out-of-order processors. The runahead execution enables the processor to make the opportunities to hit the cache by exploiting instruction-level parallelism for accurate prefetching.

There are several optimization techniques that can improve the efficiency and performance of the runahead execution. Hashemi *et al.* [40] have proposed a mechanism of filtering only the instructions that cause stalls and storing them in a runahead buffer. This mechanism allows the front-end of the pipeline to halt, which can improve energy efficiency. Naithani *et al.* [41] have clarified that processors have sufficient resources such as the issue queue and physical registers when entering the runahead mode. This insight enables processors to avoid releasing the resources for the runahead execution. They also proposed a mechanism to track multiple instruction chains that cause stalling. Ramirez *et al.* [42] have proposed using another thread to conduct runahead execution.

The previous proposals for optimizing the runahead execution have primarily focused on scalar processors executing general applications. Many of the

latest scalar processors have adopted vector extensions in order to improve performance. Since vector instructions handle vectorized multiple data, unlike the scalar register on scalar processors, the penalty of flushing physical registers and executing them again is not insignificant. This dissertation focuses on vector processors and tries to retain the data executed by the runahead execution to avoid the penalty of flushing physical registers.

Srinivasan *et al.* [43] have proposed continual flow pipelines. The continual flow pipelines drain the stalling instruction chains from the ROB and re-execute them after the data arrives. Hilton *et al.* [44] have proposed BOLT that improves energy efficiency by reusing SMT hardware to rename deferred instructions, which are the stalling instructions and their dependencies. The difference between the proposed mechanism in this dissertation and the latency tolerant executions is that the proposed mechanism executes subsequent instructions that may cause stalling. In contrast, the latency tolerant executions temporarily drain stalling instructions to the buffer and re-execute them later. The proposed mechanism in this dissertation does not require the re-execution.

Deshmukh *et al.* [45] have proposed criticality driven fetch (CDF) that focuses on the criticality of instructions and reflects the criticality for allocating the processor resources. The CDF treats load instructions as critical and solves difficult-to-predict branch instructions in advance of the other instructions, resulting in significant performance improvements and a reduction in power consumption for general-purpose processor workloads. The basic idea of the proposed mechanism in this dissertation is similar to the CDF; however, the proposed mechanism is much simpler than the CDF. Since the target processors and applications in this dissertation are further specialized for well-vectorized codes

with mask instructions than those targeted by the CDF, the proposed mechanism can realize performance improvement without the complexity for solving branch instructions.

There is a coordinated software and hardware approach to exploit ILP. Tran *et al.* [46] have proposed the SWOOP compiler that divides a program into access and execute phases. By reordering access phases and keeping execution phases in the program order, the SWOOP compiler can exploit memory and instruction level parallelism on in-order processors. For the SWOOP compiler to enable this reordering, the hardware provides lightweight information about register renaming and cache misses. The proposed mechanism in this dissertation can identify the stalling load instruction without assistance by the compiler.

2.5.3 Studies of address translation

Vesely *et al.* [47] have highlighted the issue of address translation potentially causing a bottleneck in applications with irregular memory accesses on GPUs. Their study aims to mitigate the negative impact of TLB misses. This dissertation also notes that TLB bandwidth can be a bottleneck, even in the cases where all TLB hits occur in vector gather instructions.

Puthoor *et al.* [48] have proposed a compiler-assisted method for coalescing. They analyzed the complexity of hardware coalescing and subsequently proposed a technique in which the compiler provides a clue to the processor to ensure that the requests of 64 work-items, i.e., a wavefront, fall within the same virtual page. This clue allows the hardware to only check if the first and last addresses of the wavefront are on the same page. If these addresses are on the same page, all addresses within the wavefront are contained within a single page, enabling translations to be performed with a single TLB access. Whereas

their proposal requires a clue from the compiler, the proposed method in this dissertation does not require any clue from the compiler.

The A64FX processor, which employs ARM SVE [33, 49], handles vector gather instructions by dividing the vector elements into pairs of two elements before address translation. If these pairs fit within a 128-byte space, the accesses are combined, reducing the number of TLB accesses required. However, this approach only checks virtual addresses pairwise, whereas the method proposed in this dissertation examines the entire set of virtual addresses contained within a vector register.

2.5.4 Studies of vector cache

Some research has attempted to address the issue of conflict misses. Qureshi *et al.* [50] have proposed the V-Way cache, which incorporates a flexible tag mechanism. This mechanism allows global replacement to eliminate conflict misses. The V-Way cache has extra tag entries that vary associativity in a set on-demand. These tag entries are linked to data entries through indirection pointers. This allows the selection of victims from a global perspective, rather than being confined to the same set. However, the V-Way cache incurs significant hardware costs and has a latency in finding a victim for replacement.

Sanchez *et al.* [51] have proposed the ZCache, which aims to solve conflict misses due to a lack of replacement candidates on eviction. The ZCache addresses this issue by selecting multiple replacement candidates using multiple hashing functions for the same set on each miss. If a necessary block is about to be evicted, another useless block from a different set is chosen for eviction, allowing the necessary block to be relocated to and retained in the cache. However, the ZCache requires multiple array lookups to find a block in the cache and

incurs additional overhead due to data movement during relocations, which can be costly for caches that prioritize high bandwidth.

While these approaches effectively reduce conflict misses, they come with significant hardware overhead. Additionally, their effectiveness has only been evaluated in the context of scalar processors, not vector processors. As a result, this dissertation seeks a simpler way to reduce conflict misses for vector processors, revisiting the concept of skewed-associativity [52].

There have been various investigations for the benefits of incorporating caches in vector processors. Musa *et al.* [53] have demonstrated that a vector processor with either one or two shared caches can achieve a 15-40% increase in performance compared to a version without caches. This is due to the fact that neighboring cores can utilize the shared cache to access previously stored data, leading to an improvement in cache hit rate and overall performance. However, their research have only examined vector processors with up to 4 cores. As modern vector processors have eight or more cores, it is necessary to study the effects of shared caches on these newer processors with a large number of cores. Additionally, this study does not consider the impact of cache associativity.

Egawa *et al.* [54, 55] have conducted a study on the effects of shared caches in multi-core vector processors with up to 16 cores, using real applications. They found that increasing the number of cores and the capacity of the shared cache lead to an improvement in the cache hit rate. In addition, the shared cache was shown to enhance performance efficiency and reduce power consumption. It was also determined that an 8MB shared cache configuration is optimal for a 16-core vector processor. However, their research did not address the impact of cache associativity to be discussed in this dissertation.

2.6 Conclusions

This chapter explains the modern vector processors assumed in this dissertation. To realize high sustained performance for the modern vector processors, this dissertation focuses on latency of the modern vector processors. This chapter redefines latency in this dissertation. The "latency" is defined as the time between the fetch of a vector instruction and the completion of the vector instruction.

This chapter discusses the breakdown of the latency by four factors. The first factor is due to dependencies among vector memory instructions caused by indirect memory accesses is a factor that can degrade performance even with sufficient bandwidth in the vector processor. The second factor is that the ability to hide latency by the out-of-order execution mechanism is limited for applications with a large number of instructions per loop, such as commonly run on the vector processor. The third factor is caused by the address translation required for every memory access in the vector processor with virtual memory can be a bottleneck for vector memory instructions. The fourth factor is the memory access latency that is closely related to the cache hit ratio for the vector processor.

To address the problems related to these latencies, this chapter examines two perspectives. The first perspective involves exploring various ways for mitigating latency, including two approaches to hiding latency and reducing latency. From this perspective, the first approach to hiding latency can be achieved through out-of-order and speculative execution. The second approach to reducing latency involves implementing features or adopting procedures that minimize latency.

The second perspective to solving latency problems involves taking into account both the core and memory architectures of modern vector processors.

From this perspective, there are two approaches to mitigating latency problems. The first approach is to design the processor to handle latencies through out-of-order execution and virtual memory. The second approach involves improving the memory architecture, including the design of the cache system and a prefetching mechanism to solve latency problems.

From these perspectives, this chapter clarifies the requirements to make the modern vector processors latency-tolerant by four approaches.

Chapter 3

Indirect memory access prefetcher for vector gather instructions

3.1 Introduction

Indirect memory accesses used to access data stored at non-contiguous or sparse memory locations, are often characterized by low spatial and temporal locality due to the non-sequential nature of the index data that determine the accessed data. As a result, these access patterns tend to incur additional latencies and bandwidth overheads in order to bring the necessary data into the processor. These issues can become a performance bottleneck for vector processing.

This chapter presents a hardware prefetching technique called a vector gather prefetcher that aims to improve the performance of vector gather instructions responsible for indirect memory accesses in the vector instruction set. The vector gather prefetcher operates in two phases: first, it prefetches index data that

are accessed sequentially; second, it predicts the addresses for indirect memory accesses using the prefetched index data and attempts to prefetch the data at these predicted addresses. The vector gather prefetcher aims to mitigate the impact of the increased latency caused by indirect memory accesses.

3.2 Motivation

This section introduces the prefetcher that tackles to improve the performance of indirect memory assesses. This section describes how indirect memory accesses are handled by the vector instruction set.

3.2.1 Indirect memory access prefetcher

Yu et al. [56] have investigated the use of a hardware prefetcher to address indirect memory accesses in their work, and proposed an Indirect Memory Prefetcher (IMP). IMP is composed of three components: a stride prefetcher, an indirect pattern detector, and an address generator. The stride prefetcher detects and prefetches index data, while the indirect pattern detector identifies pairs of index and indirect access data. If such a pair is found, the address generator generates the addresses for future indirect memory accesses to be prefetched.

IMP focuses on the indirect memory accesses that are usually described as $A[B[i]]$ on the code. The address of $A[B[i]]$ is expressed by the following equation.

$$ADDR(A[B[i]]) = Coeff \times B[i] + BaseAddr, \tag{3.1}$$

where, $Coeff$ is the size of each element in A, and $BaseAddr$ is the address of $A[0]$.

To prefetch the data at $A[B[i]]$, IMP follows these three steps: first, the stride prefetcher quickly detects the access pattern of the consecutive addresses in $B[i]$, and if it can determine the values of $Coeff$ and $BaseAddr$, it can calculate and prefetch the address of $A[B[i]]$ after prefetching $B[i]$. Second, the indirect pattern detector pairs the addresses of the stride access with its consecutive miss accesses, and uses these pairs to find a reasonable combination of $Coeff$

and $BaseAddr$. Using these values, the address generator calculates the address of $A[B[i]]$ based on $Coeff$, $BaseAddr$, and $B[i]$. Finally, IMP prefetches the data at this address. This process can also apply to prefetch the addresses and data at $B[i + \Delta]$ and $A[B[i + \Delta]]$.

When calculating the address of $A[B[i]]$, the address generator uses the simplified equation of Eq. (3.1) to reduce hardware costs, shown as the following equation.

$$ADDR(A[B[i]]) = (B[i] \ll shift) + BaseAddr. \tag{3.2}$$

This is because the size of each element in A can be expected to be small powers of two in real applications. Therefore, only one shifter and one adder are required instead of a multiplier.

3.2.2 Vector gather instruction

This chapter focuses on vector gather instructions in vector instruction sets to enable the efficient processing of indirect memory accesses. These instructions allow the processor to access multiple memory locations specified by a list vector, enabling flexible and efficient access to irregular memory patterns [57, 2, 58]. In contrast, standard vector load instructions are limited to sequential or stride access patterns. Algorithm 2 demonstrates how Algorithm 1 can be implemented using vector instructions, including vector load (VLD), vector multiply (VMUL), vector add (VADD), and vector gather (VGATHER). These instructions are used to calculate addresses, gather data from those addresses, and then multiply the resulting vector by a scalar value. The ability of vector gather instructions to vectorize indirect memory accesses allows vector processing units to operate on

Algorithm 2 The sequence of indirect memory accesses by vector gather instructions.

VLD	$\$v0, address, stride$	# Load $L[i]$
VMUL	$\$v1, \$s0, \$v0$	# Multiply the element size
VADD	$\$v2, \$s1, \$v1$	# Add the base address
VGATHER	$\$v3, \$v2$	# Gather $A[L[i]]$
VMUL	$\$v4, \$s2, \$v3$	# $s * A[L[i]]$

the data simultaneously.

However, vector gather instructions in vector processing may result in decreased performance due to two reasons. First, a vector gather instruction requires a list vector to generate addresses because the instruction depends on the vector load instruction that loads the list vector. If this load instruction is delayed, the vector gather instruction will also be delayed. Second, the memory accesses initiated by the vector gather instruction are likely to take a long time due to their irregular access patterns, which do not benefit from the high bandwidth of memory systems for sequential and regular access patterns. Furthermore, it is difficult to predict which data should be cached at run-time, as the values of the list vector are not known until that time. As a result, vector gather instructions may significantly decrease performance and become a bottleneck, particularly in memory-intensive applications.

If the load latency of the list vector is reduced, the vector gather instruction can be issued early. Moreover, if the irregular memory accesses issued by the vector gather instruction can be cached, the memory access latency of the vector gather can be reduced. Therefore, this dissertation proposes a prefetching mechanism for the vector gather instruction.

3.3 Vector gather prefetcher

This section discusses a mechanism for prefetching data in modern vector processors before the processor requires the data. Vector gather instructions necessitate list vectors prior to their execution, and these memory accesses are often irregular and exhibit a low locality. As a result, it can be difficult to cache this type of data. Nevertheless, by caching the list vectors beforehand, it is possible to predict the locations of indirect memory accesses as shown in Eq. (3.2). Therefore, this mechanism aims to prefetch list vectors in advance.

Figure 3.1 illustrates the overview of the vector gather prefetcher, which mainly consists of a stream list vector prefetcher and an indirect vector prefetcher. In order to prefetch the data loaded by a vector gather instruction, the vector gather prefetcher follows two steps. First, the stream list vector prefetcher monitors the addresses of vector load instructions issued for list vectors. Using the memory accesses of a list vector, the stream prefetcher forecasts the addresses of the subsequent list vectors. If the data of the subsequent list vectors are not present in the cache, the stream list vector prefetcher retrieves them. If these data have already been cached or prefetched, the indirect vector prefetcher computes the addresses of the indirect memory accesses using the value in the cache and initiates the prefetching.

This mechanism operates under the assumption that the vector instruction set includes a `VLDIDX` instruction for loading a list vector. When the processor issues `VLDIDX`, the processor determines the element size, base address, and that the loaded data will be utilized in a future vector gather instruction. Thus, the processor informs the prefetcher whether the memory requests can be processed successfully to load list vectors or not.

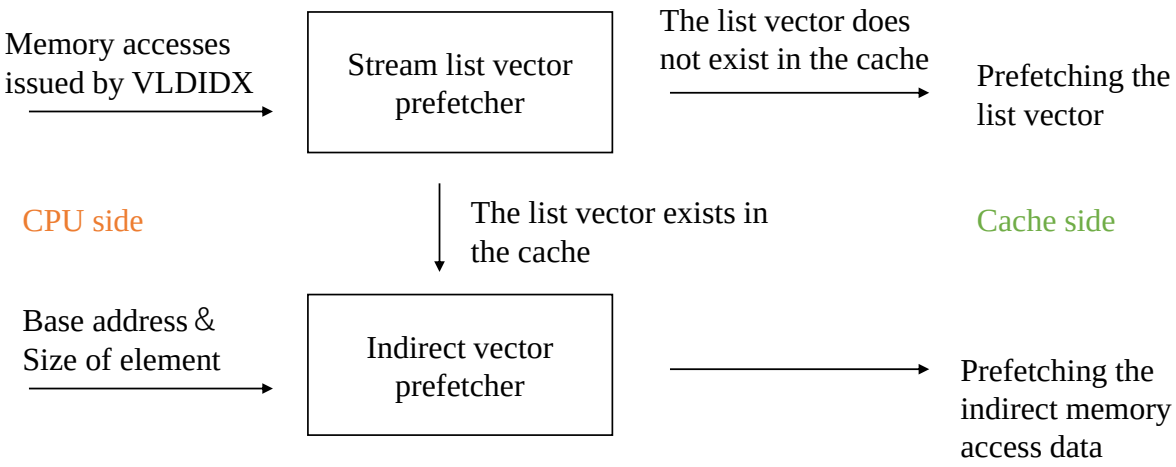


Figure 3.1: Vector gather prefetcher.

3.3.1 Stream list vector prefetcher

The stream list vector prefetcher tracks the memory accesses of `VLDIDX`. Upon arrival of the access requests of `VLDIDX`, the stream list vector prefetcher retrieves the subsequent list vectors. If the cache already holds the target list vectors, the data are forwarded to the indirect vector prefetcher. Additionally, when the target list vectors reach the cache, the data are also provided to the indirect vector prefetcher.

This prefetcher utilizes two parameters: the prefetch degree and the prefetch distance. The prefetch distance determines the timing of prefetching list vectors relative to the current accessed vectors. If the prefetch timing is too early or too late, it may give a harmful effect. Hence, an optimal value for the prefetch distance parameter is necessary. The prefetch degree determines the quantity of data prefetched at once. If there is sufficient space in the cache for prefetching, prefetching additional list vectors may enhance performance. However, as the actual cache capacity is limited, prefetching too much data may evict necessary data.

Since vector instructions are processed as a vector, prefetching follows a

similar approach. Let v represent the vector length, D represent the prefetch distance, and ϕ represent the prefetch degree. When the current accessed addresses are between $B[i]$ and $B[i + v - 1]$, the prefetcher prefetches addresses between $B[i + (v - 1)D]$ and $B[i + (v - 1)(D + \phi)]$.

3.3.2 Indirect vector prefetcher

The indirect vector prefetcher generates the addresses to be issued by vector gather instructions and prefetches the data accordingly. These addresses can be computed using Eq. (3.2) and the cached values provided by `VLDIDX` or the stream list vector prefetcher. However, it can be challenging for the prefetcher to determine at runtime which accesses are being used as indices for indirect accesses. To simplify the analysis of prefetching's effects, this study makes the assumption that the processor can directly supply the prefetcher with the element size and base address. Since the processor knows which instruction loads data for the vector gather instruction due to `VLDIDX`, it can also send this information to the prefetcher when issuing `VLDIDX`.

3.4 Evaluations

In order to examine the effect of the vector gather prefetcher, this dissertation conducts experiments by a simulator.

3.4.1 Configurations of the vector processor

The vector processor has a similar configuration to NEC SX-ACE [58], with an ability to process 256 elements per instruction. It consists of a vector core, cache, and main memory. The cache utilizes data reuse to reduce the memory access latency, and the proposed prefetching mechanism aims to further decrease the latency by prefetching data into the cache. It is integrated into the cache and tracks accesses of list vectors and indirect memory accesses, as described in Section 3.3. The other processor parameters are listed in Table 3.1.

3.4.2 Benchmark kernels

This study uses a scale kernel that multiplies scalar value s with array A. Algorithm 3 presents a pseudo-code for a normal scale kernel, and Algorithm 4 presents the scale kernel with indirect memory accesses. To vary the operational intensity of each code, the size of array P is modified. P is set to 1, 2, 4, 8, 16, 32, 64, and 128. As the performance of Algorithm 4 is heavily influenced by the contents of the index array $L[i]$, this study evaluates two types of index arrays: sequential and random.

With a sequential index array, the memory access latencies of vector gather instructions are similar to those of vector load instructions. In the case of a random index array, the memory access latencies of vector gather instructions become longer than those of vector load instructions due to the irregular access

Table 3.1: Parameters of the evaluation.

Number of cores	1
Performance of floating operation	32 GFlops
Instruction decode width	4
Processor clock cycle	1 Ghz
Vector length	256
Cache size	1 MB
Cache associativity	4-way
Cache line size	128 Bytes
Cache bandwidth	256 GB/s
Memory bandwidth	256 GB/s
Prefetch distance	1 (Sequential), 1 (Random)
Prefetch degree	32 (Sequential), 1 (Random)

Algorithm 3 VLDscale

1: **for** $i = 0, 1, ...N$ **do**
2: $\quad B[i] = s \times A[i]^P$
3: **end for**

patterns induced by the vector gather instructions.

3.4.3 Simulator

This study uses the simulator of vector processors developed based on the gem5 simulator [59], which is a general-purpose architecture simulator. The developed simulator uses an instruction trace data as an input, which is obtained by vector supercomputer SX-ACE. It simulates the occupancy of hardware resources inside the processor and calculates the various performance metrics. In this evaluation, this study uses a 16 channels DDR3 memory model in order to simulate the variation of the latencies depending on the irregular memory access.

As the simulator is trace-driven, it only manages memory access addresses,

Algorithm 4 VGTscale

1: **for** $i = 0, 1, ...N$ **do**
2: $\quad B[i] = s \times A[L[i]]^P$
3: **end for**

not the values stored at these addresses. Therefore, it cannot calculate the addresses of indirect memory accesses in the same manner as IMP, which checks the values of the index array in the cache and calculates the addresses loaded by vector gather instructions. To model the proposed prefetcher in the simulator, the relations between vector load instructions and vector gather instructions are profiled and recorded in the trace data before the simulation. The simulator then simulates the prefetcher's behavior by utilizing this recorded trace data to identify the relations, calculate the addresses, and prefetch the corresponding data. In other words, the prefetcher can effectively track accesses of index arrays and vector gather instructions in an ideal setting.

3.4.4 Performance evaluation

Figure 3.2 illustrates the performance results of the scale kernel simulation with a sequential index array. The performance is depicted on the vertical axis, while the operational intensity is depicted on the horizontal axis. The dotted line represents the upper bound of the roof-line model [32], indicating the achievable performance of the processor. The scale kernel referred to as *VLD scale* is described in Algorithm 3, while the *VGT scale sequential* kernel refers to the scale kernel outlined in Algorithm 4 without prefetching. The *VGT scale sequential with prefetching* indicates the scale kernel utilizing the proposed prefetching technique.

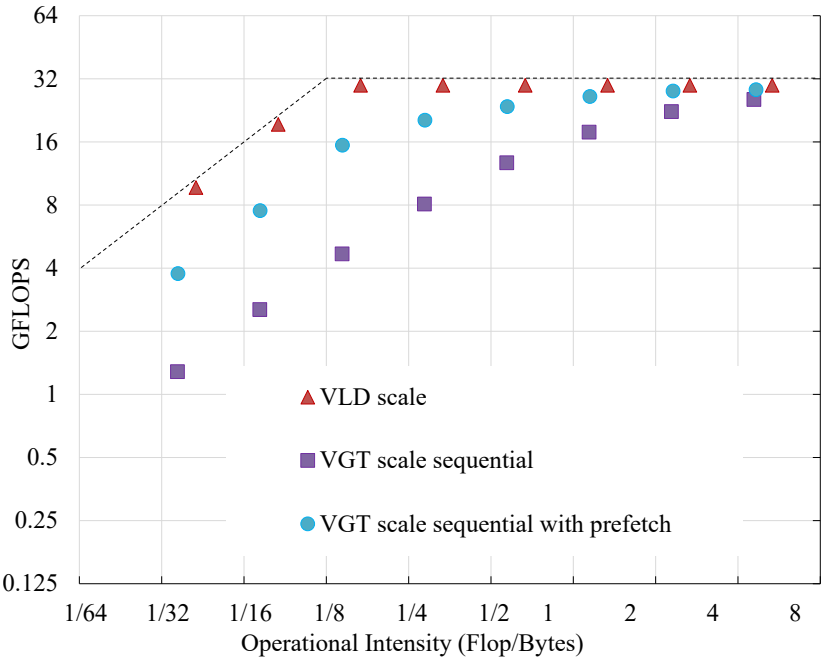


Figure 3.2: Roof-line model of a scale kernel in the case of the sequential index array.

In Figure 3.2, the *VLD scale* approaches the upper bound of performance. The *VGT scale sequential* exhibits lower performance due to the incorporation of vector gather instructions. However, the use of prefetching leads to an improvement in performance, bringing it closer to the upper bound. On average, the performance is improved by a factor of 2.1 when prefetching is utilized. The maximum performance improvement of 3.29 is achieved at an operational intensity of 0.14 corresponding to $P = 4$ in Algorithm 4. At higher operational intensities, especially $P = 4$ or larger, the improvement in performance is 1.11. This is due to the fact that high operational intensities already contribute to sufficient performance, and the presence of additional arithmetic instructions helps to hide the latency of vector gather instructions.

In the scenario where the maximum performance improvement is achieved, the cache hit rate for read accesses increases dramatically from 7.96% to 96.72%.

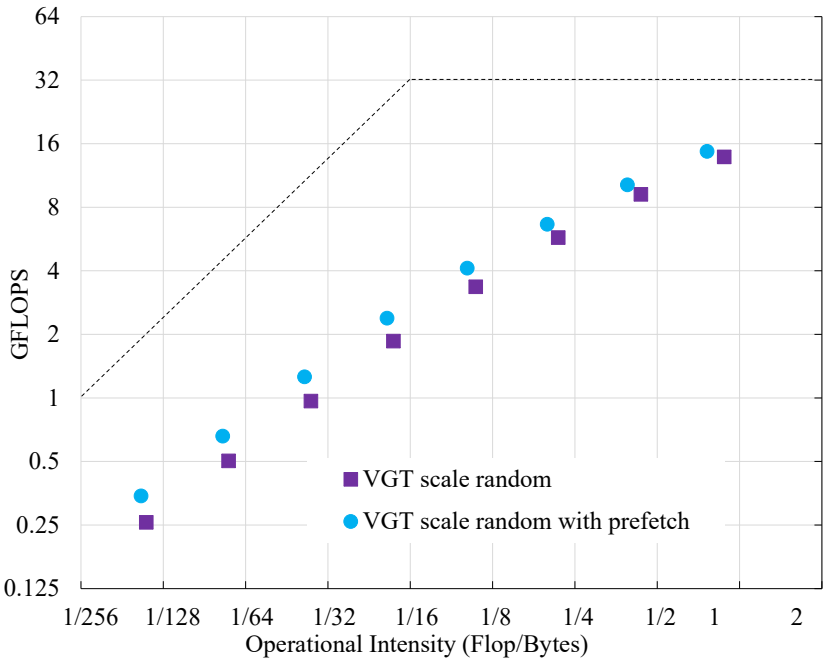


Figure 3.3: Roof-line model of a scale kernel in the case of the random index array.

The use of prefetching greatly enhances the cache hit rate. The sustained bandwidth between the processor and cache also increases significantly, from 10.15GB/s to 32.03GB/s. The sustained bandwidth between the cache and main memory similarly increases, from 13.67GB/s to 44.92GB/s. It is clear that prefetching can effectively utilize these increased bandwidths.

There is potential for further performance improvement when comparing the *VLD scale* with the *VGT scale sequential with prefetching*. This is likely due to the fact that the processor lacks the ability to generate addresses prior to issuing vector gather instructions.

Figure 3.3 presents the results of the scale kernel simulations using a random index array. The *VGT scale random* refers to the scale kernel outlined in Algorithm 4 without prefetching, while the *VGT scale random with prefetching* refers to the scale kernel when using the proposed prefetching mechanism.

On average, the performance is improved by a factor of 1.2 when prefetching

is utilized. The maximum performance improvement of 1.33 is achieved at an operational intensity of 0.0064 ($P = 1$ in Algorithm 4). When using a random index array, the *VGT scale random* is relatively closer to the *VGT scale random with prefetching*, compared to the case of the sequential index array. This is due to the fact that a single vector gather instruction in the random index array loads numerous cache blocks, leading to pressure on cache capacity and memory bandwidth. Thus, even when the prefetcher accurately attempts to prefetch the necessary blocks, the cache may not have sufficient capacity to store all of them. As a result, the performance improvement is limited.

In the scenario where maximum performance is achieved, the cache hit rate for read accesses increases significantly from 0.43% to 46.34%. The use of prefetching is successful in enhancing the cache hit rate. The sustained bandwidth between the processor and cache also increases, from 36.28GB/s to 48.35GB/s, and the sustained bandwidth between the cache and main memory similarly increases from 76.24GB/s to 106.63GB/s. The prefetching can contribute to increase sustained bandwidth.

3.4.5 Parameter study

On the vector gather prefetcher, *prefetch distance* and *prefetch degree* significantly affect the performance. Thus, Section 3.4.5 discusses the effects of these parameters.

The optimal combination of these parameters is dependent on a variety of factors. This study primarily focuses on the number of cache blocks accessed by a single vector gather instruction as a key factor. This number is closely related to the properties of the index array, such as elements having similar values. If the index array is likely to have random values, the vector gather instruction

will access multiple addresses across numerous cache blocks. However, if the elements in the index array are more likely to have similar values, the vector gather instruction will access fewer cache blocks compared to the case of a random index array.

Figure 3.4 illustrates the results of varying the *prefetch distance* and *prefetch degree* parameters in the scale kernel using a sequential index array. The x-axis and y-axis represent the *prefetch distance* and *prefetch degree*, respectively, while the z-axis represents the performance. The highest performance is achieved when the *prefetch distance* is 1 and the *prefetch degree* is 32. It can be seen in Figure 3.4 that, for appropriate values of *prefetch distance*, increasing the *prefetch degree* leads to improved performance. However, when both the *prefetch distance* and *prefetch degree* are excessively large, there is a significant drop in performance, indicated by the steep cliff in the graph. This is due to the fact that when both parameters are too large, the necessary prefetched cache blocks are prematurely evicted due to limitations on cache capacity.

Figure 3.5 displays the results of varying the *prefetch distance* and *prefetch degree* parameters with a random index array. The highest performance is achieved when both parameters are equal to 1. In the case of a random index array, a single vector gather instruction accesses a large number of cache blocks, with a maximum of 256 cache blocks accessed at once. In contrast, with a sequential index array, a vector gather instruction accesses only 16 cache blocks. The increased number of cache blocks accessed per vector gather instruction leads to the eviction of previously prefetched data by subsequent blocks being prefetched. As a result, smaller values of the parameters yield better performance.

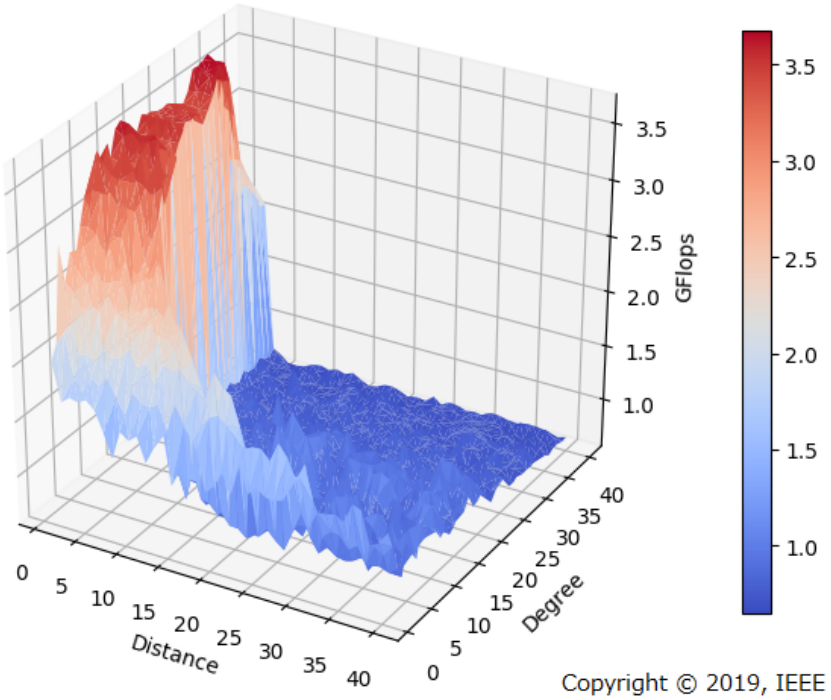


Figure 3.4: The effects of parameters, *prefetch distance* & *prefetch degree* in the case of the scale kernel with the sequential index array.

3.4.6 Effect of the number of cache blocks

Properly set parameters can lead to a slight performance increase even when using a random index array. Thus, if the prefetcher can determine the typical number of cache blocks accessed by vector gather instructions, appropriate prefetching can be implemented by adjusting the parameters. This study investigates the characteristics of the number of cache blocks and their relation to these parameters.

This study includes additional types of index arrays with the aim of causing a single vector gather instruction to access a constant number of cache blocks, depending on the array used. Table 3.2 illustrates the relationship between

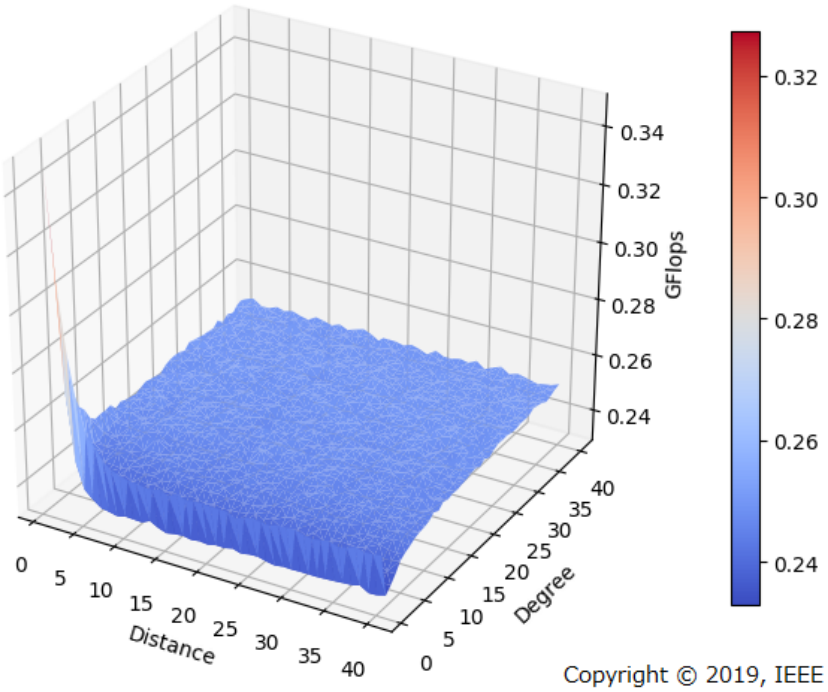


Figure 3.5: The effects of parameters, *prefetch distance* & *prefetch degree* in the case of the scale kernel with the random index array.

stride or duplicate values and the number of cache blocks accessed. In Table 3.2, "Stride" refers to an index array composed of regularly spaced values, such as {0,2,4,6,8,...}. With this type of array, vector gather instructions access more cache blocks compared to those using a sequential index array. "Duplication" refers to an index array comprising regularly repeating values, such as {0,0,1,1,2,2,...}. This type of array leads to fewer cache blocks being accessed by vector gather instructions compared to a sequential index array.

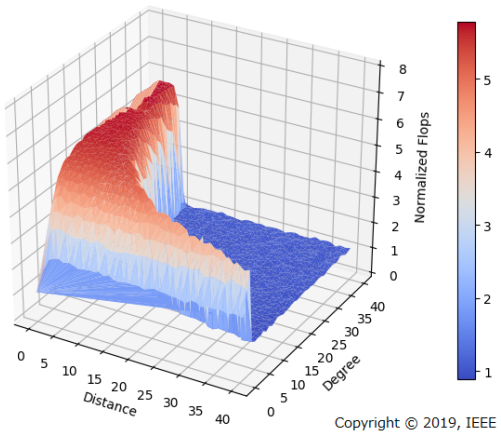
Figures 3.6 and 3.7 show the results of strided/duplicated index arrays. The x- and y-axes represent the *prefetch distance* and *prefetch degree*, respectively, while the z-axis displays the normalized sustained performance in flop/s relative to the case without the proposed prefetching. Note that while the scales of the axes in the figures are consistent, the scales of the color bars in each figure may

Table 3.2: The relation between stride or duplicate width and the number of cache blocks per vector gather instruction.

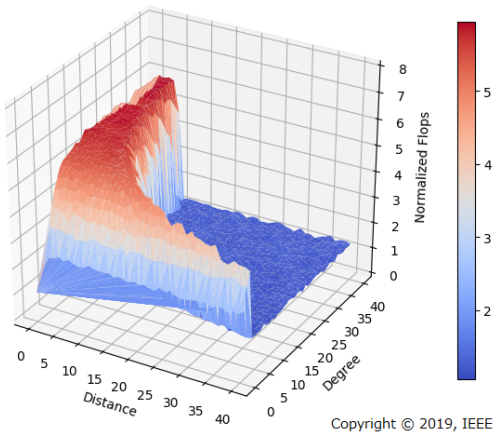
Stride/Duplication	Width	Cache blocks
Duplication	16	1
Duplication	8	2
Duplication	4	4
Duplication	2	8
Sequential	N/A	16
Stride	2	32
Stride	4	64
Stride	8	128
Stride	16	256

vary. These figures demonstrate that the optimal pair of parameters depends on the index array. When the number of cache blocks accessed by a single vector gather instruction is small, a small prefetch distance and large prefetch degree yield the highest performance. Conversely, when the number of cache blocks per vector gather instruction exceeds that of the sequential index array, the best performance is achieved with both parameters set to small values. This is due to the relationship between the number of cache blocks per vector gather instruction and spatial locality; a low spatial locality can hinder the performance of prefetching. Therefore, for the cases depicted in Figures 3.7a–3.7d where the number of cache blocks per vector gather instruction exceeds 32, it is advisable to use small values for both the *prefetch distance* and *prefetch degree*.

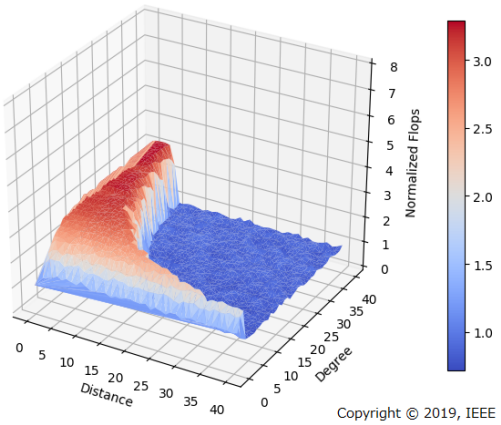
When the number of cache blocks is small, as shown in Figures 3.6a–3.6d, these parameter pairs can achieve higher performance. However, as the number of cache blocks increases, only a limited number of pairs can lead to improved performance. Therefore, adjusting these parameters according to the number of cache blocks accessed can enable more effective prefetching. It is also worth noting that the fewer number of cache blocks accessed per vector gather instruction



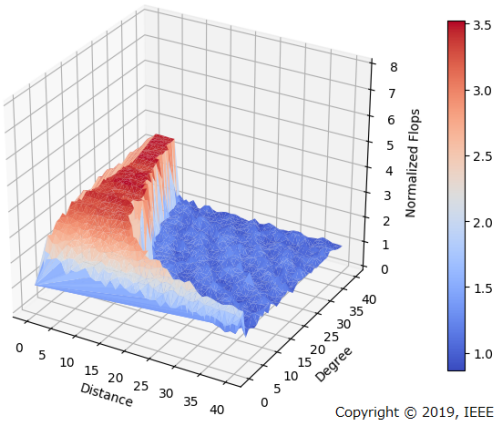
(a) Cache blocks per vector gather instruction = 1.



(b) Cache blocks per vector gather instruction = 2.



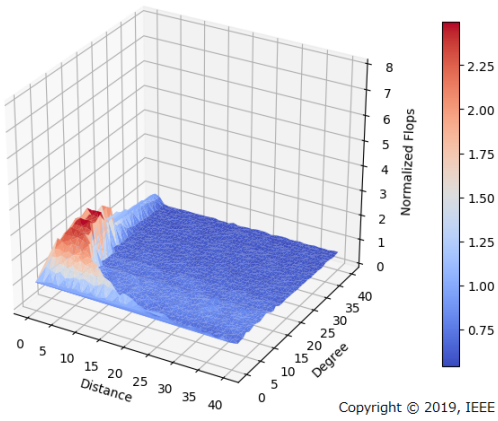
(c) Cache blocks per vector gather instruction = 4.



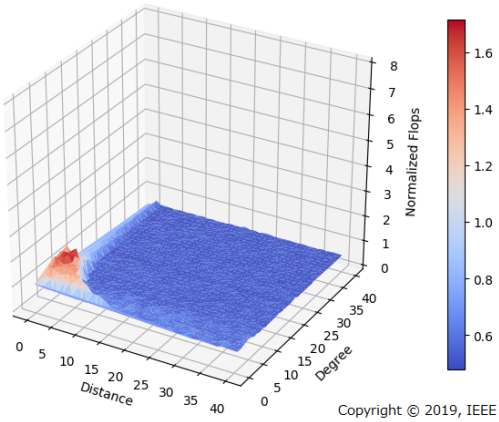
(d) Cache blocks per vector gather instruction = 8.

Figure 3.6: The performance improvement when the number of cache blocks per vector gather instruction is decreased from the sequential case.

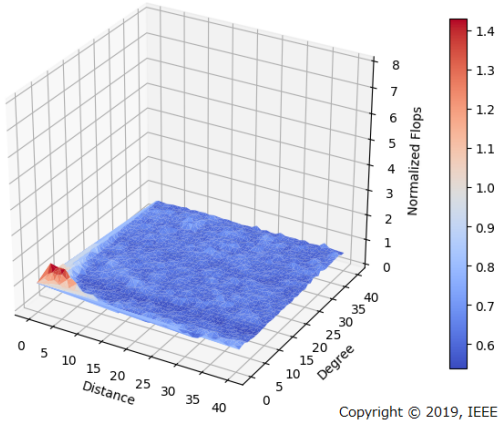
leads to the greatest performance improvements through prefetching, suggesting that spatial locality plays a crucial role in prefetching.



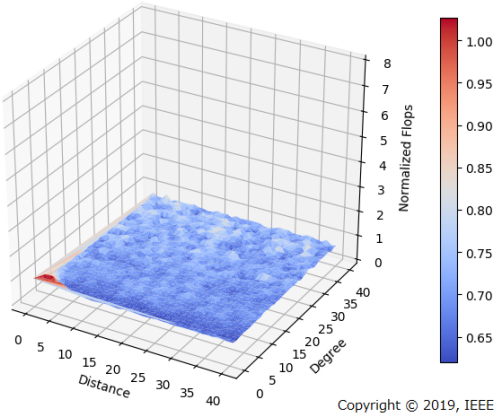
(a) Cache blocks per vector gather instruction = 32.



(b) Cache blocks per vector gather instruction = 64.



(c) Cache blocks per vector gather instruction = 128.



(d) Cache blocks per vector gather instruction = 256.

Figure 3.7: The performance improvement when the number of cache blocks per vector gather instruction is increased beyond the sequential case.

3.5 Conclusions

This chapter has presented a hardware prefetching mechanism for indirect memory accesses of vector gather instructions. The proposed prefetching mechanism aims to prefetch the index data in advance and subsequently prefetch the data of indirect memory access using the index values. The prefetching mechanism has two parameters: *prefetch distance* and *prefetch degree*, which significantly impact the performance of the prefetching.

The performance of the scale kernel using vector gather instructions can be improved by approximately 2.1 times when using a sequential index array and 1.2 times when using a random index array with the proposed prefetching mechanism. Additionally, this chapter demonstrated that the choice of *prefetch distance* and *prefetch degree*, two key parameters that impact the performance of prefetching, is crucial to achieving optimal performance, which can be sensitively affected by the values in the index array.

Additionally, this chapter have discussed the impact of the number of cache blocks per vector gather instruction on performance. The findings demonstrate that a lower number of cache blocks per vector gather instruction leads to improved performance. However, a large number of cache blocks per vector gather instruction may result in detrimental prefetching and decrease performance.

By utilizing prefetching for indirect memory accesses, the proposed mechanism can conceal the latency incurred by waiting for instructions with dependencies and thus improve performance.

Chapter 4

Criticality-aware out-of-order mechanism for vector instructions

4.1 Introduction

In order to take advantage of instruction-level parallelism (ILP), the modern vector processors utilize an out-of-order execution model [16]. These processors execute independent vector instructions by rearranging the execution order of them at runtime. While this technique has proven successful in improving performance, it raises a question for architects: how much ILP is optimal for vector processors? This ultimately depends on the individual application. The modern vector processors are designed for vectorized applications with data-level parallelism, and ideally should be able to fully utilize ILP even between iterations.

However, it is not feasible for vector processors to examine the dependencies of all the vector instructions across iterations using the traditional out-of-order

execution model. This is because increasing the ILP of the vector processor would necessitate significant resources for each vector instruction with a long vector length. Vector processors are designed with a focus on throughput, and therefore only require the minimal amount of ILP, leading to a tendency to underestimate the significance of ILP. It is difficult to further increase ILP for vector processors that already consume hardware resources for data-level parallelism.

This chapter focuses on the use of runahead execution mechanisms to make vector processors more tolerant of latency. Runahead execution is a speculative execution technique that accurately prefetches long-latency loads in subsequent instructions [38, 39, 60]. When the long-latency loads run out the instruction window and cause pipeline stalling, the processor saves the current processor state and enters a runahead mode. In this mode, the processor speculatively executes subsequent instructions that would only result in additional long-latency accesses. Once the pipeline stall is resolved, the processor discards the registers from the runahead mode and returns to the normal mode. As the runahead execution leaves the data in the cache, processor in the normal mode can access the data with shorter latency through cache hits.

Furthermore, the latest advancement in runahead execution allows for the speculative execution of only those instructions that may decrease performance rather than all instructions [41]. This enhancement allows a processor to take advantage of loop-level parallelism by considering the importance of instructions. However, this is not the case for vector processors. As vector instructions work with vectorized data, the cost of reloading from the cache and executing them again is significant. If applications require a high bandwidth to accommodate a large volume of data, runahead execution may hinder performance.

This chapter proposes a criticality-aware out-of-order mechanism for vector processors (COV). The COV has normal and critical modes, similar to the state-of-the-art runahead execution techniques. The difference is that the data produced in the critical mode remain in the vector registers after exiting the critical mode. The COV uses these data in the normal mode by migrating the instruction information in the critical mode to the normal mode using additional queues. This mechanism enables the COV to reduce the cost of an additional penalty of the runahead execution techniques and to eliminate the need to re-execute the instructions that have already been executed in the critical mode.

4.2 Motivation

4.2.1 Runahead execution

Runahead execution [38, 39, 60] has been proposed for processors to exploit ILP beyond the instruction window size. This technique improves performance by speculatively executing future instructions while the processor is stalled. When the instruction window of the processor becomes full and is unable to issue new instructions, the processor enters a runahead mode. In this mode, the processor eagerly retrieves and executes future instructions. If the stalling is resolved and the processor is able to issue new instructions again, the processor discards the registers and restores the architectural state. As the instructions executed in runahead mode are fetched and executed again in normal mode shortly after exiting runahead mode, runahead execution can bring the necessary data into the cache through accurate prefetching.

In early proposals for runahead execution, the processor copies physical registers to a backup space upon entering a runahead mode. Upon exiting runahead mode, the processor must re-fetch and re-execute the discarded instructions in normal mode, thereby limiting the benefits of runahead execution.

Naithani *et al.* [41] have proposed the precise runahead execution (PRE) to solve the limitations of conventional runahead execution mechanisms. They have enhanced the conventional runahead mechanisms in three ways. First, they have ensured that there are sufficient free registers and issue queue resources before entering the runahead mode, eliminating the need to back up these resources upon entering the runahead mode.

Second, PRE only speculatively executes stalling load slices, which are instructions causing pipeline stalling and their dependencies, unlike the conventional runahead execution that executes all instructions. A stalling load slice consists of an instruction causing pipeline stalling and its dependent instructions. To identify stalling load slices, PRE tracks the dependencies of stalling instructions iteratively. PRE accurately speculatively executes only those instructions that would significantly improve performance, even in short intervals.

Finally, PRE uses a mechanism to reuse registers during the runahead mode. By allocating released registers to new instructions during the runahead mode, precise runahead execution can reduce the number of registers used in the runahead mode.

These improvements allow the processor to perform runahead execution without incurring additional penalties for entering runahead mode, expanding the range of performance improvement.

4.2.2 Problems to apply precise runahead execution for vector processors

The idea to adopt the runahead execution mechanisms to the vector processors is derived from two insights into applications commonly executed on the vector processors. First, the kernel part of each application has loop structures. Second, there are only a limited number of instructions inside the iteration that need to wait for long memory accesses. This chapter defines this type of instruction as *critical instructions*. Since a limited number of instructions requiring long memory accesses will appear in each iteration, early dispatching of the instructions across iterations can lead to performance improvements. Figure 4.1

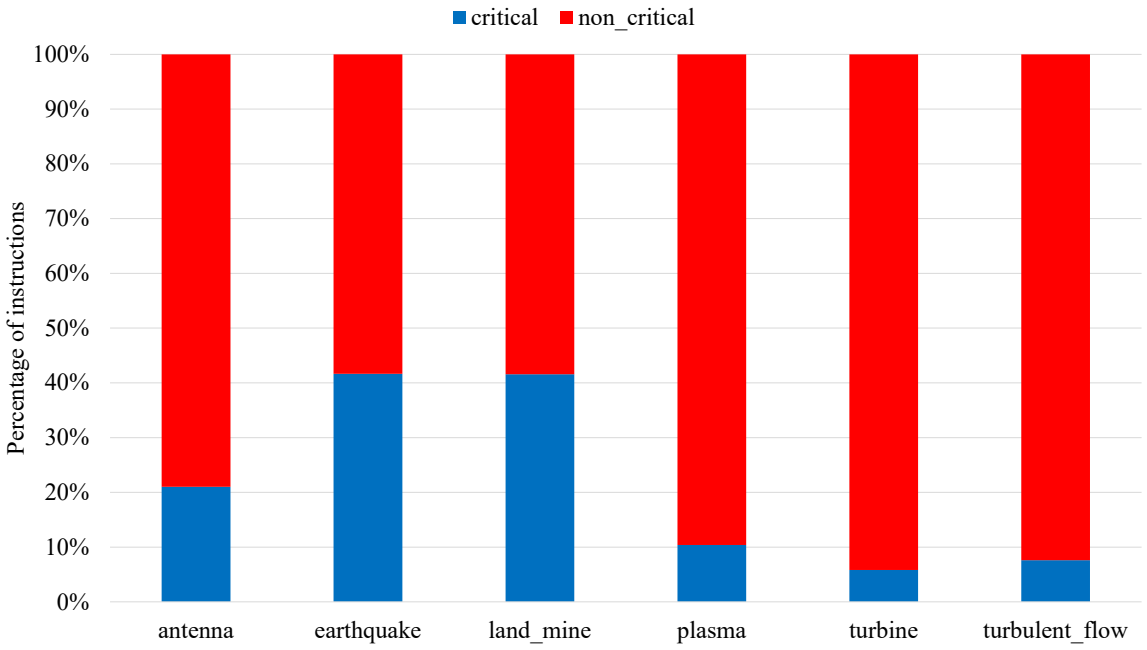


Figure 4.1: The percentage of the instructions causing pipeline stalling and their dependencies.

shows the ratio of the critical instructions and their dependencies in the kernel part of the applications. Three out of six applications contain less than 10% critical instructions, and two out of six applications contain up to 40% critical instructions. These results indicate that a limited number of instructions require long memory accesses and repeatedly appear due to the loop structure. These two insights inspire us to exploit more instruction-level parallelism among the loops by proactively executing future critical instructions. The precise runahead execution can fit this demand by awareness of the criticality.

On the other hand, to apply the runahead execution to vector processors, the runahead execution may incur another penalty. In the runahead execution, the processor has two modes: runahead mode and normal mode. The processor switches to the runahead mode if the pipeline stalling occurs and continues to speculatively execute subsequent instructions. When the runahead mode is finished, the processor flushes used registers to avoid affecting the normal mode.

Although data produced in the runahead mode is correct, the processor must re-execute the instructions and reproduce the results from data in the cache in the normal mode.

Since vector instructions deal with vectorized data, the vector register contains multiple data. Thus, unlike the scalar register on scalar processors, the costs of reloading from the cache and executing them again are not negligible. In applications that require a high bandwidth to accommodate a large amount of data, the runahead execution can hurt performance. The runahead execution can be regarded as the method to generate prefetch instructions. Unlike other prefetching techniques, the runahead execution transfers data to the registers, which will be flushed at the end of the runahead mode. Thus, the runahead execution mechanism consumes more bandwidth between the core and the cache than standard prefetching mechanisms.

Therefore, it should be desirable to use the obtained data in the runahead mode through registers instead of the cache. This dissertation considers keeping the data calculated in the runahead mode and using them in the normal mode without flushing the data.

4.3 Criticality-aware out-of-order vector processor

This dissertation proposes a criticality-aware out-of-order mechanism for vector processors (COV). The COV can dispatch critical instructions beyond the window of the conventional out-of-order mechanism by leaving the registers after exiting the runahead mode. The proposed mechanism replaces the runahead mode with *a critical mode* to realize this function. The critical mode can exit without flushing the registers to be used the registers in the normal mode. However, the vector processor faces two challenges in using the critical mode.

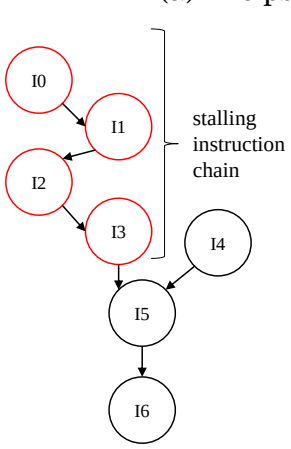
4.3.1 Challenges

If the register remains unchanged after the critical mode, there is a problem to be solved. An example of this scenario upon exiting the critical mode is depicted in Figure 4.2. A pseudo-code for vector instructions within a loop structure is shown in Figure 4.2a, and the dependency structure of the pseudo-code is depicted in Figure 4.2b. The processor can usually detect this dependency through the logical registers of the instructions. As illustrated in Figure 4.2a, it is assumed that instruction `I3` will cause pipeline stalling. Since the precise runahead execution mechanism can track stalling instructions and their dependencies, the instructions from `I0` to `I3` are marked as stalling instructions, depicted as red circles in Figure 4.2b. These instructions are referred to as the stalling instruction chain in this chapter. During the critical mode, the processor only dispatches instructions within the stalling instruction chain.

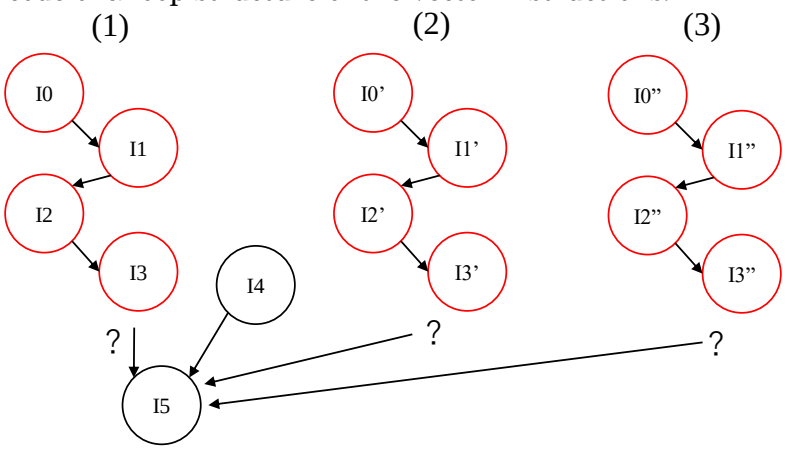
Upon encountering a pipeline stall at instruction `I3`, the processor enters the critical mode and begins issuing subsequent instructions. However, instructions

PC				
	for i+=VL			
I0		vld	$v1	Mem[i:VL]
I1		vmul	$v2	$v1 $s1
I2		vadd	$v3	$v2 $s2
I3		vgt	$v4	$v3
I4		vld	$v5	
I5		vmul	$v6	$v4 $v5
I6		vst	$v6	Mem'[i:VL]
	end			

(a) The pseudo code of a loop structure of the vector instructions.



(b) The logical dependency structure of the pseudo code.



(c) (1)First iteration in the normal mode, (2)Second iteration in the critical mode and (3)Third iteration in the critical mode.

Figure 4.2: The challenges to distinguish dependency where the normal mode and the critical mode.

`I4` through `I6` are not part of the stalling instruction chain and are therefore ignored by the processor. Instead, the processor moves on to the stalling instruction chain of the next iteration, as depicted in Figure 4.2c with iterations `I0'` through `I3'` and `I0"` through `I3"`. After exiting the critical mode, the processor returns to its normal mode and attempts to dispatch the instruction `I5`. However, the presence of three in-flight instructions, `I3`, `I3'`, and `I3"`, all of which

have a dependency on `I5`. This dependency leads to a situation where the processor is unable to determine which instruction has the true dependency on `I5` through the logical registers during the renaming process.

To identify the correct dependency, the processor has to keep two orders: the committing order and the register renaming order.

4.3.1.1 Consistency of the committing order

In an out-of-order processor, the reorder buffer (ROB) is responsible for managing in-flight instructions within an instruction window, allowing for their out-of-order execution while still ensuring that the final execution results match the original program order. As a result, both insertion into and deletion from the ROB must be performed in an in-order fashion.

However, the proposed mechanism attempts to dispatch instructions in an out-of-order fashion, which may result in their insertion into the ROB in an out-of-order manner. This can lead to a commit order that differs from the original program order. To solve this issue, the prior runahead execution mechanism [41] introduces the use of a dedicated queue called the register deallocation queue (RDQ) during the runahead mode, instead of the ROB, to avoid altering the commit order. The RDQ is responsible for managing in-flight instructions and pseudo-committing them during the runahead mode, after it is discarded upon returning to the normal mode, thereby preventing any unintended effects on the normal operation.

This study aims to realize the critical mode. However, the contents of the registers become obsolete once the RDQ is discarded at the end of the runahead mode. To retain the information of the instructions after exiting the critical mode, the RDQ must be preserved. Consequently, the processor must decide

between using the ROB or the RDQ in order to maintain the commit order of dispatched instructions between normal and critical modes.

4.3.1.2 Consistency of the register renaming order

An out-of-order processor employs a register aliasing table (RAT) to track the most recent renaming information and resolve dependencies among instructions. The mapping between logical and physical registers is updated when instructions are dispatched and committed. However, if the processor dispatches instructions in different modes, the physical registers renamed in the critical mode may contain data for instructions that are not immediately needed in the normal mode. If the RAT are updated immediately in the runahead mode, it would overwrite the aliasing information from the previous normal mode.

The prior runahead execution mechanism utilizes a new renaming mechanism to manage register renaming information during the runahead mode, discarding it upon returning to the normal mode. However, this study aims to implement the critical mode that does not flush registers upon exiting. To retain renaming information for instructions after exiting the critical mode, it cannot be discarded. Furthermore, the processor must be able to resolve dependencies when instructions with the same program counter are in-flight, as depicted in Figure 4.2c.

4.3.2 Mechanism overview

To overcome these challenges, this chapter proposes a criticality-aware out-of-order mechanism for vector processors (COV). The proposed mechanisms migrate the order information and the renamed register information of the instructions from the critical mode to the normal mode with the correct program order.

The key idea is that additional queues hold the information of the instructions in the critical mode. After returning to the normal mode, the processor dispatches the instructions referring to these queues. If instructions are executed in the critical mode, the information of these instructions is migrated from the queues used in the critical mode to queues used in the normal mode.

Figure 4.3 shows the overview of the proposed mechanism for vector processors. The processor dispatches vector instructions that may cause pipeline stalling, namely *critical instructions*, in a different path by switching modes, similar to the state-of-the-art runahead execution mechanism [41]. The vector processor has two modes: a normal mode and a critical mode.

In the normal mode, the vector processor executes instructions as usual. Additionally, our proposed mechanism has modified the rename stage in the normal mode. This dissertation calls it a *rename/contextualize* stage. When the vector processors in the normal mode find that an instruction is already dispatched in the critical mode, the vector processor updates the register aliasing table and transfers the information of the instruction from the PCQ to the ROB. This dissertation calls this process *contextualizing an instruction*.

In the critical mode, the processor dispatches the *critical instructions* beyond the instruction window. This dissertation calls the dispatched instructions in the critical mode *an early-dispatched instruction*.

To realize the critical mode and the contextualization, the proposed mechanism adds four components to an out-of-order vector processor: decoded instruction queue (DIQ), stalling instruction cache (SIC), pending commit queue (PCQ), and critical register aliasing table (CRAT), as shown in the green-shaded boxes.

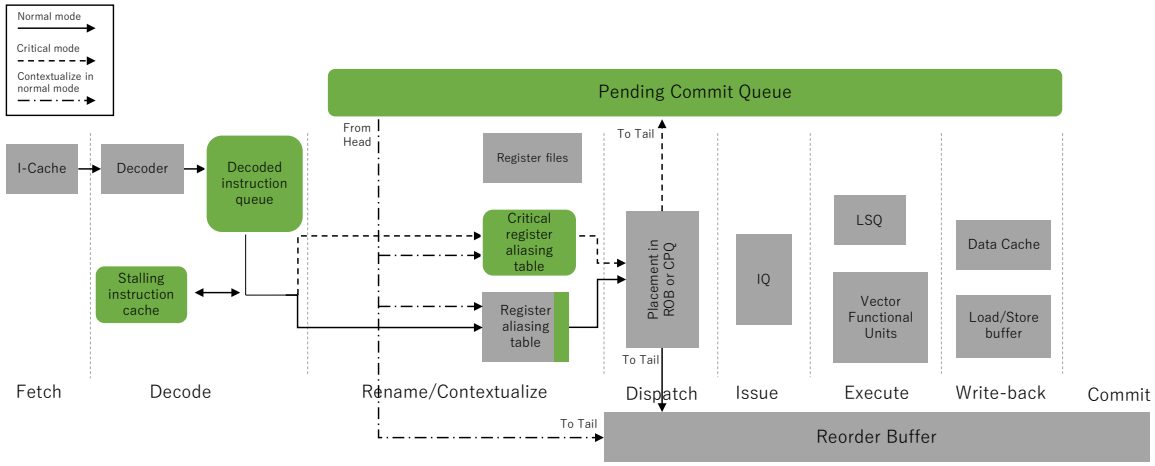


Figure 4.3: The overview of the criticality-aware out-of-order vector processor.

4.3.2.1 Decoded instruction queue (DIQ)

The DIQ is a first-in-first-out (FIFO) queue that temporarily stores decoded instructions. It is implemented as a circular buffer-like structure with two additional features. First, each entry has a 1-bit flag called a *dispatched flag*. This flag is set when the vector processor renames the operands of an instruction in the critical mode. The vector processor in the normal mode can determine whether an instruction is already in-flight in the critical mode by checking this flag.

In addition to the head and tail pointers that a circular buffer-like structure typically includes, the DIQ also has a *critical pointer* that points to a specific entry. This critical pointer saves the position at which an instruction is checked in the critical mode. Initially, the critical pointer points to the same entry as the head pointer. In the critical mode, the vector processor attempts to rename the instruction at the entry indicated by the critical pointer. If it is able to dispatch the instruction in the critical mode, the dispatched flag is set and the critical pointer is incremented. The critical pointer is also incremented if the instruction cannot be dispatched in the critical mode. In this way, the vector processor can

search for future instructions by incrementing the critical pointer in the critical mode, and find critical instructions within the DIQ. When the vector processor dispatches instructions from the head pointer in the normal mode, the critical pointer is also incremented if it points to the same entry as the head pointer.

The DIQ allows the vector processor to ensure that all instructions in the queue are checked only once for dispatch in the critical mode, thus preventing the presence of duplicate in-flight vector instructions on the processor.

4.3.2.2 Stalling instruction cache (SIC)

The SIC, which is a fully associative cache, stores the program counters (PCs) of instructions and instructions' dependencies that prevent the reorder buffer (ROB) from committing. When the ROB is delayed in committing and waits for a number of cycles beyond a certain threshold in the normal mode, the SIC stores the PC of the oldest instruction in the ROB.

4.3.2.3 Pending commit queue (PCQ)

The PCQ is a FIFO queue that temporally stores the information of the in-flight instructions dispatched in the critical mode. The entry of the PCQ has the same fields as the ROB entry.

Upon issuing an instruction in the critical mode, the vector processor adds it to the PCQ rather than the ROB. Unlike the ROB, the PCQ does not commit any instructions. Its purpose is merely to maintain the order of dispatched instructions in the critical mode. In the normal mode, the rename/contextualize stage transfers the instructions in the PCQ to the ROB, which subsequently commits them.

4.3.2.4 Critical register aliasing table (CRAT)

To obtain the operands during the critical mode, the vector processor must search PCQ's logical destination registers for the instruction's source register. However, locating the logical destination register within the PCQ necessitates a time-consuming prioritized match search. In contrast, the RAT in the ROB is designed to prevent the need for traversing the queue [61].

In this study, the CRAT is proposed as an additional table for renaming in the critical mode. Each entry in the CRAT is comprised of both a logical destination register and a physical destination register, and the table can be accessed by using logical destination registers as indices. When an entry is added to the PCQ, the vector processor will search the CRAT for the logical destination operand. If it is found, the CRAT will assign a new physical register to the logical register, and update the entry in the CRAT to reflect the assignment of this physical register to the logical destination operand. If the logical destination operand is not found in the CRAT, a new entry will be created and a new physical register will be assigned to the logical destination register. However, if there is not enough space to create a new entry, the vector processor will be unable to execute any instructions in the critical mode and will have to exit the critical mode. The CRAT and the RAT both utilize a shared pool of free physical registers.

4.3.3 Behavior of criticality-aware vector processing

The section shows the execution flow of instructions in the critical mode. The flow follows the dashed arrow in Figure 4.3.

4.3.3.1 Recognition of the stalling instruction chain in the normal mode

When the ROB reaches capacity and the number of cycles exceeds a predetermined threshold in the normal mode, the instruction at the head of the ROB is added to the SIC. Additionally, if an instruction is dispatched in the normal mode and encounters the SIC, the RAT will follow the producer PC of that instruction and add it to the SIC as well. This process of continuously tracing the chain of stalling instructions allows the gradual inclusion of the stalling instructions and their dependencies, if they will be decoded again in the next iteration of a loop. It is assumed that the target applications being considered in this chapter are constructed using loop structures, making this mechanism effective in capturing all stalling instruction chains.

4.3.3.2 Enter the critical mode

Unlike prior runahead mechanisms, such as those described in [38, 41, 39], the proposed mechanism for vector processors does not involve checkpointing the PCs of instructions in the ROB or the state of the RAT. When the ROB halts committing in the normal mode, the vector processor transitions to the critical mode. However, vector instructions in the ROB can still be executed as they would in the normal mode.

4.3.3.3 Exit the critical mode

The only distinction between the critical mode and the normal mode is the source from which the front-end dispatches instructions, either the head pointer or the critical pointer of the DIQ. As a result, the front-end enters the critical mode whenever the ROB is full in the normal mode, and returns to the normal mode when the ROB resumes committing or when there are insufficient

resources for criticality-aware execution in the critical mode.

4.3.3.4 Pipeline behaviors in the critical mode

Fetch and Decode The fetch stage and the decode stage in the critical mode do their operations as they do in the normal mode. After the instructions are decoded, they are pushed to the DIQ.

Rename/Contextualize The rename/contextualize stage attempts to rename the logical register of the instruction indicated by the critical pointer of the DIQ. The vector processor checks whether the instruction is present in the SIC. If it is, the instruction can be dispatched in the critical mode. To do so, the vector processor must ensure that there is sufficient capacity to dispatch the instruction in the critical mode by verifying two conditions. First, the vector processor must confirm that the logical register of the instruction's source operands is present in the CRAT, as the information for instructions dispatched in the critical mode should already be stored in the CRAT. Additionally, the stalling instruction chain must not be dependent on the destination operands of other instructions dispatched in the normal mode to avoid false dependencies. If the CRAT does not contain the required information, the source operands' instruction is not in-flight, or the stalling instruction chain is being constructed in the SIC, the vector processor will ignore the instruction and increment the critical pointer. The second condition that the vector processor must verify is whether the PCQ or the issue queue is full or not. If either of these queues is full, the processor runs out the resources to dispatch the instructions in the critical mode.

It is worth noting that the critical pointer is incremented to evaluate the next instruction in the queue, regardless of whether the instruction is present in the SIC, until the critical pointer reaches the same position as the tail pointer. The

instructions checked in this manner remain in the decoded instruction queue. However, the vector processor will exit the critical mode if the PCQ, CRAT, or other resources become full.

Dispatch In the dispatch stage, the vector processor adds the instructions into the PCQ instead of the ROB.

Issue, execute and write-back The instructions dispatched in the critical mode are issued, executed, and written back as the instructions dispatched in the normal mode. Until contextualized, the instructions are executed in the critical mode and stay in the PCQ.

Commit After transferring from the PCQ to the ROB, the instructions are committed in the ROB as usual. If the exception occurs or the early-dispatched path is wrong (also be treated as the exception), the ROB should handle the exception or execute the correct path.

4.3.3.5 Contextualization of early-dispatched instruction in the normal mode

Section 4.3.3.5 explains the execution path of the case of contextualizing the early-dispatched instruction in the normal mode, shown in the dot-dashed arrow in Figure 4.3.

The vector processor performs two functions in the contextualization process. First, it conveys the instruction order information from the PCQ to the ROB to maintain the committing order. Second, it ensures that the physical registers calculated in the critical mode are accessible in the normal mode by updating the RAT to reflect the proper mapping of logical registers to physical registers.

This maintains the consistency of register renaming across both modes.

During the normal mode, the vector processor dispatches instructions using the head pointer. If the dispatched flag is activated, the instruction proceeds to the rename/contextualize stage for contextualization. This proposed mechanism incorporates contextualization in order to prevent the flushing of data in physical registers that have been calculated in the critical mode.

Migrate pending commit queue entry During the contextualization, the entry indicated by the head pointer of the PCQ is transferred to the ROB. This is because the last early-dispatched instruction is the most recently contextualized. As the instructions in the DIQ are arranged according to the program order, transferring the entry to the ROB during contextualization ensures that the ROB is updated according to the program order

Update the register alias table and evict from the critical register aliasing table The contextualization process updates the RAT using the entry transferred from the PCQ to the ROB, which contains information about the physical register of the destination operands. As the information on the logical register of the destination operands required to update the RAT can be obtained from the DIQ, the vector processor is able to update the physical register of the RAT entry using the head of the PCQ.

During eviction of an entry from the PCQ, the vector processor compares the physical registers of the destination operands of the evicted entry with those of the corresponding entry in the CRAT having the same logical registers of destination operands. If these indicate different physical registers, the CRAT retains the entry as another instruction with the same logical register uses it. However, if both registers are the same, the CRAT entry is also evicted, as no

other logical registers exist in the PCQ. This contextualization stage enables the values in registers calculated in the critical mode to be made available to the next dispatched instruction in the normal mode.

4.3.4 Memory disambiguation

Since instructions in the critical mode are dispatched early, the order in which they are dispatched does not align with the normal mode. Despite the proposed mechanism statically preserving the dependencies of these early-dispatched instructions, it is possible for the addresses of early-dispatched load instructions to overlap with those of preceding store instructions. However, this may not be a concern for applications running on vector processors, because optimization for vectorization often assumes that there is no memory aliasing. In other words, vectorized programs are often optimized by adding flags such as the "__restrict" modifier in the C/C++ language, which assumes that there is no overlap. Thus, this study assumes that statically unanalyzable memory disambiguation does not occur for these applications.

4.4 Evaluation

4.4.1 Experimental setup

The proposed mechanism is evaluated by using a simulator developed based on Gem5 [62]. The configuration of the vector processor is shown in Table 4.1. The core specifications are similar to the latest vector processor, SX-Aurora TSUBASA [16]. As the sizes of the ROB and the issue queue of SX-Aurora TSUBASA are not available, this chapter assumed the sizes that most closely match the performance of the actual machine on the simulator. On the single-core configuration, the memory system scales down in proportion to the number of cores from the actual product, VE 20B of SX-Aurora TSUBASA, to evaluate the single-core configuration. The number of physical registers is 256, which conforms to the specification of the SX-Aurora TSUBASA. The architecture does not distinguish the physical registers for integer and floating-point operations. To switch the front-end of the pipeline between the critical mode and the normal mode, the vector processor takes one cycle penalty.

Since the simulator is trace-driven, instruction traces is obtained using the GDB for SX-Aurora TSUBASA. The instruction traces are five million instructions of the kernel part of each benchmark. To avoid the warming up phase of applications on simulations, the trace does not contain non-essential parts, such as initialization or other non-vectorized parts.

In the evaluation, the following four configurations are compared to clarify the advantage of the proposed mechanism for vector processors:

OoO The baseline out-of-order vector processor.

ROB144 A vector processor with the expanded ROB size of the baseline configuration for examining the effect of simply increasing the number of in-flight

Table 4.1: Baseline configuration for the out-of-order vector processor.

Frequency	1.4 GHz
Number of cores	1
Type	Out-of-order
ROB size	48
Issue queue size	48
Load queue size	32
Store queue size	32
SPU issue width	4 insts./cycle
Functional units	3 FMA (8 cycles), 2 ALU (8 cycles), 1 DIV (div 32 cycles, sqrt 64 cycles)
Register file	256 (int+fp, identical)
LLC cache size	2 MB, assoc 4
Cache bandwidth	410 GB/s, 2 cycles
Memory bandwidth	HBM-Like conf., 1GHz channels: 8, ranks: 1, banks: 16, bus: 128 bits `tRP-tCL-tRCD`: 15-15-15
decoded instruction queue size	768 entry
Stalling instruction cache size	144 entry
Stalling threshold	150 cycles
Pending commit queue size	96 entry
Critical register aliasing table	64 entry

instructions.

PRE A vector processor with the precise runahead execution [41], which is the state-of-the-art runahead mechanism. In our implementation, the PRE can track up to 4096 vector instructions. The size of the PRDQ is 96, which is the same as the size of the PCQ.

COV A vector processor with the proposed mechanism.

4.4.2 Applications

The proposed mechanism for vector processors is evaluated on three sets of applications. Each kernel has a single loop structure that iterates a sufficient number of times for each application to reach a stable state. The first application set is from the PolyBench [63, 64]. The PolyBench suite contains primitive HPC kernels without any optimization. Kernels that the NEC compiler can vectorize without specific options or modifications for the evaluation are chosen. Additionally, an optimized version of the same kernel with the PolyBench suite is prepared from NEC Numeric Library Collection, a well-optimized calculation library including BLAS/LAPACK kernels. These optimized applications are listed as "_opt" suffixes in the benchmark name. The size of datasets is "large," not to exceed the main memory size of a single Vector Engine on SX-Aurora TSUBASA on all the kernels. Tables 4.2 and 4.3 show the bytes per flop of the applications. The actual B/F indicates the bytes per flop value obtained by the count of the read and write accesses to the memory on the simulator.

The second application set is the practical application kernels of HPC applications from the different research areas, which are continuously used to measure the performance of vector processors [2, 19]. These applications show the typical behavior of the HPC applications that are run on vector processors. Table 4.4 shows the detail of the applications. As these applications are well optimized for vector processors, 99 % of the code is vectorized.

The third application set is five algorithms from Vector Graph Library [71] with seven datasets. The algorithms are graph algorithms with highly-vectorized and optimized. These algorithms are characterized as memory-intensive due to

Table 4.2: The Bytes per flop of PolyBench.

Application	Description	Actual B/F
2mm	2 Matrix Multiplications	4.00
3mm	3 Matrix Multiplications	0.16
adi	Alternating Direction Implicit solver	14.1
atax	Matrix Transpose and Vector Multiplication	4.63
bicg	BiCG Sub Kernel of BiCGStab Linear Solver	4.00
cholesky	Cholesky Decomposition	0.06
correlation	Correlation Computation	3.92
covariance	Covariance Computation	113
doitgen	Multiresolution analysis kernel (MADNESS)	0.06
fdtd-2d	2-D Finite Different Time Domain Kernel	7.29
fdtd-apml	FDTD using Anisotropic Perfectly Matched Layer	0.80
gemm	Matrix-multiply	4.00
gemver	Vector Multiplication and Matrix Addition	3.50
gesummv	Scalar, Vector and Matrix Multiplication	3.94
gramschmidt	Gram-Schmidt decomposition	6.05
jacobi-1d	1-D Jacobi stencil computation	0.01
jacobi-2d	2-D Jacobi stencil computation	10.4
lu	LU decomposition	8.60
lucmp	LU decomposition	0.43
mvt	Matrix Vector Product and Transpose	4.12
symm	Symmetric matrix-multiply	117
syr2k	Symmetric rank-2k operations	3.06
syrk	Symmetric rank-k operations	3.35
trisolv	Triangular solver	3.88

the need to access complex data structures, and they often include many integer operations and few floating-point operations. Table 4.5 shows the five algorithms used in the evaluations.

4.4.3 Assumptions

Note that since vector processors usually handle conditional branches using the mask register, this dissertation assumes that speculative executions using branch prediction do not appear in the kernel code. Additionally, this dissertation assumes that branch prediction at the end of the loop is as accurate as

Table 4.3: The Bytes per flop of the optimized version.

Application	Description	Actual B/F
gemm_opt	Matrix-multiply	0.08
symm_opt	Symmetric matrix-multiply	0.10
syr2k_opt	Symmetric rank-2k operations	0.14
syrk_opt	Symmetric rank-k operations	0.10
trmm_opt	Triangular matrix-multiply	0.12

Table 4.4: The overview of the practical application kernels.

Application	Research field	Method	Actual B/F
Antenna [65]	Electromagnetic	FDTD	0.81
TurbulentFlow [66]	CFD	Navier-Stokes equation	0.56
Earthquake [67]	Seismology	Friction Law	4.00
Landmine [68]	Electromagnetic	FDTD	6.05
Turbine [69]	CFD	LU-SGS method	0.34
Plasma [70]	Physics	Lax-Wendroff	0.88

possible and has no impact on performance.

This dissertation also assumes that the accuracy of analyzing dependencies of critical instructions is perfect. Because this dissertation targets the applications where the vector processors handle branch instructions by the vector mask registers, criticality-aware dispatching on the wrong path caused by branch misprediction hardly occurs. One of the candidates to make these assumptions feasible is that the programmer or the compiler should be able to notify which part of the code can be early-dispatchable by such as user directives.

The applications are compiled without the *-mvector-floating-divide-instruction* option as the default so that *divisions* and *square roots* can be compiled as reciprocal instructions. Although the proposed mechanism can track any vector instructions in a chain, the evaluated applications do not include *fdiv*/*sqrt* instructions in their object codes.

Table 4.5: The algorithms of Vector Graph Library.

Name	Algorithm
bfs	Breadth-First Search [28]
cc	Connected Components [29]
pr	Page Rank [72]
sssp	Single Source Shortest Paths [30]
scc	Strongly Connected Components [30]

4.4.4 Area overheads

The proposed mechanisms require four additional components: two FIFO queues, one full-associative cache, and one CAM-based table. The DIQ requires a circular buffer-type FIFO queue with three-pointers. Each entry has the same field as the instruction buffer that the out-of-order processor is internally equipped with in the decode stage (our configuration is 8-byte for each entry) and additionally requires 1-bit for the critical flag. These three pointers require a total of 30 bits; each 10-bit pointer can identify 768-entry in the DIQ. The DIQ requires approximately 6.3 KB for 768 entries. The SIC requires a total of 1152 bytes, which can store 144 8-byte instructions. The PCQ requires the same field as the ROB. Our configuration requires 83 bits for each entry and has 96 entries. Thus, the PCQ requires approximately 1 KB. The CRAT entry is 8-bit for logical registers and 8-bit for physical registers. Thus, the CRAT uses the 1 KB area budget of the CAM-based table. Overall, the proposed mechanism requires an 8.3 KB area budget.

4.4.5 Performance

Figures 4.4, 4.5, and 4.6 show the results of the performance of the PolyBench suite, the practical application kernels, and the vector graph library, where they

are normalized by the baseline out-of-order configuration. The Floating Operations Per Second (FLOPS) is used to evaluate the sustained performance of the PolyBench suite and the practical application kernels. Since the vector graph library applications do not include floating operations, the committed vector elements per second as the performance evaluation metric are used. The geometric mean in each figure is the value for the respective benchmark set. COV achieves an 8.4 % performance improvement on average compared to OoO across all the benchmarks. ROB144 achieves a 4.8 % performance improvement, and PRE degrades performance by 0.3 %. These results indicate that our proposed mechanism outperforms the processors increasing the size of the ROB and the previous work.

4.4.5.1 PolyBench case

Figure 4.4 shows that COV achieves an 8.6 % performance improvement on average in the PolyBench suite, while ROB144 and PRE achieve 3.2 % and -0.55 % performance improvements, respectively. On the benchmarks such as the *fdtd-apml* kernel, both COV and ROB144 achieve a 30% or more performance improvement, which indicates that the kernel benefits from increasing the ROB size. As shown in Table 4.2, these benchmarks have low actual B/F values, and therefore, there is room in the bandwidth for early-dispatching of the critical instructions. Since almost all memory access instructions are marked as critical in these benchmarks by the proposed mechanisms, increased ILP can improve performance.

On the other hand, *doitgen* and *jacobi-1d* do not show any performance improvement despite the low actual B/F values. Because there are almost no stalls in the processor due to sufficient memory bandwidth, COV does not dispatch

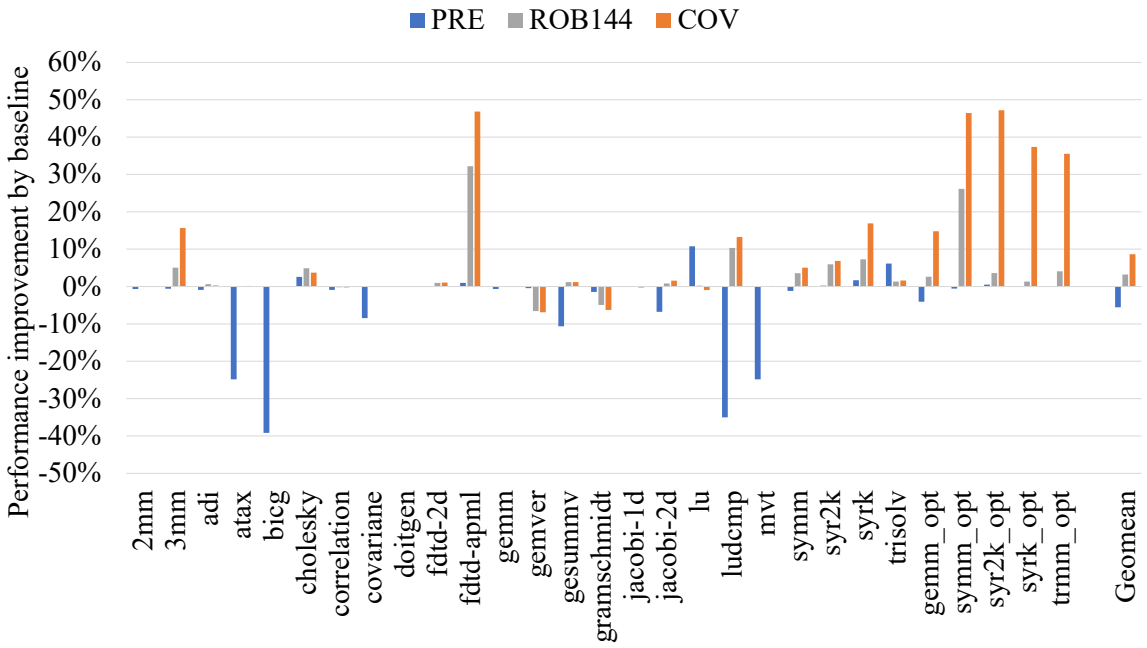


Figure 4.4: The performance evaluation results on the PolyBench suite, normalized by the baseline out-of-order configuration.

any critical instructions, resulting in no performance improvements.

COV outperforms ROB144 and PRE in the case of most kernels, especially the optimized versions such as *gemm_opt*, *symm_opt*, *syr2k_opt*, and *syrk_opt* kernels. There are two reasons why the optimized versions can achieve performance improvement. First, since the optimized versions can maximize the vector length, the efficiency of each vector instruction is simply higher than the reference version by reordering instructions. Second, the optimized versions exploit the temporal and spatial localities to improve the cache hit rate. This optimization significantly contributes to the realization of the higher performance of COV compared to ROB144 because the criticality-aware mechanism can easily find the cache misses that software optimizations cannot cover. These results imply that combining our proposal with software optimization, such as cache blocking, may enhance the possibilities for performance improvements.

3mm is a simple kernel that does three matrix multiplications. The compiler

could detect such patterns as idioms and transform the code into optimized versions. This transformation contributes to performance improvement, similar to the optimized versions discussed in the *_opt* benchmarks. Unfortunately, the compiler could not detect an idiom in the case of *2mm*.

In the cases of *gemver* and *gramschmidt*, ROB144 and COV degrade the performances. Because the performance bottleneck of these kernels is due to the lack of the ROB size, ROB144 and COV alleviate the bottleneck. However, the bottleneck moves to the lack of the issue queue in the cases of ROB144 and COV. Due to the limitations of the scheduling algorithm of the issue queue, there are cases where scheduling hurts its performance. These applications are susceptible to minor changes in the ordering of resource allocation; in such cases, the performance is degraded.

4.4.5.2 Practical application kernels

Figure 4.5 shows that COV achieves a 17.8 % performance improvement on average in the practical application kernels, while ROB144 and PRE achieve 11.7 % and 0.26 % performance improvements, respectively.

The *Antenna*, *Plasma*, *Turbine*, and *TurbulentFlow* kernels significantly achieve performance improvements in COV and ROB144, because these kernels involve a large number of instruction-level parallelism. Since these kernels have more instructions in one iteration than other kernels, as shown in Figure 4.2c, COV and ROB144 realize performance improvement.

Among these kernels, the *Turbine* kernel achieves a 27% performance improvement on COV, whereas ROB144 realizes only a 7.5% performance improvement. This indicates that the *Turbine* requires increasing instruction-level parallelism and handling critical instructions, which contributes significantly to

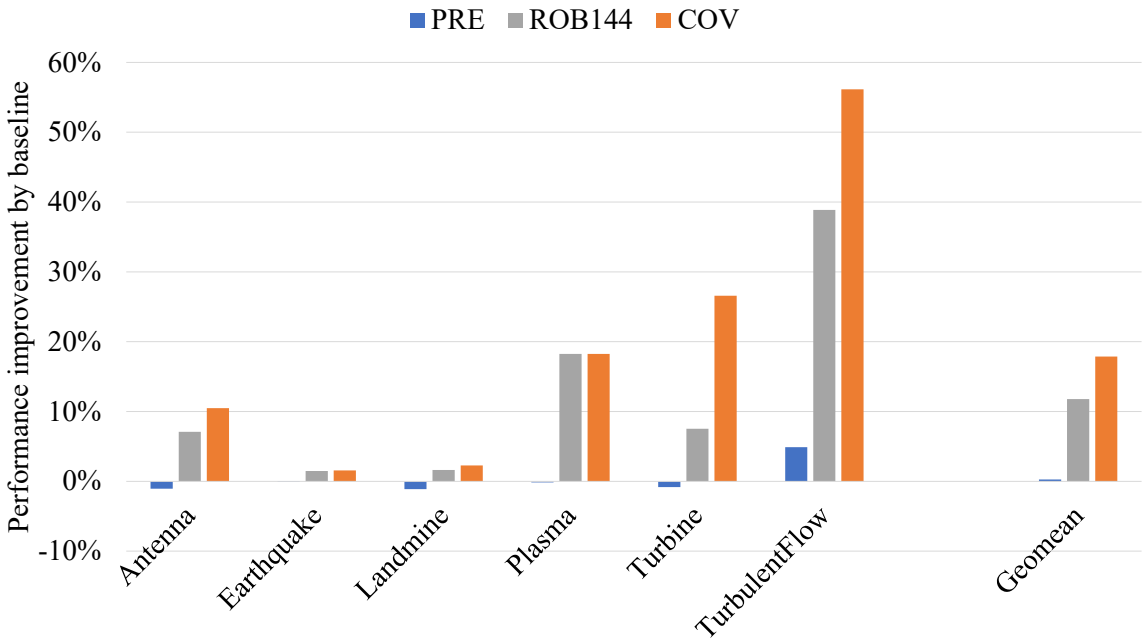


Figure 4.5: The performance evaluation results on six benchmarks, normalized by the baseline out-of-order configuration.

performance improvement. As shown in Figure 2.5, the *Turbine* kernel contains about 200 instructions in one iteration. ROB144 can only exploit 144 instructions that are not enough to reach the next stalling instruction chain. On the other hand, COV can exploit 768 instructions, which can exploit critical instructions over multiple iterations.

The *Plasma* kernel contains many vector gather instructions. The vector processor modeled in this evaluation has a limitation of reordering between a vector gather instruction and a vector store instruction in the issue queue. COV regards these instructions as critical and dispatches them in the critical mode; however, the processor cannot execute them due to the above limitations. On the other hand, ROB144 processes the instructions regardless of the criticality. Therefore, ROB144 outperforms COV. These facts indicate that the criticality-aware execution is less effective in the case of the instructions causing many hardware structural hazards.

The *Earthquake* and *Landmine* kernels are extensively memory-intensive kernels. There is no room to improve beyond the memory bandwidth bottleneck.

The PRE could only realize performance improvement on the *TurbulentFlow* kernel. The *Turbine* kernel is a cache bandwidth bottleneck whose cache hit rate is already 99% on the baseline configuration. Thus, the additional memory requests induced by PRE can deteriorate performance. In the *Antenna* kernel, the critical instructions are vector load instructions that cause the strided memory accesses. Compared to sequential memory accesses, the strided memory accesses bring more cache blocks into the memory system. The PRE speculatively executes such strided memory accesses as a result of prefetching. Although PRE could improve the cache hit rate, PRE causes a shortage of the cache bandwidth, resulting in performance degradation. Since COV and ROB144 do not increase the number of memory accesses, the *Antenna* kernel achieves performance improvement.

4.4.5.3 Vector Graph Library

Figure 4.6 shows that COV achieves a 6.8 % performance improvement on average in the vector graph library set, while ROB144 and PRE achieve 5.0 % and -0.7 % performance improvements, respectively.

COV can achieve higher performance improvements on the *bfs* and *scc* applications than the other applications. These applications suffer from the shortage of the ROB. Since ROB144 has the increased size of the ROB, ROB144 can solve the shortage and realizes performance improvement. COV can virtually increase the size of the ROB for critical instructions by the PCQ. COV initiates critical instructions by early dispatching, which enables further performance improvement.

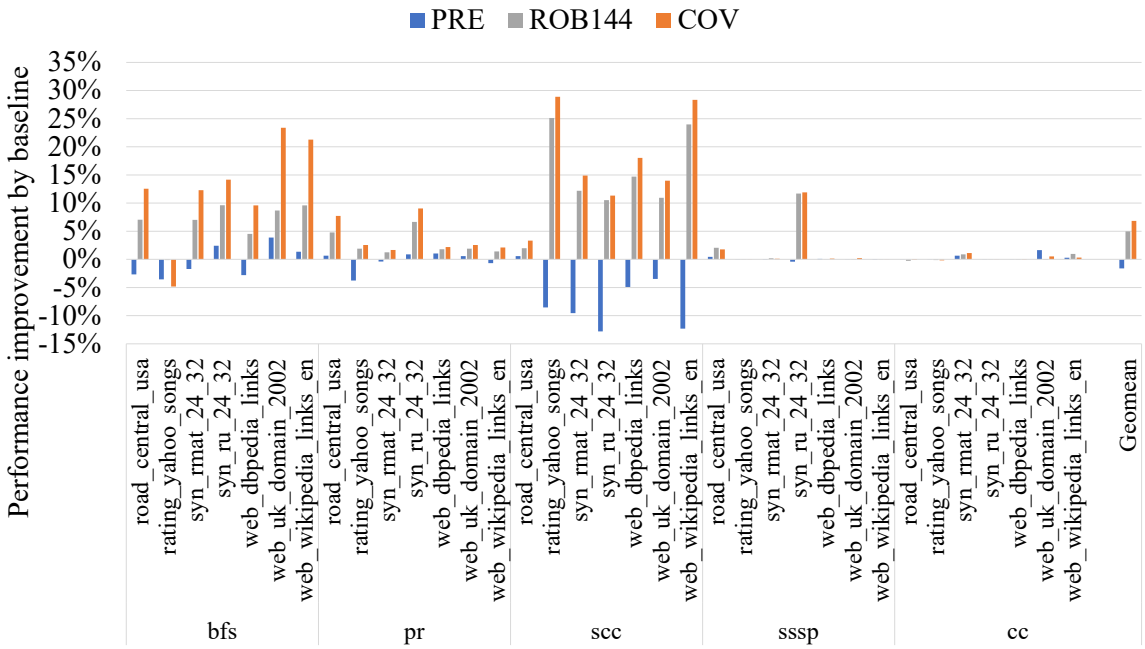


Figure 4.6: The performance evaluation results on the vector graph library, normalized by the baseline out-of-order configuration.

The *pr* algorithm achieves performance improvement on ROB144 and COV. Although the *pr* application does not contain indirect memory accesses such as vector gather or scatter instructions, some vector load instructions cause pipeline stalling. Increasing instruction-level parallelism allows the processor to process in-flight multiple stalling vector load instructions.

The *sssp* and *cc* algorithms cannot archive performance improvement in all the processors. The *sssp* and *cc* algorithms contain many vector gather instructions, and the number of memory requests that the vector processor can handle is saturated on the baseline configuration. Therefore, COV and ROB144 cannot make any remedy for these applications.

In particular, the *cc* algorithm especially contains many vector scatter instructions in addition to vector gather instructions. As discussed in the case of the *Plasma* kernel, there are limitations in reordering memory instructions between a vector scatter instruction and a vector load instruction due to the

limitation of the architectural resources, making COV difficult to improve performance.

4.4.6 Analysis of criticality-aware executed instructions

Section 4.4.6 provides how the proposed mechanism handles stalling vector instruction chains in loops. Overall, the proposed mechanism tends to identify the first load instruction of each iteration in the loop as critical in most applications. This is because these instructions are likely to cause cache misses and delay the subsequent instructions. Thus, by executing these instructions earlier, the proposed mechanism can improve performance.

There are other cases for the Turbine and TurbulentFlow kernels where the performance improvement is particularly more significant than the other applications. In the case of the Turbine kernel, the lengths of some chains are longer than ten to track the instructions with vector gather instructions. Since the Turbine kernel uses indirect memory accesses with complex index calculations, the vector gather instructions often appear in the loop. To calculate the index for the indirect memory accesses, the Turbine kernel uses several instructions such as *logical and/or instructions* and *floating max/min instructions* so that the proposed mechanism can identify these instructions as a stalling instruction chain of the vector gather instruction and its dependency.

In the case of the TurbulentFlow kernel, the proposed mechanism identifies some vector store instructions as the stalling instructions. Since a vector store instruction must have a long instruction chain, all instructions of the calculation path have been wrapped up and isolated from the normal mode when enough iterations have progressed to track the entire instructions by the SIC.

These facts indicate that the proposed mechanism can accelerate the critical

path in the loop.

4.4.7 Architectural sensitivity study

Section 4.4.7 explores the effects of the queue sizes of the mechanism, the number of physical vector registers, and the switching penalty of the mode on the performances while the other parameters are fixed to a maximum size of the parameters. The sensitivity study is performed by 44 benchmarks in which the proposed mechanism can realize 1% or more performance improvement. Note that the average values of the 44 benchmarks show the same trend as the average of all the benchmarks.

Figure 4.7 shows the performance normalized by the lowest case with the various sizes of the DIQ. The size of the DIQ indicates how far the proposed mechanism can execute future instructions. The proposed mechanism could improve the performance by the size of 768 for the DIQ. As shown in Figure 4.2c, the practical application kernels contain hundreds of vector instructions in one iteration. If the size of the DIQ covers all the number of vector instructions in one iteration, the processor can aggressively dispatch the upcoming instructions across loop iterations. It contributes to performance improvement if the resources such as vector registers and IQ slots are not utilized.

Figure 4.8 shows that the performance is saturated when the size of the PCQ is around 96. The size 96 of the PCQ is twice larger than the size of the ROB. The sizes of the issue queue and the ROB are estimated based on the real product, VE Type 20B of SX-Aurora TSUBASA, in the baseline configuration. The vector processor can still exploit ILP for the benchmarks, especially the benchmarks including critical vector instructions. If the vector processors explore critical

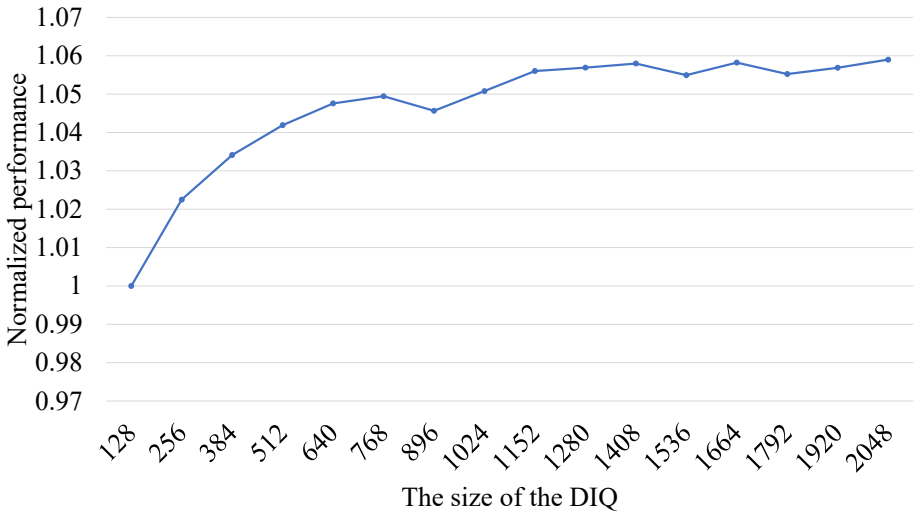


Figure 4.7: Effect of the different size of the DIQ.

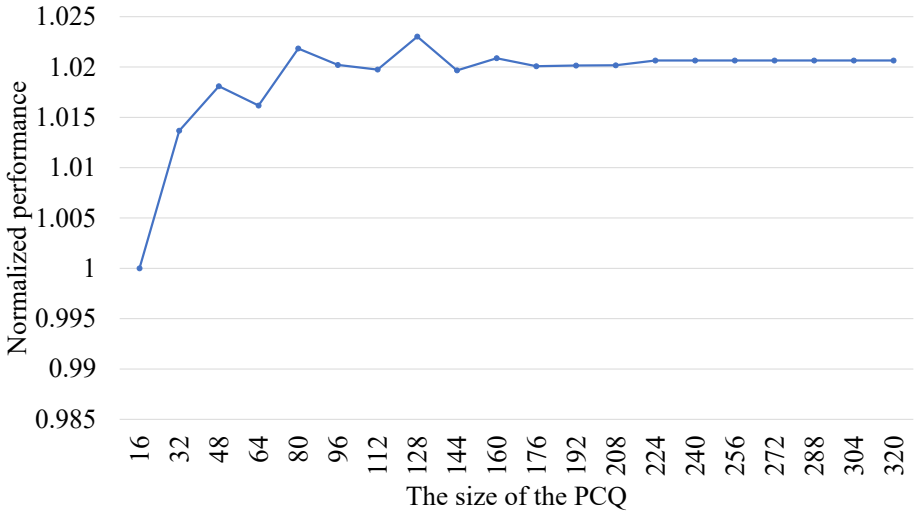


Figure 4.8: Effect of the different size of the PCQ.

instructions from future instructions, up to 96 critical instructions can be in-flight inside the processor on this configuration.

Figure 4.9 shows that the performance becomes nearly saturated when the size of the CRAT is 48. Although the number of architectural registers is 64, these results indicate that the number of CRAT entries is less than architectural registers because the CRAT is only used in the critical mode. Since the critical

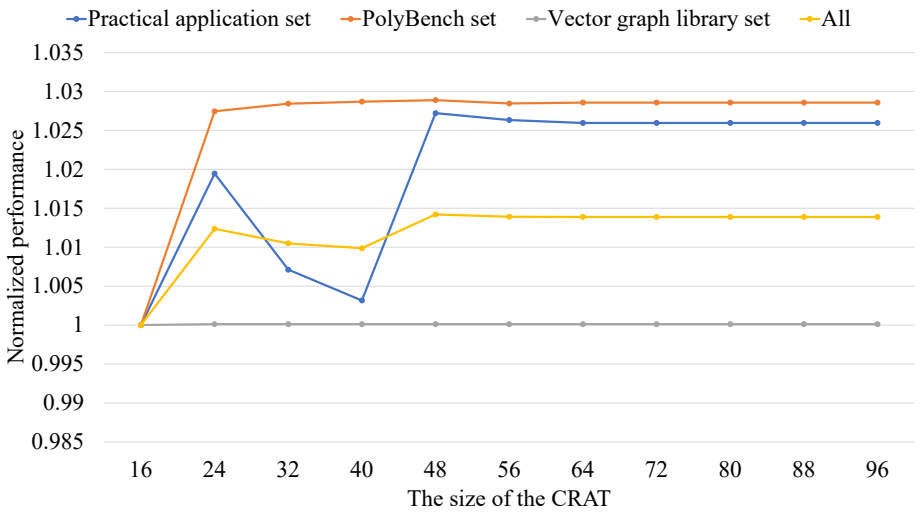


Figure 4.9: Effect of the different size of the CRAT.

instructions tracked by the SIC use a limited number of logical registers, the variation of the logical registers used in the critical mode only covers some of the logical registers.

Moreover, the performances of the Turbine and TurbulentFlow kernels are improved with 24 or more CRAT entries. However, the performances of the PolyBench suite and the vector graph library set are not improved. These kernels have long instruction chains for criticality-aware execution, as described in Section 4.4.6. In the case of having long instruction chains in the application, the proposed mechanism cannot track the whole chains with two situations to fail early-dispatching, as explained in Section 4.3.3. The first situation is the lack of resources to allocate a new entry for early-dispatching. In this situation, the proposed mechanism stops dispatching in the critical mode. The second situation is the lack of entries for source operands in the CRAT for early-dispatching. In this situation, the instruction of the source operands is already contextualized, not in-flight; the proposed mechanism cannot early dispatch the instructions of the chain. Thus, the proposed mechanism skips these instructions of the chain and

continues to seek the next instruction of other chains. This behavior causes the performance fluctuation of the practical application set shown in Figure 4.9.

On the other hand, the kernels in the vector graph library are not affected by the size of the CRAT. This is because these kernels have relatively short stalling instruction chains. The size of 16 for the CRAT is already sufficient for these kernels.

Figure 4.10 shows that the performance becomes nearly saturated when the size of the SIC is 144. The size of the SIC affects the number of critical instructions that the vector processor can track. From Figure 4.10, 144 is enough for the size of the SIC in this configuration.

Figure 4.11 shows the performance of the various switching latency of the normal mode and the critical mode, normalized by the case of the switching latency of 1. The performance declines by about 3% in the case of six cycles and gradually decreases after eight or more cycles.

Figure 4.12 shows the performance under the various number of physical vector registers, normalized by the case of 64 physical vector registers corresponding to the number of architectural vector registers of the original vector architecture. The performance becomes saturated at 144 physical vector registers on COV, while the baseline becomes saturated at 112. This is because the proposed mechanism uses more physical registers than the baseline for criticality-aware execution.

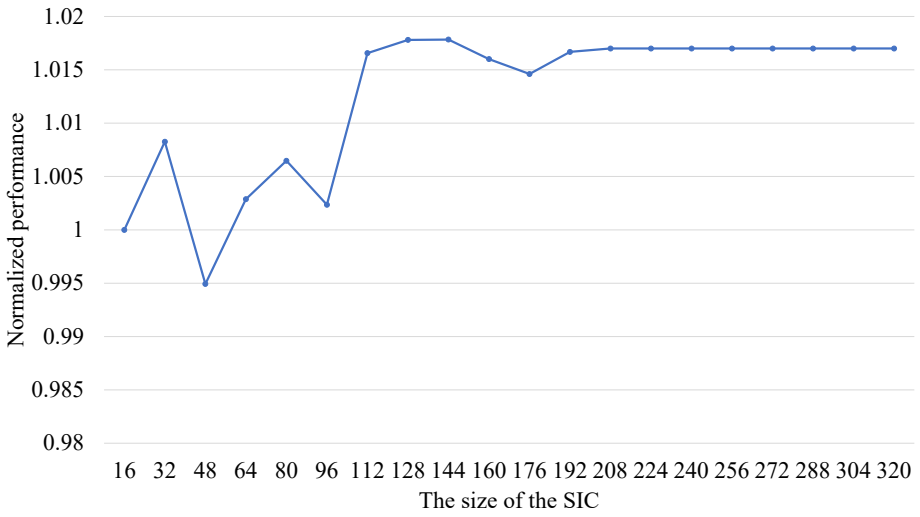


Figure 4.10: Effect of the different size of the SIC.

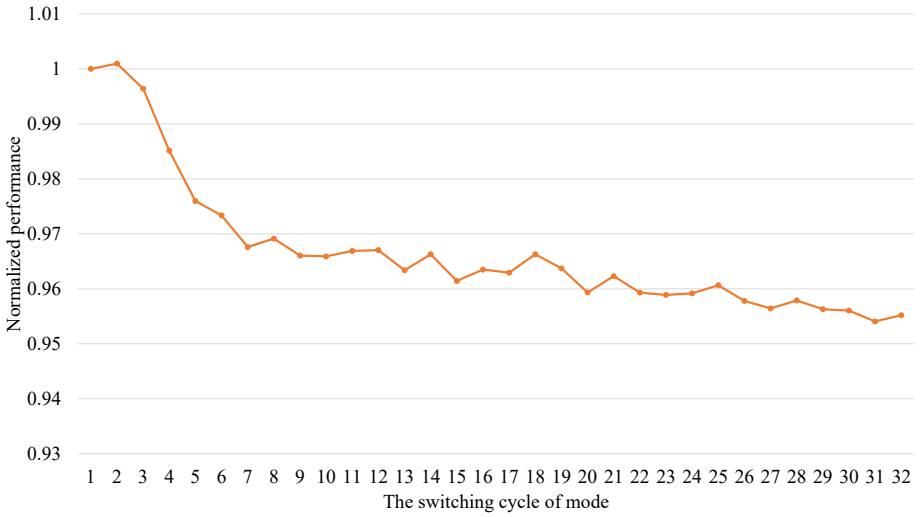


Figure 4.11: Effect of the different switching latency.

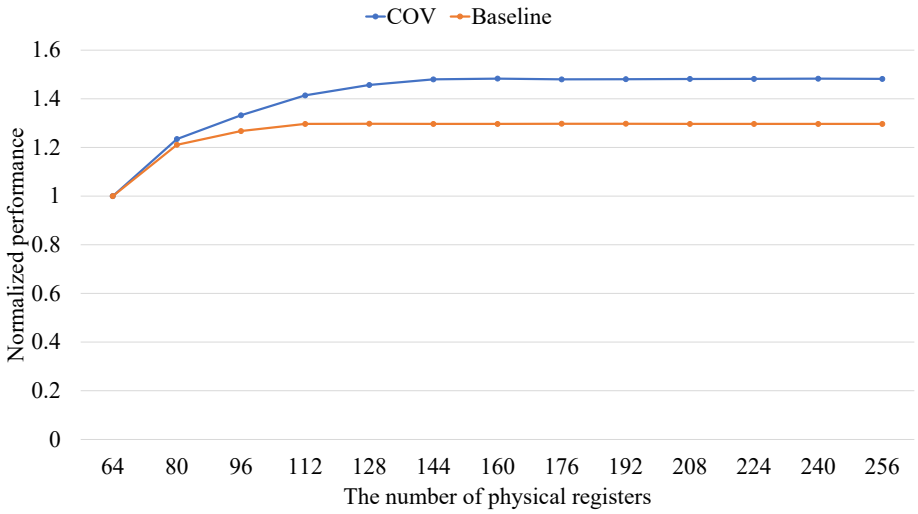


Figure 4.12: Effect of physical vector registers.

4.5 Conclusions

Modern vector processors have achieved high sustained performance in HPC applications due to their powerful instruction set. The latest vector processors utilize out-of-order execution of vector instructions to exploit ILP, as there is a significant latency gap between vector arithmetic instructions and vector memory access instructions, even in modern out-of-order vector processors.

This chapter has proposed a criticality-aware out-of-order mechanism for vector processors to further exploit ILP to hide the latency of vector instructions. The proposed mechanism early dispatches the subsequent instructions that may incur pipeline stalling, enabling the exploitation of ILP. Unlike the conventional runahead mechanisms, our proposed mechanism leaves the registers containing the results and uses them in the normal mode. The key idea is that additional queues hold the information of the instructions in the critical mode. After going back to the normal mode, the processor dispatches the instructions referring to these queues. If there are instructions executed in the critical mode, the information of these instructions is migrated from the queues used in the critical mode to queues used in the normal mode. This enables the vector processor to migrate the commit order information and the register aliasing information of the early-dispatched instruction to the normal mode. The evaluation results show that the proposed mechanism achieves an 80% performance improvement at a maximum, an 8.7% on average over the conventional mechanism in all applications.

By utilizing the proposal method, the vector processor can conceal the latency beyond conventional out-of-order mechanisms until vector instructions are issued. As a result, the vector processor can achieve latency tolerance.

Chapter 5

Page-address coalescing method of vector gather instructions

5.1 Introduction

Virtual memory enables processes to handle memory addresses that are independent of the actual memory addresses, thus preventing address conflicts between processes. When a processor accesses memory, it must translate virtual memory addresses to physical memory addresses. In the case of memory accesses using vector load instructions, the access pattern is sequential. Therefore, the vector processor can efficiently access memory in blocks of several contiguous elements. As the number of blocks accessed by the vector load instruction is smaller than the number of vector elements, the number of accesses to the Translation-look-aside-buffer (TLB) is reduced.

On the other hand, when using vector gather instructions, the access pattern becomes irregular rather than sequential. As accesses made using vector gather

instructions can be distributed throughout the physical memory space, the processor must obtain physical addresses for all virtual addresses. As a result, the number of translations required is equal to the number of vector elements. This means that TLB throughput can potentially become a limiting factor in terms of performance.

This chapter proposes a coalescing method for reducing the number of TLB accesses to reduce the address translation latency. Before translating the virtual memory addresses specified in a vector gather instruction to physical memory addresses, the proposed method reduces the number of TLB accesses by checking whether the virtual addresses are located in the same page. If the virtual addresses are in the same page, the processor can access TLB using a reduced number of coalesced virtual addresses.

5.2 Motivation

5.2.1 Vector Gather Instruction

The vector gather instruction enables indirect memory accesses as part of the vector instruction set. Listing 5.1 shows a simple code written in the C language that includes indirect memory accesses in a loop. The elements of array B are accessed using array L. Indirect memory accesses are commonly used in various numerical computations and graph applications. For instance, when working with sparse matrices in a compressed format, memory accesses become indirect memory accesses.

The vector gather instruction is responsible for managing indirect memory accesses. By accepting the value of each element in the vector register as a memory address, the vector gather instruction initiates a memory access for each vector element. The addresses contained within the vector register are process-specific addresses known as *virtual addresses*, which are distinct from the actual memory locations referred to as *physical addresses*. The memory system performs address translation using TLB to access the physical address space.

Source Code 5.1: Example of indirect memory access

```
1 for (i=0; i<n; i++){
2     A[i] = s * B[L[i]]
3 }
```

5.2.2 Vector Gather Instruction with Virtual Memory

During address translation for vector instructions, the addresses must be translated for each vector element. If consecutive accesses can be anticipated, such as with vector load instructions, the minimum number of required pages can be predicted based on the range of virtual addresses. In contrast, with vector gather instructions, where memory accesses are not sequential and the addresses are unpredictable, it is necessary to translate all of the virtual page numbers in the addresses to physical page numbers.

For sequential accesses, such as vector load instructions, a considerable amount of TLB bandwidth is not necessary. On the other hand, indirect memory accesses, such as vector gather instructions, require a greater number of address translations in comparison to sequential accesses, making TLB bandwidth vital for optimizing performance. However, increasing the TLB bandwidth can be challenging due to the high energy consumption of the TLB component in the processor [73].

5.3 Page-Address Coalescing for Vector Gather Instruction

This chapter proposes a page-address coalescing method for vector gather instructions.

5.3.1 Idea to use existing vector arithmetic units for page-address coalescing

The proposed method in this chapter aims to minimize TLB bandwidth usage for vector gather instructions by examining the number of pages accessed per vector gather instruction. If multiple data is located in the same page, it can be translated with a single TLB access, reducing TLB bandwidth and potentially improving the performance of vector gather instructions. Figure 5.1 illustrates the percentage of pages per vector gather instruction for a 64 MB page size. The input size for graph applications is maximized to fit within the memory size of the machine. From Figure 5.1, it can be seen that the majority of applications access one or two pages in a vector gather instruction. Although the percentage may be influenced by several factors such as page size, application input size, and others, this information inspires us to coalesce page addresses and reduce TLB bandwidth usage.

The proposed method in this chapter focuses on the vector arithmetic units already present in the processor. It suggests that by combining multiple arithmetic instructions, the process of page-address coalescing can be implemented to reduce the number of TLB accesses by deduplicating the pages of the virtual address of the vector gather instruction in advance.

However, if a vector gather instruction occupies the vector arithmetic units,

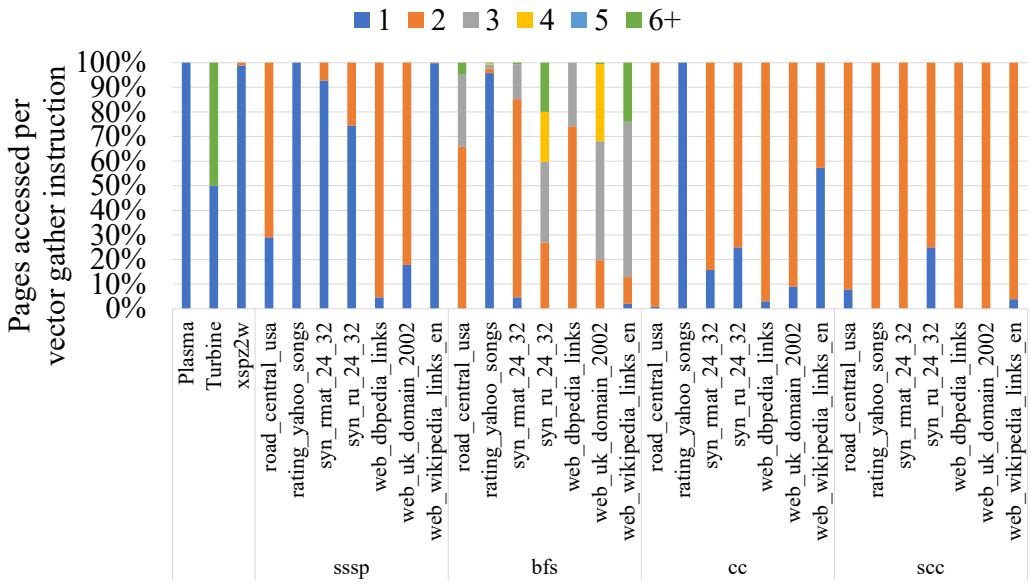


Figure 5.1: The percentage of pages per vector gather instruction.

other instructions may be unable to use them, resulting in structural hazards. To evaluate the potential of page-address coalescing, this section conducted a preliminary evaluation to determine how many cycles the benefits of page-address coalescing would outweigh the performance loss due to structural hazards. In this preliminary evaluation, two assumptions are made. First, the processor can ideally coalesce the virtual addresses before TLB accesses. Second, all vector gather instructions occupy a vector arithmetic unit for a predefined number of cycles during page-address coalescing.

Figure 5.2 displays the results of the preliminary evaluation, with the vertical axis representing the average performance with page-address coalescing normalized by the performance without coalescing, and the horizontal axis showing the number of cycles during the period each vector gather instruction occupies a vector arithmetic unit. The figure illustrates that the processor can consume at least 92 cycles for page-address coalescing per vector gather instruction without suffering from a decline in performance. This suggests that if the processor

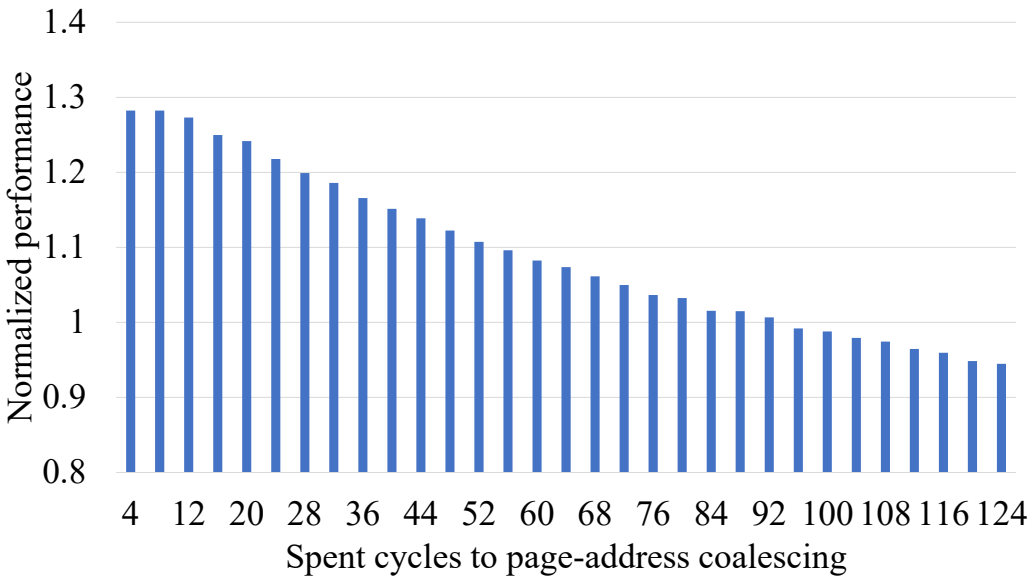


Figure 5.2: The cycles usable for page-address coalescing.

is able to perform page-address coalescing in a small number of cycles, an improvement in performance can be achieved.

5.3.2 Procedure of page-address coalescing

It is necessary to distinguish whether each element of a vector instruction accesses the same page or a different page. This must be done using operations that can be performed by the vector arithmetic unit provided in the vector processor.

Prior to discussing the details of the processor's implementation of page-address coalescing, it is important to outline the general procedure for this process. As depicted in Figure 5.3, page-address coalescing involves two stages. During the first stage, the virtual address contained within the vector operand is split into its virtual page number and offset components. In the second stage, a vector XOR operation is performed between the first element and all other

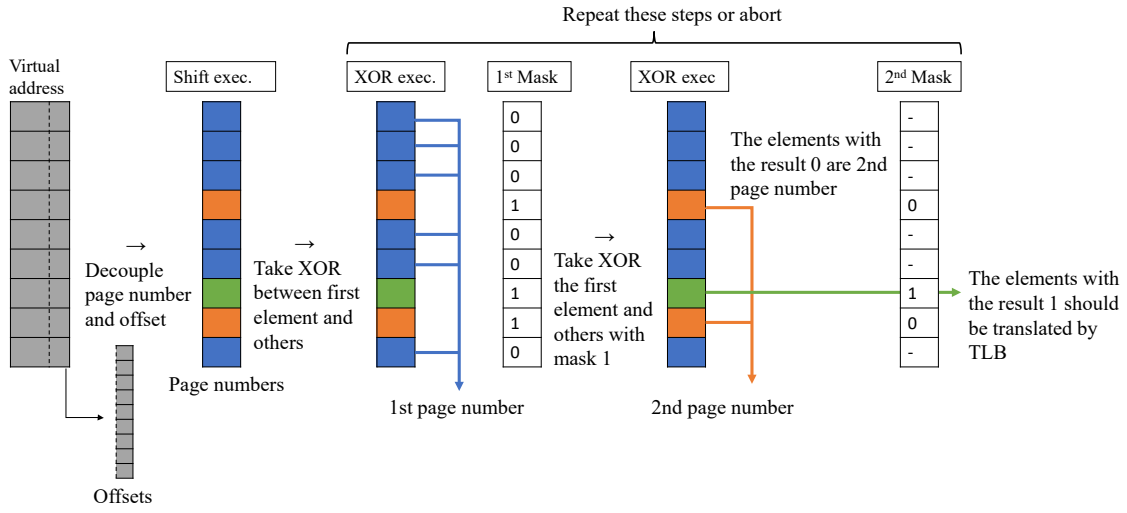


Figure 5.3: The example of the proposed address coalescing method.

elements in the vector holding the virtual page numbers. Taking the blue element at the top of Figure 5.3 as an example, the outcome of this XOR operation results in zeros in the blue box and non-zeros in the other boxes.

By these two steps, the first coalesced address can be obtained. If the result of the vector XOR operation shows zeros, then the page number is the same as the blue element in Figure 5.3. This result is then used as a mask and transferred to the mask register, with non-zero values being converted to ones, as shown in Figure 5.3. This represents the first attempt at coalescing the page addresses. If the mask is not entirely composed of zeros and further coalescing is required for the remaining elements, additional steps must be taken for the next iteration.

In the subsequent steps, the XOR operation is performed between the element with the first mask bit set to 1 and all other elements with mask bits set to 1. In the example depicted in Figure 5.3, the element corresponding to the top orange box represents the second coalesced virtual page number. As elements with non-zero mask values have not yet been coalesced, the processor executes the XOR operation using this element and other elements specified by the first mask. If the second XOR result is zero, then the page number is the same as the

orange element. This marks the second trial in coalescing page addresses.

This process can be repeated until all mask bits are zero or until the page-address coalescing is prematurely terminated. If the latter occurs, the remaining virtual page numbers are translated by the TLB as usual. In the example shown in Fig 5.3, the page number represented by the green box does not match any of the other boxes. As this page number cannot be coalesced through two iterations, it must also be translated by the TLB.

5.3.3 Implementation

To implement the page-address coalescing described in Section 5.3.2, the proposed method introduces multiple additional stages into the vector gather instruction pipeline to perform virtual address number coalescing prior to TLB accesses. This method utilizes vector arithmetic units for XOR vector operations, a mask register, and several vector registers, resources commonly available in processors that support vector instructions.

Upon decoding a vector gather instruction, it is placed into the reservation station. Here, the vector gather instruction occupies multiple vector arithmetic units to perform virtual page-address coalescing. Once virtual page-address coalescing is completed or prematurely terminated before the mask is fully zeroed out, the vector gather instruction is forwarded to the load queue, where the TLB is accessed to obtain the physical addresses.

The number of cycles required for page-address coalescing can be expressed as follows:

$$C_{coalescing} = 2 \times T + 2n \times T, \tag{5.1}$$

where $C_{coalescing}$ is the number of cycles for coalescing, n is the number of coalescing trials, and T is the cycle time for a vector arithmetic instruction. The first term of $2 \times T$ represents the cycles required to separate the virtual addresses into virtual page numbers and offsets, which is assumed to be performed using two vector instructions. The second term of $2n \times T$ accounts for the vector XOR instructions and a vector instruction for the mask.

Several processors with vector instructions may possess a chaining feature [1], which bring data to the vector unit responsible for the next vector instruction without waiting for the current vector instruction to complete. This feature may potentially reduce the number of cycles T per vector arithmetic instruction in Eq. (5.1).

5.3.4 Requirements for vector arithmetic unit

The proposed method necessitates vector arithmetic units capable of executing vector XOR between any element and the other elements. Alternatively, this can be achieved through the use of two vector instructions: one to extract a specific element from a vector to a scalar register, and the other to execute a vector XOR operation with the scalar register. Regardless of the implementation, the vector operations utilizing mask registers must be supported.

5.3.5 Trade-off of the proposal

While the proposed page-address coalescing method can alleviate the pressure to TLB, it incurs additional cycles for its own execution. Specifically, the vector gather instruction consumes these cycles as it determines whether virtual addresses can be coalesced or not. Additionally, the proposed method occupies vector arithmetic units for a period of time, potentially causing a structural hazard

as other instructions may be unable to access them, which negatively impacts the performance.

5.4 Evaluations

Section 5.4 evaluates the proposed method by using applications including vector gather instructions.

5.4.1 Experimental setup

This section presents results obtained using a simulator of a vector processor developed based on the gem5 simulator [62], which is a general-purpose architecture simulator. The processor configuration for the vector instruction set is shown in Table 5.1 and is capable of operating on a maximum of 256 elements per instruction. The core specifications are similar to those of the latest vector processor, the SX-Aurora TSUBASA [16, 74, 75]. As the TLB bandwidth of SX-Aurora TSUBASA is not publicly disclosed, the TLB bandwidth has been assumed so as to most closely approximate the performance of the actual machine on the simulator. The evaluation employs the HBM memory model in gem5 to simulate the latency variations. The developed simulator takes instruction trace data as input, which is obtained from the Vector Engine of the SX-Aurora TSUBASA. It then calculates the occupancy of hardware resources within the processor and outputs various performance metrics.

The simulations assume that the TLB does not cause any misses other than compulsory misses. This assumption is made because application designers for data-level parallelism often take measures, such as using larger page sizes and exploiting data locality, to avoid the negative impact of TLB miss penalties. If the TLB miss penalties are substantial, the proposed method may have limited effect as the TLB miss penalty becomes more dominant than the reduced address translation time offered by the proposed method.

Table 5.1: Baseline configuration for the out-of-order vector processor.

Frequency	1.4 GHz
Type	Out-of-order
ROB size	48
Issue queue size	48
Load queue size	64
Store queue size	64
SPU issue width	4 insts./cycle
Vector functional units	3 FMA (8 cycles), 2 ALU (8 cycles), 1 DIV (div 32 cycles, sqrt 64 cycles)
TLB bandwidth	2 addr./cycle
Page size	64 MB
LLC cache size	2 MB
Cache bandwidth	410 GB/s
Memory bandwidth	128 GB/s

Table 5.2: The relationship between the number of coalescing trial and occupying cycles.

number of coalescing trial	required operations	occupying cycles
1	4	16
2	6	24
3	8	32
4	10	40
5	12	48
6	14	56
7	16	64
8	18	72

The case where a dedicated hardware is added to the processor is also discussed in comparison with the proposed method. This case uses the dedicated hardware instead of the vector arithmetic units in the processor. In the case of the dedicated hardware, deduplication itself uses the same procedure as the method proposed in Section 5.3.2.

Table 5.2 shows the relationship between the numbers of trials and cycles

to run the proposed method in this evaluation. The occupying cycles are calculated from Eq. (5.1) where the T is four, under the consideration of the chaining feature in this evaluation.

The proposed method is applicable to both vector gather and vector scatter instructions. However, this evaluation covers only vector gather instructions and not scatter instructions.

5.4.2 Applications

Tables 5.3 and 5.4 list the applications evaluated in this chapter. This study includes two types of applications. The first set consists of three numerical applications that heavily utilize vector gather instructions. The actual B/F value shown represents the bytes per flop ratio obtained through the count of read and write accesses to memory on the simulator.

The second set of applications evaluated in this study consists of four algorithms from the Vector Graph Library [71], with on seven datasets. These highly vectorized and optimized graph algorithms, listed in Table 5.4, are characterized as memory-intensive due to their intense access to complex data structures, resulting in numerous vector gather instructions. The input graph size is the maximum that can be accommodated by the memory of the actual machine, the NEC SX-Aurora VE-20B model with 48GB of capacity[16].

5.4.3 Coalescing trial and performance

Figure 5.4 displays the results of the normalized performance improvement on the numerical applications averaged over all datasets. The horizontal axis shows the number of coalescing trials. The proposed method yields an average 2.2$\times$ performance improvement on the numerical applications as it can coalesce

Table 5.3: The overview of the numerical applications.

Application	Research field	Method	Actual B/F
Turbine [69]	CFD	LU-SGS method	0.34
Plasma [70]	Physics	Lax-Wendroff	0.88
Legendre	Mathematics		6.02

Table 5.4: The algorithms of Vector Graph Library.

Name	Algorithm
bfs	Breadth-First Search [28]
cc	Connected Components [29]
sssp	Single Source Shortest Paths [30]
scc	Strongly Connected Components [30]

all page addresses in vector gather instructions to a single page, resulting in a significant reduction in the number of TLB accesses, because the numerical applications almost exclusively access one page in the case of vector gather instructions.

Figure 5.5 presents the results of the normalized performance improvement on the graph applications averaged over all datasets. The horizontal axis shows the number of coalescing trials. For graph applications, where vector gather instructions frequently access multiple pages, the proposed method yields the best performance when performing page-address coalescing twice. In contrast, attempting page-address coalescing of three or more pages outweighs the advantages of reduced TLB accesses with the disadvantages of the increased number of cycles required for page-address coalescing, leading to a limited performance improvement. On average, the proposed method realizes a 1.08$\times$ performance improvement.

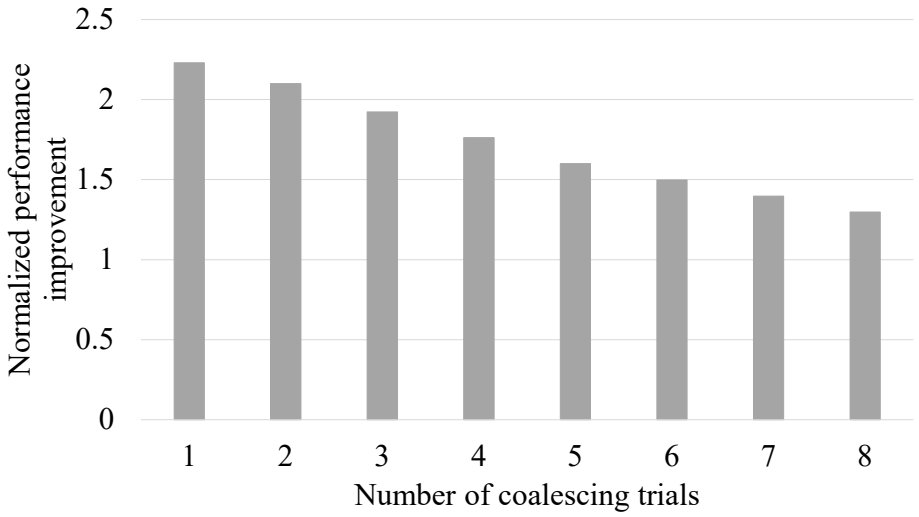


Figure 5.4: The performance results for numerical applications.

5.4.4 Discussion on performance improvement

In Figure 5.6, the performance improvement of each application normalized by the baseline is depicted. The *Dedicated Hardware* indicates the case where the dedicated hardware is added to coalesce page addresses, and the *Proposal* indicates the proposed method that uses the vector arithmetic units in the processor. The number of coalescing trials is one for the numerical applications and two for the graph applications, as previously demonstrated in Figures 5.4 and 5.5. It can be seen that the numerical applications experience a significant performance improvement, which is attributed to their primary reliance on vector floating-point arithmetic units rather than vector integer units. As the proposed method only employs vector integer units, resource conflicts are avoided.

Using dedicated hardware results in an additional 3% performance improvement compared to using the vector arithmetic units in the processor. This is due to the ability of dedicated hardware to avoid hardware conflict.

It is worth noting that the proposed method does not enhance the performance of the Breadth-First Search (BFS) application. In BFS, vector gather

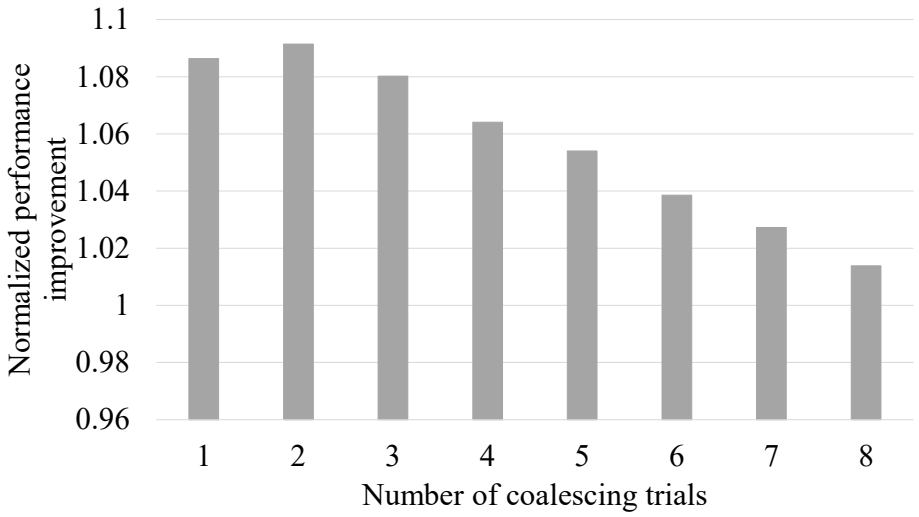


Figure 5.5: The performance results for graph applications.

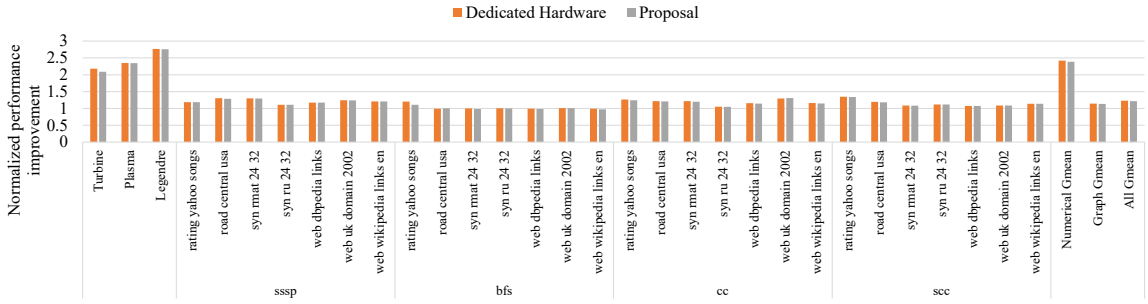


Figure 5.6: The performance improvement for each application.

instructions accesses are utilized when the vector length is short, and although two or more pages are required as depicted in Figure 5.1, TLB bandwidth does not become a performance bottleneck in this application.

5.4.5 TLB access reduction

To further understand the performance improvements achieved by the proposed method, this section examines the reduction in the number of TLB accesses. The numerical applications can be coalesced almost all addresses of vector gather instructions onto a single page. Figure 5.7 shows the average number of addresses

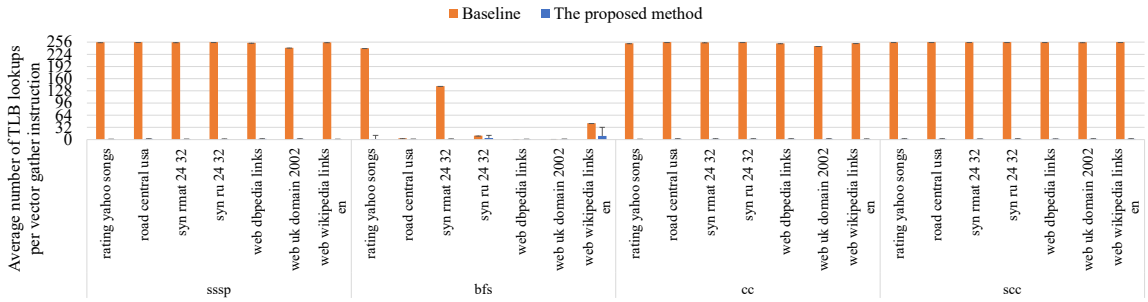


Figure 5.7: The number of addresses translated by TLB.

of vector gather instructions that needed to be translated by the TLB. In contrast, the graph applications has coalescing of addresses onto multiple pages. As shown in Figure 5.7, the average number of TLB accesses is reduced for all applications except BFS. The proposed method has a minimal effect on the performance of BFS because vector gather instructions are used with short vector lengths.

5.5 Conclusions

As vector instruction sets become more prevalent in processors, their applications are expected to benefit from the high vector processing capabilities of these processors. However, some of these applications involve irregular memory accesses in vector gather instructions, which can hinder performance improvements. The proposed method of page-address coalescing using vector arithmetic units has been demonstrated to be effective in improving the performance of applications employing vector instruction sets, particularly those that require irregular memory accesses through vector gather instructions. In numerical applications, the proposed method has achieved an average performance improvement of 2x, while in graph applications, an average improvement of 1.08x is obtained. These results highlight the usefulness of the proposed method in addressing the bottleneck caused by TLB accesses in vector gather instructions and enabling acceleration of vector instruction-based applications.

By using the proposed method, overlap of virtual addresses is eliminated and the number of addresses requiring address translation is reduced. This reduces the latency required for address translation and contributes to latency reduction.

Chapter 6

Skewed multi-banked cache for many-core vector processors

6.1 Introduction

As the number of vector cores increases, the off-chip memory can become a bottleneck for memory-intensive applications. To solve this issue, the efficient utilization of the shared cache becomes crucial for vector processors, despite of their relatively high memory bandwidth. By allowing the vector cores to reuse data stored in the shared cache, the need for unnecessary off-chip memory accesses can be reduced. As a result, it is important for applications running on vector processors to be optimized to take advantage of the shared cache as much as possible. If such optimization is successful, an increase in the number of vector cores with certain optimization to share data on the shared cache will result in a higher number of cache hits and reduced pressure on off-chip memory.

This chapter conducts a preliminary evaluation of the impact of shared cache

organizations on a many-core vector processor. By applying a simple optimization to an application, this chapter aims to increase the amount of data shared as the number of vector cores increases. As a result, if the number of vector cores sharing the cache increases, an increase in the cache hit rate can be expected. However, the preliminary evaluation reveals that a cache shared by many vector cores experiences a high number of conflict misses. While increasing the cache associativity is a common approach to reduce conflict misses, it also incurs a significant overhead for a multi-banked cache.

In this chapter, a skewed cache design is proposed for many-core vector processors. The skewed cache employs skewed associativity [76], which helps to reduce the occurrence of conflict misses by avoiding simultaneous data requests from multiple vector cores to use the same cache set. Two features of the skewed cache are also examined, including hashing functions and replacement policies. The proposed design utilizes odd-multiplier displacement hashing for effective skewing and the static re-reference interval prediction policy for reasonable replacement. The performance of the skewed cache is evaluated using a stencil calculation kernel with varying the number of vector cores sharing the cache and its associativity. The results show that the proposed cache can effectively eliminate conflict misses and achieve almost ideal hit rates in shared cache configurations. By improving the cache hit ratio, the vector processor can achieve a performance improvement.

6.2 Motivation

This section first presents the organization of the many-core vector processor that is considered in this study. The impact of conflict misses on the hit rates of shared caches is then preliminarily evaluated using various shared cache configurations of the many-core vector processor.

6.2.1 Many-core Vector Processors with Multi-banked Shared Cache

As vector cores become more powerful, there is an increasing demand for improved memory performance to support the data needs of these cores. However, the improvement of memory performance lags behind that of computing capability. To bridge this gap, cache memory becomes essential, and is thus implemented in modern vector processors. For instance, the NEC SX series of vector processors, starting with the SX-9, includes an on-chip multi-banked cache memory [18, 2]. This section assumes that the many-core vector processor also includes such a multi-banked cache memory.

In the future, it is expected that as the number of vector cores increases, several cores will be connected to one cache. Figure 6.1 illustrates an example in which each cache is shared by M vector cores when the total number of vector cores is N. The relationship between M and N can be expressed as follows:

$$C = \frac{N}{M}, \tag{6.1}$$

where C is the number of caches, and M is the number of vector cores connected to one cache. Equation (6.1) demonstrates that the number of caches C depends on the values of M and N. Thus, the number of cache banks in a cache, b, can be

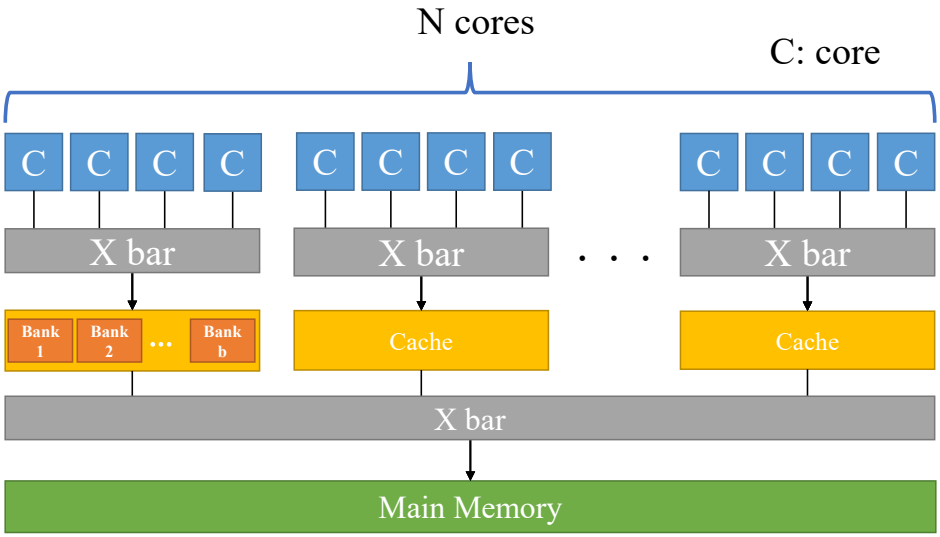


Figure 6.1: Example of M cores sharing the same cache in the N vector cores processor

calculated as follows:

$$b = \frac{B}{C} = \frac{BM}{N}, \tag{6.2}$$

where B is the total number of banks in a many-core vector processor.

In this section, the total number of vector cores, N, and the total number of banks in a many-core vector processor, B, are held constant as various configurations of the shared cache are analyzed under the same computing capability and memory performance. As a result, only the number of vector cores connected to the cache, M, influences the number of banks per cache, b, according to Equation (6.2). It should also be noted that the capacity of each bank is fixed, and thus the number of banks per cache determines the overall cache capacity.

6.2.2 Conflict Misses on The Many-core Vector Processor

6.2.2.1 3D 7-point Stencil Calculation

This section investigates various configurations of the shared cache using the 3D 7-point stencil kernel, a representative memory-intensive computing kernel. Stencil computations, which include this kernel, play a significant role in scientific and engineering simulation codes.

Algorithm 5 provides a pseudo-code for the 3D 7-point stencil kernel, which computes the arithmetic mean of a central element and its six neighboring elements along the x, y, and z axes. The calculation is repeated for all elements in a 3D space. Thus, for each iteration of the innermost loop of Algorithm 5, a total of seven elements are needed.

Figure 6.2 illustrates the elements used for one calculation in Algorithm 5, with translucent elements representing the calculation space and solid elements representing the seven elements used in the calculation.

6.2.2.2 Stencil Calculation with A Shared Cache

This section assumes that the outermost loop regarding the z-axis in Figure 5 is parallelized. As a result, each iteration of the z-axis loop is assigned to a different core in a cyclical manner. Specifically, the k-th iteration of the outermost loop is calculated by the core with ID $(k \bmod N)$.

During the stencil calculation, each vector core accesses the central element and its neighboring elements. If these elements are already being accessed by other vector cores, they may have been placed in the cache, allowing them to be reused and reducing pressure on the off-chip memory.

Using the described parallelization, the theoretical cache hit rate of the 3D 7-point stencil calculation can be calculated. It is important to note that a single

Algorithm 5 3D 7-point stencil calculation

```
1: for z = 1, ...N_z − 1 do
2:     for y = 1, ...N_y − 1 do
3:         for x = 1, ...N_x − 1 do
4:             b[z][y][x]=(a[z][y][x]+
5:                     a[z][y][x-1]+a[z][y][x+1]+
6:                     a[z][y-1][x]+a[z][y+1][x]+
7:                     a[z-1][y][x]+a[z+1][y][x] )/7
8:         end for
9:     end for
10: end for
```

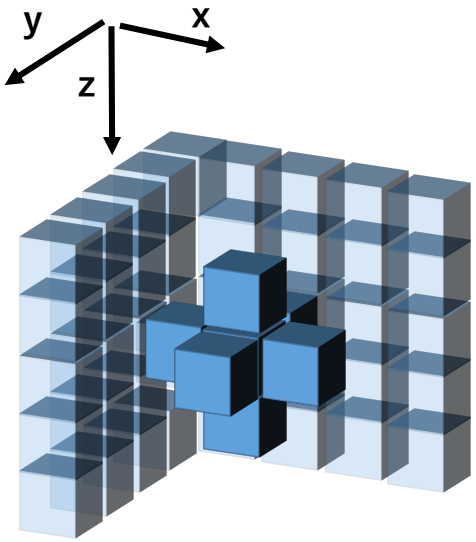


Figure 6.2: The elements used in one calculation

layer of elements in the x-y plane does not fit in the cache, but a single line of elements along the y-axis does. The theoretical cache hit rate is first calculated in the case where the vector cores do not share the cache at all. Figure 6.3 illustrates which elements will hit the cache. The translucent elements represent the group of elements calculated by one core, while the solid blue element in Figure 6.3 cause cache misses regardless of whether the cache is shared. The red elements: a$[z][y][x]$, a$[z][y][x$-1$]$, a$[z][y][x+1]$, and a$[z][y$-1$][x]$ should hit because the last iteration brings them into the cache regardless of whether the cache is shared or not. The solid purple and green elements cannot be considered hits

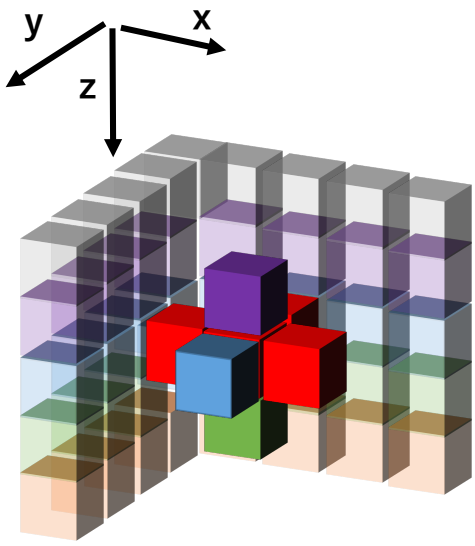


Figure 6.3: The elements shared on the cache

because a single layer of the x-y plane does not fit in the cache. As a result, in the case where vector cores do not share a cache, the theoretical hit rate becomes $4/7$ $(57.14\,\%)$.

The theoretical cache hit rate can be calculated for the case where vector cores share the cache with their neighboring cores. The purple and green elements can be retrieved from the cache because the neighboring cores also bring these elements into the cache. This means that a$[z\text{-}1][y][x]$ and a$[z{+}1][y][x]$ can also be retrieved from the cache. However, cores with numbers 0 or $M-1$ only have one neighboring core, so either a$[z\text{-}1][y][x]$ or a$[z{+}1][y][x]$ will always be missed for these cores. Based on this information, it can be determined that at most $2/7$ $(28.57\,\%)$ of elements in the 3D 7-point stencil calculation can be shared by the shared cache.

When M iterations of the outermost loop are calculated in parallel, there are $7 \times M$ accesses by M cores. If the first core calculates loop 1, the second core calculates loop 2, and so on, with the M-th core calculating loop M, the cores in charge of loop 1 and loop M can only reuse 5 elements from the shared cache,

while the other cores can reuse 6 elements. Based on this, the number of hits in one iteration for all cores can be calculated as $6(M-2)+5\times 2$. Therefore, the theoretical hit rate of the 3D 7-point stencil calculation can be expressed as follows:

$$H_7 = \frac{6(M-2)+5\times 2}{7M} = \frac{6}{7} - \frac{2}{7M}. \tag{6.3}$$

Equation (6.3) can predict the hit rate of the stencil calculation theoretically based on the number of cores sharing a single cache, M. It can be observed that as M increases, the cache hit rate monotonically increases, eventually asymptotically approaching $6/7$ $(85.71\,\%)$.

Overall, an increase in the number of vector cores sharing a cache allows more vector cores to reuse the data stored in that cache. This can result in an increase in the theoretical cache hit rate for stencil calculations.

6.2.3 Preliminary Evaluation

Equation (6.3) expresses the upper-bound of the cache hit rate because this equation only considers the reusability of the elements, and other effects are ignored. In order to confirm the model defined by Equation (6.3), a preliminary evaluation is conducted to investigate the hit rate in the many-core vector processor by varying the number of cores that share a cache. The stencil calculation is parallelized to share the data, as discussed in Section 6.2.2.2. Thus, the larger number of cores suggests the higher cache hit rate, as expected from Equation (6.3). Under this situation, various configurations of a many-core vector processor are evaluated. The number of cores sharing a cache is set to 1, 2, 4, 8, 16, and 32, and the cache associativity is set to 4, 8, 16, and 32. The other

configurations correspond to the configuration described in Section 6.4.1.

The results of the preliminary evaluation are depicted in Figure 6.4, where the vertical axis represents the cache hit rate and the horizontal axis indicates the number of cores sharing a single cache and the associativity. As the number of cores sharing a cache increases, the theoretical cache hit rate also increases, as shown in the figure. However, the cache hit rate of the set-associative cache with the LRU replacement policy becomes significantly low when there is a large number of cores sharing a single cache and low associativity. This is due to conflict misses occurring when multiple cores simultaneously access the same set during the stencil calculation. To effectively share data among vector cores using a shared cache, it is necessary to reduce the number of conflict misses. It is also important that the cache mechanism can achieve a high hit rate in order to realize latency tolerance for vector processors.

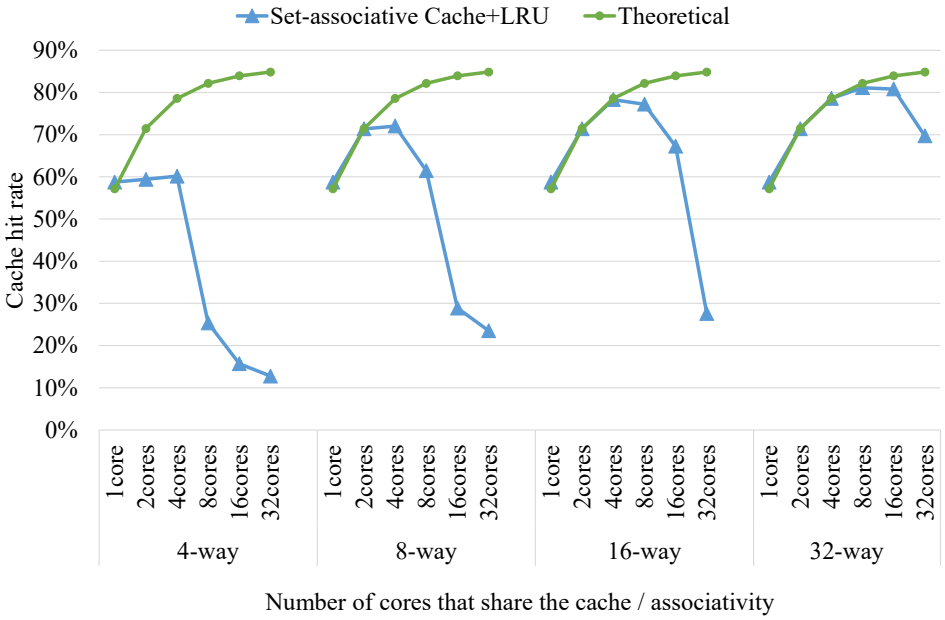


Figure 6.4: The cache hit rate when the number of cores sharing the same cache and the number of the associativity are changed

6.3 Skewed Multi-banked Cache for Many-core Vector Processors

Skewed-associativity [76] is an effective method of eliminating conflict misses without increasing associativity. This chapter proposes a skewed cache for many-core vector processors in order to reduce the number of conflict misses on vector load/store data in a shared cache. A skewed cache eliminates conflict misses by allocating blocks to sets using a hashing function for each way. Figures 6.5 and 6.6 demonstrate the difference between a 2-way set-associative cache and a 2-way skewed cache.

In Figure 6.5, the set-associative cache allocates addresses D1, D2, and D3 to the same set. As a result, if the blocks of these addresses are stored in this order, the block of D1 inserted first will be evicted when the block of D3 is inserted, leading to a conflict miss. In contrast, in Figure 6.6, the hashing function f_0 for way 0 and the hashing function f_1 for way 1 independently generate different set indices, resulting in these addresses being indexed to different sets. This means that the block of D3 is stored in a different set from that of D1, allowing the block of D1 to be avoided and resulting in no conflict miss.

To achieve a high cache hit rate on the skewed cache, the hashing function and replacement policy are crucial. The hashing function plays a key role in the skewed cache's ability to avoid set conflicts, and therefore, it should output non-biased values to prevent blocks from being placed in the same set and causing conflict misses. It is also desirable to be able to easily create various hashing functions based on a single rule to produce different outputs for each way.

The replacement policy is also important. One challenge with the skewed cache is that it is difficult to implement the Least Recently Used (LRU) policy

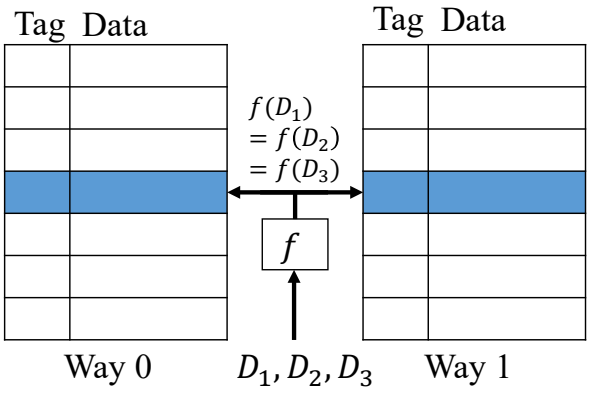


Figure 6.5: The 2-way set-associative.

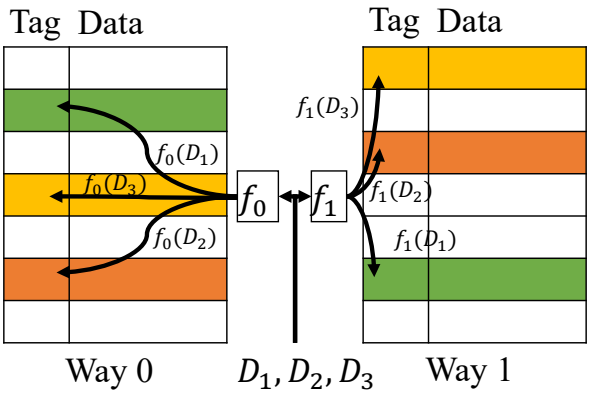


Figure 6.6: The 2-way skewed-associative.

with a higher associativity of three or more. This is because LRU generally determines which block to evict based on the insertion order of blocks in the set. In the case of the set-associative cache, replacement candidates are always chosen from the same set, and the evicted block is selected from among them. Therefore, it is necessary to maintain the insertion order within the sets. However, the skewed cache selects replacement candidates from different sets based on the address of the block and the hashing function. As a result, the insertion order within the sets cannot determine whether the blocks were recently used or not. If absolute timestamps are added to every block, it is possible to implement LRU for the skewed cache, but this would require an impractical hardware cost.

6.3.1 Hashing Functions

6.3.1.1 XOR-based Hashing Function

A XOR-based hashing function is used when proposing skewed-associativity [76, 77, 78]. The following equation calculates the set index:

$$index = \sigma^w(A_2) \oplus A_1 \bmod n_{set}, \tag{6.4}$$

where A_1 and A_2 are fields of an address as shown in Figure 6.7. The bit lengths of A_1 and A_2 are $\log_2(n_{set})$, and σ^w represents a w-bit circular shift operation, where w generally represents the way ID for each way. The $\oplus$ symbol denotes the bit-wise exclusive OR operator. For instance, the hashing function for way 0 is $A_2 \oplus A_1 \bmod n_{set}$, and that for way 1 is $\sigma^1(A_2) \oplus A_1 \bmod n_{set}$.

The XOR-based hashing function has the advantage of simplicity in implementation and the ability to satisfy the requirements for skewing. It spreads blocks that may be mapped to the same set in one way across other sets of the other ways, and it prevents two blocks with the same higher bits (A_2 or tag field in Figure 6.7) from being mapped to the same set.

However, the XOR-based hashing function is known to suffer from certain stride patterns called the pathological behavior [79]. For example, if n_{set} is 16, A_1 is fixed, and A_2 varies as a stride of 8, the XOR-based hashing function will generate the sequence of set indices 1,9,1,9,...,1,9. This problem also occurs even when w changes. Furthermore, if either A_1 or A_2 is 0 or 1111...111, the outputs always become the same value regardless of w. Therefore, this dissertation examines another hashing function for the skewed cache.

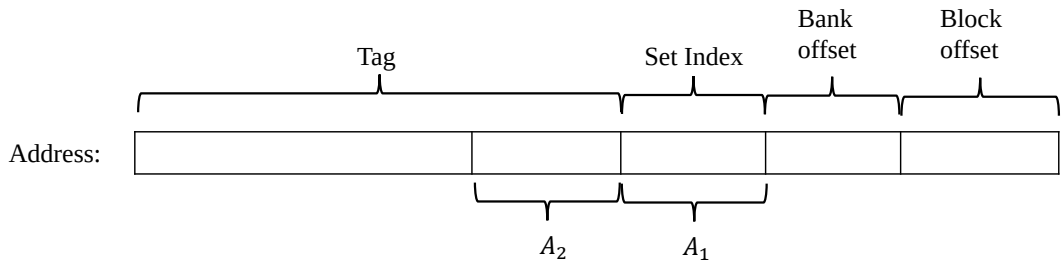


Figure 6.7: The bit field of hashing function of the given address

6.3.1.2 Odd-multiplier Displacement Hashing Function

This dissertation investigates the Odd-Multiplier Displacement Hashing Function (oDisp) [79] as an alternative to the XOR hashing function. The oDisp has been shown to be a more uniform hashing function than those based on XOR. It is expected to result in fewer conflict misses, even in the cases where access patterns include strides of a particular length. The set index can be expressed as follows:

$$index = (o \times A_2 + A_1) \bmod n_{set}, \tag{6.5}$$

where o is an odd number chosen specifically for each way.

In addition, the oDisp has a low hardware cost. According to Equation (6.5), the set index is obtained by adding an arbitrary odd number multiplied by A_2 to A_1 and taking the remainder with respect to the number of sets. This multiplication can be implemented using a single logical shifter and two adders, rather than a dedicated multiplier. Moreover, it is easy to vary the output of the hashing function for each way by using different odd numbers. Therefore, the index of the oDisp can be simplified as follows:

$$index = ((A_2 \ll w) + A_2 + A_1) \bmod n_{set}, \tag{6.6}$$

where w is the way number and $\ll$ denotes a logical left-shift operation.

6.3.2 Replacement Policies

6.3.2.1 Not Recently Used Not Recently Written

There have been several investigations on replacement policies for skewed caches. Seznec *et al.* [52] proposed the Not Recently Used Not Recently Written policy (NRUNRW), which is based on the Not Recently Used (NRU) approach. The flow of NRUNRW is depicted in Algorithm 6. This policy uses a single bit per cache block, referred to as the Recently Used bit (RU), to identify a non-recently used cache block for eviction. If no such block can be identified based on the RUs of the candidate blocks, the policy will select a clean block among the candidates. The RUs of all cache blocks are reset at intervals when the number of cache requests reaches one-fourth of the total number of cache blocks.

While NRUNRW has the advantage of requiring minimal hardware resources, Seznec [78] found that its performance falls short of that of the Least Recently Used (LRU) policy due to the randomness introduced by the RU resetting. Thus, this dissertation will also examine an alternative replacement policy for skewed caches.

6.3.2.2 Static Re-Reference Interval Prediction

To achieve a hit rate comparable to that of the Least Recently Used (LRU) policy at a practical hardware cost, the proposed skewed cache employs the Static Re-Reference Interval Prediction (SRRIP) policy [80] as its replacement strategy.

SRRIP, which is an extension of the Not Recently Used (NRU) policy, is able to deliver hit rates similar to those of LRU with a simple algorithm and low hardware requirements. The flow of SRRIP is illustrated in Algorithm 7. This

Algorithm 6 The flow of NRUNRW policy.

```
Require: Candidates
Ensure: An evicted block
 1: if counter > (the number of blocks / 4) then
 2:     Reset RU of all blocks
 3:     Set counter 0
 4: else
 5:     Increment counter
 6: end if
 7: if Cache hits then
 8:     Set RU of the hit block to 1
 9: else if Cache misses then
10:     g1 :=Candidates.where(RU is 0)
11:     g2 :=Candidates.where(RU is 1 and clean)
12:     g3 :=Candidates.where(RU is 1 and dirty)
13:     if g1 is not empty then
14:         Return one randomly from g1
15:     else if g2 is not empty then
16:         Return one randomly from g2
17:     else if g3 is not empty then
18:         Return one randomly from g3
19:     end if
20: end if
```

policy assigns an m-bit Re-Reference Prediction Value (RRPV) to each cache block, which is used to predict when the block will be re-referenced. SRRIP takes advantage of the higher reusability of recently hit blocks compared to newly inserted blocks, allowing them to remain in the cache while other blocks are replaced. As a result, only recently hit blocks tend to be retained in the cache.

SRRIP is particularly suitable for use in skewed caches, as the RRPVs of each block can be used to predict the re-reference interval of a block independently of the RRPVs of other blocks in the same set. This enables the selection of an evicted block even when replacement candidates come from different sets, as is the case in skewed caches.

Algorithm 7 The flow of SRRIP policy

Require: Candidates
Ensure: An evicted block

```
if Cache hits then
    Set RRPV of the hit block to 0
else if Cache misses then
    while True do
        for c in Candidates do
            if c.RRPV is 3 then
                Return c
            end if
        end for
        Increment RRPVs of all candidates
    end while
end if
```

6.4 Evaluation

6.4.1 Experimental Environment

To assess the impact of the skewed cache on the performance of many-core vector processors, this section conducts experiments using a simulator developed based on the gem5 simulator [59], which is a general-purpose architecture simulator. The simulator takes as input an instruction trace data obtained from the SX-ACE vector supercomputer, and uses it to simulate the utilization of hardware resources within the processor and calculate various performance metrics.

The simulation of the many-core vector processor is implemented using pseudo cores, which only issue requests to the memory system. This allows for the efficient simulation of many-core vector processors. In the stencil calculation targeted in this study, it is assumed that parallelization occurs in the outermost loop of Algorithm 5 discussed in Section 6.2.2.2. Therefore, the widths specified for the pseudo cores correspond to the amount of data processed in a single iteration along the z-axis. The calculation space is set to 2048x2048x512.

Table 6.1 presents the system configurations for the many-core vector processor. The total number of cores is set to 32, based on trends in the development of many-core vector processors, and the configuration of each core is based on that of the NEC SX-ACE [18]. The Bytes/Flop (B/F) value of the system used for evaluation is set to 0.125, reflecting the increasing gap between computing capability and memory performance and the corresponding trend towards lower B/F values in newer generation vector processors (e.g. SX-ACE and SX-Aurora TSUBASA have B/F values of 1.0 and 0.5, respectively).

Several parameters influence the performance of the multi-banked cache memory. The associativity is varied between 4, 8, 16, and 32, and the number of

Table 6.1: Configurations of the simulation

Base architecture	NEC SX-ACE
Total number of core	32
Number of cores sharing a same cache	1, 2, 4, 8, 16, 32
Main memory bandwidth	256GB/s
Total cache size (size per bank)	32MB (128KB)
Associativity	4, 8, 16, 32
Cache block size	128Bytes
MSHR (Target) [81]	8 (8)
System B/F	0.125 B/F

cores sharing a single cache is set to 1, 2, 4, 8, 16, and 32. The total capacity and the total number of banks are both fixed at 32MB and 256 banks, respectively. The total capacity is kept constant to focus on the effect of the shared cache, while the total number of banks is held constant to maintain equal bandwidth across all configurations.

In addition to the proposed skewed cache, this study also evaluates the performance of a conventional set-associative cache for comparison. The Not Recently Used Not Recently Written (NRUNRW), Least Recently Used (LRU), and Static Re-Reference Interval Prediction (SRRIP) policies are used as replacement policies for the skewed cache. In the evaluation, SRRIP is implemented using two bits for the Re-Reference Prediction Value (RRPV), which has been shown to achieve performance comparable to that of LRU [80]. Since it is challenging to implement LRU in a realistic hardware cost for skewed caches, LRU is implemented in this study by judging the insertion order of blocks based on the absolute timestamps assigned to them.

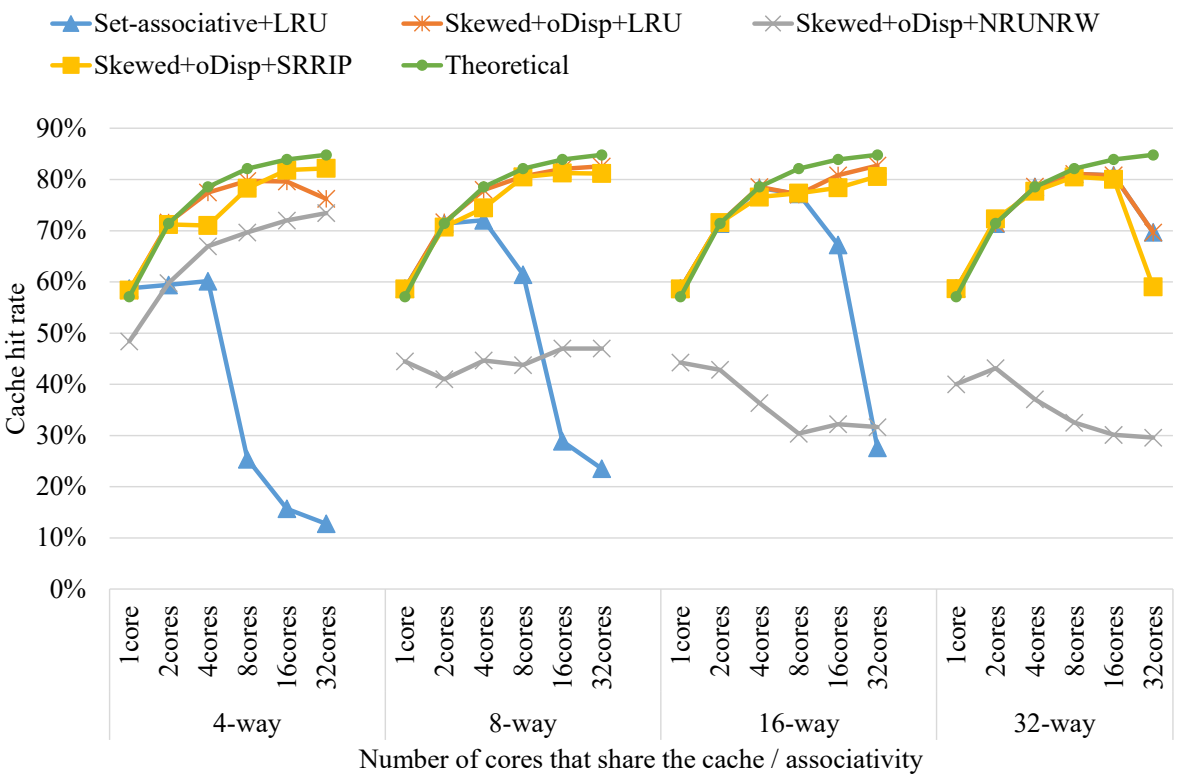


Figure 6.8: Cache hit rate result with the same hashing function, oDisp, except set-associative cache

6.4.2 Evaluation Results and Discussion

6.4.2.1 Cache Hit Rate

Figures 6.8 and 6.9 show the cache hit rates for the conventional cache and the skewed cache with various replacement policies and hashing functions. In these figures, the vertical axis represents the cache hit rate, and the horizontal axis represents the number of cores sharing a cache and the associativity of the cache.

Figure 6.8 presents the cache hit rates for different replacement policies. To better compare the performance of these policies, the odd-multiplier displacement hashing function is used. The hit rates of the proposed skewed cache with

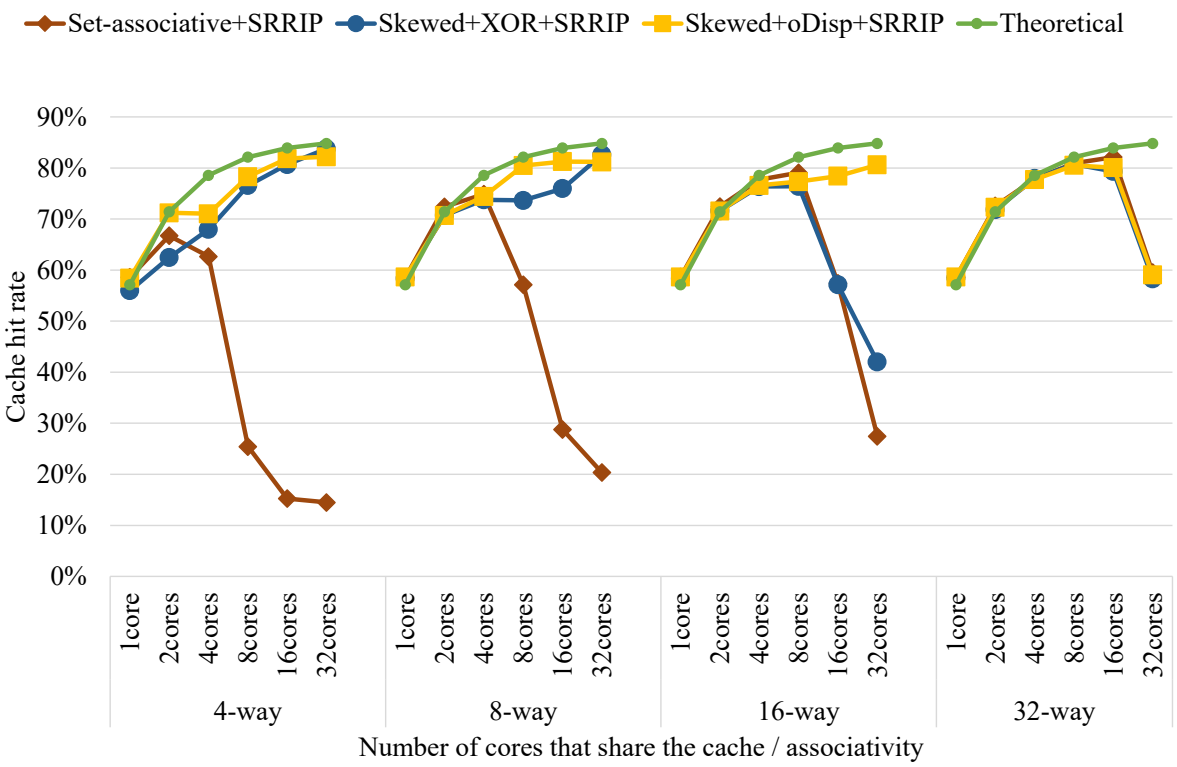


Figure 6.9: Cache hit rate result with the same replacement policy, SRRIP

SRRIP are very close to the theoretical hit rates for each configuration. SRRIP closely follows the theoretical values only when the associativity is 4-way or 8-way, although the hit rates for higher associativities are slightly lower than those of LRU. When the cache is 4-way or 8-way, the number of hits for the skewed cache with SRRIP is improved by up to 70% and 60% compared to the set-associative cache, respectively. In the case where 32 cores share a 16-way cache, the number of hits increases by approximately 50%.

In contrast, the cache hit rate significantly decreases for low associativity in the set-associative cache. In particular, when the associativity is 4-way, 8-way, or even 16-way and the number of cores sharing the cache is high, the difference between the set-associative cache and the theoretical hit rates becomes noticeable. This is due to the relatively regular memory access patterns in the stencil calculation. In the evaluation, the stencil calculation is parallelized as described

in Section 6.2.2.2. Each core sends memory access requests for the z-iteration assigned to it. However, differences in these requests can be primarily distinguished by the tag field of their addresses, leading to requests being indexed to the same set in the set-associative cache and resulting in conflict misses. On the other hand, the skewed cache can avoid conflict misses because it uses a larger range of an address field, including part of the tag field, allowing it to distinguish the difference in the z-iteration on the address to determine the set.

As a replacement policy for the skewed cache, the hit rate of LRU is almost equal to the theoretical value. When LRU is applied, the skewed cache can maintain a higher hit rate than the set-associative cache or a high hit rate equal to the skewed cache with SRRIP. This is because the skewed cache with LRU is based on timestamps, allowing blocks to be replaced optimally. Although its implementation requires significant hardware costs, it is possible to eliminate more conflict misses than SRRIP regardless of the associativity. It is also worth noting that NRUNRW has a low hit rate in all the cases. This is because all RUs are reset at each fixed access interval, causing randomly selected evictions every interval and leading to necessary blocks suffering from random eviction.

When the number of cores that share a single cache exceeds 16 and the associativity is set to 32-way, both SRRIP and LRU exhibit a decrease in cache hit rate. This can be attributed to two factors. First, a high associativity reduces the number of sets per way, increasing the likelihood of conflict misses by repeatedly selecting the same set. Second, the implementation of the oDisp hashing function in the skewed cache, as shown in Equation (6.6), may also contribute to the decrease in hit rate for larger values of the number of ways, as the calculation of the index for these larger values is not affected by the shifted A_2 when it exceeds the bitfield length of A_2. It is worth noting that the miss rates

for SRRIP are slightly lower than those of LRU, because SRRIP is approaching a more random replacement strategy, and resulting in an increase in misses when encountering numerous replacement candidates that have the same RRPVs at high associativity.

As depicted in Figure 6.9, the hit ratio of the skewed cache with varying hashing functions is evaluated, using SRRIP as the replacement policy for both the skewed cache and the set-associative cache. The results show that both hashing functions generally achieve high cache hit rates, but the XOR-based hashing function exhibits a decline in hit rate for extremely high associativity and large numbers of cores sharing a single cache. This decline can be attributed to two factors. First, the skewed cache with the XOR-based hashing function is prone to a specific access pattern known as pathological behavior, as mentioned in Section 6.3.1.1, which is suspected to contribute to the observed degradation in the hit rate, particularly in the cases of 16-way caches shared by 16 or 32 cores. Second, the implementation of the XOR-based hashing function may also contribute to the decline in the hit rate for larger values of the number of ways, as the bit rotation used in the function leads to the recalculation of the same index for these larger values. In contrast, the oDisp hashing function is able to handle such access patterns and maintain a stable, high hit rate. It is worth noting that the set-associative cache with SRRIP experiences a similar decline in the hit rate as the set-associative cache with LRU, as shown in Figure 6.8, due to the same underlying cause.

6.4.2.2 Performance

Figure 6.10 compares the performance of the set-associative cache based on LRU with the proposed skewed cache using oDisp and SRRIP, which has been shown

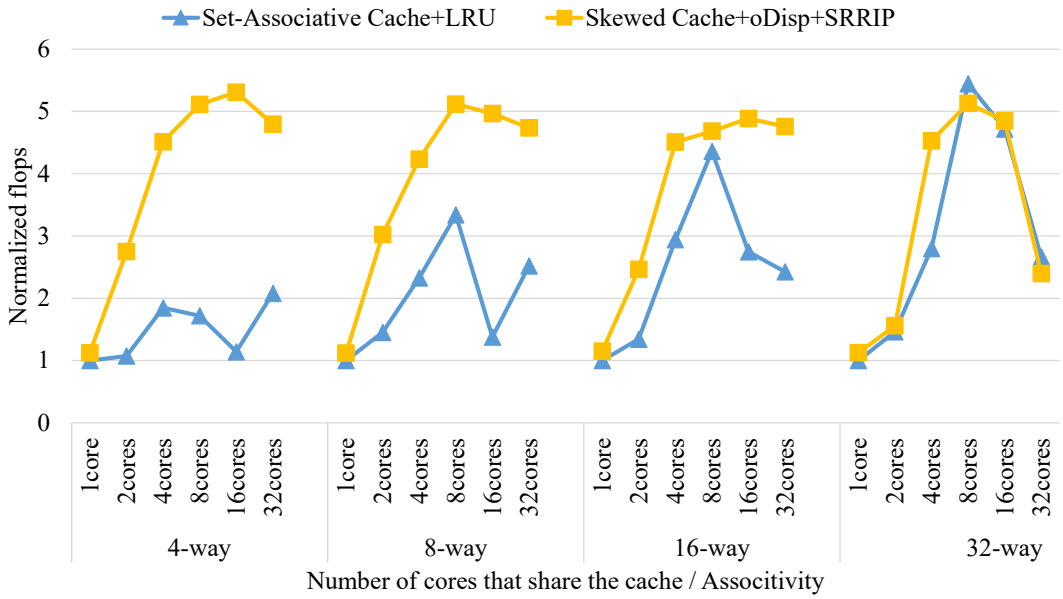


Figure 6.10: Performance comparison of the conventional set-associative cache and the proposed skewed cache

to achieve a high hit rate with a reasonable implementation cost. The vertical axis in the figure represents the performance normalized to the case in which each core has a private set-associative cache, and the horizontal axis denotes the number of cores sharing a single cache and the associativity.

First of all, the skewed cache demonstrates superior performance compared to the conventional set-associative cache, particularly when the associativity is low. This is due to the skewed cache's ability to eliminate conflict misses effectively. It has been shown that SRRIP can achieve nearly ideal cache hit rates in the skewed cache at reasonable implementation costs. Additionally, the oDisp hashing function enables the skewed cache to avoid performance degradation caused by conflict misses.

As seen in Figure 6.10, the proposed cache outperforms the set-associative cache when the associativity is low. Specifically, in the case of 16 cores sharing a 4-way cache, the skewed cache exhibits a six-fold improvement in performance over the set-associative cache. However, when the associativity is sufficient,

there is little difference in performance between the skewed and set-associative caches, as both are able to effectively reduce conflict misses.

In Figure 6.10, there are instances where the performance of a cache shared by 16 cores is higher than that of a cache shared by 32 cores, despite the latter having a higher cache hit rate. This can be attributed to cache bank conflicts. The multi-banked configuration of the many-core vector processor used in this study causes memory access requests to be directed to specific cache banks based on the memory address. Therefore, when multiple cores share a single cache, the likelihood of accessing the same cache bank increases. Additionally, the write buffer size and MSHR per bank in this configuration are 16 and 8, respectively. As a result, when 32 cores share a cache, their requests may become concentrated on certain banks due to conflicts, which leads to shortages in the write buffer or MSHR, resulting in performance degradation.

6.5 Conclusions

This chapter presents a skewed multi-banked cache for many-core vector processors. The skewed cache can prevent concurrently requested blocks from occupying the same cache set by using a hashed value of the block address to determine the appropriate cache set for storing the block. The chapter discusses the implementation of two key features of the skewed cache: the hashing functions and the replacement policies. Three replacement policies (LRU, NRUNRW, and SRRIP) and two hashing functions (XOR-based and oDisp) have been evaluated for use in the skewed cache.

The evaluation results show that the skewed cache with SRRIP and oDisp can increase the hit rate by up to 70% and produces results that are closest to the theoretical upper bound of the shared cache hit rate. From the separate evaluations of the hashing functions and replacement policies, it is found that SRRIP achieves the highest hit rate with low hardware overhead, and oDisp addresses the issues with the XOR-based hashing function. The evaluation results also reveal that the skewed cache can achieve a six-fold improvement in performance over the conventional set-associative cache for stencil calculations.

By the mechanism proposed in this chapter, the cache can supply data to the vector processor with a short latency by increasing the hit rate. This enables the processor to be the latency tolerance.

Chapter 7

Conclusions

Processor performance can be classified into two types: computing performance and memory performance. The computing capability refers to the number of calculations per second that a processor execute. Thanks to advancements in semiconductor technologies, the computing performance has steadily increased over the previous decades. However, due to the end of Dennard's scaling and the stagnation of clock frequency, processors are required to improve computing performance while reducing power consumption.

Vector instruction set is one of the promising solutions to satisfy these requirements. Vector instruction set enables the processors to handle multiple elements by one instruction. Since a compiler guarantees that the elements in a vector are independent, the processors can achieve data-level parallelism by processing the data in parallel by hardware.

Modern vector processors are designed to handle vector instructions, enabling to achieve high sustained performance, particularly in HPC applications. Memory-intensive applications are well-suited to vector processors in the fields of science and engineering, which rely on long vector lengths and high memory

bandwidth to improve computational capabilities. As these applications continue to evolve to satisfy the demands of improved accuracy and broader applicability, the performance requirements for the vector processors are also rising.

In addition, new workloads such as graph processing and machine learning have emerged as commonplace applications. These new applications are memory-intensive because they employ complicated data access patterns to solve advanced algorithms. Because of this, the memory system has easily become a limiting factor for these applications. High sustained performance for these applications has been desired for vector processors. However, the high memory bandwidth of vector processors may be underutilized in some numerical simulations and new applications due to latency, i.e., the time between issuing a vector instruction and its completion.

Vector processors have traditionally not considered latency issues because their vector processing mechanism allows them to hide latency through pipelined data accesses. However, the latency has become a performance bottleneck due to the following three reasons. First, as semiconductor manufacturing technologies improve computing capabilities, the time required for each vector operation tends to decrease, diminishing the latency concealment capability of vector processors. Second, the number of instructions that the processor can handle simultaneously may be inadequate for handling certain memory access patterns. Vector instructions that handle irregular memory accesses often require multiple instructions for a single access, leading to longer latencies than sequential accesses. Finally, applications with irregular memory access patterns may not utilize the cache system, as the data may not have locality and therefore cannot be stored in the cache. These factors have led to a need for architectures that make vector processors more tolerant of latency.

In Chapter 2, the importance of latency tolerance for vector processors is examined. The modern vector processors assumed in this dissertation are first introduced. Second, this chapter defines the latency that this dissertation addresses. In order to realize high sustained performance, this dissertation focuses on the latency of modern vector processors. "Latency" is defined as the time between the fetch of a vector instruction and the completion of the vector instruction. This chapter divides latency-related problems into four categories. First, the latency due to dependencies among vector memory instructions caused by indirect memory accesses is a factor that can degrade performance even with sufficient bandwidth in the vector processor. Second, the latency that can be hidden by the out-of-order execution mechanism is limited for applications with a large number of instructions per loop, such as commonly run on the vector processor. Third, the latency caused by the address translation required for every memory access in the vector processor with virtual memory can be a bottleneck for vector memory instructions. Finally, the memory access latency is closely related to the cache hit ratio for the vector processor.

To solve the problems related to these latencies, this dissertation examines two perspectives. The first perspective involves exploring various ways for mitigating the latencies, including both approaches to hiding latency and reducing latency. From this perspective, one approach to hiding latency can be achieved through out-of-order and speculative execution. Another approach to reducing latency involves implementing features or adopting procedures that minimize latency.

The second perspective to solving latency-related problems involves taking

into account both the core and memory architectures of modern vector processors. From this perspective, there are two approaches to mitigating latency-related problems. The first approach is to design the processor to handle latencies through out-of-order execution and virtual memory. The second approach involves improving the memory architecture, including the design of the cache system and a prefetching mechanism to solve latency-related problems.

These approaches from the two perspectives are discussed in the subsequent chapters. Finally, Chapter 2 discusses previous work related to the discussion and approaches in this dissertation.

Chapter 3 proposes an indirect memory access prefetcher for vector gather instructions consisting of two prefetchers. The first prefetcher is the stream list vector prefetcher that loads the data of the index array as sequential accesses. The second prefetcher is the indirect vector prefetcher that loads the data of indirect memory accesses. The prefetching mechanism uses the index values to load the data of the indirect memory access in advance. The prefetch distance and prefetch degree are discussed for the prefetching mechanism to realize performance improvement over several spatial localities of vector gather instructions. The proposed mechanism realizes performance improvement of the scale kernel with vector gather instructions by 2.1 times on the sequential index array and 1.2 times on the random index array, respectively. By the proposed mechanism, the latency of preceding dependent instructions is hidden by prefetching.

Chapter 4 proposes a criticality-aware out-of-order mechanism for vector processors. The proposed mechanism exploits ILP by prematurely dispatching subsequent instructions that may cause pipeline stalling. The basic concept is that the proposed mechanism retains the vector registers storing the results to avoid

occupying core-cache bandwidth redundantly. The proposed mechanism prepares several queues to retain information of the vector registers. By using these information, the proposed mechanism reorders vector instructions beyond conventional out-of-order mechanism and keep the true dependencies of these instructions. By realizing this concept, the vector processor can exploit ILP without occupying core-cache bandwidth redundantly. The evaluation results show that the proposed mechanism achieves up to 80% performance improvements on several memory-intensive applications. By the proposed mechanism, the vector processors can issue the instructions that cause long latency over the conventional out-of-order mechanisms, solving the latency of issuing.

Chapter 5 proposes an address coalescing method for vector instructions that utilizes arithmetic vector units already built in the vector processor. The virtual memory decouples the memory addresses handled by a process from the actual memory addresses, thereby preventing memory address conflicts between processes. When a processor accesses memory, it is necessary to translate virtual memory addresses to physical memory addresses. Since vector instructions handle multiple elements in one instruction, multiple addresses must be translated. This translation may cause a long latency. The proposed method reduces the number of translations by deduplicating the virtual addresses of vector instructions. The evaluation results show that the proposed method can achieve an average 2x performance improvement in numerical applications and 1.08x in graph applications. These results suggest that the proposed method can reduce the latency for address translations of vector instructions.

Chapter 6 proposes a skewed cache for vector processors that prevents cache blocks using the same cache set. The skewed cache uses a hash function to determine the set that is the place of the data in the cache. This mechanism prevents

the cache from using the same set for addresses, reducing cache misses due to conflicts. This chapter discusses two components that affect the ability to reduce conflict misses: the hashing function and the replacement policy. The evaluation results show that the proposed mechanism can increase the number of hits by up to 70% and marks excellent results very close to those of the theoretical upper bound of the hit rate of the shared cache. By using the proposed mechanism, the cache works efficiently, which significantly reduces the memory access latency.

This dissertation presents four proposals that solve the latency-related problems of modern vector processors, enabling the development of latency-tolerant vector processors. These proposals are expected to enable modern vector processors to achieve high sustained performance on memory-intensive applications by realizing tolerance to latency. As a result, these solutions to latency-related problems will be valuable for architects designing vector processors in the future.

As future work, it may be worthwhile to evaluate all the proposed methods combined together in a single vector architecture. It would also be interesting to assess the effectiveness of the proposed methods applied to vector scatter instructions that perform writes using indirect memory accesses and are often used vector gather instructions together, while this dissertation primarily focused on vector gather instructions with regard to indirect memory accesses.

Bibliography

[1] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 5th ed., 2011.

[2] K. Komatsu, S. Momose, Y. Isobe, O. Watanabe, A. Musa, M. Yokokawa, T. Aoyama, M. Sato, and H. Kobayashi, "Performance Evaluation of a Vector Supercomputer SX-Aurora TSUBASA," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 685–696, 2018.

[3] R. Egawa, K. Komatsu, Y. Isobe, T. Kato, S. Fujimoto, H. Takizawa, A. Musa, and H. Kobayashi, "Performance and Power Analysis of SX-ACE Using HP-X Benchmark Programs," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 693–700, 2017.

[4] "Intel 64 and IA-32 Architectures Optimization Reference Manual."

[5] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, A. Reid, A. Rico, and P. Walker, "The ARM Scalable Vector Extension," *IEEE Micro*, vol. 37, no. 2, pp. 26–39, 2017.

[6] D. A. Patterson, “Latency Lags Bandwith,” *Commun. ACM*, vol. 47, p. 71–75, oct 2004.

[7] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 5th ed., 2013.

[8] R. Egawa, S. Momose, K. Komatsu, Y. Isobe, A. Musa, H. Takizawa, and H. Kobayashi, “Early evaluation of the SX-ACE processor,” in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC14)*, 2014.

[9] A. Onodera, K. Komatsu, S. Fujimoto, Y. Isobe, M. Sato, and H. Kobayashi, “Optimization of the himeno benchmark for SX-Aurora TSUBASA,” in *Benchmarking, Measuring, and Optimizing* (F. Wolf and W. Gao, eds.), (Cham), pp. 127–143, Springer International Publishing, 2021.

[10] K. Komatsu, A. Onodera, E. Focht, S. Fujimoto, Y. Isobe, S. Momose, M. Sato, and H. Kobayashi, “Performance and Power Analysis of a Vector Computing System,” *Supercomputing Frontiers and Innovations*, vol. 8, no. 2, 2021.

[11] I. V. Afanasyev, V. V. Voevodin, V. V. Voevodin, K. Komatsu, and H. Kobayashi, “Analysis of Relationship Between SIMD-Processing Features Used in NVIDIA GPUs and NEC SX-Aurora TSUBASA Vector Processors,” in *Parallel Computing Technologies* (V. Malyshkin, ed.), (Cham), pp. 125–139, Springer International Publishing, 2019.

[12] T. Soga, A. Musa, K. Okabe, K. Komatsu, R. Egawa, H. Takizawa, H. Kobayashi, S. Takahashi, D. Sasaki, and K. Nakahashi, “Performance of

SOR methods on modern vector and scalar processors," *Computers & Fluids*, vol. 45, no. 1, pp. 215–221, 2011. 22nd International Conference on Parallel Computational Fluid Dynamics (ParCFD 2010).

[13] R. Egawa, K. Komatsu, and H. Kobayashi, "Designing an HPC Refactoring Catalog Toward the Exa-scale Computing Era," in *Sustained Simulation Performance 2014* (M. M. Resch, W. Bez, E. Focht, H. Kobayashi, and N. Patel, eds.), (Cham), pp. 91–98, Springer International Publishing, 2015.

[14] R. Egawa, K. Komatsu, and H. Takizawa, "Designing an open database of system-aware code optimizations," in *2017 Fifth International Symposium on Computing and Networking (CANDAR)*, pp. 369–374, 2017.

[15] K. Asifuzzaman, M. Abuelala, M. Hassan, and F. J. Cazorla, "Demystifying the Characteristics of High Bandwidth Memory for Real-Time Systems," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pp. 1–9, 2021.

[16] Y. Yamada and S. Momose, "Vector engine processor of NEC's brand-new supercomputer SX-Aurora TSUBASA," in *Intenational symposium on High Performance Chips (Hot Chips2018)*, 2018.

[17] S. Y. Hou, W. C. Chen, C. Hu, C. Chiu, K. C. Ting, T. S. Lin, W. H. Wei, W. C. Chiou, V. J. C. Lin, V. C. Y. Chang, C. T. Wang, C. H. Wu, and D. Yu, "Wafer-Level Integration of an Advanced Logic-Memory System Through the Second-Generation CoWoS Technology," *IEEE Transactions on Electron Devices*, vol. 64, no. 10, pp. 4071–4077, 2017.

[18] R. Egawa, K. Komatsu, S. Momose, Y. Isobe, A. Musa, H. Takizawa, and H. Kobayashi, "Potential of a Modern Vector Supercomputer for Practical Applications: Performance Evaluation of SX-ACE," *J. Supercomput.*, vol. 73, no. 9, pp. 3948–3976, 2017.

[19] T. Soga, A. Musa, Y. Shimomura, R. Egawa, K. Itakura, H. Takizawa, K. Okabe, and H. Kobayashi, "Performance evaluation of nec sx-9 using real science and engineering applications," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pp. 1–12, 2009.

[20] A. Musa, T. Abe, T. Kishitani, T. Inoue, M. Sato, K. Komatsu, Y. Murashima, S. Koshimura, and H. Kobayashi, "Performance Evaluation of Tsunami Inundation Simulation on SX-Aurora TSUBASA," in *Computational Science – ICCS 2019* (J. M. F. Rodrigues, P. J. S. Cardoso, J. Monteiro, R. Lam, V. V. Krzhizhanovskaya, M. H. Lees, J. J. Dongarra, and P. M. Sloot, eds.), (Cham), pp. 363–376, Springer International Publishing, 2019.

[21] F. D. P. Michels, L. M. Schnorr, and P. O. A. Navaux, "Investigating Oil and Gas CSEM Application on Vector Architectures," in *Computational Science and Its Applications – ICCSA 2022 Workshops* (O. Gervasi, B. Murgante, S. Misra, A. M. A. C. Rocha, and C. Garau, eds.), (Cham), pp. 650–667, Springer International Publishing, 2022.

[22] S. Yoshida, A. Endo, H. Kaneyasu, and S. Date, "First experience of accelerating a field-induced chiral transition simulation using the sx-aurora tsubasa," *Supercomputing Frontiers and Innovations*, vol. 8, p. 43–58, Aug. 2021.

[23] L. Solis-Vasquez, E. Focht, and A. Koch, “Mapping irregular computations for molecular docking to the sx-aurora tsubasa vector engine,” in *2021 IEEE/ACM 11th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, pp. 1–10, 2021.

[24] I. V. Afanasyev, D. I. Lichmanov, V. Y. Rudyak, and V. V. Voevodin, “Efficient implementation of liquid crystal simulation software on modern hpc platforms,” *Supercomputing Frontiers and Innovations*, vol. 8, p. 104–125, Oct. 2021.

[25] A. Ungethüm, L. Schmidt, J. Pietrzyk, D. Habich, and W. Lehner, “Mastering the nec vector engine accelerator for analytical query processing,” in *2021 IEEE 37th International Conference on Data Engineering Workshops (ICDEW)*, pp. 60–65, 2021.

[26] M. Kumagai, K. Komatsu, F. Takano, T. Araki, M. Sato, and H. Kobayashi, “Combinatorial Clustering Based on an Externally-Defined One-Hot Constraint,” in *2020 Eighth International Symposium on Computing and Networking (CANDAR)*, pp. 59–68, 2020.

[27] M. Kumagai, K. Komatsu, M. Sato, and H. Kobayashi, “Ising-based combinatorial clustering using the kernel method,” in *2021 IEEE 14th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, pp. 197–203, 2021.

[28] I. V. Afanasyev, V. V. Voevodin, K. Komatsu, and H. Kobayashi, “Developing an Efficient Vector-Friendly Implementation of the Breadth-First Search Algorithm for NEC SX-Aurora TSUBASA,” in *Parallel Computational Technologies*, (Cham), pp. 131–145, Springer International Publishing, 2020.

[29] I. V. Afanasyev and V. V. Voevodin, "Developing Efficient Implementations of Connected Component Algorithms for NEC SX-Aurora TSUBASA," *Lobachevskii Journal of Mathematics*, vol. 41, pp. 1417–1426, Aug 2020.

[30] I. Afanasyev, A. Antonov, D. Nikitenko, V. Voevodin, V. Voevodin, K. Komatsu, O. Watanabe, A. Musa, and H. Kobayashi, "Developing efficient implementations of Bellman-Ford and Forward-Backward graph algorithms for NEC SX-ACE," *Supercomputing Frontiers and Innovations*, vol. 5, no. 3, pp. 65–69, 2018. Publisher Copyright: © The Authors 2018.

[31] K. Murakami, K. Komatsu, M. Sato, and H. Kobayashi, "A processor selection method based on execution time estimation for machine learning programs," in *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 779–788, 2021.

[32] S. Williams, A. Waterman, and D. Patterson, "Roofline: An Insightful Visual Performance Model for Multicore Architectures," *Commun. ACM*, vol. 52, pp. 65–76, Apr. 2009.

[33] "A64FX Microarchitecture Manual v1.7."

[34] J. Power, M. D. Hill, and D. A. Wood, "Supporting x86-64 address translation for 100s of GPU lanes," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 568–578, 2014.

[35] S. Ainsworth and T. M. Jones, "Software Prefetching for Indirect Memory Accesses: A Microarchitectural Perspective," *ACM Trans. Comput. Syst.*, vol. 36, pp. 8:1–8:34, June 2019.

[36] M. A. Al Farhan and D. E. Keyes, "Optimizations of Unstructured Aerodynamics Computations for Many-core Architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, pp. 2317–2332, Oct 2018.

[37] S. Ainsworth and T. M. Jones, "An event-triggered programmable prefetcher for irregular workloads," *SIGPLAN Not.*, vol. 53, pp. 578–592, Mar. 2018.

[38] J. Dundas and T. Mudge, "Improving Data Cache Performance by Pre-Executing Instructions under a Cache Miss," in *Proceedings of the 11th International Conference on Supercomputing*, ICS '97, (New York, NY, USA), p. 68–75, Association for Computing Machinery, 1997.

[39] O. Mutlu, J. Stark, C. Wilkerson, and Y. Patt, "Runahead execution: an alternative to very large instruction windows for out-of-order processors," in *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.*, pp. 129–140, 2003.

[40] M. Hashemi and Y. N. Patt, "Filtered runahead execution with a runahead buffer," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 358–369, 2015.

[41] A. Naithani, J. Feliu, A. Adileh, and L. Eeckhout, "Precise Runahead Execution," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 397–410, 2020.

[42] T. Ramirez, A. Pajuelo, O. Santana, and M. Valero, "Runahead Threads to improve SMT performance," in *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pp. 149–158, 2008.

[43] S. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton, "Continual flow pipelines: achieving resource-efficient latency tolerance," *IEEE Micro*, vol. 24, no. 6, pp. 62–73, 2004.

[44] A. Hilton and A. Roth, "BOLT: Energy-efficient Out-of-Order Latency-Tolerant execution," in *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pp. 1–12, 2010.

[45] A. Deshmukh and Y. N. Patt, "Criticality Driven Fetch," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, (New York, NY, USA), p. 380–391, Association for Computing Machinery, 2021.

[46] K.-A. Tran, A. Jimborean, T. E. Carlson, K. Koukos, M. Själander, and S. Kaxiras, "Swoop: Software-hardware co-design for non-speculative, execute-ahead, in-order cores," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, (New York, NY, USA), p. 328–343, Association for Computing Machinery, 2018.

[47] J. Vesely, A. Basu, M. Oskin, G. H. Loh, and A. Bhattacharjee, "Observations and opportunities in architecting shared virtual memory for heterogeneous systems," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 161–171, 2016.

[48] S. Puthoor and M. H. Lipasti, "Compiler assisted coalescing," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, PACT '18, (New York, NY, USA), Association for Computing Machinery, 2018.

[49] M. Sato, Y. Ishikawa, H. Tomita, Y. Kodama, T. Odajima, M. Tsuji, H. Yashiro, M. Aoki, N. Shida, I. Miyoshi, K. Hirai, A. Furuya, A. Asato, K. Morita, and T. Shimizu, "Co-Design for A64FX Manycore Processor and Fugaku," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15, 2020.

[50] M. K. Qureshi, D. Thompson, and Y. N. Patt, "The V-Way cache: demand-based associativity via global replacement," in *32nd International Symposium on Computer Architecture (ISCA'05)*, pp. 544–555, 2005.

[51] D. Sanchez and C. Kozyrakis, "The ZCache: Decoupling Ways and Associativity," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 187–198, 2010.

[52] A. Seznec and F. Bodin, "Skewed-associative caches," in *PARLE '93 Parallel Architectures and Languages Europe* (A. Bode, M. Reeve, and G. Wolf, eds.), (Berlin, Heidelberg), pp. 305–316, Springer Berlin Heidelberg, 1993.

[53] A. Musa, Y. Sato, T. Soga, K. Okabe, R. Egawa, H. Takizawa, and H. Kobayashi, "A Shared Cache for a Chip Multi Vector Processor," in *Proceedings of the 9th Workshop on MEmory Performance: DEaling with Applications, Systems and Architecture*, MEDEA '08, (New York, NY, USA), pp. 24–29, ACM, 2008.

[54] R. Egawa, Y. Funaya, R. Nagaoka, A. Musa, H. Takizawa, and H. Kobayashi, "Design and early evaluation of a 3-D die stacked chip multi-vector processor," in *2010 IEEE International 3D Systems Integration Conference (3DIC)*, pp. 1–8, 2010.

[55] R. Egawa, Y. Funaya, R. Nagaoka, Y. Endo, A. Musa, H. Takizawa, and H. Kobayashi, "Effects of 3-D stacked vector cache on energy consumption," in *2011 IEEE International 3D Systems Integration Conference (3DIC), 2011 IEEE International*, pp. 1–6, 2012.

[56] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, "IMP: Indirect Memory Prefetcher," in *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, (New York, NY, USA), pp. 178–190, ACM, 2015.

[57] Intel Corporation., "*Intel* ®64 and IA-32 Architectures Software Developer's Manual V2."

[58] S. Momose, T. Hagiwara, Y. Isobe, and H. Takahara, "The Brand-New Vector Supercomputer, SX-ACE," in *Proceedings of the 29th International Conference on Supercomputing - Volume 8488*, ISC 2014, (New York, NY, USA), pp. 199–214, Springer-Verlag New York, Inc., 2014.

[59] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 Simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011.

[60] O. Mutlu, H. Kim, and Y. Patt, "Techniques for efficient processing in runahead execution engines," in *32nd International Symposium on Computer Architecture (ISCA'05)*, pp. 370–381, 2005.

[61] G. Sohi, "Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers," *IEEE Transactions on Computers*, vol. 39, no. 3, pp. 349–359, 1990.

[62] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 Simulator," *SIGARCH Comput. Archit. News*, vol. 39, p. 1–7, Aug. 2011.

[63] L.-N. Pouchet, "PolyBench/C 3.2."

[64] M. A. Abella-González, P. Carollo-Fernández, L.-N. Pouchet, F. Rastello, and G. Rodríguez, "PolyBench/Python: Benchmarking Python Environments with Polyhedral Optimizations," in *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, CC 2021, (New York, NY, USA), p. 59–70, Association for Computing Machinery, 2021.

[65] H. Sato, Y. Takagi, and K. Sawaya, "High Gain Antipodal Fermi Antenna with Low Cross Polarization," *IEICE Transactions on Communications*, vol. E94.B, no. 8, pp. 2292–2297, 2011.

[66] T. Tsukahara, K. Iwamoto, and H. Kawamura, "Evolution of material line in turbulent channel flow," pp. 549–554, 01 2007.

[67] K. Ariyoshi, T. Matsuzawa, and A. Hasegawa, "The key frictional parameters controlling spatial variations in the speed of postseismic-slip propagation on a subduction plate boundary," *Earth and Planetary Science Letters*, vol. 256, no. 1, pp. 136–146, 2007.

[68] M.Sato, T.Kobayashi, Z.Zeng, G.Fang, and X.Feng, "High resolution GPR system for landmine detection," *Proceedings of Int. Conf. Requirements and Technologies for the Detection, Removal and Neutralization of Landmine and UXO, Brussels, Belgium*, pp. 548–553, September 2003.

[69] Y. Sasao and S. Yamamoto, “Numerical Prediction of Unsteady Flows Through Turbine Stator-Rotor Channels With Condensation,” vol. 1, 01 2005.

[70] Y. Katoh, T. Ono, and M. Iizima, “Numerical simulation of resonant scattering of energetic electrons in the outer radiation belt,” *Earth, Planets and Space*, vol. 57, no. 2, pp. 117–124, 2005.

[71] I. Afanasyev, V. Voevodin, K. Komatsu, and H. Kobayashi, “VGL: a high-performance graph processing framework for the NEC SX-Aurora TSUBASA vector architecture,” *Journal of Supercomputing*, 2021.

[72] I. Afanasyev, V. Voevodin, V. Voevodin, K. Komatsu, and H. Kobayashi, “Developing Efficient Implementations of Shortest Paths and Page Rank Algorithms for NEC SX-Aurora TSUBASA Architecture,” *Lobachevskii Journal of Mathematics*, vol. 40, pp. 1753–1762, Nov. 2019.

[73] V. Karakostas, J. Gandhi, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Unsal, “Energy-efficient address translation,” in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 631–643, 2016.

[74] “SX-Aurora TSUBASA Architecture Guide Revision 1.1.”

[75] K. Komatsu, A. Onodera, E. Focht, S. Fujimoto, Y. Isobe, S. Momose, M. Sato, and H. Kobayashi, “Performance and Power Analysis of a Vector Computing System,” *Supercomputing Frontiers and Innovations*, vol. 8, p. 75–94, Aug. 2021.

[76] A. Seznec, “A Case for Two-way Skewed-associative Caches,” in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA ’93, (New York, NY, USA), pp. 169–178, ACM, 1993.

[77] F. Bodin and A. Seznec, “Skewed associativity improves program performance and enhances predictability,” *IEEE Transactions on Computers*, vol. 46, no. 5, pp. 530–544, 1997.

[78] A. Seznec, “A New Case for Skewed-Associativity,” Research Report RR-3208, INRIA, 1997.

[79] M. Kharbutli, Y. Solihin, and J. Lee, “Eliminating Conflict Misses Using Prime Number-Based Cache Indexing,” *IEEE Trans. Comput.*, vol. 54, no. 5, pp. 573–586, 2005.

[80] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer, “High Performance Cache Replacement Using Re-reference Interval Prediction (RRIP),” *SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 60–71, 2010.

[81] D. Kroft, “Lockup-free Instruction Fetch/Prefetch Cache Organization,” in *Proceedings of the 8th Annual Symposium on Computer Architecture*, ISCA ’81, (Los Alamitos, CA, USA), pp. 81–87, IEEE Computer Society Press, 1981.

Acknowledgements

First of all, I would like to express my deepest gratitude to my supervisor, Professor Hiroaki Kobayashi, for his invaluable guidance and support throughout the course of this project. His expertise and professionalism have been instrumental in helping me to complete this dissertation.

I would like to thank Professor Takafumi Aoki for his thoughtful review of this dissertation and helpful comments. I would also like to thank Associate Professor Kazuhiko Komatsu and Associate Professor Masayuki Sato, for their valuable feedback and encouragement throughout this process.

I am grateful to my lab members for their camaraderie and assistance in the lab. Their insights and hard work have contributed greatly to the success of this project. I am also grateful to my family and friends for their love and support, and for believing in me even when I had doubts.

Finally, I would like to thank all of the participants who generously gave their time and energy to participate in this study. Your contribution is greatly appreciated.

Thank you to all who have supported me to complete this dissertation.