Adaptable Register File Organization
for
Vector Processors

Cristóbal Ramírez Lazo
Department of Computer Architecture
Universitat Politècnica de Catalunya - BarcelonaTech

A thesis submitted for the degree of

Doctor of Philosophy in Computer Architecture

March 2022

Advisors:

Dr. Adrian Cristal Kestelman
Department of Computer Architecture
Universitat Politècnica de Catalunya - BarcelonaTech

Dr. Marco Antonio Ramírez Salinas
Microtechnology and Embedded System Lab,
Centro de Investigación en Computación,
Instituto Politécnico Nacional, México.

Abstract

Nowadays, Exascale computing has become the new milestone for supercomputing. Exascale systems will be strongly constrained by energy efficiency. Therefore, SIMD processing plays an important role in the development of the new Exascale systems. In that sense, the quest for extreme energy efficiency hardware has renewed interest in vector processors.

Today, there are two main vector processors design trends. On the one hand, we have vector processors designed for long vectors lengths such as the SX-Aurora TSUBASA which implements vector lengths of 256 elements[1] (16384-bits). On the other hand, we have vector processors designed for short vectors such as the Fujitsu A64FX that implements vector lengths of 8 elements (512-bit) ARM SVE. However, short vector designs are the most widely adopted in modern chips. This is because, to achieve high-performance with a very high-efficiency, applications executed on long vector designs must feature abundant DLP, then limiting the range of applications. On the contrary, short vector designs are compatible with a larger range of applications. In fact, in the beginnings, long vector length implementations were focused on the HPC market, while short vector length implementations were conceived to improve performance in multimedia tasks. Furthermore, those short vector length extensions have evolved to better fit the needs of modern applications, including features taken from the old vector machines. For example, up until recently, short vector extensions did not offer the more sophisticated addressing modes of vector architectures, namely strided accesses and gather-scatter accesses. This feature, and others, enables these short vector designs to be exploited on scientific applications, engineering, financial analysis, physics simulations, etc. In that sense, we believe that this compatibility with a large range of applications featuring high, medium and low DLP is one of the main reasons behind the trend of building parallel machines with short vectors. Short vector designs are area efficient and are "compatible" with applications having long vectors; moreover, these short vector architectures are not as efficient as longer vector designs when executing high DLP code.

In this thesis, we propose a novel vector architecture that combines the area and resource efficiency characterizing short vector processors with the ability to handle large

[1] From now on, one element corresponds to a 64-bit double-word. For example, a configuration with MVL= 8 elements represents a 512-bits implementation.

DLP applications, as allowed in long vector architectures. In this context, we present AVA, an Adaptable Vector Architecture designed for short vectors (MVL = 16 elements), capable of reconfiguring the MVL when executing applications with abundant DLP, achieving performance comparable to designs for long vectors. The design is based on three complementary concepts. First, a two-stage renaming unit based on a new type of registers termed as Virtual Vector Registers (VVRs), which are an intermediate mapping between the conventional logical and the physical and memory registers. In the first stage, logical registers are renamed to VVRs, while in the second stage, VVRs are renamed to physical registers. Second, a two-level VRF, that supports 64 VVRs whose MVL can be configured from 16 to 128 elements. The first level corresponds to the VVRs mapped in the physical registers held in the 8KB Physical Vector Register File (P-VRF), while the second level represents the VVRs mapped in memory registers held in the Memory Vector Register File (M-VRF). While the baseline configuration (MVL=16 elements) holds all the VVRs in the P-VRF, larger MVL configurations hold a subset of the total VVRs in the P-VRF, and map the remaining part in the M-VRF. Third, we propose a novel two-stage vector issue unit. In the first stage, the second level of mapping between the VVRs and physical registers is performed, while issuing to execute is managed in the second stage.

This thesis also presents a set of tools for designing and evaluating vector architectures. First, a parameterizable vector architecture model implemented on the gem5 simulator which helps to evaluate novel ideas on vector architectures. Second, a Vector Architecture model implemented on the McPAT framework to evaluate power and area metrics, with a target clock rate as a design constraint. Finally, the RiVEC benchmark suite, a collection of ten vectorized applications from different domains focusing on benchmarking vector microarchitectures. These tools are open for the computer architecture community.

Keywords: Computer Architecture, Vector Architectures, Data-level Parallelism, Microarchitecture, RISC-V.

Acknowledgments

First and foremost, I am extremely grateful to my thesis directors, Dr. Adrián Cristal Kestelman and Dr. Marco Antonio Ramírez Salinas, for their invaluable advice, continuous support, and patience during my Ph.D. study. I would also like to give special thanks to Dr. Osman Sabri Unsal, who got involved supporting and advising me in my Ph.D. study as much as my thesis directors, and Prof. Mateo Valero Cortés, who, with his extensive knowledge in vector architectures, helped to improve our work. It has been a privilege to learn from all of you.

I would like to thank Dr. Daniel Jiménez González, Dr. Adrià Armejach Sanosa, and Dr. Carlos Álvarez Martínez for participating in my thesis pre-defense and providing great feedback and suggestions to improve the final version of the thesis.

I would like to thank Prof. Dionisios Pnevmatikatos, and Dr. Vasilis Karakostas for the excellent internship at the Computing Systems Lab at the National Technical University of Athens.

I would like to thank Mr. Lawrence Whitehill for making grammatical corrections and suggestions to improve the writing quality of this thesis.

I would also like to express my gratitude to my colleagues and wonderful friends from the Barcelona Supercomputing Center, who have accompanied me and supported me during this long journey.

Last but not least, I would like to thank my wife Maria and both of our families. Thanks Maria for your tremendous understanding and encouragement. You supported me through thick and thin. We have shared countless great moments together, and I know we will share many more in the next phase of our life with the new little member of our family, our daughter Lucia. Words cannot express how grateful I am to you.

The research leading to these results has received funding from *CONACyT* Mexico under a Ph.D. grant No. 472106, the Spanish State Research Agency - Ministry of Science and Innovation (contract PID2019-107255GB), and the European Union Regional Development Fund within the framework of the ERDF Operational Program of Catalonia 2014-2020 with a grant of 50% of the total cost eligible, under the DRAC project [001-P-001723].

Contents

Contents IV
1 Introduction 1
1.1 Vector Architectures 2
1.2 SIMD Multimedia extensions 5
1.3 The RISC-V Vector Extension 7
1.4 Motivation 8
1.5 Thesis Objectives 10
1.6 Thesis Contributions 11
1.7 Thesis Organization 12
2 Tools for Designing and Evaluating Vector Architectures 14
2.1 gem5 Vector Architecture Model 16
2.1.1 Scalar Core 17
2.1.2 Vector Engine 18
2.1.2.1 Vector Renaming. 19
2.1.2.2 Reorder Buffer. 20
2.1.2.3 Vector Issue Queues. 20
2.1.2.4 Vector Lanes. 21
2.1.2.5 Vector Memory Unit (VMU). 23
2.1.2.6 Lane Interconnection. 25
2.1.2.7 Capabilities and limitations. 26
2.1.2.8 Early Access. 26
2.2 The McPAT framework 27
2.3 The RiVEC Benchmark Suite 30
2.3.1 Study of existing Benchmark Suites 31
2.3.2 Methodology 32
2.3.3 Vectorized applications of the RiVEC Benchmark Suite 33
2.3.3.1 Axpy 36
2.3.3.2 Blackscholes 38
2.3.3.3 Canneal 41
2.3.3.4 Jacobi-2D 45
2.3.3.5 LavaMD2 47

2.3.3.6 PathFinder 49
2.3.3.7 Particle-Filter 51
2.3.3.8 Somier 55
2.3.3.9 Streamcluster 57
2.3.3.10 Swaptions 60
2.4 Evaluation 63
2.4.1 Evaluation Environment 63
2.4.2 Area Evaluation 64
2.4.3 Performance and Energy Evaluation 67
2.4.3.1 Axpy 67
2.4.3.2 Blackscholes 69
2.4.3.3 Canneal 72
2.4.3.4 Jacobi-2D 75
2.4.3.5 LavaMD2 77
2.4.3.6 Pathfinder 79
2.4.3.7 Particle-Filter 81
2.4.3.8 Somier 84
2.4.3.9 Streamcluster 86
2.4.3.10 Swaptions 88
2.5 Related Work 91
2.6 Summary 92
3 Adaptable Vector Architecture 93
3.1 Background and Motivation 94
3.2 Adaptable Vector Architecture (AVA) 97
3.2.1 Two-stage Renaming Unit: Virtual, Physical and Memory Registers 99
3.2.2 Two-level Vector Register File 103
3.2.3 Two-stages Vector Issue Unit 106
3.2.4 Recovering the microarchitectural state 108
3.2.5 AVA Functional Description 108
3.3 Evaluation Methodology 110
3.4 Performance, Energy, and Area Evaluation 112
3.4.1 Axpy 114
3.4.2 Blackscholes 116

3.4.3 LavaMD2 119
3.4.4 Particle-Filter 121
3.4.5 Somier 123
3.4.6 Swaptions 125
3.5 Synthesis and place-and-route 128
3.6 Related Work 130
3.7 Summary 131
4 Conclusions and Future Work 132
4.1 Conclusions 133
4.2 Future Work 135
5 Publications 137
5.1 Publications 138
List of Figures 139
List of Tables 143
Bibliography 145

Chapter 1

Introduction

If you were plowing a field, which would you rather use:
Two strong oxen or 1024 chickens?

Seymour Cray, Father of the Supercomputer
(arguing for two powerful vector processors versus many simple processors)

This chapter first introduces a category of parallel hardware termed Single Instruction Multiple Data (SIMD) and covers two variations: Vector Architectures and Multimedia SIMD instruction set extensions. Since our proposal is based on the RISC-V ISA, a brief introduction of this ISA is given. Then, the motivation behind our work is discussed, followed by the objectives that are addressed in this thesis. Next, the contributions of this thesis are presented, and finally, the thesis organization is outlined.

Parallelism at multiple levels is now the driving force of computer designs, where energy is one of the primary constraints. One effective way to achieve high-performance and efficiency is the exploitation of data-level parallelism (DLP). In this sense, parallel architectures can deliver good performance at a lower cost. One category of such parallel hardware organization is termed Single Instruction Multiple Data (SIMD) [1]. Two variants of SIMD are multimedia extensions and vector architectures [2]. Multimedia extensions allow executing a set of predefined operations over vector registers of a fixed length. In contrast, in a Vector Architecture, there is no single preferred vector length, just the Maximum Vector Length (MVL) is defined, and the application can use any vector length that does not exceed the MVL. Nowadays, most commodity CPUs implement architectures that feature SIMD instructions. Typical examples for Multimedia extensions include Intel x86's MMX, SSE, AVX, AVX2, and AVX-512 [3], MIPS's MDMX , and MSA [4], ARM's NEON [5]. While classical vector extensions for NEC [6] and CRAY [7] are well-known, "the return of the vectors" include such contemporary vector architectures as ARM's SVE [8], SVE2 [9], and RISC-V V extension [10]. The following lines introduce Vector Architectures and Multimedia extensions.

1.1 Vector Architectures

An elegant interpretation of SIMD is called a Vector Architecture, which has been closely identified with supercomputers designed by Seymour Cray [11] [12]. A key element of these architectures is that arithmetic/logic and load/store instructions operate on sets of vectors instead of individual data items. Moreover, instead of having, for example, 32 Arithmetic Logic Units (ALU) to perform 32 operations simultaneously, vector architectures typically exploit long execution pipelines to obtain good performance at a lower cost. One of the main features of vector architectures is the Vector Register File (VRF), where each vector register can hold a large number of elements, and the maximum number of elements is represented by the MVL parameter, which can vary depending on the hardware implementation [2]. Additionally, Vector Architectures introduce the concept of Vector Length (VL), where each application can choose the most convenient VL that does not exceed the MVL.

Figure 1.1 shows the basic structure of a Vector Architecture, highlighting the vector register file, where for this simple example, each vector register can hold 32 elements[2] (2048-bits). Also, it is possible to read eight elements of the first and second sources in only one cycle and allocate it in the source buffers, which are responsible for keeping

[2] From now on, one element corresponds to a 64-bit double-word. For example, a configuration with MVL= 8 elements represents a 512-bits implementation.

the functional unit busy. The functional unit writes one 64-bit result in every cycle in the destination buffer; once this buffer completes a vector register file line (8 elements), it proceeds to write back to the vector register file.

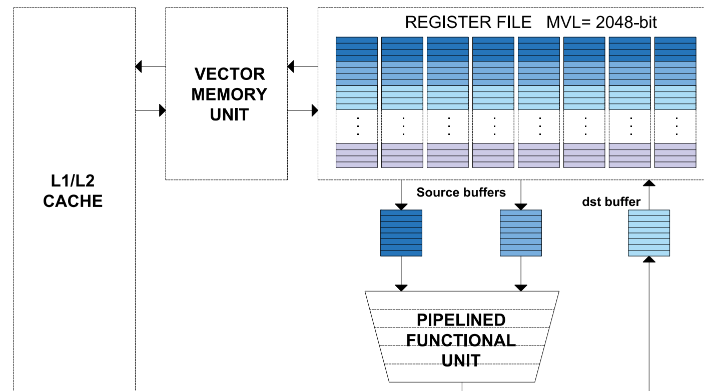

Figure 1.1 *Basic structure of a Vector Architecture*

Vector architectures that include multiple lanes can produce two or more results per clock cycle. Adding multiple vector processing lanes is a popular technique that leads to an advantage in performance and scalability, as shown by Asanovíc [13]. In a multi-lane vector architecture (See Figure 1.2), one lane operates with a register subset of the overall VRF, and a data path subset of the overall vector functional units data paths, where all the lanes work fully synchronized [14]. Inside each lane, the area can be dominated by the VRF, as reported in Ara [15] and Hwacha [16]. Furthermore, multi-lane vector architectures need extra hardware to control the synchronization between lanes, and also a lane-interconnection network to allow data movement between all the lanes.

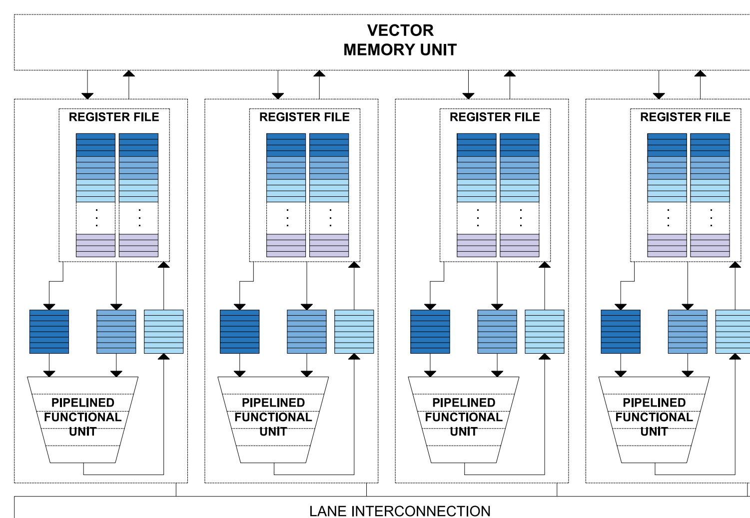

Figure 1.2 *Basic structure of a multi-lane Vector Architecture*

Vector architectures have been traditionally applied to the supercomputing domain (Cray, NEC, IBM) in the 1970s up to the 1990s. The Cray-1 [11] introduced in 1976, was a register-based machine, and the first supercomputer to successfully implement the vector processor design where arithmetic instructions operate on vector registers while separate vector load and store instructions move data between memory and vector registers. In the early eighties, the Japanese manufacturers (NEC, Fujitsu, and Hitachi) entered the vector supercomputer market, introducing lines of parallel vector computers [17]. In the early 90s, there was a radical change in the computer industry. The introduction of faster microprocessors substantially changed the supercomputing market mainly because the FLOPS/$ is substantially lower for commodity-based supercomputers, although vector supercomputers could achieve higher FLOPS. Thus, the idea of building parallel machines based on many out-of-order microprocessors offered an attractive alternative instead of vector supercomputers [17].

In the late 1990s, there were many academic research proposals on vector architectures. Asanović [13] proposed to build a vector microprocessor with the silicon CMOS fabrication technology of that time. He argued that the resulting vector microprocessor could be the fastest, cheapest, and most energy-efficient processor for many future applications. Espasa [18] proposed using decoupling techniques in a vector processor. A first study proposes to split the instruction stream into three different streams through a set of queues: scalar computation instructions, vector computation instructions, and memory accessing instructions (both vector and scalar) and showed that the performance of vector programs could be significantly improved. In a second study, Espasa [19] demonstrated that dynamic scheduling commonly applied to the superscalar processors like register renaming and out-of-order execution could also be applied to the vector processors and obtain significant advantages. Applying dynamic scheduling, they reported a speedup of 1.24-1.72x for realistic memory latencies. Espasa et al. [20] developed Tarantula, a vector extension to the Alpha architecture which support vector operations of up to 128 elements. The studies with Tarantula showed that the vector paradigm could be fully exploited in a real microprocessor environment by integrating Tarantula to the Alpha virtual-memory cache-coherent system and provide good support for non-unit strides and gather/scatter instructions.

Those academic ideas were studied by the processor manufacturers to conceive new designs. A clear example is the Cray X1 [21], launched in 2003, a distributed shared memory multiprocessor with a vector ISA (NV-1), capable of scaling to thousands of processors. The design features a VRF that holds 32 physical registers with an MVL of 64 elements, where each element is 64-bit. The Cray X1 was the first industry product to implement a decoupled vector micro-architecture. Decoupling between the vector

memory unit and the vector execution unit facilitates the dynamic tolerance of memory latency. Moreover, decoupling between the vector and scalar execution units allows scalar execution to run ahead. The next Cray design launched in 2007 was called BlackWidow [22]. Like its predecessor, BlackWidow implements decoupling from the scalar core and improves scalar-vector synchronization primitives with a new vector ISA (NV-2). A large VRF was implemented that has 32 physical registers with MVL=128 elements, each 64-bit wide. The design was organized as an eight-lane configuration, where each lane is associated with 16 elements of every vector register.

One modern example of a vector architecture is the SX-Aurora TSUBASA [23] launched in 2018. SX-Aurora TSUBASA features eight vector cores in a single chip with a frequency of 1.6 GHz. Each vector core includes a scalar processor that provides the basic functionality as a processor (Fetch, decode, exception handling, etc.) and a decoupled Vector Processing Unit (VPU). The VPU includes renaming with 256 physical registers and Out-of-Order scheduling. The MVL is 256 elements each 64-bit wide, and it has 32 Vector Lanes with each lane featuring four pipelines (FMA0, FMA1, ALU0/FMA2, and ALU1/Store), executing up to three arithmetic operations plus one memory operation in parallel.

1.2 SIMD Multimedia extensions

Nowadays we can find vector instructions in all ISAs, including desktop or mobile processors, servers and in processors for supercomputing. However, the current era of SIMD processors grew up in the desktop computer market rather than the supercomputer market. In the 1990s, microprocessors became powerful enough to run multimedia processing tasks, and the demand grew for this particular type of computing. In response, microprocessors vendors turned to SIMD to meet the demand.

Like vector instructions, a SIMD multimedia instruction specifies the same operation on vectors of data. Unlike vector machines with large register files which can hold large number of elements, SIMD multimedia instructions tend to specify fewer operands leading to much smaller register files. Figure 1.3 shows a general view of a 512-bit SIMD multimedia implementation where a complete vector register (512-bit) is read for each source, and send to the functional units to be computed in parallel.

In 1994, Hewlett-Packard introduced the Multimedia Acceleration eXtension (MAX) [24] instructions into the PA-RISC ISA to accelerate MPEG decoding. Sun Microsystems introduced the Visual Instruction Set (VIS) [25] for the SPARC V9 microprocessors. In 1996, MIPS introduced the MIPS Digital Media eXtension (MDMX). However, the first widely deployed desktop SIMD was with Intel's MMX [26] extensions in 1997 which was added to the x86 architecture in the Intel Pentium processors. MMX added only eight

new registers to the architecture (MM0-MM7), but the vector was small (64-bit), and it was only possible to execute a 1x64-bit operation or packed format of 2x32-bit, 4x16-bit, and 8x8-bit operations. Furthermore, MMX only provided integer operations.

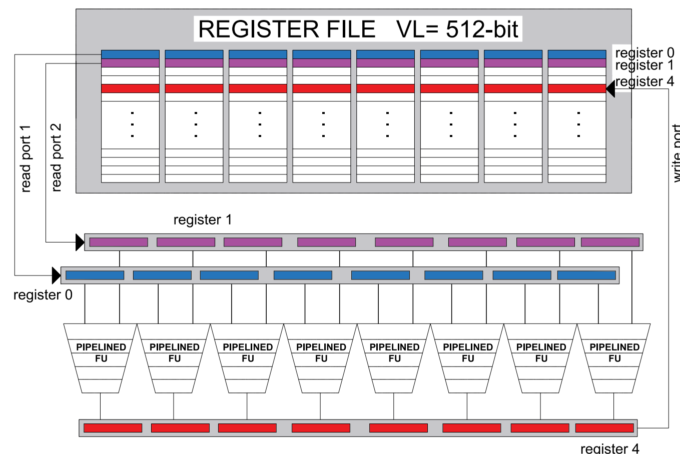

Figure 1.3 *Basic structure of a 512-bit SIMD multimedia implementation.*

In 1999 the Streaming SIMD Extension (SSE) [27] was released and added to the x86 architecture and the Pentium III. The SSE instruction set added eight new 128-bit registers and support for floating-point operations. In all, SSE added 70 new instructions. SSE was subsequently expanded by intel to SSE2 (2001), SSE3 (2004), SSSE3 (2006), SSE4 (2006), and SSE5 (2007) [3].

The next Intel SIMD addition was the Advanced Vector Extensions (AVX) [28] in 2008. The SSE registers were increased from 128 to 256 bits and renamed for AVX. The number of registers was increased from 8 to 16. AVX introduced a three-operand instruction format that allowed the preservation of the input registers. In the same year, ARM introduced NEON [5], a SIMD extension originally for the ARMv7 architecture. It combined 64-bit and 128-bit SIMD instructions that provide standardized acceleration for media and signal processing applications.

Enhancements followed with AVX2 (2013) [28] which added integer 256-bit SIMD instructions, floating-point fused multiply-add, vector shifts and, gather support, enabling vector elements to be loaded from non-contiguous memory locations.

The most recent addition to the Intel SIMD extension is AVX-512 [29]. With AVX-512 the register length was expanded from 256 to 512 bits. The number of registers was expanded from 16 to 32. AVX-512 introduces gather with 512-bit registers and also a scatter instruction, which stores elements from a contiguous vector into non-adjacent memory locations.

All of these developments have been designed for vector lengths of between two and eight 64-bit elements or more for smaller data types. These new SIMD architectures need to be distinguished from older ones, the newer architectures are then considered “short-vector” architectures, as earlier SIMD and vector supercomputers had vector lengths from 64 up to 256 elements.

One important aspect of SIMD Multimedia extensions is that they have fixed the number of data operands in the instruction opcode, which has led to the incorporation of hundreds of instructions in the MMX, SSE, and AVX extensions to the x86 architecture. On the contrary, Vector Architectures defines the MVL, which, combined with the Vector Length (VL) register that specifies the number of elements for the current operation, avoids using many opcodes.

Additionally, technology advances have allowed increasing the vector width in every new incarnation of the SIMD multimedia extensions. This fact leads SIMD multimedia extensions to not only be considered for multimedia applications but also now are extensively used in scientific applications. Also, to better fit with modern scientific applications, features taken from the old vector machines are starting to be integrated on the new SIMD multimedia extensions, getting closer to those old vector machines. For example, up until recently, SIMD multimedia did not offer the more sophisticated addressing modes of vector architectures, namely strided accesses and gather-scatter accesses. AVX-512 is a clear example of this new trend. SIMD multimedia extensions did not offer the mask registers (predication registers) to support conditional execution of elements as in vector architectures. However, this can change in the future AVX 1024-bit or 2048-bit implementations. Probably, in short time, the main differentiator will be either to implement the same number of functional units as number of elements hold in a vector register, or exploit the functional unit pipeline by executing a subset of the total operations every cycle.

1.3 The RISC-V Vector Extension

RISC-V [30] is a free and open-source hardware Instruction Set Architecture (ISA) that enables a new era of processor innovation through open standard collaboration. The project began in 2010 at the University of California, Berkeley, RISC-V ISA delivers a new level of free, extensible software and an open ISA.

As of June 2019, version 2.2 of the user-space ISA and version 1.11 of the privileged ISA are frozen, permitting software and hardware development to proceed. Furthermore, a great effort has been made to propose a Vector Extension which four stable releases

(0.7, 0.8, 0.9, and 1.0-rc), being the v1.0-rc the candidate for the frozen spec for public review.

In that sense, the RISC-V community has chosen relevant features from the long history of vector computing and has defined a standard ISA vector extension [10]. One of the main goals of this new vector extension is to make it an attractive alternative for different market segments. We next highlight three key features of this RISC-V vector extension:

- The vector extension is a Vector Length Agnostic ISA. It means that the length of the vectors is not prescribed as in the common SIMD ISAs (i.e., Intel AVX, AVX2, and AVX-512), allowing the vendor to choose the vector register size while the binary code is portable between different hardware implementations.
- The concept of Vector Length (VL), a feature taken from the classical vector architectures of the 70s, is introduced. Unlike SIMD ISAs where one instruction operates over the whole vector, in a vector architecture, just the Maximum VL (MVL) is defined, and the application can choose any VL that does not exceed the MVL.
- Register Grouping (RG): RG's primary goal is to provide greater execution efficiency for those applications that present high DLP. RG allows that multiple vector registers can be grouped together so that a single vector instruction can operate on multiple vector registers as if it was a single “wider” register at the cost of having fewer available architectural registers

New opportunities for academia and industry have been opened with the incorporation of this new vector extension. In fact, this vector extension has arrived just at the most convenient moment where the quest for extreme energy efficiency hardware has renewed interest in vector architectures. It has not been long since RISC-V announced the first stable release of the RISC-V vector extension, and there are already several open and commercial-based products. Some examples are Ara [15] from ETH Zurich, Xuantie-910 [31] from the Chinese company Alibaba, and the Sifive Performance P270 core [32] from the American company Sifive, to name a few.

1.4 Motivation

Supercomputing has always been instrumental as an initial testing ground for innovative architectures. Today, the new milestone for supercomputing is Exascale. In the most basic sense, Exascale (10^{18} floating-point operations per second) will provide the capability to perform more realistic simulations about the processes involved in

precision medicine, regional climate, the unseen physics in materials discovery and design, the fundamental forces of the universe, and much more. However, highly energy-efficient hardware substrates are needed to achieve Exascale performance levels within the 20 MW power envelope. Vector processors are a prime candidate for such substrates as they are typically highly energy-efficient, for example, by computing on operands composed of vectors instead of scalars, therefore requiring fewer instructions to fetch, or by processing multiple vector instructions simultaneously through techniques such as chaining. In that sense, recent Exascale projects have shown a renewed interest in Vector Architectures. Some examples are the European Processor Initiative (EPI) [33] and the Japanese Post-K [34] projects. The EPI project proposed a RISC-V based design, aiming to develop power-efficient and high throughput accelerators. On the other hand, in the Post-K project context, Fujitsu put into operation the Fugaku supercomputer, which is currently number 1 in the list of TOP500 fastest supercomputers in the world [35]. Fugaku features the Fujitsu ARM A64FX vector processor, which adopts the ARM Scalable Vector Extension (SVE) [8] as an efficient way to achieve Exascale-class performance.

Although both the ARM SVE and the RISC-V vector extensions took inspiration from the more traditional vector architectures, such as the Cray-1 [11], there is a remarkable difference between them. While ARM SVE allows implementations from 128-bits up to 2048-bits, RISC-V does not limit the MVL, thereby allowing implementations not only with short and medium-size vectors but also long vector designs that are akin to classic vector supercomputers [11] [12] [21] and modern vector processors [23] [36]. For example, the Aurora vector processor from NEC [23] can multiply-accumulate two 256 element double-precision floating-point vectors in a single instruction.

The vector architectures designed for long vectors are limited to a specialized subset of applications, where relatively high DLP must be present to achieve excellent performance with high efficiency. However, scientific applications are getting more diverse, and the vector lengths in practical applications vary widely. For example, stencil and graph processing kernels usually feature shorter vectors, while high-performance computing, physics simulation and financial analysis kernels usually feature longer vectors [37]. We believe that this wide diversity is one of the main reasons behind the trend of building parallel machines with short vectors. Short vector designs are area efficient and are "compatible" with applications having long vectors; however, these short vector architectures are not efficient as longer vector designs when executing high DLP code.

To help to address this wide diversity of vector lengths in practical applications, new vector extensions such as RISC-V V-extension and ARM SVE adopt the VLA programming. In VLA, the length of the vectors is not prescribed as in the common Multimedia SIMD ISAs, allowing the vendor to choose the vector MVL while guaranteeing portability of the binary code between different hardware implementations. However, since hardware architectures are designed to target specific MVLs, designing for only short or long MVL leads to inefficiencies when leveraging different DLP patterns. In this work, we tackle this challenge by proposing a novel vector architecture that combines the area and resource efficiency characterizing short vector processors with the ability to handle large DLP applications, as allowed in long vector architectures.

1.5 Thesis Objectives

The main goal of this thesis is to propose a novel Vector Architecture able to adapt the microarchitecture according to the application characteristics to do an efficient use of the resources and improve performance. In order to achieve this goal, the tasks can be broken into individual objectives.

The first objective is to study the behavior of a set of applications when executing on different Vector Architecture configurations. This is varying the vector length, number of lanes, lane-interconnection, memory hierarchy, etc. This objective can be broken into individual goals.

- Since the RISC-V Vector extension ISA is still in development, there is no standard tool to evaluate this type of research, and a common practice among the researchers is to develop their own tools. This practice is a limitation both in the development of the research and in the analysis of the resultant data from the experiments. Beneath this necessity, the gem5 simulator is taken and extended with a parameterizable timing model of a Vector Processing Unit supporting the execution of the new RISC-V Vector instructions over several configurations.
- The second goal is to develop a vectorized benchmark suite; a collection composed of data-parallel applications that can be classified according to the modules that are stressed in the vector architecture. One important requirement is to provide applications from different domains to explore different scenarios.
- The third goal is to study the vectorized suite executed on the extended gem5 simulator in order to see how they perform on the different VL, number of lanes, etc.

Based on the data collected from the initial study, **the second objective** of this work is to propose a reconfigurable Vector Architecture, which means that it can modify its own structures depending on application needs. This objective can be broken into individual goals.

- Design and implement a reconfigurable Vector Architecture model on the previous extended gem5 simulator.
- Evaluate performance with a set of applications taken from the vectorized benchmark suite.
- Evaluate area, energy, achievable frequency.

1.6 Thesis Contributions

This thesis makes the following contributions:

The main contribution of this thesis is a novel register file organization for vector processors that provides the ability to adapt the MVL depending on the application needs. Our design termed as AVA (Adaptable Vector Architecture) is a VPU designed for short vector lengths (16 elements), but with the ability to reconfigure the MVL, unlocking the benefits of having a longer vector (128 elements) microarchitecture when abundant DLP is present in the application. To enable this feature, AVA microarchitecture revolves around three complementary concepts: First, a two-stage renaming unit based on a new type of registers termed as *Virtual Vector Registers* (VVRs). Second, a two-level vector register file that supports 64 VVRs with MVL from 16 to up to 128 elements depending on the configuration. The first level corresponds to the VVRs mapped in the physical registers held in the 8KB Physical Vector Register File (P-VRF), while the second level represents the VVRs mapped in memory registers held in the Memory Vector Register File (M-VRF). Third, a novel two-stage vector issue unit. In the first stage, the mapping between the VVRs and physical registers is supported by the Swap-Mechanism, while issuing to execute is managed in the second stage. Since the hardware is designed for short vector lengths, it is possible to keep the P-VRF modest in size (8KB for 64 16-elements vector registers), so it has the advantage of area efficiency of short vector designs. Our results demonstrate that by having a modest VPU designed for short vectors, plus our novel scheduling mechanism, it is possible to: (1) obtain competitive performance when comparing AVA with the equivalent long vector hardware (i.e., VPU with a MVL=128 elements, VRF=64KB); (2) save around 53% of the total VPU area compared with a long vector hardware (i.e., VPU with a MVL=128 elements, VRF=64KB); (3) obtain energy savings when reconfiguring for longer vector lengths; and (4) achieve higher working frequencies.

The next contribution of this thesis is a complete framework for designing and evaluating vector architectures. The framework is composed of three tools:

- First, a vector architecture model implemented on the gem5 simulator [38]. This model allows designers to evaluate different novel ideas on vector architectures. The model corresponds to a decoupled vector architecture, and several vector micro-architecture configurations can be evaluated since the number of physical vector registers, MVL, number of queue entries, issue scheme, number of lanes, the latency of the functional units, latency and topology of the lanes interconnection and number of memory ports are customizable.

- Second, a parameterizable vector architecture model implemented on the McPAT [39] framework to evaluate area and power metrics, with a target clock rate as a design constraint. The model can be configured by varying the MVL, the number of lanes, and the number of functional units. Also, since the vector register file is a critical element in a vector architecture, it is possible to model detailed internal organization such as defining the number of memory banks inside each lane, or the number of read and write ports.

- Third, the RiVEC benchmark suite [40], a collection composed of ten data-parallel applications from different domains. All the applications (C and C++ programs) were extended by adding the RISC-V vectorized version. The implementations make use of intrinsics, and the code was written in a vector length agnostic fashion. The applications present different possible scenarios that may occur within different vector architecture designs that can operate from short to long vector lengths, taking into account the different modules that can be evaluated in a vector architecture such as the lanes, the interconnection between lanes and the memory management.

Finally, the complete framework [38] [39] [40] is open for the computer architecture community.

1.7 Thesis Organization

The rest of the document is organized as follows:

Chapter 2 presents a set of tools for designing and evaluating vector architectures. First, the gem5 simulator and the McPAT framework extended with a parameterizable vector architecture model are presented. Second, the RiVEC benchmark suite is described. And finally, an evaluation of the benchmark suite is highlighted.

Chapter 3 presents AVA, our novel proposal, which allows managing different DLP patterns in an efficient way. It describes the proposal, and present the evaluation for performance, area, and energy.

Chapter 4, summarizes the contributions presented in this thesis and points to future research directions.

Chapter 5 shows the publications related to this research.

Chapter 2

Tools for Designing and Evaluating Vector Architectures

We call these algorithms data-parallel algorithms because their parallelism comes from simultaneous operations across large sets of data rather than from multiple threads of control.

W. Daniel Hillis and Guy L. Steele
"Data parallel algorithms" Commun. ACM (1986)

This chapter presents the basic tools for initial research on RISC-V Vector Architectures. First, the gem5 simulator is extended to support the execution of RISC-V Vector instructions by adding a parameterizable Vector Architecture model for designers to evaluate different approaches according to the target they pursue. Second, to obtain area and power metrics, the McPAT framework is extended by modeling the main modules of a Vector Architecture. Third, a novel Vectorized Benchmark Suite is presented, a collection composed of ten data-parallel applications from different domains that can be classified according to the modules that are stressed in the vector architecture. Finally, a study of several VPU configurations is presented.

One of the most used platforms for computer-system architecture research encompassing system-level architecture as well as processor microarchitecture is gem5 [41] which can be used to test those novel ideas on vector architectures. For Multimedia extensions, gem5 supports Intel's MMX and SSE (64-bit and 128-bit extensions) instructions, which are implemented as part of the core microarchitecture. However, support for more recent extensions such as AVX2 and AVX-512 is missing. On the vector architecture side, there is full support for ARM SVE, where the MVL allowed by the architecture is 2048-bit (32 elements each 64-bit). However, despite the relevance of vector architectures, gem5 does not have a public distribution, which includes a vector architecture model that evaluates different implementations including short (around 512-bit), medium (around 4096-bit), and large (around 16384-bit or more) vectors combined with a flexible and customizable model that fits with the research requirements. In consequence, researchers have to limit their explorations to the MVL allowed by the current models, decreasing the possible scenarios that could allow a flexible and customizable model without MVL limitation. In this sense, the incorporation of the new RISC-V vector extension will offer to the computer architecture community maximum freedom in the research and development of new acceleration technologies where the MVL can be chosen by the architect instead of being restricted by the architecture.

On the benchmark suites side, as novel architecture designs have appeared, the need for new benchmark suites arises. There are several suites to measure single-core performance over data-parallel applications, such as Parboil [42] and Polybench [43]. Also, there are several suites focused on parallel computing on general-purpose CPU architectures such as PARSEC [44] and HPC Challenge Benchmark Suite [45], as well as others for heterogeneous computing such as Rodinia [46], and Polybench/GPU [43], covering MPI, OpenMP, OpenCL, and CUDA programming models, while SIMD Suites are very limited such as ParVec [47]. It is well-known that many applications can benefit from vector execution achieving higher performance, higher energy efficiency, and greater resource utilization. Moreover, the effectiveness of the hardware depends not only on the hardware design but also on the compiler's ability to vectorize the code to be executed. However, there is a tremendous variation in how different compilers perform in vectorizing programs [48]. Supporting auto-vectorizing large codes is currently too limiting for the compilers. Achieving high percentage of vectorization as well as good quality vectorized code typically relies on the programmer´s effort. For example, rewriting the code to obtain well-structured control flow or vectorizing the code using intrinsics. This effort is one of the principal reasons for not having many vectorized benchmark suites. Despite this, suites to evaluate the different modules that compose a Vector

Architecture have received little attention from previous work on benchmark development.

One contribution of this thesis is to provide a complete framework to evaluate performance, area, energy, and achievable frequency to enable researchers to test novel ideas on vector architectures. Our gem5-based simulator baseline model corresponds to a decoupled vector architecture, and different vector micro-architecture implementations can be evaluated since the number of physical vector registers, MVL, number of queue entries, issue scheme, number of lanes, the latency of the functional units, latency and topology of the lanes interconnection and number of memory ports are customizable. To help evaluate these architectures, a novel Vectorized Benchmark Suite was developed which covers the different possible scenarios that may occur within different vector architecture designs that can operate from short MVL to large MVL, taking into account the different modules that can be evaluated in a vector architecture such as the lanes, the interconnection between lanes and the memory management.

This chapter is organized as follows. In Section 2.1, a detailed description of our vector architecture model implemented on gem5 is shown. Section 2.2 presents the vector architecture model implemented on the McPAT framework. Then, the RiVEC Benchmark Suite is presented in Section 2.3, describing how the vectorized versions were implemented and showing the degree of vectorization achieved. Once the tools are detailed in Section 2.4, a study of the scalability for each application executed on different configurations of the gem5-based vector architecture model is highlighted. Section 2.5 focuses on the related work. Finally, Section 2.6 summarizes the key points of this chapter.

2.1 gem5 Vector Architecture Model

The gem5 simulator has been extended to model a decoupled vector architecture. The customization provided by the parameter-based model allows the designer to obtain a vector engine design capable of achieving a tradeoff between performance, energy efficiency and area. In that sense, it is possible to simulate a design that fits with the researcher's requirements. For example, a vector engine designed for HPC, by setting a design for large vectors (256 64-bit elements), composed of a renaming unit capable of supporting 64 physical registers, a vector arithmetic and a memory queue with sixteen entries; these features can be set-up to work with say, eight lanes. In contrast, the vector engine can also be targeted for the embedded market segment by setting a design for short vectors (8 64-bit elements), reducing the number of available physical registers, and with only one-lane configuration. The main goal of this work is to obtain the more

flexible and customizable vector engine for researchers; in that sense, several decisions were taken into account.

Create a model based on RISC-V. The adoption of RISC-V is a key factor for having maximum freedom both in the research and development of the new technologies, without the limitation from hardware and software ecosystems. Furthermore, the new V extension includes a key feature which is unbounded MVL size, which, in combination with a flexible and customizable model, could lead to set designs that support very long vectors.

Decoupled vector engine. A decoupled design provides several advantages. First, decoupling between the vector memory unit and the vector execution unit facilitates the dynamic tolerance of memory latency. Secondly, decoupling between the vector and scalar execution units allows scalar execution to run ahead. Implementing a vector engine as a pipeline tightly coupled to an aggressive out-of-order superscalar core is a typical implementation. In fact, most commodity CPUs that feature SIMD instructions work in this way. However, these designs are optimized for short vectors such as Intel's AVX-512, where the VRF size is around 2KB (32 vector registers each 512-bit wide), and the area overhead added to the superscalar core is acceptable without limiting the maximum frequency of the superscalar core. The case for vector engines designed for long vectors is a different story. In this case, the VRF size is around 64KB (32 vector registers each 16,384-bit wide) or more when implementing renaming, which could lead to a big area overhead and could limit the maximum frequency of the superscalar core. Contemporary vector architectures are implemented as decoupled engines running around 1.5GHz (e.g., Cray BlackWidow runs at 1.3 GHz, and SX-Aurora TSUBASA runs at 1.6 GHz), mainly limited by the big structures needed to hold long vectors. In contrast, superscalar cores run at higher frequencies. Furthermore, by having a decoupled design, it is possible to study different possibilities to get energy-efficient architectures. Examples include clock gating, turning off the clock of inactive modules to save energy and dynamic power, or applying dynamic voltage-frequency scaling when there are periods of low activity where there is no need to operate at the highest clock frequency and voltage.

2.1.1 Scalar Core

Different fully parameterizable CPU models are provided by gem5, such as the in-order CPU and the out-of-order CPU, which allows micro-architectural simulations. In this work, extra support to the in-order CPU pipeline was added to recognize the vector instructions and perform different tasks before sending these instructions to the vector engine. The core runs concurrently with the vector engine, so most of the scalar operations are amortized underneath vector execution. The scalar core is responsible

for fetching and decoding the vector instructions and carrying them through the pipeline. Most of the vector instructions are treated as *nop* operations in the scalar core. Furthermore, if the vector instruction has a scalar operand as a source, it must read the scoreboard to check if the source operand is ready. Then, the vector instructions continue to the next stages until they reach the commit stage, where they are sent to the vector engine. With this, the vector instruction execution won't be interrupted by any possible control hazard, such as a miss branch prediction generated by older scalar instructions.

Two different simulation modes are provided by gem5, Full System (FS) mode, and Syscall Emulation (SE) mode. The first provides the ability to simulate a full system. It can boot an operating system, handle interrupts, exceptions, and fault handlers. The second, the SE mode, focuses on the CPU and memory system and does not emulate the entire system, which implies that interrupts, exceptions, and fault handlers are trapped and managed by the host OS without running a handler routine to manage the event. The gem5 RISC-V implementation still does not have the support to run in FS mode. Consequently, the vector architecture model is available only to run in SE mode. Having said the above, once the vector instruction is sent to the vector engine, it can be retired from the scalar pipeline since any exception generated by a vector instruction, such as a page fault caused by a memory request, is trapped and managed by gem5.

2.1.2 Vector Engine

Once the vector instructions arrive at the vector engine, they are first renamed to remove false dependencies, increasing the amount of instruction-level parallelism (ILP) that can be exploited. Then, two operations are performed in parallel. The first is to assign one entry in the reorder buffer, and the second is to allocate the instruction in a temporal queue depending on the instruction type (arithmetic or memory). Once assigned in the corresponding queue, the instruction waits until it fulfills the requirements to be issued; its operands become ready, and the corresponding execution unit is available (the memory unit or the vector lane). Then, when the instruction completes execution, the commit is made by retiring the instruction and freeing up the hardware resources consumed. Our decoupled architecture design has some other unique features for efficiency. For example, as explained in Section 3.2.3, the vector lane architecture was carefully tuned to minimize pipeline bubbles due to structural hazards. The following subsections present a more detailed description of the components of the vector architecture model.

Figure 2.1 shows the general view of the vector engine model. Some specific configurations are also included to explain the interaction between the internal modules.

The model presented corresponds to a multi-lane vector engine. By setting eight lanes, only one memory port could be enough to feed all the lanes, taking into account that the cache line size is set to 512-bit (8 elements each 64-bit), and with every cache line request, it is possible to send one element to each lane in an interleaved fashion. The MVL is set to 256 elements (16384-bits). The VRF line size is set to 8 elements(512-bits). Also, a ring topology for lane interconnection is chosen.

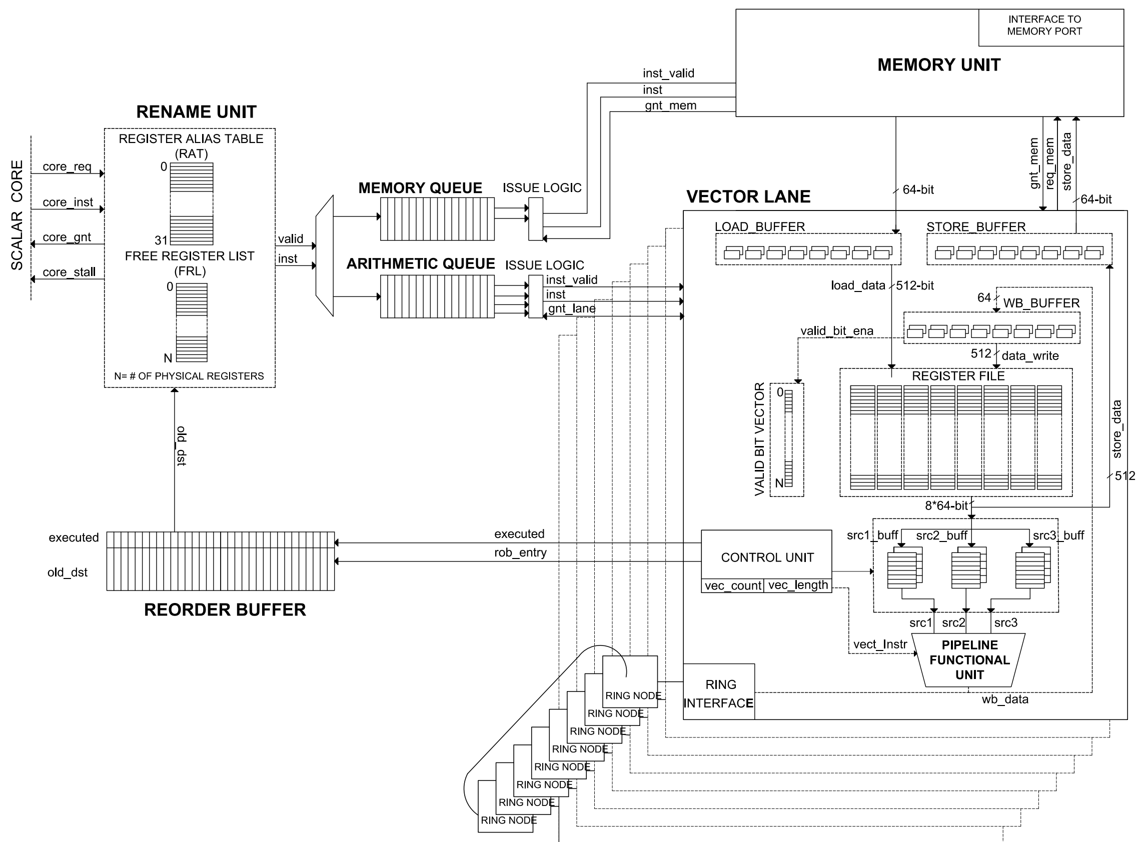

Figure 2.1 *gem5 Vector Architecture Model.*

2.1.2.1 Vector Renaming.

As part of the dynamic scheduling implemented in the design, register renaming is performed. The goal of the renaming is to remove false dependencies by changing the names of the source logical registers to its corresponding physical register that was mapped previously. Additionally, the logical destination register is renamed to a new physical register. This is performed by reading a structure termed as the Free Register List (FRL), which contains all the available physical registers. This mapping is stored in another structure termed as the Register Alias Table (RAT), where the logical destination operates as the write address. At the same time, the logical sources and destination

registers read the RAT in order to obtain the corresponding physical source registers and the physical destination register that was mapped by a previous instruction, also known as old-destination. Then, these structures are coupled with a dependency check logic to analyze the instruction and solve any write-after-write dependencies. The physical registers in-flight (old-destinations) that are no longer used are appended to the free register list at commit time. Detailed information about the commit process can be found in Section 3.2.2. Finally, the number of physical registers can be set by the designer.

2.1.2.2 Reorder Buffer.

The implemented vector architecture model permits choosing the issue scheme, which can be in-order or out-of-order; for that reason, a structure to preserve the program order is needed. A Reorder Buffer (ROB) is a structure that allows instructions to be committed in-order. Also, it holds important information about the instruction that can be useful during and at the end of its execution, such as the program counter, the physical *old-destination*, and a bit field termed as *executed.* The *executed* bit allows knowing if the instruction has been completed or not. The number of ROB entries can be set by the designer.

When a new instruction arrives at the ROB, it is allocated in the next available entry signaled by the tail pointer (write pointer). At the same time, the address of the assigned entry is sent together with the instruction to the corresponding queue. In that way, the instructions know their locations in the ROB, and they can write to it when it is needed.

The *executed* bit field associated with the corresponding ROB entry is set when an instruction finalizes its execution. It means that the instruction is ready to be committed. However, since the commit is performed in-order, the instruction must wait its turn to start this process. The head pointer defines the turn (read pointer). When the instruction pointed by the head pointer has the *executed* bit set, it means that it can commit. If this is the case, the physical *old-destination* is written back to the FRL structure, to be assigned later to a new instruction. Also, the head pointer advances to the next entry to evaluate a new instruction in the next cycles.

2.1.2.3 Vector Issue Queues.

As mentioned before, the design of the vector engine corresponds to a decoupled vector architecture, meaning that memory instructions and arithmetic instructions are buffered in different queues (arithmetic and memory queue) until fulfilling all the requirements to be issued. In this scheme, it is allowed to execute independent memory instructions ahead of arithmetic instructions and vice versa. This stage is called Issue,

and it is in charge of dispatching instructions to the vector lanes or to the vector memory unit.

The issue stage is composed of two fundamental modules termed as Instruction Queue and the Issue Logic. The scheduling can be configured to use an in-order or out-of-order issue scheme. In addition, the number of entries in the issue queues also can be configured.

The instructions are issued as soon as they fulfill the requirements. First, the source operands must be ready; this is done by reading a structure called *Valid-Bit* (more detailed information about the *Valid-bit* structure can be found in Section 3.2.4). Secondly, the hardware resources needed for execution must be available. An important restriction is that the vector lanes only support the execution of one arithmetic instruction at a time. This means that for certain arithmetic instructions, all source operands can be ready. However, the issue queue must wait until the previous instruction finishes its execution. Note that it is possible to execute memory operations at the same time.

In the special case of the memory queue, if an out-of-order issue scheme is selected, a dynamic memory disambiguation logic is enabled to check for possible memory hazards between load and stores held in the queue. Once the instruction arrives at the memory queue, the disambiguation process sets a bit called *memory hazard*. First, the load is disambiguated against all the stores in the queues. In this case, for every memory reference (load/store), there is a Vector Base Address (VBA), a Vector Length (VL), a Vector Stride (VS), and Standard Element Width (SEW) in bytes. The memory range accessed by a vector reference is defined as a set of memory locations located between VBA and VBA+(VL*VS*SEW) -1. Then, there is a memory hazard between a vector load and vector store if their corresponding memory ranges overlap at least one byte. Scatters/gathers operations (more detailed information about gather/scatter instructions can be found in Section 3.2.5) represent a special case where characterizing by a memory range implies more complex implementations. Then, these operations are executed in order.

2.1.2.4 Vector Lanes.

Figure 2.1 shows a simplified picture of the internal modules that comprise one vector lane. The vector engine can be configured with the required number of lanes. A key aspect is the VRF. In gem5, this is modeled as a simple memory, and it is possible to choose the number of read/write ports. However, the designer should take into account that in a hardware implementation, the number of ports would be highly constrained, especially in a large register file. This is because adding additional ports to

an SRAM memory could increase area and limit the maximum operating frequency or require more than one cycle to read/write in the VRF.

One important source of overhead is the *start-up time*, which is the latency in clock cycles until the pipeline is full [48]. The start-up time is principally determined by the pipeline latency of the vector functional unit. Moreover, the number of read ports in the VRF also can influence the *start-up time.* For example, in a vector engine designed for low power, a one read/write port SRAM memory can be used. In that sense, when a vector multiply-add operation arrives at the lane, in a first cycle, it can read the source1, in a second cycle, it can read the source2, and finally, in the third cycle, it can read the source3. All these read operations take three cycles, which are added to the start-up time. On the contrary, if a VRF with three read ports and one write port is chosen, the read of the three operands can be made in only one cycle. It is a design decision that can be taken according to the final target. For long vector length implementations executing an arithmetic operation can take several cycles, then paying three cycles could be negligible. For a short vector length implementation where the full vector can be computed in less than a dozen cycles, paying three cycles in every instruction could lead to a severe performance loss.

When multiple lanes are enabled, each lane operates with a register subset of the overall VRF. The elements of a vector register are interleaved across all the lanes. Figure 2.2 shows a detailed example using the same configuration presented in Figure 2.1 (eight-lane configuration with an MVL of 256 elements). Lane 0 is the owner of element 0, lane 1 is the owner of element 1, lane 2 is the owner of element 2, and so on.

0	8	16	24	32	40	48	56
64	72	80	88	96	104	112	120
128	136	144	152	160	168	176	184
192	200	208	216	224	232	240	248

LANE 0

1	9	17	25	33	41	49	57
65	73	81	89	97	105	113	121
129	137	145	153	161	169	177	185
193	201	209	217	225	233	241	249

LANE 1

2	10	18	26	34	42	50	58
66	74	82	90	98	106	114	122
130	138	146	154	162	170	178	186
194	202	210	218	226	234	242	250

LANE 2

3	11	19	27	35	43	51	59
67	75	83	91	99	107	115	123
131	139	147	155	163	171	179	187
195	203	211	219	227	235	243	251

LANE 3

4	12	20	28	36	44	52	60
68	76	84	92	100	108	116	124
132	140	148	156	164	172	180	188
196	204	212	220	228	236	244	252

LANE 4

5	13	21	29	37	45	53	61
69	77	85	93	101	109	117	125
133	141	149	157	165	173	181	189
197	205	213	221	229	237	245	253

LANE 5

6	14	22	30	38	46	54	62
70	78	86	94	102	110	118	126
134	142	150	158	166	174	182	190
198	206	214	222	230	238	246	254

LANE 6

7	15	23	31	39	47	55	63
71	79	87	95	103	111	119	127
135	143	151	159	167	175	183	191
199	207	215	223	231	239	247	255

LANE 7

Figure 2.2 *VRF elements distribution for a MV=256 elements and eight-lane configuration.*

The designer can specify the VRF line size (512-bit in the example shown in Figure 2.1). Then, every read operation to the register file will return a VRF line size; for that reason, it is necessary to perform operands buffering to store the elements read and to keep a constant stream of data to the functional unit, avoiding bubbles in the pipeline.

As soon as the first result is computed, it is sent to a structure called Write-Back Buffer (WB). This structure holds the resultant data (one 64-bit element per cycle) from the functional units. Once the WB buffer gets the total elements corresponding to one

VRF line (512-bit in the example shown in Figure 2.1), the data can be written back in the VRF.

Each lane has a 64-bit bus to communicate with the Vector Memory Unit. When a load operation is performed, the Vector Memory Unit receives a complete cache line (512-bit in the example shown in Figure 2.1) and sends it in parallel to each lane (one 64-bit element). The Load Buffer (LB) is the structure in charge of collecting data from the Vector Memory Unit. Once the LB completes one VRF line (eight 64-bit elements), it proceeds to write back to the VRF. Figure 2.3 shows a more detailed example with the same configuration presented in Figure 2.1. On the vector memory unit side, there is a cache line with eight 64-bit elements. Those elements are sent in an interleaved fashion to each lane, meaning that Lane 0 is the owner of element 0, lane 1 is the owner of element 1, lane 2 is the owner of element 2, and so on. On the contrary, store operations read a complete line (512-bit) from the VRF and store it in the Store Buffer. This operation is performed at the same time in all lanes. Then, the store buffer sends one 64-bit element to the Vector Memory Unit each cycle.

When an instruction completes execution, the corresponding physical destination must be marked as ready to issue new instructions that were waiting for it. This is done in a structure called Valid-Bit, which for every physical register, one bit is added to the structure. For example, for a vector engine with 64 physical registers, a 64-bit Valid bit structure is implemented.

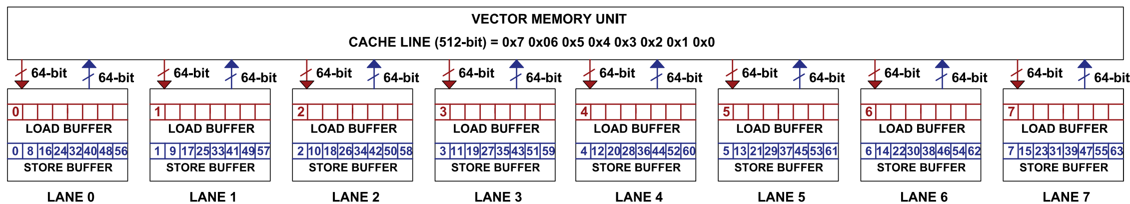

Figure 2.3 *Vector Load/Store buffer behavior*

2.1.2.5 Vector Memory Unit (VMU).

VMU receives instructions (load/store) from the memory queue, and it cannot accept a new instruction until it finishes its current work. This module is in charge of managing the requests to memory. The VMU supports unit-stride, strided, and indexed (gather/scatter) addressing modes. Vector unit-stride operations access elements stored contiguously in memory, starting from the effective base address. Vector strided operations access the first memory element at the effective base address and then access subsequent elements at address increments given by the byte offset specified by a scalar source. Vector indexed operations add the contents of each element of the

vector offset operand specified by the second vector source operand to the effective base address to give the effective address of each element.

Once it receives the memory instruction, as well as the memory address, the VL, and the stride (1 for unit-stride access), the VMU generates all the requested addresses and puts them in a FIFO. Then, the requests to memory are sent in-order in a pipeline fashion. However, the memory system can answer in a different order (hit under miss). Additionally, it is possible to set the number of Miss Status and Handling Registers (MSHRs), which for a long vector length implementation can be a relevant factor to consider. The MSHRs implement a queue that holds the list of outstanding memory requests. Each memory request is assigned to an MSHR object representing a particular memory block that has to be read or written to complete the command. When the memory request is sent, a unique order number is assigned to each read/write request as they appear on the slave port.

The memory port can be connected directly to any level of the memory hierarchy. Once a request to the memory system is sent, the timing is managed by the gem5 memory model. Several configurations can be used; for example, Figure 2.1.a shows a configuration connecting the vector memory port to L1 cache. In another configuration (Figure 2.1.b), it is possible to bypass the first-level cache to read/write from L2 cache. Vector architectures designed for long vector lengths are typically connected to L2 cache since a vector memory instruction can amortize long memory latency over many elements with a small performance degradation [49].

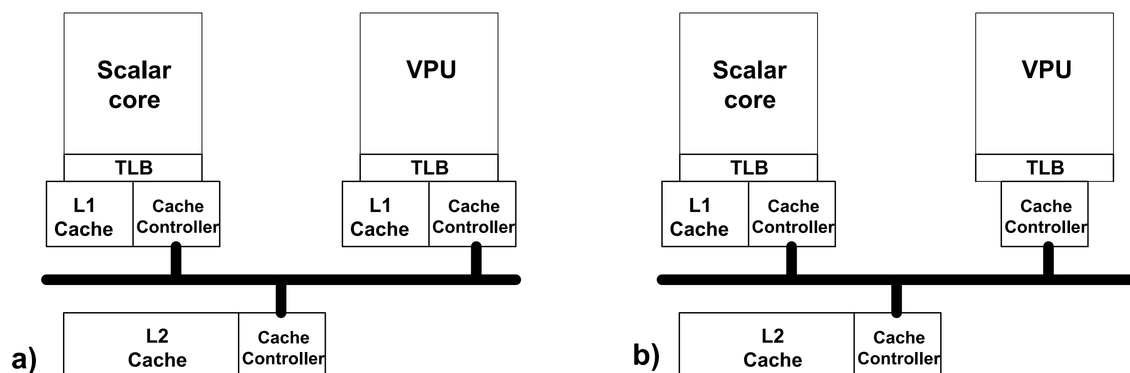

Figure 2.4 *Two possible memory configurations. a) the VPU is connected to its own L1 cache. b) the VPU is connected to L2 cache.*

gem5 provides several cache coherence protocols based on bus snooping and directory, being the MOESI snooping protocol the default configuration. Inclusion is optional. All

the results presented in this thesis have the default configuration (MOESI snooping protocol with non-inclusive caches), however, researchers studying aspects of the memory subsystem have the opportunity of exploring different coherence protocols as well as modifying the cache inclusion policy. Request from the scalar core or VPU propagate toward main memory in the following fashion: An L1 cache miss is broadcast on the local L1/L2 bus, where it is snooped by the other L1s. If there is no response, the request is serviced by the L2. If the request misses in the L2, then after some delay (currently set equal to the L2 hit latency), the L2 will issue the request on its memory-side bus, where depending on the configuration, it will possibly be snooped by other L2s and then be issued to an L3 or memory. In the case presented in Figure 2.4.b, the VPU does not include a L1 cache, however it is required to preserve the TLB to translate from virtual to physical addresses, and the coherent cache controller which is in charge of broadcast the request on the local L1/L2 bus. In the case of a vector store, the coherent cache controller will snoop each request to invalidate the other L1s blocks in case there is a copy, and then, write the corresponding data in L2.

Finally, the latency of accessing to L2 can be amortized, because a single access is initiated for the entire vector rather than to a single word. The default value of a L1 hit latency is set to 4 cycles, while L2 hit latency is set to 12 cycles. The latency of each cache can be configured, as well as the size, block size, number of MSHRs, etc. One important consideration is that caches inherently deal with unit stride data, so that while increasing block size can help reduce miss rates for large scientific data sets with unit stride, increasing block size can have a negative effect for data that is accessed with non-unit stride.

2.1.2.6 Lane Interconnection.

The vector extension can be configured with different numbers of lanes, where the lanes work fully synchronized. However, there is a class of instructions that involves communication between different vector lanes, basically for moving and addressing data such as vector *slides*, *vector reductions,* and vector *register gather* instructions. The *slide* instructions move elements up and down a vector register. The vector *reduction* instruction takes a vector register group of elements and performs a reduction using some binary operator to produce a scalar result that is written in the element 0 of a vector register. The *vector register gather* instruction reads elements from a first source vector register group at locations given by a second source vector register group and writes it in a destination vector register.

Therefore, an interconnection network is necessary to support this class of instructions. Two different interconnection networks (crossbar and ring network) are

modeled. In the example shown in Figure 2.1, the vector lanes include a ring node (router) to communicate with the neighboring node. This interconnection could limit the performance for those applications which make intensive use of the lane interconnection, but it is cheap in terms of area. On the contrary, the crossbar interconnection could achieve excellent performance, but it implies a considerable increase in area.

2.1.2.7 Capabilities and limitations.

Four versions (0.7, 0.8, 0.9, and 1.0-rc) have been released so far regarding the RISC-V vector extension. However, between the four different versions, the changes are small. For sure, there will be more updates before the specification is frozen as an official release, and it is believed that point is close. In that sense, this work has started to add vector extension support to gem5. However, the full specification is not implemented, leaving as future work the implementation of atomic operations, permutation operations, register grouping, and exception handling for the full system mode.

2.1.2.8 Early Access.

Progress is being made on integrating the Vector Architecture model on the official gem5 repository [50]. It is possible to get early access by cloning [51], a fork of the official gem5 repository that includes the Vector Architecture model.

2.2 The McPAT framework

Designing a vector architecture can be a complex task. Choosing parameters such as the MVL, the number of physical registers, the number of lanes, the number of pipelines within the lane, the number of memory ports, are some of the main decisions that the architect must take in earlier steps. Additionally, all of these parameters must be considered not only taking into account performance metrics but also the different requirements/limitations of the project, such as area and power. In that sense, the gem5 simulator extended with our parameterizable vector architecture model covers the performance metrics. However, a model is still missing that helps obtain a first approximation of the area and power cost that these choices can imply once implemented at RTL level, which will help the architect make accurate decisions in a shorter time.

McPAT [52] is an architectural integrated power, area, and timing modeling framework, that focuses on power and area modeling, with a target operational frequency as a design constraint. McPAT supports comprehensive design space exploration for multicore and manycore processor configurations ranging from 90nm to 22nm. At the microarchitectural level, McPAT includes models for the fundamental components of a chip multiprocessor, including in-order and out-of-order cores, networks-on-chip, cache hierarchy, integrated memory controllers, and multiple-domain clocking. At the circuit and technology levels, McPAT supports critical-path timing modeling, area modeling, and dynamic, short-circuit, and leakage power modeling for bulk CMOS, SOI, and double-gate transistors. Additionally, McPAT has a flexible XML interface to facilitate its use with many performance simulators such as gem5.

Combined with a performance simulator, McPAT enables architects to consistently quantify the cost of new ideas and assess the tradeoffs of different architectures. In that sense, in this research work we decided to extend the McPAT framework with a VPU model which can be configured via the XML interface. Then, the reports obtained from gem5 simulator can be ported to McPAT framework to evaluate the cost of the different VPU configurations.

As mentioned before, McPAT includes models for the fundamental components such as integer and floating-point functional units or memory structures that are used for defining modules such as the scalar register file. These components are reused for the VPU model, and via the XML interface, we can setup their specific configuration.

Then, our McPAT VPU model presented in Figure 2.5 can be configured with the following parameters:

Number of lanes. The model can be configured as a unique lane, with several functional units inside (SIMD Multimedia style), or a multilane design.

Number of integer functional units per lane. The model can be configured with or without integer functional units. The value is for only one lane, and will be replicated in case a multilane design.

Number of floating-point functional units per lane. The model can be configured with or without floating-point functional units. The value is for only one lane, and will be replicated in case a multilane design.

Maximum Vector Length. The MVL in bits supported by the microarchitecture.

Number of Vector Physical Registers. Number of vector physical registers implemented in the VRF. The VRF size in bits is calculated by multiplying the MVL by the Number of Vector Physical Registers.

Number of banks per lane (VRF). In the case of multi-lane or single-lane configurations, each lane has a slide of the VRF. However, still it is possible to divide the VRF slice into multiple memory banks. This can be an important consideration when implementing a large VRF. Then by using smaller memory structures can be possible to achieve higher working frequencies.

Number of VRF read ports. The number of read ports for every VRF slice.

Number of VRF write ports. The number of write ports for every VRF slice.

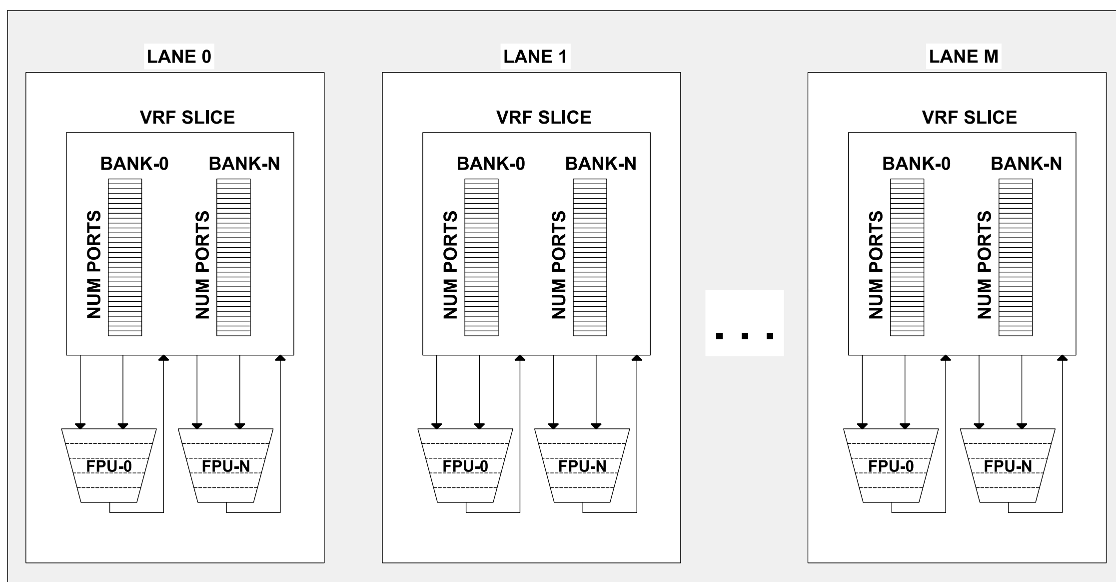

Figure 2.5 *Vector architecture model implemented on the McPAT framework.*

The previously described parameters allow modeling a wide diversity of VPU designs. For example, a traditional cray-style vector architecture for long vectors or a SIMD-Multimedia style design for short vectors. Additionally, since the VRF represents a key element inside a VPU model, it is important to allow to model different hardware configurations. For example, modeling multi-ported VRF is crucial, due to increasing the number of ports has a super-linear impact on the power/area results, as demonstrated by Arima et al. [53] and Zyuban et al. [54]. Furthermore, the McPAT model can infer multi-ported memory structures with more than two ports. However, those custom structures are usually not provided by the synthesis tools, where memory compilers usually offer a variety of memory structures varying between one and two access ports, targeting low density or high-performance implementations. In that sense, modeling techniques such as the LVT technique [55], which provides multi-ported designs at the cost of replicating and banking dual-port memories is also allowed by our model, allowing to get accurate results in case implementing at RTL level a similar technique.

2.3 The RiVEC Benchmark Suite

The RiVEC Benchmark Suite is a collection composed of ten data-parallel applications from different domains. The suite focuses on benchmarking vector microarchitectures; nevertheless, it can also be used for Multimedia microarchitectures. Applications are Vector Length Agnostic; therefore, applications can be tested using short, medium, and large MVL implementations.

When we say that the applications are data-parallel, it does not mean that all applications have high data-level parallelism. Data-level parallelism can be found at different levels (low/medium/high data-level parallelism). This property is essential in order to have different scenarios, which can be limited having only high data-parallel applications.

The current implementation targets RISC-V Architectures; however, it can be easily ported to any Vector/SIMD ISA. It includes a wrapper library that maps vector intrinsics and math functions to the target architecture.

In order to select the final applications, a study of different benchmarks suites was performed taking into account the following criteria:

- *Applications from different domains*. Although the vector architectures have traditionally been applied to the supercomputing domain, this suite does not explore a single application domain, as was done by several benchmark suites.
- *Applications with different degree of data-level parallelism*. Having different degree of data-level parallelism helps to have different scenarios. While some vector architectures designs can take advantage of high data-level parallelism found in the application, these architectures could poorly execute some kind of application that presents low data-level parallelism. This property is attractive because it allows us to find the weaknesses of some proposals/designs.
- *Cover most of the Vector ISA*. Found applications that use almost all vector ISA is difficult. However, we tried to cover this by selecting applications with different instruction uses. For example, we found applications in which most of the instructions are memory operations, or applications where arithmetic operations consume most of the execution time. Furthermore, in vector architectures, there are new instructions compared with scalar instructions, for example, the element manipulation instructions as the slide operations or operations with masks. We found some applications which are very intensive using these kinds of instructions.

The RiVEC Benchmark Suite is available to the computer architecture community to evaluate vector architecture designs. It is openly available at GitHub [40].

2.3.1 Study of existing Benchmark Suites

In order to select the applications to be part of the RiVEC Benchmark Suite, a study of the existing benchmark suites for computer architecture was performed. As mentioned before, the initial goal was to find applications from different domains, with different degree of data-level parallelism, and cover most of the Vector ISA. The most significant benchmarks suites for this work are described below.

PARSEC. The Princeton Application Repository for Shared-Memory Computers (PARSEC) [44] is a benchmark suite intended for research, and it is composed of multithreaded programs. The suite focuses on emerging workloads and was designed to be representative of next-generation shared-memory programs for chip-multiprocessor. This benchmark suite has diversity in its applications, and we believe that this property is ideal for our research since we wish to have different scenarios, in which we can test different microarchitectural ideas. This suite is originally focused on multithreaded architectures, mainly exploiting task-level parallelism. The current version (PARSEC 3.0) of the suite contains 13 programs from many different areas such as computer vision, video encoding, financial analytics, animation physics, and image processing.

ParVec [47] [56] is a vectorized version of the PARSEC benchmark suite. ParVec can target SSE, AVX, and NEON SIMD architectures by means of custom vectorization and math libraries. The performance and energy efficiency improvements from vectorization depend greatly on the percentage of code that can be vectorized. The ParVec benchmark suite is available for the research community.

Rodinia [46] is a benchmark suite for heterogeneous computing. It was developed to help architects to study emerging platforms. Rodinia includes applications and kernels which target multicore CPU and GPU platforms using OpenMP, OpenCL, and CUDA implementations. The choice of applications was inspired by Berkeley's dwarf taxonomy.

Polybench [43] is a high DLP benchmark suite composed by 30 numerical computations with static control flow, extracted from operations in various application domains (linear algebra computations, image processing, physics simulation, dynamic programming, statistics, etc.).

It can be seen that there are many suites for parallel computing on general-purpose CPU architectures. Also, there are others for heterogeneous computing, covering MPI, OpenMP, OpenCL, and CUDA programming models. SIMD Suites are very limited. However, suites for vector architectures fall into a gap that is not covered by previous

benchmark development. The Vectorized Benchmark Suite is released to address this concern.

The next sections describe the followed methodology to vectorize the programs. A short description of the code is shown for each vectorized program, also, an analysis before and after the vectorization, and finally, the obtained results after being executed on the implemented model of the RISC-V Vector Accelerator are presented.

2.3.2 Methodology

The methodology used in order to obtain a vectorized suite has the following steps:

Profiling. Profiling is needed to identify the functions that are time-consuming inside an application. In order to do profiling, we used the Linux GCC profiling tool gprof.

Kernel Analysis. We need to analyze the functions inside the application that consume most of the execution time, and if they are vectorizable. There are simple ways to see if a potential vectorizable function exists. For example, if the function contains loops, if there are dependencies inside the loop, the kind of memory request needed (strided, indexed), etc. Once the analysis ends, it is possible to identify those applications that can be improved after being vectorized.

Kernel Vectorization. In order to write vector code, we use the available vector builtins reference from the compiler team in EPI project. We also develop a math library with the math functions needed by the suite, like exponential and logarithm functions. The written code is compatible with different VLs, by simply compiling with the flag -DPARSEC_USE_$4, where $4 is the MVL.

Simulator Verification. In this step, we compare the results given by the scalar code compiled for RISC-V architectures versus the vector implementation. This task is accomplished using the Spike ISA simulator. Spike is a RISC-V ISA simulator that implements a functional model of one or more RISC-V harts. Spike is fast due to being only functional, and it is enough to check if the vector implementation gives the same results as the scalar one.

Performance Analysis. Once the applications are verified, it is time to analyze the timing results. This task must be done using the gem5 simulator, which provides a detailed timing model of a system, including a RISC-V vector extension and the scalar core. This task usually takes several hours for some small input datasets or days for some large input datasets.

2.3.3 Vectorized applications of the RiVEC Benchmark Suite

This section presents ten vectorized applications. According to this first study, these benchmarks contain different degree (high, medium, or low) of data-level parallelism. Furthermore, the applications feature different characteristics allowing the evaluation of different modules in a vector architecture.

Table 1 presents general information about the ten applications selected for the RiVEC Benchmark Suite. The RiVEC benchmark suite is composed by several application domains as well as different algorithmic models, covering several scenarios which can be found in modern applications. Additionally, applications can be categorized as having regular DLP, irregular DLP, or a mix of both. On the one hand, well-structured data access with regular and well-known address streams, including well-structured control flow corresponding to a regular DLP; and on the other hand, less-structured data access with dynamic and difficult to predict address streams, and less structured control flow representing irregular DLP [57].

Table 1. *RiVEC Benchmark Suite.*

Application	Application Domain	Algorithmical Model	DLP Pattern	Benchmark Suite
Axpy	HPC	BLAS	Regular	-
Blackscholes	Financial Analysis	Dense Linear Algebra	Regular	PARSEC/PARVEC
Canneal	Engineering	Unstructured Grids	Irregular	PARSEC/PARVEC
LavaMD2	Molecular Dynamics	N-Body	Regular	Rodinia
Jacobi-2D	Engineering	Dense Linear Algebra	Regular	PolyBench
Particle-Filter	Medical Imaging	Structured Grids	Mix	Rodinia
Pathfinder	Grid Traversal	Dynamic Programming	Regular	Rodinia
Somier	Physics Simulation	Dense Linear Algebra	Regular	-
Streamcluster	Data Mining	Dense Linear Algebra	Mix	PARSEC/PARVEC
Swaptions	Financial Analysis	MapReduce	Regular	PARSEC/PARVEC

Table 2 presents detailed characteristics for every application, such as the supported VL and the memory access pattern; it is also indicated which applications stress the different vector microarchitecture modules such as the lane functional units or lane interconnection network; the final row shows if the application has intense communication with the scalar core, these applications feature a tight mixture of scalar and vector operations and accesses.

All the applications (C and C++ programs) were extended by adding the RISC-V vectorized version. The implementations make use of intrinsics, and the code was written in a Vector Length Agnostic fashion, meaning that the same binary can be executed in different Vector Engine configurations with any modification.

Table 2. *Application characteristics.*

Application		Axpy	Blackscholes	Canneal	Jacobi-2D	LavaMD2
Vector Length	Short	✓	✓	✓	✓	✓
	Medium	✓	✓	✓	✓	✓
	Large	✓	✓		✓	
Memory Unit	Unit-stride	✓	✓		✓	✓
	Strided					✓
	Indexed			✓		
Vector Lane	Arithmetic	✓	✓	✓	✓	✓
	Mask		✓			
Interconnection Network	Slides				✓	
	Reductions			✓		✓
Intensive Comm. with the Scalar core				✓		

Application		Particle - Filter	Pathfinder	Somier	Streamcluster	Swaptions
Vector Length	Short	✓	✓	✓	✓	✓
	Medium	✓	✓	✓	✓	✓
	Large	✓	✓	✓		✓
Memory Unit	Unit-stride	✓	✓	✓	✓	✓
	Strided					
	Indexed	✓			✓	
Vector Lane	Arithmetic	✓	✓	✓	✓	✓
	Mask	✓			✓	✓
Interconnection Network	Slides		✓	✓		
	Reductions	✓			✓	✓
Intensive Comm. with the Scalar core		✓			✓	

It is important to mention that several efforts are being made by the community to include the support for the new vector standard to the compiler. However, when we open source the first version of the benchmark suite to the community (September 2020), the support was at an initial stage, limiting use to assembly code or intrinsics at best. In that sense, at that point, it was not possible to use auto-vectorization to obtain a different set of instructions. Having said that, all the applications (C and C++ programs) were extended by adding the RISC-V vectorized version. The implementations make use of intrinsics, meaning that the compiler will substitute the intrinsic by a sequence of predefined vector instructions. In that sense, the vector compiler only takes the decision to insert spill code (vector load/store) when the number of vector registers is not

sufficient, and vector move instructions when a vector register is used as an argument in a function. Also, the code was written in a Vector Length Agnostic fashion, meaning that the same binary can be executed in different Vector Engine configurations with any modification. By August 2021, auto-vectorization is supported in the vector compiler, but still limited. Auto-vectorization was tested with all the applications. However, the results showed that still more work is needed to fully rely on the compiler, since our hand-vectorized version achieve better percentages of vectorization in all the cases.

For all the applications, five input sets are available: tiny, small, medium, large, and native. Tiny data sets allow fast explorations of new features which takes seconds to be completed. Most of the applications for this data set are compatible only with short vectors. Small data sets provide short simulations which take a few minutes to be completed. In this scenario, most of the applications provide the opportunity to explore short, medium, and long vector lengths implementations but with few iterations. Medium data sets provide simulations that can be completed in a few hours. Large data sets provide simulations that take between 8 hours up to 48 hours. Finally, native data sets represent huge data sets defined only to be used on native machines.

An initial analysis of these programs shows that some of them present high data-level parallelism together with a high percentage of vector arithmetic instructions (i.e. *blackscholes*), and for these programs, it is usual that vector processors achieve excellent computational throughput. However, other programs present limited data-level parallelism (i.e. *canneal*), and for these programs is normal that long vector length processors don't fully utilize their big structures (i.e. vector register file). This kind of application is interesting because it is possible to study how to better exploit the microarchitecture to achieve better computational throughput for these programs. Furthermore, there are some other benchmarks that present high data-level parallelism together with data element manipulation vector instructions (i.e. *pathfinder)*; this feature allows to evaluate different interconnection topologies (ring, crossbar, etc.) between multiple lanes in a vector architecture. And finally, applications that make intensive use of masked operations, also special operations with masks that send resultant data to the scalar core are interesting to evaluate (i.e. *particlefilter*).

All results discussed in this document were performed using the large input sets. The next subsections describe how the vectorized versions were implemented. Furthermore, it is discussed the degree of the vectorization achieved and how it could lead us to get some initial insights about expected performance.

2.3.3.1 Axpy

Axpy is a Basic Linear Algebra Subprogram (BLAS). This function implements the function $Y = A * X + Y$ where X and Y are vectors, while A represents a scalar value. Then, computations are based on fused multiply-add operations, one for each element of the input vectors. Because of its low arithmetic intensity, Axpy is a typical memory-bound kernel. Table 3 shows the five inputs sets available for Axpy.

Table 3. *Axpy input data sets*

Simulation size	Value
simtiny	vector size in Kilo elements = 8
simsmall	vector size in Kilo elements = 128
simmedium	vector size in Kilo elements = 512
simlarge	vector size in Kilo elements = 2,048
native	vector size in Kilo elements = 65,536

Degree of vectorization. Table 4 presents some statistics of the Axpy application for both the scalar and the vectorized implementations. The analysis for the Vectorized code takes into account six different MVL configurations: short-vectors (MVL=8 and 16 elements), medium-size vectors (MVL=32 and 64 elements), and long-vectors (MVL=128 and 256 elements). *Total Instructions* represent the number of executed instructions (*Scalar Instructions* + *Total Vector Instructions*). *Scalar Instructions* represent only the instruction executed by the scalar core. *Vector Memory Inst* represents the *Vector Loads* + *Vector Stores*. *Vector Arithmetic Inst* represents all the arithmetic instructions executed in the functional units, either integer or floating-point. *Vector to Scalar Inst* represents those instructions that move one element from a vector register to the scalar core. *Vector Elem Manipulation Inst* represents the *Vector Slides* + *Vector Reductions*. *Total Vector* Instructions represent the instructions executed by the vector engine (*Vector Memory Inst* + *Vector Arithmetic Inst* + *Vector to Scalar Inst* + *Vector Elem Manipulation Inst*). *Vector Operations* represents the number of operations performed by the vector instructions, while scalar instructions perform only one operation per instruction, vector instructions perform VL operations per vector instruction. *% of vectorization* is defined as the ratio of *Vector Operations* over the total number of operations (*Scalar Instructions* + *Vector Operations*). Thus, all previous data plus the *Average VL* give us an idea if most of the hardware resources will be used through the program execution. The same table structure is used for all applications in this study.

***Table 4**. Instruction Level Characterization of Axpy application.*

	Scalar Code	Vectorized Code		
		MVL=8 elements	MVL=16 elements	MVL=32 elements
Total Instructions	134,217,738	25,165,839	12,582,927	6,291,471
Scalar Instructions	134,217,738	16,777,231	8,388,623	4,194,319
Vector Memory Inst		6,291,456	3,145,728	1,572,864
Vector Loads		4,194,304	2,097,152	1,048,576
Vector Stores		2,097,152	1,048,576	524,288
Vector Arithmetic Inst		2,097,152	1,048,576	524,288
Vector to Scalar Inst		0	0	0
Vector Elem Manipulation Inst		0	0	0
Vector Slides		0	0	0
Vector Reductions		0	0	0
Total Vector Instructions		8,388,608	4,194,304	2,097,152
Total Vector Operations		67,108,864	67,108,864	67,108,864
% of Vectorization		80%	89%	94%
Average VL (elements)		8	16	32
SA-speedup		1.6000	1.7778	1.8824
		Vectorized Code		
		MVL=64 elements	MVL=128 elements	MVL=256 elements
Total Instructions		3,145,743	1,572,879	786,447
Scalar Instructions		2,097,167	1,048,591	524,303
Vector Memory Inst		786,432	393,216	196,608
Vector Loads		524,288	262,144	131,072
Vector Stores		262,144	131,072	65,536
Vector Arithmetic Inst		262,144	131,072	65,536
Vector to Scalar Inst		0	0	0
Vector Elem Manipulation Inst		0	0	0
Vector Slides		0	0	0
Vector Reductions		0	0	0
Total Vector Instructions		1,048,576	524,288	262,144
Total Vector Operations		67,108,864	67,108,864	67,108,864
% of Vectorization		97%	98%	99%
Average VL (elements)		64	128	256
SA-speedup		1.9394	1.9692	1.9845

Axpy is composed of two main tasks. The first one is the initialization phase which has been omitted since we focus only on the *axpy_ref* function, which represents the region of interest (ROI). This function basically computes fused multiply-add operations. The vectorization of this function is straightforward since it presents a very regular DLP. The vectorized function is composed of two vector loads, one vector fused multiply-add, and finally, one vector store.

The *Total Instructions* drops considerably for the vectorized versions, achieving *% of Vectorization* from 80% for MVL=8 elements, up to 99% for MVL=256 elements. This is not only because one vector instruction represents many scalar operations of the same type (i.e., a vadd.v instruction for a configuration of VL=256 elements represents 256 64-bit scalar add instructions), but also it removes many control instructions needed to execute the desired number of operations. All the scalar instructions needed to write the "for" loop, or the data movement from/to memory produced by the limited number of physical scalar registers are removed. As the VL increases, the percentage of vectorization increases because of the ratio resulting from the number of *Total Vector Operations*, which remains equal, over the number of the total operations (*Scalar Instructions* + *Total Vector Operations*), which decreases as a result of the reduction of the number *Scalar Instructions*, as shown in Table 4.

One important difference between the scalar and the vector execution is that the execution of the vector operations can be pipelined because all vector operations of one vector instruction are independent. So benefits from vectorization are two-fold: reduction in the total number of operations and faster execution of vector operations thanks to pipelining. Then, as the MVL is increased, it is possible to hide the latency of individual operations.

From previous data, it is possible to get an initial insight about the expected performance when executing the program in the vector architecture model. For example, obtaining the ratio between the number of the *Total instructions* of the serial version divided by the total number of operations (*Vector Operations* + *Scalar Instructions*), obtains a Static Analysis speedup (SA-speedup) from 1.6x for MVL=8 elements up to 1.98x for MVL=256 elements, showing that it is possible to achieve better performance by only increasing the MVL.

Several factors can influence reducing or increasing this speedup. The execution of the remaining scalar instructions can be amortized underneath vector execution or the use of parallel lanes. In the case of the use of parallel lanes, Axpy could not be a factor in increasing performance since it is a memory-bound kernel. Then, as an initial conclusion, it can be said that for larger MVLs, it is possible to achieve higher speedups than 1.6X. As the number of lanes is increased, no speedup increase would be expected.

2.3.3.2 Blackscholes

The Blackscholes application calculates the prices for a portfolio of European options analytically with the Black-Scholes partial differential equation (PDE). This application represents the wide field of analytic PDE solvers in general and their application in computational finance in particular. This benchmark is an iterative data-

parallel computation, and the performance is limited by the amount of floating-point operations that the processor can perform. This application was taken from PARSEC, and a detailed description can be found in [44].

The application starts reading the portfolio with a defined number of options (numOptions) from a text file which provides the initialization and the control reference values, and store it in a structure (OptionData). The inputs for Blackscholes are sized as shown in Table 5.

Table 5. *Blackscholes input data sets*

Simulation size	Value
simtiny	16 options
simsmall	512 options
simmedium	16,384 options
simlarge	65,536 options
native	10,000,000 options

The function BlkSchlsEqEuroNoDiv computes basic floating-point operations, including fadd, fsub, fmul, fdiv, fsqrt, also logarithmic and exponential functions. Inside this function, the function CNDF (Cumulative Normal Distribution Function) is called a couple of times and finally, the output price (OptionPrice) is selected according to the type of option (otype) inside an "if" statement. Inside the function CNDF, basic floating-point operations are performed, and one call to an exponential function. Also, there are some "if" statements in order to check for negatives values at the inputs and to select the output (OutputX) according to the sign.

And finally, the program writes the prices for all options to the output file prices.txt. If error checking was enabled at compile-time it also compares the result with the reference price.

The runtime profile of the original code (scalar code) shows that around 12% of the total runtime is used to read the portfolio and store it in the structure called OptionData. Around 85% of the runtime is used to compute the BlkSchlsEqEuroNoDiv function including CNDF, where most of the time is spent computing the logarithmic and exponential functions.

Degree of vectorization

Table 6 presents some statistics of the Blackscholes application for both the scalar and the vectorized implementations. Blackscholes is a regular DLP application where there are no dependencies between each price computation. The runtime profile of the

scalar code shows that around 12% of the total runtime is spent in the initialization phase, which is not vectorizable. This task executes 573,256,509 scalar instructions which are not taken into account in the results presented in Table 6 to present the % Vectorization only of the region of interest (ROI).

Table 6. *Instruction Level Characterization of the Blackscholes application.*

	Scalar Code	Vectorized Code		
		MVL=8 elements	MVL=16 elements	MVL=32 elements
Total Instructions	3,180,479,876	139,264,913	69,632,913	34,816,913
Scalar Instructions	3,180,479,876	54,477,713	27,239,313	13,620,113
Vector Memory Inst		2,867,200	1,433,600	716,800
Vector Loads		2,457,600	1,228,800	614,400
Vector Stores		409,600	204,800	102,400
Vector Arithmetic Inst		81,920,000	40,960,000	20,480,000
Vector to Scalar Inst		0	0	0
Vector Elem Manipulation Inst		0	0	0
Vector Slides		0	0	0
Vector Reductions		0	0	0
Total Vector Instructions		84,787,200	42,393,600	21,196,800
Total Vector Operations		1,356,595,200	1,356,595,200	1,356,595,200
% of Vectorization		96%	98%	99%
Average VL (elements)		8	16	32
SA-speedup		2.2539	2.2983	2.3212
		Vectorized Code		
		MVL=64 elements	MVL=128 elements	MVL=256 elements
Total Instructions		17,408,913	8,704,913	4,352,913
Scalar Instructions		6,810,513	3,405,713	1,703,313
Vector Memory Inst		358,400	179,200	89,600
Vector Loads		307,200	153,600	76,800
Vector Stores		51,200	25,600	12,800
Vector Arithmetic Inst		10,240,000	5,120,000	2,560,000
Vector to Scalar Inst		0	0	0
Vector Elem Manipulation Inst		0	0	0
Vector Slides		0	0	0
Vector Reductions		0	0	0
Total Vector Instructions		10,598,400	5,299,200	2,649,600
Total Vector Operations		1,356,595,200	1,356,595,200	1,356,595,200
% of Vectorization		99%	99%	99%
Average VL (elements)		64	128	256
SA-speedup		2.3327	2.3386	2.3415

Around 85% of the runtime is used to compute the BlkSchlsEqEuroNoDiv and CNDF functions which correspond to the ROI. In this code section, basic floating-point

operations are computed, including fadd, fsub, fmul, fdiv, fsqrt, as well as logarithmic and exponential functions. The vectorization of these functions was straightforward since it presents a very regular DLP. Furthermore, logarithmic and exponential functions were also vectorized since most of the time is spent computing these functions.

The total number of instructions drops considerably for the vectorized versions, achieving *% of Vectorization* from 96% for MVL=8 elements up to 99% for MVL=256 elements. As the VL increases, the *% of Vectorization* increases because of the ratio resulting from the number of *Total Vector Operations*, which remains equal, over the number of the total operations (*Scalar Instructions* + *Total Vector Operations*), which decreases as a result of the reduction of the number *Scalar Instructions*, as shown in Table 6.

From previous data, it is possible to get an initial insight about the expected performance. In this case, the ratio between the number of the *Total instructions* of the serial version divided by the total number of operations (*Vector Operations* + *Scalar Instructions*), obtains a SA-speedup from 2.25x for MVL=8 elements up to 2.34x for MVL=256 elements, showing that it is possible to achieve better performance by only increasing the MVL.

Several factors can influence reducing or increasing this speedup. The execution of the remaining scalar instructions can be amortized underneath vector execution or the use of parallel lanes. Increasing the number of lanes, would be expected to obtain a linear speedup increase since it is a high DLP application. Then, as an initial conclusion, it can be said that for larger MVLs, it is possible to achieve higher speedups than 2.25X. Contrary to the Axpy application, Blackscholes represents a compute-bound application. Then, as the number of lanes increases, obtaining a linear speedup increase would be expected, especially for larger MVL configurations where the % of Vectorization is higher. Then, the SA-speedup presented in Table 6 can be multiplied by the number of lanes as a very initial insight.

2.3.3.3 Canneal

The Canneal application is focused on minimizing the routing cost of a chip design using cache-aware simulated annealing (SA). SA is a metaheuristic to approximate global optimization in a large search space for an optimization problem. This application is representative of engineering workloads and features fine-grained parallelism with lock-free synchronization and pseudo-random worst-case memory access pattern. As presented in Table 1, Canneal represents applications with irregular DLP because of its less-structured data accesses with dynamic and difficult to predict address streams, and

less structured control flow. This application was taken from PARSEC, and a detailed description can be found in [44].

The application starts creating an array of locations (*location_t*) of size given by the chip size parameter and initializes it with coordinates (location x, location y). Then, the reading of the netlist is performed, and it is stored in an array of type *netlist_elemt,* which is the class that contains all information about the current element (name, inputs, outputs and a pointer to the present location). The locations are assigned in order, meaning that the first element has a pointer to the location "x=0, y=0", next element to the location "x=0, y=1", and so on until all the elements in the netlist are assigned. All this task consumes near of the 67% of the total execution time for simlarge input set, and decreases as the input size increase; i.e. for a native input set this task consumes near of the 15% of the total execution time.

The annealing algorithm is implemented in the *Run* function of the *annealer_thread* class. There are four key tasks. The first one is to pick a pair of elements from the netlist using the Mersenne Twister pseudo-random number generator. The second one is to compute the difference of the routing cost (*calculate_delta_routing_cost and swap_cost*) for the two cases: the elements in the original location and the elements swapped. *Swap_cost* function consumes most of the kernel's execution time. The third one is to evaluate the result in the function "*accept_move*", if the result of the total routing cost is negative, it means that the routing cost is smaller and the function proceeds to swap the elements; otherwise, it evaluates the change in cost and the current temperature to decide whether the swap should be performed or not. And finally, the fourth one, accepted swaps are executed by calling "*swap_locations"* which swap the pointers of the locations. The application ends calling the function *total_routing_cost* which calculates the final routing cost of the entire chip using the Manhattan distance formula. In order to increase data reuse, the algorithm keeps one element of the previous iteration and choose randomly one new element, which reduces cache misses notably.

The inputs for Canneal are sized as shown in Table 7.

Degree of vectorization

The candidate function to vectorize is *swap_cost*, which consumes most of the execution time without taking into account the initialization which takes 20.4%. This is potentially a very vectorizable function because it is composed of a couple of *"for"* loops in which three basic operations are performed: subtraction, absolute value, and addition. However, the data needed to perform these operations are the locations (coordinate x and y) of each input and output of the picked nodes; but these locations are not

contiguous data in memory. The consequence is the need to use vector indexed load instructions, which are very costly operations. Furthermore, to use the vector indexed load operations, it is necessary to create the vector of pointers to each input/output element to have access to the pointer of the current element location. Creating it in every iteration is not good for the performance. Therefore, the class *netlist_elem* was expanded with a new array of pointers called *fan_locs* that stores the pointers to all inputs and outputs of each element; it is created in the initialization phase.

Table 7. *Canneal input data sets*

Simulation size	Value
simtiny	100 swaps per temperature step, 300º start temperature, 100 netlist elements, 8 temperature steps
simsmall	10,000 swaps per temperature step, 2000º start temperature, 100,000 netlist elements, 32 temperature steps
simmedium	15,000 swaps per temperature step, 2000º start temperature, 200,000 netlist elements , 64 temperature steps
simlarge	15,000 swaps per temperature step, 2000º start temperature, 400,000 netlist elements, 128 temperature steps
native	15,000 swaps per temperature step, 300º start temperature, 2,500,000 netlist elements, 6000 temperature steps

Once the new array of pointer is added to the original code, the function *swap*_cost can be vectorized easily, by first loading the the *fan_locs* arrays of the picked nodes, and then taking those loaded addresses to perform a couple of load indexed operations. Computing the routing cost is vectorized by using vector arithmetic instructions and reduction operations, sending the final result to the core to compute the final routing cost and deciding if it should be swapped or should not.

Table 8 presents some statistics for the Canneal application. This application presents big differences with respect to the previous ones. As is shown, it presents a very low percentage of vectorization ranging from 62% for MVL=8 elements up to 93% for MVL=256 elements. While in the other applications the defined VL remained constant throughout the execution, in this case, the VL depends on the number of inputs and outputs for each pseudo-random picked element, varying from 0 to 22 connections in each element for the simlarge input set. Meaning that the larger VL in this application is 44 32-bit elements (1408-bit) because there is a maximum of 22 connections per element and two coordinates per connection. Then, it is expected that for configurations with MVL=32 elements or bigger, the instruction count remains equal, as shown in Table 8. However, even that the instruction count is equal, configurations with MVL=64, 128, and 256 elements show higher *Average VL* values which also impacts on the number of *Total Vector Operations*. This is because there are vector copy instructions introduced by the compiler to back up the content of some vector registers. Copying vector registers is a

common optimization introduced by the compiler when it is needed to save the content of a register before being modified since it can be used in the next iterations. However, the compiler typically copies the whole vector register (VL = MVL), although the effective VL is much less than the vector register width for applications with short vectors. This is done in this way since proving that elements past current VL will not be read is difficult for the compiler. In that sense, short vector applications which present this behavior can incur in performance overhead as well as wasting energy.

Table 8. *Instruction Level Characterization of the Canneal application.*

	Scalar Code	**Vectorized Code**		
		MVL=8 elements	MVL=16 elements	MVL=32 elements
Total Instructions	1,244,087,460	831,224,337	772,912,250	772,478,954
Scalar Instructions	1,244,087,460	741,393,809	710,664,805	710,395,893
Vector Memory Inst		21,499,311	15,467,022	15,422,190
Vector Loads		18,766,967	12,734,678	12,689,846
Vector Stores		2,732,344	2,732,344	2,732,344
Vector Arithmetic Inst		57,401,837	41,315,733	41,196,181
Vector to Scalar Inst		5,464,690	5,464,690	5,464,690
Vector Elem Manipulation Inst		5,464,690	5,464,690	5,464,690
Vector Slides		0	0	0
Vector Reductions		5,464,690	5,464,690	5,464,690
Total Vector Instructions		89,830,528	67,712,135	67,547,751
Total Vector Operations		1,235,169,760	1,576,423,143	2,144,641,094
% of Vectorization		62%	69%	75%
Average VL (elements)		6.87	11.64	15.87
SA-speedup		0.6294	0.5453	0.4366

		Vectorized Code		
		MVL=64 elements	MVL=128 elements	MVL=256 elements
Total Instructions		772,478,954	772,478,954	772,478,954
Scalar Instructions		710,395,893	710,395,893	710,395,893
Vector Memory Inst		15,422,190	15,422,190	15,422,190
Vector Loads		12,689,846	12,689,846	12,689,846
Vector Stores		2,732,344	2,732,344	2,732,344
Vector Arithmetic Inst		41,196,181	41,196,181	41,196,181
Vector to Scalar Inst		5,464,690	5,464,690	5,464,690
Vector Elem Manipulation Inst		5,464,690	5,464,690	5,464,690
Vector Slides		0	0	0
Vector Reductions		5,464,690	5,464,690	5,464,690
Total Vector Instructions		67,547,751	67,547,751	67,547,751
Total Vector Operations		3,280,287,658	5,553,691,653	10,100,499,642
% of Vectorization		82%	89%	93%
Average VL (elements)		24.28	41.10	74.76
SA-speedup		0.3122	0.1988	0.1151

The final performance can be highly affected by the previous behavior. At this moment, our vector compiler cannot take care of this special case. We cannot generalize this result as a bad quality code since if we execute the binary on a short vector implementation, this will not expose this behavior. However, executing in a long vector implementation will lead to executing more operations even than the scalar version. Then, from previous data, the initial insight about the expected performance is the following. The ratio between the number of the *Total instructions* of the serial version divided by the total number of operations (*Vector Operations* + *Scalar Instructions*), obtains a SA-speedup of 0.62x for MVL=8 elements and decreases up to 0.11x for MVL=256 elements. Finally, Increasing the number of lanes can increase performance since the percentage of arithmetic operations is high. However, we can anticipate that the best hardware could be a multilane design for short vectors.

2.3.3.4 Jacobi-2D

Jacobi-2D is an iterative algorithm for determining the solutions of a diagonally dominant system of linear equations. This solver is often used in computational science as part of scientific and engineering applications. This application was taken from PolyBench, and a detailed description can be found in [43].

The inputs for Jacobi-2D are sized as shown in Table 9.

Table 9. *Jacobi-2D input data sets*

Simulation size	Value
simtiny	matrix size 32x32, 8 iterations
simsmall	matrix size 128x128, 64 iterations
simmedium	matrix size 256x256, 100 iterations
simlarge	matrix size 256x256, 2000 iterations
native	matrix size 1024x1024, 25,000 iterations

Degree of vectorization

The Jacobi solver is a very interesting application because it can be vectorized by using vector arithmetic operations, vector memory instructions, and vector element manipulation instructions. In that sense, not only the Lanes and the Vector Memory Unit are evaluated, but also the interconnection between the lanes. These vector element manipulation instructions are *vslide1up.v* and *vslide1down.v*, which move elements one position up and down a vector register. It is possible to load a fraction of one row and operate on it by applying *vslide1up.v* to obtain the left neighboring nodes and *vslide1down.v* to obtain the right neighboring nodes. Once the left and right neighboring

nodes are aligned, and top and bottom neighboring nodes are loaded, it is possible to operate on them in parallel.

Table 10. *Instruction Level Characterization of the Jacobi-2D application.*

	Scalar Code	Vectorized Code		
		MVL=8 elements	MVL=16 elements	MVL=32 elements
Total Instructions	4,660,908,013	1,464,606,013	732,318,013	366,174,013
Scalar Instructions	4,660,908,013	1,074,206,013	537,118,013	268,574,013
Vector Memory Inst		65,280,000	32,640,000	16,320,000
Vector Loads		32,768,000	16,384,000	8,192,000
Vector Stores		32,512,000	16,256,000	8,128,000
Vector Arithmetic Inst		260,096,000	130,048,000	65,024,000
Vector to Scalar Inst		0	0	0
Vector Elem Manipulation Inst		65,024,000	32,512,000	16,256,000
Vector Slides		65,024,000	32,512,000	16,256,000
Vector Reductions		0	0	0
Total Vector Instructions		390,400,000	195,200,000	97,600,000
Total Vector Operations		3,104,900,000	3,104,900,000	3,104,900,000
% of Vectorization		74%	85%	92%
Average VL (elements)		7.95	15.90	31.81
SA-speedup		1.1153	1.2798	1.3816
		Vectorized Code		
		MVL=64 elements	MVL=128 elements	MVL=256 elements
Total Instructions		183,102,013	91,566,013	45,798,013
Scalar Instructions		134,302,013	67,166,013	33,598,013
Vector Memory Inst		8,160,000	4,080,000	2,040,000
Vector Loads		4,096,000	2,048,000	1,024,000
Vector Stores		4,064,000	2,032,000	1,016,000
Vector Arithmetic Inst		32,512,000	16,256,000	8,128,000
Vector to Scalar Inst		0	0	0
Vector Elem Manipulation Inst		8,128,000	4,064,000	2,032,000
Vector Slides		8,128,000	4,064,000	2,032,000
Vector Reductions		0	0	0
Total Vector Instructions		48,800,000	24,400,000	12,200,000
Total Vector Operations		3,104,900,000	3,104,900,000	3,104,900,000
% of Vectorization		96%	98%	99%
Average VL (elements)		63.62	127.25	254.5
SA-speedup		1.4389	1.4694	1.4851

Table 10 presents some statistics for the Jacobi-2D application. The total number of instructions drops considerably for the vectorized versions, achieving *% of Vectorization* from 74% for MVL=8 elements, up to 99% for MVL=256 elements. As the VL increases, the *% of Vectorization* increases because of the ratio resulting from the number of *Total*

Vector Operations, which remains equal, over the number of the total operations (*Scalar Instructions* + *Total Vector Operations*), which decreases as a result of the reduction of the number *Scalar Instructions*.

As an initial insight based on the data presented in Table 10, it is possible to obtain a SA-speedup of 1.11X. Additionally, for larger MVLs it is possible to achieve higher speedups. As increasing the number of lanes a linear speedup increase would be expected, especially for larger MVL configurations where the % of Vectorization is higher.

2.3.3.5 LavaMD2

LavaMD2 is a memory-bound application that calculates the particle potential and relocation due to mutual forces between particles within a large 3D space. This space is divided into cubes, or large boxes, that are allocated to individual cluster nodes. The large box at each node is further divided into cubes, called boxes. Twenty-six neighbor boxes surround each box (the home box). Home boxes at the boundaries of the particle space have fewer neighbors. Particles only interact with those other particles that are within a cutoff radius since ones at larger distances exert negligible forces. Thus the box sizes are chosen so that the cutoff radius does not span beyond any neighbor box for any particle in a home box, thus limiting the reference space to a finite number of boxes. The application first emulates partitioning of the particle space into boxes. Then, for every particle in the home box, the nested loop processes interactions first with other particles in the home box and then with particles in all neighbor boxes. The processing of each particle consists of a single stage of calculation that is enclosed in the innermost loop. This application was taken from Rodinia, and a detailed description can be found in [46].

The inputs for LavaMD2 are sized as shown in Table 11.

***Table 11**. LavaMD2 input data sets*

Simulation size	Value
simtiny	1 large box
simsmall	4 large boxes
simmedium	6 large boxes
simlarge	10 large boxes
native	100 large boxes

Degree of vectorization

Table 12 presents some statistics for the LavaMD2 application. The total number of instructions drops considerably for the vectorized versions, achieving *% of Vectorization*

from 97% for MVL=8 elements, up to 99% for MVL=256 elements. This application also shows similar behavior to Canneal where the Average VL does not remain constant throughout the execution. However, in this case, there are two main reasons because of this slight variation.

Table 12. *Instruction Level Characterization of the LavaMD2 application.*

	Scalar Code	Vectorized Code		
		MVL=8 elements	MVL=16 elements	MVL=32 elements
Total Instructions	24,615,519,089	1,221,009,837	658,332,614	470,779,164
Scalar Instructions	24,615,519,089	409,662,917	238,960,606	182,065,460
Vector Memory Inst		71,651,328	40,040,448	29,503,488
Vector Loads		63,221,760	31,610,880	21,073,920
Vector Stores		8,429,568	8,429,568	8,429,568
Vector Arithmetic Inst		739,695,592	379,331,560	259,210,216
Vector to Scalar Inst		0	0	0
Vector Elem Manipulation Inst		0	0	0
Vector Slides		0	0	0
Vector Reductions		0	0	0
Total Vector Instructions		811,346,920	419,372,008	288,713,704
Total Vector Operations		12,804,068,581	13,092,269,875	14,372,529,077
% of Vectorization		97%	98%	99%
Average VL (elements)		7.89	15.60	24.89
SA-speedup		1.8629	1.8465	1.6913
		Vectorized Code		
		MVL=64 elements	MVL=128 elements	MVL=256 elements
Total Instructions		283,221,885	283,238,998	283,248,849
Scalar Instructions		125,166,485	125,183,598	125,193,449
Vector Memory Inst		18,966,528	18,966,528	18,966,528
Vector Loads		10,536,960	10,536,960	10,536,960
Vector Stores		8,429,568	8,429,568	8,429,568
Vector Arithmetic Inst		139,088,872	139,088,872	139,088,872
Vector to Scalar Inst		0	0	0
Vector Elem Manipulation Inst		0	0	0
Vector Slides		0	0	0
Vector Reductions		0	0	0
Total Vector Instructions		158,055,400	158,055,400	158,055,400
Total Vector Operations		14,506,522,181	15,854,932,313	18,551,752,575
% of Vectorization		99%	99%	99%
Average VL (elements)		45.89	50.15	58.68
SA-speedup		1.6823	1.5404	1.3180

First, vector store instructions are similar for all the configurations. This is because once a full home box computation is done, the resultant forces (xfA_v, xfA_x, xfA_y, and xfA_z) are stored back to memory for the next computation, and this process is repeated the same number of times no matter the MVL. Also, this vector store instruction uses VL=1, contributing to reduce the Average VL to 7.89 for MVL=8 elements, 15.60 for MVL=16 elements, and so on.

Second, the number of particles per box is 96 32-bit elements, leading to requiring up to 48 elements (3072-bits) in each vector register. Then, for the MVL=8, 16, 32, and 64 elements configurations, the number of scalar and vector instructions is reduced in every configuration, while for larger configurations, the number of scalar and vector instructions remains equal. However, in the same way as Canneal, copying vector registers is performed, and is the cause of increasing the number of Total Vector Operations in every configuration.

From previous data, the initial insight about the expected performance is the following. The ratio between the number of the *Total instructions* of the serial version divided by the total number of operations (*Vector Operations* + *Scalar Instructions*), obtains a SA-speedup of 1.86x for MVL=8 elements and decreases up to 1.31x for MVL=256 elements. Finally, Increasing the number of lanes can increase performance since the percentage of arithmetic operations is high. However, we can anticipate that the best hardware could be a multilane design for medium-size vectors.

2.3.3.6 PathFinder

PathFinder application uses of ghost zone optimization technique (dynamic programming) to find a path on a 2-D grid. The application starts creating an array of weights (*wall*) of size given by *cols* and *rows* parameters and initializes it with random numbers. The algorithm is implemented in the *run* function. The algorithm starts from the bottom row to the top row with the smallest accumulated weights, where each step of the path moves straight ahead or diagonally ahead. The algorithm iterates row by row where each node selects a neighboring node in the previous row that has the smallest accumulated weight and adds its own weight to the sum. The applications finalize by printing all the shortest paths and the consumed time to find them. This application was taken from Rodinia, and a detailed description can be found in [46].

The inputs for PathFinder are sized as shown in Table 13.

***Table 13**. Pathfinder input data sets*

Simulation size	Value
simtiny	256 columns, 8 rows
simsmall	1024 columns, 128 rows
simmedium	2048 columns, 256 rows
simlarge	2048 columns, 1024 rows
native	4096 columns, 4096 rows

Degree of vectorization

This application features a high percentage of vector element manipulation instructions. In this sense, it is possible for researchers to use this application in order to evaluate different interconnection topologies (ring, crossbar, etc.) between multiple lanes in a vector architecture.

One interesting aspect of this application is that the algorithm implemented to find the shortest paths inside *run* function is composed of a nested loop. For each node, comparisons with its corresponding neighboring nodes are performed to find the smallest weight and add it to the current node weight. This task is easily implemented using the vector *slide1up* and *slide1down* operations to accommodate the neighboring nodes in the same position and operate on it to finally store the resultant data. Vector element manipulation instructions reported in Table 14 consumes 40% of the total vector instructions without take into account vector memory operations.

The total number of instructions drops considerably for the vectorized versions, achieving *% of Vectorization* from 83% for MVL=8 elements, up to 99% for MVL=256 elements. As the VL increases, the *% of Vectorization* increases, as shown in Table 14.

From previous data, as an initial insight about the expected performance, it is obtained a SA-speedup from 2.7x for MVL=8 elements up to 3.21x for MVL=256 elements, showing that it is possible to achieve better performance by only increasing the MVL.

Several factors can influence reducing or increasing this speedup. The execution of the remaining scalar instructions can be amortized underneath vector execution or the use of parallel lanes. Then, as an initial conclusion, it can be said that for larger MVLs, it is possible to achieve higher speedups than 2.7X. Additionally, as the number of lanes is increased, obtaining a speedup increase would be expected, especially for larger MVL configurations where the % of Vectorization is higher. Then, the SA-speedup presented in Table 14 can be multiplied by the number of lanes as a very initial insight.

Table 14. *Instruction Level Characterization of the Pathfinder application.*

	Scalar Code	Vectorized Code		
		MVL=8 elements	MVL=16 elements	MVL=32 elements
Total Instructions	5,433,477,921	436,308,256	220,250,467	112,206,924
Scalar Instructions	5,433,477,921	331,553,056	180,967,267	92,565,324
Vector Memory Inst		39,283,200	19,641,600	9,820,800
Vector Loads		26,188,800	13,094,400	6,547,200
Vector Stores		13,094,400	6,547,200	3,273,600
Vector Arithmetic Inst		39,283,200	19,641,600	9,820,800
Vector to Scalar Inst		0	0	0
Vector Elem Manipulation Inst		26,188,800	13,094,400	6,547,200
Vector Slides		26,188,800	13,094,400	6,547,200
Vector Reductions		0	0	0
Total Vector Instructions		104,755,200	39,283,200	19,641,600
Total Vector Operations		1,676,083,200	1,676,083,200	1,676,083,200
% of Vectorization		83%	91%	95%
Average VL (elements)		8	16	32
SA-speedup		2.7064	2.9466	3.0835
		Vectorized Code		
		MVL=64 elements	MVL=128 elements	MVL=256 elements
Total Instructions		58,192,720	31,183,142	17,681,517
Scalar Instructions		48,371,920	26,272,742	15,226,317
Vector Memory Inst		4,910,400	2,455,200	1,227,600
Vector Loads		3,273,600	1,636,800	818,400
Vector Stores		1,636,800	818,400	409,200
Vector Arithmetic Inst		4,910,400	2,455,200	1,227,600
Vector to Scalar Inst		0	0	0
Vector Elem Manipulation Inst		3,273,600	1,636,800	818,400
Vector Slides		3,273,600	1,636,800	818,400
Vector Reductions		0	0	0
Total Vector Instructions		9,820,800	4,910,400	2,455,200
Total Vector Operations		1,676,083,200	1,676,083,200	1,676,083,200
% of Vectorization		97%	99%	99%
Average VL (elements)		64	128	256
SA-speedup		3.1568	3.1948	3.2141

2.3.3.7 Particle-Filter

The Particle-Filter (PF) application is a statistical estimator of the location of a target object given noisy measurements of the state. In image analysis, the PF has a very large amount of applications. Feature tracking like surveillance from facial recognition to the following of vehicles in traffic are good examples. Also video compressing using PF is of

interest. This particular implementation is optimized for tracking cells, particularly leukocytes (white blood cells). The problem with most PF implementations is that the computational cost is prohibitive for real-time applications, so it is a good challenge for vector processors, which could provide enough speedup for this kind of application. This application uses special operations with masks, which send resultant data to the scalar core, and were not used in previous applications. These vector instructions are vfirst.m which finds the lowest-numbered active element of the source mask vector that has its least-significant bit set, and writes that element's index to a scalar register; and vpopc.m which counts the number of mask elements of the active elements of the vector source mask register that have their least-significant bit set, and writes the result to a scalar register. Also, this application uses complex operations such as logarithm, cosine, and square root. We consider that this instruction mixture needs to be evaluated. This application was taken from Rodinia, and a detailed description can be found in [46].

The application is divided into two main sections. The first one is in charge of generating a synthetic video sequence, which simulates the motion of a white blood cell with additive noise by picking a point in each frame, dilating that point, and then adding random Gaussian noise to the frame. This task is implemented in the *videoSequence* function, which receives as an input the video resolution of the screen (128x128) and the number of frames to generate. This task consumes 1% of the total execution time for a simlarge configuration.

The second one is the PF which implemented in the *particleFilter* function. PF takes the video sequence as input, with a predefined motion model representing the stimated path that the object will follow, this task consumes 3% of the *particleFilter* function for one frame. For every frame in the provided video sequence, the algorithm begins tracking the target object by making a series of guesses about the current frame given what is already known from the previous frame. The PF then determines the likelihood of each of those guesses ocyrring using a predefined likelihood model, this task consumes 6% of the *particleFilter* function for one frame. Subsequently, the PF normalizes those guesses based on their likelihoods and then sums the normalized guesses to determine the current position of the object. Finally, the FP updates the guesses based on the current location of the object. This task consumes 91% % of the *particleFilter* function for one frame. The process is repeated for all remaining frames in the video.

The inputs for Particle-Filter are sized as follows:

Table 15. *Particlefilter input data sets*

Simulation size	Value
simtiny	128x128 resolution , 2 frames , 256 particles
simsmall	128x128 resolution , 8 frames , 1024 particles
simmedium	128x128 resolution , 16 frames , 4096 particles
simlarge	128x128 resolution , 24 frames , 8192 particles
native	128x128 resolution , 1440 frames , 16,384 particles

Degree of vectorization

The task in charge of applying a predefined motion model that represents the estimated path that the object will follow is a good candidate to be vectorized because the same operations are applied to all objects in the frame. Furthermore, to apply the motion model, it is necessary to generate a sequence of random numbers using the Box-Muller transformation, which makes use of expensive operations such as logarithm, cosine, and square root.

The task which consumes most of the execution time is the guesses updates based on the current location of the object. These new guesses are used by the following frame in the video to iterate again. The task is implemented in a nested "*for*" loop, which performs a sequential search returning an index value to update the arrays *arrayX* and *arrayY*. This task can be implemented by first using a vector comparison instruction to obtain a mask representing the active elements for that iteration. Later, the vfirst.m instruction is used to know if there is at least one active element in the generated mask and its corresponding position. When the criteria are met, the position of each active element is obtained to finally use the vpopc.m instruction to check if all elements in the vector have been set, breaking the inner loop. Otherwise, the program continues with a new iteration until all elements get the updated position.

Table 16 presents some statistics for the PF application. The total number of instructions drops considerably for the vectorized versions, achieving *% of Vectorization* from 72% for MVL=8 elements, up to 91% for MVL=256 elements. However, when mapping the sequential search in a vector fashion, the number of the Total Vector Operations increases and represents more individual operations than the scalar version.

From previous data, it is possible to obtain an initial insight about the expected performance. Although the number of the Total Instructions decreases, the vectorized versions executes more individual operations which leads to obtain a SA-speedup from 0.61x for MVL=8 elements up to 0.77x for MVL=256 elements, showing that taking into account only the instruction and operation counts it should not be possible to improve

performance versus the scalar version. However, increasing this speedup still would be possible for the following reasons: (1) The execution of the remaining scalar instructions can be amortized underneath vector execution. (2) The use of parallel lanes. And, (3) as longer MVLs, the amount of latency amortized per vector instruction can be maximized. Then, as an initial conclusion, it can be said that for larger MVLs, combined with a multilane vector architecture, it is possible to achieve higher speedups than 0.61X.

Table 16. *Instruction Level Characterization of the ParticleFilter application.*

	Scalar Code	**Vectorized Code**		
		MVL=8 elements	MVL=16 elements	MVL=32 elements
Total Instructions	5,303,827,138	3,241,372,346	1,887,425,956	1,210,252,489
Scalar Instructions	5,303,827,138	2,464,598,103	1,499,037,353	1,016,056,746
Vector Memory Inst		238,592	119,296	59,648
Vector Loads		164,864	82,432	41,216
Vector Stores		73,728	36,864	18,432
Vector Arithmetic Inst		583,573,451	291,787,837	145,895,000
Vector to Scalar Inst		192,962,200	96,481,470	48,241,095
Vector Elem Manipulation Inst		0	0	0
Vector Slides		0	0	0
Vector Reductions		0	0	0
Total Vector Instructions		776,774,243	388,388,603	194,195,743
Total Vector Operations		6,214,193,944	6,214,217,648	6,214,263,776
% of Vectorization		72%	81%	86%
Average VL (elements)		8	16	32
SA-speedup		0.6111	0.6876	0.7336
		Vectorized Code		
		MVL=64 elements	MVL=128 elements	MVL=256 elements
Total Instructions		871,737,448	702,501,558	617,873,199
Scalar Instructions		774,638,133	653,950,467	593,596,220
Vector Memory Inst		29,824	14,912	7,456
Vector Loads		20,608	10,304	5,152
Vector Stores		9,216	4,608	2,304
Vector Arithmetic Inst		72,948,583	36,475,367	18,238,759
Vector to Scalar Inst		24,120,908	12,060,812	6,030,764
Vector Elem Manipulation Inst		0	0	0
Vector Slides		0	0	0
Vector Reductions		0	0	0
Total Vector Instructions		97,099,315	48,551,091	24,276,979
Total Vector Operations		6,214,356,160	6,214,539,648	6,214,906,624
% of Vectorization		89%	90%	91%
Average VL (elements)		64	128	256
SA-speedup		0.7589	0.7722	0.7790

2.3.3.8 Somier

Somier is an application from the physics Simulation domain. The program describes the trajectory of an object in a 3D-space. Given the mass M and applying a force F, the acceleration of the object is calculated. Then using the derivative with respect to time, the speed of the object is obtained. Finally, the derivative with respect to time is reapplied to obtain the object's position over time.

The inputs for Particle-Filter are sized as follows:

Table 17. *Somier input data sets*

Simulation size	Value
simtiny	Steps 2, dim 64
simsmall	Steps 4, dim 64
simmedium	Steps 2, dim 128
simlarge	Steps 4, dim 128
native	Steps 4, dim 1024

Degree of vectorization

There are four main candidate functions to be vectorized: *compute_forces*, *acceleration*, *velocities*, and positions. The *compute_forces* represents the most challenging function to be vectorized since it is required to compute the force contribution of the neighboring nodes (X+1, X-1, Y+1, Y-1, Z+1, and Z-1) for each point. To do that, the force contribution is computed inside a three levels nested loop (X,Y, and Z). The *acceleration* function can be vectorized by simply dividing all the computed forces by the mass M. The *velocities* function can be vectorized by simply multiplying the acceleration by the derivative with respect time. Finally, the *positions* functions can be vectorized by simply multiplying the velocity by the derivative with respect time.

Table 18 presents some statistics for the Somier application. The total number of instructions drops considerably for the MVL=8 elements vectorized version, achieving 56*% of Vectorization.* However, as the MVL is increased, a few increases in the % of Vectorization are shown. This behavior is because when mapping the *force_contribution* function in a vector fashion, there is a section of scalar code not vectorized, which remains constant executing around 640,300,015 scalar instructions.

From previous data, it is possible to obtain an initial insight about the expected performance. Although the number of the Total Instructions decreases when comparing with the scalar version, the vectorized versions executes almost the same number individual operations which leads to obtain a SA-speedup from 3.74 for MVL=8 elements

up to 3.95x for MVL=256 elements. However, increasing this speedup still would be possible for the following reasons:

(1) The execution of the remaining scalar instructions can be amortized underneath vector execution.

Table 18. *Instruction Level Characterization of the Somier application.*

	Scalar Code	Vectorized Code		
		MVL=8 elements	MVL=16 elements	MVL=32 elements
Total Instructions	6,254,373,928	850,576,201	742,767,817	688,863,625
Scalar Instructions	6,254,373,928	731,605,321	683,282,377	659,120,905
Vector Memory Inst		55,452,672	27,726,336	13,863,168
Vector Loads		39,230,208	19,615,104	9,807,552
Vector Stores		16,222,464	8,111,232	4,055,616
Vector Arithmetic Inst		63,518,208	31,759,104	15,879,552
Vector to Scalar Inst		0	0	0
Vector Elem Manipulation Inst		0	0	0
Vector Slides		0	0	0
Vector Reductions		0	0	0
Total Vector Instructions		118,970,880	59,485,440	29,742,720
Total Vector Operations		940,613,520	940,613,520	940,613,520
% of Vectorization		56%	58%	59%
Average VL (elements)		7.90	15.81	31.62
SA-speedup		3.7402	3.8515	3.9096

	Vectorized Code		
	MVL=64 elements	MVL=128 elements	MVL=256 elements
Total Instructions	661,911,529	648,435,481	647,304,985
Scalar Instructions	647,040,169	640,999,801	640,188,793
Vector Memory Inst	6,931,584	3,465,792	3,244,608
Vector Loads	4,903,776	2,451,888	2,304,432
Vector Stores	2,027,808	1,013,904	940,176
Vector Arithmetic Inst	7,939,776	3,969,888	3,871,584
Vector to Scalar Inst	0	0	0
Vector Elem Manipulation Inst	0	0	0
Vector Slides	0	0	0
Vector Reductions	0	0	0
Total Vector Instructions	14,871,360	7,435,680	7,116,192
Total Vector Operations	940,613,520	940,613,520	940,671,630
% of Vectorization	59%	59%	60%
Average VL (elements)	63.25	126.5	132.18
SA-speedup	3.9394	3.9544	3.9563

(2) as longer MVLs, the amount of latency amortized per vector instruction can be maximized. However, in this case, the use of parallel lanes would not show a high-performance increase since the number of Vector Memory Instructions is almost the same than the Vector Arithmetic Instructions. Then, as an initial conclusion, it can be said that for larger MVLs, it is possible to achieve higher speedups than 3.74X, and as the number of lanes is increased, only little improvement can be seen.

2.3.3.9 Streamcluster

The Streamcluster application solves the online clustering problem. For a stream of input points, it finds a predetermined number of medians so that each point is assigned to its nearest center. Streamcluster is a common operation where large amounts of continuously produced data have to be organized under real-time conditions, for example, network intrusion detection, pattern recognition, and data mining. This application was taken from PARSEC, and a detailed description can be found in [44].

The program is memory-bound for low-dimensional data and becomes increasingly computationally intensive as the dimensionality increases.

The parallel gain computation is implemented in the function *pgain*. Given a preliminary solution, the function computes how much cost can be saved by opening a new center, this task spends 98.37% of the total execution time. Inside this function it is called the function *dist*, where the Euclidian distance squared between two points is calculated as the cumulative addition of the distances between each of the point's dimensions. If the heuristic determines that the change would be advantageous, the results are committed. *dist* function consumes 95% of the total execution time, and it is composed by a "*for*" loop where are performed a couple of simple arithmetic operations. Finally, the program writes the computed results to an output file.

The inputs of Streamcluster are defined as follows:

***Table 19**. Streamcluster input data sets*

Simulation size	Value
simtiny	1024 input points, block size 1024 points, 16 point dimensions, 10-20 centers, up to 1,000 intermediate centers allowed
simsmall	2048 input points, block size 2048 points, 32 point dimensions, 10-20 centers, up to 1,000 intermediate centers allowed
simmedium	4096 input points, block size 4096 points, 64 point dimensions, 10-20 centers, up to 1,000 intermediate centers allowed
simlarge	8192 input points, block size 8192 points, 128 point dimensions, 10-20 centers, up to 1,000 intermediate centers allowed
native	1,000,000 input points, block size 200,000 points, 128 point dimensions, 10-20 centers, up to 5000, intermediate centers allowed

Degree of vectorization

The *dist* function is the candidate to be vectorized. This function is highly vectorizable, but the number of vector arithmetic operations is almost the same that the memory operations needed in each iteration, which means that a vector implementation could be limited by the memory subsystem. Then, it is implemented by using two vector loads and two vector arithmetic operations. Outside the inner "for" loop, a vector reduction is needed to get the cumulative addition. The resultant scalar value of the reduction is sent immediately to the scalar core to compute the cost of opening a new center. This is done by using the *vfirst.m* instruction. Note that this last step could cause a huge impact on performance since before starting a new iteration, it is necessary to evaluate if the cost of opening a new center would be advantageous. This computation is made by the scalar core, meaning that for every iteration, the vector engine receives a block of instructions, computes them, and returns a scalar value. Later the vector engine will be idle while the scalar core evaluates the results.

Table 20 presents some statistics Streamcluster application, for this application, the largest MVL is 64 elements, it is related to input parameter "dim", which for simlarge configuration is set to 128 elements each 32-bits. The number of instructions for MVL=8 elements is reduced by 7.12X, and as the MVL is increased, the number of instructions decreases up to 14.1X. On the other hand, the % of Vectorization increases from 82% for MVL=8 elements, up to 96% for MVL=256 elements.

One interesting thing in this application is that the number of *Total Vector Operations* is not the same for different MVL configurations. As we increase the MVL, the number of vector operations also increases. This variation in the number of Vector Operations happens because of two different reasons. First, outside (before and after) the nested "for" loop, there are vector instructions that initialize a couple of vector registers and performs a reduction of the final result. Then, the number of *Total Vector Operations* are equal to the number of *Total Vector Instructions* multiplied by the *Average VL* parameter, meaning that the larger the *VLs* is, the more vector operations are executed outside the nested loop. Second, in the same way as Canneal and LavaMD2, copying vector registers is performed, then for MVL configurations bigger than 64 elements, copying operations start moving data elements that will be not used.

As a preliminary observation, it can be said that it is an application that does not benefit much from larger MVLs because more vector operations are executed for this case compared with shorter MVLs.

Table 20. *Instruction Level Characterization of the Streamcluster application.*

	Scalar Code	Vectorized Code		
		MVL=8 elements	MVL=16 elements	MVL=32 elements
Total Instructions	16,386,167,611	2,298,533,650	1,649,001,354	1,324,235,250
Scalar Instructions	16,386,167,611	1,797,915,148	1,378,425,545	1,161,914,809
Vector Memory Inst		216,510,752	108,255,368	54,127,684
Vector Loads		216,510,752	108,255,368	54,127,684
Vector Stores		0	0	0
Vector Arithmetic Inst		270,575,828	162,320,441	108,192,757
Vector to Scalar Inst		13531922	13531921	13531921
Vector Elem Manipulation Inst		0	0	0
Vector Slides		0	0	0
Vector Reductions		0	0	0
Total Vector Instructions		500,618,502	270,575,809	162,320,441
Total Vector Operations		8,009,896,032	8,658,425,888	10,388,508,224
% of Vectorization		82%	86%	90%
Average VL (elements)		8	16	32
SA-speedup		1.6707	1.6326	1.4187
		Vectorized Code		
		MVL=64 elements	MVL=128 elements	MVL=256 elements
Total Instructions		1,161,852,302	1,161,852,302	1,161,852,302
Scalar Instructions		1,053,659,545	1,053,659,545	1,053,659,545
Vector Memory Inst		27,063,842	27,063,842	27,063,842
Vector Loads		27,063,842	27,063,842	27,063,842
Vector Stores		0	0	0
Vector Arithmetic Inst		81,128,915	81,128,915	81,128,915
Vector to Scalar Inst		13531921	13531921	13531921
Vector Elem Manipulation Inst		0	0	0
Vector Slides		0	0	0
Vector Reductions		0	0	0
Total Vector Instructions		108,192,757	108,192,757	108,192,757
Total Vector Operations		13,848,672,896	18,450,246,092	27,660,154,532
% of Vectorization		93%	95%	96%
Average VL (elements)		64	85.26	127.82
SA-speedup		1.0996	0.8401	0.5707

Based on the data presented in Table 20, it is possible to get an initial insight about the expected performance. A SA-speedup of 1.67x for MVL=8 elements and one lane configuration could be achieved. As the MVL increases, it is not clear if speedup improvements could be expected since the number of Vector Operations also increases. The increase in the number of parallel lanes could give a slight speedup increase. As mentioned before, this application is memory-bound for the large input set. Then, the

speedup could be limited by the memory subsystem. This discussion continues in Section 2.4.1.9, showing the results of the application executed on different VPU configurations.

2.3.3.10 Swaptions

The swaptions application is an Intel RMS workload that uses the Heath-Jarrow-Morton (HJM) framework to price a portfolio of swaptions based on Monte Carlo simulation to compute the prices. The HJM framework describes how interest rates evolve for risk management and asset-liability management for a class of models. Its central insight is that there is an explicit relationship between the drift and volatility parameters of the forward-rate dynamics in a no-arbitrage market. This application was taken from PARSEC, and a detailed description can be found in [44].

The program stores the portfolio in the swaptions array. Each entry corresponds to one derivative. The function HJM_Swaption_Blocking has the routines to compute various security prices. It invokes two main functions HJM_Sim-Path_Forward_Blocking which is responsible for 93.8% of the total execution time, and *Discount_Factors_Blocking,* which consumes 6.21% of the total execution time. Both functions are partially vectorizable.

HJM_Sim-Path_Forward_Blocking computes and stores an HJM Path for given inputs, where all data structures are initialized and calls for the most time-consuming functions are performed (*RanUnif, serialB, CumNormalInv*), also contains a couple of vectorizable for loops where are performed basic arithmetic operations. The first steps inside this function are memory assignment, and random initialization (*RanUnif* function). This initialization consumes nearly 10% of the total execution time. This function can be vectorized by defining MVL number of seeds instead of only one as in the scalar version. By vectorizing this function the output differs by a very small difference. It is because the following generated random numbers are calculated based on the new vector of seeds instead of only one. The final standard error is sometimes slightly smaller or slightly bigger. The next function is *serialB* which generates the cumulative normal distribution matrix necessary to calculate the HJM paths. It consumes 26% of the total execution time. This function is implemented using three nested “for” loops that iterate over: (1) the number of factors in the HJM framework (iFactors, set to 3), (2) the number of time-steps (iN, set to 11) and the block size (BLOCK_SIZE, set to 32). Inside this nested *“for”* loop the function *CumNormalInv* is called to compute the inverse of the cumulative normal distribution function. This function consumes 22.98% of the total execution time. This function has a regular DLP pattern and computes basic floating-point operations, including *fadd*, *fsub*, *fmul*, *fdiv*, and also logarithmic functions.

Table 21 shows the inputs data sets for Swaptions. Swaptions prints the resulting swaption prices to the console.

***Table 21**. Swaptions input data sets*

Simulation size	Value
simtiny	8 swaptions, 512 simulations
simsmall	8 swaptions, 4096 simulations
simmedium	32 swaptions, 8192 simulations
simlarge	64 swaptions, 16384 simulations
native	128 swaptions, 1,000,000 simulations

Degree of vectorization

Table 22 presents some statistics Swaptions application. The number of instructions for MVL=8 elements is reduced by 5.13x, and as the MVL is increased, the number of instructions decreases up to 58x. On the other hand, the percentage of vectorization increases as the MVL is increased, achieving up to 98% of vectorization. Considering previous data, it is possible to achieve a SA-speedup from 1.38x for MVL=8 elements up to 1.62x for MVL=256 elements, showing that it is possible to achieve better performance by only increasing the MVL. This application presents a regular and high DLP pattern. For this reason, as the number of lanes increases, a linear speedup increase would be expected, especially for larger MVL configurations where the *% of Vectorization* is higher.

Table 22. *Instruction Level Characterization of the Swaptions application.*

	Scalar Code	Vectorized Code		
		MVL=8 elements	MVL=16 elements	MVL=32 elements
Total Instructions	11,762,554,240	2,289,642,246	1,210,763,856	670,739,072
Scalar Instructions	11,762,554,240	1,404,738,310	768,311,888	449,513,088
Vector Memory Inst		112,023,552	56,011,776	28,005,888
Vector Loads		62,595,072	31,297,536	15,648,768
Vector Stores		49,428,480	24,714,240	12,357,120
Vector Arithmetic Inst		772,880,384	386,440,192	193,220,096
Vector to Scalar Inst		0	0	0
Vector Elem Manipulation Inst		0	0	0
Vector Slides		0	0	0
Vector Reductions		0	0	0
Total Vector Instructions		884,903,936	442,451,968	221,225,984
Total Vector Operations		7,079,231,488	7,079,231,488	7,079,231,488
% of Vectorization		83%	90%	94%
Average VL (elements)		8	16	32
SA-speedup		1.3864	1.4989	1.5624

	Vectorized Code		
	MVL=64 elements	MVL=128 elements	MVL=256 elements
Total Instructions	401,357,791	266,323,210	200,489,352
Scalar Instructions	290,744,799	211,016,714	172,836,104
Vector Memory Inst	14,002,944	7,001,472	3,500,736
Vector Loads	7,824,384	3,912,192	1,956,096
Vector Stores	6,178,560	3,089,280	1,544,640
Vector Arithmetic Inst	96,610,048	48,305,024	24,152,512
Vector to Scalar Inst	0	0	0
Vector Elem Manipulation Inst	0	0	0
Vector Slides	0	0	0
Vector Reductions	0	0	0
Total Vector Instructions	110,612,992	55,306,496	27,653,248
Total Vector Operations	7,079,231,488	7,079,231,488	7,079,231,488
% of Vectorization	96%	97%	98%
Average VL (elements)	64	128	256
SA-speedup	1.5960	1.6135	1.6220

2.4 Evaluation

The data and the analysis presented in this chapter have the goal of establishing a base model for research on vector architectures. Rather than presenting approaches for the best performance, a discussion is presented that evaluates the results obtained in the analysis presented in Section 2.3.3 and the results obtained when the vectorized benchmarks are executed in the previously presented gem5 simulator with several vector engine configurations. Additionally, area and energy results from the McPAT framework are obtained to present a discussion of the tradeoffs between performance, area, and energy metrics.

2.4.1 Evaluation Environment

Table 23. *gem5 evaluation environment.*

<table>
<tr><th colspan="4">Config. 1 to 4</th><th colspan="4">Config. 5 to 8</th><th colspan="4">Config. 9 to 12</th><th colspan="4">Config. 13 to 16</th><th colspan="4">Config. 17 to 20</th><th colspan="4">Config. 21 to 24</th></tr>
<tr><td colspan="24">Scalar Core
Clock Frequency - 2 GHz
Dual-Issue 64-bit RISC-V superscalar in-order pipeline,</td></tr>
<tr><td colspan="24">Vector Engine - Clock Frequency - 1 GHz</td></tr>
<tr><td colspan="4"># Lanes</td><td colspan="4"># Lanes</td><td colspan="4"># Lanes</td><td colspan="4"># Lanes</td><td colspan="4"># Lanes</td><td colspan="4"># Lanes</td></tr>
<tr><td>1</td><td>2</td><td>4</td><td>8</td><td>1</td><td>2</td><td>4</td><td>8</td><td>1</td><td>2</td><td>4</td><td>8</td><td>1</td><td>2</td><td>4</td><td>8</td><td>1</td><td>2</td><td>4</td><td>8</td><td>1</td><td>2</td><td>4</td><td>8</td></tr>
<tr><td colspan="4">MVL 8 elements (512-bit)</td><td colspan="4">MVL 16 elements (1024-bit)</td><td colspan="4">MVL 32 elements (2048-bit)</td><td colspan="4">MVL 64 elements (4096-bit)</td><td colspan="4">MVL 128 elements (8192-bit)</td><td colspan="4">MVL 256 elements (16384-bit)</td></tr>
<tr><td colspan="24">Renaming with 64 Physical Registers</td></tr>
<tr><td colspan="4">4R/2W VRF: 4KB</td><td colspan="4">4R/2W VRF: 8KB</td><td colspan="4">4R/2W VRF: 16KB</td><td colspan="4">4R/2W VRF: 32KB</td><td colspan="4">4R/2W VRF: 64KB</td><td colspan="4">4R/2W VRF: 128KB</td></tr>
<tr><td colspan="24">1 pipelined arithmetic unit / Lane</td></tr>
<tr><td colspan="24">Ring topology for Lane Interconnection</td></tr>
<tr><td colspan="24">VMU with 1 Memory Port connected to L2, 512-bit memory interface</td></tr>
<tr><td colspan="24">Memory System</td></tr>
<tr><td colspan="24">32KB L1I – hit latency 4 clock cycles – cache line 512-bit – 4 MSHRs
32KB L1D – hit latency 4 clock cycles – cache line 512-bit – 4 MSHRs
1MB L2 – hit latency 12 clock cycles – cache line 512-bit – 32 MSHRs
2 GB DDR3-1600 Memory - DRAM memory access latency per DRAM burs 46.25ns</td></tr>
</table>

The vector engine is attached to a superscalar in-order processor, with the system configurations shown in Table 23. Twenty-four configurations are evaluated for the vector engine. First, from one up to eight lanes are configured. By doing this and setting only one memory port, it could be enough to feed up to eight lanes. This is taking into account that the cache line size is set to 512-bit (8 elements each 64-bit), and with every cache line request it is possible to send one element to each lane in an interleaved fashion. The MVL allowed varies from 8 to 256 elements. All the configurations implement renaming with 64 physical registers, leading to VRF sizes from 4KB to 128KB for the different configurations. Issue queues with in-order issue logic are set. Each lane features only one pipelined arithmetic unit. Also, a ring topology for lane interconnection

is chosen. The designer is able to choose simpler or more aggressive configurations according to the research requirements.

2.4.2 Area Evaluation

Before discussing performance results, it is interesting to present the area cost for each of the configurations presented in Table 23. Figure 2.6 shows the McPAT area evaluation for the 24 configurations presented in Table 23. We configure McPAT for the 22nm technology node. L2 Cache and the scalar core, including L1 instruction and data caches, are plotted separately from the VPU configurations to ease comparisons.

For the VPU, only the main contributors are modeled. This is the multi-ported VRF and the FPUs. Each lane is equipped with one FPU. Then, as we increase the number of lanes, the area consumed by the FPUs doubles in every configuration from 0.12mm^2 for one lane configuration up to 0.94mm^2 for eight-lane configuration.

For the VRF, it has been considered the following configurations:

For one-lane configuration means that the VRF is centralized in only one lane. The six configurations are listed below:

- MVL=8 (VRF-4KB): eight 4R/2W 64-bit*64-entries memory macros.
- MVL=16 (VRF-8KB): eight 4R/2W 64-bit*128-entries memory macros.
- MVL=32 (VRF-16KB): eight 4R/2W 64-bit*256-entries memory macros.
- MVL=64 (VRF-32KB): eight 4R/2W 64-bit*512-entries memory macros.
- MVL=128 (VRF-64KB): eight 4R/2W 64-bit*1024-entries memory macros.
- MVL=256 (VRF-128KB): eight 4R/2W 64-bit*2048-entries memory macros.

Note that for MVL=8 it has been implemented as eight 4R/2W 64-bit*64-entries memory structures. With this, in every read operation, it is possible to read the eight memories in parallel and store the elements in the source buffers to eventually feed the functional units. We will keep a small size for the source buffers, then in all the configurations, no more than eight elements in parallel will be read from each lane. When MVL=16, it has been implemented as eight 4R/2W 64-bit*128-entries memory structures. It is basically doubling the size of each memory macro. Note that this increase does not double the overall size. This happens because internal circuitry such as the pre-charge logic and the sense amplifiers, which represent an important area overhead in the SRAM memory structure, is independent to the number of entries. Then, the size increase is

related to the increase in the memory cells by itself and some logic around such as the row decoders.

For two lanes configuration means that each lane has held half of the overall VRF. The six configurations are listed below:

- MVL=8 (VRF-4KB): eight 4R/2W 64-bit*64-entries memory macros.
- MVL=16 (VRF-8KB): sixteen 4R/2W 64-bit*64-entries memory macros.
- MVL=32 (VRF-16KB): sixteen 4R/2W 64-bit*128-entries memory macros.
- MVL=64 (VRF-32KB): sixteen 4R/2W 64-bit*256-entries memory macros.
- MVL=128 (VRF-64KB): sixteen 4R/2W 64-bit*512-entries memory macros.
- MVL=256 (VRF-128KB): sixteen 4R/2W 64-bit*1024-entries memory macros.

In this case, the size increase pattern is also present in every MVL configuration. However, differences can be seen when comparing with one-lane configuration. For example, for MVL=16 and one-lane configuration, eight 4R/2W 64-bit*128-entries memory structures are implemented in 0.18mm^2. On the contrary, for two-lane configuration, eight 4R/2W 64-bit*64-entries memory structures are implemented per lane, leading to require 0.25mm^2. While the first one has memory macros with 128 entries, the second one implements memory macros with 64 entries but doubles the number of macros. Then, it is important to highlight that is a designer decision of how the VRF is implemented. Implementing several small memory macros implies more area, while implementing few big memory macros could lead to area savings. However, there are a couple of factors that also must be taking into account: first, each individual access (read/write operation) has a higher energy cost for bigger memory macros. Second, bigger memory macros have longer access time, which can influence on the final working frequency. The above considerations will be discussed in the next study.

For four lanes configuration, each lane holds a quarter of the overall VRF and presents the same pattern explained before. The six configurations are listed below:

- MVL=8 (VRF-4KB): eight 4R/2W 64-bit*64-entries memory macros.
- MVL=16 (VRF-8KB): sixteen 4R/2W 64-bit*64-entries memory macros.
- MVL=32 (VRF-16KB): thirty-two 4R/2W 64-bit*64-entries memory macros.
- MVL=64 (VRF-32KB): thirty-two 4R/2W 64-bit*128-entries memory macros.
- MVL=128 (VRF-64KB): thirty-two 4R/2W 64-bit*256-entries memory macros.

- MVL=256 (VRF-128KB): thirty-two 4R/2W 64-bit*512-entries memory macros.

Finally, for eight lanes configuration, each lane holds an eighth of the overall VRF and presents the same pattern explained before. The six configurations are listed below:

- MVL=8 (VRF-4KB): eight 4R/2W 64-bit*64-entries memory macros.
- MVL=16 (VRF-8KB): sixteen 4R/2W 64-bit*64-entries memory macros.
- MVL=32 (VRF-16KB): thirty-two 4R/2W 64-bit*64-entries memory macros.
- MVL=64 (VRF-32KB): sixty-four 4R/2W 64-bit*64-entries memory macros.
- MVL=128 (VRF-64KB): sixty-four 4R/2W 64-bit*128-entries memory macros.
- MVL=256 (VRF-128KB): sixty-four 4R/2W 64-bit*256-entries memory macros.

There are two interesting things to discuss in this configuration. First, from MVL=8 to MVL=64, the area increase double in each MVL configuration because of all of these configurations implements the same size of memory macros, but duplicating the number in each configuration. However, for MVL=128 and MVL=256, we double the size of the memory macro instead of double the number of memory macros. This is because we are using up to 8 memory macros per lane. Second, note that in this case, the 128 KB VRF area is 2.65mm^2, which is bigger than the 1MB L2 shared cache. This is because internally the L2 cache implements dual-port memory structures. When banking it, creates the illusion of having several memory ports but with the limitation that each port can address only a portion of the overall cache. On the contrary, the VRF implements 4R/2W memory macros which are costlier, as the area numbers show.

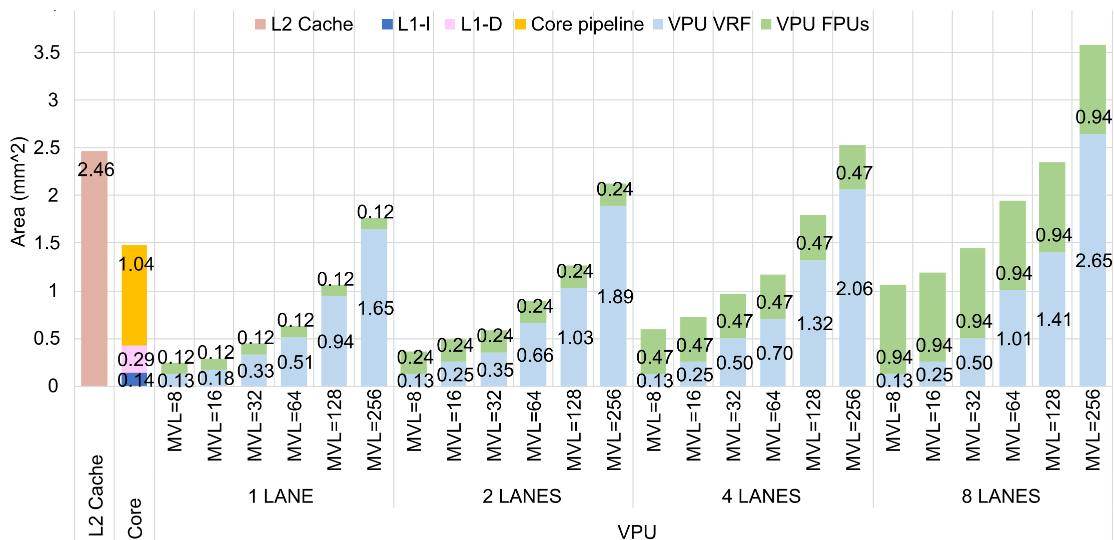

Figure 2.6 *McPAT area evaluation.*

2.4.3 Performance and Energy Evaluation

2.4.3.1 Axpy

Figure 2.7 shows the execution time (left axis) and speedup (right axis) obtained for the different configurations. Also, the SA-speedup (right axis) is shown to discuss these results. The obtained speedup for MVL=8 and one lane configuration is 1.21x, which is lower than the SA-speedup (1.60x). However, as the MVL is increased, speedup improvements are seen. This happens because many of the remaining scalar instructions are amortized underneath vector execution, and the amount of latency amortized per vector instruction is maximized. Regarding the use of parallel lanes, the expectations are accurate. Since Axpy is a memory-bound kernel, a marginal speed-up increase is obtained as the number of lanes is increased.

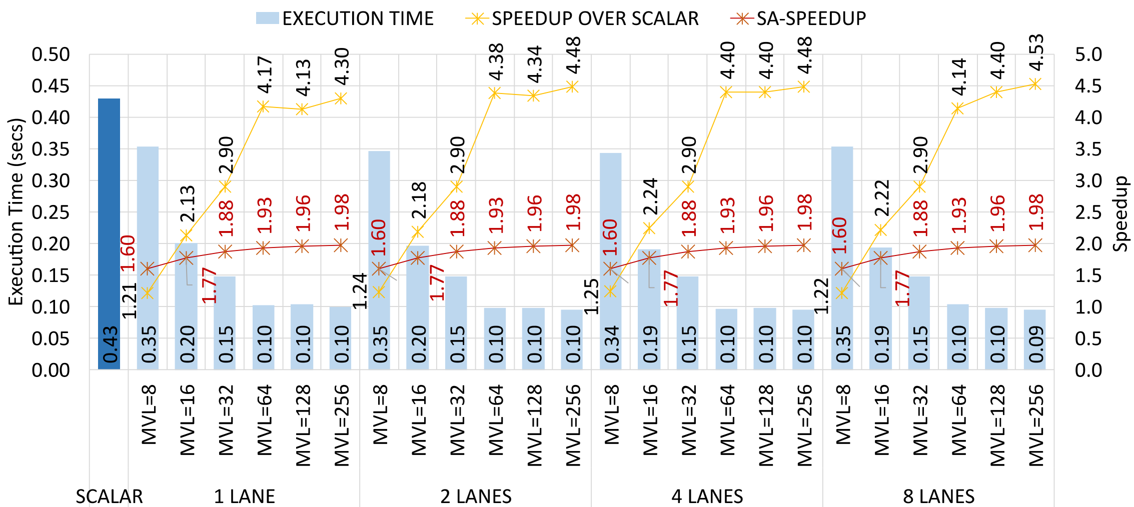

Figure 2.7 *Axpy runtime/speedup for different configurations.*

Figure 2.8 shows the performance/mm^2 efficiency metric to discuss how good is the hardware configuration for the different benchmarks. In this case, the most efficient configuration corresponds to one lane with MVL=16 elements. It is clear that having more than one lane is not beneficial in terms of performance/mm^2. A stronger memory subsystem, maybe adding memory ports or prefetching is necessary to go faster in memory-bound applications like Axpy.

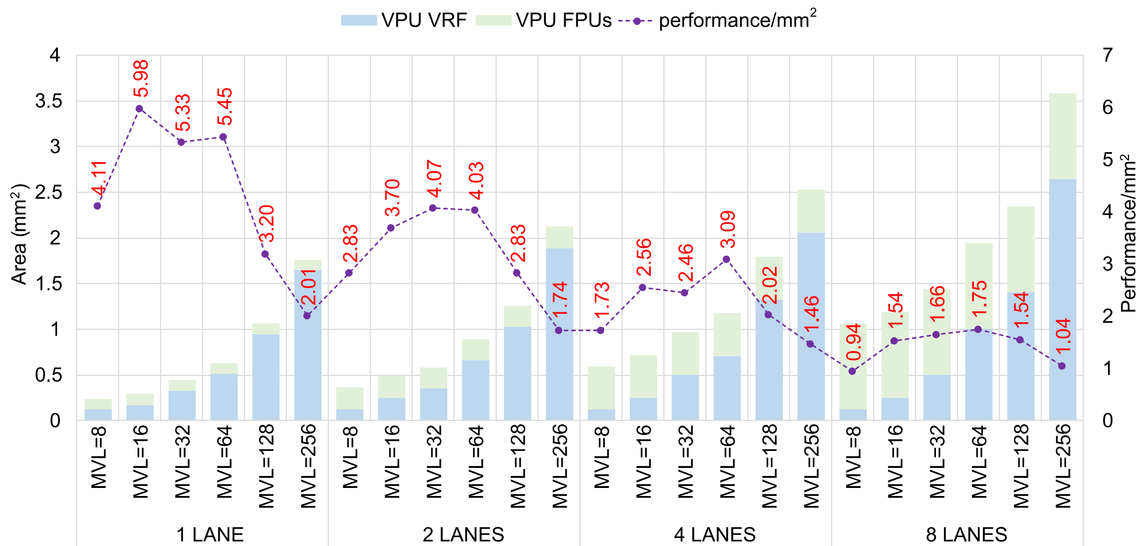

Figure 2.8 *Axpy performance/mm² efficiency.*

Figure 2.9 shows energy consumption (left axis) and Energy Delay Product (EDP) efficiency metric (right axis) to discuss the tradeoff between the different configurations. We evaluate four main energy contributors: the scalar core, which includes both L1-I and L1-D caches, the VPU VRF, the VPU floating-point units, and the L2 cache. For all of them, we present dynamic and leakage energy.

First, for the scalar core, we can see an important dynamic energy decrease from the scalar version to the vectorized versions. This is because the total number of instructions executed for the scalar version is 134,217,738, while for MVL=256, the instruction count is reduced to only 786,447. There is also an important leakage energy decrease, mainly because the execution of the vectorized versions is faster.

Second, for the VPU VRF, there are interesting results for both dynamic and leakage energy. For one-lane configuration, dynamic energy increases as we increase the MVL configuration. This happens because as the MVL is increased, the size of the memory macro is increased. As mentioned before, individual read/write operations are costlier for bigger memory macros. Regarding the leakage energy, we can see that it is increased as the MVL is increased. This happens because longer MVL configurations lead to bigger VRFs. Also, it is interesting to compare with eight lanes configuration. Similar to one lane configuration, the leakage energy increases as the MVL increases. However, the leakage represents an important contributor in this new scenario since 64 memory macros are required for the bigger configurations. Also is interesting to see that for MVL=8, MVL=16, MVL=32, and MVL=64, the dynamic energy is the same. This happens because the four configurations implement 4R/2W 64-bit*64-entries memory macros and only double the number of memory macros in each configuration. However, for MVL=128

and MVL=256, the dynamic energy increases because now are implemented bigger memory macros.

Third, for the VPU FPUs we can see that as we increase the number of lanes, the leakage energy increases as expected. Regarding dynamic energy, all the configurations consume the same energy because the total number of arithmetic operations is the same no matter the MVL or lane configuration.

Fourth, for the 1MB L2 cache, we can see that leakage energy represents an important contributor because it is the biggest structure. As mentioned before, this is a memory-bound application. Dynamic energy is very close for all the configurations.

Finally, normalized Energy Delay Product (EDP) efficiency metric is shown, where MVL=64 represents the most efficient configuration for 1,2 and 4 lane configurations, while MVL=128 represents the most efficient configuration for eight lanes configuration.

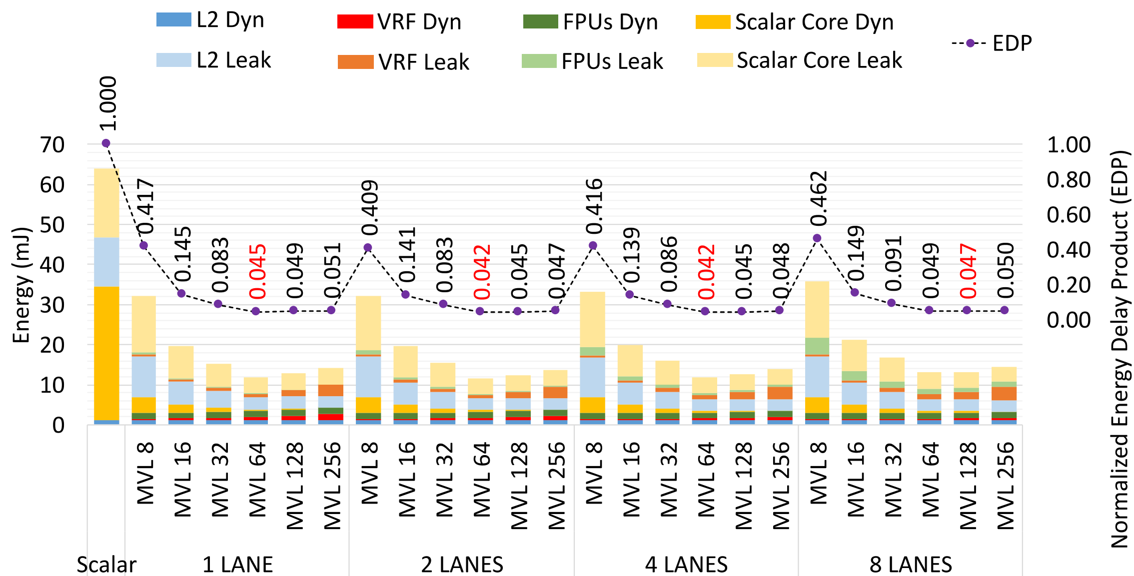

Figure 2.9 *Axpy energy consumption (left axis) and normalized Energy Delay Product (right axis) for different configurations.*

2.4.3.2 Blackscholes

Figure 2.10 shows the execution time (left axis) and speedup (right axis) obtained for the different configurations. Also, the SA-speedup (right axis) is shown to discuss these results. The obtained speedup for MVL=8 and one lane configuration is 2.86x, which is higher than the SA-speedup (2.25x). This happens because many of the remaining scalar instructions can be amortized underneath vector execution. As the MVL is increased, speedup improvements are seen, as discussed in Section 2.3.3.2. Regarding the expected linear increase as the number of lanes is increased, it is not

completely true for the different MVL configurations. Configurations with small and medium-size MVL do not benefit considerably from adding more lanes, unlike configurations that use large vectors. This is mainly because, in all configurations, the start-up time is incurred. For low MVL configurations, the start-up time is high compared to the total execution time of the instruction. In this case, the advantage for regular and high DLP applications for large MVL and the start-up time becomes minimal compared with the total execution time of the instruction. Then showing almost a linear speed-up increase for MVL=256 and the deferent lane configurations (4.83x, 8.97x, 15.67x, and 24.92x).

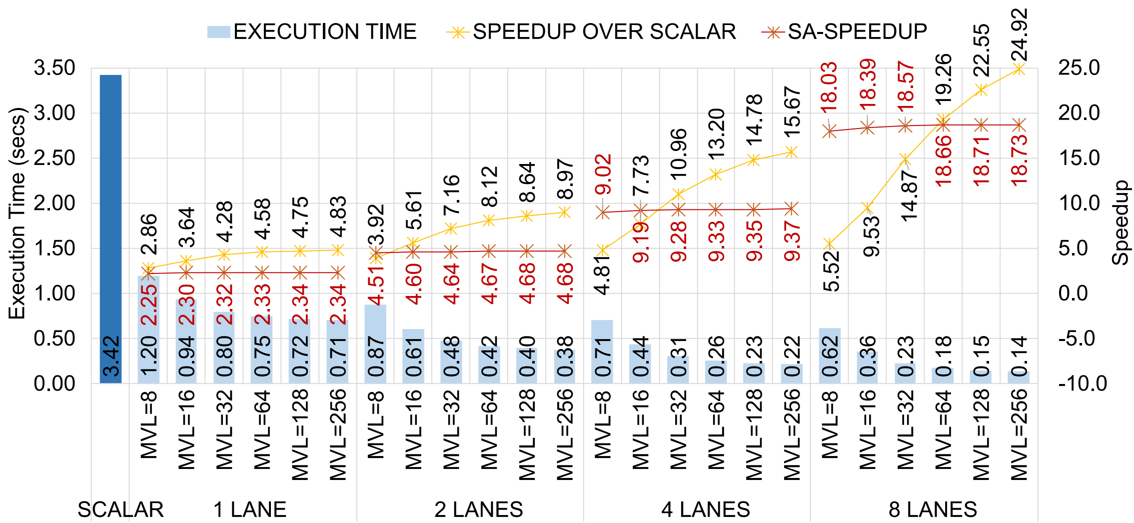

Figure 2.10 *Blackscholes runtime/speedup for different configurations.*

Figure 2.11 shows the performance/mm^2 efficiency metric for Blackscholes. In this case, the most efficient configuration corresponds to one lane with MVL=16 elements. In this new scenario with a compute-bound application which takes advantage of the parallel lanes. Note that the efficiency has a slight decrease for multi-lane configurations. This happens because even if the performance increases for multilane configurations, the VRF area for a specific MVL can increase as we increase the number of lanes depending on how the memory macros are organized. For example, a configuration with MVL=256 leads to a VRF of 128 KBs no matter the number of lanes. However, the organization is different for each multi-lane configuration. For one lane, it has been implemented as eight 4R/2W 64-bit*2048-entries memory structures. For two lanes, it has been implemented as eight 4R/2W 64-bit*1024-entries memory structures per lane. For four lanes, it has been implemented as eight 4R/2W 64-bit*512-entries memory structures per lane. Finally, for eight lanes, it has been implemented as eight 4R/2W 64-bit*256-entries memory structures per lane. Note that in all the configurations we have

eight memory structures per lane in order to read in parallel all of them and allocate eight elements in the source buffers to feed the functional units. Then as we increase the number of lanes, the memory macros become smaller, but represents a higher number (from 8 memory macros for 1 lane to 64 memory macros for 8 lanes). However, there is an internal circuitry such as the pre-charge logic and the sense amplifiers, which represent an important area overhead in the SRAM memory structure, which is independent to the number of entries. Then as more memory macros are required, these internal circuitry is replicated for each macro.

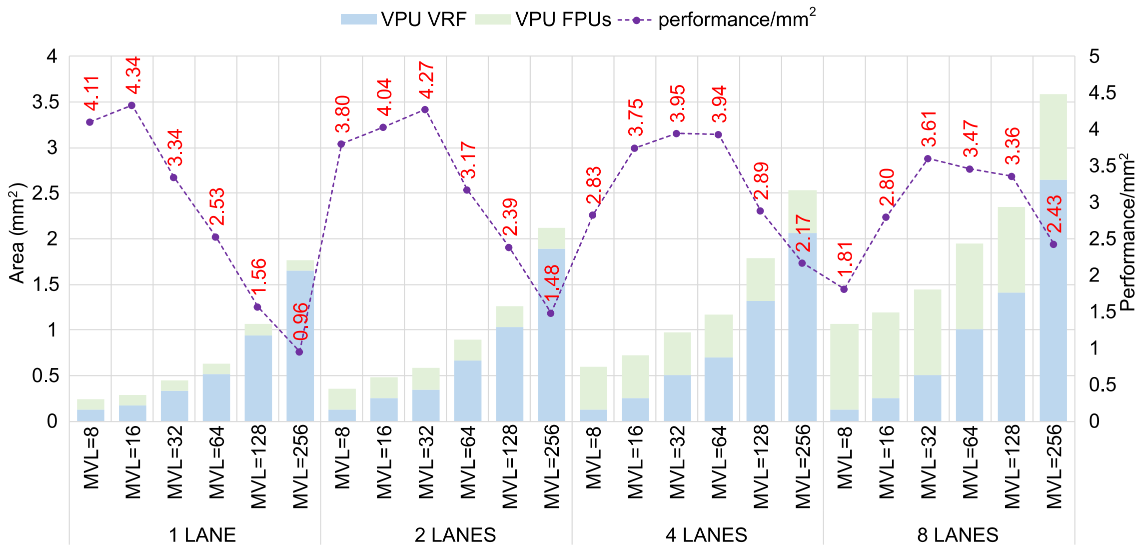

Figure 2.11 *Blackscholes performance/mm² efficiency.*

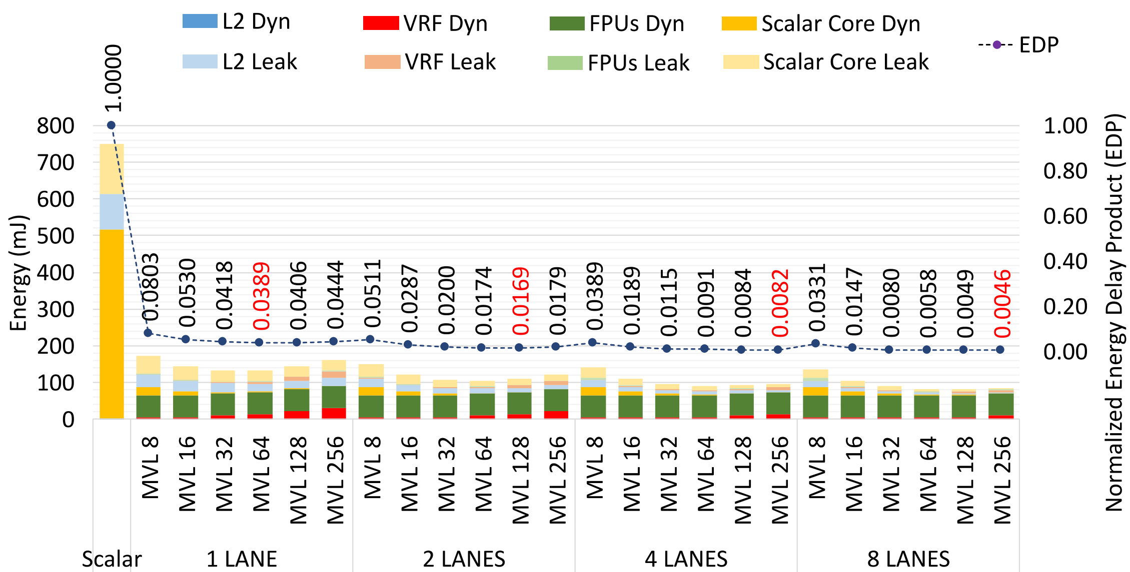

Figure 2.12 *Blackscholes energy consumption (left axis) and normalized Energy Delay Product (right axis) for different configurations.*

Figure 2.12 shows energy consumption (left axis) and Energy Delay Product (EDP) efficiency metric (right axis) for Blackscholes. Contrary to Axpy, Blackscholes is a compute bound application where for the vectorized versions most of the dynamic energy is consumed by the floating-point units. As the MVL is increased, the dynamic and leakage energy is reduced for the scalar core since Blackscholes features a very high percentage of vectorization. The overall instruction count is reduced from 3,180,479,876 for the scalar version to 4,352,913 for the vectorized version with MVL=256.

It is clear that for compute-bound applications such as Blackscholes, long vector length processors combined with several lanes (i.e., MVL=256 and 8 Lanes) not only provide the best performance but also energy savings when compared with short vector length processors, as shown by the EDP results in Figure 2.12.

2.4.3.3 Canneal

Contrary to Blackscholes, which benefited from any MVL because of its high DLP, Canneal can only exploit shorter vector lengths implementations. As presented in Table 1, this is an irregular DLP application, which increases the complexity to improve the performance even for the vectorized version. Although the analysis presented in Section 2.3.3.3 mentions that the largest VL for this application is 22 elements, the application was executed with all the configurations. This is done to show the behavior when applications with short vectors are executed in hardware for large vectors.

As was pointed out in Section 2.3.3.3, the configuration with MVL=8 elements has a SA-speedup of 0.63x, and this speedup decreases as the MVL parameter is increased. Results presented in Figure 2.13 exhibit a behavior close to that expected. In this case, the configuration with an MVL=16 obtained the best performance, achieving 1.51x of speedup over the scalar version for the single lane configuration and 1.95x of speedup for the eight-lane configuration. As the MVL parameter was increased, the speedup started to decrease as expected. This difference with the SA-speedup suggests that about half of the remaining scalar instructions can be amortized beneath vector execution.

In general, the low speedup is mainly because this application has an irregular DLP pattern, including intensive indexed memory accesses, which are very expensive in terms of latency. Furthermore, performance is limited by the large number of scalar instructions executed, and as mentioned before, about half of scalar instructions cannot be amortized underneath vector execution. This is because the scalar core waits for the result from the vector engine to compute the final routing cost and decides if the current element should be swapped or not. For larger MVL configurations, the speedup starts to decrease as was expected. In fact, for MVL=128 and 256, the scalar version is faster

than the vectorized version since the number of Vector operations increases notably because of the complementary instructions added by the compiler and which uses the MVL allowed by the hardware.

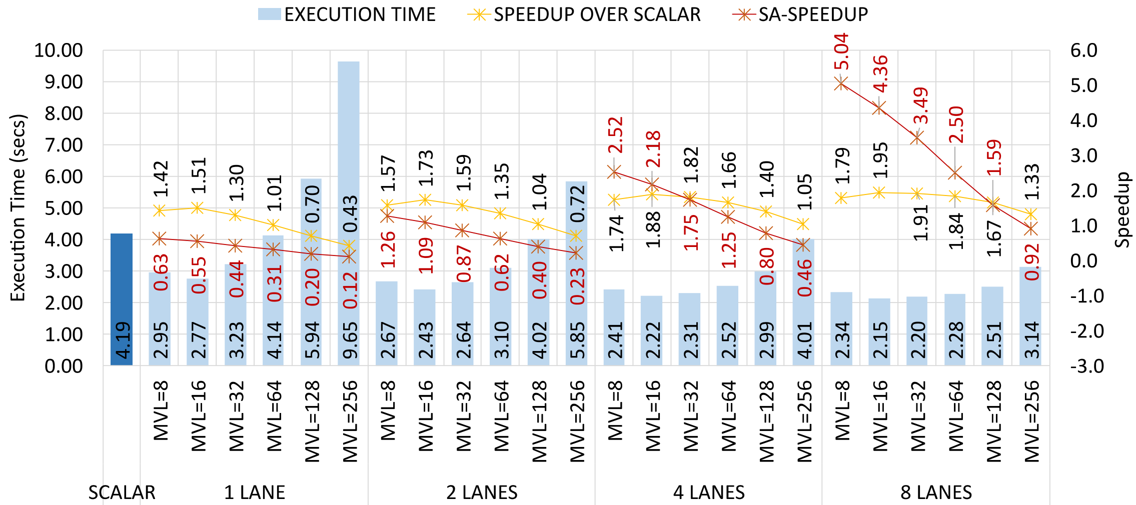

Figure 2.13 *Canneal runtime/speedup for different configurations.*

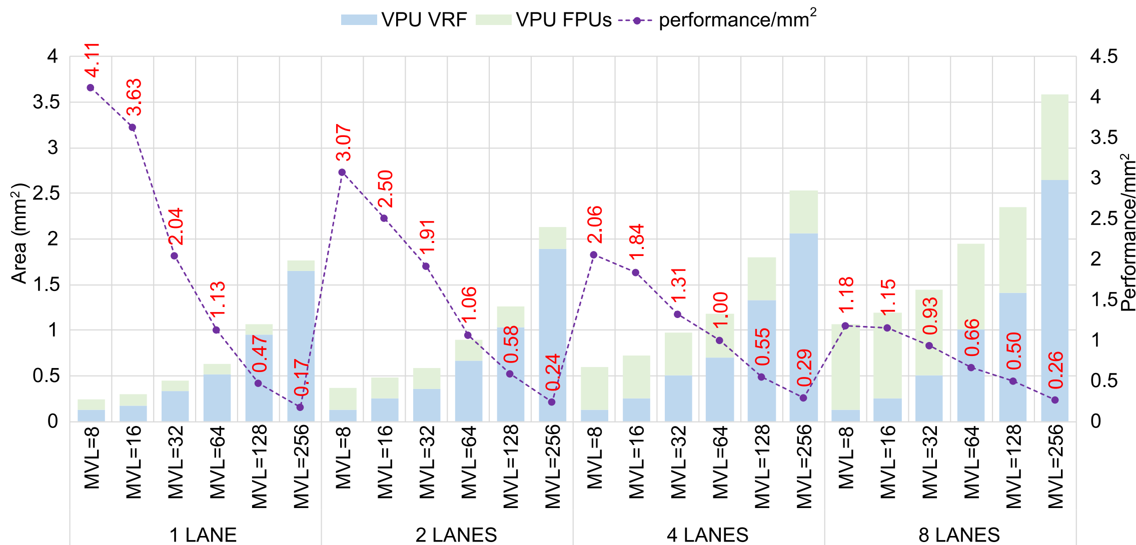

Figure 2.14 *Canneal performance/mm² efficiency.*

Figure 2.14 shows the performance/mm² efficiency metric for Canneal. In this case, the most efficient configuration corresponds to one lane with MVL=8 elements. This application lets us visualize that the hardware is efficient only for MVL=8 configuration. Doubling the number of lanes from one to two gets a very close efficiency. However, when setting four or eight lanes configuration, the efficiency drops drastically. This happens because, for this short vector configuration, the start-up time is costly. Then,

having only one lane helps to start-up time represents a smaller percentage because arithmetic operations spend more time in execution.

Figure 2.15 shows energy consumption (left axis) and Energy Delay Product (EDP) efficiency metric (right axis) for Canneal. Contrary to Axpy and Blackscholes that are a very high DLP applications compatible with long vector lengths, Canneal features short vectors. As shown in the performance results, performance degradation is the result of executing this application on a long vector length hardware. Now, we are going to highlight some energy issues.

First, it is clear that the most energy-efficient configuration for Canneal is MVL=16 no matter the lane configuration. From MVL=32 to MVL=256, there is a considerable energy consumption increase, especially for configurations with few lanes. As mentioned before, complementary instructions are added by the compiler, using the MVL allowed by the hardware. One of these complementary instructions is to copy a complete vector register. Then, when MVL=256, the 256 elements are copied to the destination register no matter that only are required few of them. This behavior can be seen on the VRF dynamic energy, which increases for MVL equal or bigger than 32 because more read/write operations to the VRF are required for longer vector length implementations. FPU dynamic energy is the same for all the configurations since copy instructions do not require a functional unit to be executed.

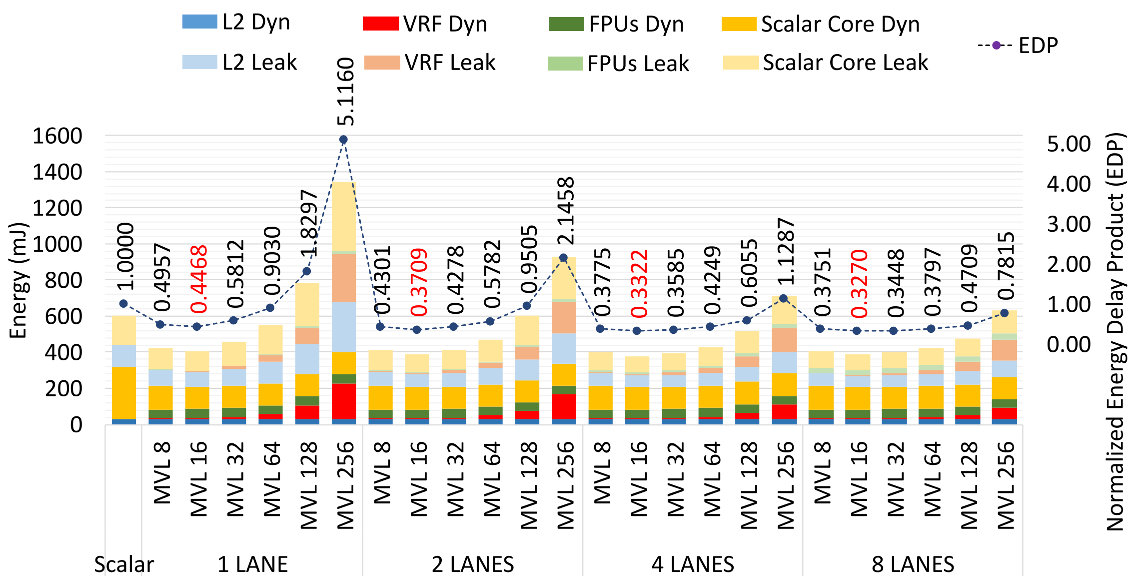

***Figure 2.15** Canneal energy consumption (left axis) and normalized Energy Delay Product (right axis) for different configurations*

Finally, scalar core dynamic energy is reduced when compared with the scalar version. However, vectorized versions achieve the best configuration at MVL=16, and

for bigger configurations, it remains constant since the same number of instructions are executed on the scalar pipeline.

2.4.3.4 Jacobi-2D

Results presented in Figure 2.16 show a speedup of 2.25x for MVL=8 and one-lane configuration, which is higher than the SA-speedup (1.12X). As the MVL increases, the speedup increases up to 3.54X, as was expected. This happens because most of the remaining scalar instructions can be amortized beneath vector execution, and the amount of latency amortized per vector instruction is maximized. Regarding the increase in the number of lanes and the expected linear increase, it is not completely accurate. Configurations with small and medium-size MVL do not benefit considerably from adding more lanes, unlike configurations that use large vectors. Looking at the configurations with MVL=256, almost a linear speedup increase can be seen. As mentioned before, the speedup increase is strongly related to the start-up time. It is incurred in all configurations, but for larger MVL configurations, the start-up time becomes negligible.

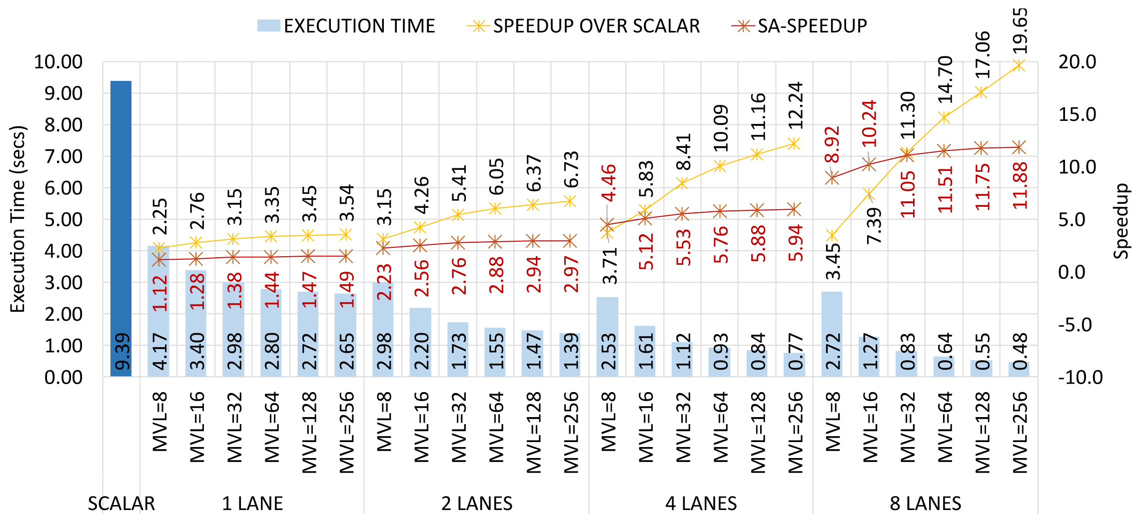

Figure 2.16 *Jacobi-2D runtime/speedup for different configurations.*

Figure 2.17 shows the performance/mm^2 efficiency metric for Jacobi-2D. This application presents similar results to Blackscholes. In this case, the most efficient configuration corresponds to one lane with MVL=16 elements. In this scenario with a compute-bound application, having more lanes is worthy. Although the efficiency is slightly less for multi-lane configurations because of the area increase in the VRF explained before. However, the efficiency is very close.

Figure 2.18 shows energy consumption (left axis) and Energy Delay Product (EDP) efficiency metric (right axis) for Jacobi-2D. Similar to Blackscholes, Jacobi-2D is a compute-bound application where for the vectorized versions, most of the dynamic

energy is consumed by the floating-point units. As the MVL is increased, the dynamic and leakage energy is reduced for the scalar core since Jacobi-2D features a very high percentage of vectorization. The overall instruction count is reduced from 4,660,908,013 for the scalar version to 45,798,013 for the vectorized version with MVL=256. Long vector length processors combined with several lanes (i.e., MVL=256 and 8 Lanes) provide not only the best performance but also energy savings when compared with short vector length processors, as shown by the EDP results in Figure 2.18.

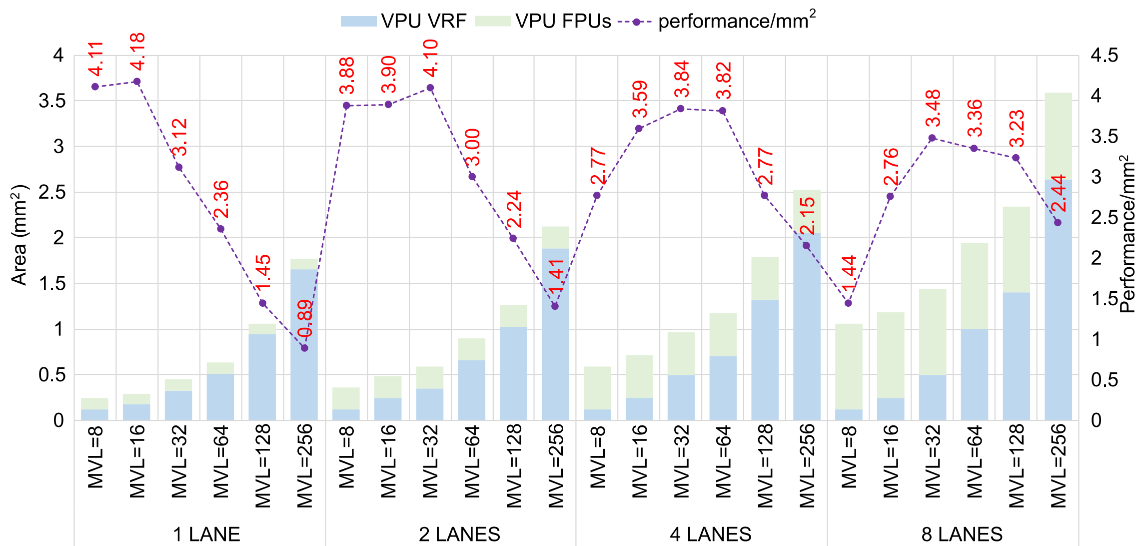

Figure 2.17 *Jacobi-2D performance/mm² efficiency.*

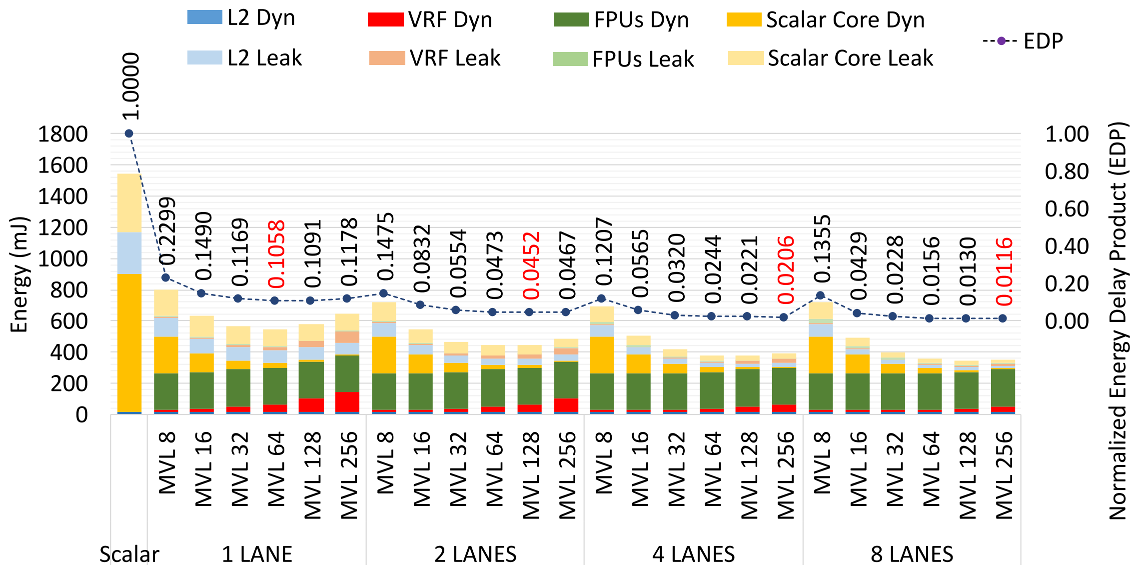

Figure 2.18 *Jacobi-2D energy consumption (left axis) and normalized Energy Delay Product (right axis) for different configurations.*

2.4.3.5 LavaMD2

Results presented in Figure 2.19 show a speedup of 2.78x for MVL=8 and one-lane configuration, which is higher than the SA-speedup (1.86x). As the MVL increases, a slight speedup increase (3.3x) can be seen for MVL=16. However, for the larger MVL configurations, there is a speed-up decrease as was expected. This is mainly caused by the increase in the number of *Total Vector Operations*. Regarding the expected speed-up increase for the multi-lane configuration, it is very close to the initial conclusions. Configurations with small MVL do not benefit considerably from adding more lanes, unlike configurations that use medium-size vectors. On the contrary, configurations with long vectors show a performance decrease because the largest VL of the application is 64 elements, then for MVL=128 and 256 elements, extra operations are executed.

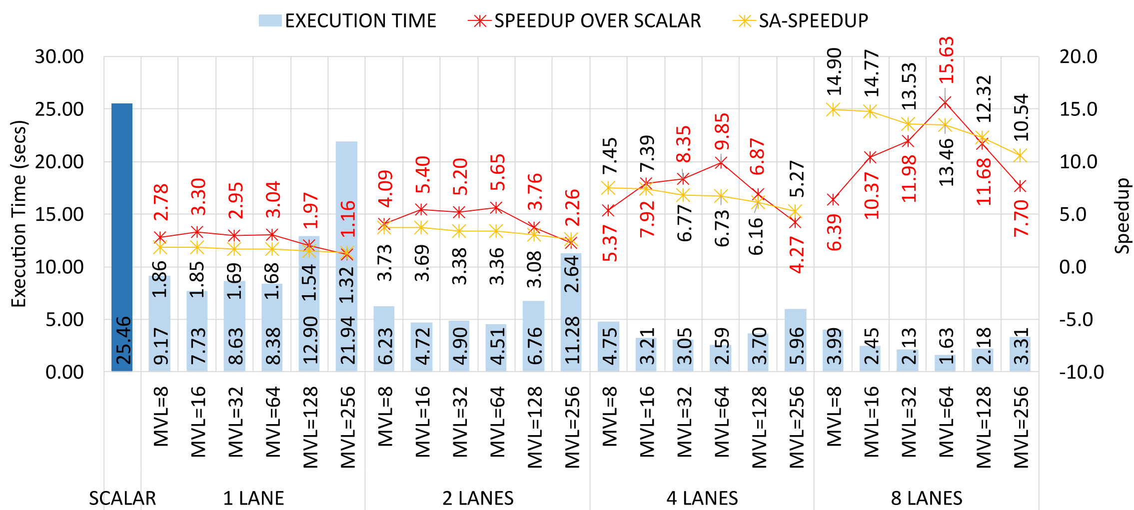

Figure 2.19 *LavaMD2 runtime/speedup for different configurations.*

Figure 2.20 shows the performance/mm² efficiency metric for LavaMD2. In this case, the most efficient configuration corresponds to one lane with MVL=8 elements. It is clear that a compute-bound application benefits from multilane implementations.

Figure 2.21 shows energy consumption (left axis) and Energy Delay Product (EDP) efficiency metric (right axis) for LavaMD2. LavaMD2 is a high DLP application compatible with short and medium-size vector lengths. LavaMD2 is a compute-bound application where most of the dynamic energy is consumed by the floating-point units for the vectorized versions. As the MVL is increased, the dynamic and leakage energy is reduced for the scalar core since Jacobi-2D features a very high percentage of vectorization. The overall instruction count is reduced from 24,615,519,089 for the scalar version to 283,248,849 for the vectorized version with MVL=256.

As shown in the performance results (Figure 2.19), performance degradation is the result of executing this application on a long vector length hardware (MVL=128 and MVL=256). This happens because the longer VL for this application is 48 elements, being MVL=64 the most efficient configuration for 2,4 and 8 lane configurations. Note that for MVL=64, there is a slight increase in the VRF dynamic energy compared with shorter MVL configurations; however, benefits can be seen since dynamic energy in the scalar core is reduced. From MVL=64 to MVL=256, there is an energy consumption increase, especially for configurations with few lanes. As mentioned before, complementary instructions are added by the compiler, using the MVL allowed by the hardware.

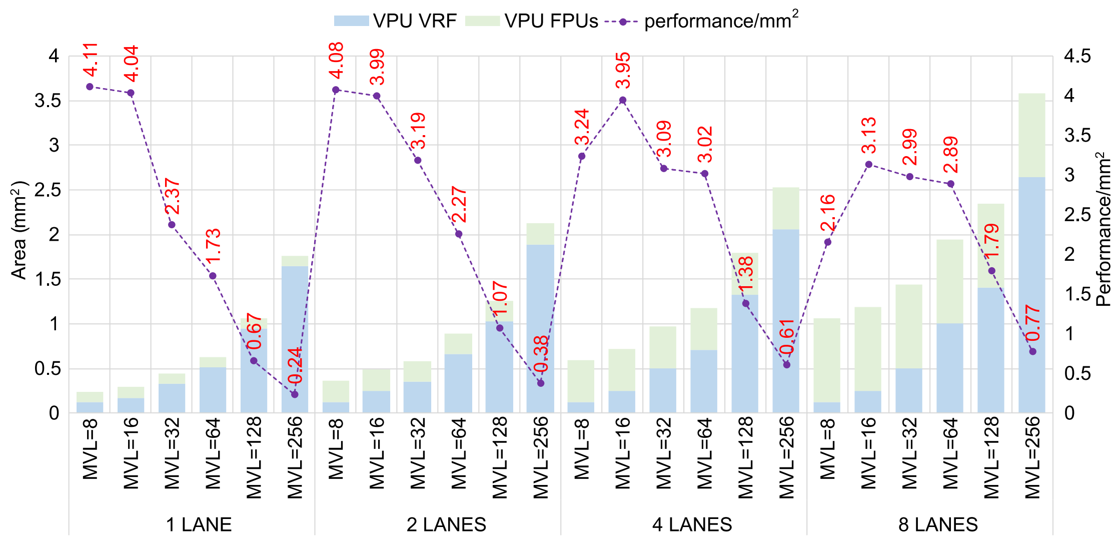

Figure 2.20 *LavaMD2 performance/mm² efficiency.*

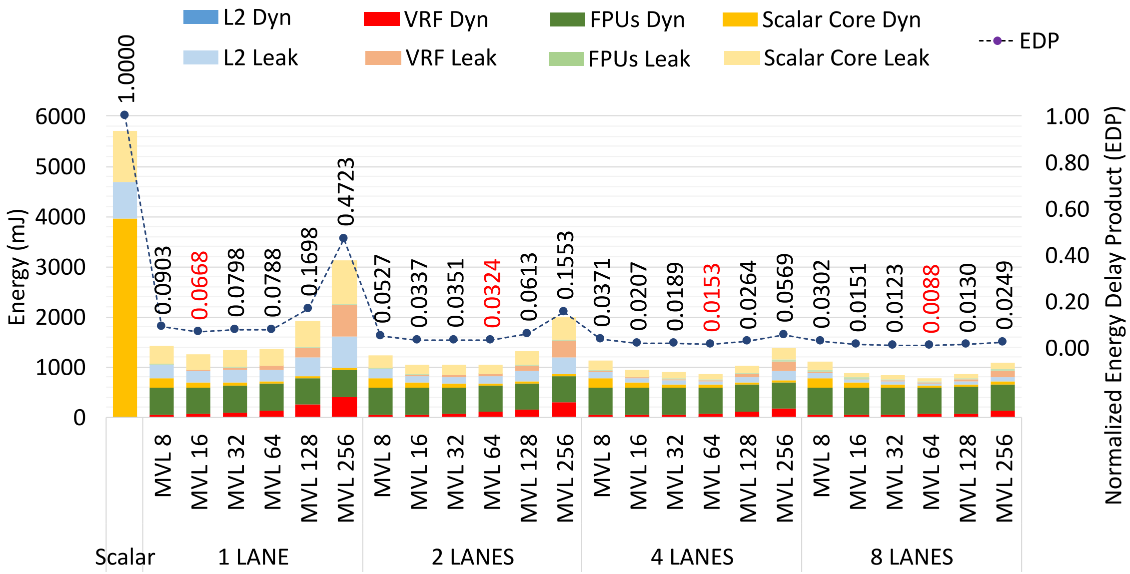

Figure 2.21 *LavaMD2 energy consumption (left axis) and normalized Energy Delay Product (right axis) for different configurations.*

2.4.3.6 Pathfinder

As mentioned in Section 2.3.3.6, the Pathfinder application is interesting since it presents a high percentage of vector element manipulation instructions. Thus, it is possible to evaluate the implemented interconnection topology between lanes. In this case, the base model is using a ring interconnection, where in order to move one element to another lane, the cost in latency is the distance between the origin and the destination lanes. Several elements can be computed in parallel in multiple lanes. In this particular case, the algorithm makes use of slide1up and slide1down vector instructions, where the elements are displaced by only one position. In that sense, the ring interconnection is enough to get a good speedup for this application since it will require only one cycle to move one element from the current lane to the destination lane. Also, each lane can send one element to the ring interconnection in the same cycle, and one cycle later, and all the lanes will receive their corresponding data. It is clear that these operations can take advantage of the parallel lanes.

The results presented in Figure 2.22, exhibit a behavior very close to the SA-speedup (2.71X) for MVL=8 and one-lane configuration. As the MVL parameter increases, higher speedups can be achieved because scalar instructions can be amortized beneath vector execution, and the amount of latency amortized per vector instruction is maximized. For multi-lane configurations, configurations with small and medium-size MVL do not benefit considerably from adding more lanes, unlike configurations that use large vectors. As mentioned several times before, this is because,

in all configurations, the start-up time has to be paid, but for larger MVLs, the start-up time becomes negligible.

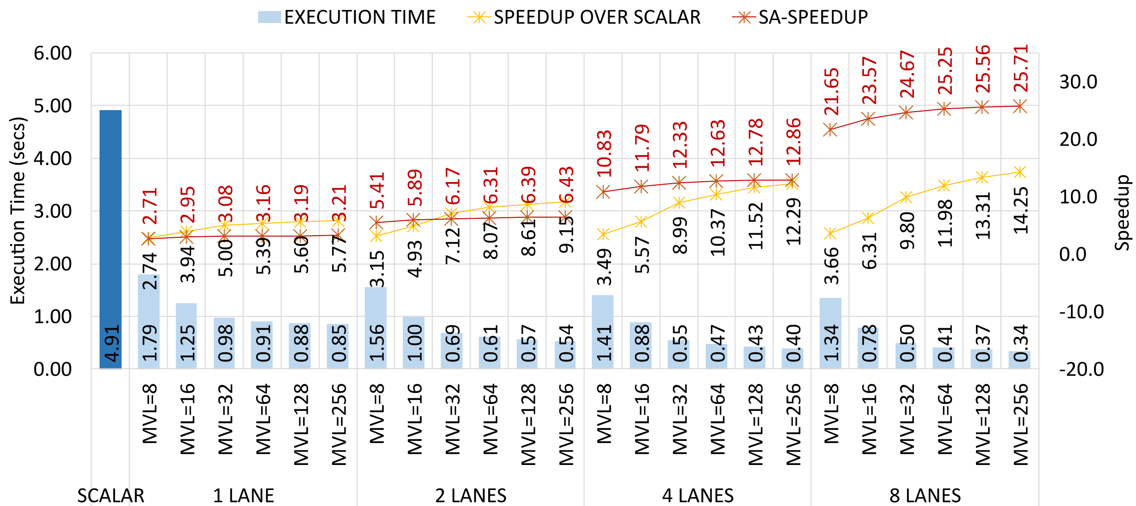

Figure 2.22 *Pathfinder runtime/speedup for different configurations.*

Figure 2.23 shows the performance/mm^2 efficiency metric for Pathfinder. In this case, the most efficient configuration corresponds to one lane with MVL=16 elements. As the number of lanes increases, MVL=32 becomes the most efficient configuration. Although the efficiency decreases for multi-lane configurations, it is worth adding more lanes.

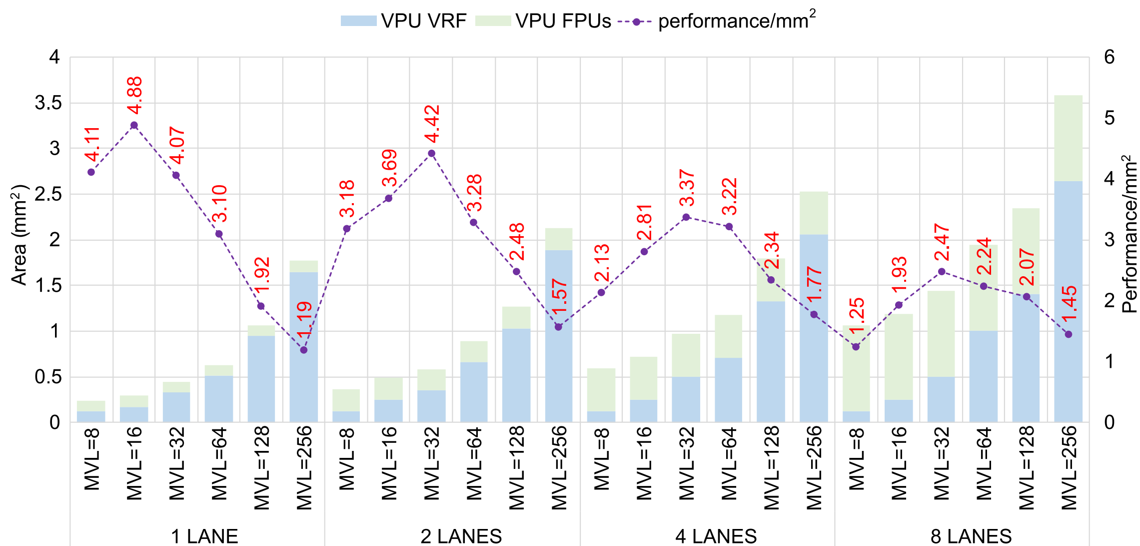

Figure 2.23 *Pathfinder performance/mm^2 efficiency.*

Figure 2.24 shows energy consumption (left axis) and Energy Delay Product (EDP) efficiency metric (right axis) for Pathfinder. As the MVL is increased, the dynamic and

leakage energy is reduced for the scalar core since Pathfinder features a very high percentage of vectorization. The overall instruction count is reduced from 5,433,477,921 for the scalar version to 17,681,517 for the vectorized version with MVL=256.

Vectorizing Pathfinder provides better performance levels and considerable energy savings even for short vector length processors. Energy savings are enhanced as we increase the MVL and the number of parallel lanes. For one-lane configuration, the most energy-efficient configuration corresponds to MVL=64. The most energy-efficient configuration for two lanes configuration corresponds to MVL=128. Finally, for four and eight-lane configurations, the most energy-efficient configuration corresponds to MVL=256, as shown by the EDP results presented in Figure 2.24.

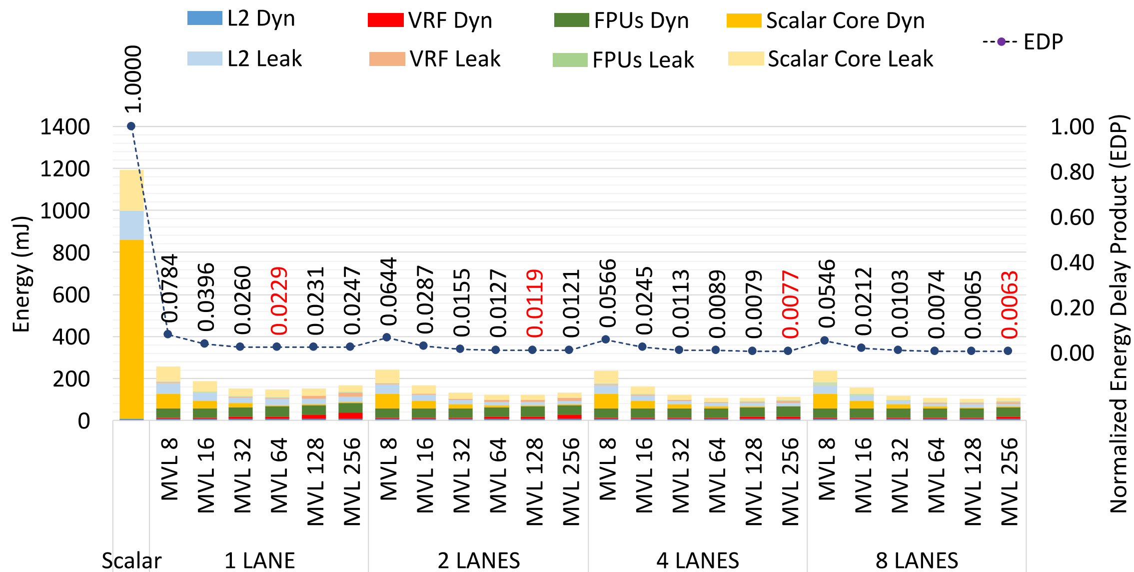

Figure 2.24 *Pathfinder energy consumption (left axis) and normalized Energy Delay Product (right axis) for different configurations.*

2.4.3.7 Particle-Filter

Particle-Filter (PF) is an interesting application to analyze because it combines the use of expensive operations like logarithm, cosine, and square root with special operations with masks. *vfirst.m* and *vpopc.m* vector mask instructions write the final results to a scalar register. In that sense, these operations cause the scalar core to stall because of the higher number of scalar dependencies.

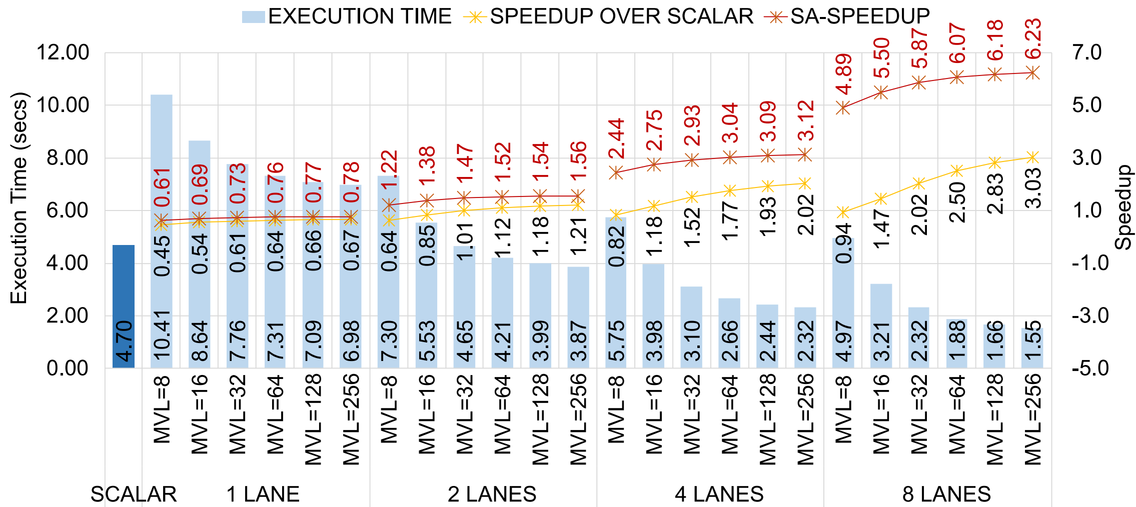

Figure 2.25 *Particle-Filter runtime/speedup for different configurations.*

According to the static code analysis presented in Section 2.3.3.7, the SA-speedup is 0.61x for MVL=8 and one-lane configuration. There is no speedup over the scalar version, as can be seen in Figure 2.25. In general, all the SA-speedups are higher than those already obtained for the different configurations. As previously suggested, the final speedup would be affected by a considerable number of stalls in the scalar core, which would not be removed until the vector engine finishes the computation of the current iteration. Based on the results, it can be concluded that for applications such as Particle-Filter in which many of the remaining scalar instructions cannot be amortized beneath vector execution because of the generated scalar dependencies, there could be significant improvements by using an out-of-order superscalar core instead of a superscalar in-order core. For the out-of-order case, it would be possible to advance independent scalar instructions and also continue feeding the vector engine.

Figure 2.26 shows the performance/mm^2 efficiency metric for Particle-Filter. In this case, the most efficient configuration corresponds to one lane with MVL=8 elements. As the number of lanes increases, MVL=32 becomes the most efficient configuration. Although the efficiency decreases for multi-lane configurations, it is worth adding more lanes.

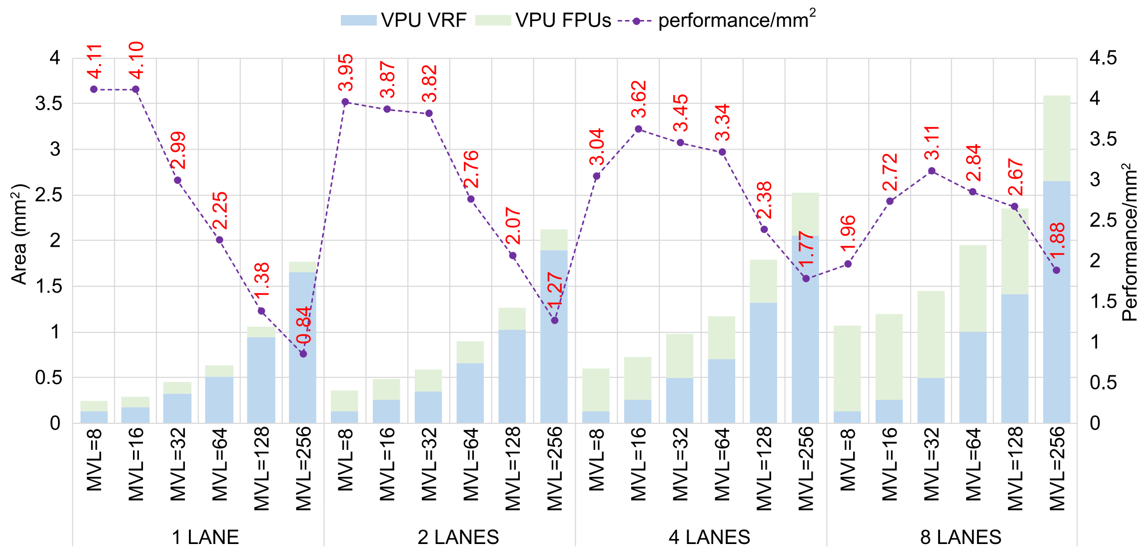

Figure 2.26 *Particle-Filter performance/mm² efficiency.*

Figure 2.27 shows energy consumption (left axis) and Energy Delay Product (EDP) efficiency metric (right axis) for Particle-Filter. This application presents an interesting behavior to analyze. First, as presented in the instruction-level characterization of the ParticleFilter application (Table 16), vectorizing this application incurs in executing more individual operations when compared with the scalar version. As explained before in the corresponding static analysis (Section 2.3.3.7), these extra operations are created when mapping the sequential search in a vector fashion.

For one-lane configuration, performance is worse for all the MVL configurations mainly for the following reasons: First, since performance is worse, leakage energy increases in general for all the modules. Second, the overall instruction count is reduced by vectorizing the application, however, it is not a dramatic reduction as in other high DLP applications such as Blackscholes or Pathfinder. For the scalar version we have 5,433,477,921 instructions, while for the vectorized version there are 3,241,372,346 for MVL=8, and 617,873,199 for MVL=256.

As we increase the number of lanes, performance improves, being better for longer MVL configurations. Then, energy savings can be seen, being the most efficient configuration of eight lanes with MVL=256, as shown by the EDP results presented in Figure 2.27.

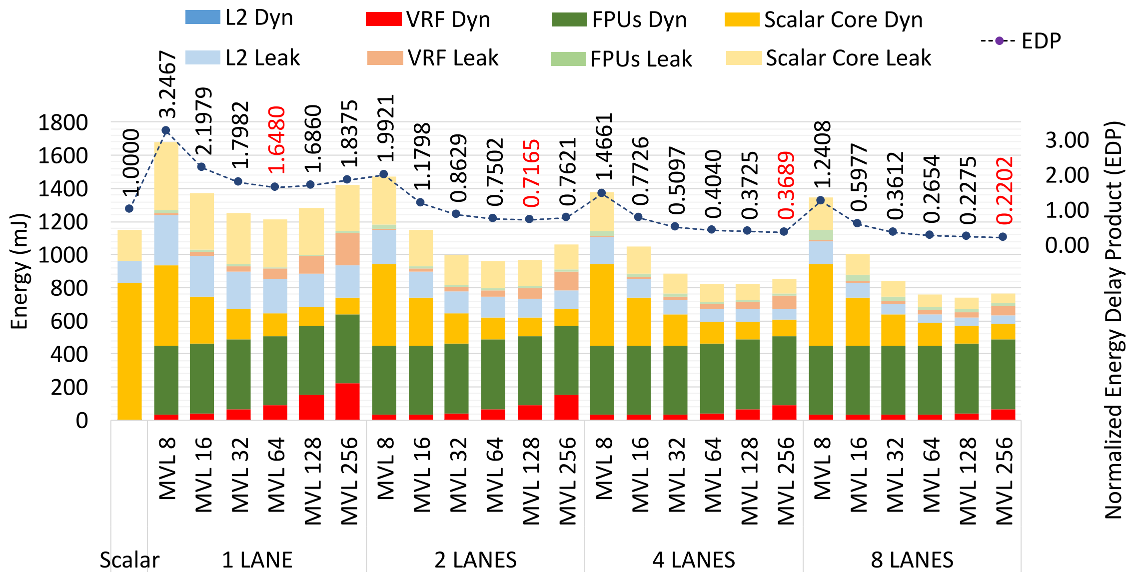

Figure 2.27 *Particle-Filter energy consumption (left axis) and normalized Energy Delay Product (right axis) for different configurations.*

2.4.3.8 Somier

Somier results are presented in Figure 2.28. On the one hand, for one lane configuration and the different MVL configurations, the speedups are very close to the SA-speedups. As the MVL increases, slight improvements can be seen. On the other hand, the addition of parallel lanes provides a marginal speedup increase. As mentioned before, the speedup can be mainly limited by: (1) the memory subsystem since this application is memory bound, and (2) the high number of scalar instructions which cannot be amortized beneath vector execution.

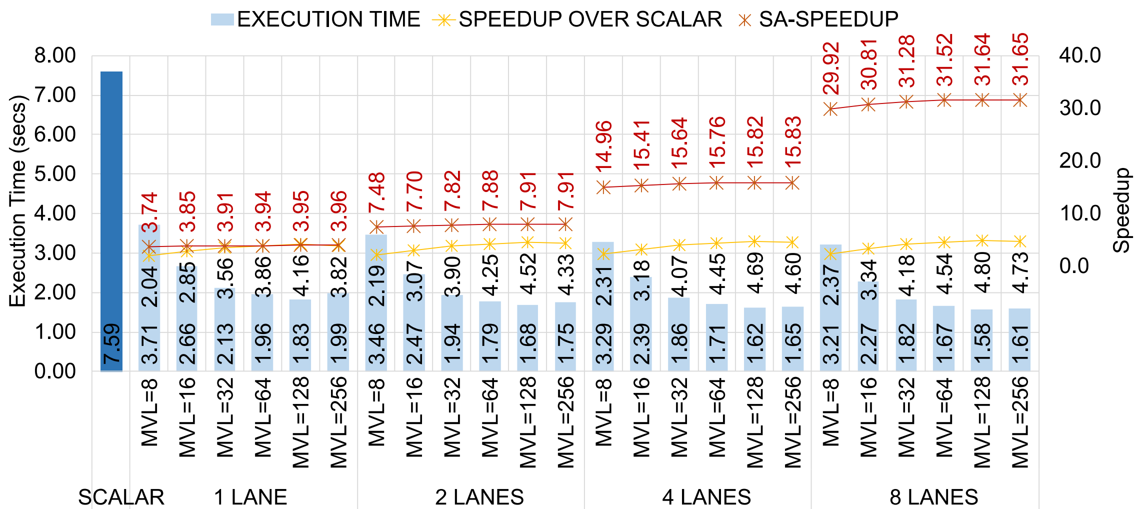

***Figure 2.28** Somier runtime/speedup for different configurations.*

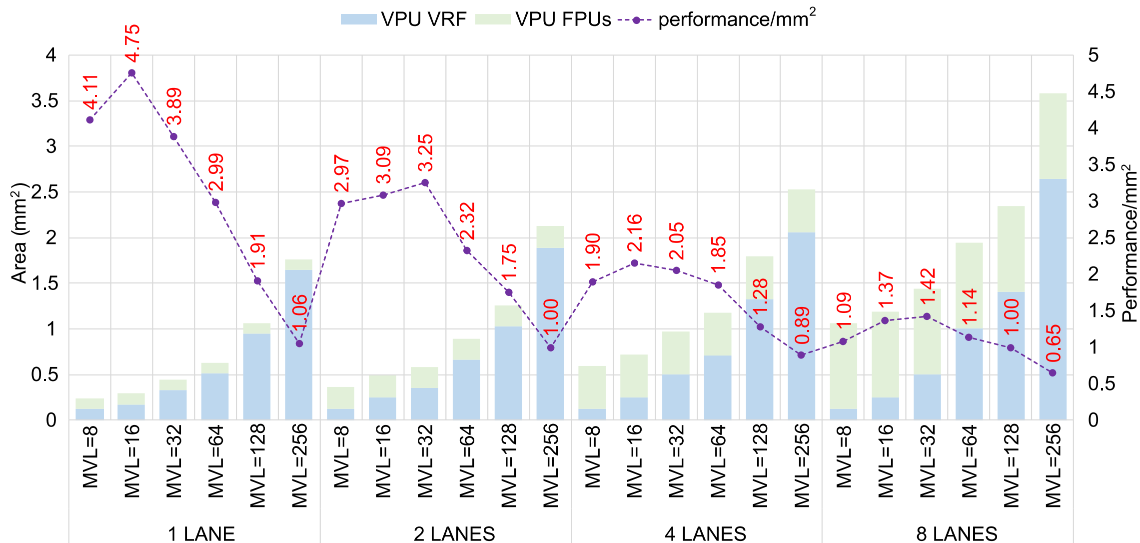

***Figure 2.29** Somier performance/mm² efficiency.*

Figure 2.29 shows the performance/mm² efficiency metric for Somier. In this case, the most efficient configuration corresponds to one lane with MVL=16 elements. Doubling the number of lanes causes the efficiency to drop drastically. This happens because Somier is a memory-bound application. Then increasing the number of functional units does not add incredible speedups for making the eight lanes configuration affordable.

Figure 2.30 shows energy consumption (left axis) and Energy Delay Product (EDP) efficiency metric (right axis) for Somier. This application presents an interesting behavior to analyze. First, as presented in the instruction-level characterization of the Somier application (Table 18), for the scalar version we have 6,254,373,928 instructions, while

for the vectorized version there are 850,576,201 for MVL=8, and 647,304,985 for MVL=256. Between both vectorized versions there is a small difference in the final instruction count. This behavior is because when mapping the *force_contribution* function in a vector fashion, there is a section of scalar code that cannot be vectorized, which remains constant executing around 640,300,015 scalar instructions. Therefore, the execution is serialized, and the vector processor cannot start to do the final computation before the scalar core finalizes execution. In that sense, this section of the code is not possible to hide beneath vector execution. In fact, note that the dynamic energy consumed by the scalar core represents an important contribution to the final energy consumption.

In general, good energy savings can be seen by vectorizing Somier, but limited by the code that is not vectorizable.

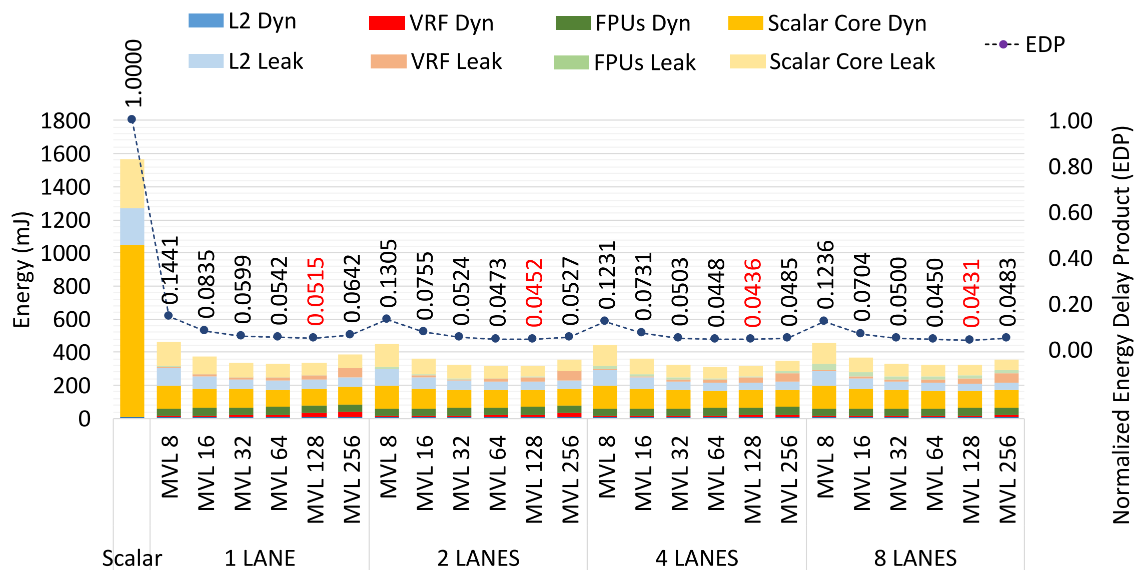

Figure 2.30 *Somier energy consumption (left axis) and normalized Energy Delay Product (right axis) for different configurations.*

2.4.3.9 Streamcluster

Results are presented in Figure 2.31. For MVL=8 and one lane configuration, a speedup of 2.17x is obtained, which is very close to the SA-speedup (1.67). As the MVL slightly increases, improvements can be seen. However, from MVL=64 to MVL=256, the speedup starts decreasing. Several factors are causing this low speedup increase and the sudden decrease when MVL=64. First, for larger vector lengths, the number of vector operations increases notably, then, more time is needed to execute those vector operations. Secondly, there is a reduction operation after the "for" loop. This operation is executed only once regardless of the VL size. Then, for short vectors, this operation

has relatively less overhead; but for long vectors, this operation consumes more time. Finally, the resultant scalar value of the reduction is sent immediately to the scalar core to compute the cost of opening a new center. Thus, in this step, the core stalls until it receives this data, then computes the cost, and finally iterates again.

The addition of parallel lanes helps to achieve better speedups, especially for long vector length configurations. As mentioned before, this application is memory-bound. In combination with intensive communication with the scalar core, the speedup is mainly limited by: (1) the memory subsystem, and (2) issue scheme in the scalar core.

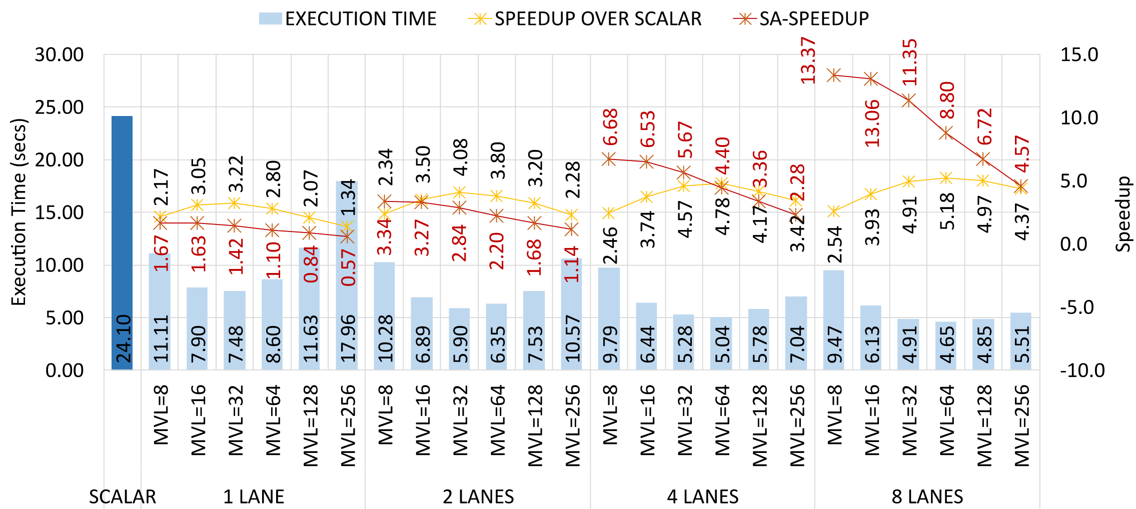

Figure 2.31 *Streamcluster runtime/speedup for different configurations.*

Figure 2.32 shows the performance/mm^2 efficiency metric for Streamcluster. For one lane configuration, the most efficient configuration corresponds to MVL=16. As we increase the number of lanes, efficiency starts raising. For eight lane configuration, the most efficient configuration corresponds to MVL=32. This is an expected result since Streamcluster is compatible only with short and medium-size vector lengths.

Figure 2.33 shows energy consumption (left axis) and Energy Delay Product (EDP) efficiency metric (right axis) for Streamcluster. As presented in the instruction-level characterization of the Streamcluster application (Table 20), this application is compatible with short and medium-size vector lengths. For MVL=64, MVL=128, and MVL=256, the instruction count remains equal. However, the number of vector operations increases as described before. This behavior leads consume more time to execute the application, and therefore, increase the overall leakage energy. Note that the dynamic energy consumed by the scalar core represents an important contribution to the final energy consumption.

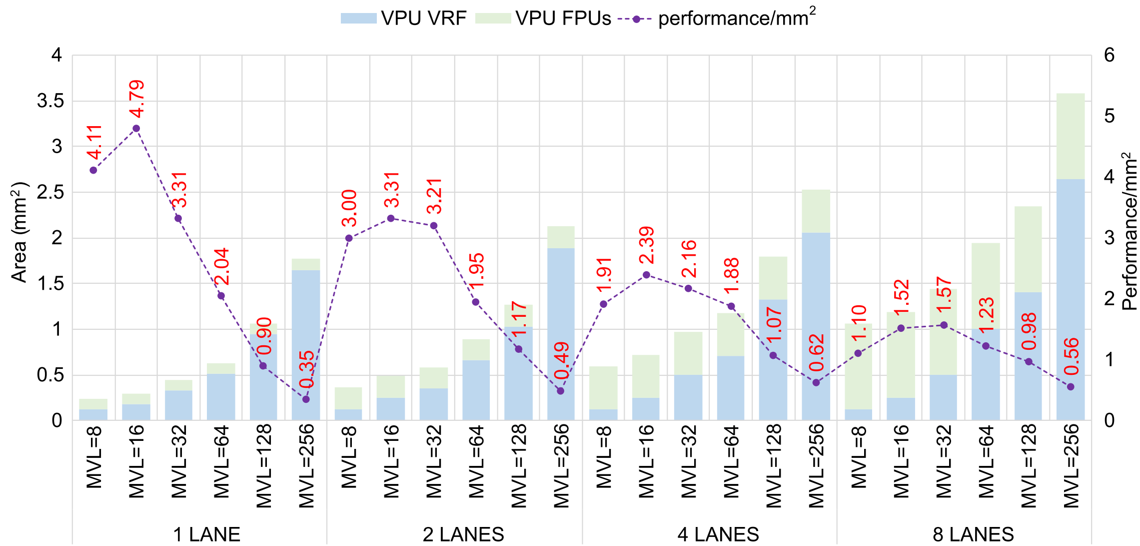

***Figure 2.32** Streamcluster performance/mm² efficiency.*

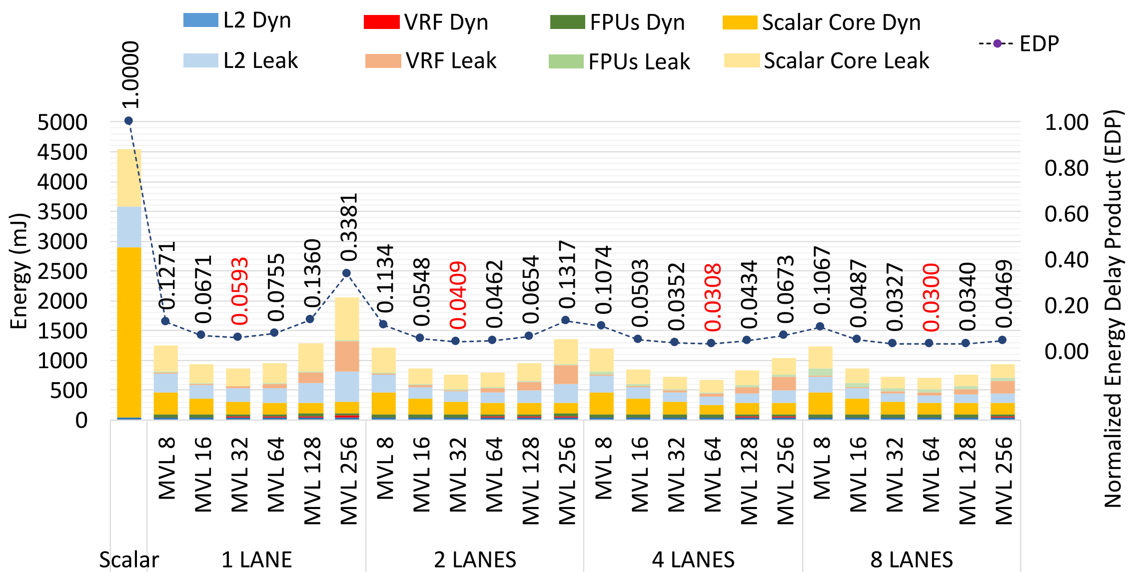

***Figure 2.33** Streamcluster energy consumption (left axis) and normalized Energy Delay Product (right axis) for different configurations.*

2.4.3.10 Swaptions

For the scalar implementation, a block size of 32 elements presents the best performance. For a larger block size, the L1 cache miss rate increases. For the vectorized version, it is a little bit different. The block size parameter (BLOCK_SIZE) is related to the MVL. Then, a block size of 256 elements gives better performance, achieving a 1.6x speedup over the scalar version regardless of the cache miss increase for a one-lane configuration. In [44] a miss rate study is presented for all programs in the

PARSEC benchmark suite, where for Swaptions, it is shown that for a cache size of 1MB, the miss rate is reduced notably when comparing versus smaller cache sizes.

It is interesting to compare the obtained versus the expected results. The obtained speedup for MVL=8 and one lane configuration is 1.04X, which is lower than the SA-speedup (1.39X). Configurations with small and medium-size MVL do not benefit considerably from adding more lanes, unlike configurations that use large vectors. This is mainly because, in all configurations, the start-up time is incurred. Contrary, for MVL=256 and the different number of lanes almost a linear speedup increase can be seen.

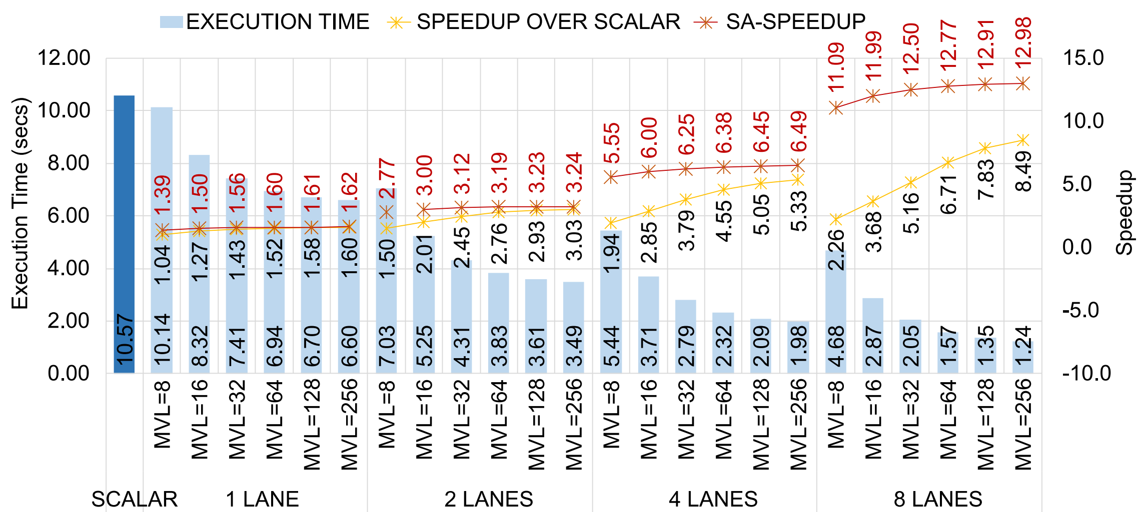

Figure 2.34 *Swaptions speedup for different cache configurations.*

Figure 2.35 shows the performance/mm² efficiency metric for Swaptions. In this case, the most efficient configuration corresponds to one lane with MVL=16 elements. This compute-bound application shows that having more lanes is worthy. Although the efficiency is slightly less for multi-lane configurations, the efficiency does not drop drastically as applications such as Canneal, Somier, and Streamcluster.

Figure 2.36 shows energy consumption (left axis) and Energy Delay Product (EDP) efficiency metric (right axis) for Swaptions. Swaptions is a compute bound application where for the vectorized versions most of the dynamic energy is consumed by the floating-point units. As the MVL is increased, the dynamic and leakage energy is reduced for the scalar core since Swaptions features a very high percentage of vectorization. The overall instruction count is reduced from 11,762,554,240 for the scalar version to 200,489,352 for the vectorized version with MVL=256.

It is clear that for compute-bound applications, long vector length processors combined with several lanes (i.e., MVL=256 and 8 Lanes) not only provide the best performance but also energy savings when compared with short vector length processors, as shown by the EDP results in Figure 2.36.

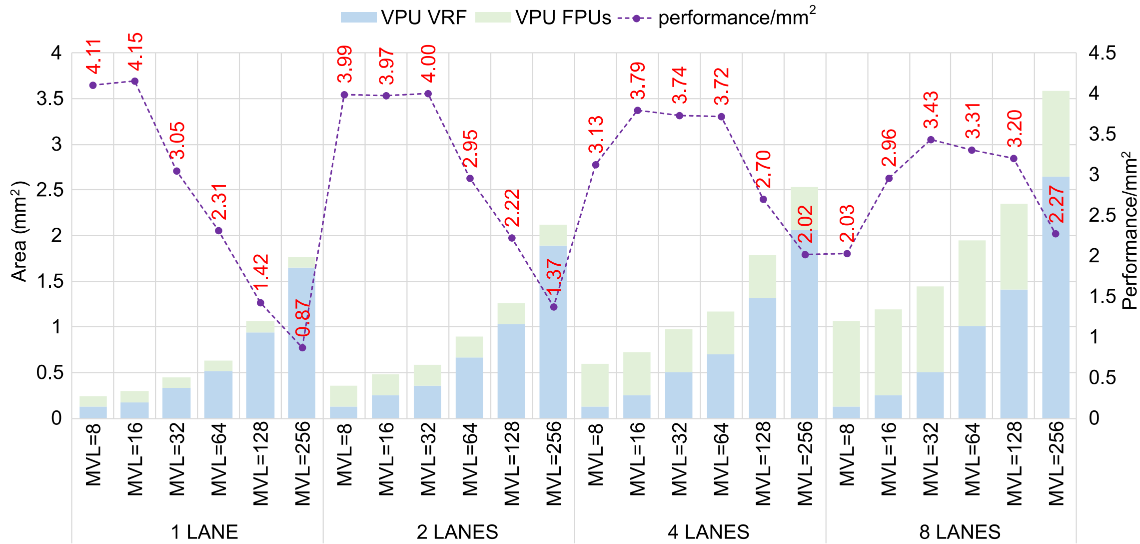

Figure 2.35 *Swaptions performance/mm² efficiency.*

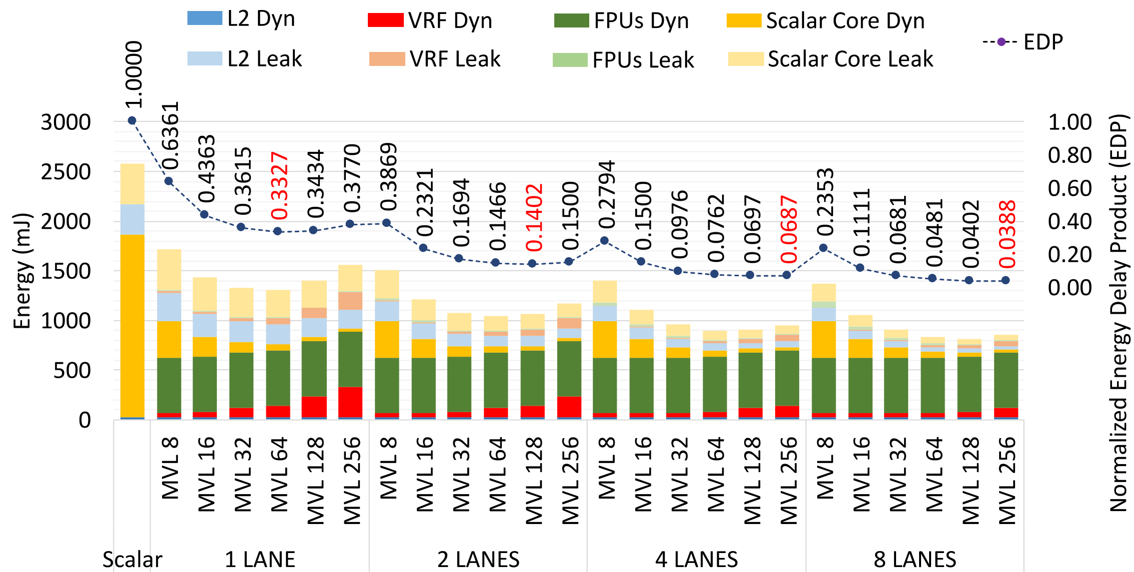

Figure 2.36 *Swaptions energy consumption (left axis) and normalized Energy Delay Product (right axis) for different configurations.*

2.5 Related Work

Stanic [58] presents a set of tools for rapid initial research on vector architectures. The first tool is called VALib, a library that enables hand-crafted vectorization of applications by adding calls which is similar to programming using intrinsics. VALib is not bound to any specific vector ISA. By using this tool, it is possible to collect data for detailed instruction-level characterization and to generate input traces for a second tool called SimpleVector. This second tool is a fast trace-driven simulator used to estimate the execution time of a vectorized application on a candidate vector micro-architecture. This simulator can be used for preliminary evaluation and early parameter exploration but does not provide the accuracy given by execution-driven simulators.

Cebrian [47] [56] presents PARVEC, a vectorized version of the PARSEC benchmark suite. ParVec vectorized 8 of the 13 applications of the PARSEC suite for SSE, AVX and NEON ISA's. Some of them obtained high speed up over the scalar implementation (Blackscholes, Swaptions), others (Fluidanimate, Vips) did not get any speed up improvement mainly because of the nature of the application (organization of the input, not related to size, etc.). The ParVec suite is available for the computer architecture community. The lack of a micro-architectural simulator for those ISA's doesn't allow the computer architecture community to test new ideas at the micro-architectural level. Although the ISA's supported by PARVEC can be classified as Multimedia instruction set extensions, the similarities with the code for RISC-V Vector Extension ISA is extensive, mainly for arithmetic and covert operations. However, there are others like slide operations that usually are not presented in short vector ISA's. In this sense, PARVEC was a great tool for understanding how to vectorize some applications from the PARSEC Benchmark Suite.

The ARM Architecture research team has been working on tools for the community to boost the use of ARM infrastructure in academia, and they have presented these tools in several talks about Vector Architecture for HPC based on Arm SVE [59] [60] [61]. The SVE tool-suite includes the Arm Compiler, the Arm Instruction Emulator, and the Research Enablement kit, which allows system modeling using gem5. The implemented gem5 models correspond to the Armv8-A based CPU timing model (HPI) with support for SVE. The toolkit also includes documentation about how to run the benchmarks, specifically the PARSEC benchmark suite.

2.6 Summary

This chapter has presented three tools. First, an extended version of the gem5 simulator that includes a RISC-V Vector Architecture model. This model can be configured with different parameters (MVL, number of physical registers, number of lanes, etc.), having a flexible and customizable model that fits different research requirements. Second, an extended version of the McPAT framework that includes a Vector Architecture model to obtain area and energy metrics. Third, the RiVEC Benchmark Suite, a collection composed of data-parallel applications from different domains that focuses on benchmarking vector microarchitectures. In addition, a study of every vectorized application and its corresponding execution in the gem5 vector engine model is given, highlighting the degree of vectorization achieved with the applications and the close relationship with the expected and obtained performance. Finally, area and energy metrics are presented for all the presented hardware configurations.

From the previous study we can conclude two main ideas. The vector architectures designed for long vectors are limited to a specialized subset of applications, where relatively high DLP must be present to achieve excellent performance with high efficiency. For example, Blackscholes, Jacobi-2D, and Swaptions, which present regular and high DLP patterns, exploit designs for long vectors. On the contrary, executing applications only compatible with short vectors such as Canneal in hardware for large vectors brings several disadvantages, as was shown in the study. On the other hand, vector architectures designed for short vectors are compatible with applications featuring short, medium, and long vectors. We believe that this wide diversity is one of the main reasons behind the trend of building parallel machines for short vectors.

Therefore, finding a way to obtain a more general vector architecture able to handle different DLP patterns efficiently is a challenge in examination. A vector architecture with the ability to adapt its own structures depending on the application's needs could improve the performance and efficiency for those applications where most of the hardware remains unused.

Chapter 3

Adaptable Vector Architecture

The most efficient way to execute a vectorizable
application is a vector processor

Jim Smith,
International Symposium on Computer Architecture
(1994)

Summary

This chapter presents AVA, our Adaptable Vector Architecture that can reconfigure its MVL from short vector lengths (16 elements) to long vector lengths (128 elements). The hardware is designed for short MVL to keep its physical register file modest in size (8KB), so it has the advantage of area efficiency of short vectors. But, it can reconfigure to longer MVLs by mixing different techniques which allow to have an innovative VRF organization.

Today there are two main design trends for vector processors. First vector processors designed for long vectors such as the SX-Aurora TSUBASA [23] which implements vector lengths of 256 elements (16384-bits), and vector processors designed for short vectors such as the Fujitsu A64FX [34] which implements vector lengths of 8 elements (512-bit) ARM SVE. Being the second one the most widely adopted in modern chips. While long vector designs are limited to a specialized subset of applications, where high DLP must be present to achieve excellent performance with a very high efficiency. Short vector designs are compatible with a larger range of applications. In fact, the first long vector length implementations were focused on the HPC market, while short vector length implementations were conceived to improve performance in multimedia tasks. However, those short vector length extensions have evolved, and nowadays are also being exploited on scientific applications, engineering, financial analysis, physics simulations, etc. In that sense, we believe that this compatibility with different applications featuring high, medium and low DLP is one of the main reasons behind the trend of building parallel machines with short vectors. Short vector designs are area efficient and are "compatible" with applications having long vectors; however, these short vector architectures are not efficient as longer vector designs when executing high DLP code.

This chapter is organized as follows. In Section 3.1, the background and motivation is presented. Then, in Section 3.2, presents a detailed description of AVA. Section 3.3 shows the evaluation methodology used. The performance evaluation is shown in Section 3.4. Additionally, area and energy metrics are presented in Section 3.5 and 3.6. Section 3.7 focuses on related work. Finally, Section 3.8 summaries the key points of this chapter.

3.1 Background and Motivation

One conclusion from the study presented in Section 2.4, is that multi-lane vector processors designed for long vectors lengths achieve excellent computational throughput for programs with high DLP. However, applications lacking abundant DLP are unable to fully utilize the hardware resources in the vector lanes. However, when the Application Vector Length is notably smaller than the MVL, multiple inefficiencies arise:

- Short vector applications cannot fully use each vector register width, as a portion of each vector register remains underutilized during the whole program execution.
- When the number of architectural vector registers is not sufficient, the compiler generates spill code (vector load/store instructions) to backup the content of some vector register in memory. With this, it is possible to assign

the architectural vector register to new instructions. At compilation time, the compiler is not aware of the Application Vector Length. In that sense, the spill code includes load/store of vector registers with the MVL value, even though the application only needs a portion of them. This behavior could lead to a performance degradation as well as energy waste.

- Copying vector registers is a common optimization introduced by the compiler when it is needed to save the content of a register before being modified since it can be used in the next iterations. However, the compiler typically copies the whole vector register (VL = MVL), although the effective VL is much less than the vector register width for applications with short vectors. This is because proving that elements past current VL will not be read is difficult for the compiler. In that sense, short vector applications which present this behavior incur in performance overhead as well as wasting energy.

Long vectors bring several advantages such as maximizing the amount of latency amortized per vector instruction. In that sense, different ideas have been studied trying to preserve multi-lane vector processors designed for long vector lengths, while being able to exploit different DLP patterns in an efficient way by reconfiguring the available resources. A couple of the more representative examples for this related work are described below.

Krashinsky et al. proposed the *Vector Thread Architecture* [62], a hybrid multithreaded vector architecture that provides a control processor and an array of slave virtual processors to the programmer. Virtual processors execute strings of vector instructions packaged into blocks termed as *atomic instruction blocks*. To execute high DLP code, the control processor can use vector-fetch commands to broadcast *atomic instruction blocks* to be executed in all virtual processors. For each vector instruction inside the *atomic instruction block,* each virtual processor executes a subset of the vector elements as in the traditional multi-lane designs. On the contrary, to execute thread-parallel code, each virtual processor can use thread-fetches commands to direct its own control flow by fetching its own *atomic instruction blocks* as an efficient way to execute short vectors. However, in order to really exploit short vectors, the application must feature thread level parallelism, otherwise, several virtual processors would remain underutilized. Additionally, to support this hybrid paradigm and create a high-performance vectorized code, the compiler modifications are considerable.

Rivoire et al. proposed *Vector Lane Threading* [14], an architectural enhancement that allows idle vector lanes to run short-vector or scalar threads. When running low DLP

code, they assign the different lanes across several threads. Then, the combination of threads can saturate the available computational resources. In that sense, the microarchitecture allows the exploitation of data-level and thread-level parallelism to achieve higher performance. However, in order to run scalar threads there are required considerable modifications inside each lane to support fetch, decoding, exception handling, etc., making the intra-lane logic complex. Additionally, it is required an instruction cache for each lane.

While the above approaches also feature some reconfigurability, their base vector processor design targets long vectors, which is costly in terms of area and resources. In contrast, AVA is centered around a design targeting short vectors, which is inherently area and resource-efficient. However, AVA reconfigurability enables this short vector design to perform as well as a vector processor designed for long vectors. Additionally, featuring a small VRF offers several advantages, such as the opportunity to implement multi-ported memory structures, a feature that for large memory structures could be costly in terms of area and power, or sometimes prohibitive depending on the design requirements.

MVL reconfigurability has also been proposed at ISA level. For example, the new RISC-V vector extension [10] includes a novel feature called Register Grouping (RG), whose main goal is to increase the execution efficiency for applications featuring high DLP. RG allows grouping multiple vector registers together, so that a single vector instruction can operate on multiple vector registers as if it was a single "wider" register at the cost of having fewer available architectural registers. The Vector Length Multiplier (LMUL) represents the default number of vector registers that are combined to form a vector register group. Specifically, LMUL supports four different configurations (i.e., 1,2,4,8). For those values, the MVL can be increased by 1x, 2x, 4x and 8x while reducing the number of architectural registers from 32 defined by the vector ISA down to 16, 8 and 4, respectively. It is worth noticing that when the application needs more architectural registers than the one available at that time, spill code is generated by the compiler. The bigger the LMUL configuration, the higher the probability of generating spill code. When implementing renaming, physical vector registers are also reduced by LMUL. This implies that for a renaming of 64 physical vector registers for the LMUL=8 configuration there are only 8 register groups available, 4 assigned initially in the Register Alias Table (RAT) and 4 in the Free Register List (FRL). This leads to accepting only four vector instructions before the FRL is empty, and a stall occurs in the scalar core.

Similarly, AVA pursuits the same RG goal, which is to provide the capability to operate on longer vectors when applications exhibit abundant DLP. However, AVA

allows this reconfigurability completely at hardware level, preserving the 32 architectural vector registers defined by the vector ISA, regardless of the MVL configuration. Therefore, AVA can accept as many vector instructions as the number of free registers it has. Larger instruction windows allow exploiting ILP. Additionally, while RG is an exclusive feature of RISC-V, AVA can be implemented over different microarchitectures regardless of the target vector ISA.

Vector architectures have been proposed for embedded systems [63] [64] [16] despite their popular association with high-area. In fact, vector processors are also suitable and even more efficient for power-constrained embedded systems, since vector execution provides energy-efficiency benefits of amortizing instruction supply energy (fetch, decode, and issue) across many operations. Furthermore, even though a larger VRF incurs higher access energy, longer VLs are still beneficial for embedded applications, as established by Gobieski et al. [64]. AVA perfectly matches with embedded systems, since one of the main ideas is to implement a small VRF, while able to continue executing long vectors.

Maximizing the use of expensive hardware resources such as vector registers is an important goal in the computer architecture community, since energy-efficient hardware is required from the HPC market to achieve Exascale levels to the embedded market for ultra-low-power embedded systems.

In this thesis, we tackle this challenge by proposing a novel vector architecture termed as AVA, that combines the area and resource efficiency characterizing short vector processors with the ability to handle large DLP applications, as allowed in long vector architectures.

3.2 Adaptable Vector Architecture (AVA)

AVA is initially designed targeting short vectors lengths (MVL = 16 elements). However, AVA has the ability of reconfiguring the MVL when executing applications with abundant DLP, achieving performance comparable to designs for long vectors. As Figure 3.1 shows, the default configuration of the AVA microarchitecture supports 64 physical vector registers, having a MVL of 16 elements, and thus resulting in a Physical Vector Register File (P-VRF) of 8KB distributed between 8 lanes. However, the adaptability of the proposed architecture allows to scale the MVL ranging from 16 to 128 elements, while keeping the same P-VRF size and the same number of available registers for renaming termed as Virtual Vector Registers (VVRs). To enable this feature, the P-VRF is complemented by a Memory Vector Register File(M-VRF) that does not have direct access to the functional units. Specifically, when the MVL is equal to 16 elements, all the 64 VVRs are held in the P-VRF and none in the M-VRF. Instead, when the MVL is greater

than 16 elements, the VVRs are distributed among the P-VRF and the M-VRF. For example, when the MVL is equal to 128, 8 VVRs are held in the P-VRF, while the remaining 56 are allocated in the M-VRF. The interaction between the P-VRF and the M-VRF is handled by the following components:

1. A two-stage renaming unit, composed of a first stage that maps the 32 logical registers (ISA registers) to the 64 VVRs, and by a second stage, that maps the 64 VVRs to the physical registers located in the P-VRF and/or to the memory registers located in the M-VRF.

2. A two-stage vector issue unit, the first level of which determines which, if any, source VVR of the issuing instruction need to be moved from the M-VRF to the P-VRF. Also, if a new physical register is required, but there is not any free physical register, the content of a selected VVR need to be moved from the P-VRF to the M-VRF, then, freeing up one physical register. We term these operations as swap operations. The second level manages the actual issue of the instruction to the execution units.

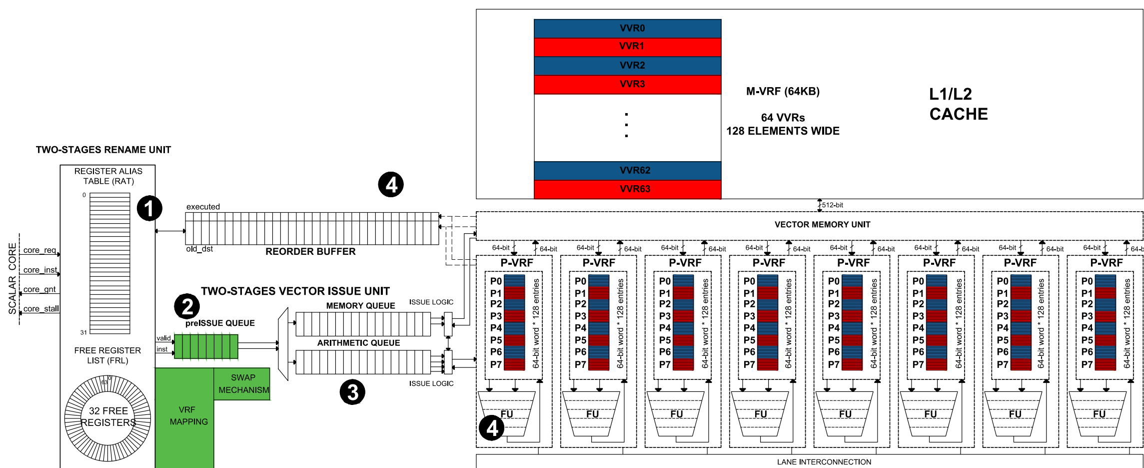

Figure 3.1 *AVA microarchitecture overview. The new hardware additions are highlighted in green, involving the second stage of the renaming unit (VRF-Mapping) and the first stage of the Vector Issue Unit (pre-issue queue and Swap-Mechanism).*

As a general example of how AVA modules interact, Figure 3.1 shows the life cycle of one vector instruction in AVA modules, denoted by steps from 1 to 4.

In ❶ the instruction arrives to the renaming stage, where the instruction operands are renamed from logical registers to VVR. In the next stage, the instruction payload is sent to the pre-issue stage.

At ❷ the pre-issue stage is in charge of mapping the VVRs to physical registers. If the source operands are located in the M-VRF, a Swap-Mechanism moves the related VVRs from the M-VRF to the P-VRF. Additionally, one physical register is assigned in case the vector instruction requires one physical register to write-back the result. In case there are no available physical registers, the Swap-Mechanism selects and copies one VVR located in the P-VRF to the M-VRF, thus freeing a physical register for the instruction.

Once the vector instruction operands have been renamed to physical registers, the vector instruction is sent to the second issue stage ❸, which consists of the arithmetic and memory queues.

❹ Once the instruction is issued and executed, the Reorder Buffer marks it as executed, and waits for its turn to commit.

The next subsections describe the three key components of AVA in more detail (two-stage renaming unit, two-level vector register file and two-stage vector issue unit), followed by a detailed functional description of the overall design.

3.2.1 Two-stage Renaming Unit: Virtual, Physical and Memory Registers

AVA implements a two-stage renaming unit which is based on a new type of registers termed as VVRs, which are an intermediate mapping between the logical registers and the physical and memory registers. Main structures are shown in Figure 3.2.

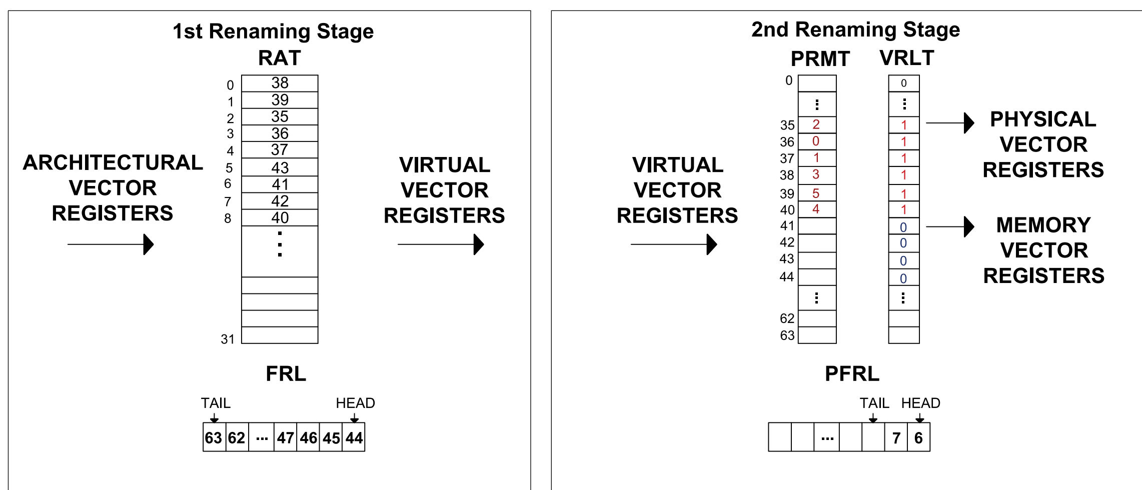

Figure 3.2. *Main structures of the two-stages renaming unit.*

In the first stage, logical registers are renamed to VVRs using the conventional structures: The RAT, a 6-bit x 32-entries structure in charge of keeping the mapping

between the logical registers and the VVRs, and the FRL which contains the available VVRs to be assigned as a virtual destination.

Freeing up Virtual Vector Registers. Freeing up VVRs is performed when an instruction commits. Then, the corresponding old destination VVR is sent to the FRL. Additionally, the corresponding Register Access Counter (RAC) (see Section 3.2.3 for RAC details) is set to 0.

In the second stage, the VRF-Mapping logic keeps track of which VVRs are either mapped to physical or memory registers. This logic is composed of three simple structures. The First structure is the Physical Register Mapping Table (PRMT), a 6-bit x 64-entries structure in charge of keeping the correspondences between the VVRs and the physical registers. The second structure is the Vector Register Location Table (VRLT), a 1-bit x 64-entries structure that indicates if a given VVR is located in the physical or memory registers. Third, the Physical Free Register List (PFRL) is a structure that holds the available physical registers to be assigned.

Freeing up Physical Registers. The freeing up of a physical register occurs in two distinct cases:

1. AVA exploits the concept of aggressive register reclamation [65] to enable physical register usage to closely match the true lifetime of registers. In this sense, it is possible to claim and free a physical register that will not be longer used. The aggressive register reclamation is applied only when: (a) a RAC (see Section 3.2.3 for RAC details) counter reaches zero, meaning that a specific VVR has been renamed, that all the consumers have read the VVR, and that the VVR has become an old destination of a younger instruction, and (b) there are no older vector memory instructions in the pipeline. In this scenario, the corresponding physical register assigned to the VVR which has its count equal to zero can be pushed to the PFRL structure. Note that by updating the RAC counters at commit time we ensure that the freeing up will not create a conflict in case of a recovery event (branch miss prediction or exception in the scalar pipeline) arises. This is because we are ensuring that all the instructions that read that VVR have been committed.
2. When a physical register for the new instruction is needed, but there is no RAC count with 0. In this case, based on the information provided by the RAC counters, the VVR mapped in the P-VRF which has the lowest count, and does not match with any of the instruction virtual source operands is selected. The selected VVR is sent to the M-VRF and free the corresponding physical register.

a)

1st Renaming Stage

RAT

ARCHITECTURAL VECTOR REGISTERS

VIRTUAL VECTOR REGISTERS

VADD V7 V2 V4

VADD V44 V35 V37

oldDst 42

FRL

TAIL HEAD

63 62 ... 47 46 45 44

2nd Renaming Stage

PRMT VRLT

VIRTUAL VECTOR REGISTERS

PHYSICAL VECTOR REGISTERS

MEMORY VECTOR REGISTERS

VADD V44 V35 V37

VADD V6 V2 V1

PFRL

TAIL HEAD

... 7 6

b)

1st Renaming Stage

RAT

ARCHITECTURAL VECTOR REGISTERS

VIRTUAL VECTOR REGISTERS

VADD V7 V2 V4

VADD V44 V35 V37

oldDst 42

FRL

TAIL HEAD

63 62 ... 47 46 45 44

2nd Renaming Stage

PRMT VRLT

VIRTUAL VECTOR REGISTERS

PHYSICAL VECTOR REGISTERS

MEMORY VECTOR REGISTERS

VADD V44 V35 V37

VADD V6 V2 VX

PFRL

TAIL HEAD

... 7 6

c)

1st Renaming Stage

RAT

ARCHITECTURAL VECTOR REGISTERS

VIRTUAL VECTOR REGISTERS

VADD V7 V2 V4

VADD V44 V35 V37

oldDst 42

FRL

TAIL HEAD

63 62 ... 47 46 45 44

2nd Renaming Stage

PRMT VRLT

VIRTUAL VECTOR REGISTERS

PHYSICAL VECTOR REGISTERS

MEMORY VECTOR REGISTERS

VADD V44 V35 V37

VADD VX V2 V1

PFRL

HEAD

EMPTY !

TAIL

Figure 3.3 *Base cases when renaming a vector instruction.*

Figure 3.3 shows three base cases when renaming a vector instruction.

The first example in Figure 3.3.a shows the simplest scenario. The vector instruction *VADD V7 V2 V4* arrives at the renaming unit's first stage, where the architectural vector source operands read the RAT to obtain the corresponding mapping to the source VVRs 35 and 37, and the old destination VVR 42. In parallel, the VVR 44 is obtained from the FRL as the new VVR destination. Then, the renamed instruction corresponds to *VADD V44 V35 V37.* In the next cycle, the previously renamed instruction *VADD V44 V35 V37* arrives at the second stage of the renaming unit, where the VVR source operands read the PRMT and the VRLT to obtain the current mapping of those VVRs. First, by reading the VRLT we know that the VVRs 35 and 37 are located in the P-VRF. Therefore, the values read from the PRMT (physical vector register 2 and 1) represent a valid mapping in the P-VRF. In parallel, the physical vector register 6 is obtained from the PFRL as the new physical destination. With this, the renaming process finalizes successfully.

The second example in Figure 3.3.b shows the scenario when one of the source VVRs is located in the M-VRF. The vector instruction *VADD V7 V2 V4* arrives at the renaming unit's first stage, where the architectural vector source operands read the RAT to obtain the corresponding mapping to the source VVRs 35 and 37, and the old destination VVR 42. In parallel, the VVR 44 is obtained from the FRL as the new VVR destination. Then, the renamed instruction corresponds to *VADD V44 V35 V37.* In the next cycle, the previously renamed instruction *VADD V44 V35 V37* arrives to the second stage of the renaming unit, where the VVR source operands read the PRMT and the VRLT to obtain the current mapping of those VVRs. First, by reading the VRLT we know that the VVRs 35 is located in the P-VRF. Therefore, the value read from the PRMT (physical vector register 2) represents a valid mapping in the P-VRF. However, for the VVR 37 is different since we read “0” from the VRLT, meaning that the VVR 37 is located in the M-VRF. Therefore, the value read from the PRMT (physical vector register X) represents an invalid mapping in the P-VRF, and must not be considered. In parallel, the physical vector register 6 is obtained from the PFRL as the new physical destination. At this moment, the renaming process is not complete since the vector instruction (VADD V6 V2 VX) is not fully renamed to physical vector registers. To finalize the renaming process, the renaming unit is supported by the Swap-Mechanism, which is in charge of solving the problem. The renaming unit then finalizes the renaming phase. Later, Section 3.2.3 describes how the Swap-Mechanism solves the problems, but for the moment, we only know that the problem will be solved.

The second example in Figure 3.3.c shows the scenario when both of the source VVRs are located in the P-VRF, however, there are not a physical vector register

available in the PFRL to assign as a new physical destination. The vector instruction *VADD V7 V2 V4* arrives at the renaming unit's first stage, where the architectural vector source operands read the RAT to obtain the corresponding mapping to the source VVRs 35 and 37, and the old destination VVR 42. In parallel, the VVR 44 is obtained from the FRL as the new VVR destination. Then, the renamed instruction corresponds to *VADD V44 V35 V37.* In the next cycle, the previously renamed instruction *VADD V44 V35 V37* arrives at the second stage of the renaming unit, where the VVR source operands read the PRMT and the VRLT to obtain the current mapping of those VVRs. First, by reading the VRLT we know that the VVRs 35 and 37 are located in the P-VRF. Therefore, the values read from the PRMT (physical vector register 2 and 1) represent a valid mapping in the P-VRF. In parallel, it is required one physical vector register to the PFRL to be used as the new physical destination. However, the PFRL is empty. To finalize the renaming process, the renaming unit notifies this problem to the Swap-Mechanism, which is in charge of solving it. The renaming unit then finalizes the renaming phase.

In addition to the presented cases, it is possible to find a combination of those cases. For example, that both source VVRs are located in the M-VRF, and at the same time, there are no physical registers available in the PFRL. All these problems are notified to the Swap-Mechanism, who will take care of solving each one of them.

Finally, contrary to the RISC-V RG proposal where the number of logical and physical registers is reduced by LMUL factor, our model allows preserving the same number of Logical and VVRs no matter if the MVL increases.

3.2.2 Two-level Vector Register File

The adaptability of AVA allows to reconfigure the MVL from 16 elements up to 128 elements while keeping the same modest P-VRF size. It is achieved by backing the P-VRF with a second level VRF termed as M-VRF. In this scheme, the VVRs that are being used or will soon be used are assigned to the first level (i.e., P-VRF) allowing them to have direct access to the functional units. On the other hand, the VVRs that are not being used or will not be used soon are assigned to the second level (i.e., M-VRF). Additionally, each VVR is associated with one entry of the valid-bit structure (1-bit x 64-entries) which indicates a valid data. When a VVR is assigned at renaming time, the associated Valid-bit is set to 0. Once the vector instruction executes, the associated Valid-bit is set to 1.

Since our baseline microarchitecture features an 8-Lane VPU, the P-VRF is implemented as eight 4R-2W 1-KB (64-bit words x 128 entries) SRAM memory structures distributed between the eight lanes. The P-VRF contains 64 physical registers where each register is 16 elements wide for the baseline configuration, as illustrated in Figure 3.1. Note that our model is restricted to execute one arithmetic operation plus one

memory operation in parallel. Accordingly, three read ports and one write port are assigned to the arithmetic pipeline, while one read port and one write port are assigned to the memory unit. Adding more arithmetic pipelines would increase the required VRF ports, which has a super-linear impact on the power/area results, as demonstrated by Arima et al. [53] and Zyuban et al. [54].

Table 24. *Physical Vector Register File Configurations.*

Physical Regs	64	32	21	16	12	10	9	8
MVL	16	32	48	64	80	96	112	128

Furthermore, by setting a configuration register, it is possible to configure the VPU for longer MVLs at the cost of reducing the number of physical registers that can be held in the P-VRF. For example, we can configure from 64 physical vector registers (16 elements each), down to 8 physical vector registers (128 elements each) in multiples of 16 elements as summarized in Table 24. Note that supporting all the proposed configurations does not incur in additional routing overhead. Indeed, the read/write control logic iterates MVL/lanes times until it completes the read/write operation.

When MVL is higher than 16 elements, we need to reserve memory to hold the M-VRF. In our experiments, in each *c++* application, we include a function called *set_mvrf* which performs three main tasks: (1) configures the MVL that best matches the application characteristics, (2) performs a *malloc* assignment for reserving memory to allocate the M-VRF, and (3) sends the base address of the M-VRF to the VPU. However, ideally, the OS takes care of reserving the memory space for each thread.

```
int main (int argc, char **argv) {
   FILE *file;
   file = fopen(inputFile, "r");    //Read input data from file
   rv = fscanf(file, "%i", &numOpt);
   data = (float *)malloc(numOpt*sizeof(float));
   prices = (float*)malloc(numOpt*sizeof(float));
   for ( loopnum = 0; loopnum < numOpt; ++ loopnum ) {
      rv = fscanf(file, "%f", &data[loopnum]);
   }
   rv = fclose(file);

#ifdef USE_RVA
  set_mvrf (numOpt,_epi_e32);
#endif // USE_RVA

unsigned long int gvl;
for (i=0; i<numOpt; i += gvl) {
  gvl=__builtin_epi_vsetvl(numOpt-i,__epi_e32, __epi_m1);
  BlkSchlsEqEuroNoDiv_vector (&(prices[i]), gvl ... );
} ...
```

```
void set_mvrf (int numOpt, int sew) {
   unsigned long int gmvl;
   unsigned int* virtual_vrf;

   gmvl =_builtin_epi_vsetmvl(numOpt,_sew);
   virtual_vrf = (int*)malloc(gmvl*sizeof(int) * 64);
   send_virtual_vrf_base_addr(virtual_vrf[0]);
}
```

Input file

```
65,536
42.00   0.20  0.50 C 0.00    4.759423036851750055
42.00   0.20  0.50 P 0.00    0.808600016880314021
100.00  0.15  1.00 P 0.00    3.714602051381290071
100.00  0.15  1.00 C 0.00    8.591659601309890704
60.00   0.30  0.25 C 0.00    2.133371966735750025
100.00  0.10  0.10 C 0.00    10.895610714793999563
100.00  0.10  0.50 C 0.00    14.421570828843300660
100.00  0.10  1.00 C 0.00    18.630859120667498274
...
```

Figure 3.4 *M-VRF definition. Gray box shows the main function, yellow box shows the M-VRF definition, and blue box shows the input dataset.*

To exemplify how the function *set_mvrf* works, Figure 3.4 shows a reduced version of the Blackscholes application. In the gray box the Blackscholes's main function is displayed. The first lines show the opening of a text file that contains the input dataset to be computed (the input dataset is presented in the blue box). As mentioned previously, different datasets are provided for each application. Then, the number of European options to be computed is read from the input file. For the large input dataset, this number corresponds to 65,536 European options *(numOpt)*. Then, the 65,536 European options are copied to the *data* structure.

After the initialization phase is finished and thus the size of the input dataset is known, we configure AVA through the *set_mvrf* function. The yellow box shows the implementation of the three main tasks in the *set_mvrf* function.

1. First, a new configuration intrinsic termed as *_builtin_epi_vsetmvl* is defined, which requests and sets the MVL with value *numOpt* (65,536). The larger MVL supported by AVA for a 32-bit element width is 256 elements. Then, the intrinsic *_builtin_epi_vsetmvl* returns the granted MVL (*gmvl*) of 256 elements. With this, the VPU is configured with 64 VVRs, each having 256 elements. However, since the P-VRF is only 8KB, there is space to hold only 8 VVRs.
2. After setting the MVL, the next step is to define the M-VRF in memory. This is done by performing a *malloc* memory assignment of size *gmvl x sizeof(int) x 64-VVRs*, resulting in 64KB corresponding to the larger M-VRF supported for AVA. There are other applications such as Streamcluster that, as an input, receive streams of 128 elements, each 32-bits. Then, when configuring the MVL, this will be set to 128 elements, each 32-bits, and when reserving memory for the M-VRF will lead to only 32KB. Then, the size of the M-VRF depends on the MVL previously set.
3. Finally, the intrinsic *send_virtual_vrf_base_addr* sends the base address of the M-VRF to the VPU in order to know the location in memory of each VVR.

Finally, coming back to the gray box, the vectorized functions can be executed iterating over the granted vector length (*gvl*) value until completing *numOpt* operations. For this example, the *gvl is always equal to the* MVL. This means that vectors of 256 elements are computed inside the BlkSchlsEqEuroNoDiv_vector function in every iteration.

3.2.3 Two-stages Vector Issue Unit

The two-stages vector issue unit is composed of the pre-issue stage and the issue stage. We now explain both stages in turn.

***Pre-issue stage*:** The first level of mapping from Logical Registers to VVF occurs in the renaming stage. Pre-issue stage performs the second level of mapping between the VVRs and physical registers.

As mentioned before, when MVL>16, a subset of the VVRs is held in the P-VRF, while the remaining VVRs are allocated in the M-VRF. In case a new physical register is required, but there is not any free physical register, the content of a selected VVR is sent to the M-VRF to free one physical register, which is assigned to the new instruction. Eventually, VVRs previously moved to the M-VRF can be needed by a new vector instruction, which then requires to move the content back to the P-VRF. In consequence, AVA implements a Swap-Mechanism module which is composed of two main structures: the RAC and the Swap-Logic. We now describe the operation of both.

1. The RAC is a 3-bit x 64-entry structure where each entry holds how many times a specific VVR is read. At the first stage of the renaming, the RAC counters are updated. First, the new destination and source VVRs increment the corresponding register count, while the old destination VVR decrements the corresponding count. At commit time, the counters are updated again. This time, the source VVRs decrement the corresponding counter. The RAC helps to take decisions based on the count of each individual VVR which are described in the next paragraphs.
2. The Swap-Logic decides which VVRs should be swapped to the M-VRF, and creates memory operations termed as Swap-Stores. Swap-Logic also decides when it is required to move VVRs from the M-VRF to the P-VRF, and creates operations termed as Swap-Loads. The Swap-Logic takes advantage of the information provided by the RAC counters to decide which VVR allocated in the P-VRF should be swapped to the M-VRF. The VVR mapped in the P-VRF which has the lowest count (1 is the lowest count for swaps, and 0 is the count for aggressive register reclamation) is selected for the swap, and selection logic also checks that the candidate VVR does not match with any of the instruction's virtual source operands to avoid deadlock.

Pre-issue stage implements an in-order issue scheme. A vector instruction is ready to be issued to the second level only when it has been fully renamed from VVRs to

physical registers. However, renaming the instruction from VVRs to physical registers involves several steps evaluated in the following order:

1. Source VVRs are mapped to the corresponding physical register by reading the PRMT and the VRLT structures indexed by the source VVRs. There are two possible scenarios for each source operand:
    a. If the value read from the VRLT is equal to "1", it indicates that the physical register obtained from the PRMT structure is valid and it is located in the P-VRF, and the corresponding source VVR can be mapped immediately.
    b. On the contrary, if the value read from the VRLT is equal to "0", the VVR is located in the M-VRF and is loaded to the P-VRF to be used. In this second scenario, it is required to notify the conflict to the Swap-Mechanism. A couple of tasks are performed by the Swap-Mechanism: First the swap mechanism verifies that there is at least one physical register available to load the values from the M-VRF. In case there are not free physical registers, a Swap-Store is created and sent to the memory queue to store the content of one VVR selected by the Swap-Logic from the P-VRF to the M-VRF. With this, the associated physical register can be freed and pushed to the PFRL. Following the above verification, the Swap-Mechanism creates a Swap-Load. This Swap-Load is sent to the memory queue to load the VVR from the M-VRF to the P-VRF.
2. If the vector instruction requires to write-back its result, a new physical register must be assigned. In case there are no free physical registers, the verification step must be repeated. Then, the new available physical register can be assigned as the physical destination.
3. Finally, once the instruction has been renamed, it is issued to the second level only if there is availability in their corresponding queue. Otherwise, a stall is signaled until there is at least one free slot for the instruction.

Issue stage is composed of the memory and arithmetic queues in charge of issuing the vector instruction. Individually each queue performs in-order issue. However, since the memory queue is decoupled from the arithmetic queue, there is a light out-of-order behavior.

The introduction of swap operations might lead to deadlock. To avoid deadlocks, AVA must guarantee that each issuing instruction in the arithmetic or memory queues must have its source VVRs mapped to the P-VRF. This is done by following two rules:

1. Swap-Stores created to free one physical register must notify to the new owner of the physical register that it has executed, meaning that the VVR previously mapped in the physical register is now in the M-VRF. Only then and then it is possible to write-back new data to the physical register.

2. Swap-Loads must wait until all the consumers of the previously VVR mapped in P-VRF have read the register. Since each queue operates in-order, issue stage only needs to check the consumers inside the arithmetic queue.

Once the instruction is issued and executed, it will be marked in the reorder-buffer as executed, only waiting for its turn to commit.

3.2.4 Recovering the microarchitectural state

After some event such as a misprediction or memory exception, AVA can roll back and recover the correct microarchitectural state. The renaming tables (RAT and FRL pointers) and the Valid-bit are the only mandatory structures to be recovered. Therefore, AVA implements only one copy that is updated every time a vector instruction commits.

Recovering the RAC counters is optional, since every time that a VVR is freed, the respective count is also set to zero. Thus, not recovering the state of the counters does not imply any correctness issue.

3.2.5 AVA Functional Description

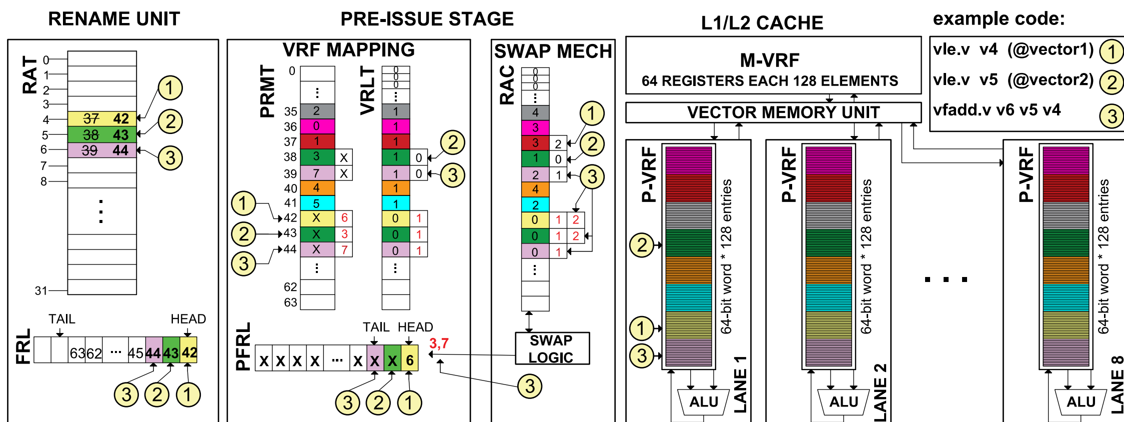

Figure 3.5 *Register Mapping example*

Figure 3.5 illustrates the AVA functional behavior based on the execution of three instructions. The selected MVL configuration is 128 elements, meaning that only 8 physical registers are available. Also, to demonstrate how the Swap-Mechanism works, we assume that several vector instructions were executed previously, meaning that some physical registers were previously assigned to older instructions.

Once the scalar core sends the first instruction to the decoupled VPU, it is received by the first stage of the renaming unit. In this stage, the logical registers are renamed to VVRs. Since the instruction is a vector load, only the destination logical register 4 reads the RAT to obtain the associated old destination VVR 37. The destination VVR 42 is obtained from the FRL. In parallel, the new destination VVR 42 increments the corresponding RAC entry, while the old destination VVR 37 decrements the corresponding RAC entry. In the next cycle, the instruction advances to the pre-issue stage. Since it is a load, the only requirement is to obtain a physical register to be used as a destination. At this moment, the PFRL has the physical register 6 available, which is assigned as the physical destination. Then, this new mapping is written in the location 42 of the PRMT. Additionally, the corresponding entry in the VRLT is set to 1, indicating that the VVR 42 is now mapped in the physical registers. After this, the instruction is sent to the memory queue in the second stage to wait for execution.

As also the second instruction is a vector load, performing the same process in the renaming unit as the previous load, the VVR 43 is assigned as the virtual destination. In parallel, the new destination VVR 43 increments the corresponding RAC entry, while the old destination VVR 38 decrements the corresponding RAC entry. Note that after the decrement, the count reaches “0”, meaning that it is possible to reclaim physical register 38 to be used for a new physical destination, since it is guaranteed that any younger instruction will never use the data stored in the VVR 38. Additionally, location 38 in the VRLT is set to “0”, indicating that the VVR 38 is no longer mapped in the physical registers. In the following cycle, the instruction advances to the pre-issue stage, where the PFRL points to physical register 3 as being available, which is assigned as a physical destination for the load. Then, the instruction is sent to the memory queue in second stage waiting to be executed.

The last instruction corresponds to a vector addition. This time, the sources and destination logical registers read the RAT to obtain the associated source VVRs 42 and 43, and old destination VVR 39 respectively. The VVR 44 is assigned as the destination. In parallel, the source VVRs 42 and 43, and the new destination VVR 44 increments the corresponding RAC entry, while the old destination 39 decrements the corresponding RAC entry. After the decrement, the count reaches “1”, which means that this time it is

not possible to reclaim physical register 39. In the next cycle, the instruction advances to the pre-issue stage. In the pre-issue stage, the PFRL does not have any physical register available, and there are no VVR counters mapped in the physical registers that have reached "0". This forces a swap operation. To do the swap, the RAC entry with the lowest count is selected, which corresponds to VVR 39, which will be sent to the memory registers. Subsequently, a Swap-Store operation is created and issued to the memory queue to send the content of VVR 39 to the memory registers. Finally, the physical register 7 is freed and pushed to the PFRL, to be assigned later as a physical destination for the vector addition. Then, the instruction is sent to the arithmetic queue in the second stage where it waits for being issued to execution. Once every instruction commits, all source VVRs will decrease the associated RAC entry by one.

3.3 Evaluation Methodology

To evaluate AVA, we use as a base platform the previously presented parameterizable decoupled vector architecture [18] based on the RISC-V Vector extension [37] [38] modeled on the gem5 simulator presented in Chapter 2. We added the necessary modifications to implement the AVA architecture, substantially modifying the issue stage which also includes the queues, the Swap-Mechanism, and the VRF read/write logic.

Table 25*. Vector Processing Unit Configurations*

<table>
<tr><th>BASELINE-X1</th><th>BASELINE -X2</th><th>BASELINE -X3</th><th>BASELINE -X4</th><th>BASELINE -X8</th></tr>
<tr><td colspan="5">Dual-Issue 64-bit RISC-V superscalar in-order pipeline
Clock Frequency - 2 GHz</td></tr>
<tr><td colspan="5">Vector Processing Unit (VPU)
8 Lanes (1 pipelined arithmetic unit / Lane)
Clock Frequency - 1 GHz</td></tr>
<tr><td>MVL 16 elements
(1024-bit)</td><td>MVL 32 elements
(2048-bit)</td><td>MVL 48 elements
(3072-bit)</td><td>MVL 64 elements
(4096-bit)</td><td>MVL 128 elements
(8192-bit)</td></tr>
<tr><td colspan="5">64 Physical Registers</td></tr>
<tr><td>4R/2W VRF:
8KB</td><td>4R/2W VRF:
16KB</td><td>4R/2W VRF:
24KB</td><td>4R/2W VRF:
32KB</td><td>4R/2W VRF:
64KB</td></tr>
<tr><td colspan="5">Vector Memory Queue - 32 entries - in-order Issue
Vector Arithmetic Queue - 32 entries - in-order Issue</td></tr>
<tr><td colspan="5">Ring topology for Lane Interconnection</td></tr>
<tr><td colspan="5">VMU connected to L2 Bus, 512-bit memory interface</td></tr>
<tr><td colspan="5">Memory System</td></tr>
<tr><td colspan="5">32KB L1I – hit latency 4 clock cycles – cache line 512-bit – 4 MSHRs
32KB L1D – hit latency 4 clock cycles – cache line 512-bit – 4 MSHRs
1MB L2 – hit latency 12 clock cycles – cache line 512-bit – 32 MSHRs
2 GB DDR3-1600 Memory - DRAM memory access latency per DRAM burs 46.25ns</td></tr>
</table>

Table 26. *AVA and RG configurations and their corresponding equivalence with the five configurations in Table 25.*

BASELINE -X1	BASELINE -X2	BASELINE -X3	BASELINE -X4	BASELINE -X8
AVA-X1 (64-PREG)	AVA-X2 (32-PREG)	AVA-X3 (21-PREG)	AVA-X4 (16-PREG)	AVA-X8 (8-PREG)
RG-LMUL1	RG-LMUL2	NA	RG-LMUL4	RG-LMUL8

Table 25 presents five system configurations where a VPU is attached to a scalar core. The VPU configurations vary the MVL's. BASELINE-X configurations denote a vector architecture designed for a specific MVL and is the baseline to compare against. BASELINE-X1 corresponds to the baseline hardware with 64 physical registers with MVL=16 elements (1024-bits), leading to a VRF size of 8KB. The remaining configurations (from BASELINE-X2 to BASELINE-X8) represent a costly hardware implementation, increasing the MVL size in every configuration leading to VRF sizes from 16-KB up to 64-KB for the largest configuration.

Table 26 shows five different AVA configurations. AVA-X1 represents the baseline model (64 physical registers with MVL=16 elements). AVA-X2 to AVA-X8 represents the AVA configurations that after reconfiguring AVA-X1 match the BASELINE-X2 to BASELINE-X8 configurations. Also, the number of physical registers available for each configuration is shown. In the same way, the equivalent configurations for RISC-V RG are listed (LMUL1, LMUL2, LMUL4, and LMUL8). It is important to emphasize that both AVA and RG proposals use the baseline configuration with an 8-KB VRF for all their configurations.

Table 27. *Applications from RiVEC Benchmark Suite*

Application	Application Domain	Algorithmic Model	Architectural vector registers required by the compiler
Axpy	HPC	BLAS	2
Blackscholes	Financial Analysis	Dense Linear Algebra	23
LavaMD2	Molecular Dynamics	N-Body	15
Particle-Filter	Medical Imaging	Structured Grids	13
Somier	Physics Simulation	Dense Linear Algebra	13
Swaptions	Financial Analysis	MapReduce	24

Table 27 shows six applications from the RiVEC Benchmark Suite [37] [40] presented in Chapter 2. There are two main considerations for selecting only six applications:

1. First, the applications must be compatible with longer vector lengths. Applications such as Canneal which are compatible only with short vector lengths would not add meaningful data to this study. Then, it was selected applications compatible with medium and long vector lengths.

2. When there are no swap operations, it is expected to have exactly the same performance as the BASELINE configurations. However, when swap operations appear, different behaviors can arise, and it is interesting to analyze them. Therefore, to expose this extra memory traffic, applications that impose high pressure in the use of vector architectural registers to the compiler are the best candidates. Table 27 shows the number of architectural registers required by the compiler for each application.

For all the applications, we have compiled four versions. The first version is compiled using the flag for LMUL=1. The resulting binary is used to evaluate the baseline configuration (MVL=16), and all the AVA and BASELINE configurations. The following ones use the flags to compile for the LMUL2, LMUL4, and LMUL8 configurations respectively to evaluate RG.

To obtain area and energy, we use the extended McPAT framework configured for the 22nm technology node. We model AVA, RG, and the five BASELINE configurations presented in Table 25. Finally, AVA was successfully implemented at RTL level on an in-house VPU. More details are shown in Section 3.6.

3.4 Performance, Energy, and Area Evaluation

To demonstrate that AVA improves performance, the execution time and speedup is presented for all the applications and all the VPU configurations including AVA, RG, and the five BASELINE configurations.

To demonstrate that AVA is energy-efficient, the energy consumption for all the evaluated applications is obtained from the McPAT framework configured for 22nm technology. The application statistics introduced in the McPAT model corresponds to the gem5 outputs. Dynamic and leakage energy results are reported only for the main contributors: The L2 cache, the VRF, and the FPUs. The required AVA structures also are modeled; however, it represents only 0.4% of the overall VPU energy consumption for the AVA-X1 configuration. Since the number of issued instructions is reduced as the MVL value is increased, the energy consumed by the required AVA structures is also reduced for larger MVL configurations. We include the extra energy dissipation of AVA in the VRF Dynamic/Leakage bars for all the AVA configurations.

To demonstrate that AVA is area-efficient, Figure 3.6 presents the area of AVA and the five BASELINE configurations from the McPAT framework configured for 22nm technology. Note that for all the configurations, a total of 8 lanes are set. Each lane is equipped with 1 FPU and a slice of the overall VRF. We pay special attention to the VRF. The VRF configuration for AVA and the five BASELINE configurations are listed below:

- AVA (VRF-8KB): eight 4R/2W 64-bit*128-entries memory macros.
- BASELINE-X1 (VRF-8KB): eight 4R/2W 64-bit*128-entries memory macros.
- BASELINE-X2 (VRF-16KB): sixteen 4R/2W 64-bit*128-entries memory macros.
- BASELINE-X3 (VRF-16KB): twenty-four 4R/2W 64-bit*128-entries memory macros.
- BASELINE-X4 (VRF-32KB): thirty-two 4R/2W 64-bit*128-entries memory macros.
- BASELINE-X8 (VRF-64KB): sixty-four 4R/2W 64-bit*128-entries memory macros.

Note that all of these configurations implement the same size of memory macros but duplicate the number in each configuration. The area estimate also includes the scalar core, the 32KB L1-I and L1-D caches, and the 1MB L2 cache.

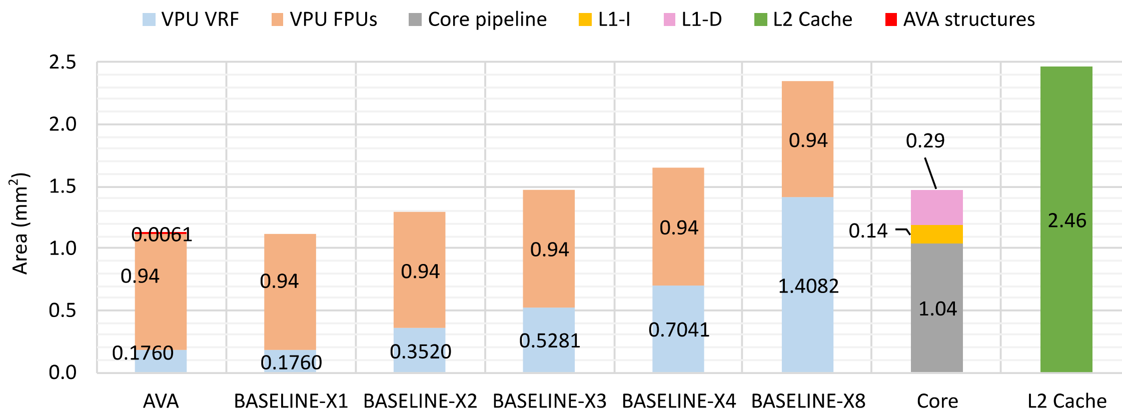

Figure 3.6 *Area results for AVA and the different BASELINE configurations obtained from McPAT*

Note that BASELINE-X8 is almost the same size as the 1MB L2 cache. This is because the multi-ported 64KB VRF represents 59% of the overall VPU area for the BASELINE-X8 configuration. Also, the VRF occupies almost the same area as the scalar core, including L1 caches. When comparing BASELINE-X8 with AVA-X1 and BASELINE-X1, the 64KB VRF is 8x bigger than the smallest configurations. For the AVA configuration, the required AVA structures add a negligible 0.55% area overhead to the VPU, while reducing the total VPU area by 53% compared with the BASELINE-X8 configuration.

To demonstrate the AVA performance/mm^2 efficiency, the performance obtained for each BASELINE and AVA configuration is divided between the area of their

corresponding configuration. Note that for AVA, the area is 1.122mm^2 for all the configurations.

3.4.1 Axpy

Performance Evaluation. The first application is Axpy, which represents the ideal scenario for both RG and AVA where RG-LMUL8 and AVA-X8 obtain the same performance compared to a VPU designed for long vectors (BASELINE-X8), and achieving a speedup of 2x with respect to the baseline configuration (BASELINE-X1), as illustrated in Figure 3.7.c. Also, as shown in Figure 3.7.a, neither spill code from the compiler nor swap operations from the Swap-Logic are created since Axpy only uses two logical vector registers. Figure 3.7.b shows the percentage of vector arithmetic and memory instructions. For all the configurations, the vector memory instructions (Vmemory) represent 75%, and the vector arithmetic instructions (Varithmetic) represent 25% of the total vector instructions.

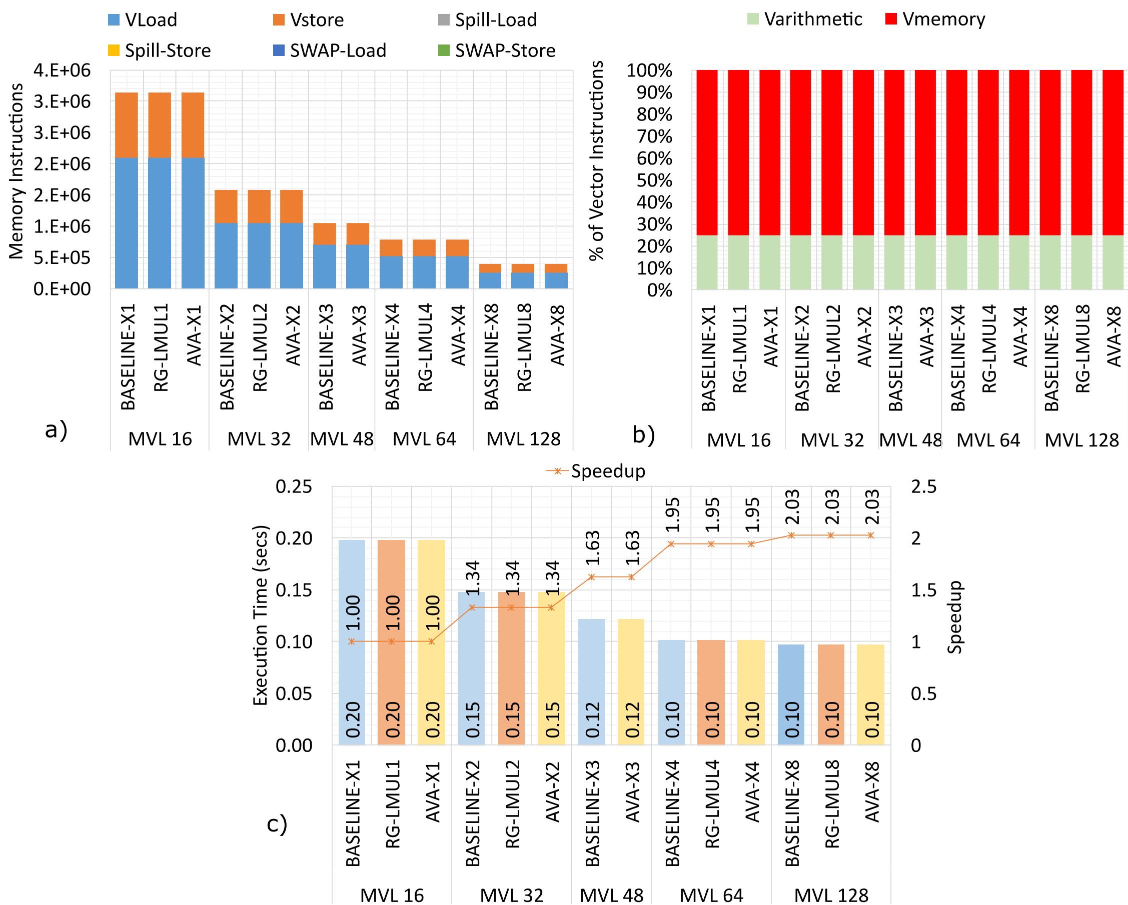

Figure 3.7 *Axpy performance evaluation. a) Vector Memory Instruction count including spill operations generated by the compiler and swap operations generated by AVA. b) % of vector instruction, c) Execution-time and speedup compared to BASELINE-X1.*

Energy Evaluation. Figure 3.8 shows energy consumption (left axis) and Energy Delay Product (EDP) efficiency metric (right axis) for Axpy. Axpy does not exhibit spill/swap operations. As the MVL is increased, less total energy is consumed. For the VPU, the Dynamic energy is constant since no spill/swap operations are added, while for the scalar core, dynamic energy is reduced since fewer instructions are fetched, decoded, and executed in the scalar pipeline. Since larger configurations perform faster, leakage energy is reduced. Note that BASELINE-X2, BASELINE-X3, BASELINE-X4, and BASELINE-X8 configurations double the leakage in each configuration because they implement larger multi-ported VRF memories from 16KB up to 64KB. Then, both RG and AVA configurations consume less energy than the equivalent BASELINE configuration. When compared with the BASELINE-X1 configuration, AVA saves 46% of the overall energy consumption by reconfiguring for long vectors. Finally, EDP results show that the most efficient configurations are AVA-X8 and RG-LMUL8.

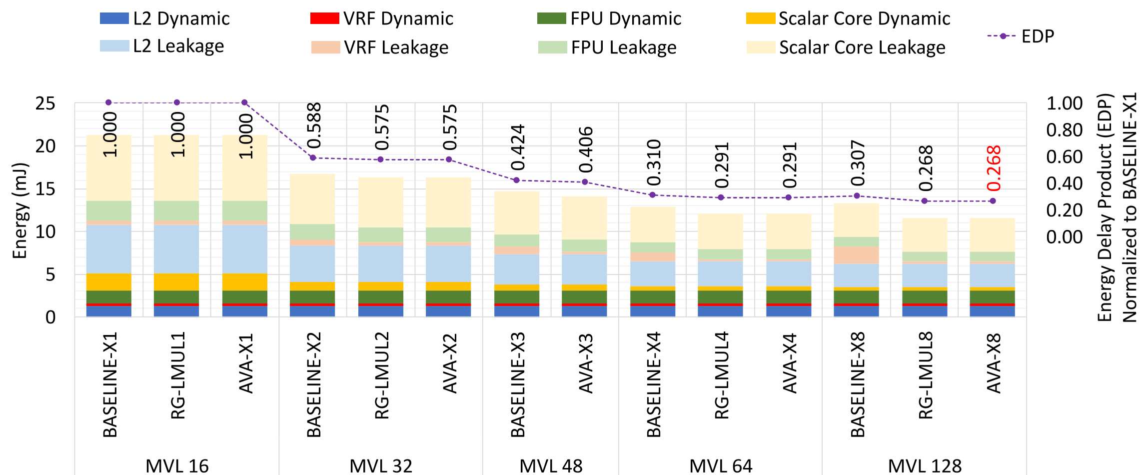

Figure 3.8 *Axpy energy consumption (left axis) and normalized Energy Delay Product (right axis) for different configurations.*

Performance/mm². Figure 3.9 shows the performance/mm² efficiency achieved for AVA when comparing versus the BASELINE and LMUL configurations. When reconfiguring AVA for longer MVL configurations, it is clear that every square millimeter is better exploited. For example, reconfiguring for AVA-X8, the performance/mm² efficiency doubles AVA-X1.

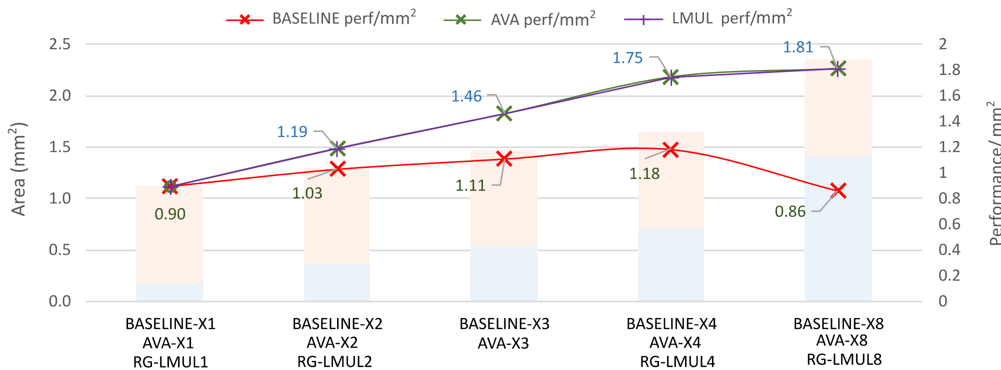

Figure 3.9 *Area results obtained from McPAT for 22nm technology node, and performance/mm^2 for each configuration.*

3.4.2 Blackscholes

Performance Evaluation. The second application is Blackscholes. This high DLP application is interesting to analyze since the vector compiler requires 23 architectural vector registers to obtain the final binary. At first glance, we can see that there is high pressure in the use of vector logical registers. For LMUL=2,4 and 8, the compiler can make use of only 16,8 and 4 architectural vector registers respectively, and for any of those configurations, spill code is added as shown in Figure 3.10.a. AVA presents a similar behavior. However, it is interesting to see that for AVA-X2 there are no swap operations. This is because the scheduling is done using 32 physical vector registers, meaning that we have enough vector registers to compute the application without generating swap operations. On the other hand, swap operations are generated starting from AVA-X4. Also, the number of swap operations is slightly less than the number of spill code operations generated by the compiler. This is because AVA performs the scheduling based on the available physical registers, which are always double compared to LMUL. Figure 3.10.c shows the performance results. For AVA-X2 there are no swap operations, thus a similar performance to BASELINE-X2 is achieved, and a speedup of 1.58x over the baseline configuration. For AVA-X3, the percentage of memory operations represents 11.9% of the total vector instructions. AVA-X3 achieves a speedup of 1.72x over the baseline configuration, and slightly lower than the equivalent BASELINE-X3 configuration. This implies that almost all the extra memory traffic generated by the swap operations can be hidden beneath vector arithmetic execution. Also, the swap operations that were not able to be hidden beneath vector arithmetic execution, cause a stall in the vector arithmetic pipeline because of a read after write dependency, decreasing the performance by only 3.3% compared with the equivalent BASELINE-X3 configuration. For AVA-X8, the percentage of memory operations represents 38% of the total vector instructions, causing that a larger percentage of swap operations cannot be hidden beneath vector arithmetic execution, leading to a performance degradation of 30.8%

compared with the equivalent BASELINE-X8 configuration. For all the configurations, AVA performs better than RG since less memory traffic is generated.

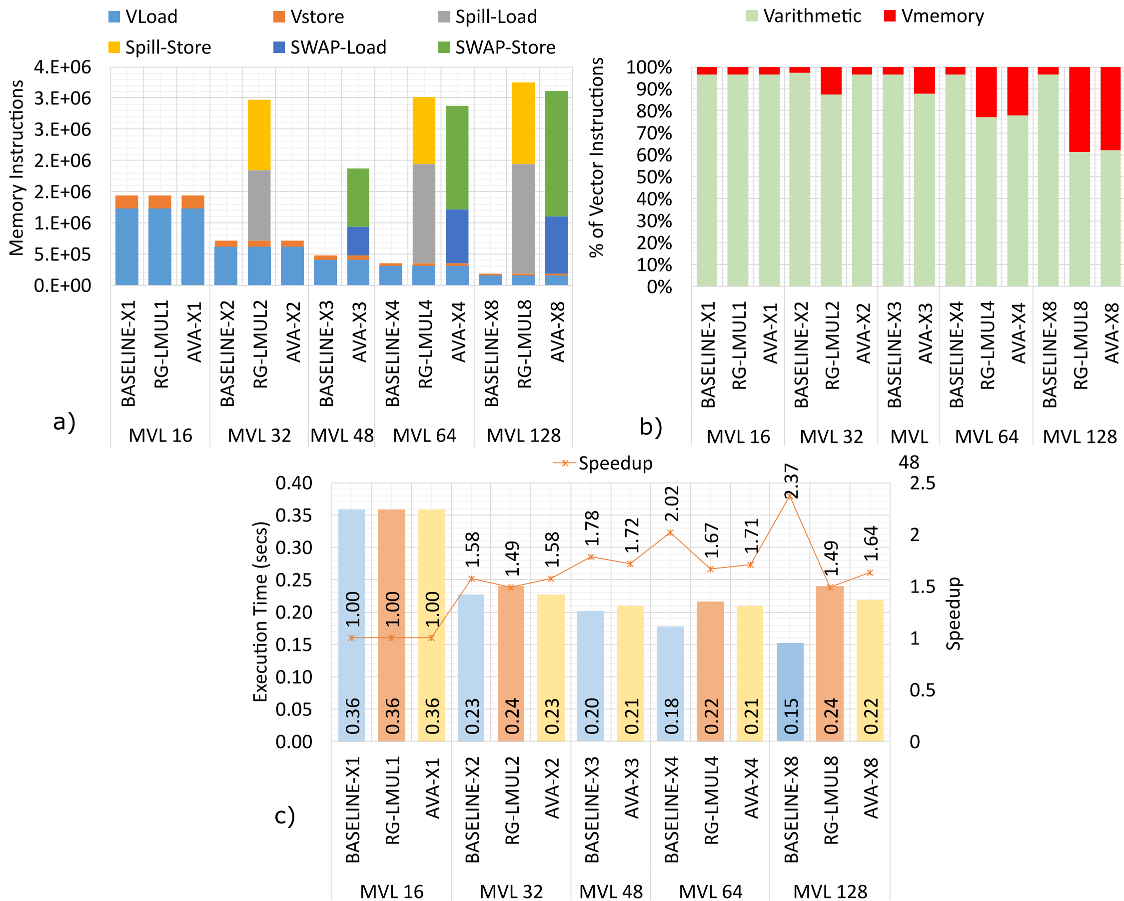

Figure 3.10 *Blackscholes performance evaluation. a) Vector Memory Instruction count including spill operations generated by the compiler and swap operations generated by AVA. b) % of vector instruction, c) Execution-time and speedup compared to BASELINE-X1.*

Energy Evaluation. Figure 3.11 shows energy consumption (left axis) and Energy Delay Product (EDP) efficiency metric (right axis) for Blackscholes. Blackscholes generates an important number of spill/swap operations for the RG-LMUL8 and AVA-X8, leading to extra energy dissipation which is wasted to support those operations, as shown by the L2 Dynamic value. However, Blackscholes features a very high percentage of vectorization, where the overall instruction count is reduced from 3,180,479,876 for the scalar version to 8,704,913 for the vectorized version with MVL=128. Then, even when there is an increase in the L2 Dynamic energy caused by the swap operations, reconfiguring for longer vector lengths leads to energy savings for AVA when compared

with BASELINE-X1. Finally, looking at the EDP results, the most efficient configurations for AVA are represented by AVA-X3 and AVA-X4, which are very close.

Performance/mm². Figure 3.12 shows the performance/mm² achieved for AVA when comparing versus the BASELINE and LMUL configurations. When reconfiguring AVA for longer MVL configurations, AVA-X3 and AVA-X4 achieve the best performance/mm² efficiency. This is because the swap operations hurt the final performance when reconfiguring for AVA-X8.

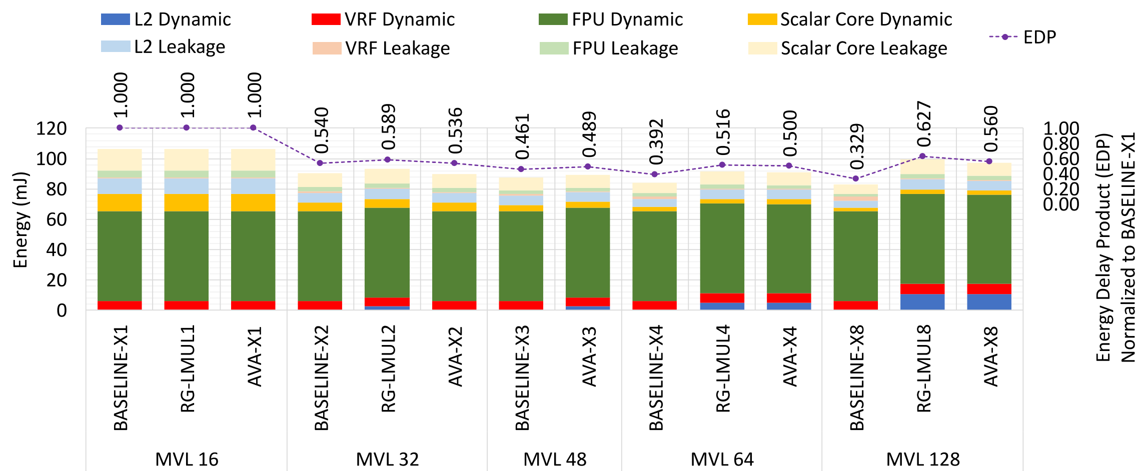

Figure 3.11 *Blackscholes energy consumption (left axis) and normalized Energy Delay Product (right axis) for different configurations.*

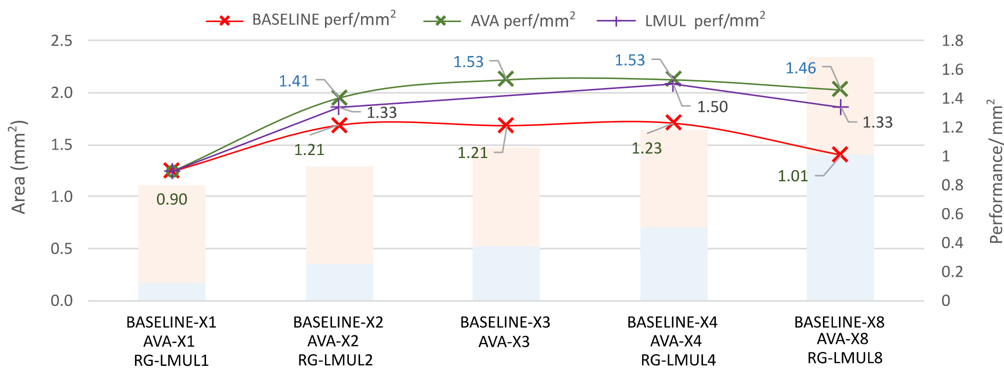

Figure 3.12 *Area results obtained from McPAT for 22nm technology node, and average performance/mm² for each configuration.*

3.4.3 LavaMD2

Performance Evaluation. For LavaMD2, the vector compiler uses 15 architectural vector registers to create the final binary, which implies that for RG-LMUL2, no spill code is necessary. However, for RG-LMUL4 and RG-LMUL8, spill code is generated, as shown in Figure 3.13.a, causing an increase in memory operations from 9% for RG-LMUL1 configuration to up to 43% for RG-LMUL8 configuration, as shown in Figure 3.13.b. For AVA, the swap operations are few compared with the equivalent spill code generated by the RG-LMUL configuration. Figure 3.13.c shows the performance results. First, this application makes use of a fixed vector size of 48 elements, meaning that for the configurations with a larger MVL than 48 elements, we cannot make full use of each vector register, and a portion of each vector register remains unused during all the program execution. For AVA the best configuration is AVA-X3. AVA-X3 not only executes the 48 elements with only one instruction, but also 21 physical registers are available for the computation, thereby avoiding swap operations, as shown in Figure 3.13.b. Also, it achieves a speedup of 1.67x, better than any of the RG- LMUL configurations and equal to the equivalent BASELINE configuration. Finally, another interesting result is for RG-LMUL8, where the performance decreases notably. The reason is because for this configuration, the memory operations represent 43% of the overall vector instructions. Also, 81% of the memory operations are spill code. As described in Section 3.1, the spill code is always executed using the MVL. As a result, the memory operations become the bottleneck since all the arithmetic operations (57%) are executed with VL=48, while spill code is executed with VL=MVL=128.

Energy Evaluation. Figure 3.14 shows energy consumption (left axis) and Energy Delay Product (EDP) efficiency metric (right axis) for LavaMD2. LavaMD2 has interesting results. Energy consumption increases notably for RG-LMUL8 and AVA-X8. This is because LavaMD2 features medium-size vectors, with MVL=48 providing the optimal energy. Spill/swap operations are always executed with the MVL value. For RG-LMUL8 and AVA-X8 configurations, spill/swap operations are executed with an MVL=128, although elements past VL=48 are not used, leading to a drastic energy consumption increase. However, when running the application, AVA will select the optimal configuration (AVA-X3), avoiding wasting unnecessary energy.

Performance/mm^2. Figure 3.15 shows the performance/mm^2 achieved for AVA when comparing versus the BASELINE configurations. When reconfiguring AVA for longer MVL configurations, AVA-X3 achieves the best performance/mm^2 efficiency. On the contrary, when reconfiguring for AVA-X8, there is an efficiency loss. This is because LavaMD2 requires at most an MVL=48; larger configurations do not exploit the hardware resources.

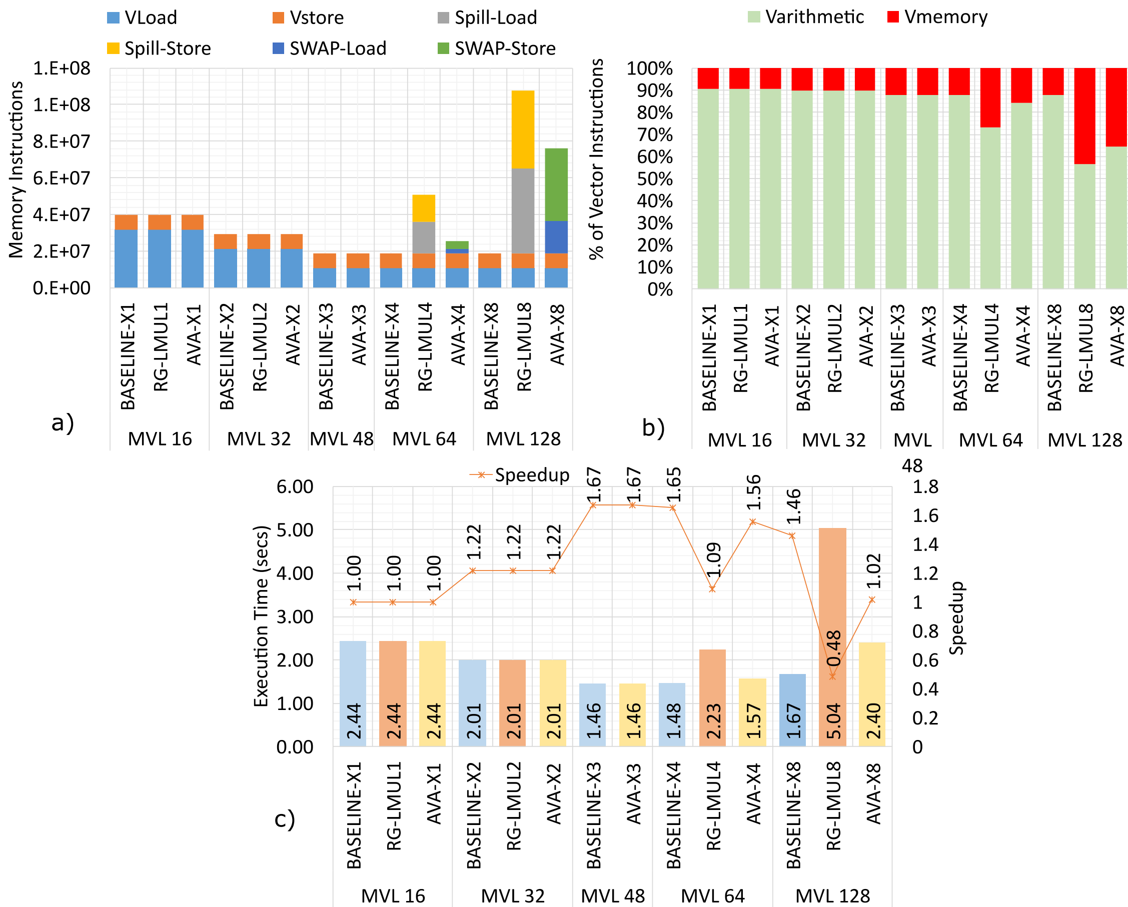

Figure 3.13 *LavaMD2 performance evaluation. a) Vector Memory Instruction count including spill operations generated by the compiler and swap operations generated by AVA. b) % of vector instruction, c) Execution-time and speedup compared to BASELINE-X1.*

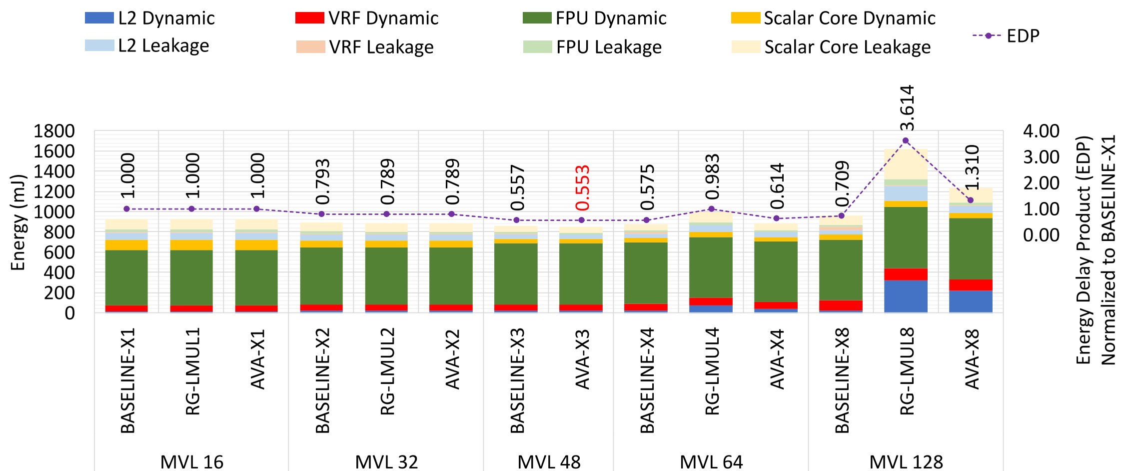

Figure 3.14 *LavaMD2 energy consumption (left axis) and normalized Energy Delay Product (right axis) for different configurations.*

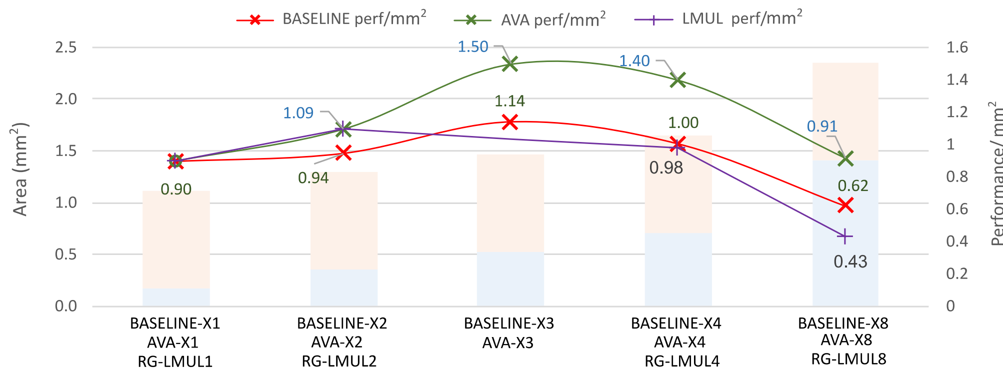

Figure 3.15 *Area results obtained from McPAT for 22nm technology node, and performance/mm² for each configuration.*

3.4.4 Particle-Filter

Performance Evaluation. For Particle-Filter, the compiler requires 13 architectural vector registers to generate the final binary, which implies that for RG-LMUL2, AVA-X2, and AVA-X3, no spill/swap operations are added. On the other hand, spill/swap operations are generated for RG-LMUL4, RG-LMUL8, and AVA-X8, as shown in Figure 3.16.a. However, although there is an increase in the number of vector memory instructions (spill and swap operations), when comparing with the number of vector arithmetic instructions, the percentage is negligible, representing only 0.15% of vector memory instructions for the larger configuration. Then, those vector memory operations are perfectly hidden beneath vector arithmetic execution, achieving similar performance levels as the corresponding BASELINE configuration as shown in Figure 3.16.c.

Energy Evaluation. Figure 3.17 shows energy consumption (left axis) and Energy Delay Product (EDP) efficiency metric (right axis) for Particle-Filter. For Particle-Filter, spill/swap operations are generated for RG-LMUL4, RG-LMUL8, and AVA-X8. However, these extra operations represent a negligible percentage compared to the overall vector instruction count. As a consequence, as the MVL is increased, less total energy is consumed. Dynamic energy is almost constant for the VPU, while there is a notable reduction in the scalar core because of the reduction in the instruction count. Since larger configurations improve performance, leakage energy is reduced for AVA and RG. On the contrary, BASELINE-X2, BASELINE-X3, BASELINE-X4, and BASELINE-X8 configurations double the leakage in each configuration because they are implementing larger multi-ported VRF memories from 16KB up to 64KB. Then, both RG and AVA configurations consume less energy than the equivalent BASELINE configuration. Compared with the BASELINE-X1 configuration, AVA saves 30% of the overall energy

consumption by reconfiguring for long vectors. Finally, looking at the EDP results, AVA-X8 becomes the most efficient configuration for AVA.

Performance/mm². Figure 3.18 shows the performance/mm² achieved for AVA when comparing versus the BASELINE configurations. When reconfiguring AVA for longer MVL configurations, AVA-X8 achieves the best performance/mm² efficiency. Although swap operations are generated for AVA-X8, this does not impact on the final performance, achieving the same performance levels as BASELINE-X8. Therefore, with much less area, AVA is more performance/mm² efficient.

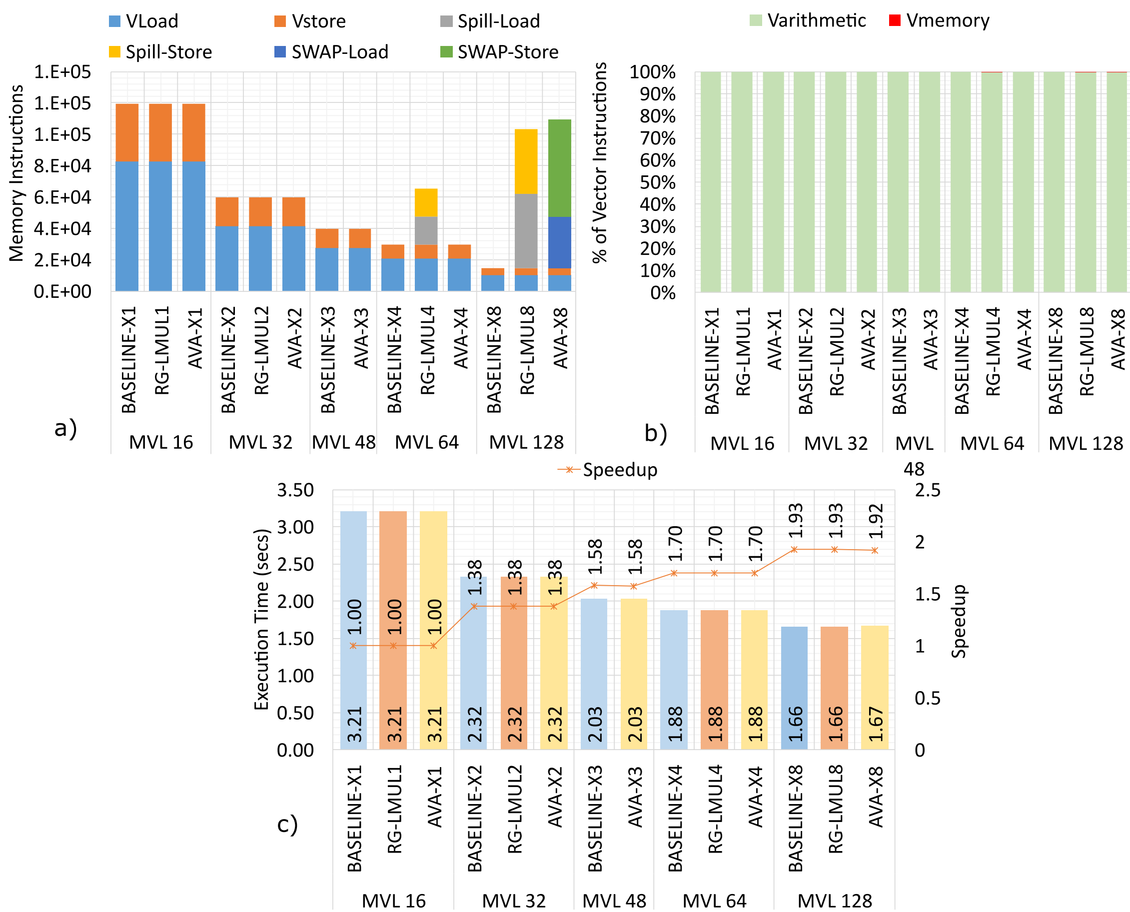

Figure 3.16 Particle-Filter performance evaluation. a) Vector Memory Instruction count including spill operations generated by the compiler and swap operations generated by AVA. b) % of vector instruction, c) Execution-time and speedup compared to BASELINE-X.

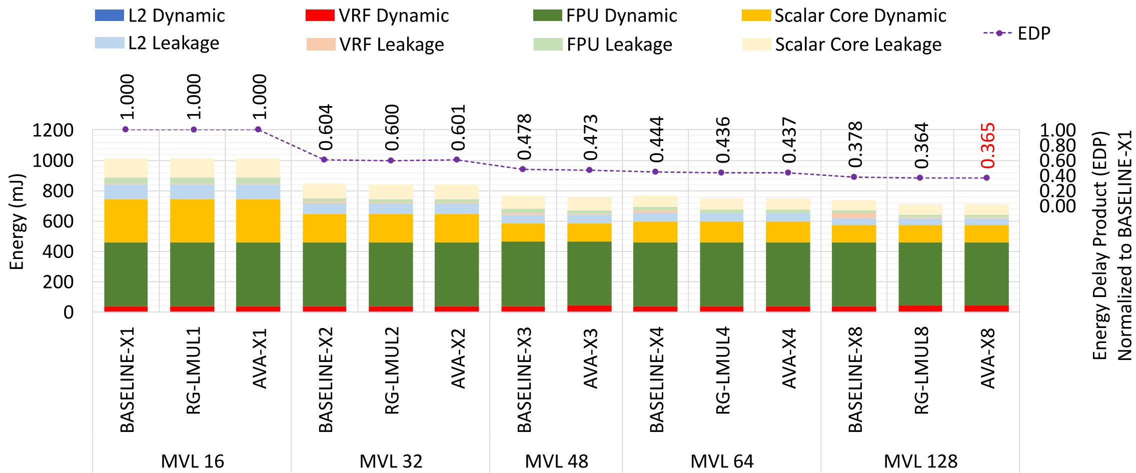

Figure 3.17 *Particle-Filter energy consumption (left axis) and normalized Energy Delay Product (right axis) for different configurations.*

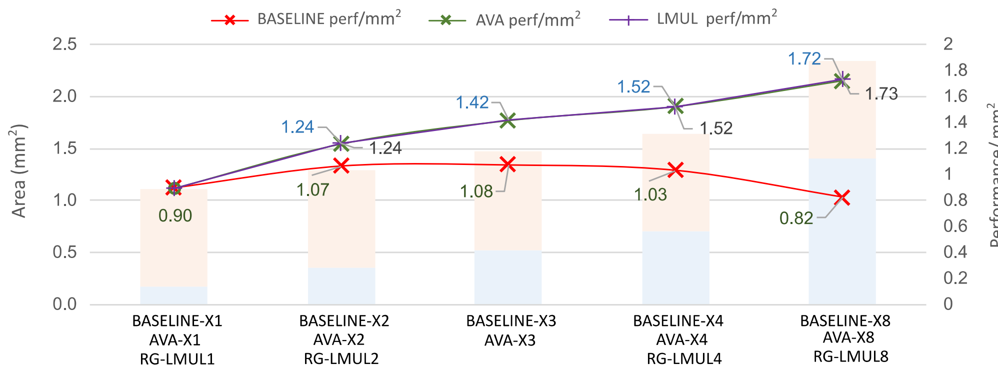

Figure 3.18 *Area results obtained from McPAT for 22nm technology node, and performance/mm² for each configuration.*

3.4.5 Somier

Performance Evaluation. For Somier, the vector compiler uses 13 architectural vector registers to generate the final binary. Spill/swap operations are generated only for RG-LMUL8 and AVA-X8. For RG-LMUL8 there was an increase in the percentage of memory operations from 46% to 68%, as shown in Figure 3.19.b. For AVA-X8, few swap operations were generated. Figure 3.19.c shows the performance results. In this case, the BASELINE-X4, RG-LMUL4, and AVA-X4 achieve the best speedup with 1.43x. For AVA-X8 and RG-LMUL8 there was a small performance degradation because of the additional memory traffic.

Energy Evaluation. Figure 3.20 shows energy consumption (left axis) and Energy Delay Product (EDP) efficiency metric (right axis) for Somier. Somier features a low

percentage of vectorization; therefore, dynamic energy in the scalar core represents the main energy contributor for all the configurations. VRF leakage for BASELINE-X8 also represents an important energy contributor. When comparing AVA-X8 with BASELINE-X8, it is clear the advantage of having an 8KB VRF where leakage contribution does not cause a big impact on the overall energy consumption. Additionally, spill code and swap operations are generated only for AVA-X8 and RG-LMUL8, increasing the L2 dynamic energy, especially for RG. Finally, looking at the EDP results, AVA-X4 becomes the most efficient configuration.

Performance/mm². Figure 3.21 shows the performance/mm² achieved for AVA when comparing versus the BASELINE configurations. When reconfiguring AVA for longer MVL configurations, AVA-X4 achieves the best performance/mm² efficiency. For AVA-X8, there is a decrease in efficiency because of the performance degradation caused by the extra swap operations required.

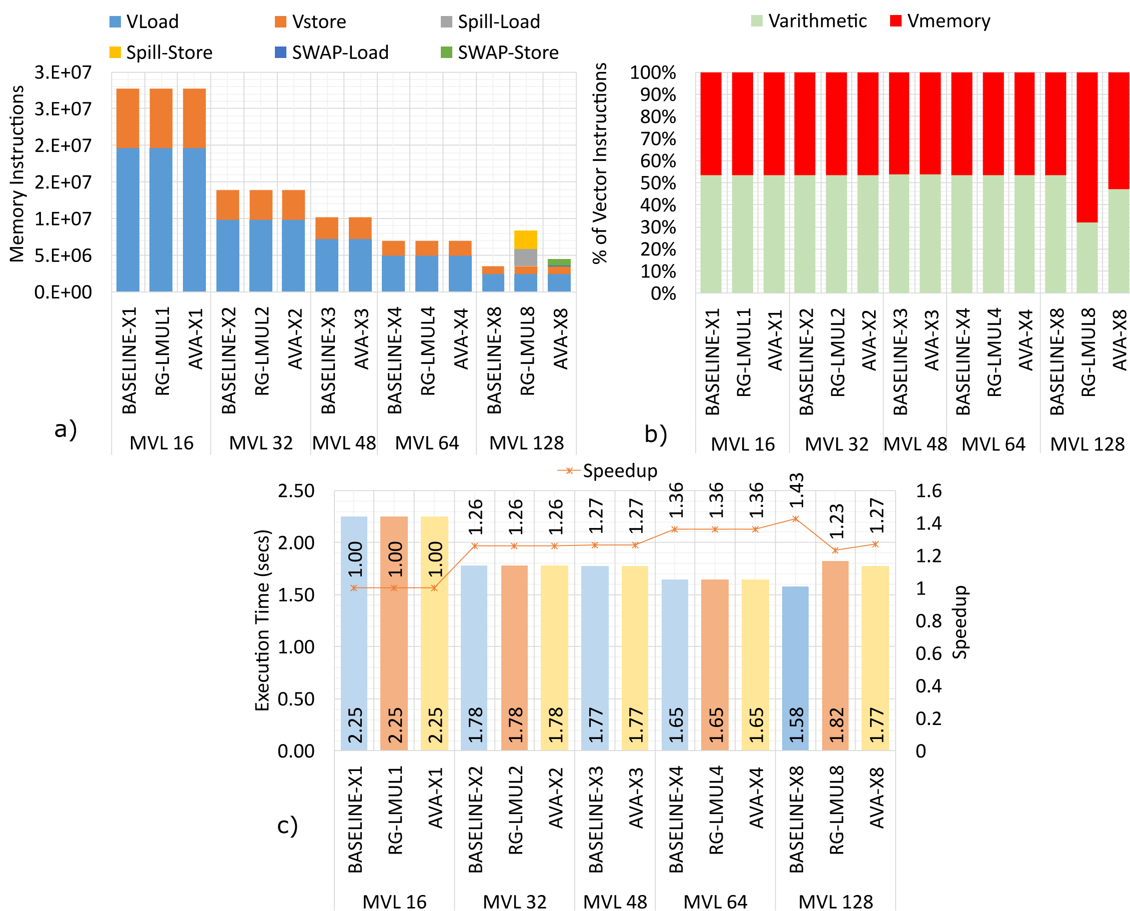

Figure 3.19 Somier performance evaluation. a) Vector Memory Instruction count including spill operations generated by the compiler and swap operations generated by AVA. b) % of vector instruction, c) Execution-time and speedup compared to BASELINE-X1.

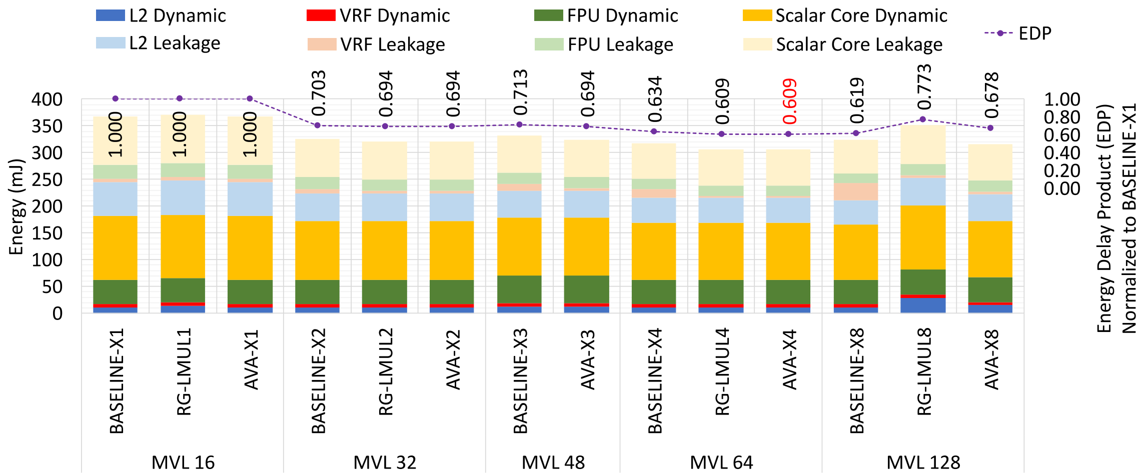

Figure 3.20 *Somier energy consumption (left axis) and normalized Energy Delay Product (right axis) for different configurations.*

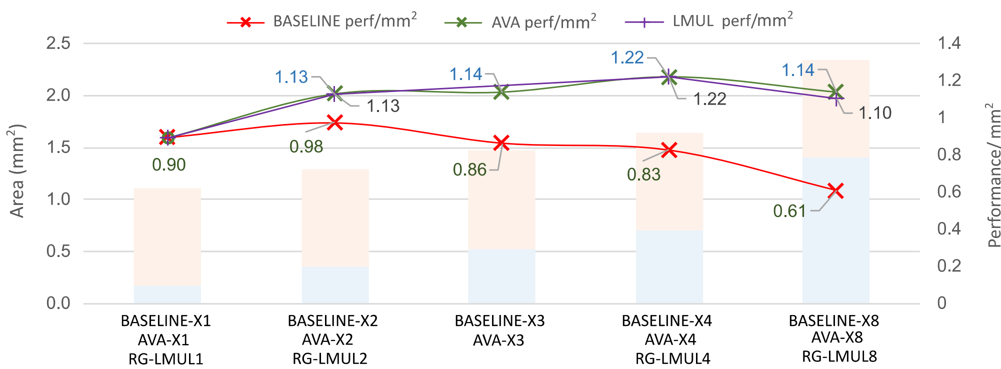

Figure 3.21 *Area results obtained from McPAT for 22nm technology node, and performance/mm² for each configuration.*

3.4.6 Swaptions

Performance Evaluation. Finally, for Swaptions, the vector compiler uses 24 architectural vector registers to generate the final binary, which implies that for RG-LMUL2, RG-LMUL4, and RG-LMUL8, spill code is generated as shown in Figure 3.22.a, causing an increase in the percentage of memory operations from 12% in the BASELINE-X1 configuration up to 34% in the RG-LMUL8 configuration as shown in Figure 3.22.b. For AVA, the swap operations appear starting from AVA-X3, obtaining almost the same number as the compiler-generated spill code for RG. AVA-X8 achieves a speedup of 1.78x while the BASELINE-X8 configuration achieves 2.15x with respect to the BASELINE-X1 configuration.

Energy Evaluation. Figure 3.23 shows energy consumption (left axis) and Energy Delay Product (EDP) efficiency metric (right axis) for Swaptions. Swaptions generate an important number of spill/swap operations for the RG-LMUL8 and AVA-X8, leading to extra energy dissipation wasted to support those operations. However, even consuming extra energy on the swap operations, energy savings are obtained compared with the short vector configurations by reconfiguring to AVA-X8.

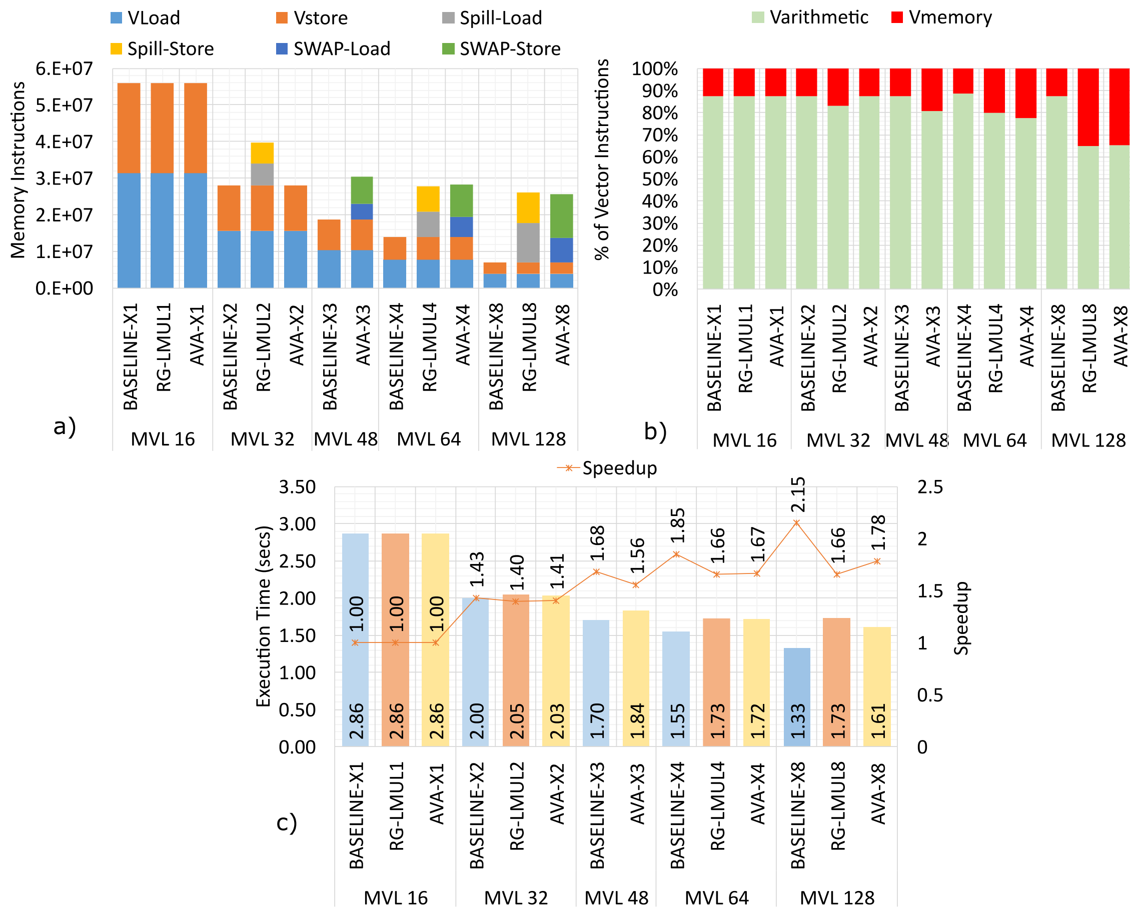

Figure 3.22 *Swaptions performance evaluation. a) Vector Memory Instruction count including spill operations generated by the compiler and swap operations generated by AVA. b) % of vector instruction, c) Execution-time and speedup compared to BASELINE-X1.*

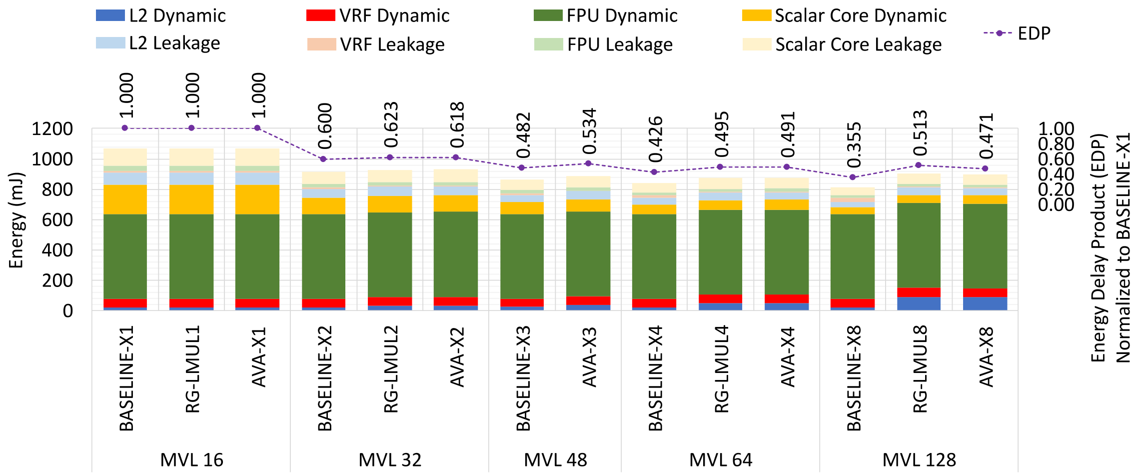

Figure 3.23 *Swaptions energy consumption (left axis) and normalized Energy Delay Product (right axis) for different configurations.*

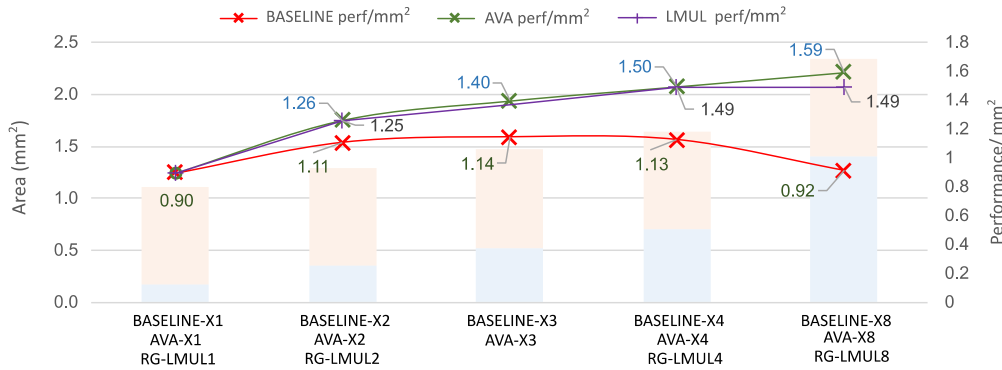

Figure 3.24 *Area results obtained from McPAT for 22nm technology node, and average performance/mm² for each configuration.*

Performance/mm². Figure 3.24 shows the performance/mm² achieved for AVA when comparing versus the BASELINE configurations. When reconfiguring AVA for longer MVL configurations, AVA-X8 achieves the best performance/mm² efficiency. Although for AVA-X3, AVA-X4, and AVA-X8 are generated swap operations, performance close to the BASELINE configurations is achieved. Therefore, with much less area, AVA is more performance/mm² efficient.

As shown above, AVA provides performance improvements for all the evaluated applications, being competitive with BASELINE designs for longer vectors. Additionally, as demonstrated, although there is additional memory traffic, AVA provides energy savings by reconfiguring for longer MVLs. Finally, Ava offers a higher performance/mm² efficiency.

3.5 Synthesis and place-and-route

Finally, we also perform experiments with design automation tools to get accurate results for the area and achievable frequency. Towards this goal, we added the required AVA support to an in-house VPU. We present the synthesis and place-and-route results for AVA and BASELINE-X8 configurations. To provide the 4R-2W VRF, we implemented the LVT technique [55] which provides multi-ported memories at the cost of replicating and banking dual-port memories.

We obtain area, power, and achievable frequency using Cadence tools, Genus for synthesis, and Innovus for place-and-route. We selected the GLOBALFOUNDRIES 22FDX 8T technology libraries, and we implemented the VRF slices using the Synopsys High-Performance Dual-Port SRAM cell-based Register File Memory Compiler (R2PH). The target frequency was 1GHz.

Table 28. *Post-place-and-route results*

	WNS (ns)	Power (mW)	Area (mm^2)	Density
BASELINE-X8	-0.190	2290	3.90	61%
-VRF Macros		388	1.252	
AVA	+0.119	1732	1.98	61.8%
-AVA structures		5.266	0.0042	-
-VRF Macros		184	0.257	

Post-place-and-route results for the typical corner (TT 0.8V 25C^{o}) are summarized in Table 28, and the obtained layouts are shown in Figure 3.25, for both configurations. Regarding area results, for the AVA configuration, the required AVA hardware structures incur a negligible 0.21% area overhead. On the other hand, the total chip area is reduced by 50.7% compared with the BASELINE-X8 configuration, validating the McPAT results.

Regarding the timing performance, target constraints are met only for AVA with a positive slack of 0.119ns. However, for BASELINE-X8 there is a negative slack of -0.244ns, due to the critical paths stemming from the longer wires between the SRAMs and the lane logic. Based on our synthesis and place-and-route experiments, we can confirm that the small size required for AVA helps to achieve higher working frequencies due to a higher robustness against different physical floor planning options.

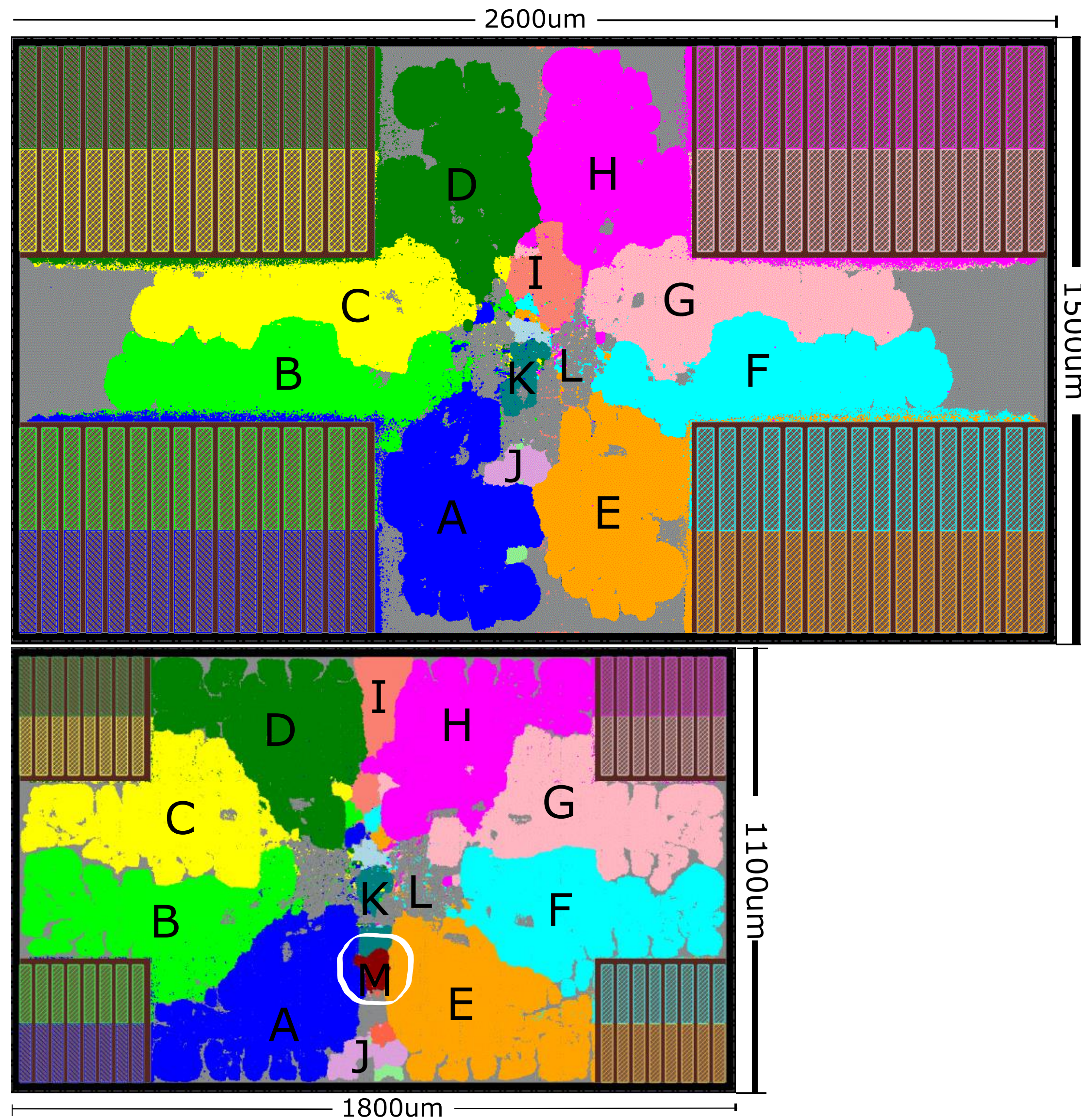

Figure 3.25 *BASELINE-X8 design is on the top and AVA on the bottom. PnR results for 22 nm technology of two instances of the VPU with eight lanes, highlighting main internal blocks: A) lane 1; B) lane 2; C) lane 3; D) lane 4; E) lane 5; F) lane 6; G) lane 7; H) lane 8; I) Vector Memory Unit; J) ROB; K) Instruction queue; L) Remaining modules such as memory queue, renaming unit and ring lane interconnection; M) AVA structures. Circled area marks the added AVA structures. VRF memory macros can be identified on the corners.*

3.6 Related Work

AVA partially leverages different computer architecture techniques that were developed for out-of-order cores, VLIW processors, and GPUs. While the concepts might be familiar at high level, we adapt and substantially tailor these techniques for vector processors to propose the novel adaptable VRF design. The following lines briefly describe the related work.

Different alternatives to exploit efficient use of physical registers was widely studied. González et al. [66] [67] proposed a dynamic register renaming approach where the key idea is to delay the allocation of the physical registers until write-back. To this end, a technique termed as Virtual-Physical Registers was proposed. Virtual-Physical Registers are not related to any storage location; they are merely tags to keep track of the dependencies and are therefore not related to AVA. Although AVA proposes a two-stage renaming unit, unlike the Virtual-Physical Registers concept, our VVRs are assigned at renaming time, while physical registers are assigned at issue time, and combined with the RAC counters, exploiting the use of the vector registers as soon as they can be reused.

Based on the fact that a physical register can be reused when it is guaranteed that the value in it can never be used by any later instruction, several studies [65], [68], [69] associated a counter with each physical register, to keep track of the pending read operations. In these techniques, a physical register is freed whenever the associated counter is zero. Such aggressive register reclamation schemes enable physical register usage to closely match the true lifetime of registers. AVA exploits the concept of aggressive register reclamation to free a physical register that will not be longer used. Additionally, AVA extend the use of the associated counters to decide the best option to perform swaps between Physical and VVRs.

The idea of using memory to provide a backing store to the register file has been has also been widely studied for out-of-order cores [70], VLIW processors [71], and GPUs [72] [73]. In this work, we apply it to vector processors as a key mechanism to offer a variety of MVL configurations. Additionally, we have unified the idea of a two-level VRF with the concept of VVRs and physical registers, which in combination with the Swap-Mechanism presents a balanced design which is able to efficiently handle different DLP patters.

3.7 Summary

This chapter introduces AVA, an Adaptable Vector Architecture with the ability to reconfigure the MVL, unlocking the benefits of having a longer vector microarchitecture when abundant DLP is present. Our results demonstrate that by having a modest VPU designed for short vectors, plus our novel scheduling mechanism, it is possible to obtain a very competitive performance when comparing AVA with the equivalent BASELINE long vector configurations. As a first approximation, we obtain area and energy metrics from McPAT, demonstrating that AVA can save around 53% of the total VPU area compared with a configuration for long vectors. Additionally, we demonstrate that supporting long vectors not only improve performance, but also leads to energy savings for several workloads. Finally, we implemented AVA at RTL level, synthesized and place-and-routed in 22nm technology, demonstrating that AVA not only provides an area-efficient design, but also allows higher frequencies.

Chapter 4

Conclusions and Future Work

It would appear that we have reached the limits of what
it is possible to achieve with computer technology,
although one should be careful with such statements,
as they tend to sound pretty silly in 5 years.

John Von Neumann,
(1949)

This chapter presents the conclusions of the research done in this thesis work. Additionally, future research directions are highlighted.

4.1 Conclusions

Vector processors provide one effective way to achieve high performance and efficiency when applications contain abundant DLP. On the software side, modern scientific applications are getting more diverse, and the vector lengths in those applications vary widely. On the hardware side, today, there are two main design trends for vector processors: vector processors designed for long and short vectors lengths. Long vector designs are limited to a specialized subset of applications, where high DLP must be present to achieve excellent performance with very high efficiency. In contrast, Short vector designs are compatible with a more extensive range of applications. Short vector designs are area efficient and are "compatible" with applications having long vectors; however, these short vector architectures are not efficient as longer vector designs when executing high DLP code. To overcome this limitation imposed at the hardware level by the MVL design parameter, the main objective of this thesis is to propose a novel Vector Architecture able to adapt the microarchitecture according to the application characteristics to efficiently use the resources and improve performance and energy results.

To fulfill the main objective, several steps were done to develop a complete environment to test our ideas, as shown in the thesis timeline in Figure 4.1. In that sense, the first contribution corresponds to a complete framework for designing and evaluating vector architectures: the gem5 simulator and the McPAT framework extended with a parameterizable vector architecture model, and the RiVEC benchmark suite. The primary goal of these tools was to serve as a base platform for this research. This primary goal was successfully accomplished by presenting a study of the applications when executed on the gem5 and McPAT vector architecture models (September 2019). Additionally, in Jun 2020, we open-source these tools trying to contribute to the computer architecture community, expecting to be used as the base for research on RISC-V Vector Architectures. Also, we invite the community to contribute to this effort by adding missing features to the gem5 model, or by adding new applications to the RiVEC benchmark Suite.

After one year and a half that we open source the complete framework, we are glad to see that the tools have been well accepted by the community and are being used. In fact, adding missing support or updating to the latest specs is something the community and we are continually working on. For the same reason, as shown in Figure 4.1, gem5 VPU model and RiVEC bars do not have an ending. Some simple metrics which allows

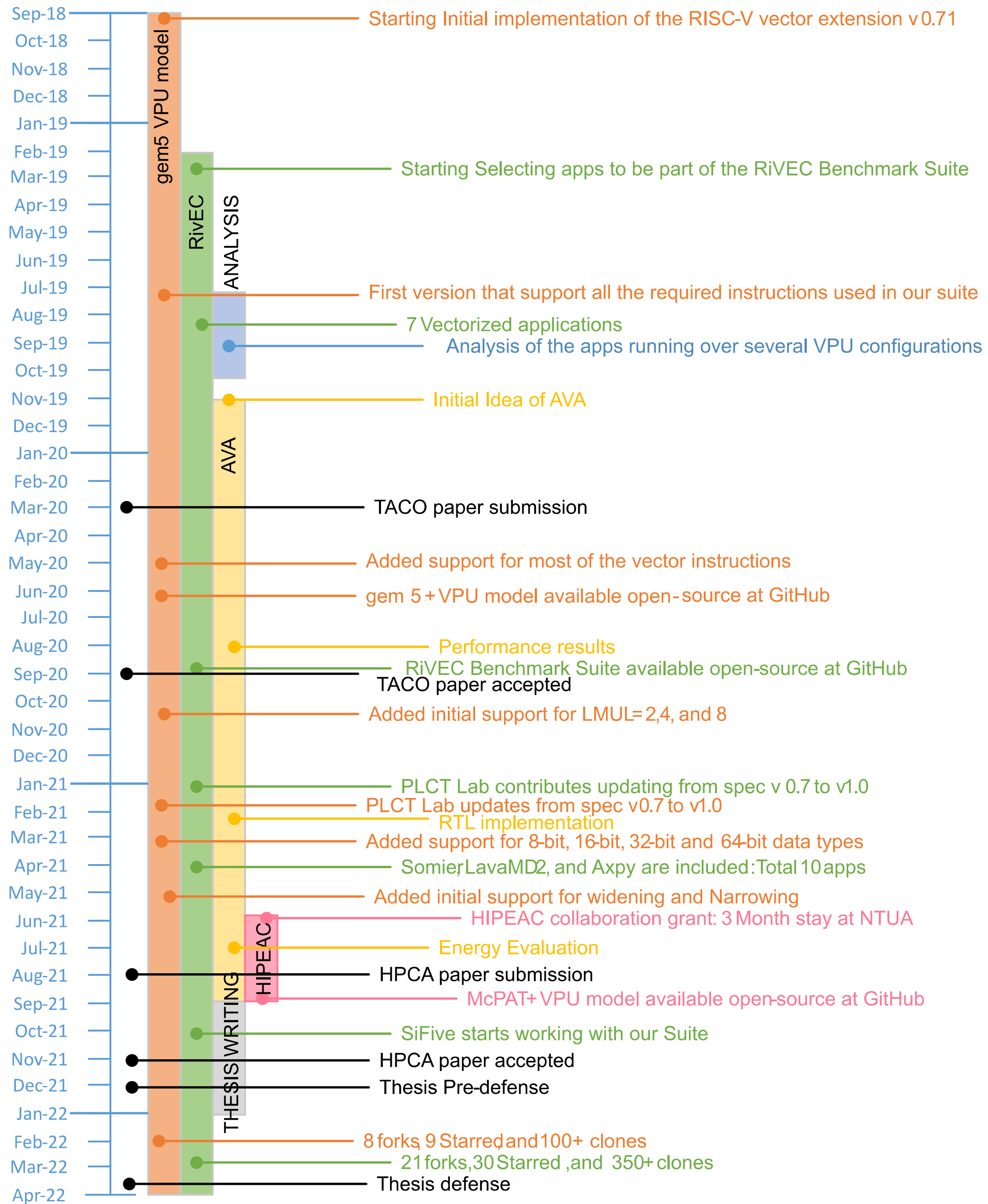

Figure 4.1 *Thesis timeline.*

us to know that people is using our tools are the GitHub repository insights. For example, the RiVEC benchmark suite was the most popular with around 2000 views and more than 350 Clones, 30 starts, and 21 forks from people working actively. Companies such as SiFive are also exploring our suite, extending and adapting it to their environment. People from the Chinese academy contribute to this effort by updating the RiVEC

benchmark suite from the RISC-V V extension version v0.71 to v1.0. The gem5 simulator also has been used by researchers. Regarding metrics, it has around 1000 views and 100 Clones, 9 starts, and 8 forks. Researchers have contributed by adding missing support such as 8-bit and 16-bit integer support, or reporting bugs that were successfully solved.

Finally, the main contribution of this thesis is called AVA, an Adaptable Vector Architecture that combines the area and resource efficiency characterizing short vector processors with the ability to handle large DLP applications, as allowed in long vector architectures. AVA combines different computer architecture techniques such as a two-stage renaming unit based on a new type of registers termed as Virtual Vector Registers (VVRs); a two-level VRF, that supports 64 VVRs whose MVL can be configured from 16 to 128 elements; and a novel two-stage vector issue unit in charge of scheduling the execution of the vector instructions. We showed that by implementing the required AVA structures in a vector processor, it is possible to achieve higher performance, energy efficiency, and area efficiency levels. Additionally, AVA can be implemented over SIMD multimedia extensions or vector architectures regardless of the target vector ISA. For example, talking about fixed-length ISAs, we can implement our technique on the AVX extension (VL=128-bits) and be able to support the execution of AVX2(VL=256-bits) and AVX-512(VL=512-bits) instructions on the same hardware. This can be very interesting for ultra-low-power embedded systems that cannot afford an AVX-512.

4.2 Future Work

This thesis opens the door to further optimizations for AVA. In particular, some vector instructions, such as vector reductions, only update the first element of the destination vector registers, while other vector instructions replicate the same value over all the elements of a vector register. In the AVA microarchitecture, vector registers are an important resource. As the MVL is increased, fewer physical vector registers are available. In that sense, implementing an extension (64-bit x 8 entries) of the P-VRF for holding those scalar results alleviates the data movement between the two-level of the VRF. Initial results on this idea are very promising, encouraging us to continue exploring this new optimization.

A second interesting optimization proposal devised by Francesco Minervini is the following. Vector register gather instructions read elements from a first source vector register at locations given by a second source vector register and writes those elements in a new vector register (vd[x] = vs2[vs1[x]]). Executing this operation is typically expensive in terms of total latency to be executed. For AVA, there are two possible

scenarios. In case the first source vector register that holds the elements is located in the M-VRF, the vector register gather instruction can be converted to something similar to a load indexed instruction, then loading the data in the order it is required to be written in the destination vector register. In a second scenario, if there is no free physical vector register to be assigned as the destination for the vector register gather instruction, instead of selecting one VVR to be sent to the M-VRF, this operation can be executed as a store indexed operation.

Similar to the previous optimization proposal, there are several optimizations that can exploit the fact of having a two-level VRF. For example, combining the concept of in-cache computing with AVA. In this scheme, some computation can be performed inside the lanes using the P-VRF and the functional units, and other computations can be performed in the M-VRF. Also, slides or register gather operations do not need a functional unit to be computed. This can influence reducing data movement between the P-VRF and the M-VRF, and improve performance.

We presented the base model, and evaluate it attaching the VPU to an in-order core. In that sense, one research direction of this work is to study the impact of attaching the VPU an out-of-order core. Applications like Particle-Filter which has intensive communication with the scalar core, suffers when attaching the VPU to an in-order core, because the scalar core stalls until the VPU finish the current computation of the vector instructions, to then, send back a scalar value to the scalar core. Using an out-of-order core, it would be possible to advance independent scalar instructions and also continue feeding the vector engine. However, this has several implications since an out-of-order core is a complex and costly design, which maybe for some systems is not affordable.

Another possible direction would be investigating on multicore designs, then exploiting both DLP and TLP. AVA allows to execute vector lengths of up to 128 elements when high DLP is present. If we combine this feature with multi-core or many-core configurations, the results can be very promising. Then, allowing to execute long vectors on commodity CPUs at the same area cost of modern CPUs as the Fujitsu A64FX, and not restricting the design to a specific subset of applications such as Aurora Tsubasa vector engine.

Chapter 5

Publications

If we knew what it was we were doing, it would not be called research, would it?

Albert Einstein

The research done in this thesis has resulted in the following publications.

5.1 Publications

The contents of this thesis led to the following publications:

- **Cristóbal Ramírez**, César Alejandro Hernández, Oscar Palomar, Osman Unsal, Marco Antonio Ramírez, and Adrián Cristal. 2020. *A RISC-V Simulator and Benchmark Suite for Designing and Evaluating Vector Architectures*. ACM Transactions on Architecture and Code Optimization (TACO). 17, 4, Article 38 (December 2020), 30 pages. DOI:https://doi.org/10.1145/3422667

- **Cristóbal Ramírez Lazo**, Enrico Reggiani, Carlos Rojas Morales, Roger Figueras Bagué, Luis Alfonso Villa Vargas, Marco Antonio Ramírez Salinas, Mateo Valero Cortés, Osman Sabri Unsal, Adrián Cristal. 2022. *Adaptable Register File Organization for Vector Processors*. 28th IEEE International Symposium on High-Performance Computer Architecture (HPCA 2022). To appear

The following publications are related but not included in this thesis:

- Enrico Reggiani, **Cristóbal Ramírez Lazo**, Roger Figueras Bagué, Adrián Cristal Kestelman, Mauro Olivieri, and Osman Unsal, *BiSon-e: A Lightweight and High-Performance Accelerator for Narrow Integer Linear Algebra Computing on the Edge*. 22nd Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'22). To appear.

- Jaume Abella, Calvin Bulla, Guillem Cabo, Francisco J Cazorla, Adrián Cristal, Max Doblas, Roger Figueras, Alberto González, Carles Hernández, César Hernández, Víctor Jiménez, Leonidas Kosmidis, Vatistas Kostalabros, Rubén Langarita, Neiel Leyva, Guillem López-Paradís, Joan Marimon, Ricardo Martínez, Jonnatan Mendoza, Francesc Moll, Miquel Moreteó, Julián Pavón, **Cristóbal Ramírez**, Marco A Ramírez, Carlos Rojas, Antonio Rubio, Abraham Ruiz, Nehir Sonmez, Víctor Soria, Lluís Terés, Osman Unsal, Mateo Valero, Iván Vargas, and Luís Villa. *An Academic RISC-V Silicon Implementation Based on Open-Source Components.* 2020 XXXV Conference on Design of Circuits and Integrated Systems (DCIS), 2020, pp. 1-6.

List of Figures

Figure 1.1 Basic structure of a Vector Architecture 3

Figure 1.2 Basic structure of a multi-lane Vector Architecture 3

Figure 1.3 Basic structure of a 512-bit SIMD multimedia implementation. 6

Figure 2.1 gem5 Vector Architecture Model. 19

Figure 2.2 VRF elements distribution for a MV=256 elements and eight-lane configuration. 22

Figure 2.3 Vector Load/Store buffer behavior 23

Figure 2.4 Two possible memory configurations. a) the VPU is connected to its own L1 cache. b) the VPU is connected to L2 cache. 24

Figure 2.5 Vector architecture model implemented on the McPAT framework 28

Figure 2.6 McPAT area evaluation. 66

Figure 2.7 Axpy runtime/speedup for different configurations. 67

Figure 2.8 Axpy performance/mm^2 efficiency 68

Figure 2.9 Axpy energy consumption (left axis) and normalized Energy Delay Product (right axis) for different configurations. 69

Figure 2.10 Blackscholes runtime/speedup for different configurations. 70

Figure 2.11 Blackscholes performance/mm^2 efficiency 71

Figure 2.12 Blackscholes energy consumption (left axis) and normalized Energy Delay Product (right axis) for different configurations. 71

Figure 2.13 Canneal runtime/speedup for different configurations 73

Figure 2.14 Canneal performance/mm^2 efficiency. 73

Figure 2.15 Canneal energy consumption (left axis) and normalized Energy Delay Product (right axis) for different configurations 74

Figure 2.16 Jacobi-2D runtime/speedup for different configurations. 75

Figure 2.17 Jacobi-2D performance/mm^2 efficiency. 76

Figure 2.18 Jacobi-2D energy consumption (left axis) and normalized Energy Delay Product (right axis) for different configurations. 76

Figure 2.19 LavaMD2 runtime/speedup for different configurations. 77

Figure 2.20 LavaMD2 performance/mm^2 efficiency. 78

Figure 2.21 LavaMD2 energy consumption (left axis) and normalized Energy Delay Product (right axis) for different configurations. 79

Figure 2.22 Pathfinder runtime/speedup for different configurations. 80

Figure 2.23 Pathfinder performance/mm^2 efficiency. 80

Figure 2.24 Pathfinder energy consumption (left axis) and normalized Energy Delay Product (right axis) for different configurations. 81

Figure 2.25 Particle-Filter runtime/speedup for different configurations. 82

Figure 2.26 Particle-Filter performance/mm^2 efficiency. 83

Figure 2.27 Particle-Filter energy consumption (left axis) and normalized Energy Delay Product (right axis) for different configurations. 84

Figure 2.28 Somier runtime/speedup for different configurations. 85

Figure 2.29 Somier performance/mm^2 efficiency. 85

Figure 2.30 Somier energy consumption (left axis) and normalized Energy Delay Product (right axis) for different configurations. 86

Figure 2.31 Streamcluster runtime/speedup for different configurations. 87

Figure 2.32 Streamcluster performance/mm^2 efficiency. 88

Figure 2.33 Streamcluster energy consumption (left axis) and normalized Energy Delay Product (right axis) for different configurations. 88

Figure 2.34 Swaptions speedup for different cache configurations. 89

Figure 2.35 Swaptions performance/mm^2 efficiency. 90

Figure 2.36 Swaptions energy consumption (left axis) and normalized Energy Delay Product (right axis) for different configurations. 90

Figure 3.1 AVA microarchitecture overview. The new hardware additions are highlighted in green, involving the second stage of the renaming unit (VRF-

Mapping) and the first stage of the Vector Issue Unit (pre-issue queue and Swap-Mechanism). 98

Figure 3.2. Main structures of the two-stages renaming unit. 99

Figure 3.3 Base cases when renaming a vector instruction. 101

Figure 3.4 M-VRF definition. Gray box shows the main function, yellow box shows the M-VRF definition, and blue box shows the input dataset. 104

Figure 3.5 Register Mapping example 108

Figure 3.6 Area results for AVA and the different BASELINE configurations obtained from McPAT 113

Figure 3.7 Axpy performance evaluation. a) Vector Memory Instruction count including spill operations generated by the compiler and swap operations generated by AVA. b) % of vector instruction, c) Execution-time and speedup compared to BASELINE-X1. 114

Figure 3.8 Axpy energy consumption (left axis) and normalized Energy Delay Product (right axis) for different configurations. 115

Figure 3.9 Area results obtained from McPAT for 22nm technology node, and performance/mm^2 for each configuration. 116

Figure 3.10 Blackscholes performance evaluation. a) Vector Memory Instruction count including spill operations generated by the compiler and swap operations generated by AVA. b) % of vector instruction, c) Execution-time and speedup compared to BASELINE-X1. 117

Figure 3.11 Blackscholes energy consumption (left axis) and normalized Energy Delay Product (right axis) for different configurations. 118

Figure 3.12 Area results obtained from McPAT for 22nm technology node, and average performance/mm^2 for each configuration. 118

Figure 3.13 LavaMD2 performance evaluation. a) Vector Memory Instruction count including spill operations generated by the compiler and swap operations generated by AVA. b) % of vector instruction, c) Execution-time and speedup compared to BASELINE-X1. 120

Figure 3.14 LavaMD2 energy consumption (left axis) and normalized Energy Delay Product (right axis) for different configurations. 120

Figure 3.15 Area results obtained from McPAT for 22nm technology node, and performance/mm^2 for each configuration. 121

Figure 3.16 Particle-Filter performance evaluation. a) Vector Memory Instruction count including spill operations generated by the compiler and swap operations generated by AVA. b) % of vector instruction, c) Execution-time and speedup compared to BASELINE-X. 122

Figure 3.17 Particle-Filter energy consumption (left axis) and normalized Energy Delay Product (right axis) for different configurations. 123

Figure 3.18 Area results obtained from McPAT for 22nm technology node, and performance/mm^2 for each configuration. 123

Figure 3.19 Somier performance evaluation. a) Vector Memory Instruction count including spill operations generated by the compiler and swap operations generated by AVA. b) % of vector instruction, c) Execution-time and speedup compared to BASELINE-X1. 124

Figure 3.20 Somier energy consumption (left axis) and normalized Energy Delay Product (right axis) for different configurations. 125

Figure 3.21 Area results obtained from McPAT for 22nm technology node, and performance/mm^2 for each configuration. 125

Figure 3.22 Swaptions performance evaluation. a) Vector Memory Instruction count including spill operations generated by the compiler and swap operations generated by AVA. b) % of vector instruction, c) Execution-time and speedup compared to BASELINE-X1. 126

Figure 3.23 Swaptions energy consumption (left axis) and normalized Energy Delay Product (right axis) for different configurations. 127

Figure 3.24 Area results obtained from McPAT for 22nm technology node, and average performance/mm^2 for each configuration. 127

Figure 3.25 BASELINE-X8 design is on the top and AVA on the bottom. PnR results for 22 nm technology of two instances of the VPU with eight lanes, highlighting main internal blocks: A) lane 1; B) lane 2; C) lane 3; D) lane 4; E) lane 5; F) lane 6; G) lane 7; H) lane 8; I) Vector Memory Unit; J) ROB; K) Instruction queue; L) Remaining modules such as memory queue, renaming unit and ring lane interconnection; M) AVA structures. Circled area marks the added AVA structures. VRF memory macros can be identified on the corners. 129

Figure 4.1 Thesis timeline. 134

List of Tables

Table 1. RiVEC Benchmark Suite. 33

Table 2. Application characteristics. 34

Table 3. Axpy input data sets 36

Table 4. Instruction Level Characterization of Axpy application. 37

Table 5. Blackscholes input data sets 39

Table 6. Instruction Level Characterization of the Blackscholes application. 40

Table 7. Canneal input data sets 43

Table 8. Instruction Level Characterization of the Canneal application. 44

Table 9. Jacobi-2D input data sets 45

Table 10. Instruction Level Characterization of the Jacobi-2D application. 46

Table 11. LavaMD2 input data sets 47

Table 12. Instruction Level Characterization of the LavaMD2 application. 48

Table 13. Pathfinder input data sets 50

Table 14. Instruction Level Characterization of the Pathfinder application. 51

Table 15. Particlefilter input data sets 53

Table 16. Instruction Level Characterization of the ParticleFilter application. 54

Table 17. Somier input data sets 55

Table 18. Instruction Level Characterization of the Somier application. 56

Table 19. Streamcluster input data sets 57

Table 20. Instruction Level Characterization of the Streamcluster application 59

Table 21. Swaptions input data sets 61

Table 22. Instruction Level Characterization of the Swaptions application. 62

Table 23. gem5 evaluation environment. 63

Table 24. Physical Vector Register File Configurations. 104

Table 25. Vector Processing Unit Configurations... 110

Table 26. AVA and RG configurations and their corresponding equivalence with the five configurations in Table 25... 111

Table 27. Applications from RiVEC Benchmark Suite... 111

Table 28. Post-place-and-route results 128

Bibliography

[1] M. J. Flynn, "Very high-speed computing systems," in *IEEE* , 1966.

[2] H. J. &. P. D.A, Computer Architecture: A quantitative Approach , Inc., 6th edition., Morgan Kaufmann Publishers Inc., 2019.

[3] Intel, "Instruction Set Extensions and Future Features Programming Reference," 2020.

[4] MIPS, "MIPS® Architecture for Programmers : The MIPS64 SIMD Architecture Module," 2016.

[5] ARM, "Neon Programmer's Guide," 2013.

[6] "SX-Aurora TSUBASA Architecture Guide, Revision 1.1," NEC Corporation, 2018.

[7] C. C. systems, "CRAY-2 Computer System Functional Description," 1985.

[8] 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. 02, pp. 26-39, 2017.

[9] "Introducing SVE2," [Online]. Available: https://developer.arm.com/tools-and-software/server-and-hpc/compile/arm-instruction-emulator/resources/tutorials/sve/sve-vs-sve2/introduction-to-sve2.

[10] "Working draft of the proposed RISC-V V vector extension," RISC-V, 2021. [Online]. Available: https://github.com/riscv/riscv-v-spec. [Accessed 1 July 2021].

[11] R. M. Russell, "The CRAY-1 computer system," *Communications of the ACM,* vol. 21, p. 63–72, 1977.

[12] C. R. Inc, "The CRAY-2 computer system," 1985.

[13] K. Asanović, "Vector Microprocessors," Ph.D. thesis, University of California , Berkeley, 1998.

[14] S. Rivoire, R. Schultz, T. Okuda and C. Kozyrakis, "Vector Lane Threading," in *International Conference on Parallel Processing (ICPP'06)*, 2006.

[15] M. Cavalcante, F. Schuiki, F. Zaruba, M. Schaffner and L. Benini, "Ara: A 1 GHz+ Scalable and Energy-Efficient RISC-V Vector Processor with Multi-Precision Floating Point Support in 22 nm FD-SOI," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* 2020.

[16] C. E. Kozyrakis and D. A. Patterson, "Scalable, vector processors for embedded systems," *IEEE Micro,* 2003.

[17] R. Espasa, M. Valero and J. Smith, "Vector Architectures, past, present and future," in *12th international conference on Supercomputing, ACM*, 1998.

[18] R. Espasa and Mateo Valero, "Decoupled Vector Architectures," in *Second International Symposium on High-Performance Computer Architecture*, San Jose, CA, USA, 1996.

[19] R. Espasa, "Advanced Vector Architectures," Ph.D. thesis, Universitat Politècnica de Catalunya, 1997.

[20] R. Espasa, F. Ardanaz, J. Emer, S. Felix, J. Gago, R. Gramunt, I. Hernandez, T. Juan, G. Lowney, M. Mattina and A. Seznec, "Tarantula: A Vector Extension to the Alpha Architecture (ISCA)," in , Anchorage, 2002.

[21] D. Abts, S. Scott and D. J. Lilja, "So many states, so little time: Verifying memory coherence in the Cray X1," in *International Parallel and Distributed Processing Symposium*, 2003..

[22] D. Abts, A. Bataineh, S. Scott, G. Faanes, J. Schwarzmeier, E. Lundberg, T. Johnson, M. Bye and G. Schwoerer, "The Cray BlackWidow: A Highly Scalable Vector Multiprocessor," in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, Reno, NV, USA, 2010.

[23] Y. Yohei, "Vector Engine Processor of NEC's Brand-New Supercomputer SX-Aurora TSUBASA," NEC Corporation, 2018.

[24] R. Lee and J. Huck, "64-bit and multimedia extensions in the PA-RISC 2.0 architecture," *COMPCON '96. Technologies for the Information Superhighway Digest of Papers,* pp. 152-160, 1996.

[25] L. Kohn, G. Maturana, M. Tremblay, A. Prabhu and G. Zyner, "The visual instruction set (VIS) in UltraSPARC," *Digest of Papers. COMPCON'95. Technologies for the Information Superhighway,* pp. 462-469, 1995.

[26] A. Peleg and U. Weiser, "MMX technology extension to the Intel architecture," *IEEE Micro,,* vol. 16, pp. 42-50, 1996.

[27] S. K. Raman, V. Pentkovski and J. Keshava, "Implementing streaming SIMD extensions on the Pentium III processor," *IEEE Micro,* vol. 20, pp. 47-57, 2000.

[28] Intel, "Intel Advanced Vector Extensions Programming Referencce," 2011.

[29] Intel, "Intel Advanced Vector Extensions 512," [Online]. Available: https://www.intel.es/content/www/es/es/architecture-and-technology/avx-512-overview.html. [Accessed 11 30 2021].

[30] "RISC-V," [Online]. Available: https://riscv.org. [Accessed 2021].

[31] C. Chen, X. Xiang, C. Liu, Y. Shang, R. Guo, D. Liu, Y. Lu, Z. Hao, J. Luo, Z. Chen, C. Li, Y. Pu, J. Meng, X. Yan, Y. Xie and X. Qi, "Xuantie-910: A Commercial Multi-Core 12-Stage Pipeline Out-of-Order 64-bit High Performance RISC-V Processor with Vector Extension," in *47th Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[32] Sifive, "SiFive Performance P270," 22 Jun 2021. [Online]. Available: https://www.sifive.com/cores/performance-p270. [Accessed 30 Nov 2021].

[33] "European Processor Initiative," [Online]. Available: https://www.european-processor-initiative.eu/accelerator/. [Accessed May 2020].

[34] T. Yoshida, "Introduction of Fujitsu's HPC processor for the Post-K," in *Hot Chips*, 2016.

[35] "TOP500, the list," [Online]. Available: https://www.top500.org/lists/top500/2020/11/. [Accessed 10 03 2021].

[36] D. Abts and e. al., "The Cray BlackWidow: A Highly Scalable Vector Multiprocessor," in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, Reno, NV, USA, 2010.

[37] C. Ramírez, C. Hernández, O. Palomar, O. Unsal, M. A. Ramírez and A. Cristal, "A RISC-V Simulator and Benchmark Suite for Designing and Evaluating Vector Architectures," *ACM Trans. Archit. Code Optim.,* vol. 17, 2020.

[38] C. R. Lazo, "gem5 vector architecture model," [Online]. Available: https://github.com/RALC88/gem5/tree/develop. [Accessed 1 12 2021].

[39] C. R. Lazo, "McPAT - vector architecture model," [Online]. Available: https://github.com/RALC88/mcpat. [Accessed 1 12 2021].

[40] C. R. Lazo, "RiVEC Benchmark Suite," [Online]. Available: https://github.com/RALC88/riscv-vectorized-benchmark-suite. [Accessed 1 12 2021].

[41] N. Binkert, B. Beckmann, G. Black , S. K., A. Saidi, A. Basu , J. Hestness, D. Hower , T. Krishna , S. Sardashti, R. Sen, K. Sewell , M. Shoaib, N. Vaish, M. D. Hill and D. A. Wood, "The gem5 simulator," *SIGARCH Computer Architecture News,* vol. 39, p. 1–7, 2011.

[42] J. A. Stratton, C. Rodrigues, . I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, . G. D. Liu and W.-m. W. Hwu, "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing," IMPACT Technical Report, 2012.

[43] L.-N. Pouchet, "PolyBench: the Polyhedral Benchmark suite," Ohio State University, [Online]. Available: http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/. [Accessed 6 Jun 2020].

[44] C. Bienia, "Benchmarking Modern Multiprocessors," Ph.D. Thesis, Princeton University, 2011.

[45] P. Luszczek, J. Dongarra1, D. Koester, R. Rabenseifner, B. Lucas, J. Kepner, J. McCalpin, D. Bailey and D. Takahashi, "Introduction to the HPC Challenge Benchmark Suite," ICL Technical Report, 2005.

[46] S. Che, M. Boyer, J. Meng, D. Tarjan, . J. Sheaffer, S.-H. Lee and . K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *IEEE International Symposium on Workload Characterization (IISWC)*, 2009.

[47] J. M. Cebrian, M. Jahre and L. Natvig, "PARVEC: Vectorizing the PARSEC Benchmark Suite," *Computing Frontiers,* 2015 .

[48] J. L. Hennessy, Computer architecture : a quantitative approach, Cambridge, MA : Morgan Kaufmann Publishers, 2019.

[49] C. Kozyrakis and D. Patterson, "Overcoming the limitations of Conventional Vector Processors," in *30th Symposium on Computer Architecture (ISCA)*, 2003.

[50] "gem5 official repository," [Online]. Available: https://gem5.googlesource.com/public/gem5/. [Accessed 15 06 2020].

[51] C. R. Lazo, "McPAT - vector architecture model," [Online]. Available: https://github.com/RALC88/mcpat. [Accessed 1 12 2021].

[52] L. Sheng , H. A. Jung , R. D. Strong, J. B. Brockman, D. M. Tullsen and N. P. Jouppi, "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," in *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009.

[53] E. Arima, Y. Kodama, T. Odajima, M. Tsuji and M. Sato, "Power/Performance/Area Evaluations for Next-Generation HPC Processors using the A64FX Chip," in *IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS)*, 2021 .

[54] V. V. Zyuban and P. M Kogge, "The energy complexity of register files," in *International symposium on Low power electronics and design*, 1998.

[55] C. E. LaForest and G. Steffan, "Efficient multi-ported memories for FPGAs," in *ACM/SIGDA international symposium on Field programmable gate arrays*, 2010.

[56] J. Cebrian, L. Natvig and M. Jahre, "Scalability Analysis of AVX-512 Extensions," *The Journal of supercomputing,* 2019.

[57] Y. Lee, R. Avizienis, . A. Bishara, R. Xia, . D. Lockhart , . C. Batten and K. Asanović, "Exploring the Tradeoffs between Programmability and Efficiency in Data-Parallel Accelerators," in *International Symposium on Computer Architecture (ISCA)*, San Jose, CA, USA, 2012.

[58] M. Stanic, Ó. Palomar , I. Ratkovic, M. Duric, O. S. Unsal, A. Cristal and M. Valero, "VALib and SimpleVector: tools for rapid initial research on vector architectures," in *Computing Frontiers*, 2014.

[59] A. Rico, J. J. and G. G. , "Tutorial: Vector Architecture exploration with gem5," International Conference on Supercomputing (ICS'18), 2018. [Online]. Available: https://www.rico.cat/files/ICS18-gem5-sve-tutorial.pdf.

[60] A. Rico, "Vector Architecture for HPC and ML," Severo Ochoa Research Seminars, BSC, 2018. [Online]. Available: https://www.bsc.es/sites/default/files/public/u2416/arm_sve_seminar_bscupc_arico.pdf.

[61] A. Rico, "Vector Processing with Arm SVE," System on Chip Design Seminar, 2018. [Online]. Available: http://users.ece.utexas.edu/~gerstl/ee382m_f18/lectures/SVE_Seminar_UT_Rico.pdf.

[62] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper and K. Asanovic, "The vector-thread architecture," in *31st Annual International Symposium on Computer Architecture.*, 2004.

[63] D. Dabbelt, C. Schmidt, E. Love, H. Mao, S. Karandikar and K. Asanovic, "Vector Processors for Energy-Efficient Embedded Systems," in *Third ACM International Workshop on Many-core Embedded Systems*, 2016.

[64] G. Gobieski, A. Nagi, N. Serafin, M. Isgenc, N. Beckmann and B. Lucia, "MANIC: A Vector-Dataflow Architecture for Ultra-Low-Power Embedded Systems," in *52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019.

[65] H. Akkary and S. T. Srinivasan, "Checkpoint processing and recovery: towards scalable large instruction window processors," in *36th Annual IEEE/ACM International Symposium on Microarchitecture*, 2003.

[66] A. Gonzalez, J. Gonzalez and M. Valero, "Virtual-physical registers," in *Fourth International Symposium on High-Performance Computer Architecture*, 1998.

[67] T. Monreal, A. Gonzalez, M. Valero, J. Gonzalez and V. Vinals, "Delaying physical register allocation through virtual-physical registers," in *32nd Annual ACM/IEEE International Symposium on Microarchitecture*, 1999.

[68] M. Moudgill, K. Pingali and S. Vassiliadis, "Register renaming and dynamic speculation: an alternative approach," in *26th annual international symposium on Microarchitecture (MICRO 26)*, 1993.

[69] J. Smith and G. S. Sohi, "The microarchitecture of superscalar processors," in *IEEE*, 1995.

[70] D. W. Oehmke, N. L. Binkert, T. Mudge and S. K. Reinhardt, "How to Fake 1000 Registers," in *38th annual IEEE/ACM International Symposium on Microarchitecture (MICRO 38)*, 2005.

[71] J. Zalamea, J. Llosa, E. Ayguade and M. Valero, "Two-level hierarchical register file organization for VLIW processors," in *33rd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-33*, 2000.

[72] M. Sadrosadati, A. Mirhosseini, A. Hajiabadi, S. Borna Ehsani, H. Falahati, H. Sarbazi-Azad, M. Drumond, B. Falsafi, R. Ausavarungnirun and O. Mutlu, "Highly Concurrent Latency-tolerant Register Files for GPUs," *ACM Trans. Comput. Syst.,* vol. 37, p. 36, 2021.

[73] J. Kloosterman, J. Beaumont, D. Anoushe Jamshidi, J. Bailey, T. Mudge and S. Mahlke, "Regless: just-in-time operand staging for GPUs," in *50th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO50*, 2017.