

FEDERAL UNIVERSITY OF SANTA MARIA
TECHNOLOGY CENTER
GRADUATE PROGRAM IN COMPUTER SCIENCE

Michael Guilherme Jordan

**BOOSTING SIMD BENEFITS THROUGH A RUN-TIME
AND ENERGY EFFICIENT DLP DETECTION**

Santa Maria, RS
2019

Michael Guilherme Jordan

**BOOSTING SIMD BENEFITS THROUGH A RUN-TIME AND ENERGY EFFICIENT
DLP DETECTION**

Dissertation presented to the Graduate Program in Computer Science (PPGCC) from the Federal University of Santa Maria (UFSM, RS) as requirement to obtain the degree of Master of Computer Science.

Advisor: Prof. Dr. Mateus Beck Rutzig

Santa Maria, RS

2019

Jordan, Michael Guilherme

Boosting SIMD Benefits through a Run-time and Energy Efficient
DLP Detection / por Michael Guilherme Jordan. – 2019.

92 f.: il.; 30 cm.

Advisor: Mateus Beck Rutzig

Dissertation - Federal University of Santa Maria, Technology Center,
Graduate Program in Computer Science , RS, 2019.

1. DLP. 2. SIMD. 3. Vectorization. 4. ARM NEON. I. Rutzig,
Mateus Beck. II. Boosting SIMD Benefits through a Run-time and
Energy Efficient DLP Detection.

© 2019

All rights reserved to Michael Guilherme Jordan. Reproduction of parts or whole of this work
can only be done by citing the source. E-mail: michael.jordan@ecompu.ufsm.br

Michael Guilherme Jordan

**BOOSTING SIMD BENEFITS THROUGH A RUN-TIME AND ENERGY EFFICIENT
DLP DETECTION**

Dissertation presented to the Graduate Program in Computer Science (PPGCC) from the Federal University of Santa Maria (UFSM, RS) as requirement to obtain the degree of Master of Computer Science.

Approved in February 22, 2019:

Mateus Beck Rutzig, Dr. (UFSM)
(Presidente/Orientador)

Antonio Carlos Schneider Beck Filho, Dr. (UFRGS)

Carlos Henrique Barriquello, Dr. (UFSM)

Santa Maria, RS

2019

ACKNOWLEDGMENT

After an intensive period of two years, today is the day: writing this note of thanks is the finishing touch on my dissertation. It has been a period of intense learning for me, not only in the scientific level, but also on a personal level. I would like to reflect on the people who have supported and helped me so much throughout this period.

I would first like to thank my advisor, Prof. Mateus Beck Rutzig, Ph.D., for your valuable guidance, patience, friendship, excellent cooperation and for all the opportunities I was given to conduct my research and further my dissertation.

I would also like to thank my parents, José Inácio Jordan and Silvia Inês Hoffmann Jordan, and my girlfriend, Tamires Dolores Santos Pereira, for their wise counsel and all the emotional support. You are always there for me.

In addition, I would like to thank my colleagues and friends, from GMICRO Research Group at UFSM, Tiago Knorst and Julio Vicenzi, for their wonderful collaboration. You supported me greatly.

Finally, I would like to thank my friends: Rafael Fão de Moura, Denise Lange Albrecht, Iaçanã I. Weber, Michel Duarte, Juliana Brondani, Giovani Soares, Luana Palma, Deivis Strieder, the RPG Group, the Football Group, among others. Thanks for all the fun times and the support given to each other by deliberating over our problems and findings.

Thank you very much, everyone!

Michael G. Jordan

Santa Maria, RS, January 29, 2019.

ABSTRACT

BOOSTING SIMD BENEFITS THROUGH A RUN-TIME AND ENERGY EFFICIENT DLP DETECTION

AUTHOR: MICHAEL GUILHERME JORDAN

ADVISOR: MATEUS BECK RUTZIG

Multimedia applications have been widely present in embedded devices. Due to their intrinsic nature, such application domain is benefited from Data Level Parallelism (DLP). In order to improve performance-energy tradeoff, current processors enable DLP by coupling SIMD (Single Instruction Multiple Data) engines, such as Intel AVX, ARM NEON and IBM Altivec. Special libraries and compilers are used to support DLP execution on such engines. However, timing overhead on hand coding is inevitable since most software developers are not skilled to extract DLP using unfamiliar libraries. Considering the auto-vectorization through compiler, although improving software productivity, it breaks software compatibility. Besides, both methods are limited to static code analysis, which compromises performance gains.

In this dissertation, we propose a runtime DLP detection named as Dynamic SIMD Assembler (DSA), which transparently identifies vectorizable code regions to execute in the ARM NEON engine. Due to its dynamic fashion, DSA keeps software compatibility and avoids timing overhead on software developing process. Results show that DSA outperforms ARM NEON auto-vectorization compiler by 32% since it applies the partial vectorization of loops and covers wider vectorizable regions, such as Dynamic Range, Sentinel and Conditional Loops. In addition, DSA outperforms hand-vectorized code using ARM library by 26% reducing 45% of energy consumption with no penalties over software development time.

Keywords: DLP. SIMD. Vectorization. ARM NEON.

RESUMO

AUMENTANDO OS BENEFÍCIOS SIMD POR MEIO DE UMA DETECÇÃO DE DLP EM TEMPO DE EXECUÇÃO E ENERGETICAMENTE EFICIENTE

AUTOR: MICHAEL GUILHERME JORDAN

ORIENTADOR: MATEUS BECK RUTZIG

Aplicações multimídia estão amplamente presentes em dispositivos embarcados. Devido à sua natureza intrínseca, este nicho de aplicação é beneficiado pelo Paralelismo a Nível de Dados (DLP). Para melhorar a relação performance-energia, os processadores atuais habilitam o DLP pelo acoplamento de engines SIMD (Single Instruction Multiple Data), como Intel AVX, ARM NEON and IBM Altivec. Bibliotecas e compiladores especiais são usados para suportar a execução de DLP nesses mecanismos. No entanto, a sobrecarga de tempo aplicada a vetorização através de programação manual é inevitável, uma vez que a maioria dos desenvolvedores de software não tem habilidade para extrair o DLP usando bibliotecas desconhecidas. Considerando a auto-vetorização através do uso de compilador, apesar de melhorar a produtividade de software, tal método quebra compatibilidade de software. Além disso, ambos os métodos estão limitados à análise de código estático, o que compromete os ganhos de desempenho.

Nesta dissertação, propomos uma detecção de DLP em tempo de execução chamada Dynamic SIMD Assembler (DSA), que identifica de forma transparente as regiões de código que podem ser vetorizadas para serem executadas no mecanismo ARM NEON. Devido à sua forma dinâmica, a DSA mantém compatibilidade de software e evita a sobrecarga de tempo no processo de desenvolvimento de software. Os resultados mostram que a DSA supera a auto-vetorização através do uso do compilador ARM NEON em 32%, pois aplica a vetorização parcial de loops e abrange mais regiões vetorizáveis, como Loops de Tamanho Dinâmico, Loops Sentinela e Loops Condicionais. Além disso, a DSA supera a programação manual através do uso da biblioteca ARM em 26% reduzindo 45% do consumo de energia sem penalidades em relação ao tempo de desenvolvimento do software.

Palavras-chave: DLP. SIMD. Vetorização. ARM NEON.

LIST OF FIGURES

CONCEPTUAL ANALYSIS

Figure 1 – Scalar Registers vs Vector Registers	19
Figure 2 – VMIPS Overview	20
Figure 3 – ARM A8 Processor Schematic	21
Figure 4 – ARM NEON Engine	22
Figure 5 – Hand-code Programming Overview	24
Figure 6 – Auto-vectorization Compiler Overview	25
Figure 7 – Just-in-time and Traditional Compiler Comparison	27
Figure 8 – Cross-iteration dependency example	28

DYNAMIC SIMD ASSEMBLER

Figure 9 – System Overview	33
Figure 10 – System Functionality Overview	34
Figure 11 – Loop Examples	35
Figure 12 – DSA Execution Flow	36
Figure 13 – Cross-iteration Dependency Prediction	39
Figure 14 – Partial Vectorization Analysis	39
Figure 15 – Count Loop Example	41
Figure 16 – Function Loop Example	41
Figure 17 – Outer Loop Example	42
Figure 18 – DSA Conditional Loop State Machine	44
Figure 19 – Conditional Loop Vectorization Analysis	46
Figure 20 – Conditional Code Loop Analysis Mapping and Data Storage	47
Figure 21 – Conditional Code Loop Execution Mapping and Data Storage	48
Figure 22 – Conditional Code Loop Array Map Logic	49
Figure 23 – Sentinel Loop Cross-iteration Analysis and Execution	52
Figure 24 – Dynamic Range Loop Cross-iteration Analysis	53
Figure 25 – SIMD Instruction Generation Steps	54
Figure 26 – Leftovers	56
Figure 27 – Single Elements Method	56
Figure 28 – Overlapping Method	57
Figure 29 – Larger Arrays Method	58

METHODOLOGY

Figure 30 – DSA Simulation Model	60
Figure 31 – O3CPU - DSA Implementation	61
Figure 32 – DSA Energy Analysis	62

ARTICLE 1

Figure 1 – System Overview	67
Figure 2 – System Functionality Overview	67
Figure 3 – State Machine of DSA	67
Figure 4 – DSA Execution	68
Figure 5 – Loop Detection Stage Behavior	68
Figure 6 – Data Collection Stage Behavior	69
Figure 7 – Data Collection Stage	69
Figure 8 – Dependency Analysis Stage	69
Figure 9 – Example of a Cross-iteration Dependency Prediction Process	70

Figure 10 – Store ID/Execution Stage Behavior	70
Figure 11 – ARM NEON Parallelism	70
Figure 12 – NEON Auto-Vectorization vs. DSA Vectorization Performance	71
ARTICLE 2	
Figure 1 – System Overview	74
Figure 2 – System Functionality Overview	74
Figure 3 – State Machine of DSA	75
Figure 4 – DSA Execution	75
Figure 5 – ARM NEON Parallelism	75
Figure 6 – Example of a Cross-iteration Dependency Prediction Process	76
Figure 7 – Vectorizable, Dynamic Range, Conditional Code and Function Loops	76
Figure 8 – Conditional Loop DSA State Machine	76
Figure 9 – Conditional Code Coverage Stage	77
Figure 10 – Conditional Code Loop Vectorization Analysis	77
Figure 11 – Conditional Code Loop Analysis Mapping and Data Storage	77
Figure 12 – Conditional Code Loop SIMD Execution	78
Figure 13 – DRL Type A, DRL Type B	78
Figure 14 – DRLA Cross-iteration Analysis	78
Figure 15 – DRLB Cross-iteration Analysis and Execution	79
Figure 16 – ARM NEON Compiler AutoVec. vs. ARM NEON Original DSA vs. ARM NEON Extended DSA Performance	80
ARTICLE 3	
Figure 1 – System Overview	83
Figure 2 – Execution Flow	83
Figure 3 – Example of Loops	83
Figure 4 – DSA Analysis and Execution Process	84
Figure 5 – Example of a Cross-iteration Dependency Prediction Process	85
Figure 6 – DSA Partial Vectorization Technique	85
Figure 7 – Percentage of Loop Types in the Selected Applications	86
Figure 8 – Performance Improvements over ARM Original Execution	87
Figure 9 – Energy Savings over ARM Original Execution	87

LIST OF TABLES

INTRODUCTION

Table 1 – Factors that Limit or Prevent the Automatic Loop Vectorization	14
Table 2 – Vectorization Techniques Comparison	28

RELATED WORK

Table 3 – Related Works and Proposed Technique Characteristics	31
----------------------------------------------------------------------	----

METHODOLOGY

Table 4 – Systems Setup	62
-------------------------------	----

ARTICLE 1

Table 1 – Related Works and Proposed Technique Characteristics	67
Table 2 – Systems Setups	70
Table 3 – Area overhead of DSA	71

ARTICLE 2

Table 1 – Related Works and Proposed Technique Characteristics	74
Table 2 – Systems Setups	79
Table 3 – DSA Latency	80

ARTICLE 3

Table 1 – System Setups	86
Table 2 – DSA Detection Latency	86
Table 3 – DSA Energy Consumption	86

LIST OF ABBREVIATIONS AND ACRONYMS

ALU	Arithmetic Logic Unit
CGRA	Coarse Grain Reconfigurable Architecture
CID	Cross-iteration Dependency
CIDP	Cross-iteration Dependency Prediction
CUDA	Compute Unified Device Architecture
CPU	Central Processing Unit
DLP	Data Level Parallelism
DRAM	Dynamic Random Access Memory
DSA	Dynamic SIMD Assembler
FU	Functional Unit
FP	Floating Point
GPP	General Purpose Processor
GPU	Graphic Processing Units
HDL	Hardware Description Language
ID	Identification
ILP	Instruction Level Parallelism
ISA	Instruction Set Architecture
JIT	Just-in-time
LLVM	Low Level Virtual Machine
LPA	Loop-Oriented Pointer Analysis
LRU	Least Recently Used
McPAT	Multicore Power, Area, and Timing
MMX	Multimedia Extensions
NCID	No Cross-Iteration Dependency
ROB	Reorder Buffer
RTL	Register Transfer Level
SIMD	Single Instruction Multiple Data
SM	State Machine
SoC	System-on-Chip
SRP	Samsung Reconfigurable Processor
SSE	Streaming SIMD Extensions
SW	Software

TLP	Thread Level Parallelism
VC	Verification Cache
VHDL	VHSIC Hardware Description Language

SUMMARY

1	INTRODUCTION	14
2	CONCEPTUAL ANALYSIS	17
2.1	ILP, TLP AND DLP EXPLOITATION	17
2.2	SIMD ARCHITECTURES	18
2.2.1	Vector Architectures	18
2.2.2	SIMD Instruction Set Extensions	20
2.2.3	Graphic Processing Units	23
2.3	CODE VECTORIZATION	23
2.3.1	Hand-code Programming Vectorization	24
2.3.2	Auto-vectorization Compiler	24
2.3.3	Just-in-time Vectorization Compilers	26
2.3.4	Critical Analysis	27
2.4	CROSS-ITERATION DEPENDENCIES	28
3	RELATED WORKS	29
3.1	AUTO-VECTORIZATION COMPILER AND VECTOR LIBRARY APPRO- ACHES	29
3.2	ISA/HARDWARE MODIFICATION APPROACHES	30
3.3	JUST-IN-TIME APPROACHES	31
4	DYNAMIC SIMD ASSEMBLER	33
4.1	SYSTEM OVERVIEW	33
4.2	DSA COVERAGE	34
4.3	DSA OVERVIEW	35
4.4	CROSS-ITERATION DEPENDENCY VERIFICATION	38
4.5	PARTIAL VECTORIZATION	39
4.6	DSA - ANALYSIS AND EXECUTION	40
4.6.1	Count Loops	40
4.6.2	Function Loops	41
4.6.3	Inner/Outer Loops	42
4.6.4	Conditional Loops	43
4.6.4.1	<i>Conditional Loops Vectorization</i>	45
4.6.4.2	<i>Conditional Loop SIMD Execution</i>	47
4.6.4.3	<i>Conditional Loop DSA Limitations</i>	48
4.6.5	Sentinel Loops Vectorization	50
4.6.6	Dynamic Range Loop Vectorization	52
4.7	GENERATING SIMD INSTRUCTIONS	53
4.8	DEALING WITH LEFTOVERS	55
4.8.1	Single Elements	56
4.8.2	Overlapping	57
4.8.3	Larger Arrays	57
5	METHODOLOGY	59
5.1	O3CPU PROCESSOR/DSA IMPLEMENTATION	60
5.2	DSA AND O3CPU ENERGY RESULTS	61
5.3	SYSTEMS SETUP	62
6	ARTICLE 1 - IMPROVING SOFTWARE PRODUCTIVITY AND PER- FORMANCE THROUGH A TRANSPARENT SIMD EXECUTION	64

7	ARTICLE 2 - RUNTIME VECTORIZATION OF CONDITIONAL CODE AND DYNAMIC RANGE LOOPS TO ARM NEON ENGINE.....	71
8	ARTICLE 3 - BOOSTING SIMD BENEFITS THROUGH A RUN-TIME AND ENERGY EFFICIENT DLP DETECTION	80
9	DISCUSSION	87
10	CONCLUSION AND FUTURE WORK	89

1 INTRODUCTION

The benefits of the classical transistor shrink may cease in 2021 (COURTLAND, 2016), along with it, the increasing number of multimedia applications has been demanding for more and more performance. In order to provide such performance requirements considering the technological limitation, most architectural solutions attempt to exploit some inherent parallelism available in such applications.

In this scenario, the exploitation of Data Level Parallelism (DLP) has gained increasing relevance since multimedia algorithms are plentiful of Data-Parallel Statements. The DLP can be classified as the capability of performing operations simultaneously over multiple data. Currently, Single Instruction Multiple Data (SIMD) engines are used in market processors to boost multimedia application performance through DLP exploitation. ARM NEON (ARM LIMITED, 2008), Intel SSE/AVX (LOMONT et al., 2011) and IBM AltiVec (DIEFFENDORF et al., 2000) are SIMD engines coupled to general purpose processors (GPP) with the purpose of benefiting from the energy-performance tradeoff on data-parallel applications. The execution of such engines is supported by vector instructions that are applied to vectorizable regions in code. The most significant parcel of vectorizable regions is found in loop statements, which have the property of repeating operations over multiple data. In order to convert loops to vector instructions, SIMD engines apply vectorization techniques such as: auto-vectorization through compiler or hand-coding vectorization. The hand-coding vectorization consists on using low-level functions available on specific libraries to convert vectorizable regions (loops) in SIMD instructions during programming time. Such method requires programming expertise reducing software productivity. The auto-vectorization technique lies on converting vectorizable regions to SIMD instructions during compile time, which does not affect software productivity since no specific library usage is required.

Table 1 shows some factors that inhibit the automatic loop vectorization through compiler. Some limitations can be overcome by combining both techniques (Lines 8 and 10 - Table 1). However, loops with dynamic behavior, such as sentinel loops, dynamic ranged loops and conditional loops, which depend on information generated during execution time, are not efficiently vectorized by such methods since both operate during programming or compile time (statically) (Lines 4, 9 and 12 – Table 1).

Table 1 – Factors that limit or prevent the automatic loop vectorization

	Inhibiting Factor	Extent to which applies
1	No vector access pattern	If variables in a loop lack a vector access pattern, the compiler cannot automatically vectorize the loop.
2	Data dependencies between different iterations of a loop	Where there is a possibility of the use and storage of arrays overlapping on different iterations of a loop, there is a data dependency problem. A loop cannot be safely vectorized if the vector order of operations can change the results, so the compiler leaves the loop in its original form or only partially vectorizes the loop.
3	Memory hierarchy	Performing relatively few arithmetic operations on large data sets retrieved from main memory is limited by the memory bandwidth of the system. Most processors are relatively unbalanced between memory bandwidth and processor capacity This can adversely affect the automatic vectorization process.
4	Iteration count not fixed at start of loop	For automatic vectorization, it is generally best to write simple loops with iterations that are fixed at the start of the loop. If a loop does not have a fixed iteration count, automatic addressing is not possible.
5	Carry-around scalar variables	Carry-around scalar variables are a problem for automatic vectorization because the value computed in one pass of the loop is carried forward into the next pass.
6	Pointer aliasing	Pointer aliasing prevents the use of automatically vectorized code.
7	Indirect addressing	Indirect addressing is not vectorizable because the NEON unit can only deal with vectors stored consecutively in memory.
8	Separating access to different parts of a structure into separate loops	Each part of a structure must be accessed within the same loop for automatic vectorization to occur.
9	Inconsistent length of members within a loop structure	If members of a loop structure are not all the same length, the compiler does not attempt to use vector loads.
10	Calls to non-inline functions	Calls to non-inline functions from within a loop inhibits vectorization. If such functions are to be considered for vectorization, they must be marked with the <code>__inline</code> or <code>__forceinline</code> keywords.
11	Source code without loops	Automatic vectorization involves loop analysis. Without loops, automatic vectorization cannot apply.
12	if and switch statements	Extensive use of if and switch statements in loop can affect the efficiency of automatic vectorization.

Besides both methods present performance limitations due their static fashion, the automatic vectorization and hand-coding programming also require code recompilation, which breaks binary compatibility. To overcome such limitations, Just-in-time (JIT) compiler vectorization approaches emerge. The JIT compiler is capable of exploiting DLP by monitoring vectorizable regions present in a code during runtime. By its dynamic fashion, it is possible to vectorize loops with dynamic behavior and no code recompilation is required, since a JIT compiled code is ISA (Instruction Set Architecture) independent. However, a JIT compiled code generation demands more time than a binary generation, which is produced by an auto-vectorization compiler. In addition, such method requires monitor tasks to detect vectorizable regions, which results in performance penalties.

The solution proposed in this dissertation lies on removing the dependency of static DLP exploitation methods through an engine that is capable of exploiting DLP during execution time. In this way, we created the Dynamic SIMD Assembler (DSA). The DSA can analyze vectorizable regions during runtime and generate SIMD instructions based on such regions. By operating at runtime, DSA increases software productivity, keeps binary compatibility and

embraces parallelism opportunities that have both static and dynamic behavior. Unlike a JIT compiler approach, the DSA implies in no performance penalties, since it detects vectorizable regions parallel to the binary execution by using its own hardware. Considering the DSA system proposed in (JORDAN, 2018), it is capable of outperforming the ARM NEON auto-vectorization technique in 32%. When compared to the ARM NEON library usage approach (Hand-vectorized Code), it can outperform such method by 26%. In addition, the DSA achieves 45% of energy savings over the ARM original execution.

The remaining chapters of this dissertation are based on the Integrated Scientific Articles format, where the formatting imposed by each conference will be respected. Chapter 2 presents the conceptual analysis. Chapter 3 presents the related works. In Chapter 4, the description and implementation of the DSA is addressed. Chapter 5 discusses the methodology used to perform experiments. Scientific articles are presented during chapter 6, 7 and 8, where the order of presentation respects the submission dates of each article. In chapter 9 there is a discussion about the articles. Finally, the conclusion will be presented during chapter 10. It is important to emphasize that all the articles present in this dissertation were submitted and approved in the following conferences: Improving Software Productivity and Performance through a Transparent SIMD Execution (Chapter 6 - SBCCI), Runtime Vectorization of Conditional Code and Dynamic Range Loops to ARM NEON Engine (Chapter 7 - SBESC) and Boosting SIMD Benefits through Run-time and Energy Efficient DLP Detection (Chapter 8 - DATE).

2 CONCEPTUAL ANALYSIS

This chapter provides a quantitative sight of the concepts mentioned in this work. The section 2.1 presents all types of parallelism in which an application may present. Section 2.2 discusses Conventional SIMD Architectures. Section 2.3 presents Code Vectorization techniques applied to SIMD Architectures.

2.1 ILP, TLP AND DLP EXPLOITATION

Pipelining technique (HENESSY; PATTERSON 2011) is able to overlapping the execution of instructions when they are data independent. The potential overlap among instructions is denominated Instruction Level Parallelism (ILP) since the instructions can be executed in parallel in order to accelerate applications. However, the study suggested by (WALL, 1991), proves that there are acceleration bounds related with ILP exploitation. The approach takes five processors, ranging from a best one (perfect branch predictor, perfect memory alias analysis and perfect register renaming) to a worst one (branches always mispredicted, no alias analysis, no register renaming). It is shown that the limits of ILP could be as high as 20 instructions per cycle in the perfect processor, for most of the benchmarks.

To overcome such limit, many micro-architectural techniques like superscalar execution, out-of-order execution, register renaming and speculative execution have been applied considering the hardware perspective (HENESSY; PATTERSON 2011). From the software perspective, compile and programming techniques that involves prediction of data and control flow, loop unrolling and software pipelining (ALLEN et al., 2001) (AHO et. al, 2014) are constantly applied.

Thread Level Parallelism (TLP) emerged as a performance and energy alternative due to the limits on performance gains imposed by ILP exploitation. The TLP Exploitation is achieved when each processor executes threads of the same application over different processors using the same or different data.

According to the Amdahl's Law (AMDAHL, 1967), the serial portions of a program that cannot be executed in parallel limits the speed-up provided by the TLP technique. Plenty researches (HILL et al., 2008) (SUN et al., 2010) reevaluate Amdahl's law premise. To expand the ILP and TLP exploitation the Data Level Parallelism (DLP) emerges.

In contrast to the TLP concept, which divides different operations to execute over the

same or different data concurrently, the DLP is based on running the same operation over a dataset. DLP opportunities are mostly present in application loops, where operations are executed multiple times over vector structures.

To improve applications performance by exploiting DLP, SIMD (Single Instruction Multiple Data) architectures, such as ARM NEON (ARM LIMITED, 2008), Intel SSE/AVX (LOMONT et al., 2011) and IBM AltiVec (DIEFFENDORF et al., 2000), are widely present in market processors. Such SIMD architectures are usually coupled with special vector libraries and compilers that enable the DLP exploitation over applications.

2.2 SIMD ARCHITECTURES

Considering Patterson and Hennessey's approach (HENESSY; PATTERSON 2012), there are three SIMD variations: Vector Architectures, SIMD Instruction Set Extensions and Graphic Processing Units (GPUs).

2.2.1 Vector Architectures

Vector Architectures are based on applying SIMD instructions into a single processor's execution pipeline. Such approaches are easier to understand and compile than other SIMD variations since there are few vector instructions that operate over a fixed data vector length and their vector loads and stores specify regular access pattern, leading to less memory misalignment issues.

However, vector architectures are considered more expensive than the SIMD Extensions, mainly due the cost of sufficient dynamic random access memory (DRAM) bandwidth, given the general reliance on caches to meet memory performance demands on conventional microprocessors.

Vector architectures gather sets of data scattered about memory, place them into large, sequential register files, operate on data in those register files and then store the results back into memory. A single instruction operates over data vectors, which results in dozens of register-to-register operations on independent data elements. These large register files work as controlled buffers to hide memory latency and to take advantage of the large memory bandwidth. Since vector loads and stores are deeply pipelined, the program relies on long memory latency only once per vector load or store and once per element load/store, thus amortizing the latency over

multiple elements.

Figure 1 shows a comparison between Scalar and Vector Registers. As it can be seen, Vector Registers can hold multiple elements of n-bit per register while the Scalar Register holds a single n-bit element per register. In such case, the Scalar Register has 16 scalar registers holding 32-bit element each while the Vector Register has 16 vector registers holding 8 elements, 32-bit per element.

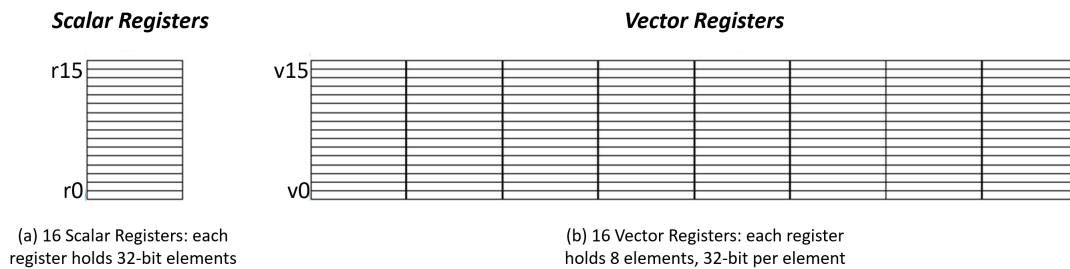


Figure 1 – Scalar Registers vs Vector Registers

Vector Architectures are usually composed of: Vector Functional Units, Vector Registers, Vector Load-Store units and Scalar Registers. Figure 2 presents a Vector architecture example (VMIPS). As can be seen, the VMIPS is composed of:

- **Vector Registers:** VMIPS has eight vector registers holding 64 elements, 64-bit per element. Such registers must provide enough ports to feed all the vector functional units. The VMIPS has 16 read ports and 8 write ports that are connected to the functional unit inputs or outputs through crossbar switches. The large number of ports is one of the reasons of the long memory latency;
- **Vector Functional Units (FUs):** Each unit is fully pipelined, which means that all units are capable of starting a new operation on every clock cycle. A control unit is needed to detect structural and data hazards. The functional units present on the figure are the Floating-point FUs (FP add/subtract, FP multiply, FP divide), Integer FU and Logical FU;
- **Vector load/store unit:** The vector memory unit load or stores a vector to or from memory. The VMIPS vector loads and stores are also fully pipelined. In this way, words can be moved between the vector registers and memory with a bandwidth of one word per clock cycle (after an initial latency). This unit is also capable of handle scalar loads and stores;
- **Scalar Registers:** Such registers provide input data to the vector functional units. They are also responsible for computing addresses to pass to the vector load/store unit. These

are the normal general-purpose and floating-point registers present in the original MIPS. One input of the vector functional units locks scalar values as they are read out of the scalar register file;

- Cross-bar: Responsible for connecting Vector Registers, Functional Units and Load/Store Units.

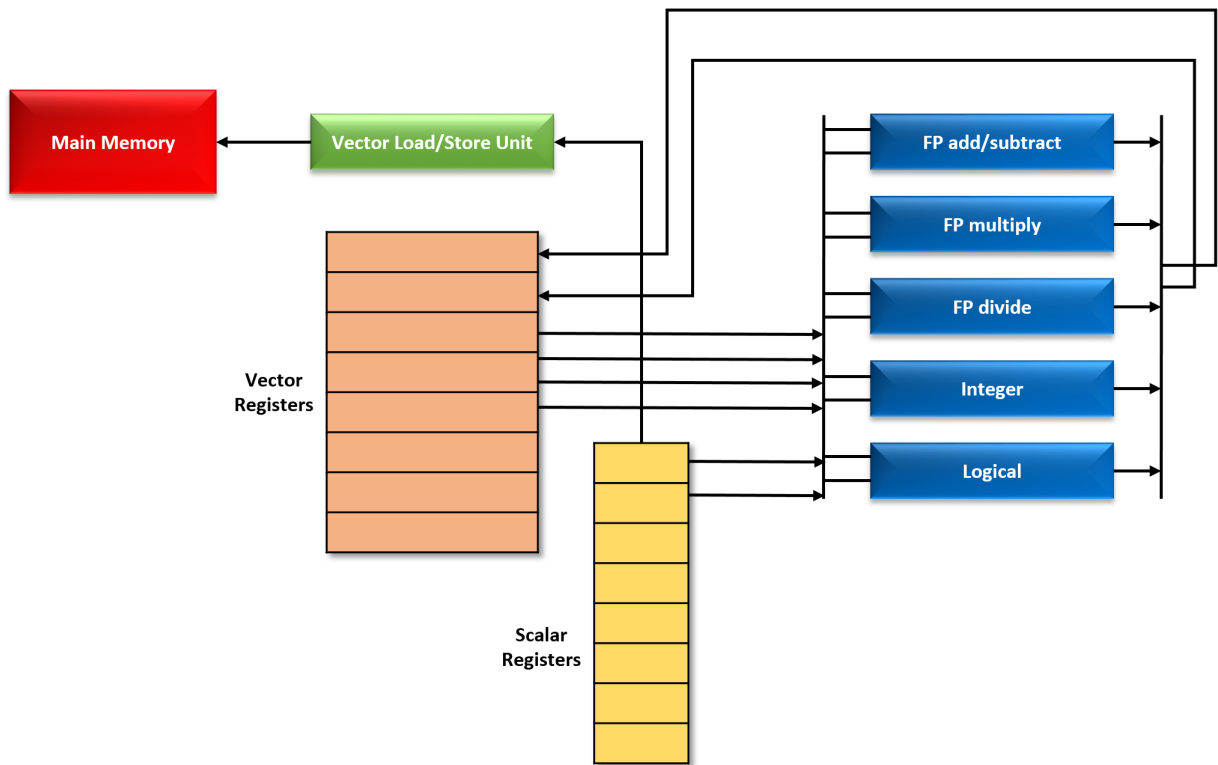


Figure 2 – VMIPS Overview

2.2.2 SIMD Instruction Set Extensions

The SIMD Instruction Set Extensions (SIMD Extensions) are found in most modern instruction set architectures that support multimedia applications. Considering x86 architectures, the SIMD instruction extensions started with the MMX (Multimedia Extensions) in 1996, which were followed by several SSE (Streaming SIMD Extensions) versions in the next decade. Nowadays, such architectures are commonly seen in Intel AVX and ARM NEON instruction set extensions.

SIMD Extensions have been coupled to general-purpose processors since many multimedia applications do not fully explore the vector structure sizes offered by Vector Architectures. By partitioning such structures, a vector engine could perform simultaneous operations

on short vectors, offering more flexible vector operations. A vector structure of 128-bit, could perform parallel operations over sixteen 8-bit operands, eight 16-bit operands, four 32-bit operands or two 64-bit operands. Unlike vector machines with large register files, which can hold up to sixty-four 64-bit elements each of 8 vector registers (VMIPS), SIMD Extensions run over fewer operands and consequently use much smaller register files.

In contrast to vector architectures, SIMD Extensions fix the number of data operands in the opcode leading to the addition of hundreds of instructions. Vector architectures have a vector length register that specifies the number of operands for the current operation. Besides, SIMD Extensions do not offer the more refined addressing modes present in vector architectures. Such particularities make it harder for the compiler to generate SIMD code and increase the difficulty of programming for SIMD extensions.

However, besides such weaknesses, Multimedia SIMD Extensions are prominent due to their smaller cost to add to the standard arithmetic unit. Another advantage of using SIMD Extensions lies on the fact that a lot of memory bandwidth is needed to support a vector architecture, which many computers and embedded devices do not support. Also, the use of short, fixed-length of SIMD extensions makes it easy to introduce flexible instructions that can be applied to new media standards, such as instructions that consume fewer of more operands than vector can produce or instructions that perform permutations.

An example of SIMD Multimedia Extension is the ARM NEON engine. The ARM NEON is a solution for exploiting Data Level Parallelism on embedded devices. It works as a co-processor, where vector statements (NEON statements) are executed in their own pipeline.

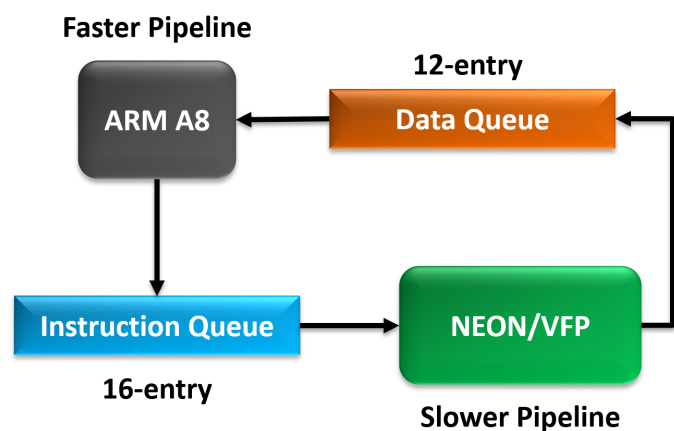


Figure 3 – ARM A8 Processor Schematic

Figure 3 presents a simple ARM A8 Processor schematic. As it can be seen, such architecture operates through the use of Instruction and Data queues to perform vector instructions

in the ARM NEON/VFP Engine. ARM A8 has a faster pipeline than the NEON Engine, which means that scalar instructions and vector instructions are executed over independent pipelines. Some ARM A8/NEON Engine aspects are:

- NEON instructions execute in their own 10-stage pipeline;
- ARM can dispatch 2 NEON instruction per cycle to the Instruction Queue;
- 16-entry instruction queue holds NEON instructions until they can enter the NEON pipeline;
- 12-entry data queue holds operations results until they can be received by the ARM A8 general-processor;
- The ARM general-processor will not stall until the NEON queue fills or some data hazard between scalar and vector instruction is found. That means that the ARM general-processor can dispatch several NEON instructions while performing other work until the NEON finishes its execution.

Figure 4 shows the different degrees of parallelism that can be obtained through the 128-bit wide NEON Engine depending on the type of data involved in the SIMD instruction. As it can be seen, we can perform up to 16 operations simultaneously with 8-bit integer data (.I8). With 32-bit float data (.F32), only 4 operations can be performed in parallel.

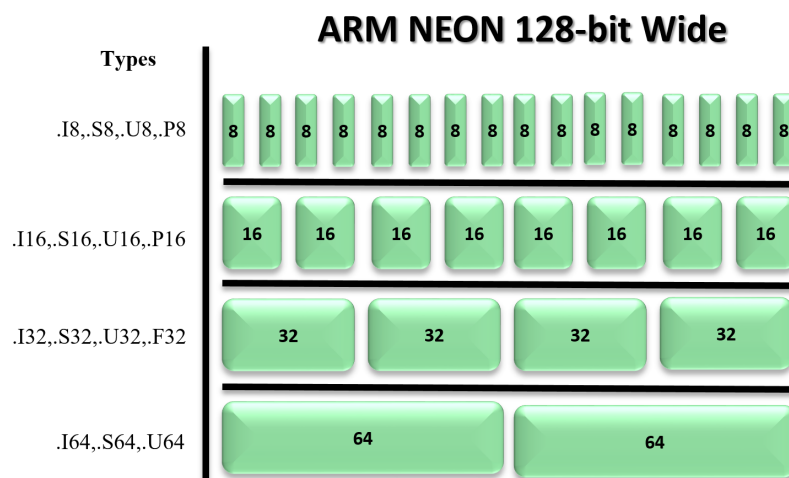


Figure 4 – ARM NEON Engine

2.2.3 Graphic Processing Units

The GPU offers higher performance potential on exploiting thread level parallelism than traditional multicore computers since it is composed of thousands of processing elements. In 2006, NVIDIA created the Compute Unified Device Architecture (CUDA), a parallel processing technology that enables acceleration in general-purpose computing performance. With the specific programming language CUDA C [NVIDIA 2011], it is possible to control such processing elements and, in this way, it is possible to explore not only graphical applications but also to optimize general-purpose applications with high data-level parallelism.

Like vector architectures, GPUs work well with DLP issues. Both styles have gather-scatter data transfer and mask registers, and GPU processors have even more registers than do vector processors. In addition, both vector architectures and GPUs do not abstract hardware complexity, which demands high programming expertise to generate efficient code, affecting software productivity. Unlike most vector architectures, GPUs also rely on multithreading within a single multithread SIMD processor to hide memory latency. Besides, the GPU has many simple functional units and no scalar processor, opposed to a few deeply pipelined units like a vector processor.

2.3 CODE VECTORIZATION

In 1970 decade (RUSSEL, 1977), the first computer to successfully implement a vector processor emerged. Since then, with the multimedia applications arise, vector processors and SIMD engines are present in most computers and embedded devices. Such processors have their potential exploited by Code Vectorization techniques. Code Vectorization is an optimization technique that exploits DLP through the use of SIMD instructions. Most DLP opportunities are present in loops which operate the same instruction over multiple data. Depending the number of data elements that can be merged into one vector operation, an application can reach high acceleration. To enable code vectorization, SIMD engines adopt three common methods: hand-code programming vectorization, auto-vectorization through compiler and Just-in-time (JIT) Compiler vectorization.

2.3.1 Hand-code Programming Vectorization

An efficient code vectorization is challenging. Hand-code programming, where the programmer directly indicates which SIMD instruction to use, demands huge effort from the programmer since most vectorization libraries are not portable when targeting different Instruction Set Architectures (ISAs).

Figure 5 presents a Matrix Sum hand-code algorithm adapted to the NEON, SSE and AltiVec extensions. As it can be seen, such vectorization libraries do not abstract hardware complexity, requiring a high programming expertise. In the NEON case, the 64-bit vector registers can place two float elements while the 128-bit vector registers of the AltiVec and SSE approaches can vectorize four float elements in parallel. Besides, each approach has its own functions to enable the use of each SIMD Engine. To solve such complexity, auto-vectorization techniques have been added to compilers in order to perform Code Vectorization automatically.

<pre>float sum=0; for (i=0; i<n; i++) sum += a[i+2]; }</pre> <p style="text-align: center;">(a) Scalar</p>	<pre>float sum; v2float vsum={0,0}; for (i=0; i<n; i+=2) { // vsum += a[i+2:i+3]; vx = vld1_f32 (&a[i+2]); vsum = vadd_f32(vx,vsum); } sum = finalize_reduc(vsum);</pre> <p style="text-align: center;">(b) NEON (64-bit Wide)</p>
<pre>float sum; v4float vsum={0,0,0,0}; for (i=0; i<n; i+=4) { // vsum += a[i+2:i+5]; vx = movdqu (&a[i+2]); vsum = vadd(vx,vsum); } sum = finalize_reduc(vsum);</pre> <p style="text-align: center;">(c) SSE (128-bit Wide)</p>	<pre>float sum; v4float vsum={0,0,0,0}; vm = get_permute_vect (&a[2]); va = lvx (&a[0]); for (i=0; i<n; i+=4) { vb = lvx (&a[i+4]); vx = vperm (va,vb,vm); vsum = vadd(vx,vsum); va = vb; } sum = finalize_reduc(vsum);</pre> <p style="text-align: center;">(d) AltiVec (128-bit Wide)</p>

Figure 5 – Hand-code Programming Overview

2.3.2 Auto-vectorization Compiler

The Auto-vectorization Compiler is responsible for vectorizing loops during compile time. Figure 6 illustrates how the Auto-vectorization Compiler works. As it can be seen, a non-adapted code is compiled with an auto-vectorization compiler. The compilation results in

an assembly code containing SIMD instructions (vectorizable instructions). However, although improving software productivity, such method may reduce performance when compared with the hand-code programming approach.

<pre> 0: int main() 1: { 2: int a[256], b[256]; 3: int i; 4: 5: for (i = 0; i<256; i++) 6: { 7: a[i] = b[i] >> 8; 8: } 9: }</pre>	<pre> 0: .L2 1: add r2, r0, r3 2: fldd d16, [r2, #0] 3: vmov.32 r2, d16[0] 4: vmov.32 r1, d16[1] 5: mov r2, r2, asr #8 6: str r2, [r5, r3] 7: add r2, r5, r3 8: add r3, r3, #8 9: mov r1, r1, asr #8 10: cmp r3, #1024 11: str r1, [r2, #4] 12: bne .L2</pre>
(a) Original Code	(b) Assembly Code Generated by ARM NEON Auto-vectorization Compiler

Figure 6 – Auto-vectorization Compiler Overview

(MITRA et. al, 2013) compares the performance of the auto-vectorization compiler over the hand-code programming approach in several applications. Such comparison considers 10 different SoC scenarios. The results present speed-ups of 1.05 to 13.88 and 1.34 to 5.54 for using hand optimized SIMD intrinsic functions rather than gcc compiler auto-vectorization for ARM and Intel platforms respectively. There are several factors which limits the compiler auto-vectorization performance (MELNIK, 2010) (POHL et. al, 2018) (SHIN, 2007).

Some factors that inhibit the NEON Compiler auto-vectorization are (ARM Limited, 2017):

- Data dependencies between different iterations of a loop - Where there is a possibility of the use and storage of arrays overlapping on different iterations of a loop, there is a data dependency problem. A loop cannot be safely vectorized if the vector order of operations can change the results, so the compiler leaves the loop in its original form or only partially vectorizes the loop;
- Indirect addressing - Indirect addressing is not vectorizable because the NEON unit can only deal with vectors stored consecutively in memory;
- if and switch statements - Extensive and complex use of if and switch statements can affect the efficiency of automatic vectorization;

- Iteration count not fixed at start of loop - For automatic vectorization, it is generally best to write simple loops with iterations that are fixed at the start of the loop. If a loop does not have a fixed iteration count, automatic addressing is not possible;
- Memory hierarchy - Performing relatively few arithmetic operations on large data sets retrieved from main memory is limited by the memory bandwidth of the system. Most processors are relatively unbalanced between memory bandwidth and processor capacity. This can adversely affect the automatic vectorization process;
- Calls to non-inline functions - Calls to non-inline functions from within a loop inhibits vectorization. If such functions are to be considered for vectorization, they must be marked with the `__inline` or `__forceinline` keywords;
- Inconsistent length of members within a structure - If members of a structure are not all the same length, the compiler does not attempt to use vector loads;
- Pointer aliasing - Indirect addressing is not vectorizable because the NEON unit can only deal with vectors stored consecutively in memory;
- Source code without loops - Automatic vectorization involves loop analysis. Without loops, automatic vectorization cannot apply;
- Target processor - The target processor (`-cpu`) must have NEON capability if NEON instructions are to be generated. For example, Cortex-A7, Cortex-A8, Cortex-A9, or Cortex-A15.

2.3.3 Just-in-time Vectorization Compilers

Another issue that limits Hand-code Programming and Automatic-vectorization Compiler approaches lies in the fact that both of them operate statically, which means that loops with dynamic behavior or complex control flow are not efficiently vectorized. In some ISAs, such loops can not be vectorized statically.

To solve such problem, Just-in-time (JIT) compilers have emerged (NUZMAN et al., 2011) (NAKAMURA; SATOSHI; SHUICHI; 2011). Unlike a traditional compiler that produces an object file statically, a Just-in-time compiler operates over a code dynamically, which can provide portability among different ISAs. Figure 7 illustrates the differences between a Just-in-time and a Traditional Compiler operation flow.

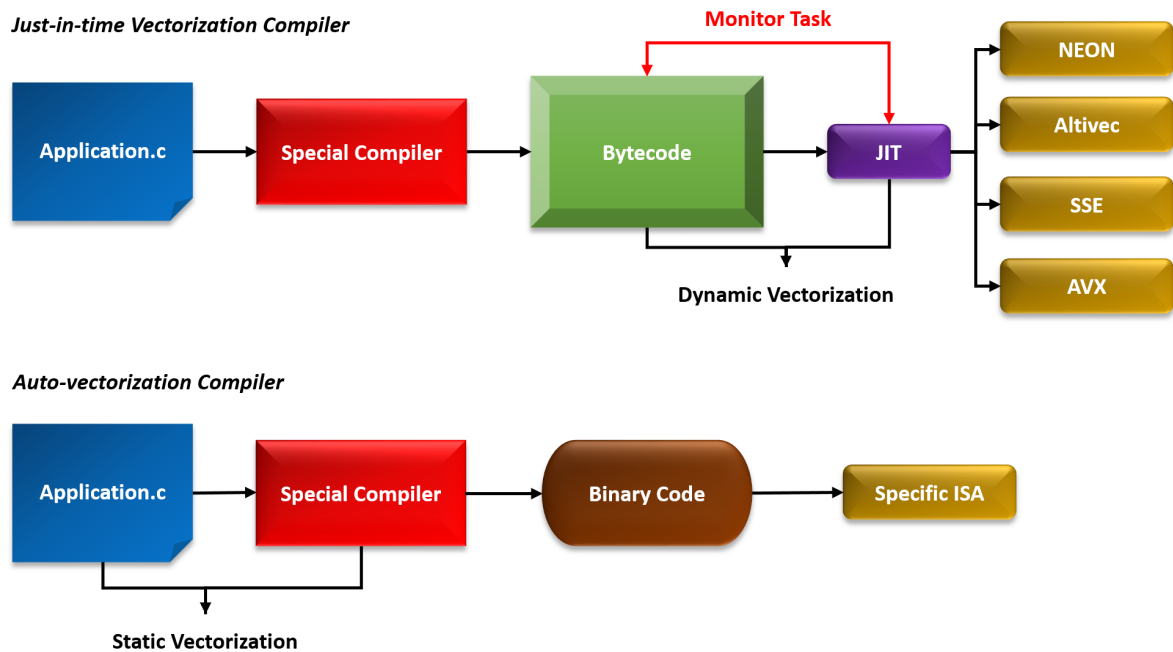


Figure 7 – Just-in-time and Traditional Compiler Comparison

As can be seen, the traditional compiler vectorizes an application statically and generates a vectorized binary to a specific ISA. The JIT compiler operates over a JIT compiled code dynamically, enabling vectorized binary generation to several ISAs. However, an application takes longer time to be compiled. Besides, to detect vectorization possibilities during runtime, a JIT compiler concurrently runs a monitor task, which cause processing demands. Such processing needs may be unacceptable in embedded devices.

2.3.4 Critical Analysis

Table 2 compares the three cited code vectorization methods: Auto-Vectorization Compiler, Hand-Code Programming and Just-in-time Compiler. As can be seen, unlike the Just-in-time approach, which operates dynamically, the Auto-Vectorization Compiler and the Hand-Code Programming methods operate statically, which inhibits efficient vectorization of dynamic vectorizable regions. Besides, the Just-in-time technique operates over a portable code, which means that no code recompilation is needed. The Auto-Vectorization Compiler and the Hand-Code Programming methods need code recompilation since they operate over specific ISA binary, which breaks binary compatibility.

While the Hand-Code programming requires huge efforts from the programmer to extract an efficient code vectorization, the Auto-Vectorization compiler focuses on increasing soft-

ware productivity by automatically extracting vectorizable regions. In such aspect, the Just-in-time approach can increase both software productivity and performance (by its dynamic nature). However, the code generation latency is greater than a specific ISA binary. Besides, the Just-in-time vectorization requires a monitor task, which requires processing demands from the system.

Table 2 – Vectorization Techniques Comparison

Technique	Code Recompilation	Software Productivity	Vectorization	Performance Penalty
Hand-Code Programming	Yes	Affected	Static	No
Auto-Vectorization Compiler	Yes	Not Affected	Static	No
Just-in-time Compiler	No	Not Affected	Dynamic	Monitor Task

2.4 CROSS-ITERATION DEPENDENCIES

A cross-iteration dependency is found in loops that require data generated from other iterations within the same loop. Such property is the main factor that inhibits a loop vectorization, limiting the DLP exploitation in a code. Figure 8 compares a loop containing no cross-iteration dependency (8.a) and a loop with cross-iteration dependency (8.b). As can be seen, the loop presented in 8.a does not need any data generated in previous loop iterations, which means that $v[0]$ can be executed independently of $v[1]$ and so on. In that case, the loop vectorization is possible. However, the loop presented in 8.b depends on data generated in previous loop iterations, which means that $v[i]$ can only be executed when $v[i-1]$ result is ready. Consequently the loop 8.b can not be vectorized.

```

for (i=0;i<40;i++){
  v[i] = a[i] + b[i];
}

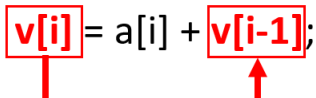
```

a. No Cross-iteration dependency loop

```

for (i=0;i<40;i++){
  v[i] = a[i] + v[i-1];
}

```



b. Cross-iteration dependency loop

Figure 8 – Cross-iteration dependency example

3 RELATED WORKS

The SIMD vectorization is widely used in several emerging market platforms, such as the Intel SSE, IBM Altivec, and ARM NEON architectures. In the academic field, several researches are exploiting Data Level Parallelism (DLP) to achieve performance improvements and energy savings.

3.1 AUTO-VECTORIZATION COMPILER AND VECTOR LIBRARY APPROACHES

(SUI et al., 2016) improves the LLVM (Low Level Virtual Machine) compiler [3] infrastructure to explore vectorization opportunities by developing a more precise Loop-Oriented Pointer Analysis (LPA) for Automatic SIMD Vectorization. This approach is able to detect more basic blocks achieving performance improvements from 2.95% to 7.23%. However, such an approach uses an auto-vectorization technique, which means that loops containing dynamic behavior are not vectorized.

(ZHOU; XUE 2016) presents the Loop-Mix compiler, also implemented in the LLVM compiler. Loop-Mix vectorizes loops regarding the data reorganization overhead caused between mixed SIMD parallelism (inter-loops and intra-loops). The technique outperforms the Loop-ILV [5] by 36%. Since the work is also implemented in the LLVM compiler, the binary compatibility is compromised, code recompilation is required and dynamic behavior loops are not covered.

(NUZMAN; IRA; AYAL 2006) evaluates and applies a compiler outer loop vectorization technique focusing on properties of modern SIMD architectures (Loop-ILV). It shows that even though current optimizing compilers do not apply outer loop vectorization, they can provide significant performance improvements over innermost loop vectorization. Loop-ILV achieves performance improvements of 3.13 and 2.77 when coupled to a Cell BE SPU and PowerPC970, respectively. Similar to our proposal, the authors focused on vectorizing both innermost and outer loops but it relies on compiler support.

Being aware that most research focuses on vectorizing loops, (TIAN et al., 2012) presented a set of new C/C++ high-level extensions for SIMD programming capable of automatic translating both functions and loops. Significant speedups (from 3.07x to 4.69x) are achieved when these optimizations are applied. Similar to aforementioned related works, it relies

on specific compiler and library to achieve performance improvements, which breaks binary compatibility and affects SW productivity.

(BRAMAS, 2017) proposes Inastemp, a lightweight opensource C++ library that provides portable SIMD-Vectorization. This approach has the same efficiency as computing for a specific architecture, providing vector instructions that can be used to develop hardware-independent computational kernels. These computational kernels are portable across compilers. Inastemp covers SSE, AVX, AVX512 and ALTIVEC/VMX instructions. While such technique improves binary portability, it compromises software productivity since code must be adapted with the suggested library and requires code recompilation. In addition, no performance gains are shown by using such technique.

ARM NEON (ARM Limited, 2008) is introduced in the ARMv6 architecture. The NEON auto-vectorization compiler generates vectorizable code by instantiating SIMD instructions. Despite the advantages of autovectorization, the static code exploitation limits the performance gains since it is difficult to identify vectorizable regions of conditional statements, function calls or even loops that contain codes between inner-loops and outer-loops. To overcome such issues, another strategy offered by the ARM to explore the NEON engine is the use of ARM NEON library, which transfer the vectorization task responsibility to the SW developer which affects SW productivity.

3.2 ISA/HARDWARE MODIFICATION APPROACHES

Liquid SIMD (CLARK et al., 2007) separates the SIMD accelerator implementation from the ISA, providing an abstraction to overcome ISA migration problems. By the use of a special compiler the Liquid SIMD translates SIMD instructions into a virtualized representation using the processor's baseline instruction set. The compiler isolates portions of the application into dataflows and converts them into architecture-specific SIMD instructions. However, the work needs compiler changes and code recompilation, which brakes binary compatibility.

(BAGHSORKHI; NALINI; YOUFENG; 2016) proposes FlexVec architecture that combines a novel partial vector code generation technique with new vector instructions to dynamically adjust vector length for loop statements affected by runtime cross-iteration dependencies. FlexVec vectorization coupled to the Intel AVX-512 ISA shows a Geomean performance improvement from 9% to 11%. Although it is able to perform optimizations over loops with cross-iteration dependencies, the method breaks binary compatibility, since it is necessary a

specific ISA adjustment and also relies on a particular compiler and library development.

(CHANG; WONYONG 2008), employed a unique memory access hardware, solving the non-aligned and irregular data memory access operations to improve the performance of a SIMD processor based on ARMv4 architecture. In addition, it develops an auto-vectorization compiler, which utilizes the proposed hardware. By applying such technique, the number of vectorized loops increases 50%, which provides 77% of performance improvement in the MPEG2 encoder execution.

Besides the research above, many studies are also focused on applying reconfigurable architectures, since besides exploiting ILP, they are also capable of exploring DLP. A reconfigurable architecture, named as Samsung reconfigurable processor (SRP), is developed for digital signal processing (KIM et al., 2012). The SRP architecture is designed to handle mobile multimedia applications efficiently. It uses a CGRA to vectorize innermost loops by using a conventional C/C++ programming model to annotate the code. Despite the huge chip area required to the CGRA, the SRP relies on compiler, library and ISA modifications. In addition, it requires a design-time step to create CGRA configurations for each application which reduces, even more, the binary compatibility and SW productivity.

3.3 JUST-IN-TIME APPROACHES

Vapor SIMD (NUZMAN et al., 2011) provides a just-in-time (JIT) compilation solution for targeting different SIMD architectures. The Vector SIMD can combine static and dynamic infrastructure for vectorization, focusing on the ability to revert efficiently and seamlessly to generate scalar instructions when the JIT compiler or target platform do not support SIMD capabilities and vector instructions when SIMD instructions are supported. Selftrans (NAKAMURA; SATOSHI; SHUICHI; 2011) is capable of vectorizing automatically the x86 binary machine code without requiring its source code, translating it into a binary code that uses SIMD units dynamically.

Both solutions solve the problem of software productivity. However, JIT approaches require a separate translation process to share the CPU (Monitor Task), which may be unacceptable in embedded systems.

Table 3 – Related Works and Proposed Technique Characteristics

Work	Code Recompilation	Library Development Support	ISA Modification	SW Productivity	Binary Compatibility	Dynamic Behavior Loops Support	JIT Compiler
LPA	Yes	No	No	Not Affected	No	No	No
Loop-Mix	Yes	No	No	Not Affected	No	No	No
Loop-ILV	Yes	No	No	Not Affected	No	No	No
Tian et al., 2012	Yes	Yes	No	Affected	No	No	No
Inastemp	Yes	Yes	No	Affected	No	No	No
ARM NEON	Yes	Yes	No	Affected	No	No	No
Liquid SIMD	Yes	No	No	Not Affected	No	Yes	No
FlexVec	Yes	Yes	Yes	Affected	No	No	No
Chang; Wonyong 2008	Yes	No	Yes	Not Affected	No	No	No
SRP	Yes	Yes	Yes	Affected	No	No	No
Vapor SIMD	No	No	No	Not Affected	Yes	Yes	Yes
Selftrans	No	No	No	Not Affected	Yes	Yes	Yes
DSA	No	No	No	Not Affected	Yes	Yes	No

Table 3 compares all the aforementioned works with the proposed approach. As it can be seen, binary and software compatibility are not prioritized in most designs since they employ ISA modification or specific libraries. JIT compiler approaches, even prioritizing software compatibility and bringing dynamic SIMD exploitation, result in processing demands from the CPU, since they need a separate process to dynamically translate code regions.

Our work proposes a transparent Dynamic SIMD Assembler that is capable of building SIMD instructions at runtime. The proposed approach coupled to the ARM NEON engine provides:

- higher performance than ARM auto-vectorization method with binary compatibility since is not necessary to recompile the source code;
- SW productivity by avoiding the use of the ARM library in the code development lifecycle to take advantage of the NEON engine processing capabilities;
- no system overhead during DSA analysis since the DSA operates in parallel with the ARM processor;
- flexible dynamic vectorization techniques that can be applied to any ISA.

4 DYNAMIC SIMD ASSEMBLER

4.1 SYSTEM OVERVIEW

Figure 9 shows the overview of the Dynamic SIMD Assembler (DSA). The DSA is coupled to the O3CPU processor (more details in section 4.9), which uses the ARMv7-A Instruction Set Architecture (ISA). As can be seen, the DSA consists of a SIMD instruction detection and generation logic and two caches (DSA Cache and Verification Cache). The DSA Cache is responsible for storing information of previously verified vectorizable loops, such as the identification of such loops and SIMD statements generated for these loops. Verification Cache (V-Cache) stores the data memory addresses accessed by the loops (more details in section 4.4).

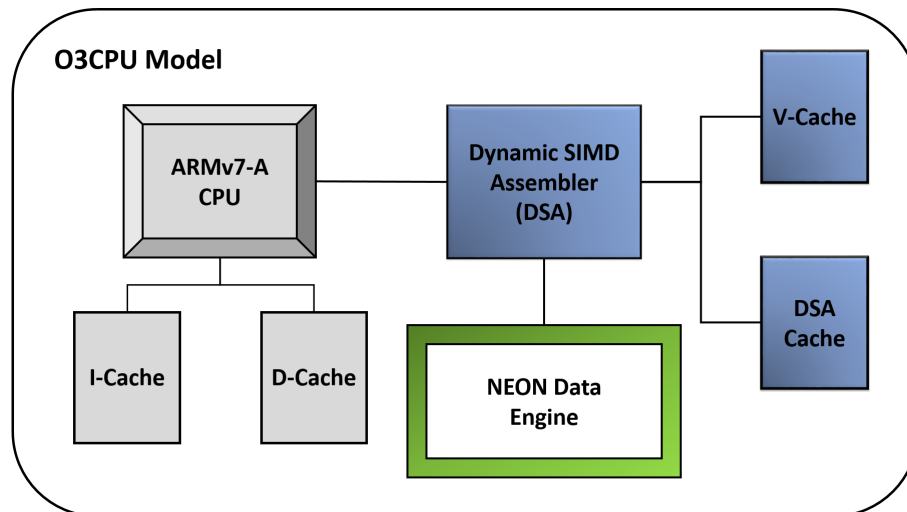


Figure 9 – System Overview

Figure 10 presents an overview about the DSA functionality. In the first scenario (Scenario 1 - DSA Loop Analysis), the DSA and the ARMv7-A processor operate in parallel. While the ARM processor executes the incoming instructions, the DSA works in probing mode, looking for a vectorizable loop to build SIMD instructions. During this step, NEON Engine remains disable. If the DSA detects a vectorizable loop, the second scenario is activated (Scenario 2 - DSA Loop Execution). In this scenario, the DSA disables the ARMv7-A processor and activates the NEON Data Engine to execute the built-in SIMD (Vectorized Instructions) statements. It is important to notice that the DSA runs in parallel with the ARMv7-A processor, which means that the critical path of the processor is not affected by the DSA.

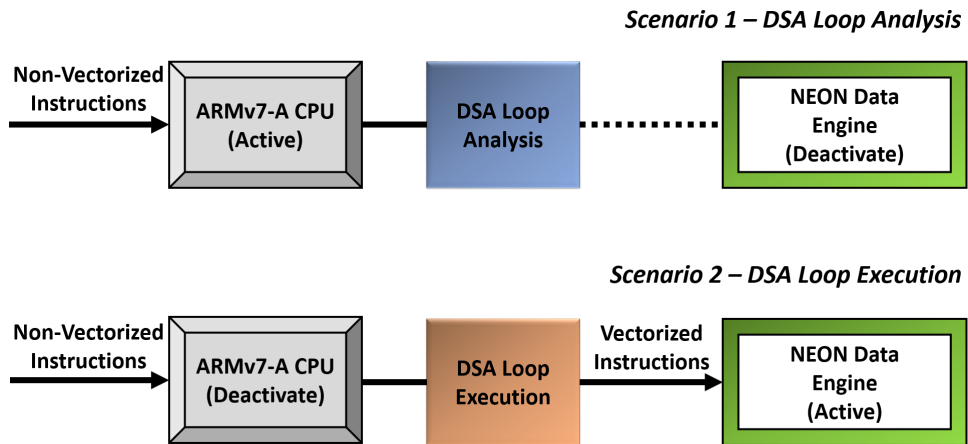


Figure 10 – System Functionality Overview

4.2 DSA COVERAGE

Figure 11 presents examples of loops that can be vectorized by the DSA (Count Loop (A), Dynamic Range Loop (B), Conditional Loop (C) and Function Loop (D)). As can be seen, pseudocode (A) presents a simple vectorizable loop which both compiler and DSA are able to vectorize. The pseudocode (B) has a Dynamic Range Loop, where the size of the loop is determined by an input or computed at runtime. The pseudocode (C) has a loop that contains conditional statements and its execution is also determined during runtime. The same analysis can be performed on the pseudocode (D), which has a loop containing a function call that depends on a variable computed during execution time. In this way, the pseudocodes (B), (C) and (D) can not be efficiently vectorized by compiler auto-vectorization methods since they depend on data computed during execution time. However, because the DSA (Dynamic SIMD Assembler) analyzes the application code during runtime, it is able to efficiently vectorize all the aforementioned situations.

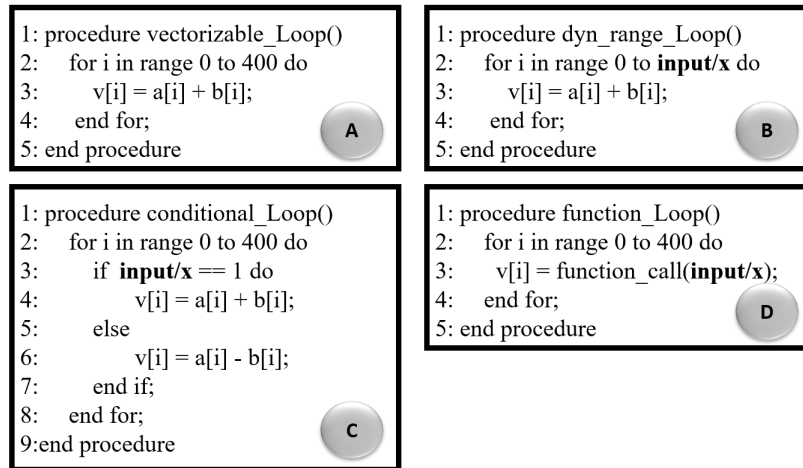


Figure 11 – Loop Examples

The DSA covers the vectorization of: Count Loops, Function Loops, Outer and Inner Loops, Dynamic Range Loops and Sentinel Loops. In addition, DSA also supports partial vectorization of loops with cross-iteration dependencies (further discussed at section 4.4).

It is important to notice that every loop type coverage implemented in the DSA was selected based on the following works: (NAKAMURA; SATOSHI; SHUICHI; 2011), (NUZMAN; ROSEN; ZAKS; 2006), (TIAN et al., 2012), (NUZMAN; ZAKS 2008) and (WU; EICHENBERGER; WANG; 2005). The partial vectorization implementation was based on the work: (BAGHSORKHI; VASUDEVAN; WU; 2016). Other state-of-the-art loop type vectorization such as complex control flow loops or loops operating over misaligned data (CHANG; SUNG 2008) are considered in our future work.

4.3 DSA OVERVIEW

The DSA detection process is based on a State Machine (SM) composed of six states: Loop Detection, Data Collection, Dependency Analysis, Store ID/Execution, Mapping and Speculative Execution. Each of these stages is activated in different iterations of the loop.

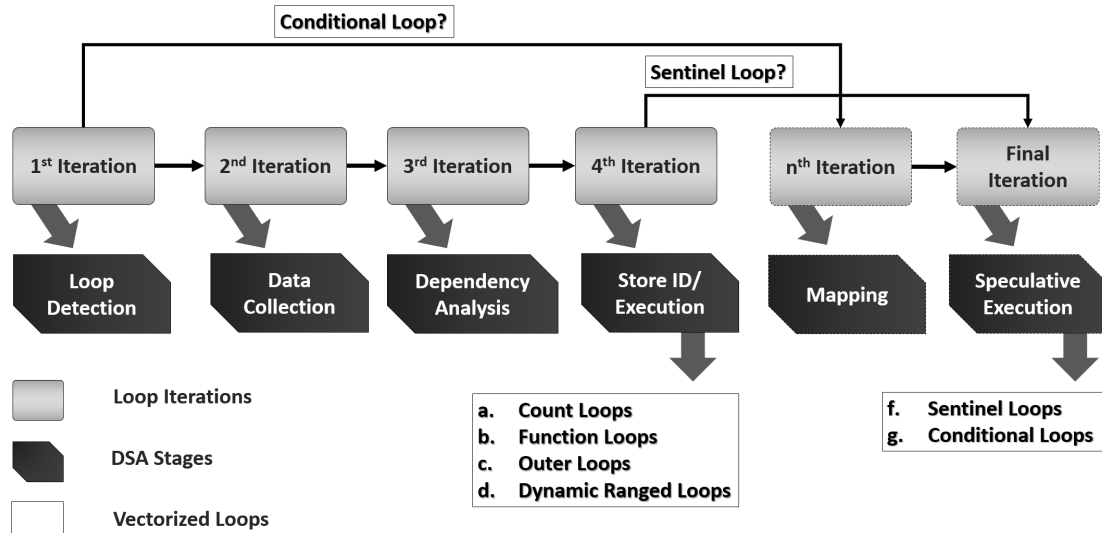


Figure 12 – DSA Execution Flow

As can be seen in figure 12, the Loop Detection stage is activated at the end of the first iteration. The Loop Detection stage is responsible for:

- detecting the presence of a loop;
- detecting the presence of conditional code and functions present within the loop;
- verifying multiple loop layers (inner-most loop and outer loops);
- accessing the DSA Cache and check if the current loop has been previously verified as vectorizable.

The Data Collection stage is triggered during the second iteration of the loop. This stage is responsible for:

- evaluating the loop range (number of iterations), vectorizable instructions and their operands;
- storing the addresses of data memory accesses in the Verification Cache;
- verifying the presence of a Sentinel Loop.

The Dependency Analysis stage is triggered in the third loop iteration. This stage is responsible for:

- analyzing the cross-iteration dependency (dependencies between two or more iterations in the same loop statement).

The Store ID/Execution stage is triggered in the fourth loop iteration. This stage is responsible for:

- concluding the vectorization of Count loops, Functions loops, Outer/Inner loops, Dynamic Range Loops and Partial Loops;
- generating and saving the loop identification (ID) in the DSA Cache;
- allowing the partial vectorization of loops with cross-iteration dependencies (when possible);
- building SIMD instructions and activating the execution of the ARM NEON engine;
- applying Leftover techniques if necessary (further discussed in section 4.8).

The Mapping stage is only activated for Conditional loops. This stage is responsible for:

- evaluating the loop range (number of iterations);
- mapping the executed conditional code statements;
- verifying cross-iteration dependencies between iterations within each condition;
- evaluating the vectorizable instructions and their operands within each condition;
- building SIMD instructions and activating the execution of ARM NEON Engine;
- applying Leftover techniques if necessary (further discussed in section 4.8).

The Speculative Execution stage is only enabled for Conditional Loops and Sentinel Loops. This stage is responsible for:

- selecting data generated during SIMD execution at the end of the Loop (Conditional Loops and Sentinel Loops);
- storing the current range for Sentinel Loops (Speculative Range);
- storing mapped conditions of the Conditional Loop for further executions in the DSA Cache;
- generating and saving the loop identification (ID) in the DSA Cache.

4.4 CROSS-ITERATION DEPENDENCY VERIFICATION

At the memory access point of view, a cross-iteration dependency exists when the same data memory address is accessed in different loop iteration. The DSA cross-iteration analysis starts in the 2^{nd} loop iteration, where the addresses of data memory accesses are saved in the Verification Cache (VC). Even having the memory addresses in the VC and comparing them to the memory addresses performed on every iteration, one cannot discard cross-iteration dependencies in future iterations. Assuming such situation, we have implemented the *Cross-iteration Dependency Prediction*.

The equations below describe the steps of the prediction process, where $M_{Read[2]}$ and $M_{Read[3]}$ are the memory addresses accessed by a *MemRead (load)* instruction in the second and third loop iterations, respectively. $M_{Read[lastiteration]}$ is the memory address accessed by a load instruction in the last executed iteration (Equation 4.4), x is the interval between $M_{Read[2]}$ and $M_{Read[lastiteration]}$ (Equation 4.1), $M_{Write[2]}$ is the memory address accessed by a *Mem-Write (store)* instruction in the second iteration (Equations 4.2 and 4.3), M_{Range} is the memory address range between the $M_{Read[2]}$ and $M_{Read[3]}$ (Equation 4.5), *CID* means Cross-Iteration Dependency and *NCID* means No Cross-Iteration Dependency.

$$M_{Read[3]} \leq x \leq M_{Read[lastIteration]} \quad (4.1)$$

$$M_{Write[2]} \in x \rightarrow CID \quad (4.2)$$

$$M_{Write[2]} \notin x \rightarrow NCID \quad (4.3)$$

$$M_{Read[lastIteration]} = M_{Read[2]} + (M_{Gap} * (lastIteration - 2)) \quad (4.4)$$

$$M_{Gap} = |M_{Read[3]} - M_{Read[2]}| \quad (4.5)$$

Considering the equations above, if the $M_{Write[2]}$ is within the memory address range of $M_{Read[3]}$ and $M_{Read[lastiteration]}$ (Equation 4.2), the loop would have a cross-iteration dependency since the load instruction of a future loop iteration could perform a memory access in the same memory address of the store instruction executed in the second loop iteration. The memory address of the load instruction executed in the last iteration is predicted based on the sum of the $M_{Read[2]}$ and the equation $(M_{Gap} * (lastIteration - 2))$ (Equation 4.4). Thus, in case of $M_{Write[2]}$ is out of the memory address interval of $M_{Read[3]}$ and $M_{Read[lastiteration]}$ (Equation 4.3), one can ensure that the loop has no cross-iteration dependency (NCID). Figure 13 illustrates an example of how Cross-iteration Dependency Prediction (CIDP) works.

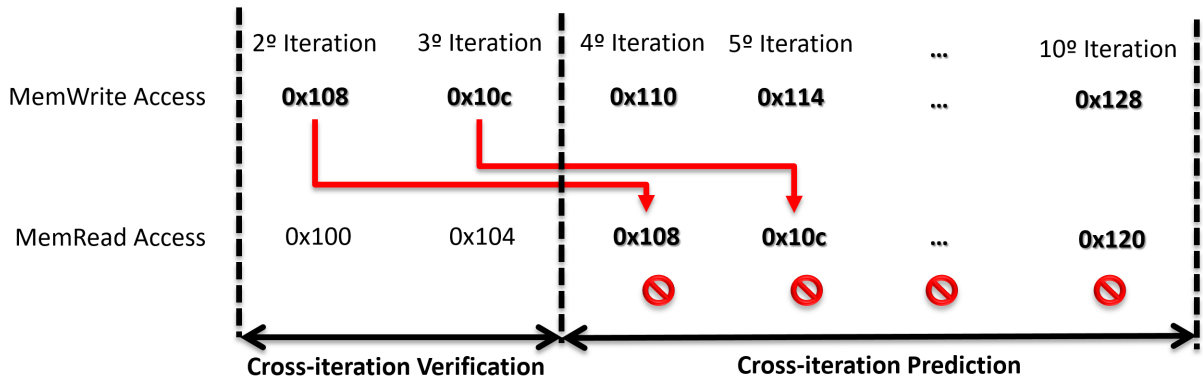


Figure 13 – Cross-iteration Dependency Prediction

In such example, the DSA detects that there is no cross-iteration dependency between 2^{nd} and 3^{rd} iterations. Thus, by the end of the 3^{rd} loop iteration, the *CIDP* is activated by applying Equation 4.5 ($M_{Gap} = |0x104 - 0x100| = 0x004$). Using Equation 4.4, one can calculate the memory address of the load instruction of the last iteration $M_{Read[lastiteration]} = 0x100 + 0x020 = 0x120$. By applying Equations 4.1 and 4.2, the *CIDP* detects that $M_{Write[2]} = 0x108$ is within the interval $(M_{Read[3]} \leq x \leq M_{Read[lastiteration]}) = 0x100 \leq x \leq 0x120$, which produces a cross-iteration dependency.

4.5 PARTIAL VECTORIZATION

Although having cross-iteration dependencies, some loops can be partially vectorized avoiding the vectorization of iterations that produce dependencies. Figure 14 shows how the partial vectorization works.

Cross-iteration Dependency Prediction

	2 ^o Iteration	3 ^o Iteration	...	11 ^o Iteration	...	20 ^o Iteration	...
MemWrite Access	0x124	0x128	...	0x148	...	0x16C	...
MemRead Access	0x100	0x104	...	0x124	...	0x148	...

Partial Loop Vectorization

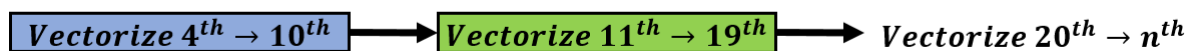


Figure 14 – Partial Vectorization Analysis

As can be seen, the *CIDP* detects dependencies between the 2nd and 11th iterations due to the data memory address *0x124*. However, there is an interval between the 2nd to the 10th iteration that can be vectorized. Thus, in this example, the DSA executes the Loop Analysis from the 1st to the 4th iteration, allowing the vectorization of the 4th to the 10th iteration. Such vectorization will generate the necessary data for the vectorization of operations from the 11th to the 19th iteration. The same process is repeated until the end of the loop execution.

4.6 DSA - ANALYSIS AND EXECUTION

4.6.1 Count Loops

Figure 15 exemplifies the execution of the DSA considering a Count Loop. As can be seen, the *Loop Detection Stage (A)* detects the loop by the end of the execution of the first iteration. In the second iteration, the *Data Collection stage (B)* identifies the loop range (400) and the value of the increment/decrement ($i = i + 1$). In addition, such stage stores the addresses of the data memory accesses (*Mem[a[i]]*, *Mem[b[i]]* and *Mem[v[i]]*) in the Verification Cache. In the third iteration the *Dependency Analysis Stage (C)* analyses dependencies between iterations (more detailed in 4.4). For the current example, the DSA verifies that there is no cross-iteration dependency and triggers the *Store ID/Execution Stage*. Such stage builds SIMD instructions to execute the remaining iterations in the ARM NEON Engine. The DSA needs four parameters to generate SIMD instructions: the data type, the loop range, the operation and the ARM NEON execution support. In such example, the parameters are: *float (F.32)*, *400*, *add*, *128-bit wide*, respectively. Considering these parameters, for the current example, the DSA generates an instruction equivalent to the *vaddq_f32* instruction of the NEON architecture (further explained in Generating SIMD Instructions). Since the corresponding ARM NEON engine can operate 128 bits in parallel and the float type is a 32-bit wide data, the DSA divides the loop range by the factor four, running the *vaddq_f32* one hundred times, instead of executing a non-vectorizable add operation four hundred times.

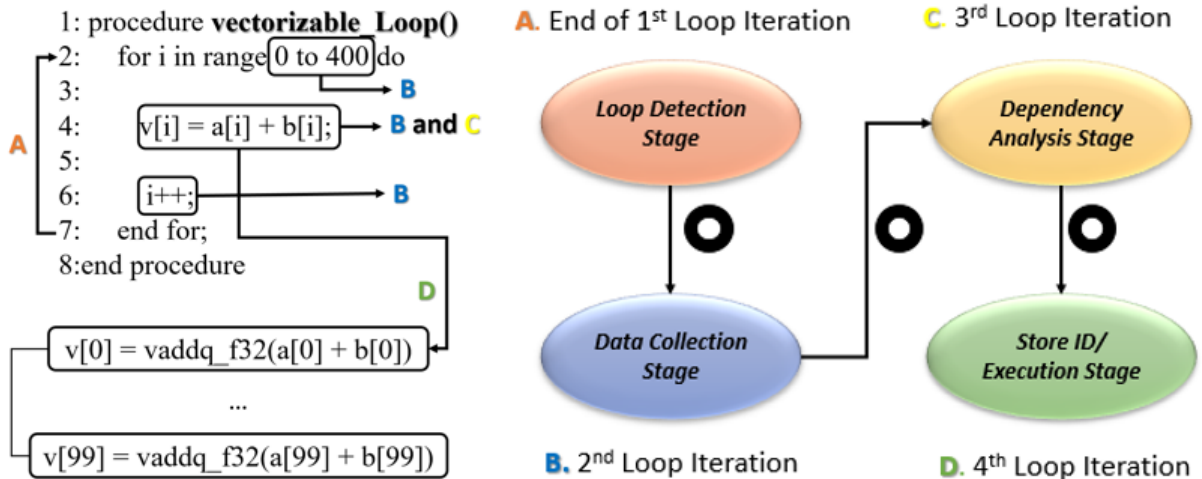


Figure 15 – Count Loop Example

4.6.2 Function Loops

Figure 16 shows how the loops containing functions are vectorized. During the *Loop Detection Stage*, the DSA keeps detecting any *jumps* that might occur. The DSA stores the *Jump* and *Return* addresses in an array periodically. If there is a *Branch* instruction that causes an instruction address regression, the DSA starts the *Data Collection Stage*. Before starting the *Data Collection Stage*, the DSA verifies if the function *Jump* and *Return* addresses (*Jump*: 4 and *Return*: 5) are within the Loop Range (2 → 7). If that is the case, the loop is classified as a *Function Loop* and instructions out of the loop range (11 → 12) are also verified during the remaining stages.

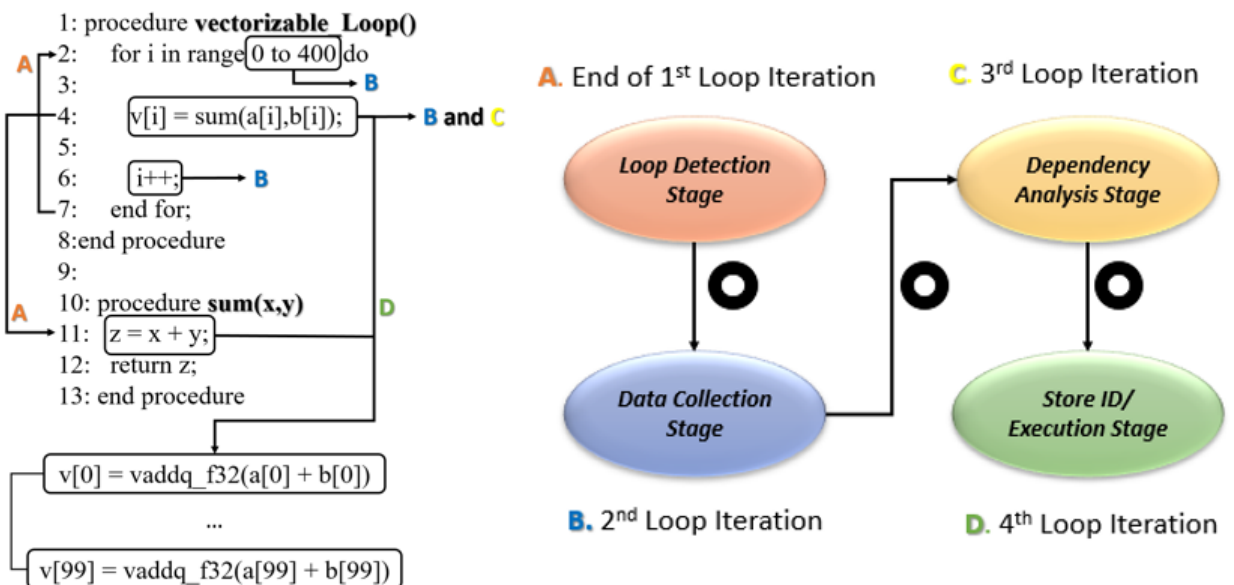


Figure 16 – Function Loop Example

4.6.3 Inner/Outer Loops

Figure 17 exemplifies the *Outer Loop* vectorization in the DSA. As can be seen, when the 1st iteration of the *Outer Loop* is executed, only the *Inner Loop* is detected by the *Loop Detection Stage* (A.1) since the *Outer-Loop* did not suffer instruction address regression. In that case, the *Inner-Loop* is tagged as vectorizable and the operation is vectorized to 36 elements (B.1, C.1, D.1). At the second time the *Outer-Loop* is executed, the DSA detects the presence of this loop and verifies if there are any vectorizable loop within this loop or any instructions between both loops. However, since the DSA still does not know the *Outer Loop* size, it cannot be vectorized in this stage. Then, the *Inner-Loop* is detected by the DSA. Since the DSA previously classified such loop as vectorizable, the *DSA Analysis* is not necessary and all the 40 elements processed are vectorized (D.1). By the end of the 2nd *Outer Loop* iteration the DSA knows it will be operating for more 8 times and there are no instructions operating over a data vector between both loops (B.2). Since there are no vectorizable instructions between the loops, by now, the DSA considers both inner and outer loop as only one, considering the loop size as 400. Since 320 elements still need to be vectorized, the DSA starts their vectorization in the 3rd iteration of the *Outer-Loop* (D.2).

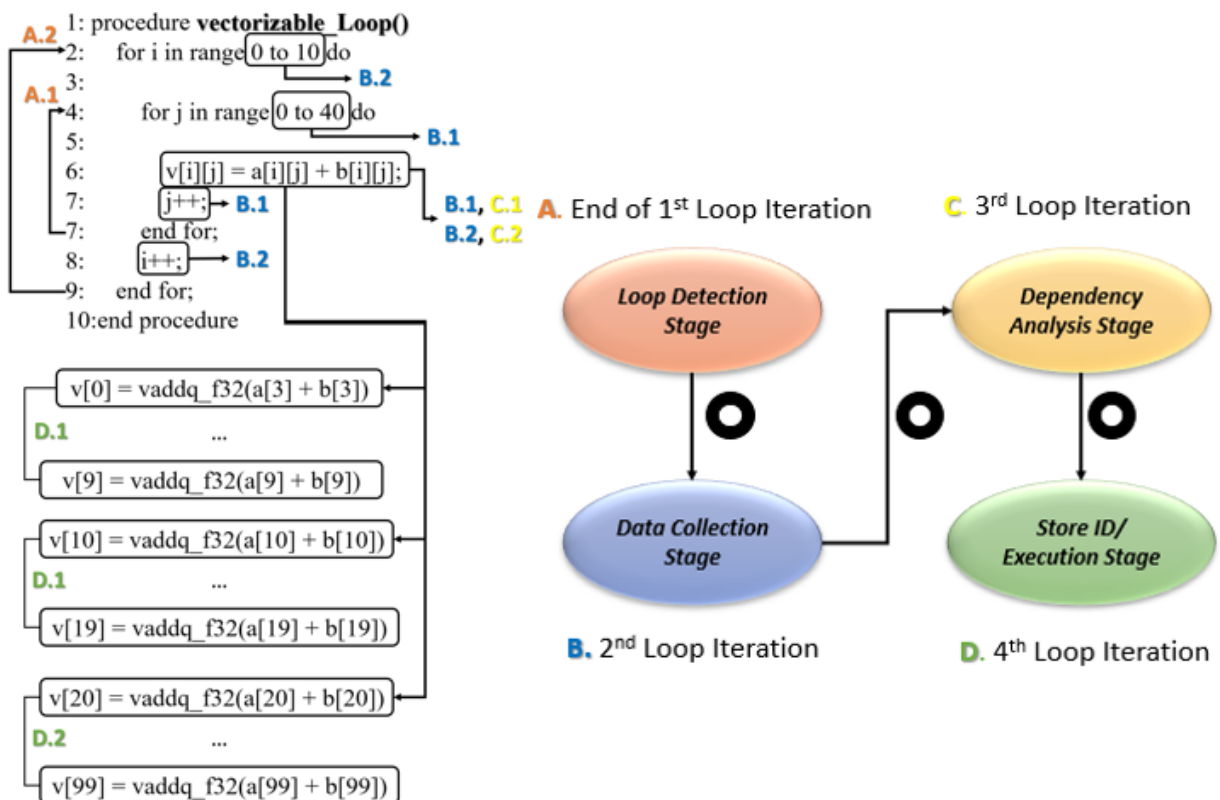


Figure 17 – Outer Loop Example

In case there are instructions between Inner and Outer Loop, we have four scenarios:

- considering a *matrix* $i \times j$, where the i represents the *Outer Loop* size and the j represents the *Inner Loop* size, if *the inner-loop instruction depends on a data generated by an outer-loop instruction*, the *Outer Loop* is vectorized i times first and then the inner loop instruction is vectorized $i * j$ times;
- considering a *matrix* $i \times j$, where the i represents the *Outer Loop* size and the j represents the *Inner Loop* size, if *the outer loop instruction depends on a data generated by an inner loop instruction*, the *Inner Loop* is vectorized $i * j$ times first and then the *Outer Loop* instruction is vectorized i times;
- considering a *matrix* $i \times j$, where the i represents the *Outer Loop* size and the j represents the *Inner Loop* size, if *both loops have data dependencies between their instructions*, the outer loop is executed sequentially while the *Inner Loop* instruction is vectorized j times every time the *Outer Loop* increases its iterations;
- considering a *matrix* $i \times j$, where the i represents the *Outer Loop* size and the j represents the *Inner Loop* size, if *both loops have no data dependencies between their instructions*, the *Outer Loop* is executed i times while the *Inner Loop* instruction is vectorized $i * j$ with no restriction in the execution order.

4.6.4 Conditional Loops

Some steps have been added on the DSA state machine to support loops with conditional code. As can be seen in figure 18, during the DSA Analysis Mode (Probing Mode), we added the stages of Mapping, which contains a Conditional Code Analysis sub-stage, and the Speculative stage. The mapping stage (for the Conditional Loop approach) is responsible for:

- evaluating the loop range within each condition;
- evaluating the vectorizable instructions and their operands within each condition;
- mapping and verifying if every conditional statement present in loop can be vectorized;
- generating and executing SIMD instructions to the already verified conditions;
- applying leftover techniques if necessary (further discussed in section 4.8).

The Speculative Stage (for the Conditional Loop approach) is responsible for:

- selecting data generated during the Mapping Stage at the end of the loop execution;
- storing the Conditional Loop mapping conditions to further executions.

In addition, the Loop Detection Stage has been extended. At the end of the first iteration, it is possible to check if there is any conditional code present in the loop through the Conditional Code Detection sub-stage.

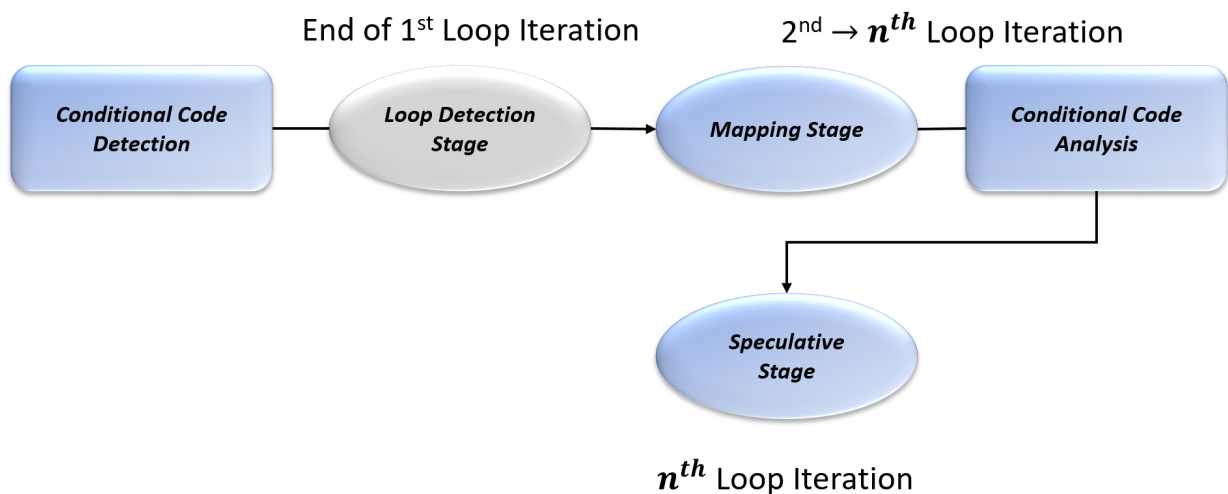


Figure 18 – DSA Conditional Loop State Machine

As can be seen, the substage of Conditional Code Detection occurs during the Loop Detection stage. During this stage, besides checking for jumps that can characterize a loop, the stage is constantly analyzing jumps that may be within the range of the loop. If a loop is found and there is a jump within the loop range, the Conditional Code Analysis phase, which occurs during the Mapping stage, is activated. In this stage, the DSA checks if the currently accessed condition is vectorizable. The condition is then marked as vectorizable or not. If the DSA detects a dependency between iterations in this condition, it classifies this loop in the DSA Cache as non-vectorizable.

As conditions are checked during loop execution, the DSA also counts and classifies the number of conditions using their instruction addresses (discussed later in section 4.6.4.1). While there are still pending conditions, the DSA continues looking for these and verifying if they are vectorizable. If such a condition is vectorizable, the DSA generates instructions and executes them. In this manner, this step is repeated until all conditions are checked. If no conditions are pending and all conditions are vectorizable, the DSA stores the loop in the DSA

Cache as vectorizable. Since there is no way to predict which conditional portion is executed in further iterations, the DSA performs a Speculative Stage, which will be discussed in section 4.6.4.2.

4.6.4.1 Conditional Loops Vectorization

Figure 19 shows an example of a Conditional Loop (2 \rightarrow 8 instruction addresses) containing two possible conditions (*A and B*). In addition, the loop execution timeline is shown at the bottom of the figure. During the first iteration (*Loop Detection Stage*), the presence of a Conditional Loop is detected. At the 2nd iteration the *Mapping Stage* is already initialized with some extra information collections (e.g.: loop size, start and end loop instruction addresses). By the 2nd iteration the *Conditional Code Analysis* step is already activated. As can be seen in the timeline, an instruction addressing gap is detected (from 5 \rightarrow 6 (*Condition A*)) during the execution of the 2nd iteration since *Condition A* is executed. In this way, the DSA starts the verification to check whether the code contained in *Condition A* is vectorizable. Since it is the first time the *Condition A* is performed, the Mapping Stage collects the data memory addresses accessed by the condition during the 2nd iteration but still cannot predict whether the condition is vectorizable or not.

During the 3rd iteration, *Condition B* is accessed. Since it is the first time the *Condition B* is performed, the Mapping Stage collects the data memory addresses accessed by the condition during the 3rd iteration but still can not predict if the condition is vectorizable. At the 4th iteration, *Condition B* is accessed another time and the Cross-iteration Dependency Prediction (CIDP) is able to classify such iteration as vectorizable by comparing the data memory addresses accessed during the 3rd and 4th iterations.

In the course of the 5th iteration, *Condition A* is accessed again. In this way, the Cross-iteration Dependency Prediction (CIDP) is able to classify such iteration as vectorizable by comparing the data memory addresses accessed during the 2nd and 5th iterations. From now on, since all conditions were classified as vectorizable, there is no need to repeat the vectorization analysis for *Conditions A and B*. Hence, during the 5th iteration, the DSA detects that there are no pending conditions to be analyzed since all instruction memory addresses within the loop have been accessed (2 \rightarrow 8).

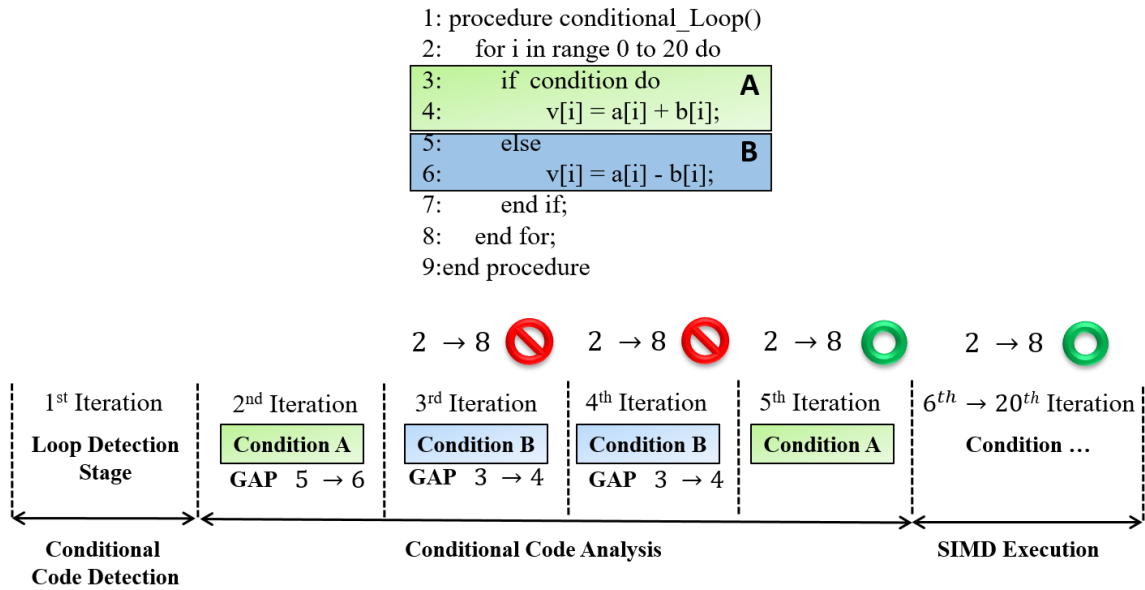


Figure 19 – Conditional Loop Vectorization Analysis

To verify if all conditions have been parsed, an address mapping becomes necessary. Figure 20 illustrates the mapping considering the example shown in Figure 19 (previously). During the 2nd iteration, *condition A* is performed, and the *Cross-iteration Prediction Analysis* begin. The DSA indexes the condition through the address of its first instruction, this information is stored in a temporary *Vector Map*. In this way, *Condition A* is indexed by address 3 (*Condition A* first instruction address (3 → 5)). During the 3rd iteration, the DSA starts the *Cross-iteration Prediction Analysis* for *Condition B*. Hence, the *Condition B* is indexed in the *Vector Map* by the address 5 (*condition B* first instruction address (5 → 6)). By the 4th and 5th iterations both conditions are classified as vectorizable. Since all instructions in the instruction addressing range (2 → 8) were executed and analyzed, in the 5th iteration the loop information is stored in the DSA cache. Such information is required to vectorize the loop without the need to repeat the vectorization analysis. The information is composed of:

- Loop ID: to identify the vectorizable regions in the loop during program execution;
- Loop Size: to generate SIMD instruction during execution;
- Conditions ID: to make the speculative execution (further discussed in the sub-section Conditional Loop SIMD Execution).

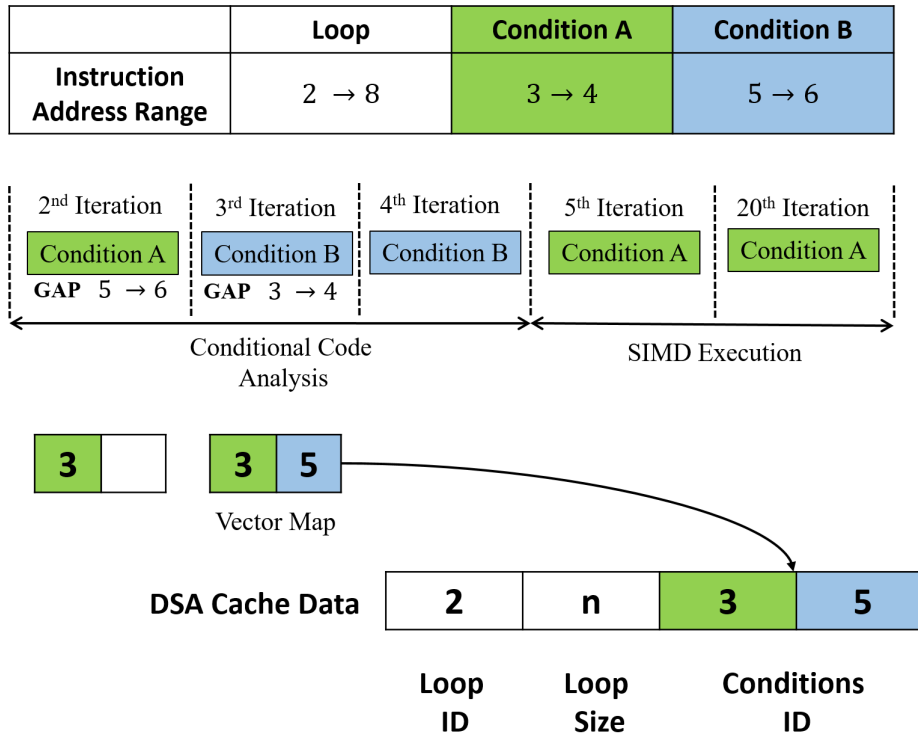


Figure 20 – Conditional Code Loop Analysis Mapping and Data Storage

4.6.4.2 Conditional Loop SIMD Execution

Figure 21 shows a SIMD execution considering the example shown in figure 19. Since condition B is verified as vectorizable during the 4th iteration, its instructions are vectorized considering the range (*Vectorize B* - 4 → 20), generating results for 16 iterations (*B - RESULTS*). In parallel, its execution is mapped to the *Vector Map* (4th Iteration - B) to later select the results produced by each condition (speculative execution). In the 5th iteration, *condition A* is executed. Since it is the first time that *statement A* is performed during SIMD execution, its instructions are vectorized considering the range of the current iteration to the end of the loop (*Vectorize A* - 5 → 20 iterations), generating results for the next 15 iterations (*A-RESULTS*). In parallel, its execution is mapped to a *Vector Map* (5th Iteration - A). At the 6th iteration, since *Condition A* has already been vectorized in the 5th iteration, its instructions are not executed (*Idle*), and only the mapping is performed (6th Iteration - A). During the 7th iteration, *Condition B* is only mapped in *Vector Map* (7th Iteration - B) since it was executed during the 4th iteration. During the last iteration (20th iteration) condition A is performed again and, since it has already been executed, it is only mapped (*Idle*) (20th Iteration - A). At the end of the loop, the DSA analyzes the *Vector Map* to select only the mapped results, while the others are discarded (*Speculative Stage*).

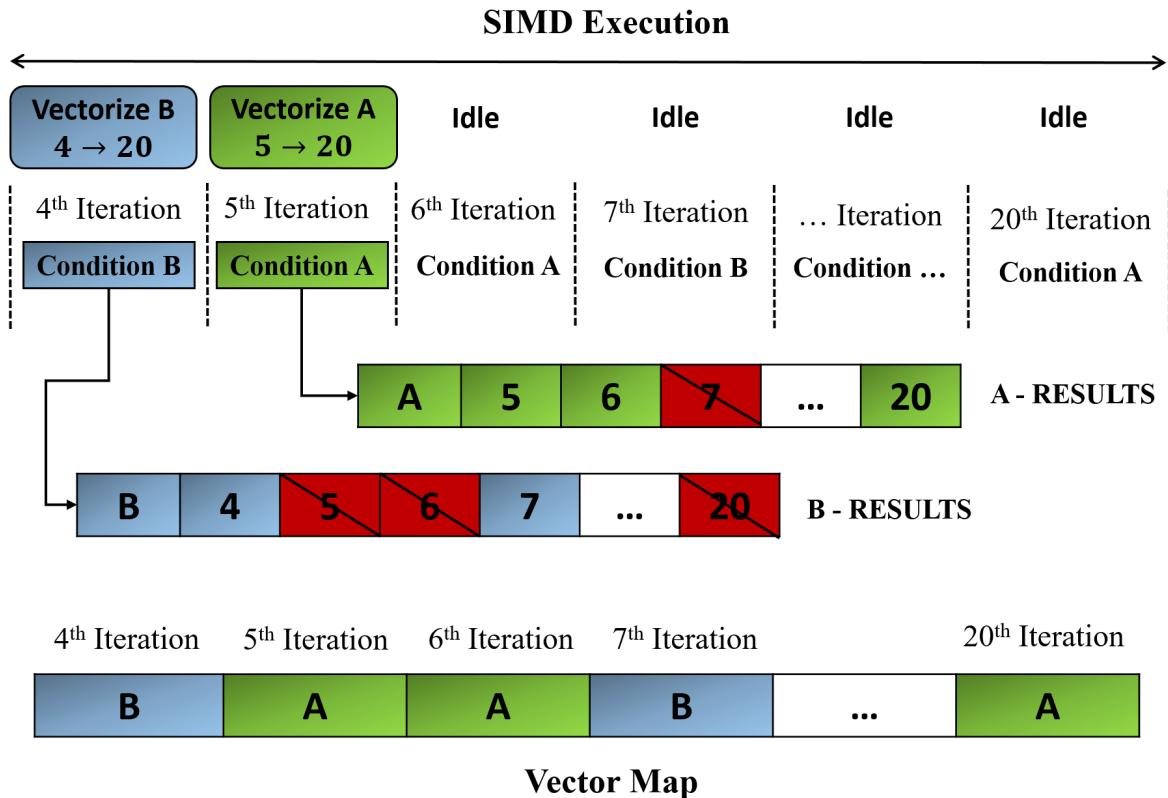


Figure 21 – Conditional Code Loop Execution Mapping and Data Storage

4.6.4.3 Conditional Loop DSA Limitations

As explained before, the Conditional Loop SIMD Execution is based on a Speculative Execution. The idea is that the DSA generates vector operation results in each condition without knowing what results will be harnessed. During the Mapping Stage, every accessed condition is mapped in a Vector Map. Such mapping is responsible for selecting the previously generated results. However, the operation results must be kept in Vector Registers to be further selected. The DSA is composed of 4 extra registers (called Array Maps), each one is 128-bit wide. Such registers are reserved to conditional loop vectorizations.

Figure 22 presents the Conditional Code Loop Array Map functionality. As can be seen, the Conditional Loop is composed of 4 conditional statements, each one with one vectorizable instruction. Supposing all operands are 32-bit wide and we are operating over 128-bit wide vector registers, we can vectorize 4 operands in parallel. Since the Loop Size is 8, the DSA must execute its Mapping and Speculative logic twice. Considering that the loop was previously classified as vectorizable by the DSA Loop Analysis, during the 1st iteration, the loop executes the condition A (Mapping Stage). In this way, the DSA vectorizes the condition A from iterations (0 → 3). During the 2nd iteration, the loop vectorizes the condition B from

iterations (0 → 3) using the *Overlapping* leftover method (further discussed at section 4.8.2). At the 3rd iteration the condition D is executed (also applying the *Overlapping* method). In the 4th iteration the condition C is executed using the *Single Elements* leftover method (discussed at section 4.8.1). Since we can only vectorize 4 elements in parallel, the Speculative Execution Stage is triggered and the DSA selects the results accessed during the Mapping Stage (Vector Map). For the next four elements (4 → 7), the same process occurs. During the 5th iteration the condition B is vectorized from iterations (4 → 7). At the 6th and 7th iterations the condition B is accessed again. Since it was previously vectorized, the DSA only maps the accesses to the Vector Map. At the 8th iteration the condition D is accessed and vectorized using the *Single Elements* leftover method. Since the DSA went through the iterations (4 → 7), the Speculative Execution Stage is triggered. In this way, the DSA selects the results accessed during the Mapping Stage (Vector Map).

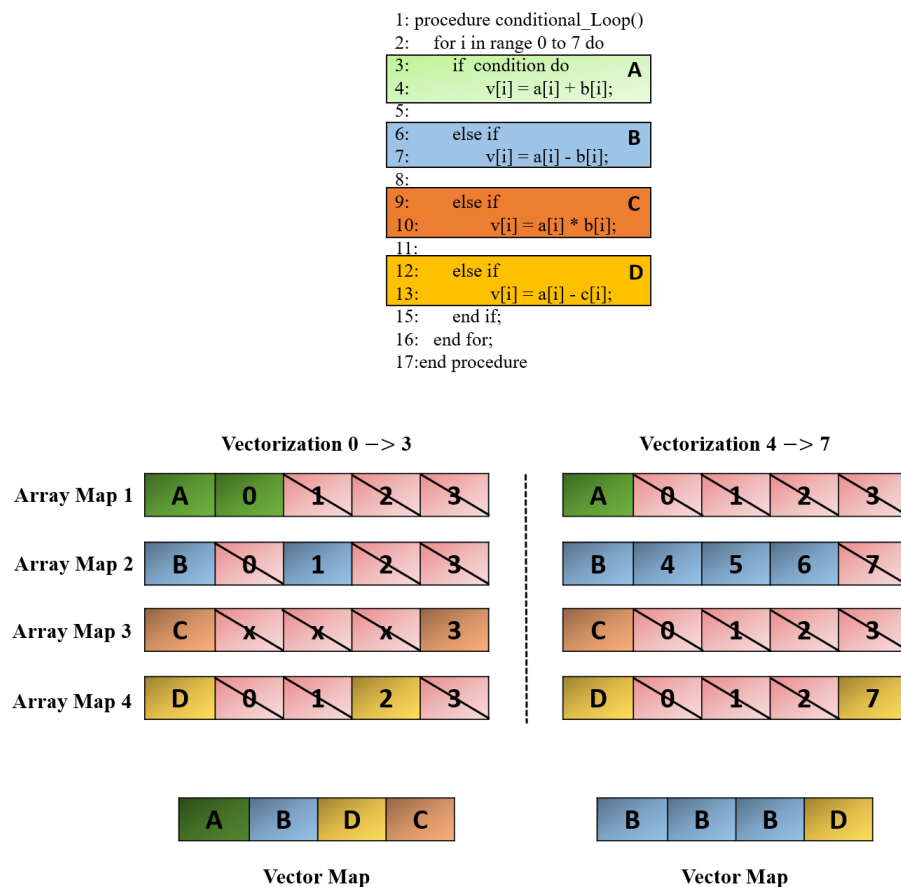


Figure 22 – Conditional Code Loop Array Map Logic

It is important to notice that the Conditional Loop vectorization is limited to the number of Array Maps available. If the Conditional Loop presents more instructions to be vectorized than Array Maps available, the DSA looks for ARM NEON Vector Registers available. If there

are no Vector Registers available, the Conditional Loop is classified as non-vectorizable.

4.6.5 Sentinel Loops Vectorization

Since Sentinel Loops have their size or stop condition calculated during loop execution, it is impossible for the DSA to know the number of times this loop will execute. To enable the execution of Sentinel Loops the DSA:

- speculates the number of times the loop will execute based on the last loop execution;
- verifies if there is cross-iteration dependency between iterations assuming a speculative loop size on every loop execution;
- partial vectorization is applied if the execution continues after determined speculative final loop iteration.

Figure 23 shows the analysis and execution of a sentinel loop. The first three iterations are responsible for checking whether there is a dependency between iterations in the loop. Since a loop size is required to predict if there is a cross-iteration dependency between iterations and there is no defined size for the Sentinel Loop, the DSA assumes a *Speculative Range*. In this way, the DSA chooses a loop size that maximizes the use of vector units. In this example, the DSA assumes a vector architecture of 128-bit Wide (*ARM NEON 128-bit Wide*). Since the size of the operands is 8 bits (*8 bits operands*), the DSA selects a speculative range of 16 (*Speculative Range 16*) in order to use all vector units. In this way, in the 3rd iteration (*Dependency Analysis Stage*), the DSA predicts that there is no dependency between iterations considering the size 16 (*Cross-iteration Prediction*) and the loop can be vectorized. In the 1st execution of the loop (*Execution Stage - 4th iteration*), the DSA executes the loop operation considering a speculative size (*Vectorize - 4 → 20*). From the 5th iteration to the 10th, the already vectorized operations are not executed (*Idle*) and the only instructions that are processed in the loop are those responsible for the stopping condition calculation. When the stop condition is reached (10th iteration), the results from iterations (4 → 10) are kept, while the operation results (11 → 20) are discarded. Since the current loop execution has 10 iterations, on the next execution the speculative range value is 16, since is the minimum operation range to be allocated at the vector units considering an operand width of 8 bits. At the second time this loop is detected by the DSA, in the *DSA Loop Analysis Stage*, it predicts that there is no cross-iteration dependency. At the 4th iteration (*Execution Stage*), the DSA executes the loop operation considering

the speculative loop range (*Vectorize* - 4 \rightarrow 20). From the 5th to the 16th iteration, the already vectorized operations are not executed. This time the loop executes until the 18th iteration (*Real Range* - 18). Despite the DSA computed results from 4th to the 20th iteration, only the results from 4th to 16th are considered, since the cross-iteration prediction was based on the range 16. The operations of the 17th and 18th iterations are sequentially executed by the ARM Processor.

There are three Sentinel Loop predicting possibilities:

- if the loop executes a smaller number of iterations than previously performed, only the results of the current range are saved and the previous loop range is replaced by the current range;
- if the loop executes a greater number of times, the remaining iterations are performed by the general purpose processor and the previous loop range is replaced by the current range;
- if the loop executes the expected number of iterations, the speculative range is retained.

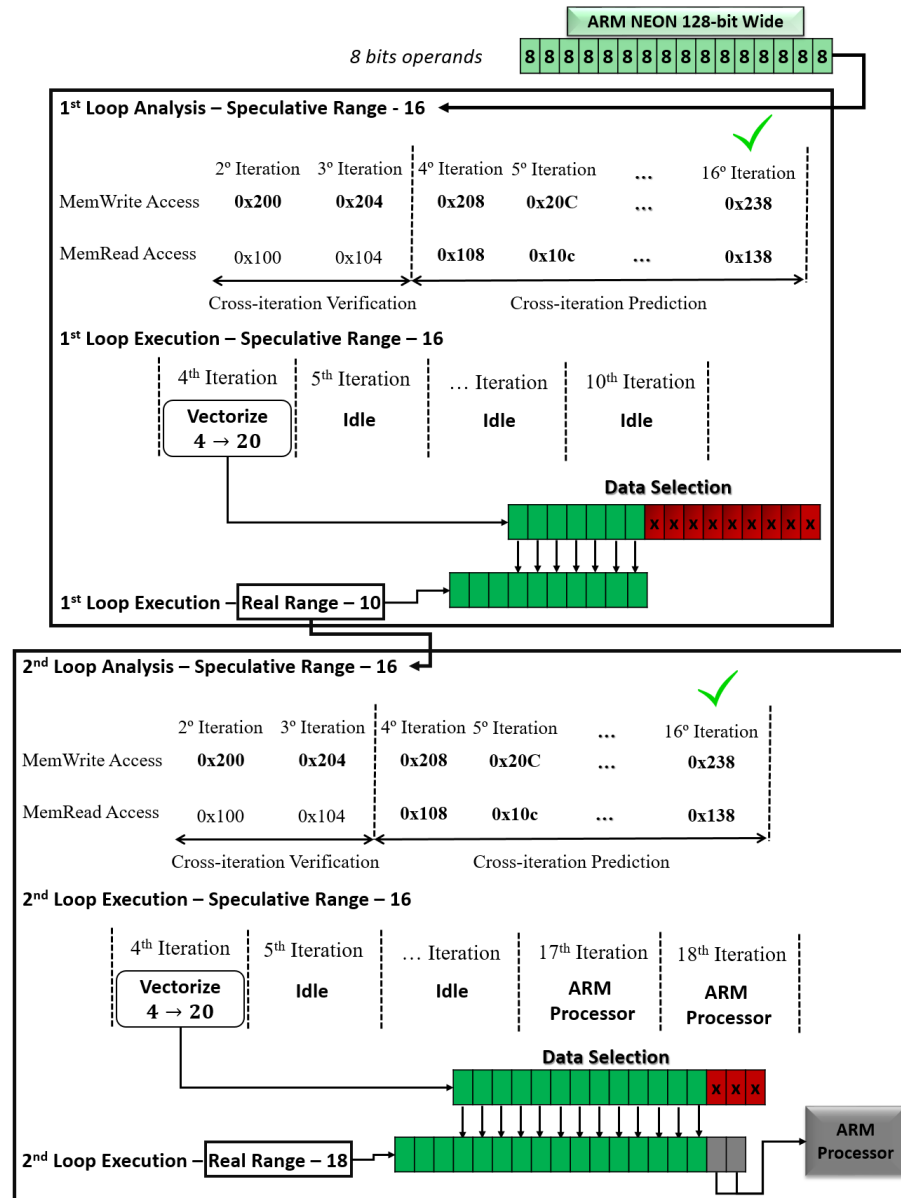


Figure 23 – Sentinel Loop Cross-iteration Analysis and Execution

4.6.6 Dynamic Range Loop Vectorization

We consider as general Dynamic Range Loops those who have their size calculated before the loop execution. Considering that, a Dynamic Range Loop can be analyzed maintaining the original DSA State Machine. However, it must be analyzed every time it repeats, since the *Dependency Analysis Stage* needs to verify if the vectorization is feasible based on loop range.

As it can be seen in figure 24, at the 1st time the loop executes, the *Dependency Analysis Stage* cannot detect any cross-iteration dependency (refer to the section 4.4). Thus, considering the loop range 5, the *DSA Loop Analysis* predicts the loop as vectorizable. However, at the 2nd time the loop executes, a cross-iteration dependency is detected at the 10th iteration (*MemRead*

Access = 0x120 = MemWrite Access). Such an example shows that different loop sizes imply in a different *DSA Loop Analysis* (such problem is solved by the *Partial Vectorization - 2.5*).

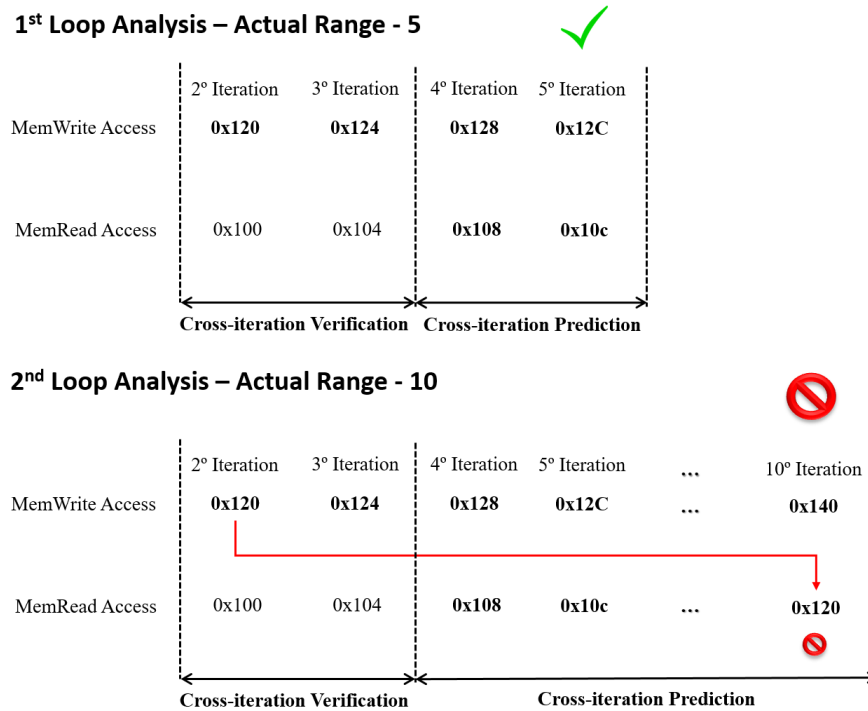


Figure 24 – Dynamic Range Loop Cross-iteration Analysis

4.7 GENERATING SIMD INSTRUCTIONS

The NEON engine execution consists of three steps: loading data to vector registers, operating over data with NEON functional units and storing the data to memory or moving data to scalar registers. Thus, in order to generate NEON instructions, the DSA should detect: the Loop Size, the Vector Data, Vectorizable Instructions and Operand Types.

Figure 25 demonstrates an example of how the DSA collects data from vectorizable regions and generates SIMD instructions. As can be seen, the present assembly is generated from a Vector Sum. To detect *Index and Stop Condition*, the DSA looks for the instructions that are responsible for incrementing the index. In such example, the respective instructions are the *ldr* and *str*. By knowing such instructions, the DSA is able to know the registers acting as indexes (in this case *r5*, *r10* and *r2*). Along with it, the DSA detects instructions that indicate the loop stop condition. In such example, the respective instruction is the *cmp* instruction which compares the constant value present in register *r4* and the memory address (index) present in *r5*.

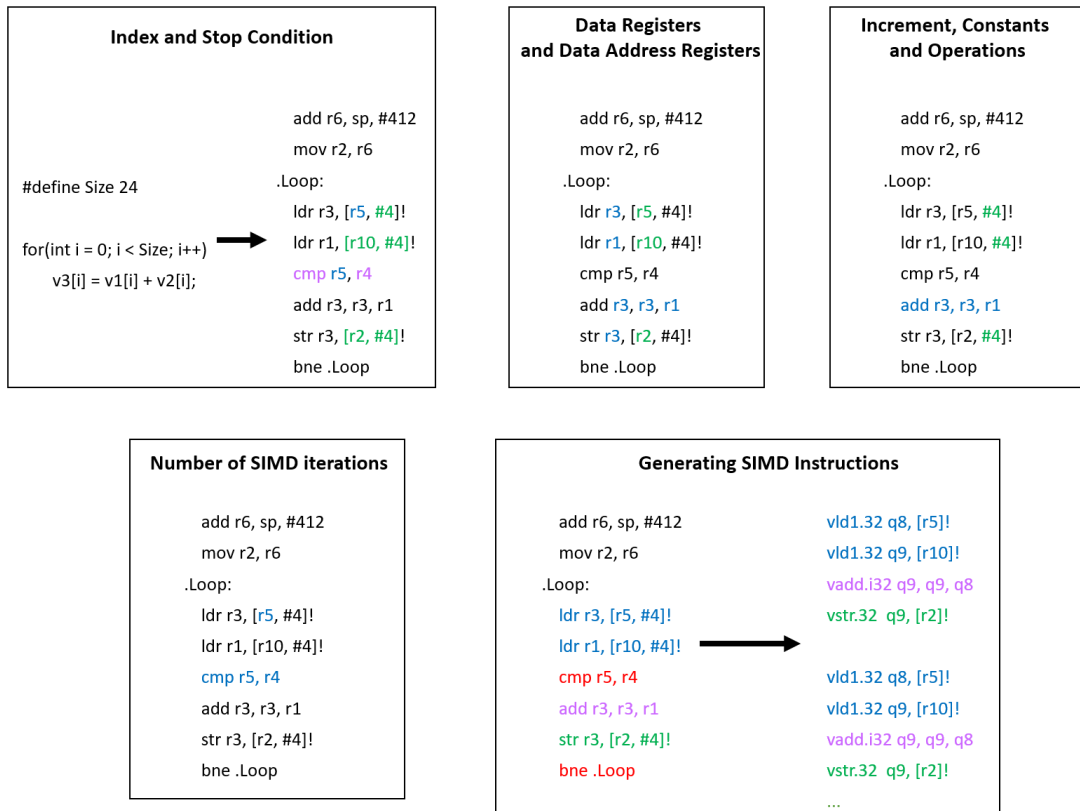


Figure 25 – SIMD Instruction Generation Steps

To evaluate which Registers contain Data and which Registers contain Data Addresses (*Data Registers and Data Address Registers*), the DSA evaluates every *MemRead* and *MemWrite* instruction types. In such example, *r3* and *r1* are used as target registers in the *ldr* and *str* instructions, which means that they are classified as Registers containing Data. Analyzing the R_n register in the *MemRead* and *MemWrite*, the *r5*, *r10* and *r2* can be classified as address registers. In such case, the index and address registers are the same, since the *ldr* and *str* instructions are able to do memory operations and increment.

To detect which data type is being processed, the DSA verifies: the *MemRead* instruction (e.g.: LDRB for Byte and LDR for Word), the instruction bytecode and its destiny register type (float or int instructions). In such example, we consider *32 bits int data*.

Increments, constants and operations must be detected by the DSA. To detect increments, the DSA evaluates the instructions that operate over registers containing Data Addresses. As can be seen, the only constant present in the loop is the loop increment (*#4*). Considering the operations detection, ALU type instructions that operate over Registers Containing Data, are classified as operations by the DSA (*add r3, r3, r1*).

Being aware about the increment (*#4*), the initial index in *r5* and the instruction that

defines the stop condition, the DSA is able to determine the number of iterations (*Size*) (*Number of SIMD Iterations*). Depending the number of iterations, the DSA may assume different instruction generations based on the leftover techniques (further discussed in section 4.8).

With the collected data, the DSA must operate over data sizes of 32 bits. Considering the ARM NEON 128-bit wide, the DSA can execute 4 data per vector instruction ($D_{pv} = NEON_w / Data_s = 128 / 32 = 4$, where D_{pv} is the amount of data per vector, $NEON_w$ is the ARM NEON width and $Data_s$ is the operands data size). At the Store ID/Execution Stage (or Speculative Stage), the DSA generates the SIMD instructions assuming the NEON execution flow (*Generating SIMD Instructions*):

- the DSA generates the vector load instructions based on the load registers containing data addresses ($r5$ and $r10$) and the operands data size (32 bits) ($vld1.32\ q8, [r5]!$ and $vld1.32\ q9, [r10]!$);
- the DSA generates the SIMD operation based on the collected operations and the operands data size (32 bits) ($vadd.i32\ q9, q9, q8$);
- the DSA generates a vector store instruction based on the store register containing data address ($r2$) and the operands data size (32 bits) ($vstr.32\ q9, [r2]!$);
- the DSA repeats the SIMD instruction generation 6 times ($n = L_{range} / D_{vu} = 24 / 4 = 6$, where n is the number of times the DSA must generate the SIMD instructions, L_{range} represents the loop range and D_{vu} is the amount of data which can be processed in parallel).

4.8 DEALING WITH LEFTOVERS

The NEON Engine execution generally implies on performing operations on data vectors of 2, 4, 8, 16 or 32 elements. However, it is also possible to find arrays that are not multiple of such values. In this way, the remaining elements must be processed individually. Figure 26 shows an example where there are remaining elements to be processed. As can be seen, there are 21 elements to be processed. Assuming that, in this example, one can perform up to 8 data per operation, the first two operations will be executed normally, whereas the third iteration will not have enough elements (multiples) to be executed. In order to enable the vectoring of non-multiple arrays, three techniques are implemented in the DSA: Single Elements, Overlap and Larger Arrays. Such techniques are discussed in the following subsections.

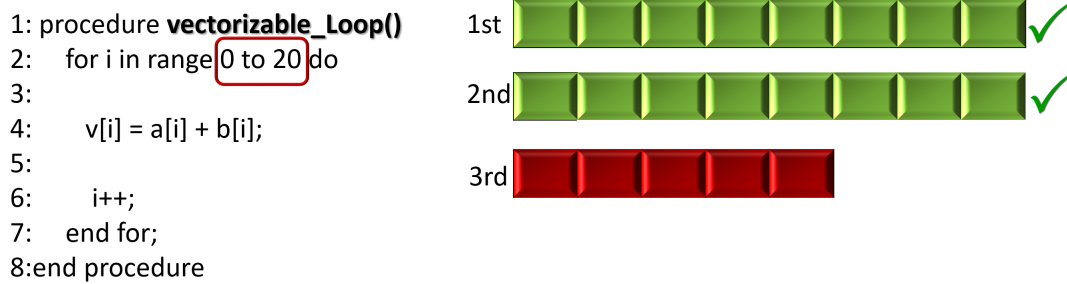


Figure 26 – Leftovers

4.8.1 Single Elements

The NEON instruction set provides LOAD and STORE instructions that can operate over individual elements. Figure 27 shows how the Single Elements method works. As it can be seen, only 2 vector operations could be executed $0 \rightarrow 7$ and $8 \rightarrow 15$ since the size of the array is not multiple of 8. In this way, elements $16 \rightarrow 20$ must be loaded, processed and stored individually.

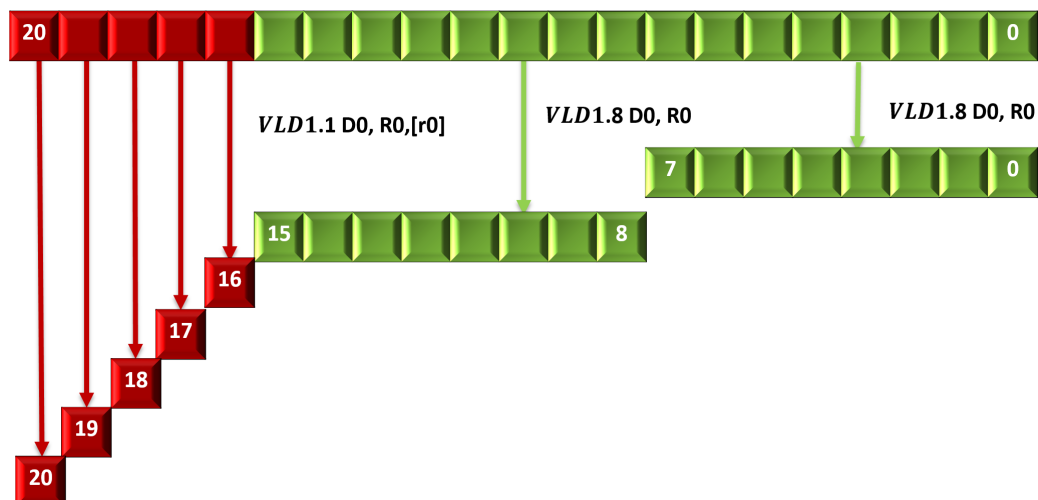


Figure 27 – Single Elements Method

The Single Elements technique is the slowest among the three techniques mentioned, since each element must be loaded, processed and stored individually. It is important to notice that such method requires the execution of two loops, the first one for the execution of the vectors and the second for the execution of the individual elements.

4.8.2 Overlapping

The Overlapping method involves processing some elements of the array twice to cover the operation of the remaining elements. Figure 28 shows the functionality of such method. As shown, since the array size is not a multiple of 8, it is unfeasible to perform three vector operations. Through the use of Overlapping, the first operation processes the data from 0 → 7 normally. However, the second instruction will repeat the operation performed on the elements 5, 6 and 7, operating over the elements 5 → 12. In this way, the last operation is performed over 8 elements 13 → 20.

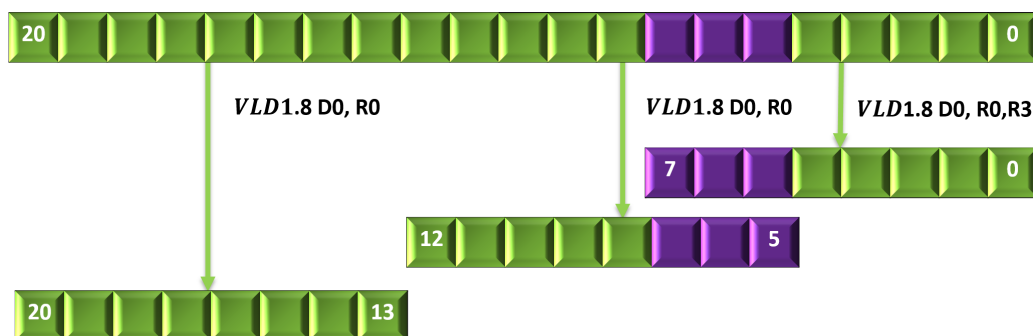


Figure 28 – Overlapping Method

However, the Overlapping method is only allowed when the operations applied over the elements do not vary the resulting array regardless of the number of times the operation is applied. In addition, the number of elements present in the array must fill at least one vector completely.

4.8.3 Larger Arrays

The Larger Arrays technique is based on changing the size of the array being processed. The value of the array is raised to the next multiple value that the vector unit supports. Figure 29 shows how the Larger Arrays method works. As can be seen, in order to make the array size (21) a multiple value for execution on the vector unit (24), three adjacent elements need to be allocated. In this way, it was possible to use the vector engine through three operations: 0 → 7, 8 → 15 and 16 → 23.

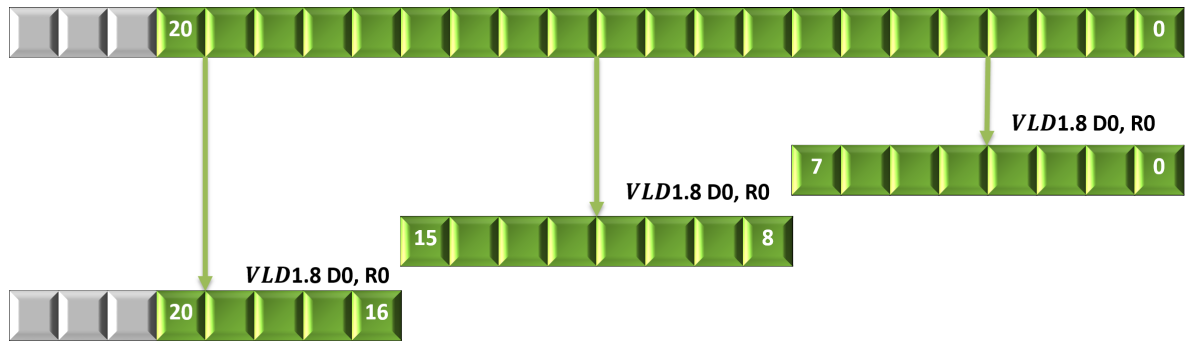


Figure 29 – Larger Arrays Method

However, allocating larger arrays results in greater memory occupation. Such size can be significant depending on the number of arrays that have leftovers to execute. These new elements at the end of the array need to be initialized to not affect the final calculation result.

5 METHODOLOGY

We coupled the DSA to the gem5 O3CPU Model (ARMv7 ISA) to evaluate the performance of the proposed approach. The gem5 simulator is a modular platform for computer-system architecture research that enables the execution of a variety of architecture binaries with Linux emulation. The gem5 provides us high simulation precision (tick precision), several architectural configurations possibilities and architectural information during runtime (e.g: instructions, registers, data and instruction memory access). Figure 30 shows how the performance results were generated. As can be seen, we extracted the results using a trace level simulation. While the gem5 O3CPU executes benchmarks, the DSA monitors all O3CPU incoming instructions and tick information. In this way, we could check the DSA Vectorization Analysis functionality. To evaluate the DSA Execution we detect the vectorizable regions and adjust the timing model replacing the scalar vectorizable instructions by vector instructions. To improve the simulation accuracy, we infer several latencies to both DSA Analysis and Execution Stages. For the DSA Analysis we consider:

- DSA Cache Access Latency;
- Verification Cache Access Latency;
- Array Map Access Latency - (Conditional Loop);
- Partial Vectorization Latency - Multiple Cross-iteration Analysis.

For the DSA Execution we consider:

- Pipeline Flush Latency;
- Non-Vectorizable Instructions Latency;
- Load Data Vector (Scalar Register to Vector Register) Latency;
- Store Data Vector (Vector Register to Scalar Register) Latency;
- Leftover operations Latency.

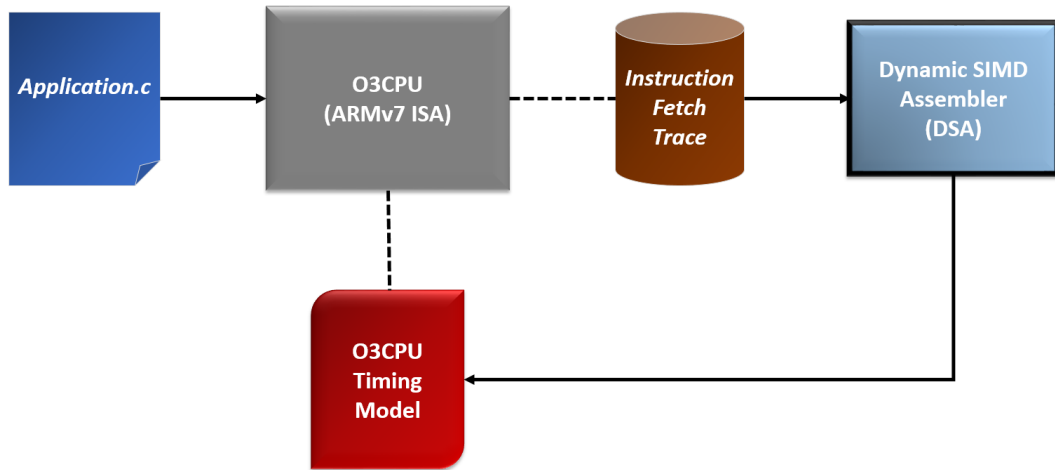


Figure 30 – DSA Simulation Model

5.1 O3CPU PROCESSOR/DSA IMPLEMENTATION

The O3CPU processor is an out-of-order model that has an ISA-independent pipeline. However, there are parts of its implementation which are composed of specific ISA functions. Currently, the processor is compatible with the Alpha, ARM and x86 architectures. This model was chosen because it presents a higher timing precision when compared to other simulators implemented in high level (eg.: SimpleScalar, MultiSim) and also because it supports the ARMv7-A ISA that implements vector instructions (ARM NEON).

Figure 31 illustrates the O3CPU pipeline steps and where DSA was coupled. As can be seen, the O3CPU processor has 7 pipeline stages, since it is an out-of-order processor, the Issue and Commit stages become present. To perform the DSA Loop Analysis Stage, the DSA must be aware of all the incoming instructions arriving on the processor and the order such instructions arrive. In this way, the vectorization analysis takes place during the Fetch Stage of the O3CPU pipeline. In order to execute SIMD instructions, the DSA stalls the Fetch Step, waits for every instruction to be flushed from the pipeline and then addresses SIMD instructions to the Issue Stage. At the end of the SIMD execution, the Fetch Step receives the instruction address that succeeds the last instruction address of the loop.

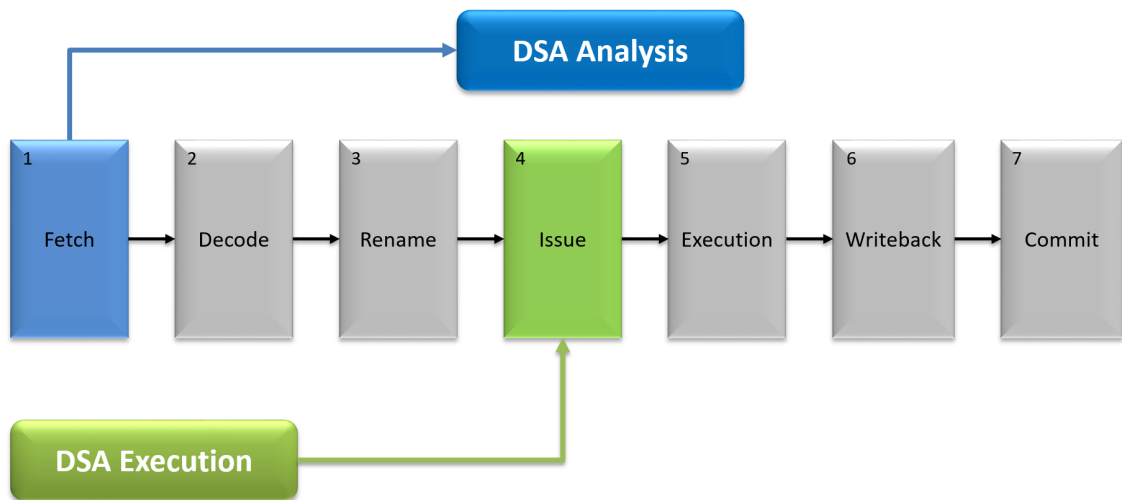


Figure 31 – O3CPU - DSA Implementation

5.2 DSA AND O3CPU ENERGY RESULTS

We used the Cadence RTL Compiler and ModelSim to gather energy results from the VHDL description of the Dynamic SIMD Assembler Analysis Stage and McPAT to gather energy results of the O3CPU Model (ARMv7 ISA). To improve the DSA energy consumption accuracy, we developed different scenarios based on different loop types. Figure 32 illustrates such exploitation. Considering the *Conditional Loop Execution* the *DSA Analysis* performs the states: *Loop Detection Stage* → *Data Collection Stage* → *Dependency Analysis Stage* → *Store ID/Execution Stage*. Unlike the *Conditional Loop* approach the *Count Loop* performs the states: *Loop Detection Stage* → *Mapping Stage* → *Speculative Stage*. As can be seen, for each scenario different logic paths (different states) are accessed resulting in several possible energy results.

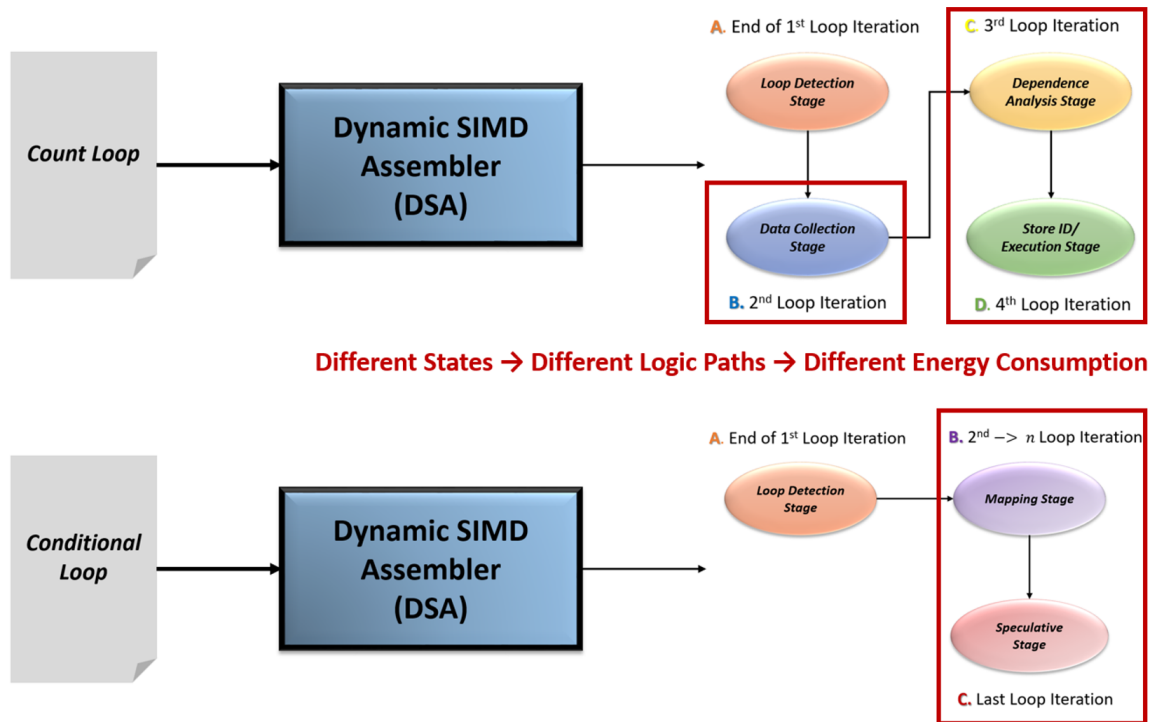


Figure 32 – DSA Energy Analysis

5.3 SYSTEMS SETUP

We have coupled the DSA to an ARMv7 ISA processor using the O3CPU model of gem5 simulator to evaluate the proposed approach. In all articles (Sections 6,7 and 8) we considered the same System Setup for all systems. Table 4 shows the configurations of all setups presented in the subsequent articles.

Table 4 – Systems Setup

Configurations	ARM Original Execution	ARM NEON DSA	ARM NEON (AutoVec and Hand-Coded)
Processor	gem5 O3CPU (ARMv7)	gem5 O3CPU (ARMv7)	gem5 O3CPU (ARMv7)
Superscalar Width	2 wide	2 wide	2 wide
CPU Clock	1GHz	1GHz	1GHz
L1 Cache	64 kb	64 kb	64 kb
L2 Cache	512 kb	512 kb	512 kb
Cache Policy	LRU	LRU	LRU
Parallelism (NEON)	Not Used	Type Dependent 128-bit Wide	Type Dependent 128-bit Wide
NEON Registers	Not Used	Sixteen 128-bit (Q0 - Q15)	Sixteen 128-bit (Q0 - Q15)
DSA Cache	-	8 kb	-
Verification Cache	-	1 kb	-
Array Maps (Conditional Loop)	-	4 (128-bit Wide)	-

It is important to notice that we coupled the same ARM NEON architecture in ARM NEON DSA and the ARM NEON Approaches, which provides the same DLP exploitation degree. Also, considering a conditional loop execution, the max number of array maps for the speculative execution is 4 (128-bit Wide), which limits the number of vectorizable instructions within conditional statements. E.g.: If a Conditional Loop has 2 conditional statements, both of them can have 4 vectorizable instructions. In case of having unused ARM NEON registers, such registers can be used to the speculative execution, increasing the number of allowed vectorizable instructions.

6 ARTICLE 1 - IMPROVING SOFTWARE PRODUCTIVITY AND PERFORMANCE THROUGH A TRANSPARENT SIMD EXECUTION

Improving Software Productivity and Performance through a Transparent SIMD Execution

Michael Guilherme Jordan, Tiago Knorst and Mateus Beck Rutzig

Electronics and Computing Department - Federal University of Santa Maria – Santa Maria – Brazil
{michael.jordan, tiago.knorst}@ecomp.ufsm.br; mateus@inf.ufsm.br

Abstract— Multimedia and DSP applications have been widely present in embedded devices. Due to their intrinsic nature, such application domains are benefited from Data Level Parallelism (DLP) exploitation, which is mostly employed in current embedded platforms by using vectorization techniques extending the underlying ISA. However, such strategy relies on specific library which affects software productivity and compiler support, such as ARM auto-vectorization approach, which breaks binary compatibility. This work proposes a transparent Dynamic SIMD Assembler (DSA) that is capable of detecting vectorizable code regions at runtime without requiring specific library or compilers. As a case study, we coupled the DSA to a 128-bit wide ARM NEON Engine. Results show that the proposed approach shows performance improvements of 31% over the original execution (without DLP exploitation). In addition, Dynamic SIMD Assembler, besides keeping binary compatibility, outperforms ARM auto-vectorization technique in 6%.

Keywords— *DLP, SIMD, Vectorization, ARM*

I. INTRODUCTION

Multimedia and DSP applications are increasingly present on current mobile devices demanding efficient software execution to respect power constraints imposed by battery supply. Instruction and Thread-Level Parallelism are widely exploited on such platforms by applying aggressive superscalar execution and increasing the number of cores encapsulated in a single die. However, such application domains are not benefited, in terms of both performance and energy, from their exploitation due to the well-known limitation of Von Neumann execution model.

Dataflow machines [13][14] and Reconfigurable Architectures [15][16] have been arising to overcome Von Neumann bottleneck by ridding the control flow execution model. However, such architectures rely on a high degree of functional unit replication, which does not respect the energy constraints of embedded devices.

Most market processors have been coupling vector processing units (i.g. IBM AltiVec [8], x86 AVX [7] and ARM NEON [6]) to allow Single Instruction Multiple Data (SIMD) execution since such application domains offer great opportunities to exploit Data Level Parallelism (DLP). As the number of functional units required to employ such execution model is smaller than Dataflow machines, it is feasible to achieve performance improvements with low power consumption by merging SIMD and Von Neumann execution models.

However, most SIMD engines rely on specific libraries, which increases the development process lifecycle affecting the software productivity. In addition, such libraries do not completely abstract the hardware complexity and most SW developers do not have enough knowledge about the architecture implementation details to exploit the potential of the vector processing engines.

Automatic code vectorization techniques extract DLP by building SIMD instructions over vectorizable code regions to exploit vector processing engines at compile time. However, although keeping

software productivity by avoiding the use of specific libraries, auto-vectorization techniques still rely on code recompilation which breaks binary compatibility. In addition, such an approach is restricted to static code exploitation, which limits the DLP extraction since it is difficult to identify vectorizable regions, such as conditional statements, which can affect the vector processing engine utilization [12].

This work proposes a transparent Dynamic SIMD Assembler (DSA) that is capable of exploiting DLP at runtime by identifying vectorizable loops to generate SIMD instructions. Unlike most market SIMD engines, due to its transparent fashion, the development process life cycle is maintained since it does not rely on specific libraries. As SIMD instructions are built at runtime, unlike automatic code vectorization techniques, binary compatibility is also maintained. Moreover, the dynamic nature of the proposed technique opens the room to achieve higher performance than static auto-vectorization approaches.

This work is organized as follows. Section II presents the Related Work. Section III presents the Dynamic SIMD Assembler System. Methodology and Results are shown in Section IV. Finally, we present the conclusion and future works in Section V.

II. RELATED WORK

The SIMD vectorization is widely used in several emerging market platforms, such as the Intel SSE, IBM AltiVec, and ARM NEON architectures. In the academic field, several researches are exploiting Data Level Parallelism (DLP) to achieve performance improvements and energy savings. Sara S. Baghsorkhi [1] proposes FlexVec architecture that combines a novel partial vector code generation technique with new vector instructions to dynamically adjust vector length for loop statements affected by runtime cross-iteration dependencies. FlexVec vectorization coupled to the Intel AVX-512 ISA shows a Geomean performance improvement from 9% to 11%. Although it is able to perform optimizations over loops with cross-iteration dependencies, the method breaks binary compatibility, since it is necessary a specific ISA adjustment and also relies on a particular compiler and library development.

Dorit Nuzman [2] evaluates and applies a compiler outer loop vectorization technique focusing on properties of modern SIMD architectures. It shows that even though current optimizing compilers do not apply outer loop vectorization, they can provide significant performance improvements over innermost loop vectorization. The proposal achieves performance improvements of 3.13 and 2.77 when coupled to a Cell BE SPU and PowerPC970, respectively. Similar to our proposal, the authors focused on vectorizing both innermost and outer loops but it relies on compiler support.

Being aware that most research focuses on vectorizing loops, Tian Xinmin [3] presented a set of new C/C++ high-level extensions for SIMD programming capable of automatic translating both functions and loops. Significant speedups (from 3.07x to 4.69x) are achieved when these optimizations are applied. Similar to aforementioned related works, it relies on specific compiler and library to achieve

performance improvements, which breaks SW compatibility and affects SW productivity.

Hoseok Chang [4], employed a unique memory access hardware, solving the non-aligned and irregular data memory access operations to improve the performance of a SIMD processor based on ARMv4 architecture. In addition, it develops an auto-vectorization compiler, which utilizes the proposed hardware. By applying such technique, the number of vectorized loops increases 50%, which provides 77% of performance improvement in the MPEG2 encoder execution.

Besides the research above, many studies are also focused on applying reconfigurable architectures, since besides exploiting ILP, they are also capable of exploring DLP. A reconfigurable architecture, named as Samsung reconfigurable processor (SRP), is developed for digital signal processing [5]. The SRP architecture is designed to handle mobile multimedia applications efficiently. It uses a CGRA to vectorize innermost loops by using a conventional C/C++ programming model to annotate the code. Despite the huge chip area required to the CGRA, the SRP relies on compiler, library and ISA modifications. In addition, it requires a design-time step to create CGRA configurations for each application which reduces, even more, the binary compatibility and SW productivity.

ARM NEON [6] is introduced in the ARMv6 architecture. The NEON auto-vectorization compiler generates vectorizable code by instantiating SIMD instructions. Despite the advantages of auto-vectorization, the static code exploitation limits the performance gains since it is difficult to identify vectorizable regions of conditional statements, function calls or even loops that contain codes between inner-loops and outer-loops. To overcome such issues, another strategy offered by the ARM to explore the NEON engine is the use of ARM NEON library, which transfer the vectorization task responsibility to the SW developer which affects SW productivity.

Table 1 compares all the aforementioned works with the proposed approach. As it can be seen, binary and software compatibility are not prioritized in their designs since they employ ISA modification or specific libraries. Our work proposes a transparent Dynamic SIMD Assembler that is capable of building SIMD instructions at runtime. The proposed approach coupled to the ARM NEON engine provides:

- higher performance than ARM auto-vectorization method with binary compatibility since is not necessary to recompile the source code;
- SW productivity by avoiding the use of the ARM library in the code development lifecycle to take advantage of the NEON engine processing capabilities.

Table 1 – Related Works and Proposed Technique Characteristics

	Code Recompilation	Library Development Support	ISA Modification	SW Productivity	Binary Compatibility
[1]	Yes	Yes	Yes	Affected	No
[2]	Yes	No	Yes	Not Affected	No
[3]	Yes	Yes	Yes	Affected	No
[4]	Yes	No	Yes	Not Affected	No
[5]	Yes	Yes	Yes	Affected	No
[6]	Yes	No	Yes	Not Affected	No
DSA	No	No	No	Not Affected	Yes

III. SYSTEM OVERVIEW

The proposed Dynamic SIMD Assembler (DSA) is tightly coupled to an ARM Cortex-A12 processor (ARMv7 ISA). Figure 1 shows the system overview. As it can be seen, the DSA is composed of a SIMD instruction logic detection and two cache memories (*DSA Cache* and *Verification Cache*). The DSA Cache is responsible for storing information about the built SIMD instructions over the vectorizable loops. The Verification Cache stores the addresses of data memory accesses performed into the vectorizable loops (more details about caches in Section IV).

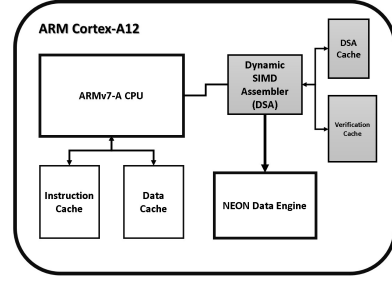


Figure 1 - System Overview

Figure 2 shows an overview of how the DSA works. In the first scenario (*Scenario 1 – Ordinary Execution*), the DSA and ARMv7 processor operate in parallel. While the ARM Cortex-A12 processor executes the incoming instructions, the DSA is in a probing mode, searching for a vectorizable loop to build SIMD instructions. In such execution mode, the NEON Engine remains deactivated. If the DSA detects a vectorizable loop, the second scenario is triggered (*Scenario 2 – DLP Exploitation*). In this scenario, the DSA deactivates the ARM Cortex A12 processor and activates the NEON Data Engine to execute the built SIMD instruction.

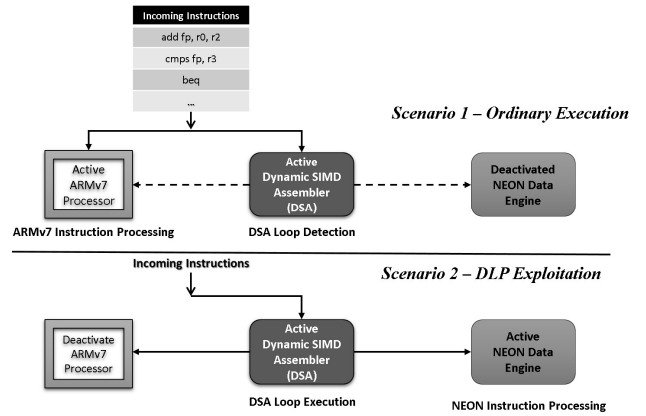


Figure 2 – System Functionality Overview

IV. DYNAMIC SIMD ASSEMBLER

This section is divided into five subsections. Subsection A shows a superficial analysis of the Dynamic SIMD Assembler (DSA). Subsection from B to E presents a more detailed analysis of the DSA stages.

A. Dynamic SIMD Assembler Overview

As shown in Figure 3, the Dynamic SIMD Assembler (DSA) detection process is based on a State Machine (SM) composed of four stages: Loop Detection, Data Collection, Dependency Analysis, and Store ID/Execution. Each one of these stages is activated in different loop iterations.

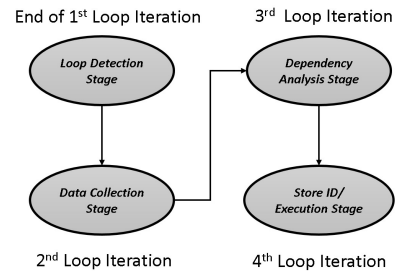


Figure 3 – State Machine of DSA

As it can be seen, the state machine starts in the Loop Detection stage and is triggered by the end of the first loop iteration. The Loop Detection stage is responsible for:

- checking the existence of innermost-loop and outer-loops at runtime;
- accessing the DSA cache, checking if the current loop is already vectorizable.

The Data Collection stage is triggered in the second loop iteration. This stage is responsible for:

- evaluating the loop range (number of iterations)
- identifying the existence of a function call inside the loop;
- storing the addresses of data memory accesses in the Verification Cache.

The Dependency Analysis stage is triggered in the third loop iteration. This stage is responsible for:

- analyzing the cross-iteration dependency (dependencies between two or more iterations in the same loop).

The Store ID/Execution stage is triggered in the fourth loop iteration. This stage is responsible for:

- generating and saving the loop identification (ID) in case of a vectorizable loop;
- building SIMD instruction and activating the execution on NEON engine.

Figure 4 exemplifies the execution of the DSA considering a vectorizable loop (*vectorizable_Loop()*) and a non-vectorizable loop procedures (*non_vectorizable_Loop()*).

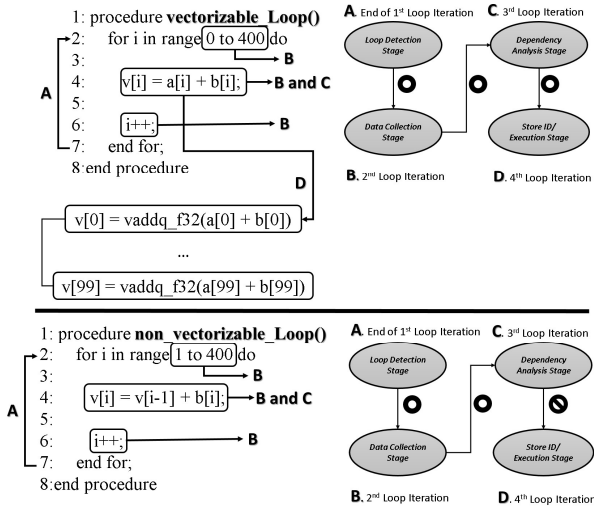


Figure 4 – DSA Execution

Considering the *vectorizable_loop()* procedure, the Loop Detection stage (A) detects the loop by the end of the execution of the first iteration (more detailed in Section C). In the second iteration, the Data Collection stage (B) identifies the loop range (400) and the value of the increment/decrement ($i = i + 1$) (more detailed in Section D). In addition, such stage stores the addresses of the data memory accesses ($Mem[a[i]]$, $Mem[b[i]]$ and $Mem[v[i]]$) in the Verification Cache (more detailed in Section E). In the third iteration, the Dependency Analysis Stage (E) analyses dependencies between iterations. For the current example, the DSA verifies that there is no cross-iteration dependency and triggers the Store ID/Execution Stage. Such stage builds SIMD instructions to execute the remaining iterations in the ARM NEON engine. The DSA needs four parameters to generate SIMD instructions: the data type, the loop range, the operation and the ARM NEON execution support. In the example of Figure 4, the parameters are: *float*, 400, *add*, 128-bit wide, respectively. Considering these parameters, for the current example, the DSA generates an instruction equivalent to the *vaddq_f32* instruction of the NEON

architecture. Since the corresponding ARM NEON engine can operate 128 bits in parallel and the float type is a 32-bit wide data, the DSA divides the loop range by the factor four, running the *vaddq_f32* one hundred times, instead of executing a non-vectorizable *add* operation four hundred times.

Considering the DSA analysis over the *non_vectorizable_Loop()* procedure, the Loop Detection and Data Collection stages behave the same as shown in the *vectorizable_Loop()* procedure. However, in the third loop iteration, during the Dependency Analysis Stage (C), a cross-iteration dependency is found ($v[i] = v[i-1] + b[i]$) which breaks the DLP detection process classifying such procedure as non-vectorizable.

B. Loop Detection Stage

Figure 5 shows the steps of the Loop Detection stage during the execution of the first and second iterations of the loop. The left side of Figure 5 shows the instruction trace that contains the memory addresses of instructions (*Inst. Address*) and instructions descriptions (*Instructions*). The right side of the Figure shows the loop detection stage steps (DSA execution).

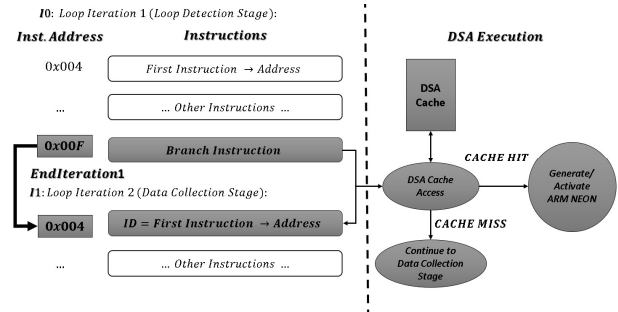


Figure 5 – Loop Detection Stage Behavior

A Branch-type instruction is responsible for triggering the Loop Detection stage. If the first instruction after a Branch (“*First Instruction*” in the example) has a memory address lower than the branch instruction address, the DSA identifies the beginning of a loop, setting its ID as the current value of the Program Counter (PC) register ($ID = Address = 0x00000004$). Whenever DSA detects a loop, the DSA Cache is accessed to verify if there exists a cache entry with the value of the current loop ID. If DSA has already evaluated this loop as vectorizable, the DSA Cache would contain the ID (*CACHE HIT*), which triggers the ARM NEON execution (*Generate/Activate ARM NEON*). However, if the ID of the loop is not found at the DSA Cache (*CACHE MISS*), the Data Collection stage is triggered.

C. Data Collection Stage

Figure 6 illustrates the behavior of the Data Collection Stage, which is triggered after the Loop Detection Stage. Similar to Figure 5, the left side of Figure 6 shows the instruction trace that contains: the memory addresses of instructions (*Inst. Address*) and instructions descriptions (*Instructions*). The right side of the Figure shows the corresponding Data Collection stage steps (DSA execution).

In the second loop iteration, the Data Collection gathers the addresses of data memory accesses (*MemRead (load)* and *MemWrite (store)* type instructions) performed in the Loop execution (*Gather Memory Address*) and stores them in the Verification Cache (*MemRead* → $Address = 0x100$ and *MemWrite* → $Address = 0x108$).

In addition, the Data Collection stage identifies the number of iterations and the value of increment/decrement of the loop. The number of iterations is calculated with the support of the *Cmps* instruction ($Address = 0x00F$), which contains, as operands, the *increment/decrement* and *limit_value*.

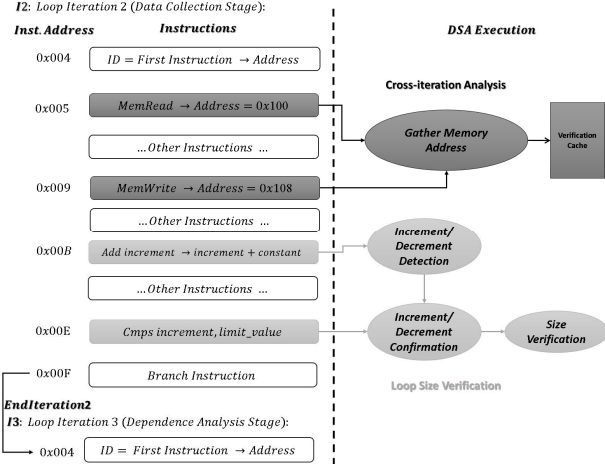


Figure 6 - Data Collection Stage Behavior

This stage also identifies function calls inside the loop by verifying the memory address gap between instructions fetched from memory. As it can be seen in Figure 7, the execution of a *Jump* instruction preceded by a context switch process (*Instructions -> Save Context and Load Context*) indicates a function call (*Jump1 -> Begin and Jump2 -> End*). To be sure that the *Jump* instruction considers a function call, the DSA verifies if the target address of the *Jump* is out the loop body addressing ($0x011 \rightarrow 0x020$). In the example of Figure 7, the loop body comprehends memory addresses from $0x004$ to $0x00F$ while the function call comprehends addresses from $0x011$ to $0x020$. The detection of function calls is mandatory to analyze cross-iteration dependencies since the increment/decrement register can be modified for an operation inside a function call.

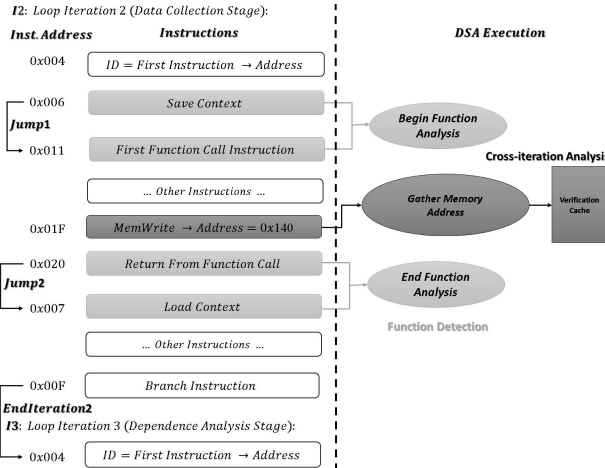


Figure 7 - Data Collection Stage

D. Dependency Analysis Stage

Figure 8 illustrates the behavior of the Dependency Analysis stage during the execution of the 3rd loop iteration. As shown in the previous subsection, the Loop Stage analysis collects data in the second loop iteration to support cross-iteration dependency verification.

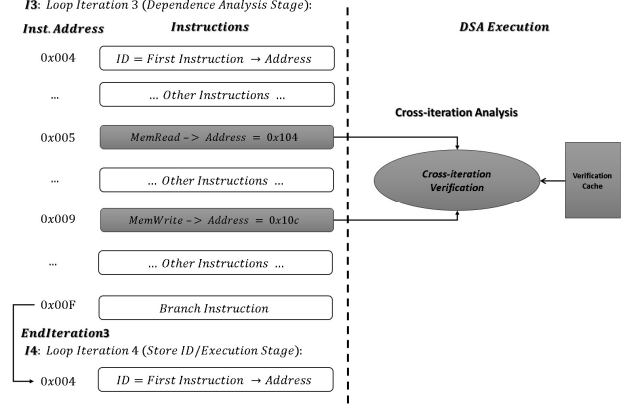


Figure 8 - Dependency Analysis Stage

If any of the data memory addresses (stored in the Verification Cache by the Data Collection Stage) matches with a data memory address accessed in the third loop iteration, a cross-iteration dependency is detected, and the loop cannot be vectorized. However, if data memory addresses stored in the second loop iteration does not match with the addresses performed in the third loop iteration, one cannot discard cross-iteration dependencies in future iterations. Assuming such possibility, we have implemented a cross-iteration dependency prediction process (*Cross-iteration Prediction*). The equations below describe such process, where $M_{Read[2]}$ and $M_{Read[3]}$ is the memory address accessed by a *MemRead* (load) instruction in the second and third loop iterations, respectively. $M_{Read[lastIteration]}$ is the memory address accessed by a load instruction in the last iteration (Equation d), x is the interval between $M_{Read[2]}$ and $M_{Read[lastIteration]}$ (Equation a), $M_{Write[2]}$ is the memory address accessed by a *MemWrite* (store) instruction in the second iteration (Equations b and c), M_{Range} is the memory address range between the $M_{Read[2]}$ and $M_{Read[3]}$ (Equation e), *CID* means Cross-Iteration Dependency and *NCID* means No Cross-Iteration Dependency.

$$M_{Read[3]} \leq x \leq M_{Read[lastIteration]} \quad (a)$$

$$M_{Write[2]} \in x \rightarrow CID \quad (b)$$

$$M_{Write[2]} \notin x \rightarrow NCID \quad (c)$$

$$M_{Read[lastIteration]} = M_{Read[2]} + (M_{Gap} * (lastIteration - 2)) \quad (d)$$

$$M_{Range} = |M_{Read[3]} - M_{Read[2]}| \quad (e)$$

As shown in equations, if the $M_{Write[2]}$ is within the memory address range of $M_{Read[3]}$ and $M_{Read[lastIteration]}$ (Equation b), the loop could have a cross-iteration dependency since the load instruction of a future loop iteration could perform a memory access in the same memory address of the store instruction executed in the second loop iteration. The memory address of the load instruction of the last iteration is predicted based on the sum of the $M_{Read[2]}$ and the equation $(M_{Gap} * (lastIteration - 2))$ (Equation e). Thus, in case of $M_{Write[2]}$ is out of the memory address interval of $M_{Read[3]}$ and $M_{Read[lastIteration]}$ (Equation c), one can ensure that the loop has no cross-iteration dependency.

Figure 9 illustrates an example of how *Cross-iteration Prediction* works. In such example, the DSA detects that there is no cross-iteration dependency between 2nd and 3rd iteration. Thus, by the end of the 3rd loop iteration, the *Cross-iteration Prediction* is activated applying Equation "e" resulting in $M_{Gap} = |0x104 - 0x100| = 0x004$. This gap is used to calculate the memory address of the load instruction of the last iteration ($M_{Read[lastIteration]}$) using Equation "d". Such

memory address results $M_{Read[lastiteration]} = 0x100 + 0x020 = 0x120$. By applying the Equations “a” and “b”, the *Cross-iteration Prediction* detects that the $M_{Write[2]} = 0x108$ is within the range of the addresses accessed by the *MemRead (load)* in the 3rd and last (10th) iterations ($M_{Read[3]} \leq x \leq M_{Read[lastiteration]} = 0x100 \leq x \leq 0x120$), which causes a cross-iteration dependency.

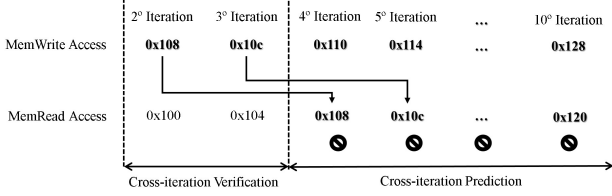


Figure 9 – Example of a Cross-iteration Dependency Prediction Process

E. Store ID/Execution Stage

Figure 10 illustrates the behavior of the Store ID/Execution Stage during the execution of 4th loop iteration, which is just triggered if the loop is vectorizable. In this stage, the loop ID and the number of iterations are saved (*Store ID*) in the DSA cache. In addition, the corresponding SIMD instruction is generated to be executed in the NEON Data Engine (Generate / Activate ARM NEON).

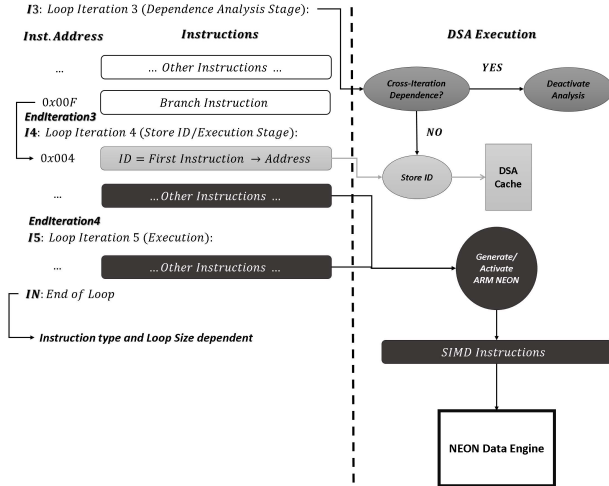


Figure 10 - Store ID/Execution Stage Behavior

As explained before, The Store ID/Execution Stage uses four parameters, collected in the previous stages, to generate a SIMD instruction: the data type, the loop range, the operation, and the ARM NEON SIMD support. Figure 11 shows the different degrees of parallelism that can be obtained through the NEON 128-bit Engine depending on the type of data involved in the SIMD Instruction.

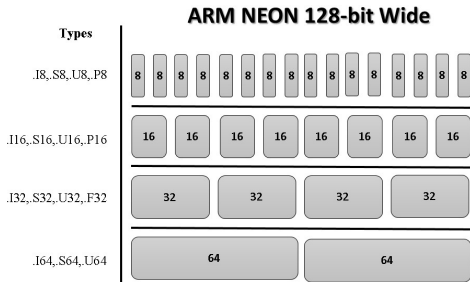


Figure 11 – ARM NEON Parallelism

V. RESULTS

A. Methodology

To evaluate the effectiveness of the proposed approach, we have coupled the DSA to the ARM Cortex A12 (ARMv7 ISA) O3 model of gem5 [9] simulator. To gather performance results, we have compared the proposed technique with an ARM Cortex A12 processor without DLP exploitation (ARM Original Execution) and with an ARM Cortex A12 processor coupled to NEON architecture (ARM NEON AutoVec) exploiting DLP through the support of ARM NEON auto-vectorization compiler. It is important to notice that we employ the same ARM NEON architecture in both ARM NEON AutoVec and ARM Dynamic SIMD Assembler, which provides the same DLP exploitation degree. Table 2 shows the configurations of ARM Original Execution, ARM NEON Dynamic SIMD Assembler, and ARM NEON AutoVec.

Table 2 – Systems Setups

	ARM Original Execution	ARM NEON Dynamic SIMD Assembler	ARM NEON AutoVec.
Processor	ArmCortex-a12	ArmCortex-a12	ArmCortex-a12
Superscalar Width	2-wide	2-wide	2-wide
CPU Clock	1 GHz	1 GHz	1 GHz
L1 Cache	64 kb	64 kb	64 kb
L2 Cache	512 kb	512 kb	512 kb
Cache Policy	LRU	LRU	LRU
Parallelism (NEON)	-	Type Dependent 128-bit Wide	Type Dependent 128-bit Wide
DSA Cache	-	8 kb	-
Verification Cache	-	1 kb	-

We have chosen benchmarks that cover three different levels of DLP to evaluate the systems shown in Table 2. We have selected three applications with a great opportunity to exploit DLP (MM [17], RGB-Gray [18], and Gaussian Filter [18]), an application with a medium opportunity to exploit DLP (Susan E [17]) and two applications with low opportunity to exploit DLP (Q Sort [17] and Dijkstra [17]). Finally, we used RTL Compiler [10] software from Cadence to gather results about area from the VHDL description of the ARM processor and Dynamic SIMD Assembler.

B. Performance

Figure 12 shows the performance improvements of the ARM NEON DSA and ARM NEON AutoVec over the ARM Original Execution. As it can be noticed, when the applications provide great opportunities to exploit DLP, the proposed technique achieves up to 61% of performance improvements over the ARM Original Execution. In cases of Gaussian Filter e Susan Edges, the performance gains are lower since such applications provide smaller opportunities to exploit DLP. The proposed approach does not show performance improvements on Quicksort and Dijkstra execution since they do not provide opportunities to exploit DLP. On average, the DSA provides performance improvements of 31% over the ARM Original Execution showing the importance of DLP exploitation.

The proposed approach outperforms ARM auto-vectorization in all benchmarks but MM 64x64. Due to the dynamic DLP analysis of DSA, it achieves 20% of performance improvements over ARM auto-vectorization technique when executing RGB-Gray. For applications with low DLP exploitation opportunities, our proposal maintained the same performance of the ARM Original Execution since DSA does not cause performance penalties when loops are not found. In such scenario, ARM NEON auto-vectorization provides performance penalties of 3% on Dijkstra and 1% on Q Sort. In addition, besides achieving 20% of performance improvements over the ARM auto-vectorization technique, the proposed approach keeps software

compatibility and does not affect the SW development life cycle due to its transparent and dynamic DLP detection.

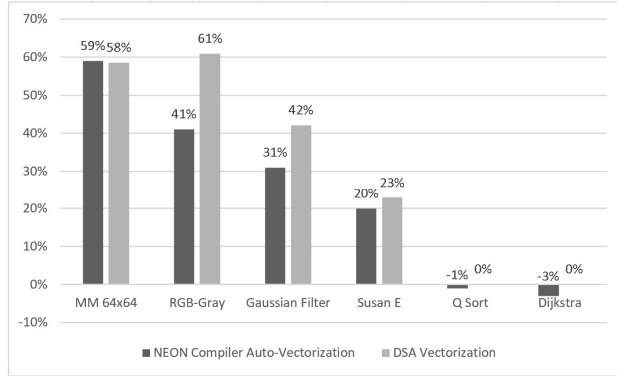


Figure 12 – NEON Auto-Vectorization vs. DSA Vectorization Performance

C. Area

Table 3 shows the area occupied by the ARM processor and the Dynamic SIMD Assembler (DSA). As it can be noticed, the logic to implement the DSA detection is just 2.18% of the ARM core. Considering the DSA and Verification Cache Memories, the total area overhead of the DSA system is 10.37%.

Table 3 – Area overhead of DSA

	Cell Area(um)	Net Area(um)	Total Area(um)
ARM Core	391158	219015	610173
DSA	8667	4607	13274
Area Overhead	2,22%	2,10%	2,18%
	Cell Area(um)	Net Area(um)	Total Area(um)
ARM Core + Caches	512912	279801	792713
DSA + Caches	53716	28520	82236
Total Area Overhead	10,47%	10,19%	10,37%

VI. CONCLUSION AND FUTURE WORK

In this work, we have proposed a transparent Dynamic SIMD Assembler (DSA) that is capable of detecting vectorizable code regions at runtime without requiring specific libraries or compilers. The proposed approach shows performance improvements of 31% over the original execution (without DLP exploitation). In addition, Dynamic SIMD Assembler, besides keeping binary compatibility, outperforms ARM auto-vectorization technique in 6% by increasing 10.37 % of the chip area. Since the Dynamic SIMD Assembler is an in-progress work, we intend the current version to support: vectorization of loops with conditional statements; partial vectorization of loops with cross-iteration dependencies; vectorization of outer-loops with data dependencies with inner-loops; vectorization of loops with dynamic range.

ACKNOWLEDGEMENT

We are grateful to the institutions listed below for the direct or indirect support that they are providing us: Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) e Fundação de Amparo à pesquisa do Estado do Rio Grande do Sul (FAPERGS).

REFERENCES

- [1] Baghsorkhi, Sara S., Nalini Vasudevan, and Youfeng Wu. "FlexVec: Auto-vectorization for irregular loops." *ACM SIGPLAN Notices*. Vol. 51. No. 6. ACM, 2016.
- [2] Nuzman, Dorit, and Ayal Zaks. "Outer-loop vectorization-revisited for short SIMD architectures." *Parallel Architectures and Compilation Techniques (PACT), 2008 International Conference on*. IEEE, 2008.
- [3] Tian, Xinmin, et al. "Compiling C/C++ SIMD extensions for function and loop vectorization on multicore-SIMD processors." *Parallel and Distributed Processing Symposium Workshops & Ph.D. Forum (IPDPSW), 2012 IEEE 26th International*. IEEE, 2012.

- [4] Chang, Hoseok, and Wonyong Sung. "Efficient vectorization of SIMD programs with non-aligned and irregular data access hardware." *Proceedings of the 2008 international conference on Compilers, architectures, and synthesis for embedded systems*. ACM, 2008.
- [5] SAIT RP Core Group. SRP SDK User Guide. Samsung Advanced Institute of Technology internal document (available by SAIT SRP SDK distribution program), 2011.
- [6] Reddy, Venu Gopal. "Neon technology introduction." *ARM Corporation* (2008).
- [7] Lomont, Chris. "Introduction to Intel advanced vector extensions." *Intel White Paper* (2011): 1-21.
- [8] Diefendorff, Keith, et al. "AltiVec extension to PowerPC accelerates media processing." *IEEE Micro* 20.2 (2000): 85-95.
- [9] Binkert, Nathan, et al. "The gem5 simulator." *ACM SIGARCH Computer Architecture News* 39.2 (2011): 1-7.
- [10] Cadence, R. T. L. "Compiler User's Manual."
- [11] Stephan Nolting (2012), Storm Core Processor System [Online]. Available at www.opencore.org. Access on 16 March 2016.
- [12] Mitra, Gaurav, et al. "Use of SIMD vector operations to accelerate application code performance on low-powered ARM and Intel platforms." *Parallel and Distributed Processing Symposium Workshops & Ph.D Forum (IPDPSW), 2013 IEEE 27th International*. IEEE, 2013.
- [13] Swanson, K. Michelson, A. Schwerin and M. Oskin, "WaveScalar," *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36., 2003*, pp. 291-302.
- [14] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello and Y. LeCun, "NeuFlow: A runtime reconfigurable dataflow processor for vision," *CVPR 2011 WORKSHOPS*, Colorado Springs, CO, 2011, pp. 109-116.
- [15] A. C. S. Beck, M. B. Rutzig, G. Gaydadjiev and L. Carro, "Transparent Reconfigurable Acceleration for Heterogeneous Embedded Applications," *2008 Design, Automation and Test in Europe, Munich, 2008*, pp. 1208-1213.
- [16] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov and E. M. Panainte, "The MOLEN polymorphic processor," in *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1363-1375, Nov. 2004.
- [17] Guthaus, Matthew R., et al. "MiBench: A free, commercially representative embedded benchmark suite." *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*. IEEE, 2001.
- [18] Bradski, Gary, and Adrian Kaehler. "OpenCV." *Dr. Dobb's journal of software tools* 3 (2000).

**7 ARTICLE 2 - RUNTIME VECTORIZATION OF CONDITIONAL CODE
AND DYNAMIC RANGE LOOPS TO ARM NEON ENGINE**

Runtime Vectorization of Conditional Code and Dynamic Range Loops to ARM NEON Engine

Michael Guilherme Jordan, Tiago Knorst, Julio Vicenzi and Mateus Beck Rutzig
Electronics and Computing Department - Federal University of Santa Maria – Santa Maria – Brazil
{michael.jordan, tiago.knorst, julio.vicenzi}@ecomp.ufsm.br; mateus@inf.ufsm.br

Abstract — SIMD engines are widely present in market processors aiming to improve performance of applications through Data Level Parallelism (DLP) exploitation. However, most SIMD engines rely on specific libraries and compilers to support DLP execution, which limits DLP gains since they are restricted to analyze static code. Dynamic SIMD Assembler (DSA) [8] is capable of exploiting DLP at runtime by identifying vectorizable loops to generate ARM NEON SIMD instructions. However, its DLP coverage capability is not fully exploited, since portion of code that depends on runtime information, such as dynamic range and conditional code loops are not exploited. In this work, we extend the DSA coverage by coupling the exploitation of conditional code and dynamic range loop vectorization. Results show that the proposed techniques improve the original DSA performance in 38% considering benchmarks with opportunities to exploit conditional code and dynamic range loops. In addition, the Extended DSA, besides keeping software productivity and binary compatibility, outperforms ARM compiler auto-vectorization by 12%.

Keywords— *DLP, SIMD, Vectorization, ARM*

I. INTRODUCTION

SIMD (Single Instruction Multiple Data) engines, such as ARM NEON [7], Intel SSE/AVX [9] and IBM AltiVec [10], are widely present in market processors to improve applications performance by exploiting Data Level Parallelism (DLP). To exploit the potential of vector processing engines most techniques rely on specific libraries to vectorize the application code. However, it is hard to predict at software development time the behavior of certain vectorizable regions of code, making it unfeasible to extract the maximum available DLP using such techniques. In addition, the use of specific libraries increases the development process lifecycle and affects software productivity.

Automatic code vectorization techniques [15] extract DLP from vectorizable code regions by building SIMD instructions to exploit vector processing engines at compile time. However, although keeping software productivity by avoiding the use of specific libraries, auto-vectorization techniques still rely on code recompilation which breaks binary compatibility. Also, such an approach is restricted to static code exploitation [12], which, similar to the employment of specific libraries, restricts the performance gains due to the low DLP coverage.

DLP opportunities are mostly present in application loops, where operations are executed multiple times over vector structures. However, three issues prevent loop vectorization: cross-iteration dependences; loops with conditional codes and dynamic range loops. Cross-iteration dependences are intrinsically non-vectorizable due to the data dependencies between iterations. However, instead of having data dependencies, loops with conditional codes and dynamic range loops present control dependencies which rely on runtime information to be vectorized. Thus, they cannot be handle through special libraries and compiler auto-vectorization techniques.

Dynamic SIMD Assembler (DSA) [8] is capable of exploiting DLP at runtime by identifying vectorizable loops to generate SIMD instructions. Unlike most market SIMD engines, due to its

transparent fashion, the development process life cycle is maintained since it does not rely on specific libraries. As SIMD instructions are built at runtime, unlike automatic code vectorization techniques, binary compatibility is also maintained. Moreover, DSA has already shown higher performance than ARM static auto-vectorization approach since it is capable of vectorizing count loops (static range loops), such as inner-loops and outer-loops and loops containing function calls at runtime [8]. However, its DLP coverage capability is not fully exploited, since conditional code and dynamic range loops are not vectorized.

In this work, we extend DLP exploitation capability of the DSA by taking advantage of its intrinsic and transparent execution that makes it possible to evaluate loops during runtime. By analyzing code during runtime it is possible to extract DLP from dynamic behavior loops. Thus, this work extends the DSA coverage by proposing the exploitation of: Conditional Code and Dynamic Range Loop vectorization. We show that the extended DSA approach outperforms the auto-vectorization compiler in 12%, on average, maintaining software productivity and binary compatibility.

This work is organized as follows; Section II presents the Related Work. Section III and IV present the Dynamic SIMD Assembler System. Section V presents the Conditional Code Loop and Dynamic Range Loop vectorization techniques. Methodology and Results are shown in Section VI. Finally, we present the conclusion and future works in Section VII.

II. RELATED WORK

The SIMD vectorization is widely used in several emerging market platforms, such as the Intel AVX, IBM AltiVec, and ARM NEON architectures. In the academic field, several researches are exploiting Data Level Parallelism (DLP) to achieve performance improvements and energy savings. Sui Yulei [1] improves the LLVM compiler [16] infrastructure to explore vectorization opportunities by developing a more precise Loop-Oriented Pointer Analysis for Automatic SIMD Vectorization. This approach is able to detect more than 273 vectorizable basic blocks achieving performance improvements from 2.95% to 7.23%. However, such an approach uses an auto-vectorization technique, which means that loops containing dynamic behavior cannot be vectorized.

Zhou Hao [2] presents the Loop-Mix compiler, also implemented in the LLVM compiler. Loop-Mix vectorizes loops regarding the data reorganization overhead caused between mixed SIMD parallelism (inter-loops and intra-loops). The technique outperforms the Loop-ILV [4] by 36%. Since the work is also implemented in the LLVM compiler, the binary compatibility is compromised, code recompilation is required and dynamic behavior loops are not covered.

Sara S. Baghsorkhi [3] proposes FlexVec architecture that combines a novel partial vector code generation technique with new vector instructions to dynamically adjust vector length for loop statements affected by runtime cross-iteration dependencies. FlexVec vectorization coupled to the Intel AVX-512 ISA shows a Geomean performance improvement from 9% to 11%. Although it is able to perform optimizations over loops with cross-iteration dependencies, it is not capable of vectorize Dynamic Range loops. Besides, the method breaks binary compatibility since it is necessary

a specific ISA adjustment and also relies on a particular compiler and library development.

Dorit Nuzman [4] proposes a compiler outer loop vectorization technique focusing on properties of modern SIMD architectures. It shows that even though current optimizing compilers do not apply outer loop vectorization, they can provide significant performance improvements over innermost loop vectorization. It shows performance improvements of 3.13 and 2.77 when coupled to a Cell BE SPU and PowerPC970, respectively. Similar to our proposal, the authors focused on vectorizing both innermost and outer loops but it relies on compiler support and cannot cover loops with dynamic range or conditional code.

Being aware that most research focuses on vectorizing loops, Tian Xinmin [5] presented a set of new C/C++ high-level extensions for SIMD programming capable of automatic translating both functions and loops. Significant speedups (from 3.07x to 4.69x) are achieved when these optimizations are applied. Similar to aforementioned related works, dynamic behavior loops are not covered and it relies on specific compiler and library to achieve performance improvements, which breaks binary compatibility and affects SW productivity.

Bramas Berenger [6] proposes Inastemp, a lightweight open-source C++ library that provides portable SIMD-Vectorization. This approach has the same efficiency as computing for a specific architecture, providing vector instructions that can be used to develop hardware-independent computational kernels. These computational kernels are portable across compilers. Inastemp covers SSE, AVX, AVX512 and ALTIVEC/VMX instructions. While such technique improves binary portability, it compromises software productivity since code must be adapted with the suggested library and requires code recompilation. In addition, no performance gains are shown by using such technique.

ARM NEON [7] is introduced in the ARMv6 architecture. The NEON auto-vectorization compiler generates vectorizable code by instantiating SIMD instructions. Despite the advantages of auto-vectorization, the static code exploitation limits the performance gains since it is difficult to identify vectorizable regions of conditional statements, function calls or even loops that contain codes between inner-loops and outer-loops. To overcome such issues, another strategy offered by the ARM to explore the NEON engine is the use of ARM NEON library, which transfer the vectorization task responsibility to the SW developer, which affects SW productivity. Since both solutions evaluate static code, they cannot vectorize dynamic behavior loops.

The Dynamic SIMD Assembler (DSA) [8] is capable of building SIMD instructions at runtime. The DSA is coupled to the ARM NEON engine, providing: higher performance than ARM auto-vectorization method with binary compatibility since it is not necessary to recompile the source code; SW productivity by avoiding the use of specific libraries in the code development lifecycle to take advantage of the NEON engine processing capabilities. This approach is capable of vectorizing count loops, such as inner-loops and outer-loops and loops containing function calls at runtime. However, the runtime capabilities still opens room for vectorizing dynamic behavior loops, which are not supported in DSA implementation.

Table 1 compares all the aforementioned works with the proposed approach. As it can be seen, except DSA, binary and software compatibility are not prioritized in all designs since they employ ISA modification or specific libraries. Both auto-vectorization and techniques that use specific libraries cannot vectorize loops that depend on data computed at runtime (dynamic behavior loops) which limits their vectorization coverage. The extension of the DSA benefits of the runtime capabilities, expanding its DLP exploitation by proposing the vectorization of:

- Dynamic Range Loop;
- Conditional Code Loop.

By applying both dynamic behavior loops vectorization techniques, the proposed approach presents performance

improvements that outperforms the auto-vectorization compiler and preserves software productivity and binary compatibility as well.

Table 1 – Related Works and Proposed Technique Characteristics

	Code Recompilation	Library Development Support	ISA Modification	SW Productivity	Binary Compatibility	Dynamic Behavior Loops Support
[1]	Yes	No	No	Not Affected	No	No
[2]	Yes	No	No	Not Affected	No	No
[3]	Yes	Yes	Yes	Affected	No	No
[4]	Yes	No	Yes	Not Affected	No	No
[5]	Yes	Yes	Yes	Affected	No	No
[6]	Yes	Yes	No	Affected	Yes	No
[7]	Yes	Yes/No	Yes	Affected	No	No
[8]	No	No	No	Not Affected	Yes	No
Extended DSA	No	No	No	Not Affected	Yes	Yes

III. DSA OVERVIEW

The Dynamic SIMD Assembler (DSA) is tightly coupled to an ARM Cortex-A12 processor (ARMv7 ISA). Figure 1 shows the system overview. As it can be seen, the DSA is composed of a SIMD instruction logic detection and two cache memories (*DSA Cache and Verification Cache*). The DSA Cache is responsible for storing information about the built SIMD instructions over the vectorizable loops. The Verification Cache stores the addresses of data memory accesses performed into the vectorizable loops (more details about caches in Section IV).

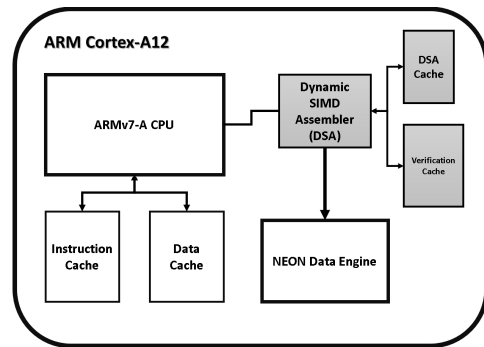


Figure 1 – System Overview

Figure 2 shows an overview of how the DSA works. In the first scenario (*Scenario 1 – Ordinary Execution*), the DSA and ARMv7 processor operate in parallel. While the ARM Cortex-A12 processor executes the incoming instructions, the DSA is in a probing mode, searching for a vectorizable loop to build SIMD instructions. In such execution mode, the NEON Engine remains deactivated. If the DSA detects a vectorizable loop, the second scenario is triggered (*Scenario 2 – DLP Exploitation*). In this scenario, the DSA deactivates the ARM Cortex A12 processor and activates the NEON Data Engine to execute the built SIMD instruction.

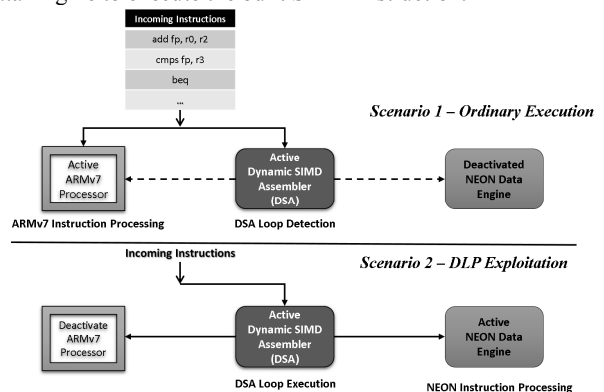


Figure 2 – System Functionality Overview

IV. DYNAMIC SIMD ASSEMBLER

This section is divided into two subsections. Subsection A shows a superficial analysis of the Dynamic SIMD Assembler (DSA).

Subsection B shows how the DSA predicts cross-iteration dependencies.

A. Dynamic SIMD Assembler Overview

As shown in Figure 3, the Dynamic SIMD Assembler (DSA) detection process is based on a State Machine (SM) composed of four stages: Loop Detection, Data Collection, Dependency Analysis, and Store ID/Execution. Each one of these stages is activated in different loop iterations.

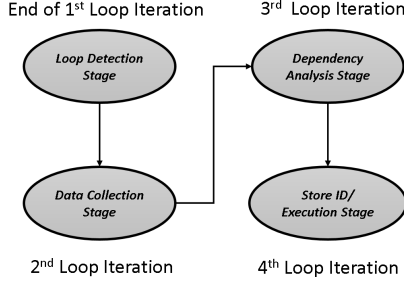


Figure 3 – State Machine of DSA

As it can be seen, the state machine starts in the Loop Detection stage and is triggered by the end of the first loop iteration. The Loop Detection stage is responsible for:

- checking the existence of innermost-loop and outer-loops at runtime;
- accessing the DSA cache, checking if the current loop is already vectorizable.

The Data Collection stage is triggered in the second loop iteration. This stage is responsible for:

- evaluating the loop range (number of iterations)
- identifying the existence of a function call inside the loop;
- storing the addresses of data memory accesses in the Verification Cache.

The Dependency Analysis stage is triggered in the third loop iteration. This stage is responsible for:

- analyzing the cross-iteration dependency (dependencies between two or more iterations in the same loop).

The Store ID/Execution stage is triggered in the fourth loop iteration. This stage is responsible for:

- generating and saving the loop identification (ID) at DSA Cache in case of a vectorizable loop;
- building SIMD instruction and activating the execution on NEON engine.

Figure 4 exemplifies the execution of the DSA considering a vectorizable loop (*vectorizable_Loop()*) and a non-vectorizable loop procedures (*non_vectorizable_Loop()*).

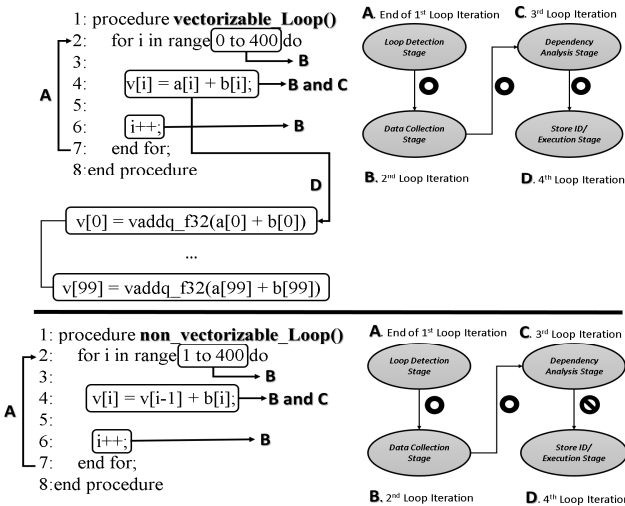


Figure 4 – DSA Execution

Considering the *vectorizable_loop()* procedure, the Loop Detection stage (A) detects the loop by the end of the execution of the first iteration. In the second iteration, the Data Collection stage (B) identifies the loop range (400) and the value of the increment/decrement ($i = i + I$). In addition, such stage stores the addresses of the data memory accesses ($Mem[a[i]]$, $Mem[b[i]]$ and $Mem[v[i]]$) in the Verification Cache. In the third iteration, the Dependency Analysis Stage (E) analyses dependencies between iterations (more detailed in subsection B). For the current example, the DSA verifies that there is no cross-iteration dependency and triggers the Store ID/Execution Stage. Such stage builds SIMD instructions to execute the remaining iterations in the ARM NEON engine. The DSA needs four parameters to generate SIMD instructions: the data type, the loop range, the operation and the ARM NEON execution support. In the example of Figure 4, the parameters are: *float*, *400*, *add*, *128-bit wide*, respectively. Considering these parameters, for the current example, the DSA generates an instruction equivalent to the *vaddq_f32* instruction of the NEON architecture. Since the corresponding ARM NEON engine can operate 128 bits in parallel and the float type is a 32-bit wide data, the DSA divides the loop range by the factor four, running the *vaddq_f32* one hundred times, instead of executing a non-vectorizable *add* operation four hundred times.

Considering the DSA analysis over the *non_vectorizable_Loop()* procedure, the Loop Detection and Data Collection stages behave the same as shown in the *vectorizable_Loop()* procedure. However, in the third loop iteration, during the Dependency Analysis Stage (C), a cross-iteration dependency is found ($v[i] = v[i-1] + b[i]$) which breaks the DLP detection process classifying such procedure as non-vectorizable.

Figure 5 shows the different degrees of parallelism that can be obtained through the NEON 128-bit Engine depending on the type of data involved in the SIMD Instruction.

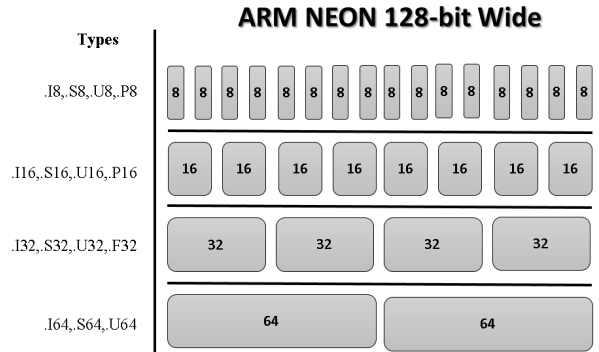


Figure 5 – ARM NEON Parallelism

B. Cross-iteration Dependency Prediction

A cross-iteration dependency exists when any of the data memory addresses accessed in a loop iteration matches with a data memory address accessed in a further loop iteration.

The cross-iteration analysis is processed during the 2nd (Data Collection Stage) and 3rd (Dependency Analysis Stage) loop iterations. During the 2nd iteration all the data memory accesses are saved at the Verification cache. The data saved at Verification Cache is compared with the data accessed at the 3rd iteration. If any of the data memory addresses accessed at the 2nd loop iteration matches with a data memory address accessed in the 3rd loop iteration, a cross-iteration dependency is detected, and the loop cannot be vectorized. However, if data memory addresses of the second loop iteration do not match with the addresses performed in the third loop iteration, one cannot discard cross-iteration dependencies in future iterations. Assuming such possibility, we have implemented a cross-iteration dependency prediction process (*Cross-iteration Prediction*). The equations below describe such process, where $M_{Read[2]}$ and $M_{Read[3]}$ is the memory address accessed by a

MemRead (load) instruction in the second and third loop iterations, respectively. $M_{Read[lastIteration]}$ is the memory address accessed by a load instruction in the last iteration (Equation 4), x is the interval between $M_{Read[2]}$ and $M_{Read[lastIteration]}$ (Equation 1), $M_{Write[2]}$ is the memory address accessed by a *MemWrite (store)* instruction in the second iteration (Equations 2 and 3), M_{Range} is the memory address range between the $M_{Read[2]}$ and $M_{Read[3]}$ (Equation 5), *CID* means Cross-Iteration Dependency and *NCID* means No Cross-Iteration Dependency.

$$M_{Read[3]} \leq x \leq M_{Read[lastIteration]} \quad (1)$$

$$M_{Write[2]} \in x \rightarrow CID \quad (2)$$

$$M_{Write[2]} \notin x \rightarrow NCID \quad (3)$$

$$M_{Read[lastIteration]} = M_{Read[2]} + (M_{Gap} * (lastIteration - 2)) \quad (4)$$

$$M_{Gap} = |M_{Read[3]} - M_{Read[2]}| \quad (5)$$

As shown in equations, if the $M_{Write[2]}$ is within the memory address range of $M_{Read[3]}$ and $M_{Read[lastIteration]}$ (Equation 2), the loop could have a cross-iteration dependency since the load instruction of a future loop iteration could perform a memory access in the same memory address of the store instruction executed in the second loop iteration. The memory address of the load instruction of the last iteration is predicted based on the sum of the $M_{Read[2]}$ and the equation $(M_{Gap} * (lastIteration - 2))$ (Equation 5). Thus, in case of $M_{Write[2]}$ is out of the memory address interval of $M_{Read[3]}$ and $M_{Read[lastIteration]}$ (Equation 3), one can ensure that the loop has no cross-iteration dependency.

Figure 6 illustrates an example of how *Cross-iteration Prediction* works. In such example, the DSA detects that there is no cross-iteration dependency between 2nd and 3rd iteration. Thus, by the end of the 3rd loop iteration, the *Cross-iteration Prediction* is activated applying Equation 5 resulting in $M_{Gap} = |0x104 - 0x100| = 0x004$. This gap is used to calculate the memory address of the load instruction of the last iteration ($M_{Read[lastIteration]}$) using Equation 4. Such memory address results $M_{Read[lastIteration]} = 0x100 + 0x020 = 0x120$. By applying the Equations 1 and 2, the *Cross-iteration Prediction* detects that the $M_{Write[2]} = 0x108$ is within the range of the addresses accessed by the *MemRead (load)* in the 3rd and last (10th) iterations ($M_{Read[3]} \leq x \leq M_{Read[lastIteration]}$) = $0x108 \leq x \leq 0x120$), which causes a cross-iteration dependency.

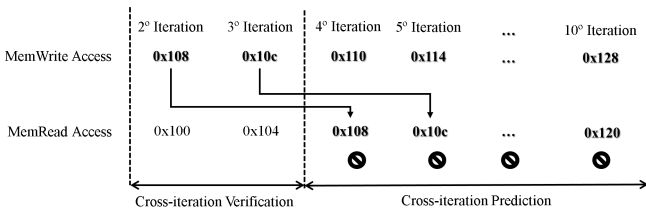


Figure 6 - Example of a Cross-iteration Dependency Prediction Process

V. CONDITIONAL CODE AND DYNAMIC RANGE LOOP ANALYSIS

The compiler does not have enough information to detect and vectorize certain types of loop, especially those whose behavior depends on user inputs. At compile time, Dynamic Range loops cannot be vectorized, since the loop size is required beforehand to allocate a certain number of SIMD operations. Concurrently, Conditional Code Loops can hardly be vectorized at compile time, since the execution of conditional portions are solved at runtime.

As it can be seen in Figure 7, the pseudocode (A) presents a simple vectorizable loop in which both the compiler and DSA would be capable of obtaining DLP. The pseudocode (B) has a dynamic range loop, where this size is determined by an input or even a data calculated at runtime. The pseudocode (C) has a loop containing conditional statements which the execution is also determined at runtime. The same evaluation can be made for the pseudocode (D) which has a loop containing a function call that depends on a variable calculated at runtime. In this way, the pseudocodes (B), (C) and (D) cannot be vectorized by the compiler auto-vectorization techniques since they depend on data manipulation at runtime. However, as the DSA (Dynamic SIMD Assembler) analyzes the application code at runtime, it is able to evaluate all aforementioned situation. Thus, the contribution of this work is the extension of the DSA to vectorize: conditional code and dynamic range loops.

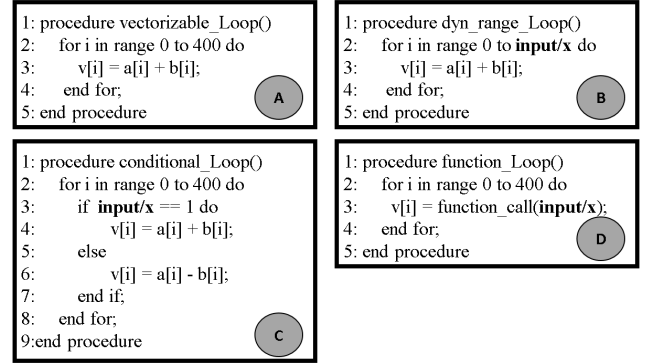


Figure 7 – Vectorizable, Dynamic Range, Conditional Code and Function Loops

A. Vectorizing Conditional Code Loop

Few steps should be added in the original DSA State Machine to support the vectorization of conditional code loop. As it can be seen in Figure 8, during the Dependence Analysis Stage, we have added the Conditional Coverage Stage. The Conditional Coverage Stage is responsible for:

- identifying if the loop body has conditional statements;
- verifying if the identified conditional statements can be vectorized.

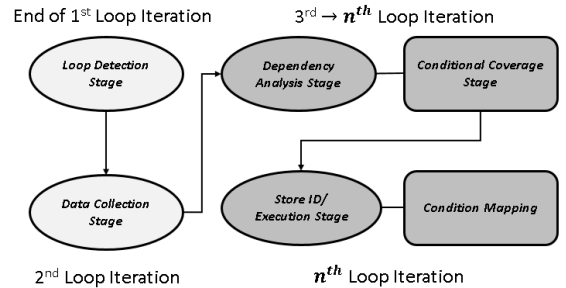


Figure 8 – Conditional Loop DSA State Machine

As it can be seen in Figure 9, the first Conditional Coverage state is the Conditional Code Detection, that occurs during the DSA's Dependence Analysis Stage (3rd Loop Iteration). In this state, we identify if there is a condition statement within the loop body. If there exists, the Conditional Code Analysis is activated. In this state, the DSA checks if the condition of the current iteration is vectorizable. The analyzed condition is then marked as vectorizable or not. If the DSA detects a cross-iteration dependency in this condition, it sets such loop in the DSA Cache as non-vectorizable.

As the conditions are verified during the loop execution, the DSA also counts the number of conditions and classifies them using their instruction addresses (further discussed at Conditional Loop Vectorization Analysis subsection). While there are still pending conditions, the DSA continues searching for them and verifying if

they can be vectorizable. In this way, this step is repeated until all conditions have been verified. If there is no pending condition and all detected conditions are vectorizable, the DSA saves the loop at the DSA Cache as vectorizable. Thus, the remaining iterations can be executed in a vectorized fashion during the DSA Store ID/Execution Stage. Since we cannot predict which condition code portion is executed at the remaining iterations, the DSA performs speculative execution (further discussed at the Conditional Code Loop SIMD Execution subsection).

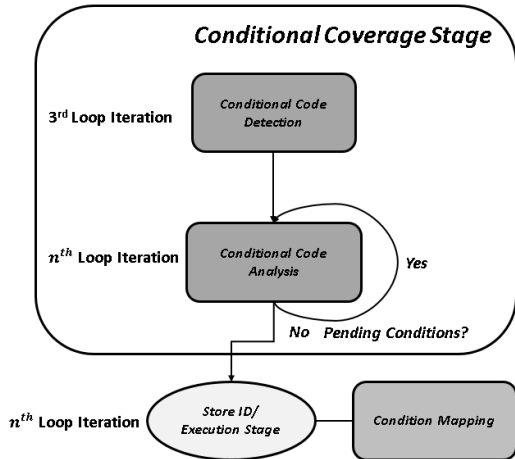


Figure 9 – Conditional Code Coverage Stage

1. Conditional Code Loop Vectorization Analysis

Figure 10 shows an example of a Conditional Code Loop (instruction addresses from 2 to 8) containing two possible conditions (A and B). In addition, the execution timeline of the loop is shown at the bottom of this Figure. During the 1st and 2nd iteration, the DSA’s Loop Detection Stage and Loop Analysis Stage are performed. At the 3rd iteration the Conditional Code Detection is active. The Conditional Code Detection verifies if there is any instruction address gap. As it can be seen in the timeline, an instruction address gap is detected (from 3 → 4 (Condition A)) at the execution of the 3rd iteration (Conditional Code Detection) since the condition B is executed. This gap confirms the existence of conditional statement within the loop. In parallel with the Conditional Code Loop confirmation, the DSA verifies if the Condition B code is vectorizable. At the 4th iteration, the Conditional Code Analysis is activated, there is no need to repeat the vectorization analysis since the Condition B is previously verified (at the 3rd iteration). Considering that there are still pending conditions since addresses from 3 → 4 were not accessed yet, the Conditional Code Analysis state is kept active. At the 5th iteration, the Condition A is executed for the first time, so the DSA verifies if the Condition A code is vectorizable. At the 6th iteration, the DSA detects that there is no pending condition to analyze since all instruction addresses were accessed (2 → 8) and conditions A and B are vectorizable, the remaining loop iterations can be executed as SIMD fashion.

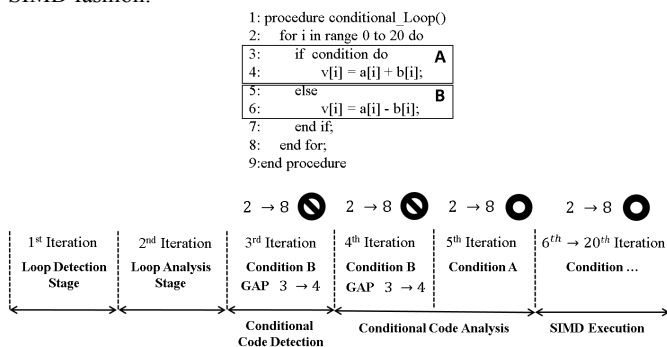


Figure 10 – Conditional Code Loop Vectorization Analysis

To verify if all condition has been analyzed, an instruction address mapping becomes necessary. Figure 11 illustrates the analysis mapping considering the example shown in Figure 10. At the 3rd iteration, the condition B is executed, its instructions are analyzed and classified as vectorizable. The DSA indexes the condition by the address of its first instruction, this information is stored into a temporary vector map. For instance, the Condition B is indexed by the address 5. At the 4th iteration, the DSA has identified that the Condition B has already been analyzed by comparing the stored ID with the instruction addresses executed in this iteration. At the 5th iteration, condition A is executed, its instructions are analyzed, classified as vectorizable and stored in the vector map indexed by its first address (in the case 3). As all instructions in the loop instruction address range (2 → 8) were executed and analyzed, at the 6th iteration we store the loop information in the DSA Cache. This information is necessary to further vectorizing the loop without repeating the vectorization analysis. The information is composed of:

- Loop ID: to identify the vectorizable loop during the program execution;
- Loop Size: to generate SIMD instructions during execution;
- Conditions ID: necessary to make the speculative execution (further discussed at Conditional Code Loop SIMD Execution).

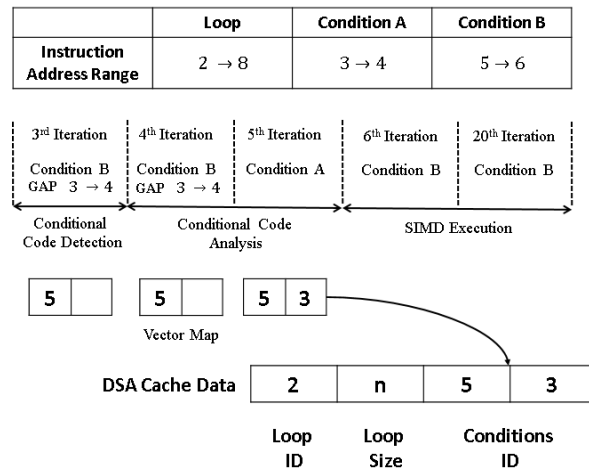


Figure 11 – Conditional Code Loop Analysis Mapping and Data Storage

2. Conditional Code Loop SIMD Execution

Figure 12 shows the SIMD Execution considering the example shown in Figure 10. Therefore, at the 6th iteration, the condition B is executed. Since is the first time the condition B is executed during SIMD Execution, its instructions are vectorized. Since the condition B is executed at the 6st iteration, its operations are vectorized considering the range (Vectorize B - 6 → 20), generating preemptive results to 14 iterations (B - RESULTS). In parallel, its execution is mapped into a vector map (6th Iteration - B) to further select the results produced by each condition (speculative execution). In the 7th iteration, condition B is executed again. Since this condition has already been vectorized at the 6th iteration, its instructions are not executed (Idle), and only the mapping is performed (7th Iteration - B). In the 8th iteration, condition A is executed. As it is the first time condition A is executed during SIMD Execution, its instructions are vectorized considering the range of the current iteration until the end of the loop (Vectorize A - 8 → 20 iterations), generating preemptive results to the 12 remaining iterations (A - RESULTS). In parallel, its execution is mapped in the Vector Map (8th Iteration - A). The 6th and 7th iterations are not considered in this vectorization since they executed condition B. At the 9th iteration, condition B is executed. Because it has already been executed, condition B is only mapped

(Idle) in Vector Map (9th Iteration - B). At last iteration (20th iteration) condition A is executed again and because it has already been executed it is only mapped (Idle) (20th Iteration - A). At the end of the loop, we use the Vector Map to select only the mapped results, while the others are discarded.

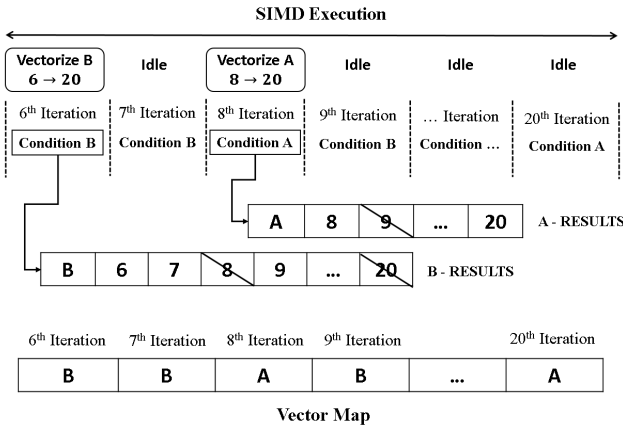


Figure 12 – Conditional Code Loop SIMD Execution

B. Vectorizing Dynamic Range Loop (DRL)

As shown in Figure 13, there are two types of Dynamic Range loops. In the Type A (DRLA), the loop size is determined by the user input or even by a variable handled at runtime, before of the loop execution. The vectorization size of such example is unfeasible to be determined at compile time. However, DSA can vectorize such DRL since such value is available at runtime. In the Type B (DRLB), the loop size or the loop stop condition is determined in the body of the loop. In this case, it is unfeasible to determine the vectorization size at both compile time and runtime. In this case, the DSA uses a speculative execution to vectorize this type of loop.

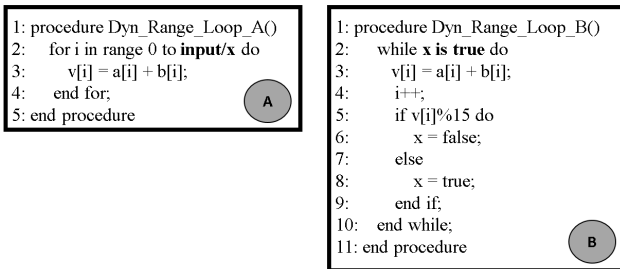


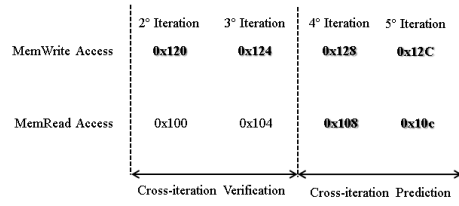
Figure 13 – DRL Type A, DRL Type B

1. DRLA - size calculated before loop execution

Since the DRLA size is calculated before the loop execution, the loop can be analyzed maintaining the original DSA state machine. However, the DRLA must be analyzed every time it repeats, since the Dependence Analysis Stage needs to verify if the vectorization is feasible based on loop range.

As it can be seen in Figure 14, at the 1st time the loop executes, the Dependence Analysis Stage cannot detect any cross-iteration dependence (refer to the Section “Cross-iteration Dependency Prediction”). Thus, considering the loop range 5, the DSA Loop Analysis predict the loop as vectorizable. However, at the 2nd time the loop executes, a cross-iteration dependence is detected at the 10th iteration (MemRead Access = 0x120 = MemWrite Access). Such an example shows that different loop sizes imply in different DSA Loop Analysis.

1st Loop Analysis – Actual Range - 5



2nd Loop Analysis – Actual Range - 10

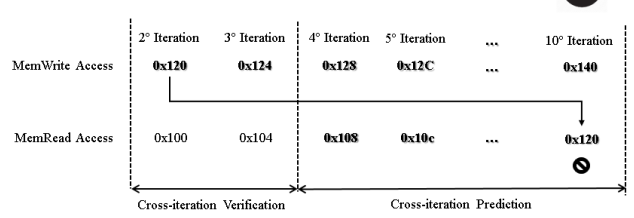


Figure 14 – DRLA Cross-iteration Analysis

2. DRLB - size calculated during loop execution

Since the DRLB size or stop condition is calculated during the loop execution, it is unfeasible for the DSA to determine the number of times the loop would execute. As a way to vectorize the DRLB the DSA:

- speculates the number of times the loop will execute based on the last loop execution;
- verifies cross-iteration dependency based on the speculative value every time the loop executes.

Figure 15 shows the DRLB analysis and execution. During the 1st Loop Analysis, the 2nd and 3rd iterations are responsible for verifying if there is a cross-iteration dependency in the loop. Since a loop range is necessary to predict a cross-iteration dependency and there is no defined range in the DRLB, the DSA assumes a speculative range. Thus, the DSA chooses a loop range that maximizes vector units’ utilization. In this example, the DSA assumes a 128-bit wide ARM NEON, since the instruction operands widths are 8 bits (8 bits operands), the DSA chooses a Speculative Range of 16 (Speculative Range - 16) in order to use all vector units. In this way, at the 3rd iteration (Dependency Analysis Stage), the DSA predicts that there is no cross-iteration dependency considering the range 16 (Cross-iteration Prediction) and the loop can be vectorized. At the 1st Loop Execution (Execution Stage – 4th iteration), the DSA executes the loop operation considering the speculative loop range (Vectorize - 4 → 20). From the 5th iteration to the 10th, the already vectorized operations are not executed (Idle) and the only instructions that are processed in the loop are those responsible for the stopping condition calculation. When the stop condition is reached (10th iteration), the results from iterations (4 → 10) are kept, while the operation results (11 → 20) are discarded. Since the current loop execution has 10 iterations, on the next execution the speculative range value is 16, since is the minimum operation range to be allocated at the vector units considering an operand width of 8 bits. At the second time this loop is detected by the DSA which, in the DSA Loop Analysis Stage, predicts that there is no cross-iteration dependency. At the 4th iteration (Execution Stage), the DSA executes the loop operation considering the speculative loop range (Vectorize - 4 → 20). From the 5th iteration to the 16th iteration, the already vectorized operations are not executed. This time the loop executes until the 18th iteration (Real Range - 18). Despite the DSA computed results from 4th to the 20th iteration, only the results from 4th to 16th are considered, since the cross-iteration prediction was based on the range 16. The operations of the 17th and 18th are sequentially executed by the ARM Processor.

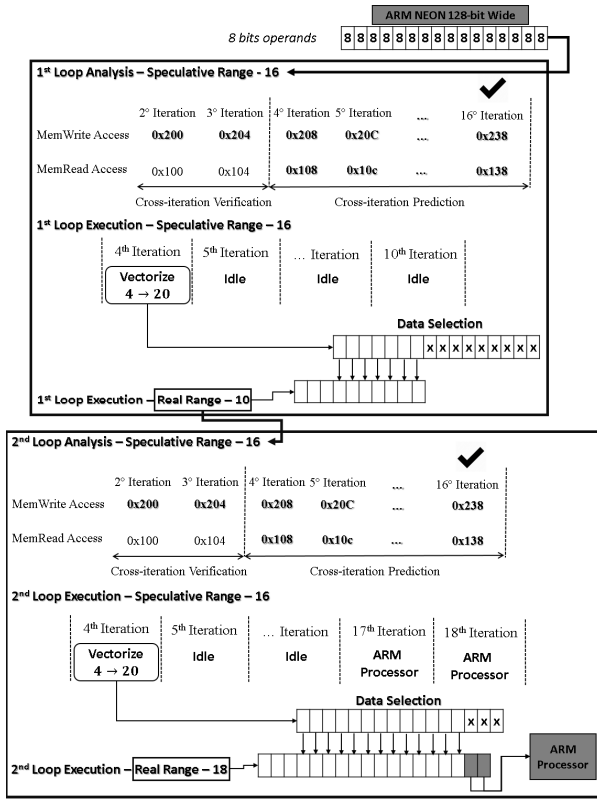


Figure 15 – DRLB Cross-iteration Analysis and Execution

There are three Dynamic Range loop predicting possibilities:

- if the loop executes a smaller number of iterations than previously performed, only the results of the current range are saved and the previous loop range is replaced by the current range;
- if the loop executes a greater number of times, the remaining iterations are performed by the general purpose processor and the previous loop range is replaced by the current range;
- if the loop executes the expected number of iterations, the speculative range is retained.

VI. RESULTS

A. Methodology

To evaluate the effectiveness of the proposed approach, we have applied the Conditional Code Loop and Dynamic Range Loop techniques to the DSA (ARM NEON Extended DSA), and coupled the DSA to the ARM Cortex A12 (ARMv7 ISA) O3 model of gem5 [11] simulator. To gather performance results, we have compared the proposed technique with:

- the original ARM NEON DSA without Conditional Code Loop and Dynamic Range Loop support (ARM NEON Original DSA);
- the ARM Cortex A12 processor without DLP exploitation (ARM Original Execution);
- the ARM Cortex A12 processor coupled to NEON architecture (ARM NEON AutoVec) exploiting DLP through the support of ARM NEON auto-vectorization compiler.

It is important to notice that we employ the same ARM NEON engine in ARM NEON Original DSA, Extended DSA and ARM NEON AutoVec approaches, which provides the same opportunities to exploit DLP. Table 2 shows the configurations of ARM Original Execution, ARM NEON AutoVec, ARM NEON Original and Extended Dynamic SIMD Assembler.

Table 2 – Systems Setups

	ARM Original Execution	ARM NEON Original DSA	ARM NEON Extended DSA	ARM NEON AutoVec
Processor	ArmCortex-A12	ArmCortex-A12	ArmCortex-A12	ArmCortex-A12
Superscalar Width	2-wide	2-wide	2-wide	2-wide
CPU Clock	1GHz	1GHz	1GHz	1GHz
L1 Cache	64 kb	64 kb	64 kb	64 kb
L2 Cache	512 kb	512 kb	512 kb	512 kb
Cache Policy	LRU	LRU	LRU	LRU
Parallelism (NEON)	Not Used	Type Dependent 128-bit Wide	Type Dependent 128-bit Wide	Type Dependent 128-bit Wide
DSA Cache	-	8 kb	-	-
Verification Cache	-	1 kb	-	-

We have chosen benchmarks that cover two different scenarios of DLP exploitation to evaluate the systems shown in Table 2. We have selected:

- three applications with Conditional Code and Dynamic Range loops (Bit Counts[13], Dijkstra[13] and Susan E[13]);
- four applications with opportunities to exploit DLP but without Conditional Code and Dynamic Range loops (MM[13], RGB-Gray[14], Gaussian Filter[14] and Q Sort[13]).

B. Performance

Figure 16 shows the performance improvements of the ARM NEON Compiler, ARM NEON Original DSA [8] and ARM NEON Extended DSA over the ARM Original Execution. As it can be noticed, when the applications contain Conditional Code or Dynamic Range loops (Dijkstra and BitCounts), the Extended DSA presents performance improvements of 45% for BitCounts and 32% for Dijkstra over the ARM Original Execution. In cases where we have a great opportunity of exploiting DLP, our approach achieves up to 61% of performance improvements over the ARM Original Execution. When Gaussian Filter and Susan Edges are considered, the performance gains are lower since such applications provide smaller opportunities to exploit DLP. On average, the Extended DSA provides performance improvements of 37% over the ARM Original Execution showing the importance of DLP exploitation.

The Original DSA cannot vectorize any loop of Dijkstra and BitCounts benchmarks since both just contain conditional code and dynamic range loops. Considering these benchmarks, the Extended DSA shows a performance improvement of 38.5% over the Original DSA. Susan E presents DLP exploitation opportunities in both Dynamic Behavior Loops and Count Loops. Thus, such benchmark is benefited from the improvements of both Original and Extended DSA, but the Extended DSA shows performance improvements of 4% over the Original DSA. In the remaining benchmarks there are no performance differences between Original DSA and Extended DSA, since such benchmarks do not contain conditional code and dynamic range loops.

The Extended DSA outperforms ARM auto-vectorization in all benchmarks but MM 64x64. Due to the extension proposed in this work, it achieves 40% of performance improvements over ARM auto-vectorization technique when executing BitCounts. The ARM auto-vectorization technique provides performance penalty of 3% considering Dijkstra benchmark, while our approach achieves a performance gain of 32%.

For the application with low DLP exploitation opportunities, our proposal maintained the same performance of the ARM Original Execution since DSA does not cause performance penalties when vectorizable loops are not found. In such scenario, ARM NEON auto-vectorization provides performance penalties of 1% in the Q Sort execution. In addition, besides achieving 40% performance improvements over the ARM auto-vectorization technique, the proposed approach keeps software compatibility and does not affect the SW development life cycle due to its transparent and dynamic DLP detection.

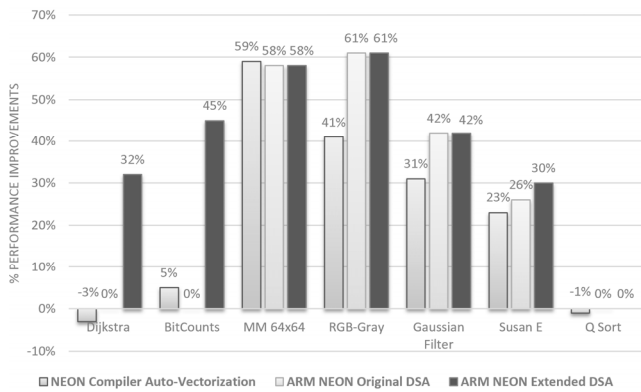


Figure 16 – ARM NEON Compiler AutoVec. vs. ARM NEON Original DSA vs. ARM NEON Extended DSA Performance

Table 3 presents the time consumed by ARM NEON Extended DSA to detect vectorizable loops and generate SIMD instructions considering the total execution time of each benchmark. As it can be seen, Dijkstra and BitCounts spend more time detecting vectorizable loops since they contain more Conditional Code and Dynamic Range loops. Benchmarks containing only static ranged vectorizable loops spent, on average, 1.5% of the execution time detecting vectorizable loops. Q Sort has no vectorizable loops but it spends 1.02% of its time analyzing non-vectorizable loops. However, since the DSA process is done in parallel with the ARM Cortex execution, no performance penalty is shown in the total execution time, as it can be noticed in Figure 16.

Table 3 – DSA Latency

	Dijkstra	BitCounts	MM 64x64	RGB-Gray	Gaussian Filter	Susan E	Q Sort
DSA Latency	28.42%	25.16%	1.42%	1.48%	1.33%	1.79%	1.02%

VII. CONCLUSION AND FUTURE WORK

In this work, we have proposed the coupling of two vectorization techniques to the Dynamic SIMD Assembler (DSA) approach. With such extension, the DSA is capable of vectorizing Dynamic Range and Conditional Code Loops. The proposed approach shows performance improvements of 37% over ARM original execution (without DLP exploitation). In addition, the extended DSA version outperforms: the original DSA up to 45% and the ARM auto-vectorization technique in 12%. For future work, we intend to support partial vectorization techniques.

ACKNOWLEDGEMENT

We are grateful to the institutions listed below for the financial support that they are providing us: Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) and Fundação de Amparo à pesquisa do Estado do Rio Grande do Sul (FAPERGS).

REFERENCES

- [1] Sui, Yulei, et al. "Loop-oriented array-and field-sensitive pointer analysis for automatic SIMD vectorization." *ACM SIGPLAN Notices*. Vol. 51. No. 5. ACM, 2016.
- [2] Zhou, Hao, and Jingling Xue. "Exploiting mixed SIMD parallelism by reducing data reorganization overhead." *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. ACM, 2016.
- [3] Baghsorkhi, Sara S., Nalini Vasudevan, and Youfeng Wu. "FlexVec: auto-vectorization for irregular loops." *ACM SIGPLAN Notices*. Vol. 51. No. 6. ACM, 2016.
- [4] Nuzman, Dorit, Ira Rosen, and Ayal Zaks. "Auto-vectorization of interleaved data for SIMD." *ACM SIGPLAN Notices* 41.6 (2006): 132-143.
- [5] Tian, Xinmin, et al. "Compiling C/C++ SIMD extensions for function and loop vectorization on multicore-SIMD processors." *Parallel and*

Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International. IEEE, 2012.

- [6] Bramas, Berenger. "Inastemp: A Novel Intrinsic-as-Template Library for Portable SIMD-Vectorization." *Scientific Programming* 2017 (2017).
- [7] Reddy, Venu Gopal. "Neon technology introduction." *ARM Corporation* (2008).
- [8] Omitted to allow blind review.
- [9] Lomont, Chris. "Introduction to Intel advanced vector extensions." *Intel White Paper* (2011): 1-21.
- [10] Diefendorff, Keith, et al. "AltiVec extension to PowerPC accelerates media processing." *IEEE Micro* 20.2 (2000): 85-95.
- [11] Binkert, Nathan, et al. "The gem5 simulator." *ACM SIGARCH Computer Architecture News* 39.2 (2011): 1-7.
- [12] Mitra, Gaurav, et al. "Use of SIMD vector operations to accelerate application code performance on low-powered ARM and Intel platforms." *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*. IEEE, 2013.
- [13] Guthaus, Matthew R., et al. "MiBench: A free, commercially representative embedded benchmark suite." *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*. IEEE, 2001.
- [14] Bradski, Gary, and Adrian Kaehler. "OpenCV." *Dr. Dobb's journal of software tools* 3 (2000).
- [15] Maleki, Saeed, et al. "An evaluation of vectorizing compilers." *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*. IEEE, 2011.
- [16] Lattner, Chris, and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation." *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 2004.

8 ARTICLE 3 - BOOSTING SIMD BENEFITS THROUGH A RUN-TIME AND ENERGY EFFICIENT DLP DETECTION

Boosting SIMD Benefits through a Run-time and Energy Efficient DLP Detection

Michael Guilherme Jordan, Tiago Knorst, Julio Vicenzi and Mateus Beck Rutzig
Electronics and Computing Department - Federal University of Santa Maria – Santa Maria – Brazil
{michael.jordan, tiago.knorst, julio.vicenzi}@ecomf.ufsm.br; mateus@inf.ufsm.br

Abstract — Data Level Parallelism has been improving performance-energy tradeoff of current processors by coupling SIMD engines, such as Intel AVX and ARM NEON. Special libraries and compilers are used to support DLP execution on such engines. However, timing overhead on hand coding is inevitable since most software developers are not skilled to extract DLP using unfamiliar libraries. In addition, DLP detection through compiler, besides breaking software compatibility, is limited to static code analysis, which compromises performance gains. In this work, we propose a runtime DLP detection named as Dynamic SIMD Assembler, which transparently identifies vectorizable code regions to execute in the ARM NEON engine. Due to its dynamic fashion, DSA keeps software compatibility and avoids timing overhead on software developing process. Results have shown that DSA outperforms ARM NEON auto-vectorization compiler by 32% since it covers wider vectorized regions, such as Dynamic Range, Sentinel and Conditional Loops. In addition, DSA outperforms hand-vectorized code using ARM library by 26% reducing 45% of energy consumption with no penalties over software development time.

Keywords— *DLP, SIMD, Vectorization, ARM NEON*

I. INTRODUCTION

Speech and vision recognition have been taking important role in the current era of cognitive computing to analyze human behaviors [1]. In particular, such application domains are benefit from Single Instruction Multiple Data (SIMD) machines since their algorithms are plentiful of Data-Parallel Statements.

Currently, SIMD engines are present in market processors, which can support the execution of such application domains. ARM NEON [2], Intel SSE/AVX [3] and IBM Altivec [4] are vector engines coupled to general purpose processors with the purpose of benefiting energy-performance tradeoff on data-parallel applications. The execution of such engines is supported by vector instructions that can be generated by: automatic vectorization through compiler and hand-coding using low-level functions available on specific programming libraries.

Both compiler techniques and libraries directives focus on exploiting Data Level Parallelism (DLP) opportunities on loops statements, since same operations are repeated over data independent structures. However, the vectorization of most loop statements, such as dynamic range, sentinel and conditional loops relies on runtime information, which restrict compiler and hand-coding DLP coverage. In addition, besides breaking software compatibility, timing overhead on hand-coding is inevitable since most software developers are not skilled to extract DLP using unfamiliar libraries.

In this work, we propose a runtime DLP detection, named as Dynamic SIMD Assembler (DSA), which transparently identifies vectorizable code regions, builds SIMD instructions and triggers the SIMD engine. Due to its dynamic fashion, DSA keeps binary compatibility and avoids timing overhead on software developing process. In addition, unlike compiler and

hand-coding techniques, DSA is capable of vectorizing all aforementioned loop statements since it is aware of runtime information, which boost the DLP coverage and, consequently, performance-energy tradeoff over techniques based on static analysis. Summarizing, this work contributes to:

- show that is mandatory a runtime vectorization exploitation to boost the applications DLP coverage;
- propose an energy efficient runtime vectorization technique capable of boost applications performance by increasing vectorization coverage of techniques based on static analysis with no penalties over software development time.

This work is organized as follows; Section II presents the Related Work. Section III presents the Dynamic SIMD Assembler System. Methodology and Results are shown in Section IV. Finally, we present conclusions and future work in Section V.

II. RELATED WORK

In the academic field, many researches have been exploiting Data Level Parallelism (DLP) to achieve performance improvements and energy savings. Sui Yulei [5] extends the LLVM compiler [6] to automatically vectorize Loop-Oriented Pointer. This technique is able to increase the number of vectorizable basic blocks achieving performance improvements from 2.95% to 7.23%.

Similar to [5], Zhou Hao [7] presents the Loop-Mix compiler that vectorizes loops focusing on reorganizing data when mixed SIMD parallelism (inter-loops and intra-loops) is considered. This technique outperforms the Loop-ILV by 36%. Sara S. Bagsorkhi [8] proposes FlexVec, a partial vectorization technique to dynamically adjust vector length for loops affected by cross-iteration dependencies. FlexVec extends the AVX-512 ISA showing a Geomean performance improvement over the original AVX ISA from 9% to 11%. Dorit Nuzman [9] proposes a compiler based technique aiming to vectorize outer loop. It shows performance improvements of 3.13 and 2.77 when coupled to a Cell BE SPU and a PowerPC970, respectively.

Tian Xinmin [10] presents a set of C/C++ directives extensions for SIMD programming capable of automatic translating both functions and loops. Significant speedups (from 3.07x to 4.69x) are achieved when these optimizations are applied. Bramas Berenger [11] proposes Inastemp, a lightweight open-source C++ library that provides SIMD Vectorization to several ISA, such as SSE, AVX, AVX512 and ALTIVEC/VMX. The authors claim that Inastemp shows the same performance on exploiting DLP than libraries developed for specific ISA.

ARM NEON [2] is a SIMD engine coupled to the ARMv7 architecture that is triggered through specific ARM SIMD

instructions. ARM supports two approaches to produce ARM NEON code: compiler auto-vectorization and ARM NEON software library.

Despite the advantages of automatic auto-vectorization through the compiler shown in [2][5][7][8][9], their static code exploitation limits the performance gains since it is not capable of identifying vectorizable regions that depends on data that is only available at runtime, such as: conditional, dynamic range loops and sentinel loops. In addition, hand coding using software libraries proposed in [2][10][11] inevitably causes timing overhead since most software developers are not skilled to extract DLP using unfamiliar low-level functions. Thus, in this work, we propose Dynamic SIMD Assembler (DSA) that automatically vectorizes code regions to execute in a SIMD engine. DSA is a hardware module coupled to an ARM Processor responsible for detecting vectorizable code regions, generating ARM SIMD instruction and triggering NEON engine at runtime. Due to its dynamic nature, DSA keeps binary compatibility and avoids timing overhead on software developing process. Moreover, performance improvements with energy savings are feasible to achieve in a wide range of application domains since the proposed approach covers larger vectorizable code regions than static analysis techniques, such as count loops, conditional statements, dynamic range and sentinel loops.

III. DYNAMIC SIMD ASSEMBLER

A. DSA Overview

The Dynamic SIMD Assembler (DSA) is tightly coupled to the ARMv7-A processor [12]. Figure 1 shows the system overview. As it can be seen, the DSA is composed of a SIMD instruction logic detection and two cache memories (*DSA Cache and Verification Cache*). The DSA Cache is responsible for storing information about the built SIMD instructions over the vectorizable loops. The Verification Cache (V-Cache) stores the addresses of data memory accesses performed into the vectorizable loops (more details about caches in Section IV).

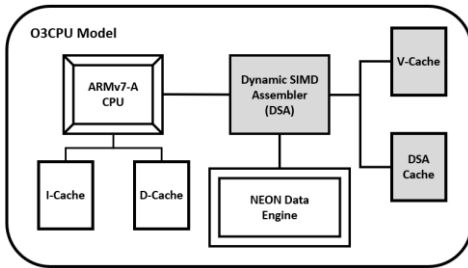


Figure 1. System Overview

Figure 2 shows an overview of how the DSA works. In the first scenario (*Scenario 1 – DSA Loop Analysis*), the DSA and ARMv7-A processor operate in parallel. While the ARM processor executes the incoming instructions, the DSA is in a probing mode, searching for a vectorizable loop to build SIMD instructions. In such execution mode, the NEON Engine remains deactivated. If the DSA detects a vectorizable loop, the second scenario is triggered (*Scenario 2 – DSA Loop Execution*). In this scenario, the DSA deactivates the ARMv7-A processor and activates the NEON Data Engine to execute the built SIMD instruction (Vectorized Instructions). It is important to notice that the DSA works in parallel with the ARMv7-A CPU execution, which means that the processor’s critical path is not affected by the DSA.

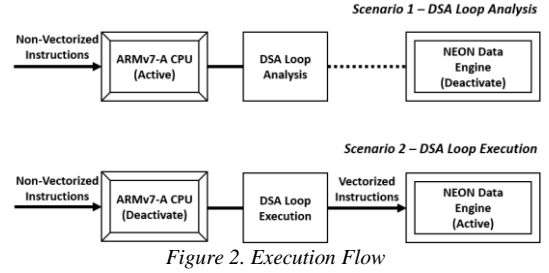


Figure 2. Execution Flow

B. Dynamic SIMD Assembler DLP Coverage

As explained before, the Dynamic SIMD Assembler is capable of exploiting vectorizable regions at runtime, which extends DLP Exploitation of hand coding using ARM library and auto-vectorization compiler. Figure 3 shows loop examples of (A) count loop; (B) dynamic range loop; (C) conditional loop; (D) loop with a function call. As it can be seen, the pseudocode (A) presents a simple vectorizable loop in which both the compiler and DSA would be capable of vectorize. The pseudocode (B) has a dynamic range loop, where the loop size is determined by an input or even a data calculated at runtime. The pseudocode (C) has a loop containing conditional statements which the execution is also determined at runtime. The same evaluation can be made for the pseudocode (D), which has a loop containing a function call that depends on a variable calculated at runtime. In this way, the pseudocodes (B), (C) and (D) cannot be vectorized by the compiler auto-vectorization techniques since they depend on data manipulation at runtime. However, as the DSA (Dynamic SIMD Assembler) analyzes the application code at runtime, it is capable of vectorizing all aforementioned situation.

Summarizing, the DSA covers full vectorization of: Count Loops, Function Loops, Outer and Inner Loops, Dynamic Range Loops and Sentinel Loops. In addition, as it can be seen in next section, the DSA also supports partial vectorization of loops with cross-iteration dependencies.

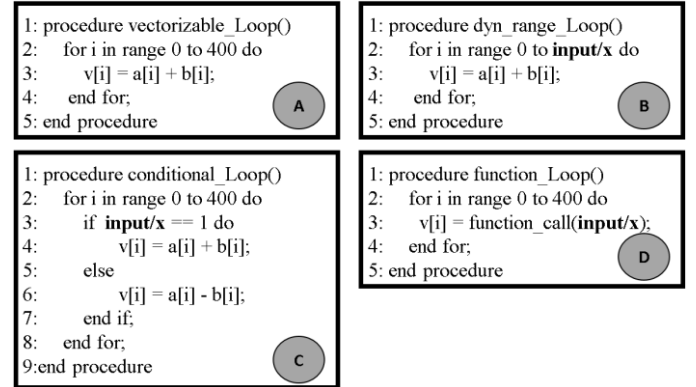


Figure 3 – Examples of Loops

C. Dynamic SIMD Assembler Overview

The Dynamic SIMD Assembler (DSA) detection process is based on a State Machine (SM) composed of six stages: Loop Detection, Data Collection, Dependency Analysis, Store ID/Execution, Mapping and Speculative Execution. Each one of these stages is activated in different loop iterations.

As it can be seen in Figure 4, the Loop Detection stage is triggered by the end of the first loop iteration. The Loop Detection stage is responsible for:

- detecting the presence of a loop;
- checking the existence of innermost-loop and outer-loops;

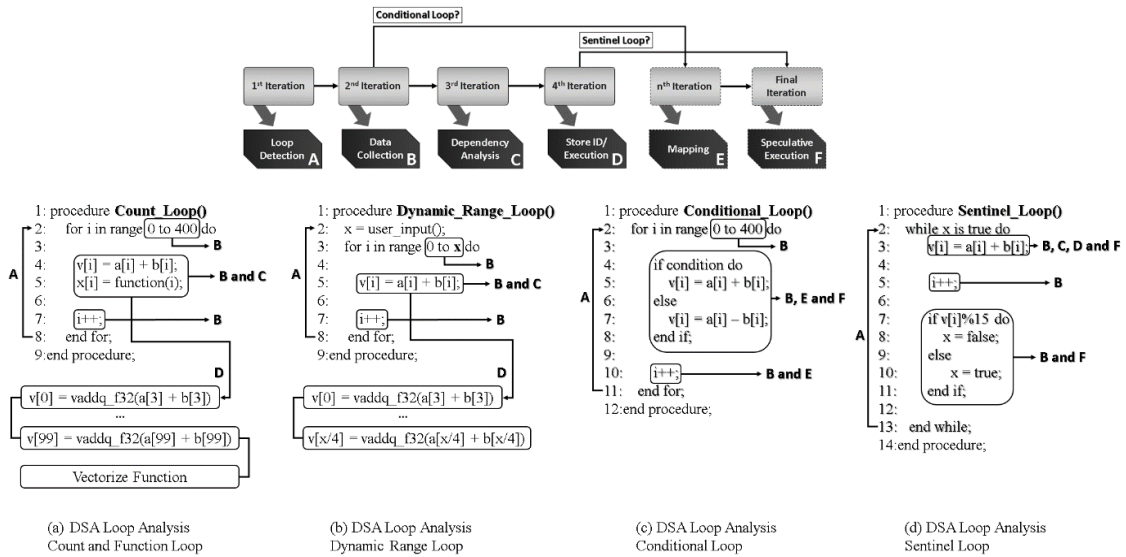


Figure 4- DSA Analysis and Execution Process

- accessing the DSA cache, checking if the current loop is already vectorizable.

The Data Collection stage is triggered in the second loop iteration. This stage is responsible for:

- evaluating the loop range (number of iterations), vectorizable instructions and their operands;
- identifying the existence of function calls and conditional code inside the loop;
- storing the addresses of data memory accesses in the Verification Cache.

The Dependency Analysis stage is triggered in the third loop iteration. This stage is responsible for:

- analyzing the cross-iteration dependency (dependencies between two or more iterations in the same loop statement).

The Store ID/Execution stage is triggered in the fourth loop iteration. This stage is responsible for:

- concluding the vectorization of Count loops, Function loops, Outer/Inner loops, Dynamic Ranged Loops, Sentinel loops and Partial loops;
- generating and saving the loop identification (ID) in the DSA Cache;
- building SIMD instruction and activating the execution of the ARM NEON engine.

The Mapping stage is only activated for Conditional loops. This stage is responsible for:

- mapping the executed conditional code statements;
- detecting cross-iteration dependencies between conditional statements.

The Speculative Execution stage is only activated for Conditional and Sentinel loops. This stage is responsible for:

- selecting data generated during the vectorization in the end of loop execution (Sentinel and Conditional Loop);
- tracking Sentinel loop range;
- storing mapped conditions of Conditional Loop for further executions.

Figure 4 shows a DSA execution example by considering: Count and Function Loop (a), Dynamic Range Loop (b), Conditional Loop (c) and Sentinel Loop (d).

Following the *Count_Loop()* (a) procedure example, the Loop Detection stage (A) detects the loop by the end of the execution of the first iteration by analyzing instruction address

gaps and branches. In the second iteration, the Data Collection stage (B) identifies the loop range (400) and the value of the increment/decrement ($i++$). In addition, such stage: stores the addresses of the data memory accesses ($Mem[a[i]]$, $Mem[b[i]]$ and $Mem[v[i]]$) in the Verification Cache; and identifies function calls inside the loop ($x[i] = function[i]$) by verifying branches and the memory address gap between instructions fetched from memory. The detection of function calls is mandatory to analyze cross-iteration dependencies since the increment/decrement register can be modified for an operation inside the function call. In the third iteration, the Dependency Analysis Stage (B) analyses data dependencies between iterations (more detailed in subsection D). For the current example, the DSA identifies that no cross-iteration dependency exists and triggers the Store ID/Execution Stage. Such stage stores the Loop ID in the DSA cache to avoid repeating loop analysis and builds SIMD instructions to execute the remaining iterations in the ARM NEON engine. The DSA needs four parameters to generate SIMD instructions: data type, loop range, operation and ARM NEON execution support. For such an example the parameters are: *float*, 400, *add*, 128-bit wide, respectively. Thus, the DSA generates an instruction equivalent to the *vaddq_f32* instruction of the NEON architecture. Since the corresponding ARM NEON engine can operate 128 bits in parallel and the float type is 32-bit wide data, the DSA divides the loop range by the factor four, running the *vaddq_f32* one hundred times, instead of executing a non-vectorizable *add* operation four hundred times.

In the *Dynamic_Range_Loop* (b) procedure example, the loop size is calculated at runtime but before the loop execution. In this case, the loop analysis passes through the same steps as the *Count_Loop* (a) example. However, instead of having a single analysis when the loop executes for the first time, the *Dynamic_Range_Loop* (b) must be analyzed on every execution, since the loop range can change on each loop execution, the Dependency Analysis Stage (C) needs to verify if the vectorization is allowed based on current value of the loop range.

Considering the *Conditional_Loop* (c) example, the Loop Detection Stage (A) detects the loop by the end of the execution of the first iteration. The Data Collection Stage (B), besides collecting the necessary data to vectorize the loop, also detects if all instruction addresses within the loop range were accessed. In case a instruction address gap is detected, a conditional loop is confirmed. In such case, the Mapping stage (E) is activated.

This stage is responsible for mapping every condition within the loop body and detecting any cross-iteration dependency. If no cross-iteration dependency is detected, all conditions within the loop can be vectorized considering the remaining loop range. During the remaining loop execution, the DSA maps every accessed condition. While the mapping is activated, the vectorized instructions (if: $v[i] = a[i] + b[i]$ and else: $v[i]=a[i]-b[i]$) are not executed. At the end of the loop, the Speculative Execution Stage (F) selects the appropriate results based on the mapping process.

The *Sentinel_Loop* (d) vectorization is based on Speculative Execution. The DSA assumes a speculative loop range when detecting such loop type since it is not feasible to have such information beforehand. In the Data Collection Stage, besides collecting all the necessary data to vectorize the loop, the DSA chooses a loop range that maximizes utilization of the functional units available in the ARM NEON engine. Assuming a 128-bit wide ARM NEON, the DSA chooses a speculative loop range of four, in order to use all vector units, since the operands width is 32 bits (*32 bit float*). In the third iteration, in the Dependency Analysis Stage (C), the DSA analyses and predicts any cross-iteration dependency based on the speculative loop range. If no Cross-Iteration Dependency is found, the Store ID/Execution stage (D) is activated and the instructions are vectorized based on the speculative loop range. In addition, the loop ID is saved in the DSA cache. The speculative execution can provide three situations:

- if the loop executes fewer iterations than the speculated number of iterations, the execution results of the speculated number of iterations are written back, the remaining results are discarded and the loop range is updated in DSA cache;
- if the loop executes greater iterations than the speculated number of iterations, the execution results of the speculated number of iterations are written back, the further iterations are executed by the general purpose processor and the loop range is updated in DSA cache;
- if the loop executes the speculated number of iterations, the ARM NEON results of the speculated number of iterations are written back and the speculative range is maintained in the DSA cache.

D. Cross-iteration Dependency Prediction

At the memory access point of view, a cross-iteration dependency exists when the same data memory address is accessed in different loop iterations. The DSA cross-iteration analysis starts in the 2nd loop iteration, where the addresses of data memory accesses are saved in the Verification Cache (VC). Even having the memory addresses in the VC and comparing them to the memory addresses performed on every iteration, one cannot discard cross-iteration dependencies in future iterations. Assuming such situation, we have implemented *Cross-iteration Dependency Prediction*. The equations below describe the steps

Cross-iteration Dependency Prediction

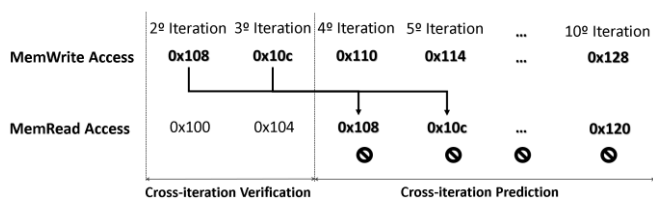


Figure 5- Example of a Cross-iteration Dependency Prediction Process

of the prediction process, where $M_{Read[2]}$ and $M_{Read[3]}$ is the memory address accessed by a *MemRead* (*load*) instruction in the second and third loop iterations, respectively. $M_{Read[lastIteration]}$ is the memory address accessed by a load instruction in the last executed iteration (Equation 4), x is the interval between $M_{Read[2]}$ and $M_{Read[lastIteration]}$ (Equation 1), $M_{Write[2]}$ is the memory address accessed by a *MemWrite* (*store*) instruction in the second iteration (Equations 2 and 3), M_{Range} is the memory address range between the $M_{Read[2]}$ and $M_{Read[3]}$ (Equation 5), *CID* means Cross-Iteration Dependency and *NCID* means No Cross-Iteration Dependency.

$$M_{Read[3]} \leq x \leq M_{Read[lastIteration]} \quad (1)$$

$$M_{Write[2]} \in x \rightarrow CID \quad (2)$$

$$M_{Write[2]} \notin x \rightarrow NCID \quad (3)$$

$$M_{Read[lastIteration]} = M_{Read[2]} + (M_{Gap} * (lastIteration - 2)) \quad (4)$$

$$M_{Gap} = |M_{Read[3]} - M_{Read[2]}| \quad (5)$$

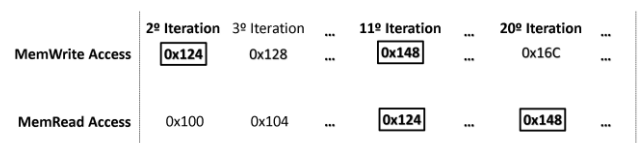
Considering the equations above, if the $M_{Write[2]}$ is within the memory address range of $M_{Read[3]}$ and $M_{Read[lastIteration]}$ (Equation 2), the loop would have a cross-iteration dependency since the load instruction of a future loop iteration could perform a memory access in the same memory address of the store instruction executed in the second loop iteration. The memory address of the load instruction executed in the last iteration is predicted based on the sum of the $M_{Read[2]}$ and the equation $(M_{Gap} * (lastIteration - 2))$ (Equation 4). Thus, in case of $M_{Write[2]}$ is out of the memory address interval of $M_{Read[3]}$ and $M_{Read[lastIteration]}$ (Equation 3), one can ensure that the loop has no cross-iteration dependency.

Figure 5 illustrates an example of how *Cross-iteration Dependency Prediction* (*CIDP*) works. In such example, the DSA detects that there is no cross-iteration dependency between 2nd and 3rd iteration. Thus, by the end of the 3rd loop iteration, the *CIDP* is activated by applying Equation 5 ($M_{Gap} = |0x104 - 0x100| = 0x004$). Using Equation 4, one can calculate the memory address of the load instruction of the last iteration $M_{Read[lastIteration]} = 0x100 + 0x020 = 0x120$. By applying Equations 1 and 2, the *CIDP* detects that $M_{Write[2]} = 0x108$ is within the interval ($M_{Read[3]} \leq x \leq M_{Read[lastIteration]}$) = $0x100 \leq x \leq 0x120$), which produces a cross-iteration dependency.

E. Dynamic SIMD Assembler Partial Vectorization

Despite having cross-iteration dependencies, loops can be partially vectorized by avoiding vectorization of iterations that produces dependencies. Figure 6 shows how the partial vectorization works. As it can be seen, the *CIDP* detects cross iteration dependency between the 2nd Iteration and the 11th Iteration due to the data memory address 0x124. However, there is a gap between the 2nd Iteration to the 10th that could be vectorized. Thus, for such an example, the DSA performs the

Cross-iteration Dependency Prediction



Partial Loop Vectorization

Vectorize 4th → 10th → Vectorize 11th → 19th → Vectorize 20th → nth

Figure 6. DSA Partial Vectorization Technique

vectorization detection process from 1st to 4th Iterations, which allows the vectorization from the 4th up to 10th iteration which provides data to the vectorization of the iterations from 11th up to 19th. The same process repeats until the end of the loop execution.

IV. RESULTS

A. Methodology

We have coupled the DSA to an ARMv7 ISA processor using the O3CPU model of gem5 [12] simulator to evaluate the proposed approach. To gather performance results, we have compared the DSA with:

- an ARMv7 ISA processor without NEON engine (ARM Original Execution);
- an ARMv7 ISA processor coupled to a NEON architecture exploiting DLP through the support of the ARM NEON auto-vectorization compiler (ARM NEON AutoVec);
- an ARMv7 ISA processor coupled to a NEON architecture exploiting DLP through hand-coded applications using ARM NEON library (ARM NEON Hand-Coded).

Table 1 shows the configurations of all setups. It is important to notice that we coupled the same ARM NEON architecture in ARM NEON AutoVec, ARM NEON Hand-Coded and DSA, which provides the same DLP exploitation degree. We used Cadence RTL Compiler [13] to gather energy results from the VHDL description of the Dynamic SIMD Assembler and McPAT of the ARMv7 processor.

Table 1 – System Setups

	ARM Original Execution	ARM NEON DSA	ARM NEON AutoVec	ARM NEON Hand-Coded
Processor	gem5 O3CPU (ARMv7)	gem5 O3CPU (ARMv7)	gem5 O3CPU (ARMv7)	gem5 O3CPU (ARMv7)
Superscalar Width	2-wide	2-wide	2-wide	2-wide
CPU Clock	1GHz	1GHz	1GHz	1GHz
L1 Cache	64 kb	64 kb	64 kb	64 kb
L2 Cache	512 kb	512 kb	512 kb	512 kb
Cache Policy	LRU	LRU	LRU	LRU
Parallelism (NEON)	Not Used	Type Dependent 128-bit Wide	Type Dependent 128-bit Wide	Type Dependent 128-bit Wide
DSA Cache	-	8 kb	-	-
Verification Cache	-	1 kb	-	-

B. Benchmarks Characterization

Aiming to create a heterogeneous workload to evaluate the proposed approach, we have selected benchmarks from different suites following their opportunities to exploit DLP: MM 64x64 [14] and RGB-Grayscale [14], which provide great opportunities; Susan E [14] and JPEG [16] that provide medium opportunities; and Bit Counts [14], Susan C [14], Susan S [14] and Gaussian Filter [15], which have low opportunities.

Figure 7 presents a static profiling that considers the percentage of each loop type in the aforementioned benchmarks. Such analysis quantifies the presence of vectorizable loops statically, which means that weight over the execution time of each loop is not considered. As it can be seen, there are different degrees of vectorization opportunities in the selected benchmarks, 86% of the Susan E loops can be vectorized but just 33% of MM 64x64 and RGB-G 320x240. However, as explained before, considering 86% of Susan E vectorizable loops, only 14.3% (Count Loops) could be vectorized at compile and programming time. On the other hand, all 33% of the loops of MM 64x64 and RGB-G 320x240 are vectorized at compile and programming time. However, on average, only 21% of the application loops can be vectorized at compile and programming time. A runtime analysis potentially

increases such coverage to 57%, since it can vectorize all considered loops types. The benchmarks characterization indicates the need for a runtime analysis to boost application performance on exploiting DLP.

As shown in Section III, the DSA produces a time overhead to detect vectorizable code regions and build NEON instructions. Table 2 shows the percentage of the execution time spent in the DSA detection process. As it can be seen, Bit Counts and Susan E spend 26.20% and 15.58% of the execution time detecting vectorizable loops. Such applications contain sentinel loops that relies on Mapping Stage execution (Figure 4) which leaves active during the whole vectorization/execution process. The remaining benchmarks spend, on average, only 1.53% of the whole execution time detecting vectorizable regions showing the acceptable overhead of the proposed approach.

Table 2- DSA Detection Latency

	Bit Counts	Gaussian Filter	Susan C	Susan E	Susan S	JPEG	MM 64x64	RGB-G 320x240
DSA Latency	26.20%	0.53%	1.91%	15.58%	3.71%	2.69%	1.74%	1.68%

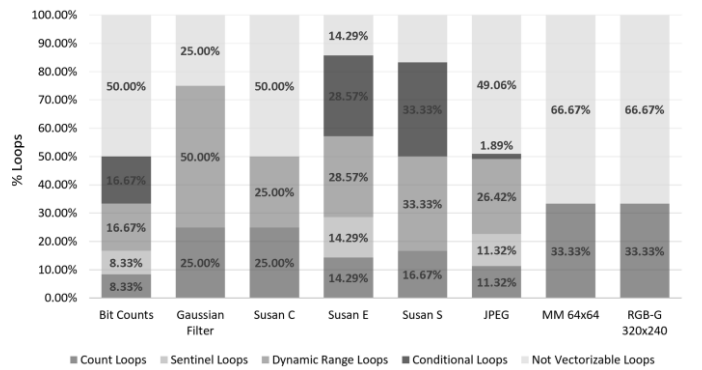


Figure 7 - Percentage of Loop Types in the Selected Applications

C. Performance

Figure 8 shows the performance improvements of the ARM NEON DSA, the ARM NEON AutoVec and the ARM NEON Hand-Coded over the ARM Original Execution. As it can be noticed, the proposed technique provides performance improvements over the ARM Original Execution in all benchmarks. The performance gains increase as the DLP opportunities increase as well. Bit Counts (low DLP opportunities) shows performance improvements of 32% while RGB-G 320x240 (great DLP opportunities) of 70%. RGB-G and MM 64x64, besides having only 33.33% of vectorizable loops (shown in Figure 7), such loops consume most of the execution time, which explains the high acceleration in both benchmarks. Results demonstrate the efficient runtime DLP exploitation of the proposed approach by showing, on average, 45% of performance improvements over ARM Original Execution running applications with heterogeneous DLP opportunities.

Due to the larger DLP exploitation opportunities of DSA over the static analysis of the ARM NEON AutoVec, the proposed approach outperforms the compiler technique in all benchmarks but MM 64x64 by only 0.6%. Performance gains of the proposed approach over compiler technique comes from the vectorization of Sentinel Loops, Dynamic Ranged Loops and Conditional Loops vectorization which are not capable to be achieved at compile time. As it can be seen in Figure 7, considering Susan E, ARM NEON AutoVec covers only 14.3% of vectorizable loops (Count Loops) while DSA boost such covering to 86% which results on 71,5% of performance

improvements over the compiler technique. In addition, the ARM NEON AutoVec provides performance penalties in Bit Counts, Gaussian Filter and Susan S since the greater latencies of NEON instructions allocated by the compiler were not diluted by DLP performance gains. Besides keeping the binary compatibility, broken by the ARM NEON AutoVec, the proposed approach provides, on average, 32% of performance improvements over the ARM NEON AutoVec technique considering benchmarks with different DLP opportunities.

Similar to the ARM NEON AutoVec, due to its dynamic DLP exploitation, the proposed approach outperforms ARM NEON Hand-Coded.

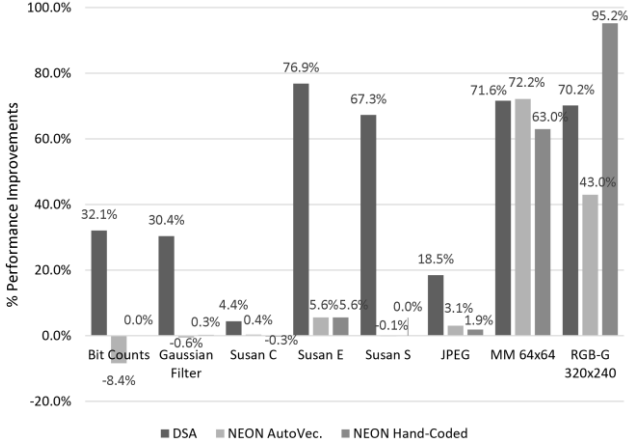


Figure 8 – Performance Improvements over ARM Original Execution

D. Energy

Figure 9 shows the energy consumption of DSA, ARM NEON AutoVec and ARM NEON Hand-Coded considering the ARM Original Execution as a baseline. As it can be seen, the DSA achieves greater energy savings than static analysis approaches in all benchmarks but RGB-G 320x240 since such an application boost performance due to code optimizations using ARM NEON library. On average, DSA achieves 45%, 31.2% and 23.5% of energy savings over ARM Original Execution, ARM NEON AutoVec and ARM NEON Hand-Coded, respectively.

Table 3 shows the energy consumption percentage of the DSA hardware relative to the whole system energy (ARMv7 CPU + NEON Engine). As it can be noticed, the DSA detection process is responsible for, at most, 11% and, on average, for 2.8% of the whole system energy. Summarizing, the experiments have shown that the lightweight DSA detection achieves higher performance than static analysis approaches with lower energy consumption maintaining binary compatibility with no penalties on software development time.

V. CONCLUSION AND FUTURE WORK

In this work, we propose the Dynamic SIMD Assembler (DSA) that automatically vectorizes code regions to execute in ARM NEON. In comparison with compiler and programming techniques, due to its dynamic nature, DSA boosts DLP coverage, keeps binary compatibility and avoids timing overhead on software developing process. Experimental results show that the DSA outperforms both ARM NEON Auto-Vectorization and ARM NEON Hand-Coded methods by 32% and 26%, respectively, while keeping binary compatibility and software productivity. In terms of energy, the DSA shows 45%, 31% and 23.5% of energy savings over the ARM Original Execution, ARM NEON AutoVec and ARM NEON Hand-Coded considering applications with heterogeneous DLP opportunities. For future works, we intend to merge DLP, ILP

and TLP exploitation in a single MPSoC by using DSA in a heterogeneous fashion.

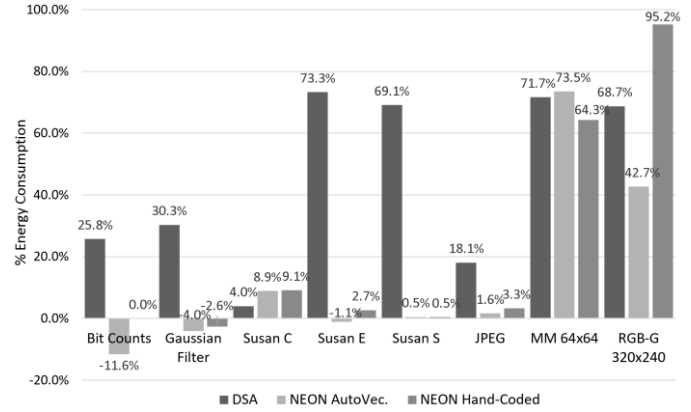


Figure 9. Energy Savings over ARM Original Execution

Table 3. DSA Energy Consumption

	Bit Counts	Gaussian Filter	Susan C	Susan E	Susan S	JPEG	MM 64x64	RGB-G 320x240
ARM+NEON Consumption	88.52%	99.77%	99.52%	93.47%	97.54%	99.11%	99.40%	99.90%
DSA Consumption	11.48%	0.23%	0.48%	6.53%	2.46%	0.89%	0.60%	0.10%

REFERENCES

- [1] Dharmendra S. Modha, Rajagopal Ananthanarayanan, Steven K. Esser, Anthony Ndirango, Anthony J. Sherbondy, and Raghavendra Singh. 2011. Cognitive computing. *Commun. ACM* 54, 8 (August 2011), 62-71.
- [2] Reddy, Venu Gopal. "Neon technology introduction." *ARM Corporation* (2008).
- [3] Lomont, Chris. "Introduction to Intel advanced vector extensions." *Intel White Paper* (2011): 1-21.
- [4] Diefendorff, Keith, et al. "AltiVec extension to PowerPC accelerates media processing." *IEEE Micro* 20.2 (2000): 85-95.
- [5] Sui, Yulei, et al. "Loop-oriented array-and field-sensitive pointer analysis for automatic SIMD vectorization." *ACM SIGPLAN Notices*. Vol. 51. No. 5. ACM, 2016.
- [6] Lattner, Chris, and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation." *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 2004.
- [7] Zhou, Hao, and Jingling Xue. "Exploiting mixed SIMD parallelism by reducing data reorganization overhead." *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. ACM, 2016.
- [8] Baghsorkhi, Sara S., Nalini Vasudevan, and Youfeng Wu. "FlexVec: auto-vectorization for irregular loops." *ACM SIGPLAN Notices*. Vol. 51. No. 6. ACM, 2016.
- [9] Nuzman, Dorit, Ira Rosen, and Ayal Zaks. "Auto-vectorization of interleaved data for SIMD." *ACM SIGPLAN Notices* 41.6 (2006): 132-143.
- [10] Tian, Xinmin, et al. "Compiling C/C++ SIMD extensions for function and loop vectorization on multicore-SIMD processors." *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*. IEEE, 2012.
- [11] Bramas, Berenger. "Inastemp: A Novel Intrinsics-as-Template Library for Portable SIMD-Vectorization." *Scientific Programming* 2017 (2017).
- [12] Binkert, Nathan, et al. "The gem5 simulator." *ACM SIGARCH Computer Architecture News* 39.2 (2011): 1-7.
- [13] Cadence, R. T. L. "Compiler User's Manual."
- [14] Guthaus, Matthew R., et al. "MiBench: A free, commercially representative embedded benchmark suite." *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*. IEEE, 2001.
- [15] Bradski, Gary, and Adrian Kaehler. "OpenCV." *Dr. Dobb's journal of software tools* 3 (2000).
- [16] Lee, Chunho, Miodrag Potkonjak, and William H. Mangione-Smith. "MediaBench: a tool for evaluating and synthesizing multimedia and communications systems." *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society, 1997.

9 DISCUSSION

The paper titled Improving Software Productivity and Performance through a Transparent SIMD Execution introduces the first version of the DSA. This work focuses on presenting a transparent hardware capable of detecting vectorizable regions at runtime without requiring specific libraries or compilers. This primary version is capable of detecting: Count Loops, Outer loops and Function Loops. To evaluate this work, we have compared the DSA performance with the ARM NEON auto-vectorization compiler. We also implemented the HDL version of the DSA to gather data about area. Unlike the auto-vectorization compiler, the DSA does not provide performance penalties running benchmarks with low DLP Exploitation opportunities. Besides, in the RGB-Gray benchmark, the DSA could achieve 20% of performance improvements over the auto-vectorization compiler with an area overhead of 2.18% over the ARM processor. Considering all benchmarks, the DSA outperforms the ARM auto-vectorization in 6%, showing that its dynamic nature could exploit vectorizable regions efficiently.

In the paper titled Runtime Vectorization of Conditional Code and Dynamic Range Loops to ARM NEON Engine the DSA was extended in order to provide vectorization of loops with dynamic behavior. The extended version is capable of detecting: Count Loops, Outer loops, Function Loops, Conditional Loops, Dynamic Sized Loops and Sentinel Loops. To evaluate this work, we compared the DSA with its primary version and with the ARM NEON auto-vectorization compiler. Due to the proposed extensions, the DSA outperforms the NEON auto-vectorization compiler by 40% considering BitCounts benchmark. In general, the extended DSA could provide 38% performance improvements over the original approach and 12% performance improvements over the NEON auto-vectorization compiler. In this paper, we also provide data about the time consumed by the DSA to detect vectorizable loops and generate SIMD instructions considering the total execution time of each benchmark. Benchmarks containing more Conditional Loops and Dynamic Range Loops (Dijkstra and BitCounts) spent more time detecting vectorizable loops. Benchmarks containing only static ranged vectorizable loops spent, on average, 1.5% of the execution time detecting vectorizable loops. For the Q Sort benchmark, which has no vectorizable loops, the DSA spent only 1.02% of the time analyzing non- vectorizable loops.

The paper titled Boosting SIMD Benefits through a Run-time and Energy Efficient DLP Detection evaluates the DSA energy consumption and applies the Partial Vectorization tech-

nique to the DSA. We also compare the performance and energy results of the DSA with an ARM NEON Hand-Coded approach and with the ARM NEON Auto-vectorization Compiler. The DSA boosts in 71,5% the performance of the Susan E benchmark over the NEON Auto-vectorization compiler and outperforms in 67% the ARM NEON Hand-Coded in Susan S benchmark. On average, the DSA outperforms the Hand-Coded approach in 26% reducing 23.5% of energy consumption and improves the Auto-vectorization Compiler performance in 32% reducing 31% of energy consumption.

All presented works show that is mandatory a runtime vectorization exploitation to boost applications DLP coverage and keep binary compatibility. Also, unlike a Just-in-time compiler, which demands a monitor task running concurrently, the DSA adds no execution time penalty, since it has its own processing hardware. Besides, the DSA applies little area overhead over the system. We coupled the DSA to an ARM ISA approach to evaluate its functionality and extract performance results, but the techniques can be adapted to any ISA.

Summarizing, the Dynamic SIMD Assembler is capable of:

- improving DLP coverage and software productivity in a runtime fashion with no execution time penalty;
- keeping binary compatibility since no code recompilation is needed;
- suggesting multi-ISA runtime vectorization techniques;
- improving performance and reducing energy consumption.

As aforementioned, the DSA was compared with the ARM auto-vectorization and library usage approaches, which are static DLP exploitation techniques. In our future works, we intend to evaluate the DSA over a Just-in-time compiler method, since they are both dynamic DLP exploitation approaches. Besides, by comparing both approaches, we expect to see which dynamic vectorization approach is the best option: the DSA hardware, which impacts on area increase, or the Just-in-time compiler, which demands no area increase but implies in performance penalties due a monitor task.

The DSA HDL implementation as well as the high-level simulator (DSA + O3CPU) can be found in (JORDAN, 2019).

10 CONCLUSION AND FUTURE WORK

In this work, we propose the Dynamic SIMD Assembler (DSA) that automatically vectorizes code regions to execute in ARM NEON during runtime. In comparison with compiler and programming techniques, due to its dynamic nature, DSA boosts DLP coverage, keeps binary compatibility and avoids timing overhead on software developing process. Experimental results show that the DSA outperforms both ARM NEON Auto-Vectorization and ARM NEON Hand-Coded methods by 32% and 26%, respectively, while keeping binary compatibility and software productivity. In terms of energy, the DSA shows 45%, 31% and 23.5% of energy savings over the ARM Original Execution, ARM NEON AutoVec and ARM NEON Hand-Coded considering applications with heterogeneous DLP opportunities.

For future works, we intend to:

- apply improved DSA instruction generation and memory hierarchy approaches in the DSA since both of them influence in the DSA analysis and execution latencies;
- implement the whole system in an HDL (hardware description language) to enable an accurate performance and energy system analysis;
- develop a software *Just-in-time* version of the DSA and compare such approach with the original DSA (hardware);
- provide an exploitation of the DSA version that mixes the benefits of a static vectorization compiler and the dynamic vectorization of the DSA (high performance version);
- expand the DSA loop vectorization (*DSA Analysis*), which means that complex control flow loops and loops with misaligned memory access will be covered;
- merge ILP, DLP and TLP exploitation in a single MPSoC by using DSA in a heterogeneous fashion.

REFERENCES

- Aho A. V., Lam M. S., Sethi R., Ullman J. D. (2014). *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley.
- Allen, Randy; Kennedy, Ken. *Optimizing compilers for modern architectures: a dependence-based approach*. San Francisco: Morgan Kaufmann, 2002.
- Amdahl, Gene M. "Validity of the single processor approach to achieving large scale computing capabilities." *Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM, 1967.
- ARM Limited 2017, "What can limit or prevent automatic vectorization", accessed 30 December 2018, <https://developer.arm.com/docs/dui0472/i/using-the-neon-vectorizing-compiler/what-can-limit-or-prevent-automatic-vectorization>.
- Baghsorkhi, Sara S., Nalini Vasudevan, and Youfeng Wu. "FlexVec: auto-vectorization for irregular loops." *ACM SIGPLAN Notices*. Vol. 51. No. 6. ACM, 2016.
- Bramas, Berenger. "Inastemp: A Novel Intrinsic-as-Template Library for Portable SIMD-Vectorization." *Scientific Programming 2017* (2017).
- Chang, Hoseok, and Wonyong Sung. "Efficient vectorization of SIMD programs with non-aligned and irregular data access hardware." *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*. ACM, 2008.
- Clark, Nathan, et al. "Liquid SIMD: Abstracting SIMD hardware using lightweight dynamic mapping." *2007 IEEE 13th International Symposium on High Performance Computer Architecture*. IEEE, 2007.
- Courtland, Rachel. "Transistors could stop shrinking in 2021." *IEEE Spectrum* 53.9 (2016): 9-11.
- Diefendorff, Keith, et al. "AltiVec extension to PowerPC accelerates media processing." *IEEE Micro* 20.2 (2000): 85-95.
- Hill, Mark D., and Michael R. Marty. "Amdahl's law in the multicore era." *Computer* 41.7 (2008).
- Hwu, Wen-reel, and Yale N. Patt. "HPSm, a high performance restricted data flow architecture having minimal functionality." *ACM SIGARCH Computer Architecture News*. Vol. 14. No. 2. IEEE Computer Society Press, 1986.
- John L. Hennessy and David A. Patterson, "Computer architecture: a quantitative approach", Fifth Edition, Elsevier, 2012.
- Jordan, Michael Guilherme, DSA Implementation. Accessed 25 January 2019, <https://github.com/dsaproject2019/DSA>.

Jordan, Michael Guilherme, Tiago Knorst, and Mateus Beck Rutzig. "Improving Software Productivity and Performance Through a Transparent SIMD Execution." *2018 31st Symposium on Integrated Circuits and Systems Design (SBCCI)*. IEEE, 2018.

Jordan, Michael Guilherme, Tiago Knorst, Julio Vicenzi and Mateus Beck Rutzig. "Runtime Vectorization of Conditional Code and Dynamic Range Loops to ARM NEON Engine." *VII Brazilian Symposium on Computing Systems Engineering (SBESC)*. 2018 (forthcoming).

Jordan, Michael Guilherme, Tiago Knorst, Julio Vicenzi and Mateus Beck Rutzig. "Boosting SIMD Benefits through a Run-time and Energy Efficient DLP Detection." *2019 Design, Automation and Test in Europe (DATE)*. 2019 (forthcoming).

Kaeli, David, and Pen-Chung Yew, eds. *Speculative execution in high performance computer architectures*. Vol. 6. CRC Press, 2005.

Kim, Changmoo, et al. "ULP-SRP: Ultra low power Samsung Reconfigurable Processor for biomedical applications." *Field-Programmable Technology (FPT), 2012 International Conference on*. IEEE, 2012.

Lomont, Chris. "Introduction to intel advanced vector extensions." *Intel White Paper* (2011): 1-21.

Melnik, Dmitry, et al. "A case study: optimizing GCC on ARM for performance of libevas rasterization library." *Proceedings of GROW* (2010).

Mitra, Gaurav, et al. "Use of SIMD vector operations to accelerate application code performance on low-powered ARM and Intel platforms." *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*. IEEE, 2013.

Mittal, Sparsh. "A survey of value prediction techniques for leveraging value locality." *Concurrency and computation: practice and experience* 29.21 (2017): e4250.

Nakamura, Takashi, Satoshi Miki, and Shuichi Oikawa. "Automatic vectorization by runtime binary translation." *Networking and Computing (ICNC), 2011 Second International Conference on*. IEEE, 2011.

Nuzman, Dorit, et al. "Vapor SIMD: Auto-vectorize once, run everywhere." *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2011.

Nuzman, Dorit, Ira Rosen, and Ayal Zaks. "Auto-vectorization of interleaved data for SIMD." *ACM SIGPLAN Notices* 41.6 (2006): 132-143.

Nuzman, Dorit, and Ayal Zaks. "Outer-loop vectorization: revisited for short SIMD architectures." *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 2008.

Nvidia, C. U. D. A. "Nvidia cuda c programming guide." Nvidia Corporation 120.18 (2011): 8.

Patterson, David A., and John L. Hennessy. *Computer Organization and Design MIPS Edition: The Hardware/Software Interface*. Newnes, 2013.

Pohl, Angela, Biagio Cosenza, and Ben Juurlink. "Control Flow Vectorization for ARM NEON." *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems*. ACM, 2018.

Reddy, Venu Gopal. "Neon technology introduction." ARM Corporation (2008).

Russell, Richard M. "The CRAY-1 computer system." *Communications of the ACM* 21.1 (1978): 63-72.

Shin, Jaewook. "Introducing control flow into vectorized code." *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*. IEEE Computer Society, 2007.

Smith, James E., and Andrew R. Pleszkun. "Implementation of precise interrupts in pipelined processors." *25 years of the international symposia on Computer architecture (selected papers)*. ACM, 1998.

Sui, Yulei, et al. "Loop-Oriented Pointer Analysis for Automatic SIMD Vectorization." *ACM Transactions on Embedded Computing Systems (TECS)* 17.2 (2018): 56.

Sun, Xian-He, and Yong Chen. "Reevaluating Amdahl's law in the multicore era." *Journal of Parallel and Distributed Computing* 70.2 (2010): 183-188.

Tian, Xinmin, et al. "Compiling C/C++ SIMD extensions for function and loop vectorization on multicore-SIMD processors." *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*. IEEE, 2012.

Wall, David W. *Limits of instruction-level parallelism*. ACM, 1991.

Wu, Peng, Alexandre E. Eichenberger, and Amy Wang. "Efficient SIMD code generation for runtime alignment and length conversion." *International Symposium on Code Generation and Optimization*. IEEE, 2005.

Zhou, Hao, and Jingling Xue. "Exploiting mixed SIMD parallelism by reducing data reorganization overhead." *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. ACM, 2016.