UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

TIAGO TREVISAN JOST

SoMMA – A Software-managed Memory Architecture for Multi-issue Processors

Dissertation presented in partial fulfillment of the requirements for the degree of Master in Computer Science

Advisor: Prof. Dr. Luigi Carro
Co-advisor: Prof. Dr. Gabriel Luca Nazar

Porto Alegre, October 2017

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Jost, Tiago Trevisan

SoMMA – A Software-managed Memory Architecture for Multi-issue Processors / Tiago Trevisan Jost. – 2017.

82 f.:il.

Orientador: Luigi Carro; Co-orientador: Gabriel Luca Nazar.

Dissertação (Mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2017.

1. Geração de Código. 2. Limitação na Banda de Memória 3. Memória gerenciada por software. I. Carro, Luigi. II. Nazar, Gabriel Luca. III. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Rui Vicente Oppermann
Vice-Reitor: Prof. Jane Fraga Tutikian
Pró-Reitor de Pós-Graduação: Prof. Celso Giannetti Loureiro Chaves
Diretor do Instituto de Informática: Profa. Carla Maria Dal Sasso Freitas
Coordenador do PPGC: Prof. João Luiz Dihl Comba
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

AGRADECIMENTOS

This work would not have been possible without the support and comprehension on a great number of people. I would like to start by thanking my colleagues and friends from the Embedded Systems Group at INF/UFRGS with whom I spent most of my time this two-year-and-so period of my Master’s. I would not dare naming them individually as some people may be forgotten.

Secondly, my most sincere gratitude to my advisors, Luigi and Gabriel, for their dedication, knowledge, comprehension and support during the whole period, and also for understanding my choice of postponing the defense in six months. I would also like to thank the people at CEA, France, where I spent my last six months as an intern, for greatly welcoming me at their company. My deepest thanks also go to all my friends outside the academic community, people from Faxinal, Porto Alegre, and everywhere, who were always there for me in times of need.

This work would have never been possible without the unconditional love and support I received from my parents, siblings and my whole family. I could never repay for all the great things you do/did/will do to me. Lastly, my heart is felt with join for sharing this dissertation with my (soon-to-be) wife, Janaina, who always gave the strength, love, and caring I needed, and for always being there for me no matter what.

SoMMA – A Software-managed Memory Architecture for Multi-issue Processors

RESUMO

Processadores embarcados utilizam eficientemente o paralelismo a nível de instrução para atender as necessidades de desempenho e energia em aplicações atuais. Embora a melhoria de performance seja um dos principais objetivos em processadores em geral, ela pode levar a um impacto negativo no consumo de energia, uma restrição crítica para sistemas atuais. Nesta dissertação, apresentamos o SoMMA, uma arquitetura de memória gerenciada por software para processadores embarcados capaz de reduz consumo de energia e *energy-delay product* (EDP), enquanto ainda aumenta a banda de memória. A solução combina o uso de memórias gerenciadas por software com a cache de dados, de modo a reduzir o consumo de energia e EDP do sistema. SoMMA também melhora a performance do sistema, pois os acessos à memória podem ser realizados em paralelo, sem custo em portas de memória extra na cache de dados. Transformações de código do compilador auxiliam o programador a utilizar a arquitetura proposta. Resultados experimentais mostram que SoMMA é mais eficiente em termos de energia e desempenho tanto a nível de processador quanto a nível do sistema completo. A técnica apresenta *speedups* de 1.118x e 1.121x, consumindo 11% e 12.8% menos energia quando comparando processadores que utilizam e não utilizam SoMMA. Há ainda redução de até 41.5% em EDP do sistema, sempre mantendo a área dos processadores equivalentes. Por fim, SoMMA também reduz o número de *cache misses* quando comparado ao processador *baseline*.

Palavras-chave: geração de código, paralelismo a nível de instrução, limite na banda de memória, processador de despacho múltiplo, memória gerenciado por software.

SoMMA – A Software-managed Memory Architecture for Multi-issue Processors

ABSTRACT

Embedded processors rely on the efficient use of instruction-level parallelism to answer the performance and energy needs of modern applications. Though improving performance is the primary goal for processors in general, it might lead to a negative impact on energy consumption, a particularly critical constraint for current systems. In this dissertation, we present SoMMA, a software-managed memory architecture for embedded multi-issue processors that can reduce energy consumption and energy-delay product (EDP), while still providing an increase in memory bandwidth. We combine the use of software-managed memories (SMM) with the data cache, and leverage the lower energy access cost of SMMs to provide a processor with reduced energy consumption and EDP. SoMMA also provides a better overall performance, as memory accesses can be performed in parallel, with no cost in extra memory ports. Compiler-automated code transformations minimize the programmer's effort to benefit from the proposed architecture. Our experimental results show that SoMMA is more energy- and performance-efficient not only for the processing cores, but also at full-system level. Comparisons were done using the ρVEX processor, a VLIW reconfigurable processor. The approach shows average speedups of 1.118x and 1.121x, while consuming up to 11% and 12.8% less energy when comparing two modified processors and their baselines. SoMMA also shows reduction of up to 41.5% on full-system EDP, maintaining the same processor area as baseline processors. Lastly, even with SoMMA halving the data cache size, we still reduce the number of data cache misses in comparison to baselines.

Keywords: code generation process, instruction-level parallelism, memory bandwidth limitation, multi-issue processors, software-managed memory

LIST OF FIGURES

Figure 1.1 – Processor and Memory Performance improvements over the years. The memory wall significantly increased since 1980, as memories focused efforts on storage capacity. Since 2005, single-core processors showed no performance improvements in favor of a multi-core approach. 13
Figure 1.2 – 32 KB cache memories with different number of ports. Values were normalized over single port. Latency and energy attributes increase significantly in multi-ported caches. 14
Figure 1.3 – Normalized Speedup on a VLIW processor. Some applications struggle with limited memory bandwidth. By increasing the number of memory ports, applications on multi-ported systems can perform considerably better than on single-ported. 15
Figure 2.1 – Constraints comparison between a 32KB Scratchpad and a 32KB Cache. We observe a major difference in terms of access time, dynamic read and write energies per access. .. 19
Figure 2.2 – (a) Simplest view on the compilation Process and (b) The correct set of tools used during the compilation process 21
Figure 2.3 – A three-parted compilation process. 22
Figure 2.4 – LLVM Compilation Overview. 23
Figure 2.5 – Clang diagnostic messages comparison with GCC 25
Figure 2.6 – C Code transformed into SSA-form. A PHI node is used to 26
Figure 2.7 – Middle-end relationship overview 27
Figure 2.8 – Code generation Phase 28
Figure 2.9 – Code examples for two VLIW processors 31
Figure 2.10 – Three primary units of Out-Of-Order processors 33
Figure 3.1 – Synergistic processing element on the IBM cell processor. Scratchpads are referred as "Local Store" 37
Figure 3.2 – A 4-issue VLIW processor example. Previous solutions are represented by the red-dotted connection between lane 0 and scratchpad (SPM), while the proposed solution is represented by the blue-dashed connection between software-managed memories (SMMs) and lanes. Our solution uses memories in parallel in order to leverage the multi-issue characteristic of the processor. We can load and store values in multiple memories at once. 40
Figure 4.1 – Location of the memory architecture in a 4-issue VLIW processor. Each lane has access to an internal memory 42
Figure 4.2 – Illustrative example for controlling SMMs 43
Figure 4.3 – Snippet of 7x7 matrix multiplication 44
Figure 4.4 – Snippet of assembly code 45
Figure 4.5 – Offset address locations for a 7x7 matrix multiplication. The example shows how matrix smm_a (shortened to a) and smm_b (shortened to b) would be placed in the SMMs. First

scheme, on the left, shows an allocation through column order, while the second, on the right, displays an allocation through row order 47
Figure 4.6 – High-level algorithm for Variable Discovery 49
Figure 4.7 – Tree height reduction algorithm........ 52
Figure 5.1 – Workflow diagram for experiments. 58
Figure 5.2 – Normalized Speedup 63
Figure 5.3 – Normalized dynamic energy comparison. Baselines have 8KB and 32KB of data cache when compared to SoMMA 4KB and SoMMA 16KB, respectively. 65
Figure 5.4 – Normalized number of accesses for the SoMMA processors in comparison to the baseline 65
Figure 5.5 – Compiler generates lane-specific nops when no instruction is executed. The example show that lanes 5, 6 and 7 have a nop after nop sequence 61
Figure 5.6 – Normalized Energy in a full-system configuration 66
Figure 5.7 – EDP reduction in a full-system configuration 68
Figure 5.8 – Data cache miss reduction 69
Figure 5.9 – DAG Heights for the main BBs with their optimized values 71
Figure 5.10 – Small snippet of C-code for a FIR filter 72
Figure 5.11 – Data placement on the SMMs for FIR 73
Figure 5.12 – Performance Speedup for a handwritten FIR in SoMMA 74

LIST OF TABLES

Table 2.1 – Categorization of Multi-issue Processors. 30
Table 5.1 – Execution time increase when adding preamble code.. 64
Table 5.2 – Energy cost per access in Caches and SMMs 59

LIST OF ABBREVIATONS AND ACRONYMS

ALU	Arithmetic and Logic Unit
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuits
AVX	Advanced Vector Extensions
BB	Basic Block
BFS	Breadth-First Search
BRAM	Block RAM
CASA	Cache-Aware Scratchpad Allocation
CU	Compute Unit
DAG	Direct Acyclic Graph
DDG	Data-Dependence Graph
DFT	Discrete Fourier Transform
DLP	Data-Level Parallelism
DRAM	Dynamic Random Accesses Memories
EDP	Energy-Delay Product
FPGA	Field-Programmable Gate Array
FIR	Finite Impulse Response
FU	Functional Unit
GPU	Graphics Processing Unit
HLL	High-Level Programming Language
ILP	Instruction-Level Parallelism
IR	Intermediate Representation
ISA	Instruction Set Architecture
SRAM	Static Random Access Memory
LLVM	Low-Level Virtual Machine
LEAP	Logic-based Environment for Application Programming
LPDDR2	Low-power DDR2
LVT	Live Value Table
MIMD	Multiple Instruction Multiple Data
O3	Out-Of-Order
OO	Object-Oriented
RTS	Real-Time Systems

SAD	Sum of Absolute Differences
SIMD	Single Instruction Multiple Data
SMM	Software-Managed Memory
SPE	Synergistic Processing Unit
SSA	Static-Single Assignment
SSE	Streaming SIMD Extensions
TLP	Thread-Level Parallelism
VEX	VLIW Example
VLIW	Very-Long Instruction Word
WCET	Worst-Case Execution Time

TABLE OF CONTENTS

1 INTRODUCTION 12
1.1 Memory Wall and Memory Hierarchies 12
1.2 Memory Bandwidth Limits, Memory Ports and their Impacts 14
1.3 Main Goals and Contributions 16
1.4 Outline 17
2 BACKGROUND 18
2.1 Software-managed Memories 18
2.1.1 Types of Scratchpad Memory 19
2.1.2 Types of Allocation 20
2.2 Compilation Process 20
2.2.1 The Structure of Compiler 21
2.2.2 LLVM 23
2.3 Multi-issue processors 29
2.3.1 Very-long Instruction Word Processor 31
2.3.2 Superscalar Processor 32
3 RELATED WORK 34
3.1 Scratchpads 34
3.1.1 Academic use 34
3.1.2 Usability in the industry 37
3.2 Multi-ported systems 38
3.3 Our approach x Other methods 39
4 HARWARE AND SOFTWARE SUPPORT 41
4.1 Hardware 41
4.2 Software 42
4.2.1 SMM-specific instructions 43
4.2.2 Preamble Code 44
4.2.3 Example Application 44
4.2.4 Code Generation Process 47
5 EXPERIMENTAL SETUP AND RESULTS 54
5.1 Tools 54
5.1.1 VLIW Example (VEX) architecture 54
5.1.2 LLVM 56
5.1.3 Cacti-p 56
5.1.4 DRAMPower 57
5.2 Methodology 57
5.2.1 Experimental Setup 57
5.2.2 Benchmark Selection 62
5.3 Experimental Results 62
5.3.1 Performance 62
5.3.2 Energy Consumption 64
5.3.3 Energy-delay Product (EDP) 68
5.3.4 Data Cache Misses 69
5.3.5 DAG Analysis 70
5.3.6 Potential for improvements: Finite Impulse Response (FIR) Filter Case 71
6 CONCLUSIONS AND FUTURE PERSPECTIVES 75
6.1 Future works 75

6.2 Publications **76**
REFERENCES **77**

1 INTRODUCTION

1.1 Memory Wall and Memory Hierarchies

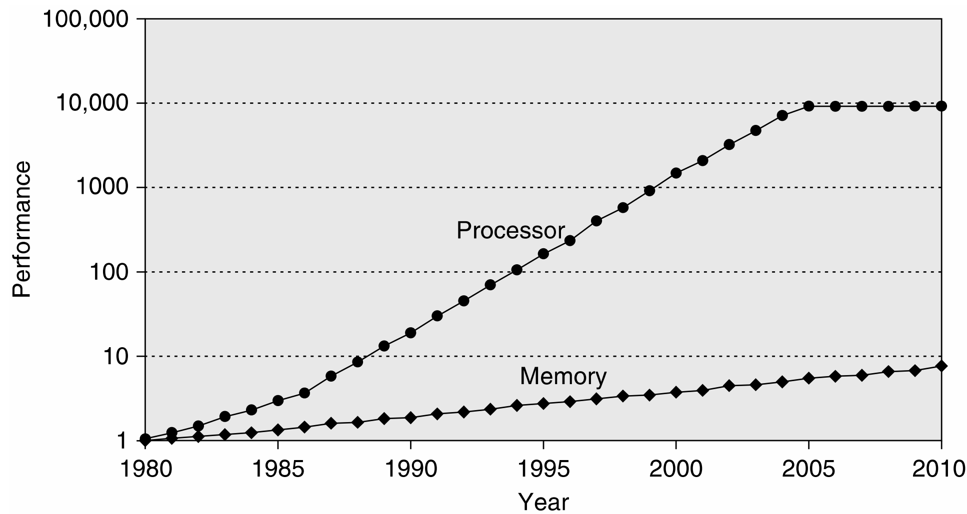
Technology scaling has allowed computer architects to exponentially increase the frequency of processors for many years. Due to power restrictions, however, the hardware industry was forced to switch to a multi-processor approach, demanding the usage of even higher memory bandwidth. On the contrary direction, memory manufacturers were primarily focused on increasing storage capacity and did not achieve the same order of speed improvement, becoming one of the bottlenecks for increasing the overall performance of a system. This adverse scenario led to an extensive concern on developing smart solutions to the limited memory bandwidth problem. Such solutions include the exploration of temporal and spatial locality through memory hierarchy, and instruction and data prefetching. Although improvements have been made, the gap between processor and memory performance is still significant.

The concept of “Memory Wall” (WULF; MCKEE, 1995), which states that the rate of improvement in DRAM speed does not follow the improvement in processor speed, became a challenge for the community in general. Figure 1.1 illustrates performance improvements in single-core processors and dynamic random accesses memories (DRAM) from 1980 to 2010, showing that the memory wall has effectively increased in the past decades. Even with the performance stabilization in single-core performance from 2005 and to present days, due to restrictions in nanometer technology, there is a significant performance gap between processors and DRAM devices.

The addition of on-chip cache in between main memory and processors’ registers has a major contribution in minimizing the existing gap between the memory system and processors. Caches were an important additional component to the system that leverage concepts of spatial and temporal locality to benefit processors. Spatial locality states that a system tends to reuse data that are close from one another, while temporal locality refers to the fact the values might also be used later, that way they should be kept near the processor.

One of the main advantages of using caches comes from the genericity they give to the application. Although there are efficient ways for software to take advantage of them, their control is done at hardware-level, taking the burden away from the software. Data that remain in the cache are basically decided depending on a set of hardware components, such as cache size, number of sets, replacement algorithm and so on (HENNESSY; PATTERSON, 2011).

Figure 1.1 – Processor and Memory Performance improvements over the years. The memory wall significantly increased since 1980, as memories focused efforts on storage capacity. Since 2005, single-core processors showed no performance improvements in favor of a multi-core approach.



Source: HENNESSY and PATTERSON (2011)

The long-lasting gap between memory bandwidth and processors was minimized when different levels of memories were introduced, creating a hierarchy of memory units. This hierarchy is not only composed of caches, but of several memories which exhibit different technologies, speed, and cost per byte. Three technologies are commonly used to build memory hierarchies. Memories closer to the processor, and therefore faster and more expensive, use static random access memory (SRAM) technology, while main memory is implemented with DRAM. The third technology used is either magnetic disks or flash memory and is considerably slower than SRAM or DRAM, however, it presents a much higher density which allows for a higher storage capacity than other technologies.

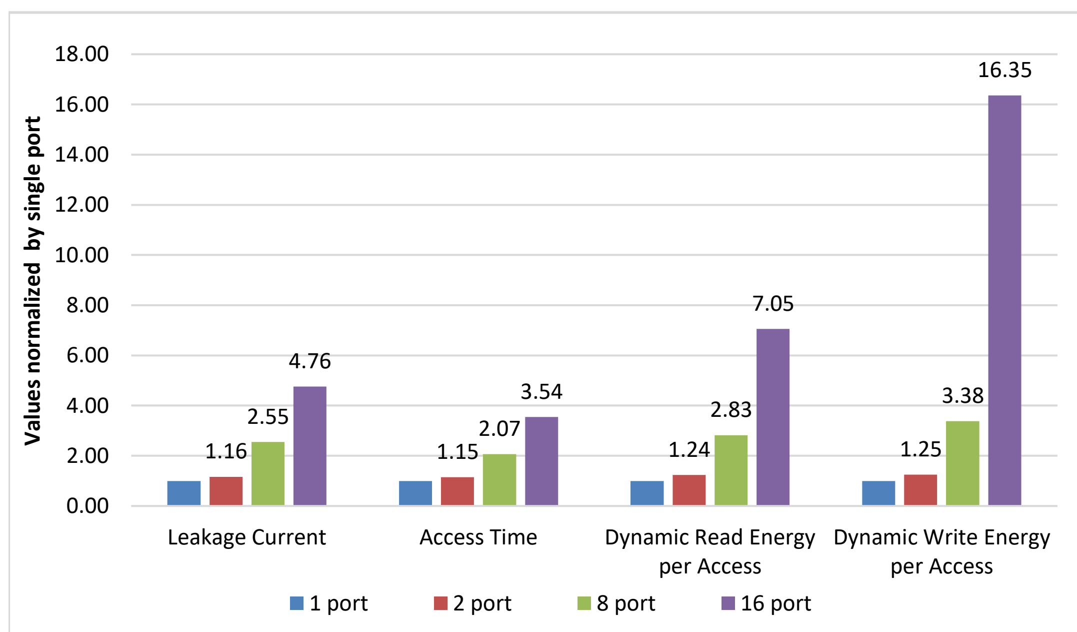
The proximity of memory to the processor leads to a higher cost per byte, which explains why caches are more expensive and have smaller storage capacity than other technologies. The goal is to provide a memory system with cost per byte almost as low as the cheapest level of memory and speed almost as fast as the fastest level (HENNESSY; PATTERSON, 2011). For many cases, however, the access time of memory, even with caches, still bounds applications to improve performance and exploit instruction-level parallelism (ILP), as different levels of memories provide distinct latencies.

1.2 Memory Bandwidth Limits, Memory Ports and their Impacts

Memory bandwidth has long been a limiting factor in performance for many applications. Processors exploiting parallelism, either in instruction or thread level, may need to perform multiple accesses in a single clock cycle to maintain parallelism, and thus, memories with multiple ports could be used to achieve that goal. Even when exploiting access locality through cache memories to minimize this limitation, multi-ported memories are known to introduce significant costs. The study in (TATSUMI; MATTAUSCH, 1999) shows that the impact of increasing the number of ports in a memory system leads a quadratic increase for cell area. Additionally, extra memory ports would also implicate on higher dynamic and static energy consumption, which are especially undesirable for embedded systems, as they are often more power- and energy-constrained than general-purpose computing platforms.

Figure 1.2 illustrates leakage current, access time, dynamic and static energy for 32 KB cache memories with 1, 2, 8, and 16 ports, in a 65nm technology obtained with the Cacti-p tool (LI, S. *et al.*, 2011). We can observe how power and energy attributes escalate in multi-ported memories. For instance, the 16-port cache showed dynamic and static energy increases of more than 7x and 16x, respectively, and even when considering a 2-ported cache, an increase of 25% in static energy might be crucial for embedded processors. Moreover, an access time increase

Figure 1.2 – 32 KB cache memories with different number of ports. Values were normalized over single port. Latency and energy attributes increase significantly in multi-ported caches.



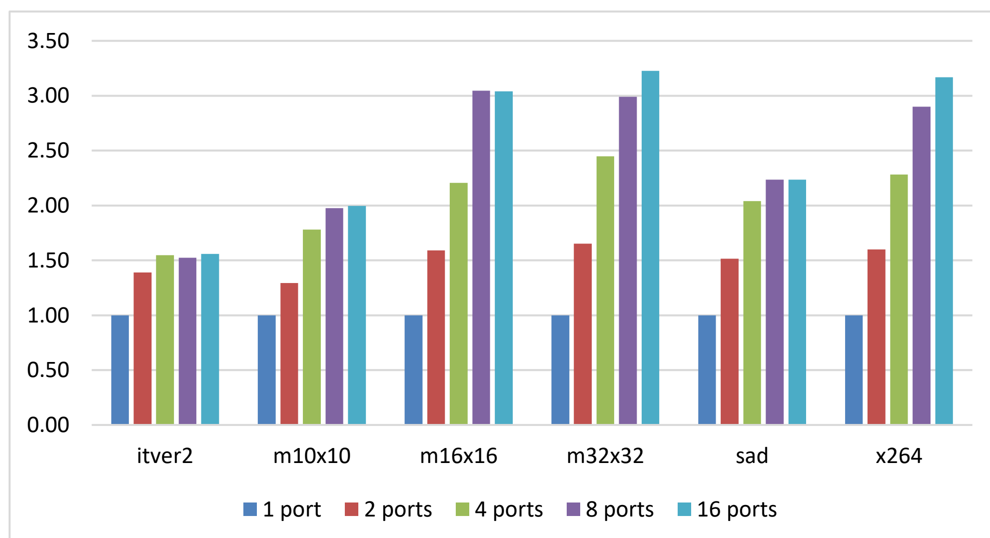
Source: author

of 15% in a 2-ported cache can possibly compromise energy consumption, as it may increase applications' execution time.

On the other hand, many applications still struggle with memory bandwidth and have their parallelism limited by the number of ports on the cache. The presence of extra ports can potentially improve instruction-level parallelism (ILP) on applications, and therefore, accelerate their execution. For instance, image and video processing applications usually offer high parallelism opportunities due to the sparse control-flow instructions. Figure 1.3 demonstrates the speedup of some applications in a very-long instruction word (VLIW) processor when one uses extra cache ports. These results illustrate how the additional ports can have a huge impact on the performance of applications, with some applications being over 3x faster in a 16-ported system.

Due to the end of Dennard scaling (DENNARD *et al.*, 1974), concerns over energy consumption play a major role in processor design. Techniques to improve performance might have a negative impact on energy consumption, thus the energy-delay product (EDP) comes into play as a metric to evaluate the quality of a system, combining the influence of performance and energy. When considering EDP, the improvements in performance would not justify the high cost of energy consumption shown in Figure 1.2 for the set of applications in Figure 1.3.

Figure 1.3 – Normalized Speedup on a VLIW processor. Some applications struggle with limited memory bandwidth. By increasing the number of memory ports, applications on multi-ported systems can perform considerably better than on single-ported.



Source: author

1.3 Main Goals and Contributions

In this dissertation, we present a new memory architecture based upon software-managed memories called SoMMA (Software-Managed Memory Architecture), that aims at reducing energy consumption and energy-delay product (EDP). SoMMA also provides an increase in memory bandwidth, without the use of multi-ported memories. Its main characteristic is the replacement of a larger cache by a combination of a smaller cache and a set of software-managed memories (SMMs) in such a way that area remains equivalent in both processors with and without the SMMs. We use a compiler-based approach to manage the extra memories and accelerate the execution of applications. Previous works like (VERMA; WEHMEYER; MARWEDEL, 2004) and (STEINKE *et al.*, 2002) proposed to use software-managed memories, also called software-controlled memories or scratchpads, as a replacement for caches aiming at reducing energy consumption; others (ANGIOLINI *et al.*, 2004; AVISSAR; BARUA; STEWART, 2002) focus on boosting the performance of embedded processors due to the reduction of cache misses.

A major advantage of SoMMA in comparison to previous strategies relies on the usage of multiple memories in parallel, inducing an increase on the total bandwidth available and creating new opportunities for ILP exploitation and energy savings. Other approaches mainly obtain gains in execution time due to avoidance of cache misses, while we also cover that, we go further by exploiting ILP through parallel accesses to memories. SoMMA also differs on how address spaces are treated. Scratchpads from previous works typically use the same address space of the memory hierarchy, i.e., a range of addresses is scratchpad-addressable, while others are dealt through the cache. On the other hand, our memories have unique address spaces, with no correlation among them whatsoever. In many cases the compiler can identify the address space being accessed by each instruction, allowing the parallelization of accesses performed to different spaces, which is crucial to the energy savings we can achieve. Furthermore, the total available bandwidth can be increased, creating new opportunities for ILP exploitation.

Although our technique relies mostly on a complex compiler approach rather than at hardware-level, our compiler transformations are merely the means to our solution. We use a solution that incorporates changes in both hardware and software and can alleviate the limit of single-port systems, providing a higher throughput. We demonstrate the efficiency of our technique in a very long instruction word (VLIW) processor, even though our technique could be adapted to other ILP-capable architectures, such as superscalar processors. Our approach can be used for application-specific integrated circuits (ASIC) processors and those built on top

of field-programmable gate arrays (FPGA), as they show the same concerns regarding bandwidth limitation. Experimental results show performance improvements while maintaining the same area of the standard memory hierarchy. Moreover, we analyze the energy efficiency for our memory system, showing significant energy and energy-delay product (EDP) reductions for the data memory hierarchy in comparison to a regular processor.

1.4 Outline

This dissertation is organized as follows. Chapter 2 presents the main ideas behind software-managed memories, how they can be categorized and a comparison with caches; we also present the main concepts of compilers and an introduction to the Low-level Virtual Machine (LLVM) framework (LATTNER; ADVE, V., 2004), a powerful framework to build compilers on which our approach relied. Moreover, we briefly introduce two types of multi-issue processors, superscalars and VLIW processors, since we can implement our technique in both. Chapter 3 presents previous works on software-managed memories and state their difference with our proposal. This chapter also covers previous techniques to design multi-ported systems more efficiently, as they can help overcome the limiting bandwidth in the memory system. Our implementation is described in Chapter 4, in which we give the hardware and software requirements, explaining the hardware changes on the processor side and the main component of the architecture: the automated code generation for software-managed memories. The experimental results carried out in this dissertation are presented in Chapter 5. Conclusions drawn from this work are encountered in Chapter 6, as well as the future works envisioned from it.

2 BACKGROUND

In this chapter, we will present the concepts that will be used during most part of this dissertation. We will discuss the main characteristics of scratchpads and multi-issue processors, both necessary herein. Even though the compiler is not the major contribution of our work, but rather the facilitator, a discussion on compiler theory and code generation is also presented, as well as an introduction to the LLVM Compiler Framework (LATTNER; ADVE, V., 2004), a powerful framework for building complex and advanced compilers and perform code optimizations.

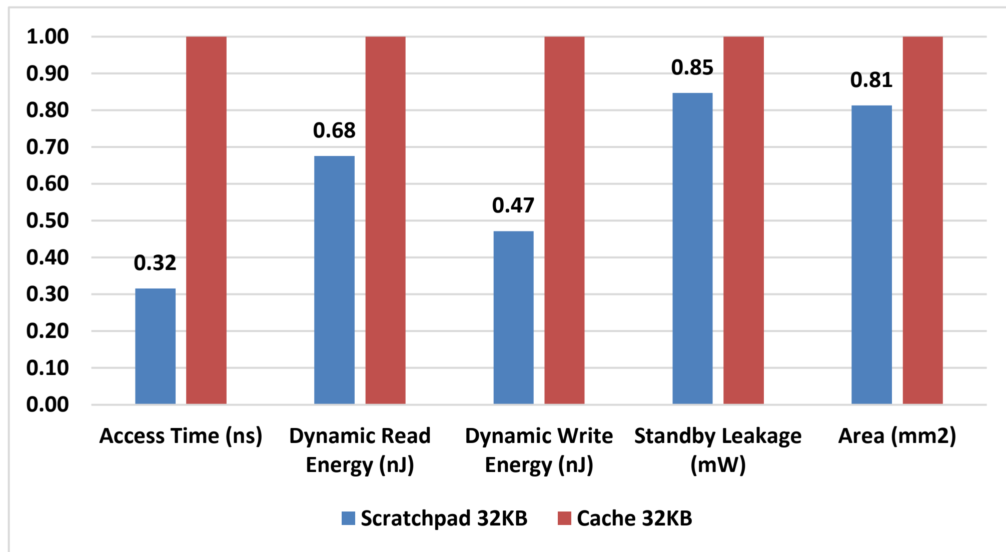
2.1 Software-managed Memories

As presented in Chapter 1, the addition of memory ports in the system would collaborate to reduce the bandwidth gap between memories and processors, albeit with a high cost in area. Caches are the most common type of memory used in today's system, due to their flexibility. The complexity of caches, which include comparators and tag arrays, adds an extra layer of energy dissipation in the system. Applications recurrently use a memory access pattern that could be understood by software. This process would alleviate the use of comparators and tag arrays, and as a consequence, save energy on the access of the cache. This concept has already been explored with the use of software-managed memories, with the aim at reducing access to the cache and thus, minimizing energy and power consumption (VERMA; MARWEDEL, 2006; VERMA; WEHMEYER; MARWEDEL, 2004). As many characteristics of an application cannot be decided in compile time, scratchpads are still limited to being used in situations where the pattern accesses are previously known.

Figure 2.1 shows a comparison between a 32-KB 4-way associative cache with a block size of 64 bytes and 32-KB scratchpad using Cacti-P (LI, S. *et al.*, 2011). We evaluated these memories in terms of area, access time, static power, dynamic read energy per access, and dynamic write energy per access. The scratchpad shows major reduction of 68% in access time, 32% in dynamic read energy and 53% in dynamic write energy, while still reducing 15% and 19% in static power and area, respectively. These results points towards the significant leverage that scratchpads have in comparison to caches.

Another aspect of scratchpads which favors their usage is predictability. Real-time embedded systems usually are bounded by a maximum acceptable time for executing a given task. The Worst-Case Execution Time (WCET) constraint is deemed very important in such

Figure 2.1 – Constraints comparison between a 32KB Scratchpad and a 32KB Cache. We observe a major difference in terms of access time, dynamic read and write energies per access.



Source: author

systems, and software-managed memories show a predictability that is easily trackable. Caches, on the other hand, are frequently time-unpredictable, since it is difficult to know whether certain data are currently available for fast access. Scratchpads simplify WCET analysis, a type of compiler analysis which computes upper bounds for execution time. The knowledge of the maximum time consumption of all pieces of code and data is a prerequisite for analyzing the worst-case timing behavior of a real-time systems and for verifying its temporal correctness (PUSCHNER; BURNS, 2000).

In the chapters that follow, we will be using the terms scratchpads and software-managed memories interchangeably, as they all have the same features.

2.1.1 Types of Scratchpad Memory

Software-controlled memories can be classified according to different characteristics. In terms of types of memories, they can be used as a replacement or as a complement to the data caches, instruction caches, or both at the same time. Because they use fewer resources than caches, as depicted in Figure 2.1, they can show improvements in performance, energy consumption and area, although without the same flexibility encountered on caches.

2.1.2 Types of Allocation

Two main schemes for allocating data into software-managed memories exists: the static and overlay-based approach. In the static-based approach, data are loaded once at the beginning of the program and remain invariant during its entire execution. Despite its simplicity, this method limits the usability of the memories considerably. When one considers that applications may use data only for a small part of their execution time, a static approach will not be efficient for them. The latter, on the other hand, tackles most of the inefficiency of the static-based method. The content of the software-managed memory changes dynamically during the execution of the program. This approach offers a flexibility that is absent from the static counterpart, as different data can be stored in a same location depending on the program point.

2.2 Compilation Process

The process of translating high-level programming languages (HLL) like C, C++, among others, to machine code is one of the cornerstones for augmenting the complexity of systems. Compilers have become an indispensable tool for any computer system inasmuch as companies need to accelerate the time-to-market of their products in order to keep the competition with other companies. The ever-changing process of manufacturing processors have made possible to accommodate multiple specialized units in the same chip, leading to addition of functionalities that required a more demanding process of compilation that takes advantage of those units.

As new paradigms and programming interfaces were created, such as OpenMP (CHANDRA, 2001), OpenACC (WOLFE, 2013), and OpenMPI (FORUM, 1994), compilers are now responsible not only for translating languages but also for understanding these paradigms making necessary changes in the code. For instance, OpenMP is an application programming interface (API) to construct parallel programs that relies on using of *C/C++ pragma* directives to instruct compilers to explore parallelism in the code. The compiler uses these directives to understand how the program should be parallelized, and infers calls to the runtime OpenMP library, which is responsible for creating and managing threads on the program (CHANDRA, 2001). Another example on the importance of the compiler in aiding programmers is commonly found in objected-oriented (OO) languages like C++. Whenever a class is user-defined without a constructor or destructor, the compiler needs to automatically generate code for such, saving time of programmers. It becomes mandatory to understand the

main aspects of the compilation, as it is a major part of our approach. In the next paragraphs, we will discuss a series of steps that compilers perform to generate machine code.

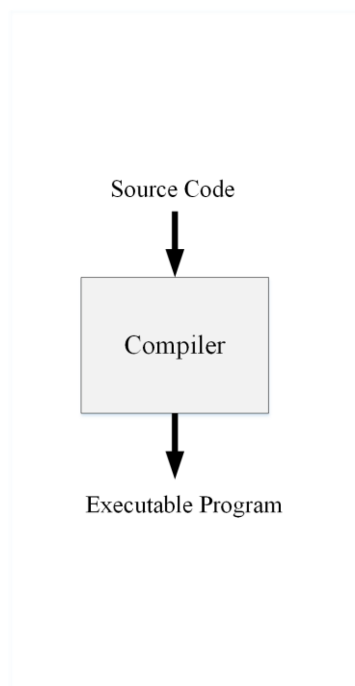
2.2.1 The Structure of Compiler

Compilers have a rather complex structure. There is a misconception among computer scientists and engineers in which part of the code generation compilers are responsible for. Figure 2.2(a) illustrates the simplest view of how a compiler works, and which is believed to be true by many programmers. From a source code written in a HLL, the compiler translates it into executable code. However, through a further look, the compilation process involves more than just the usage of the compiler. An assembler and linker are also necessary.

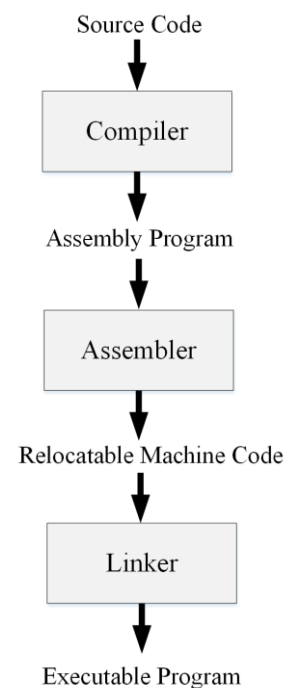
The task of the compiler is, in a deeper level, represented in Figure 2.2b. Through a series of analysis and transformations, its task is to generate target assembly code from a source code written in a HLL. Next, from the assembly code, the assembler generates what is known as relocatable machine code or object file, where internal references and addresses are resolved. Later, the linker is responsible for uniting all object files into one executable file, resolving external references and addresses issues with static or dynamic libraries. At last, the operating

Figure 2.2 – Two views on the Compilation Process

(a) Simplest view



(b) The correct set of tools used



Source: author

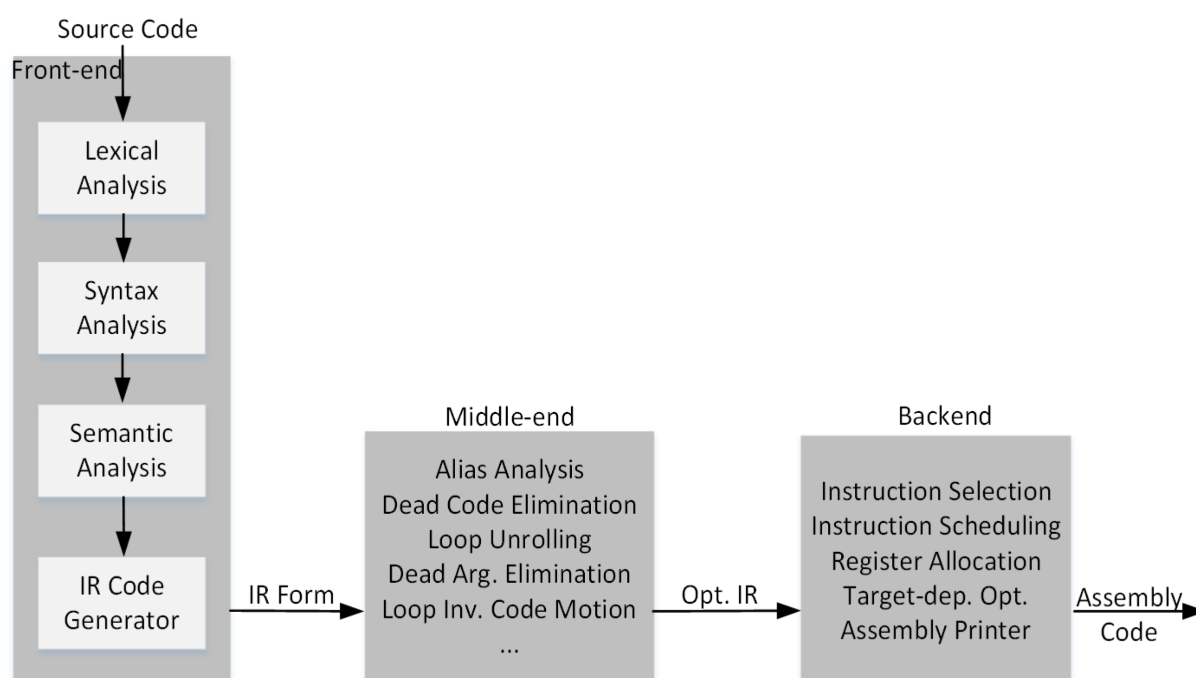
system hands off the executable program to the loader, not depicted in the figure, which places the program into memory for execution.

Popular compilers, like GCC and Clang, seem to perform assembling and linking to programmers, however, they are redirected to system's tools. In a linux-based OS, those tools are usually part of the *binutils* set of tools, and are named *as* (GNU Assembler), *ld* (GNU Linker). Because they are called directly by the compiler, programmers have the tendency to think that compilers do all the work, which is mistaken.

Moreover, we will not be focusing on how assemblers or linkers work, we target at understanding the compilation process from source code to assembly code. This process can be divided in three parts. In a nutshell, the *frontend* transforms source code to a more abstract language form, the intermediate representation (IR). Next, the *middle-end* uses the IR to perform code optimizations in order to simplify and optimize the program to make it run faster. At last, the *backend* generates the assembly code for the program. The compilation process is illustrated in Figure 2.3, detailing what tasks each part performs.

In the next paragraphs, we cover the basics of frontend, middle-end and backend through the perspective of LLVM, a powerful framework designed to explore code analysis and transformation at different level of abstractions during the lifetime of a program.

Figure 2.3 – A three-parted compilation process.



Source: author

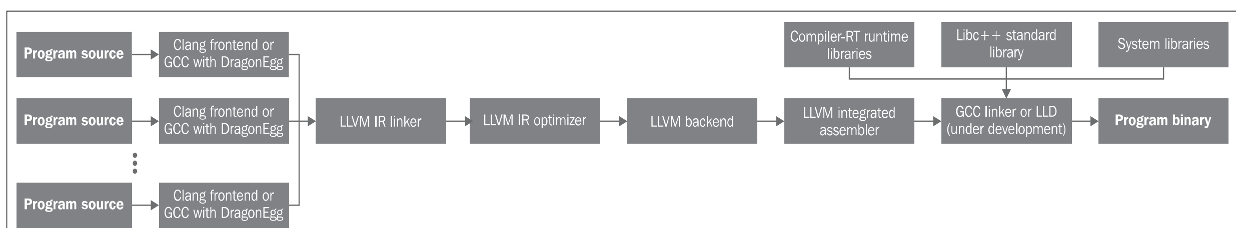
2.2.2 LLVM

LLVM is a well-known compiler framework and runtime environment consisting of a collection of modules to build, run and optimize programs (LATTNER; ADVE, V., 2004). The framework has been vastly used in academic research and the industry for many years. It has attracted interesting from Apple Inc., that acquired the rights to use LLVM and all its functionalities. Nowadays, the company uses LLVM to build programs for its mobile and desktop platforms (APPLE INC., [s.d.]).

The main characteristic of LLVM is its modular design that provides a simple and effective way of creating code analysis and transformations. The framework is organized in a different collection of tools, each with its own functionalities. Figure 2.4 depicts an overview of the LLVM compilation process from source code to program binary. Except for "Program Source" and "Program binary" boxes, all others are standalone tools part of the LLVM Umbrella project. Some of the most important tools for compiling a source code is as follows:

- *clang* is the official frontend for C, C++ and Objective-C applications. It inputs a source program and outputs the LLVM IR representation of that program, which will be processed by the LLVM IR Optimizer. It is a separate project from the LLVM Umbrella project, where developers may add new languages without interfering in the middle-end and backends as long as it generates correct LLVM IR language.
- *Opt* is the IR Target-independent Optimizer where most of the optimizations take place. The tool works in the IR language, an intermediate representation proposed by the framework. The *opt* tool works in the middle-end.
- *llc* is the LLVM backend tool. It is responsible for generating assembly code and every transformation that is executed during the backend phase (instruction selection, instruction scheduling, register allocation, among others).

Figure 2.4 – LLVM Compilation Overview.



Source: Lopes and Auler (2014)

LLVM has a very modular design, particularly the middle-end and backend that rely on the Pass Framework, an important part of the LLVM system that lets users build and test new functionalities into the platform very easily. In the next subsections, we present an overview of these three phases, as well as the Pass Framework Application Programming Interface (API).

2.2.2.1 The frontend

This phase is responsible for transforming the source code into an abstract target-independent representation, also known as the intermediate representation (IR). The usage of IR is a middle point which frontends produce from an HLL, and backends input from. It simplifies the addition of extra languages in a compiler, as code generation for an architecture is not performed one-by-one, i.e., from one HLL to one architecture. It can be divided into three analysis process: the lexical, syntax and semantics analysis; and a process of IR code generation.

The lexical analysis, also called *scanning,* is responsible for identifying the set of words that will make up the *alphabet* of our input language. The lexical analyzer reads the stream of characters in the source code and groups them into meaningful sequences called *lexemes* (AHO *et al.*, 2006). In other words, it recognizes patterns in a text. The analyzer uses these lexemes to produce *tokens*, sets of characters that have meaning on the specific language. *Tokens* group information about their characteristics on the language, such as, name, value, and type of category.

The syntax analysis, also called *parsing*, uses information produced from the lexical analysis to verify that the produced *tokens* can be generated by the grammar for the input language. Syntax analysis uses a grammatical structure in the form of a tree, known as syntax tree or parse tree, to examine the stream of tokens. Nodes of the tree represent operations and identifiers, while edges represent a dependence between them. The parse tree is built upon a grammar that defines our source language.

The last analysis of the front-end verifies for the semantics of the code. It uses the information of the symbol table and the syntax tree in order to check their consistency with the language definition. It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate code generation (AHO *et al.*, 2006).

The last process is responsible for intermediate code (IR) generation, that is the input given to the middle-end phase. Intermediate code generators commonly use an assembly

representation of target-independent instructions in the form of three address code. This representation uses instructions with three operands (one destination and two sources), which is similar to assembly code in most architectures (AHO *et al.*, 2006). The intermediate code generation aims at transforming the parse tree structures from the semantics analysis process into three address code IR.

Clang is the official frontend of LLVM which outputs LLVM IR code from a high-level program description. Lexical, syntax and semantics analysis, and IR code generation are taken care in this tool. A major advantage of Clang over GCC is on diagnostic report messages. Clang claims to use reports in a more user friendly way, with messages that are more helpful and understandable, and therefore, it can present programmers a better comprehension on exactly what is wrong with their code.

Let us illustrate through Figure 2.5. The example shows a class declaration followed by a structure declaration that misses a semi-colon, and an object instantiation of the created class. It is clear how Clang diagnostic error is more comprehensive than its counterpart. GCC does not fully understand the syntax error and reports a malformation misplaced. On the other hand, GCC supports languages that clang does not, such as Fortran, Ada, Go, many more targets.

2.2.2.2 The middle-end

The most important code optimizations in a compiler takes place in this phase. The compiler's middle-end uses a series of control-flow analysis and data-flow transformations to make the code run faster. The choice of data structures plays an important role on efficiently

Figure 2.5 – Clang diagnostic messages comparison with GCC

```
$ cat t.cc
template<class T>
class a {};
struct b {}
a<int> c;
$ gcc-4.9 t.cc
t.cc:4:8: error: invalid declarator before 'c'
 a<int> c;
         ^
$ clang t.cc
t.cc:3:12: error: expected ';' after struct
struct b {}
           ^
           ;
```

Source: https://clang.llvm.org/diagnostics.html

Figure 2.6 – C Code transformed into SSA-form. A PHI node is used to

```
if (...)                      if (...)
    a = 0                         a_1 = 0
else              ------>     else
    a = 2                         a_2 = 2

/* Use a here /*              a = PHI (a_1, a_2)
```

Source: author

optimizing code. For that reason, middle-ends usually rely on one complex data structure that efficiently offers great opportunities for optimizations: the static-single assignment (SSA) form (CYTRON *et al.*, 1991), used to represent data-flow relationships of the program.

SSA form is a type of intermediate representation, similar to assembly code that facilitates the use of code optimization. This form characterizes for presenting only single-assigned variables, i.e., a variable cannot be assigned more than once. Figure 2.6 presented a simple example of SSA-form, showing that new versions of the same variable are created when the same variables is assigned more than once.

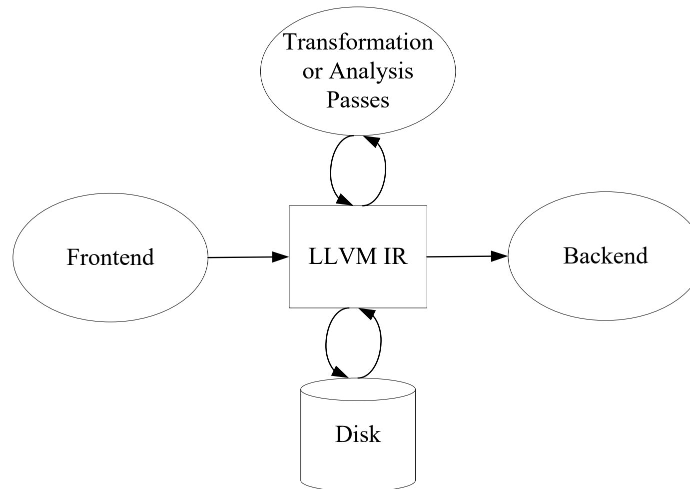
SSA has been adopted in GCC and LLVM due to its characteristics and flexibility. Data-flow can be easily represented, however, in order to represent control-flow, like branches and join nodes, we need to use a special form of assignment called a *ϕ-node (PHI-node)*. Consider the snippet of C code in Figure 2.6, which shows how a PHI-node is used to handle control flow. The value of *a* can come either from the *if* statement or the *else*, depending on the condition. PHI-nodes are important to maintain the property of single assignment of every variable, since a variable's values cannot often be known in compile time.

LLVM defines an assembly-like IR language that can operate all kinds of data structures, from scalars to vectors. The language is as target-independent as possible, but it still requires some target-specific information. For instance, C libraries define the width of data types depending on the architecture they are compiled to. Integers are 16 bits long in a MSP430 while are 32-bit long in a MIPS processor.

The LLVM IR can be represented in three different forms:

- An in-memory representation that is modeled through modules, functions, basic blocks and instruction classes;
- An on-disk human-readable representation very similar to assembly-readable form;
- An on-disk representation encoded in an executable-like style.

Figure 2.7 – Middle-end relationship overview



Source: adapted from Lopes and Auler (2014)

The relationship between the middle-end (LLVM IR) and other components is illustrated in Figure 2.7. We see that the middle-end is centralized to communicate with the front-end, backends and most optimization passes, which makes it the cornerstone for performance improvements.

2.2.2.3 The Backend

The backend, also known as code generator, is the final phase on the compilation process. It is responsible for effectively translating the intermediate representation into target code, or assembly code. The code generator is divided into a series of steps, which may vary depending on the compiler framework, but consists mainly of instruction selection, instruction scheduling, register allocation, assembly printer.

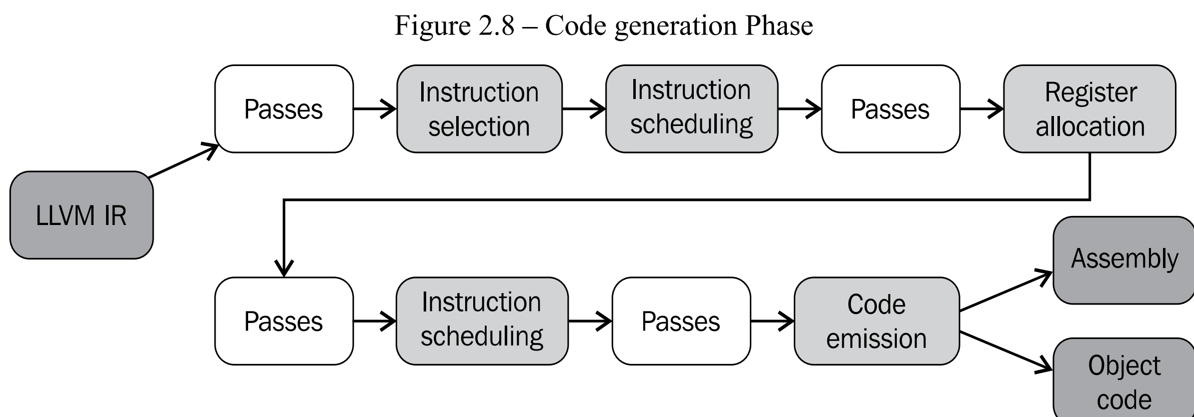
LLVM provides a modular way of building backends. It supports a wide range of targets, such as ARM, AArch64, Hexagon, MSP430, MIPS, SPARC, SystemZ, X86, etc. The backend must follow a series of steps in order to emit target code, depicted in Figure 2.8. Gray boxes represent the necessary phases for code generation inside the backend, while white boxes are strategically positioned so that backend developers can add specific transformation passes to their backends. A brief description of the gray-colored backend phases is as follows:

- The *Instruction Selection* phase converts the in-memory LLVM IR representation to a direct acyclic graph (DAG) representation, internally called

SelectionDAG, whereas nodes mainly represent instructions and operands, while edges represent dependency between them. Each DAG is associated with a single basic block. LLVM defines a non-legalized node as one that has no instruction correspondence in the target. Legalized nodes are the ones that have a target-specific instruction mapping. During this phase, the framework runs a process of DAG legalization, which aims at converting non-legalized into legalized nodes;

- The next phase performs instruction scheduling prior to register allocation. Instructions use virtual registers to represent register-like operands. The scheduler rearranges instructions, trying to maximize parallelism and minimize hazards and delays;
- After instruction scheduling, the register allocation transforms the set of unlimited virtual registers into target-specific registers, spilling to memory whenever is necessary.
- The second instruction scheduling phases takes place after register allocation. Because physical registers are used at this point, this scheduler is deemed not as effective as its pre-register allocation counterpart. However, in case of very-long instruction word process, the decision of grouping instruction in bundles is, in a way, scheduling problem that runs at this stage in LLVM.
- The last stage contemplates code emission. LLVM provides infrastructure to emit assembly and object code. Afterwards, a linker must be used to resolve external address references and generate the executable file.

Figure 2.8 – Code generation Phase



Source: Lopes and Auler (2014)

2.2.2.4 LLVM Pass API

A *LLVM Pass* is a module written in C++ which can perform program analysis or transformation. *Opt* and *llc* tools extensively use passes to add functionalities to the framework. The Pass API defines a hierarchy of C++ Classes that allow developers to implement their own code. The main subclasses of the API are:

- ModulePass: a pass that inherits this class uses an entire program as a unit, having access to functions, its basic blocks and instructions from the IR. It is usually used when a there is an interaction between functions. Optimizations that use it in LLVM are dead argument elimination, partial inlining, among others.
- FunctionPass: uses a function as a unit for processing. The framework calls the pass that inherits from this class for every function in the program. LLVM uses this pass to optimizations such as dead code elimination, constant propagation, etc.
- MachineFunctionPass: it works similarly with FunctionPass, with the difference that handles machine instructions instead of the IR. Therefore, passes from this type will run after instruction selection.
- BasicBlockPass: uses the granularity of basic blocks, thus, a *BasicBlockPass* can only modified its instructions.

This API is used for many of the optimizations that happen the middle end and backend

2.3 Multi-issue processors

Due to technology advances, the prominent use of multi-issue processors is becoming more common than ever. Multi-issue processors characterize by means of issuing more than one instructions at once, aiming at speeding up the execution of applications. Table 2.1 illustrates how these processors can be categorized.

According to the form of parallelism, this classification divides multi-issue processors on Instruction-Level Parallelism (ILP) versus Data-Level parallelism (DLP). ILP addresses the inherent parallelism found within programs, where multiple independent instructions can be executed in parallel. DLP is explored for situations in which no dependence is found within data and thus, they can be processed all together. Examples of ILP-like processors are Very-

Table 2.1 – Categorization of Multi-issue Processors.

	Static: Discovered at Compile time	Dynamic: Discovered at Runtime
Instruction-Level parallelism (ILP)	VLIW	Superscalar
Data-Level parallelism (DLP)	SIMD	Tesla Multiprocessor

Source: Patterson and Henessy (2013)

Long Instruction Word (VLIW) and Superscalars while Single Instruction Multiple Data (SIMD) type processors are DLP driven.

The improvement in manufacturing process and transistor technology have allowed the yield of chips with multiprocessor capabilities, leading to a new direction in the above-described categorization: thread-level parallelism (TLP). Even though Multiple Instructions Multiple Data (MIMD) architectures have been around for many years, they have been more-recently exploited through TLP, which may incorporate characteristics of both ILP-based and DLP-based systems, bringing processing potential to the next level. TLP is not inherently exploited by processors, meaning it does not work "out-of-the-box". It needs some deeper integration with processors by either operating systems, through scheduling multiple applications in parallel, or by the programmer, using multi-thread programming. Thus, TLP has not been used as a category in Table 2.1.

According to how parallelism is discovered, multi-issue processors can be divided into two categories: static and dynamic discovered. Discovery of parallelism in static multi-issue processors is done at compile time, usually with the unveiling of parallelism in the code, while dynamic multi-issue processors find parallelism during runtime, when the program is executed. Examples of static-discovered parallelism processors are VLIW or SIMD processors, and superscalars are examples of processors where parallelism is discovered at runtime.

The increase in the number of transistors packed within a single die have allowed processors to overlap this categorization. X86 processors, traditionally seen as superscalars, have included many instruction sets to handle DLP, such as, Streaming SIMD Extensions (SSE) and Advanced Vector Extensions (AVX). Patterson and Hennessy (2013) also divide graphics processing units (GPUs) according to how data-level parallelism is discovered. SIMD or Vector architectures discover DLP at compile time, while Tesla architectures require no vectorizing

compilers, since parallelism is discovered at runtime. NVIDIA Geforce 8800 is an example of a GPU that implements a Tesla architecture.

In the next subsections, we will be covering the basics characteristics of multi-issue processors aimed at the exploitation of ILP, VLIW and superscalar processors, highlighting the main aspects of each.

2.3.1 Very-long Instruction Word Processor

VLIW is a type of multi-issue processor that generally consists of a simple microarchitecture structure that relies on a software abstraction unit to exploit parallelism in applications. The name, very-long instruction word (VLIW), makes a reference to the fact that parallelism must be explicitly generated by the software, and a long instruction packs more than one operation at a time. The compiler has a major role in these processors, as it is the responsible for generating the parallelism necessary to exploit the multi-issue capability through a static scheduling algorithm.

Instructions that can executed in parallel are concatenated in bundles, which execute one at a time. Figure 2.9 shows examples of code for the Qualcomm Hexagon processor (CODRESCU, 2013) and the ρVEX (WONG, Stephan; AS, VAN; BROWN, 2008), two examples of VLIW processors, the former being used in the industry as a DSP processor and the latter being a reconfigurable VLIW processor built on top of an FPGA. In the figure, bundles

Figure 2.9 – Code examples for two VLIW processors

(a) Qualcomm Hexagon DSP

```
{
    p0 = cmp.rq(r0,#0)
    if (!p0.new) r2 = memw(r0)
    if (p0.new) jump:nt r31
}
{
    r1 = add(r1,add(r2,#2))
    memw(r0) = r1.new
    jumpr r31
}
```

(b) ρVEX Processor

```
;;
   c0    sth 0[$r0.13] = $r0.15
   c0    slct $r0.14 = $b0.3, $r0.14, $r0.20
   c0    and $r0.15 = $r0.17, 32768
   c0    and $r0.17 = $r0.16, 32768
   c0    shl $r0.16 = $r0.16, $r0.2
;;
   c0    xor $r0.18 = $r0.19, 4129
   c0    cmpeq $b0.0 = $r0.15, 0
;;
   c0    slct $r0.15 = $b0.0, $r0.19, $r0.18
;;
```

are delimited by curly brackets symbols, and double semi-collons are delimiters for the bundles in the assembly code in ρVEX.

Due to its static nature for discovering parallelism in code, VLIW architectures may have to pay a high cost in terms of binary code compatibility. The generated code must make use of the ISA and microarchitecture characteristics, such as the number of functional units included and their latencies. With improvements in silicon technology, wider and more resourceful implementations of VLIW processors were allowed, which led to concern of how to execute legacy code in newer processors. Thus, solution concerning maintaining backward compatibility were also explored (BRANDON; WONG, S, 2013; DEHNERT, 2003).

Examples of VLIW processors include the Itanium processor (EVANS; TRIMPER, 2003) by Intel, the ρVEX (WONG, Stephan; AS, VAN; BROWN, 2008), and Qualcomm Hexagon DSP processor (CODRESCU, 2013).

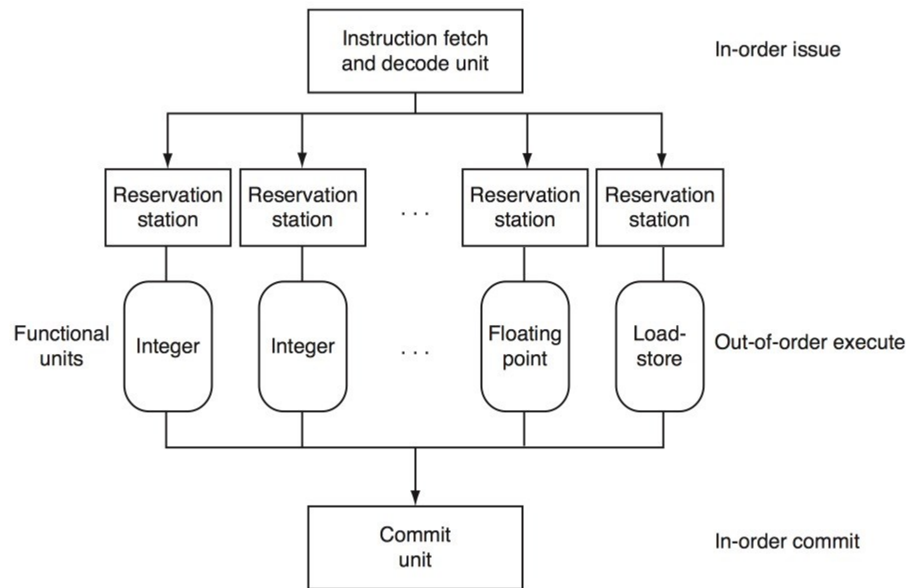
2.3.2 Superscalar Processor

A superscalar processor is a dynamic-scheduled processor that issues multiple operations at time. Opposing to VLIWs, compilers do not provide the explicit parallelism in the application, although they usually try moving dependences apart in order to increase the issue rate. Superscalars are the most common type of processors used in industry nowadays, and are highly more complex than VLIWs.

They are typically divided into two categories, in-order and out-of-order processors. In-order superscalars issue instruction in the order defined by the compiler, and processor decides how many instructions will be executed in each clock cycle according to the dependences among instructions. To achieve good performance in in-order processors, compilers must schedule instructions moving dependences apart, so that the instruction issue rate can be maximized. Even though compiler plays a key role in these processors, similar to VLIWs, code is guaranteed to execute correctly regardless of the processor model, and no recompilation is necessary.

Out-of-order (O3) superscalars are processors capable of executing instructions out of the compiler-defined order. The overall structure of this type of superscalars can be represented with three primary units: issue, execute and commit units. Figure 2.10 illustrates how these units interacts with one another.

Figure 2.10 – Three primary units of Out-Of-Order processors



Source: Patterson and Hennessy (2013)

The issue unit aims at fetching instructions, decoding and sending them to the appropriate units for execution. The operation decoded at the issue unit is handled at one or more of the execution units. Before reaching the specific functional units for execution, decoded operands and operations are passed to buffers, called reservation stations, inside the execution unit, which hold them until they are ready to execute. The calculated value is ready to go to the commit unit, and is also sent to any reservation stations that wait for its value, allowing them to execute. Commit unit holds calculated values in a *reorder buffer* until it is safe to send them back to the registers or memory. The behavior of the execution units defines what type of superscalar a processor is. In-order superscalars uses in-order executions, while out-of-order superscalars use execution units that may process instructions out of their order, which increases the instruction issue rate on the processor. Issue and commit units are always kept in-order for both types.

Moreover, this text only provides a brief overview of superscalars, and it must not be considered a complete description of their functionally. For a deeper understanding and details, reader should refer to (HENNESSY; PATTERSON, 2011; PATTERSON; HENNESSY, 2013).

3 RELATED WORK

Many related works involve the use of scratchpads as energy-efficient alternative for caches. Even though most of them were proposed more than ten years ago, some recent works have emerged, especially because energy is a major concern in computer systems. Caches have been around for many years; however, they may be found inappropriate to keep energy to an acceptable boundary.

Modern FPGAs may include tens of megabytes of memory, that is, a considerably large amount of memory is available in the device. This significant capacity is spread across numerous small block RAMs (BRAMs), each with its own memory ports and address spaces, providing an enormous bandwidth to access these internal memories. As such, FPGAs' inherent parallelism and high bandwidth are particularly beneficial when data is fetched from multiple uncorrelated BRAMs, i.e., there is no need for a unique address space. Nevertheless, from a software programmer's point of view, memory is seen as a sizable storage with a unique address space. The traditional FPGA memory model is therefore not directly suited to meet software's requirements. FPGAs' flexibility allows the combination of multiple BRAMs with the purpose of providing a unified address space, with the downside of significantly increasing area and compromising memory bandwidth.

In this section, we will cover major works proposed concerning the use of scratchpads, covering major works in the academia and industry usage. We will also present previous works that focus on the exploration and design of multi-ported memories for ASICs and FPGAs, since SoMMA can be employed in ASIC- and FPGA-based processors. Due to the high quantity of BRAMs in FPGAs, however, SoMMA is particularly interesting for the use in FPGA-based processors, as no additional price needs to be paid on memory storage.

3.1 Scratchpads

3.1.1 Academic use

The employment of scratchpads in processors can be dated back to the 1970s with a 64-byte scratchpad in the Fairchild F8 microprocessor (FAIRCHILD, 1975), allowing programmers to store any of the machine code instructions into it for fast access. In the academia, the concept started being explored in the later 90s. Panda, Dutt and Nicolau (1997)

was one of the first works to propose the use of scratchpads to partition applications' data between memories aiming at minimizing execution time. Subsequent researches have proposed the use of scratchpads in many ways: data or code storage, for improving execution time and/or energy consumption, or to work with/be a replacement for caches. Next sections will give an overview of some of the major works in scratchpads employed for code-only, data-only, and hybrid code and data solutions.

3.1.1.1 Scratchpad for Instructions

Code placement in scratchpads were initially developed to minimize the number of I-caches misses. Traces were generated to identify parts of code that execute the most for scratchpad mapping (PETTIS; HANSEN, 1990; TOMIYAMA; YASUURA, 1996). Verma, Wehmeyer and Marwedel (2004) was the first approach to propose the scratchpad working in synchrony with the I-cache. A generic algorithm called cache-aware Scratchpad Allocation (CASA) algorithm for storing instructions in scratchpads is presented. The algorithm looks for hotspots in the application in order to minimize the number of cache misses, and consequently, reduce energy consumption. The motivation to work only with code is due to instruction memory be accessed on every instruction fetch and the size of programs for mobile programs be smaller than their data.

Using a different approach, the technique in Angiolini *et al.* (2004) requires no access to either the compiler or application source code. The approach automatically moves code sections to the scratchpad area through a post-compilation process, taking the burden from the compiler. Authors state that overlay approaches are more scalable when considering huge application, but also more fragile as excessive transfers of objects might decrease performance. While segments of data are usually larger and sparser, the justification for overlay approaches have a better claim for data replacement.

Whitham and Audsley (2008) extend the work on instruction scratchpads by applying a post compilation analysis and generating the trace of the application. The focus of their work is to minimize the worst-case execution time (WCET) of the application, a very important factor for real-time systems (RTS). The instruction scratchpad is a trace scratchpad that respects constraints in WCET, depending on the information obtained from the trace.

When analyzing these works, we notice how solutions proposed to tackle instruction reallocation to scratchpads as a resolution to minimizing energy consumption and/or increasing

performance, though with different methods. Subsequent works on scratchpad placement for instructions are a variation of these techniques, which sets as the standard to how utilize scratchpads for code.

3.1.1.2 Scratchpads for Data

Another form of utilization of scratchpads come for data placement. The work in (AVISSAR; BARUA; STEWART, 2002) proposed a strategy to automatically partition global and stack data among different heterogenous memory units in embedded systems that lack caching hardware. The allocation strategy uses a static fashion where data remain fixed and unchanged throughout program execution.

A more recent work proposed by (WANG; GU; SHAO, 2015) shows an algorithm to minimize WCET and energy consumption in a system allowing data placement into scratchpad. The solution, however, simplifies the testing infrastructure with a simple in-order processor with no data cache and a perfect-hit instruction cache, which limits the usability of the technique in more complex processors.

A generic solution for combining scratchpads and caches in CPU-GPU systems is proposed by (KOMURAVELLI *et al.*, 2015). Stash is a memory organization that leverages scratchpads' compact storage and lack of tag arrays, while also being globally-addressable and visible like a cache. The solution is applied to heterogeneous systems composed of CPUs and GPUs where the GPUs access both coherent caches and private scratchpads. Stash provides a global visibility for scratchpads that are located inside each GPU compute unit (CU), that way, GPU CUs can share data.

3.1.1.3 Scratchpad Solutions for Code and Data

More flexible techniques were also proposed to utilize scratchpads for both code and data. Verma and Mardwedel (2006) present an overlay-based method for dynamically copy both variables and code segments onto a scratchpad at runtime. The proposed solution is akin to the Global Register Allocation problem, in which the compiler attempts to make virtual register assignments to physical registers to a minimum, in order to prevent spills to memory.

Udayakumaran, Domingues and Barua (2006) show one of the most complete schemes to dynamically allocate data and program code to scratchpads. Their contribution is particularly

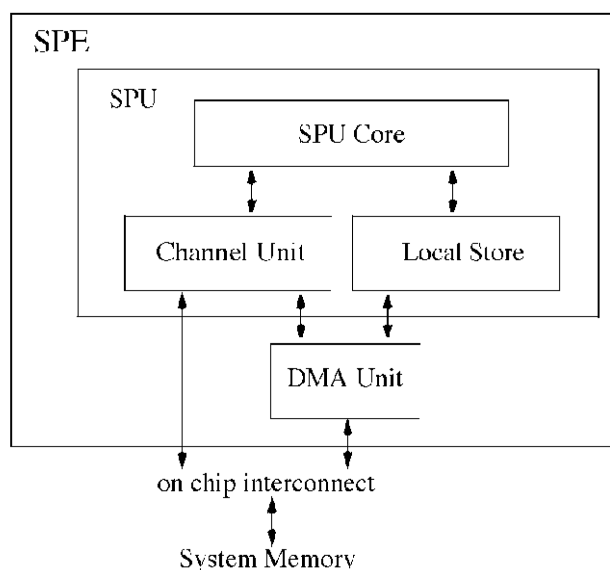
interesting because the solution not only handles global variables, but stack data as well, including both segments of data and code at a unique scheme. It targets the placement of code and global and stack data to scratchpads with the purpose of reducing energy consumption. The proposed solution is a dynamic allocation methodology for both data and code, in which the content of scratchpad can change during the program execution.

3.1.2 Usability in the industry

Scratchpads' importance has not emerged exclusively at the academia; industry has also played an important role to advance their use.

The IBM cell processor (FLACHS *et al.*, 2006), that powers Sony PlayStation 3 console, uses scratchpad memories inside its processing elements. The core features a multi-threaded POWER processing element and eight synergistic processing elements (SPE). The architecture aims at accelerating media and streaming workloads by reducing memory latency inside the SPE elements. As illustrated in Figure 3.1, each SPE is composed of processing core (SPU Core), a DMA unit that transfers data to and from the memory system, and a local storage unit, or scratchpad, that can be used to store local data or instructions. In order to execute code in the SPEs, the workloads are written in C/C++ using intrinsics for the SIMD data types and DMA

Figure 3.1 – Synergistic processing element on the IBM cell processor. Scratchpads are referred as "Local Store"



Source: Flachs *et al.* (2006)

transfers, which becomes a programmability barrier for programmer who wants to implement efficient code for this processor. Che and Chatha (2011), on the other hand, use the same IBM Cell processor to accelerate stream applications by placing their code into the scratchpad with an overlay-based method.

Moreover, the work in (MOAZENI; BUI; SARRAFZADEH, 2009) presents a memory optimization scheme to minimize the usage of shared on-chip memory, also called scratchpad memory, in the NVIDIA G80 (KIRK, 2007) graphics processing unit (GPU). The paper states that scratchpad memories help alleviating the pressure on global memory bandwidth, and therefore, maximize performance on data-intensive applications that have high usage of shared memory. The optimization is particularly designed to work in an architecture which scratchpads are shared among many threads.

3.2 Multi-ported systems

The limiting bandwidth in the memory system might also be overcome with the addition of memory ports. Because multi-ported systems can be quite costly, previous works propose techniques to design multi-ported systems more efficiently. They have targeted either a solution for application-specific integrated circuits (ASIC) or field-programmable gate array (FPGA) devices, however, we have found no work with a common solution for both types of devices.

Malazgirt *et al.* (2014) present an application-specific methodology that analyzes a sequential code, extracts parallelism and determines the number of read and writes ports necessary for such application.

LaForest and Steffan (2010) introduce a new approach for creating multi-ported memories in FPGAs that efficiently combines BRAMs while achieving significant performance improvements over other methods. It replicates memories according to the number of read/write ports. An 8-read/4-write memory, for example, can be implemented using eight 2-read/1-write memories. It also uses live value tables (LVT) to identify where the most recent copy of the data is located. Although this approach is better than others, it still requires at least doubling the area to build multi-ported memories with more than two ports.

The work in (BAJWA; CHEN, X., 2007) is applicable to application-specific integrated circuit (ASIC) devices and proposes an area and energy-efficient multi-port cache memory. Although able to minimize the costs of multiple ports, it is not able to completely mitigate them.

Recent researches have provided various contributions on the use of BRAMs. Adler *et al.* (2011) propose Logic-based Environment for Application Programming (LEAP), an

automatic memory management tool for reconfigurable logic. The main idea is to use LEAP scratchpads to provide an easy way of managing memory on FPGAs. Thus, developers can primarily focus on developing their core algorithms and less on managing memory.

Winterstein *et al.*, (2015) and Yang *et al.* (2015) extend LEAP scratchpads to automate the construction of application-specific memory hierarchies. Therefore, every application has an optimized memory hierarchy that best fits the application's characteristics.

Weisz and Hoe (2015) provide a convenient approach for designing efficient complex data structures in memory without penalizing access performance. It uses an application-level interface that models data structures, such as multi-dimensional arrays, linked lists, and simple data patterns.

Meng *et al.* (2015) propose a new memory partitioning algorithm for parallel data access. The strategy focuses on finding the access pattern for applications and choosing an optimal number of banks and offsets for each pattern data. The approach aims at maximizing bandwidth while providing simultaneous accesses to patterns for higher parallelism. This work is complementary to our own, as the authors focus on placement strategies for different memory access patterns in FPGA devices, while we focus on an architecture to provide improved performance for a given data placement and on how to automate code generation in a production compiler in either FPGAs or ASICs.

In (BETKAOUI; THOMAS; LUK, 2010) the authors present a comparison of performance and energy consumption between FPGAs and graphics processing units (GPUs) and illustrate how memory bandwidth can have an impact on those results. Although these works help identifying weak spots over memory designs in FPGAs, they lack to propose new ways to improve performance and reduce energy consumption due to distinction among the benchmarks.

3.3 Our approach x Other methods

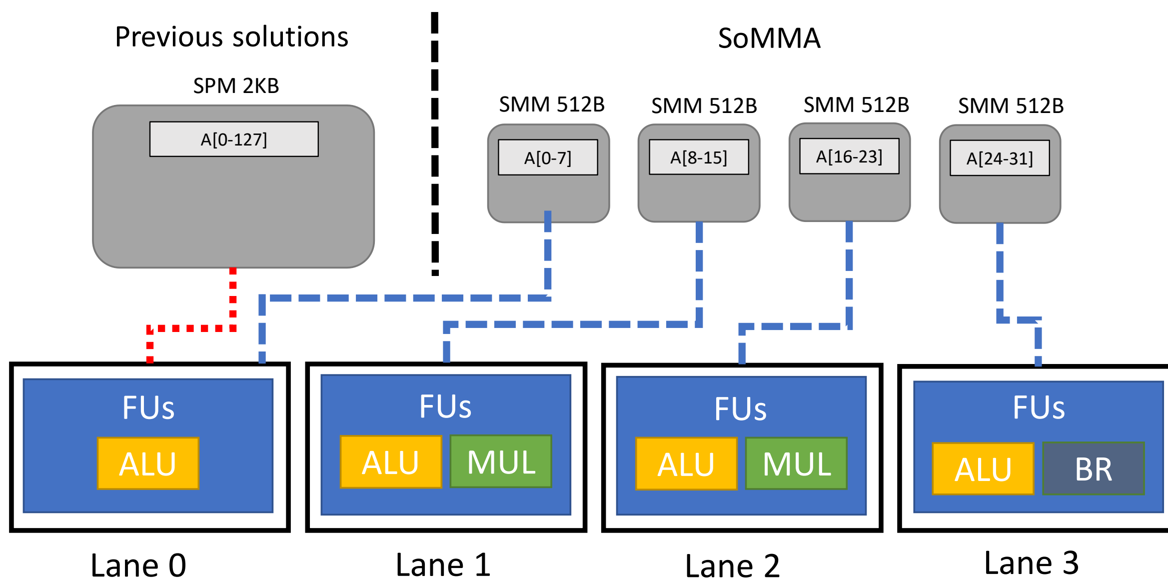
The contributions of this work are as follows:

1) Reducing execution time and energy by avoiding unnecessary cache or Dynamic Random Access Memory (DRAM) accesses, since they are slower and more energy-consuming.
2) Allowing the usage of multiple single-ported software-managed memories in parallel for data placement and, consequently, providing a higher memory throughput than other methods.

While previous works also target (1), (2) is only provided by SoMMA in multi-issue processors. If one considers a multi-issue processor, like a VLIW, and the solutions presented in Figure 3.2, SoMMA leverages the usage of multiple memories by simulating a multi-ported system with multiple single-ported memories. In the figure, previous solutions are represented by the red-dotted connection between lane 0 and scratchpad (SPM), while the proposed solution is represented by the blue-dashed connection between software-managed memories (SMMs) and lanes. Our solution uses memories in parallel in order to leverage the multi-issue characteristic of the processor. We can load and store values in multiple memories at once. The figure also depicts how a vector structure could be potentially stored to benefit the multiple memories at once.

Moreover, any attempt of numerically comparing our solution to others would result in unfair comparisons, since distinct sets of tools were used. Compiler toolset, instruction set architecture (ISA), transistor technology and benchmarks are dissimilar. Our experiments are not intended to show how our solution is more beneficial in terms of numbers, but to instigate the reader observe how our approach contemplates previous works and adds the possibility of using multiple scratchpads in parallel, bringing a new enhancement into picture.

Figure 3.2 – A 4-issue VLIW processor example. Previous solutions are represented by the red-dotted connection between lane 0 and scratchpad (SPM), while the proposed solution is represented by the blue-dashed connection between software-managed memories (SMMs) and lanes. Our solution uses memories in parallel in order to leverage the multi-issue characteristic of the processor. We can load and store values in multiple memories at once.



Source : author

4 HARWARE AND SOFTWARE SUPPORT

Memory bandwidth is the cornerstone of high-performance processors. The memory system should provide enough data such that functional units (FUs) are frequently occupied, as idle FUs also consume static energy while producing no useful values. The simplest solution to improve memory bandwidth comes from adding memory ports to the memory system. However, as presented in Figure 1.2, this solution has a deep impact in area, that cannot usually be paid in embedded systems. An alternative method for bandwidth increase relies on adding smaller and faster memory units (or scratchpads) that can benefit from data reuse. Scratchpads provide better performance and energy efficiency than caches because no tag array and replacement logic are necessary. The control of these memories is done at software level, so they are typically known as software-managed memories (SMMs) or software-controlled memories. SMMs are particularly beneficial when reuse can be guaranteed, i.e., one needs to assure that temporal and spatial locality are present in the program.

Our main goal is to mitigate memory complexity in multi-issue processors by creating a memory architecture that explores the use of multiple memories in parallel. That way, single-ported memories combine with the data cache to provide the system with a higher bandwidth without the overhead of adding new ports. We use a solution that incorporates changes in both hardware and software and can overcome the limit of single-port systems, providing a higher throughput.

The rest of this section will focus on the hardware and software requirements to integrate the memory architecture in a system. We will also provide an example that illustrates how our technique is applied to an application and we will end this section with the main component of our approach: the automated code generation process that makes the necessary changes to the applications.

4.1 Hardware

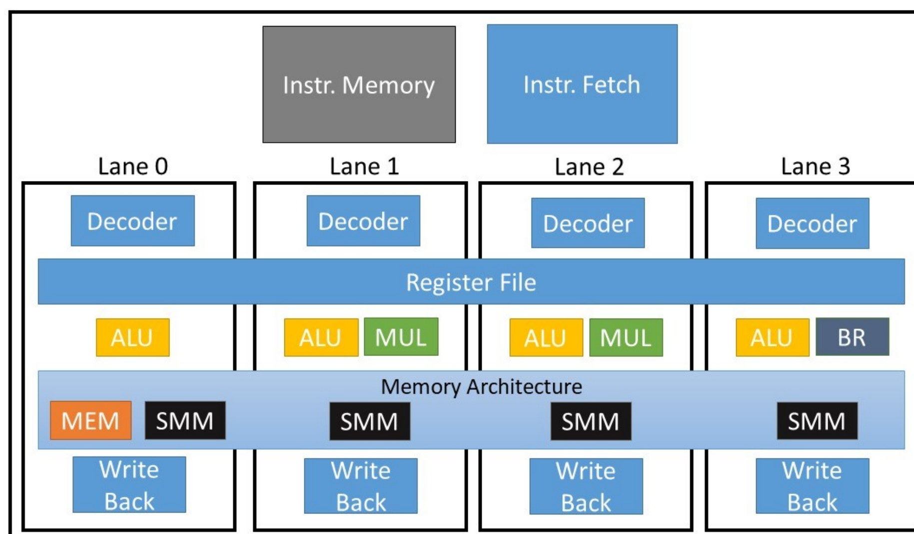
We propose a method that requires minimum hardware changes to support our SMMs. By limiting most transformations to software, we remove the costs of complex hardware structures to manage SMMs. The main difference between SMMs and caches relies on who controls them. The former is controlled by software and requires no extra hardware for tag storage and checking while the latter is hardware-controlled and the programmer has little control over where data is placed in memory.

Figure 4.1 shows an example of how our SMMs are placed within a hypothetical 4-issue VLIW processor. A memory architecture is created with the placement of a software- managed memory within each parallel lane. With the exception of Lane0, which can also access the regular data memory hierarchy through the L1 data cache (MEM block in Figure 4.1), each lane can only access its own memory's content. Because lanes can only operate in one memory, there is no need to specify which memory SMM instructions should access. Each lane only sees its own memory, and the address spaces are independent of one another. Thereby, we avoid the costs of having multi-ported memories, while still providing parallelism for the processor. SMM instructions operate in the same way as loads and stores: one register as the base register, an immediate offset that will be added to the base register, and a second register that will hold the value that must be stored to or was loaded from memory.

4.2 Software

Previous to our proposed code optimization that handles scratchpads, a series of code analysis and transformations help in the preparation to our solution. These optimizations affect the quality of the IR code produced, impacting the code generation process we have proposed. Subsection 4.2.4.3 will discussion some of these optimization in more detail.

Figure 4.1 – Location of the memory architecture in a 4-issue VLIW processor. Each lane has access to an internal memory



Source: author

4.2.1 SMM-specific instructions

Software, either compiler or assembly writing, should be capable of generating code that can differentiate main memory from SMMs. For that matter, we propose the addition of a set of load and store instructions specifically for controlling those intralane memories. Each lane only accesses its own memory and their address spaces are independent from one another. Thereby we avoid the costs of making multi-ported memories, while still providing enhanced bandwidth. Like most ordinary memory instructions, these special instructions have three operands: one register as the base register, an immediate offset that will be added to the base register, and a second register that will hold the value that must be stored to or loaded from memory.

Figure 4.2 presents an example of the use of SMM-specific instructions on the hypothetical 4-issue VLIW processor of Figure 4.1. New instructions were added to the ISA in order to facilitate the use of SMMs. Memory instructions with the suffix ‘s’ represent software-managed-memory instructions. There is no need to distinguish among which lane the instructions are, since each lane can only access one memory.

Regular single-ported memory systems would require at least 4 cycles to load four values into registers (Rows 2 to 5). By using SMM instructions, we still need to load the values through the regular cache and plus to store them into SMMs (Row 6). From that moment on, the processor can load all the values in parallel. Thus, if sufficient temporal locality is found in the access to these data, the parallel re-fetches will pay off the costs of filling the SMMs and as consequence, accelerate the execution of the application.

Figure 4.2 – Illustrative example for controlling SMMs

Lane 0	Lane 1	Lane 2	Lane 3
ldw r7 = 0[r5]			
ldw r8 = 4[r5]			
ldw r9 = 8[r5]			
ldw r10 = 12[r5]			
stws 0[r5] = r7	stws 0[r5] = r8	stws 0[r5] = r9	stws 0[r5] = r10
Following Instructions			
ldws r7 = 0[r5]	ldws r8 = 0[r5]	ldws r9 = 0[r5]	ldws r10 = 0[r5]

Source: author

Figure 4.2 shows that it is the software's entire responsibility to fill those memories with data and to know the position of each stored value for further references. Through the use of offsets and base registers, we are able to write code that dynamically changes the content of the SMMs during the execution of the program, i.e., our technique uses an overlay-based approach, improving performance even more.

4.2.2 Preamble Code

Most of the applications require a warm-up code implementation, also called preamble, to begin the insertion of new values on the uninitialized memories. This piece of code is responsible for fetching values that should reside in the SMMs from the regular memory hierarchy, i.e., through the L1 data cache. The unveiling of ILP relies on using multiple software-managed memories in parallel. Once data is loaded into the SMMs, it can be accessed with high bandwidth due to the multiple independent ports. So, the compiler's goal is maximizing the usage of these memories while still reasoning about parallel operations. Hardware support is the first step to apply our technique to a processor. The second is the code generation process, where instructions that manipulate those memories are generated. Users can decide which variables are placed within the memories through simple code annotations. The code generation process detailed herein plays the main role in our approach.

4.2.3 Example Application

In order to explain how our technique works, an example application is provided. Figure 4.3 shows a snippet of C-code for a matrix multiplication 7x7. Users can decide which variable

Figure 4.3 – Snippet of 7x7 matrix multiplication

```
for(i = 0; i < 7; i++){
    for(j = 0; j < 7; j++){
        var = 0;
        for(k = 0; k < 7; k++){
           var += smm_a[i][k] * smm_b[k][j];
        }
    res[i][j] = var;
    }
 }
```

Source: author

is placed within the memories through a simple code annotation (variables beginning with *smm_*). We have chosen to place variables *a* and *b* (denoted as *smm_a* and *smm_b*) into the SMMs, that way, we can reason about the two different placements strategies for storing array variables in multiple memories. Placing variable *a* in the SMMs, for instance, does not improve performance, as the algorithm can keep all values from each row in registers and there is no need to re-fetch them later. The example is merely illustrative in that sense, as we can present both placement strategies adopted.

We have compiled the application using the –O3 flags, which enables many target-independent optimizations to run. Our algorithm works right before register allocation, when machine instructions are within the target architecture and virtual registers are used. Figure 4.4 shows a reduced snippet of the assembly code for the application. Keep in mind the algorithm is not complete, due to space limitations. However, the idea can be extended for completion. The presented code loads values for variables *smm_a* (in BB#1) and *smm_b* (in BB#2) and multiplies values to compute a result per iteration (in #BB2).

Figure 4.4 – Snippet of assembly code

```
1   BB#0:
2         %vreg20 = MOVi <ga:@smm_a>;
3      %vreg23 = MOVi <ga:@smm_b>;
4   BB#1:
5         %vreg21 = ADDr %vreg19, %vreg18;
6         %vreg22 = ADDr %vreg20, %vreg21;
7         %vreg7 = LDW %vreg22, 20;
8         %vreg6 = LDW %vreg22, 16;
9         %vreg8 = LDW %vreg22, 24;
10        %vreg5 = LDW %vreg22, 12;
11        %vreg4 = LDW %vreg22, 8;
12        %vreg3 = LDW %vreg22, 4;
13        %vreg2 = LDW %vreg22, 0;
14  BB#2:
15        %vreg10 = PHI %vreg17, <BB#1>, %vreg11, <BB#2>;
16        %vreg24 = ADDr %vreg23, %vreg10;
17        %vreg25 = LDW %vreg24, 140;
18        %vreg26 = LDW %vreg24, 168;
19        %vreg27 = MPYLUr %vreg26, %vreg7;
20        %vreg28 = MPYLUr %vreg25, %vreg6;
21        %vreg29 = MPYHSr %vreg26, %vreg7;
22        %vreg30 = MPYHSr %vreg25, %vreg6;
23        %vreg31 = LDW %vreg24, 112;
24        %vreg32 = MPYHSr %vreg31, %vreg5;
25        %vreg33 = MPYLUr %vreg31, %vreg5;
26        %vreg34 = ADDr %vreg28, %vreg30;
27        %vreg59 = CMPNEBRegi %vreg11, 0;
          .
          .
          .
    50    STW %vreg58, %vreg9, 0;
```

Source: author

Although programs are represented in a linear form at their last stage (assembly code), compilers can realize them as *data-dependence graphs* (DDG), whereas nodes represent instructions and edges represent the dependence between them (COOPER; TORCZON, 2011). An edge in the DDG is a relationship between a *definition* and its *use*. A *definition* represents an assignment of a variable, while a *use* represents its use as an operand. For instance, the definition of register *%vreg23* (line 3) is within basic block BB#0, while its use is in BB#2 (line 16).

Our technique initiates by looking for *mov* instructions that use an SMM variable's address as operand. Such *mov* instructions are named as *def* instructions, because they define a starting point for an SMM variable. Two *def* instructions are observed within the code (lines 2 and 3), one for variable *smm_a* and another for *smm_b*. Registers used in *def* instructions are kept in a table of definitions and are used for searching memory instructions that are related to SMM variables. The algorithm will look for *uses* of such registers until it reaches a memory instruction. We say an instruction propagates an SMM variable if it reaches that variable through the *definition-use* relationship. For example, register *%vreg23* defines a starting point for an SMM variable (line 3). Its use is at line 16, which then defines another value, *%vreg24*, that is used at lines 17, 18 and 23, reaching three memory instructions for variable *smm_b*. The same idea is applied to variable *smm_a*. Memory instructions from lines 7 to 13 come from it. Through *definitions* and *uses*, the algorithm finds all memory instructions related to SMM variables, in order to replace them later.

Although it helps discovering memory instructions for SMM variables, the process above does not improve parallelism per se. The algorithm must also keep track of all offsets used for each variable. The more offsets within a basic block, the more memories can be used for the variable and the more parallelism can be exploited. The number of memories should be proportional to the number of offsets encountered per basic block, as same instructions should be used in different iterations of a loop. For this example, the inner loop was unrolled 7 times by LLVM optimizations. Thus, seven offsets are detected for variable *smm_a* in BB#1, and seven for *smm_b* in BB#2 (only three are displayed in Figure 4.4). In a processor with eight software-managed memories, the compiler would infer the use of seven memories for each variable.

Moreover, we can also notice how the offset ranges differ between those variables. This happens because in one, *smm_a*, data is fetched in-row (distance between offsets is 4, because we are dealing with 4-byte integers), while in the other, *smm_b*, data is fetched in-column (distance is 28, 7 values of 4-byte integer per row). Two different placements strategies should

Figure 4.5 – Offset address locations for a 7x7 matrix multiplication. The example shows how matrix smm_a (shortened to a) and smm_b (shortened to b) would be placed in the SMMs. First scheme, on the left, shows an allocation through column order, while the second, on the right, displays an allocation through row order

SMM 0	**SMM 1**	**SMM 2**	**SMM 3**	**SMM 4**	**SMM 5**	**SMM 6**	**SMM 7**
a[0,0]	a[0,1]	a[0,2]	a[0,3]	a[0,4]	a[0,5]	a[0,6]	
a[1,0]	a[1,1]	a[1,2]	a[1,3]	a[1,4]	a[1,5]	a[1,6]	
a[2,0]	a[2,1]	a[2,2]	a[2,3]	a[2,4]	a[2,5]	a[2,6]	
a[3,0]	a[3,1]	a[3,2]	a[3,3]	a[3,4]	a[3,5]	a[3,6]	
.	.	.	.	.	.	.	

SMM 0	**SMM 1**	**SMM 2**	**SMM 3**	**SMM 4**	**SMM 5**	**SMM 6**	**SMM 7**
b[0,0]	b[1,0]	b[2,0]	b[3,0]	b[4,0]	b[5,0]	b[6,0]	
b[0,1]	b[1,1]	b[2,1]	b[3,1]	b[4,1]	b[5,1]	b[6,1]	
b[0,2]	b[1,2]	b[2,2]	b[3,2]	b[4,2]	b[5,2]	b[6,2]	
b[0,3]	b[1,3]	b[2,3]	b[3,3]	b[4,3]	b[5,3]	b[6,3]	
.	.	.	.	.	.	.	

Source: author

be adopted for maximizing parallelism, depending on the pattern of access. Figure 4.5 shows the placement strategy used for the variables according to the observation of offsets in the basic blocks. For *smm_a*, values are spread through the SMMs in a contiguous way (one row at a time), while for *smm_b*, values are spread in a non-contiguous way (one column at a time), but within a known interval.

Finally, after gathering all information needed, the compiler generates code for the preambles and replaces memory instructions for SMM instructions. Instruction scheduling and bundling take place afterwards. During these steps, the SMM units on each lane give greater parallelism opportunities for the heuristics. In this example, load instructions in BB#2 and BB#3 will execute in different units, thus, parallelizing memory access.

4.2.4 Code Generation Process

Section 4.2.3 used a matrix application to exemplify how our approach analyzes code and performs modifications. This section presents an explanation of each step of the process, how they interact and their main characteristics. Our algorithm is divided into two main actions we call *variable discovery* and *variable transformation*. *Variable discovery* collects information about the variables for the software-managed memories while *variable transformation* makes the necessary code changes to use SMMs in case a variable is discovered in the previous process. We will discuss both actions in the next subsections, showing specific characteristics for each action and how they correlate.

4.2.4.1 Variable discovery

This is the process responsible for finding out which variables can be placed in the SMMs. The user can specify which variables may be stored in the memories through code annotations. Our algorithm currently works only with global variables and plans on extending the work to handle stack variables and heap-allocated variables are part of the future works.

Figure 4.6 shows a high-level simplified version of our algorithm for *variable discovery*. The algorithm runs in all functions of the program, trying to find variables that are allowed to be on the SMMs. It starts by traversing the basic blocks in a breadth-first search (BFS) order (Line 1). The algorithm basically searches for *def* instructions and memory instructions according to the basic block order. It aims at finding instructions that define variables, and instructions that propagate the use of these variables through basic blocks (BBs). The core of the algorithm begins at line 4, where we iterate over all instructions in the basic block. Line 6 and 7 show a fast escape for call and branch instructions, since they do not need to be checked. The algorithm may execute four functions at lines 9, 12, 15 and 16 with the following purposes:

- *isSMMVariable* (Line 9): this function checks for *def* instructions within basic blocks, looking for the code annotation given by the user. If the annotation is found, the instruction is inserted in the list of def instructions for that variable (Line 10), keeping track of its defined register.
- PropagatesSMMVariable (Line 12): This function verifies if the current instruction propagates any SMM variable through the *definition-use* relationship, keeping track of definition registers. At line 16 of Figure 4.4, e.g., %vreg23 is propagated and a new definition is found (%vreg24).
- InstrIsStoreOrLoad (Line 15): if the current instruction is a memory instruction, the algorithm should evaluate the offset and insert the instructions into a list of memory instructions for further replacement.
- *EvaluateOffset* (Line 16): we aim at using software-managed memories primarily on vector and matrix variables inside loop statements, since they offer more possibility of parallelism. When a loop is unrolled, multiple iterations overlap and different offsets can be found in a single larger iteration. Offsets are important because they tell us exactly where the data is located in memory, and therefore, they will help us in the process of placing values properly on our memories. This function evaluates the offset for the current instruction. The

compiler builds a table for offset of variables for deciding on the number of SMMs necessary for each variable.

We construct a list of all variables and its memory-related instructions (Line 13), preparing all necessary data structures for the *variable transformation* process. This process also keeps track of the location for each memory instruction added to the variable list (to which basic block it belongs).

4.2.4.2 Variable Transformation

After collecting the required information about SMM variables, we still need to perform code transformations in order to use them. We should analyze the information collected previously, and make decisions about whether code changes are necessary. *Variable Transformation* is where all the code changes take place. These changes go from code modification in memory instructions to new code insertion.

Some variables may require previous placement within the software-managed memories before their use. Others may not require that task since their values are firstly calculated and then inserted in the memories. The *preamble* code is only inserted when the first memory

Figure 4.6 – High-level algorithm for Variable Discovery

```
1 for each BB in BFS Order do
2 begin
3
4             for each Instr in BB do
5             begin
6                    if instr is Call or Branch
7                           continue;
8
9                    if isSMMVariableDef(Instr) then
10                          add Instr to ListDefVar
11                   else
12                          if PropagatesSMMVariable(Instr) then
13                                 track register on Var
14
15                            if InstrIsStoreOrLoad(Instr) then
16                                        EvaluateOffset(Instr)
17                                        add Instr on Var
18                            end if
16                          end if
17                   end if
18            end for
19     end for
```

Source: author

instruction spotted in the program that is related to the variable is a load. Otherwise, no *preamble* code is necessary, as we are already filling up our memories as the program executes. Moreover, this process substitutes regular memory instructions, that is, those that use the standard memory hierarchy, for software-managed memory references. We also have to guarantee that a single instruction can be used for iterating over multiple reference locations, independently of the executed iteration.

4.2.4.2.1 Offset Calculation

Preamble insertion firstly needs to calculate the offset for each variable location. Single-memory variables are easily calculated. Multi-memory variables, on the other hand, require extra calculations, since different addresses of one variable can live in different SMMs. Our technique uniformly assigns locations to the values depending on the number of SMMs calculated previously. Two different relocation strategies may occur:

- 1-consecutive allocation: the next data value fetched within the basic block will have a one-unit distance from the previous value, and thus, we fetch data in a contiguous manner. Variable *smm_a* in Figure 4.4 uses this approach.
- N-consecutive allocation: this scheme occurs when two data addresses within the basic block are far from one another, regarding their memory locations. This scheme deals with data that are non-contiguous in memory. *N* corresponds to the distance between two data offsets within a same basic block. Variable *smm_b* uses this approach.

Figure 4.5 illustrates both schemes in a practical example. Each column represents a software-managed memory, and each row represents a location within it. We use the same 7x7 matrix multiplication algorithm to clarify the difference between these two schemes. Our algorithm infers the use of 7 SMMs for each variable. The allocation scheme on the left can be employed for variable *smm_a* which is accessed with successive addresses, i.e., 1-consecutive allocation is employed. Variable *smm_b* fetches data from different rows, and therefore, the N-consecutive allocation is applied (where N is equal to 7 for this case). The allocation scheme depends on the distance between offsets within a basic block.

4.2.4.2.2 Preamble Insertion and number of software-managed memories

For those variables that require *preamble*, we insert new basic blocks in the program, before variables are accessed. The compiler inserts instructions to read from main memory and place values of the variable inside the software-managed memories, using the proper offset according to the information collected in the previous process. Preamble insertion adds some execution time for each variable placed in the SMMs, for that reason, it is important to choose variables that present a great data reuse.

We analyze the variable and quantify how many software-managed memories are necessary depending on the number of offsets observed in the basic blocks. The number of offsets within a single basic block must be multiple of the number of SMMs available in the processor. For instance, a 7x7 matrix multiplication benchmark, shown in Figure 4.3, unrolls the inner loop by a factor of 7 in the compiler framework. After analyzing all basic blocks, we conclude that at most seven different offsets can be spotted in a single basic block. Therefore, our algorithm will infer 7 SMMs for *smm_a* and *smm_b* (suppose that we have a total of 8), enabling our processor to fetch multiple data in parallel.

Moreover, when we cannot guarantee that data is accessed uniformly, our compiler detects that values cannot be spread out across multiple memories. A simple example would be calculating a sum of values in a vector that randomly selects a position. There is no way to assure where the value is located when using more than one SMM. In this case, our algorithm selects only one memory. We will call single-memory variables those that only use one SMM, and multi-memory variables the ones that use more than one SMM. The compiler is also aware of the processor's characteristics, such as number of software-managed memories, arithmetic and logic units (ALUs) and number of ports to the main memory hierarchy.

4.2.4.2.3 Instruction transformation

After the assignment of variables to locations, the final step is code transformation. Here the compiler checks for all definitions and memory references found in the *variable discovery* process and assigns new offsets and addresses for them. Our algorithm computes lanes and offsets according to the allocation schemes described in the previous subsection and base addresses are known when preamble insertion is performed.

4.2.4.3 Additional optimizations

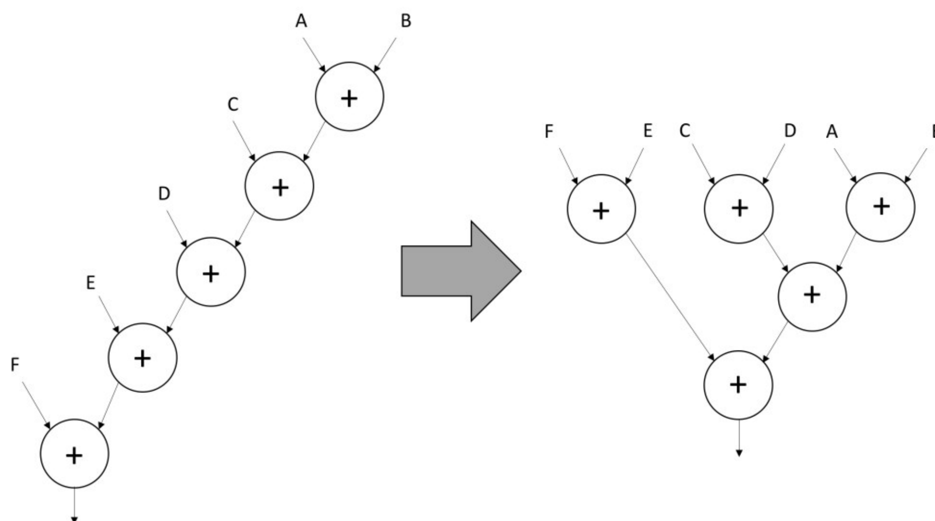
Two traditional compiler optimizations have important synergy with our architecture. Namely, loop unrolling and tree height reduction provide important gains when coupled with it.

Loop unrolling transformation has proven to be very efficient to boosting performance on our software-managed memories. It attempts to optimize the program through the execution of multiple loop iterations at the same time. Loop unrolling needs to guarantee that addresses do not alias, that way, iterations may overlap and performance may increase through unrolling. This transformation helps improving performance by allowing multiple offsets at the same basic block. And, therefore, more software-managed memories can be used to provide parallel accesses more easily.

Algorithms, such as matrix multiplication, x264, and sum of absolute differences (SAD), are characterized by having a long addition tree after the computation of each iteration. When the loop is unrolled, the result of each operation is summed with the previous computed values, in an inefficient way. This long chain of additions has little parallelism, limiting the performance gains of SoMMA. Therefore, we have also implemented a tree height reduction algorithm (KUCK, 1978), that efficiently computes and reduces the height of the addition tree for most of the algorithms.

Figure 4.7 shows the code transformation performed by our compiler. This algorithm reduces the tree height from n to $log_2 n$, and therefore, a significant performance increase can

Figure 4.7 – Tree height reduction algorithm



Source: author

be observed as more parallelism is exposed. Bear in mind that both techniques were also applied to the baseline processor, as they also benefit processor with standard memory architectures.

5 EXPERIMENTAL SETUP AND RESULTS

Our solution for maximizing ILP makes use of multi-issue processors, as the architecture demands multiple units operating at the same time. In this chapter, we present the set of tools, methodology and results of our experiments, focusing on the benefits of using SoMMA in a VLIW processor, and analyzing its impact in terms of energy, execution time and energy-delay product (EDP).

5.1 Tools

5.1.1 VLIW Example (VEX) architecture

VEX is a 32-bit clustered VLIW architecture based on the Lx/ST200 (FARABOSCHI *et al.*, 2000), a family of VLIW processors used to power embedded media devices. It features a symmetric clustered architecture that can provide scalability and customization of clusters and issue width, two characteristics that no existing architecture had at the time.

The development of VEX consisted of three components: the VEX ISA, the VEX Compiler toolchain, and the VEX Simulation System (HEWLETT-PACKARD LABORATORIES, [s.d.]). The main purpose was the development of a VLIW architecture that could be introduced to undergraduate students, in order to teach concepts related to VLIWs, compiler-specific optimizations, such as loop unrolling, and an easy-to-use platform for testing.

The VEX defines a 32-bit RISCV-like VLIW architecture that aims at accelerating audio, video, and signal processing applications. It is also characterized by having four clusters, each with its own set of functional units, latencies and register files. Clustering is a technique used to explore scalability in the VLIWs. Partitioning large issue-width VLIWs into a clustered fashioned help minimizing power and energy consumption in these processors (TERECHKO; S, [s.d.]), since the cost of adding extra ports to register files would escalate drastically in terms of area, energy consumption and latency (TATSUMI; MATTAUSCH, 1999). A 32-issue single-clustered VLIW, for example, would require 64 read and 32 write ports in a register file, while a 32-issue four-clustered VLIW would require the same 64 and 32 ports, spread in 4 different register files (16 read and 8 write ports each), which are less energy consuming.

VEX also provide a compiler toolchain based upon the Multiflow C Compiler (LOWNEY *et al.*, 1993), a robust VLIW compiler that used a compiler optimization known as

trace scheduling, aiming at finding common paths on the code and rearranging machine instructions in order to improve parallelism and performance within multiple basic blocks (FISHER, J A, 1981).

5.1.1.1 ρVEX Processors

ρVEX (WONG, Stephan; AS, VAN; BROWN, 2008) is a VLIW processor that implements the VEX ISA and supports dynamic reconfiguration and different issues widths. The processor uses a five-stage pipeline structure very similar to the original MIPS processor (PATTERSON; HENNESSY, 2013).

The processor can be reconfigured to run in 2-, 4- and 8-issue configuration. For our experiments, we have used the 8-issue ρVEX processor with eight ALUs, four multipliers, one branch unit and one memory unit. We have also included one SMM memory for each pipelane (or issue), totalizing eight internal memories, in a similar fashion to Figure 4.1.

We have chosen to use the ρVEX for our test cases since its VHDL code is accessible to the academic community, and the background of the author in the ISA is strong, therefore, the technique could be easily integrated to the hardware. As it will be shown later, however, we have taken a different approach to accelerate the experimentation process.

5.1.1.2 VEX Simulation Platform

VEX also provides a simulation system that permits the execution of VEX code in the host machine for a fast execution of applications. The compiler can generate executable code for x86 machines by outputting a C-like application that emulates the execution of VEX instructions. It also includes callbacks to a built-in level-one cache library, allowing the emulation of cache requests and providing a complete infrastructure for executing VEX applications in host machines, and therefore, no specific VEX processor is required.

Coupled with the simulator, HP provides a library where users are allowed to include new customized instructions to the simulator. Instructions that have no side effects in memory can be emulated easily. Our SMM-specific instructions, detailed in section 4.2.1, were added to this library and software-managed memories were defined as arrays of bytes, allowing the library to have access to them. An insertion of an SMM instruction leads to a callback to this library, in other words, whenever an SMM instruction needs executing, this library is called

and the specific SMM is selected to either load or store a value to/from specific local memory addresses.

5.1.2 LLVM

As discussed in Chapter 4, SoMMA uses a memory architecture that requires modifications in hardware and software. Perhaps the most important and complex aspect of SoMMA is its software layer, which employs LLVM as the compiler framework for its code transformations (see section 4.2 Software). Though our contribution mainly focuses on proposing a new memory architecture for multi-issue processors, the implementation of a new backend to support a single-clustered VEX architecture in the LLVM framework was a major part of our work, generating code for ρVEX of up to 16-issue (even though ρVEX is limited to having a maximum of 8 issues). and thus, we make no use of the default VEX Compiler toolchain for code generation.

The code transformation described in section 4.2.4 was implemented as an LLVM pass in the compiler framework. The modularity of LLVM showed to be a perfect fit for our work, as the addition of new optimizations can easily be done through the LLVM Pass Infrastructure (see in 2.2.2.4). LLVM also provides easy-to-manipulate data structures and algorithms to instructions and basic blocks, helping programmers to focus on what really matters in their transformations or analysis.

5.1.3 Cacti-p

Cacti-p (LI, S. *et al.*, 2011) is an architecture-level framework for measuring power, energy, area and time constraints of Static Random Access Memory (SRAM)-based components. It supports modeling of many power-specific characteristics, such as power gating, long channel devices, and Hi-k metal gate, which makes it suitable for a range of situations in which power-specific attributes are required.

We have used Cacti-p to obtain energy consumption information for two components: the software-managed memories and caches. Dynamic Random Access Memory (DRAM), on the other hand, was modeled with DRAMPower, as shown below. Among other information, Cacti-p outputs values for the total static power and dynamic energy per read and write operation, allowing us to measure the total energy consumption for the SMMs and caches.

Moreover, authors claim the framework provides results of power, area, and timing with an average error around 15%, and an average error for leakage power between 5% and 14% when validating against industrial SRAM designs.

5.1.4 DRAMPower

DRAMPower (CHANDRASEKAR *et al.*, [s.d.]) is an open source tool for fast and accurate DRAM power and energy estimation for DDR2/DDR3/DDR4 and LPDDR/LPDDR2/LPDDR3 memories based on JEDEC standards. It provides two levels of abstraction in order to facilitate integration with existing system design flows: command and transaction level. At command level, DRAMPower can be integrated to DRAM controllers of existing systems and simulate requests through commands. Users without memory controller access can interact with DRAMPower using the memory transactions at the transactional level.

According to (KARTHIK CHANDRASEKAR, 2014), DRAMPower provides an average accuracy of 97% when compared with real power measurements from hardware for different DRAM operations, with its model being validated against a circuit-level DRAM power model.

5.2 Methodology

This section presents the experimental setup for our tests, discussing the characteristics of processors, caches and DRAM. We also present the benchmarks for comparison and experimental results.

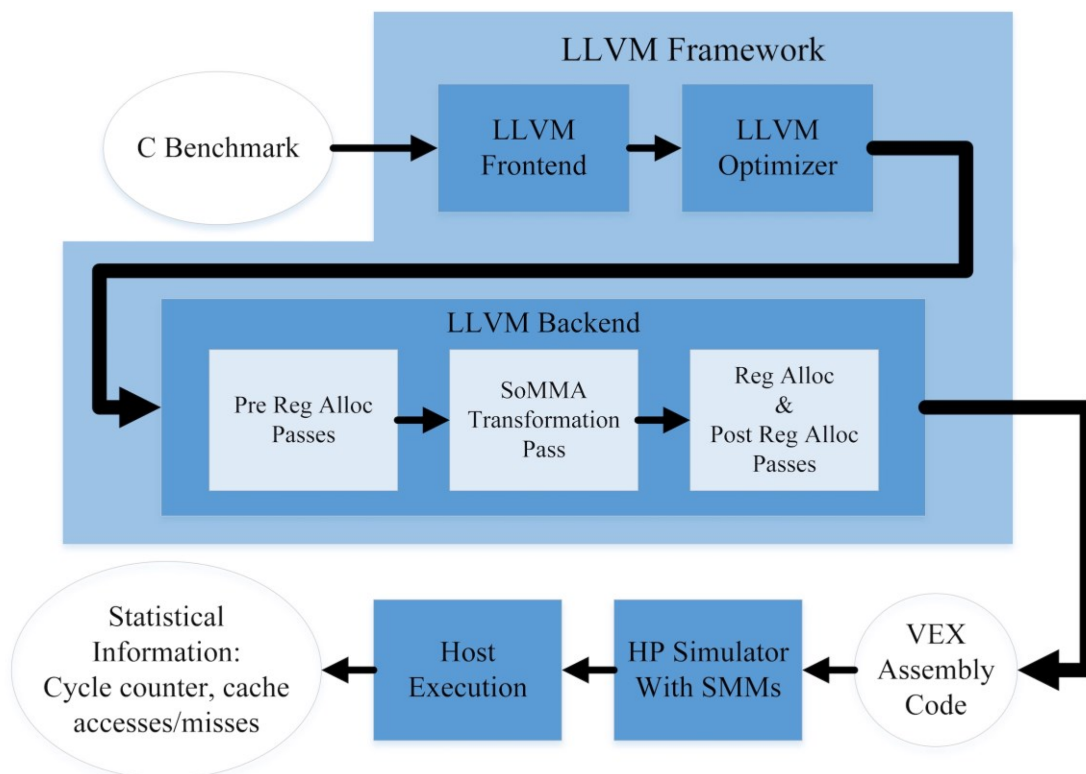
5.2.1 Experimental Setup

Experiments consist of measuring speedup, energy consumption and the energy-delay product (EDP) from different perspectives, which will be detailed in their correspondent subsections. All results consider two scenarios that aim at maintaining similar area between processors. The same amount of memory added with SMMs was subtracted from the L1 data cache, thereby maintaining the same total memory. The first scenario consists of a baseline processor with 8 KB of data cache, and a modified processor with 4 KB of data cache, and eight 512 Bytes SMMs, for a total of 8 KB. In the second scenario, we extend the size of data storage

to a total of 32 KB. Thus, the baseline processor has 32 KB of data cache, and the modified processor has 16 KB of data cache, and eight 2 KB SMMs. These are conservative approaches as the extra memory locations on the cache require additional tag storage. ICache of 8 KB and 32KB were used for each scenario, respectively, so that instructions and data memory have equivalent space. Furthermore, we consider systems with 2 GB of Low-power DDR2 (LPDDR2) memory, as we target embedded systems. Processors that implement our technique are denoted as SoMMA processors, while those that make no use of software-managed memories are the baselines.

Figure 5.1 illustrates the workflow diagram used for the experimental evaluation of SoMMA. We observe the key role of LLVM in the process, generating assembly code and performing the transformation required by SoMMA (for a better explanation of how LLVM works, refer to section 2.2.2). After the assembly code generation, a modified HP Compiler/Simulator is used to generate executable code for the x86 Host, as explained in 5.1.1.2. The simulator instructs the host application to output statistics for the executed program, with information about cycle counter, cache accesses and misses, essential for performance, energy and EDP evaluation.

Figure 5.1 – Workflow diagram for experiments.



Source: author

5.2.2 Performance

Performance was evaluated in terms of full-system configurations by taking into consideration instructions' cycles and cache accesses and misses, emulating a system with processor, data memory hierarchy (composed of either data cache only, or data cache and SMMs), instruction cache, and DRAM memory (note that misses in the caches are equivalent to data fetching on the DRAM).

5.2.3 Energy consumption

We have considered two scenarios for the evaluation of energy consumption: dynamic energy in the data memory hierarchy, and a full-system evaluation. Dynamic energy consumption was measured considering the energy cost per access, shown in Table 5.1, and total number of access (reads and writes) to data cache and SMMs. Full-system energy comparison between processors respects the following equation:

$$E_{total} = E_{proc.} + E_{ICache} + E_{DCache} + E_{DRAM}$$

where the energy is the sum of static and dynamic energy consumption.

Cacti-p (LI, S. *et al.*, 2011) was used to measure energy from cache and SMMs components, using a 65nm technology, as detailed in Section 5.1.3. DRAM energy information was obtained with DRAMPower (CHANDRASEKAR *et al.*, [s.d.]). Static energy is calculated by multiplying static power of the memory with the application's execution time. Dynamic energy in all memories is calculated according to the energy cost per access and the number of reads and writes for the memory. The formulas used to calculate static and dynamic energy consumption are as follows:

Table 5.1 – Energy cost per access in Caches and SMMs

	Read (pJ)	Write (pJ)
Cache 4KB	10.91	14.17
Cache 8KB	16.79	19.83
Cache 16KB	24.51	26.05
Cache 32KB	41.97	38.10
SMM 512B	1.34	4.98
SMM 2KB	3.35	6.24

Source: author

$$E_{static} = P_{static} * t_{exec}$$

$$E_{dynamic} = Num_{read} * E_{Read} + Num_{write} * E_{write}$$

where P_{static} represents the static power, Num_{read} is the total of reads (or loads), E_{Read} is the energy cost per read, Num_{write} is the total of writes (or stores), and E_{write} represents the energy cost per write.

For the processor core, we use energy consumption measurements from (SARTOR *et al.*, 2017; SARTOR; WONG, S; BECK, A C S, 2016), obtained through Cadence Encounter RTL compiler, using a 65 nm CMOS cell library from STMicroeletronics. The switching activity of the circuitry was set to 30%, which is a traditional value for system level analysis of microprocessors (GEUSKENS; ROSE, 2012). The equations used to estimate static and dynamic energy consumptions in the processors are as follows:

$$E_{static} = P_{static} * t_{exec}$$

$$E_{dynamic} = \sum_{i=0}^{7} P_{dyn_{Lane\,i}} * t_{exec} - P_{dyn_{Lane\,i}} * t_{nops_{Lane\,i}}$$

where $P_{dyn_{Lane\,i}}$ represents the dynamic power for lane i, and $t_{nops_{Lane\,i}}$ is the execution time that occurs a sequence of two *nop* instructions in lane i.

Note that static energy of the processor is calculated similarly to memories. On the other hand, dynamic energy is estimated using two switching activities: 30% (as shown above), and 0%, i.e., *idle* mode. When a *nop* is executed after another *nop,* there is no switching activity in the lane[1], and thus, we must discard these cases when evaluating dynamic energy.

---

[1] Note that we are not taking into consideration instruction fetch and decode in the processor pipeline, for simplicity.

Figure 5.2 – Compiler generates lane-specific *nops* when no instruction is executed. The example show that lanes 5, 6 and 7 have a *nop after nop* sequence

```
        c0      ldw $r0.21 = -516[$r0.11]
        c0      mpylu $r0.20 = $r0.18, $r0.14
        c0      mpyhs $r0.14 = $r0.18, $r0.14
        c0      mpyhs $r0.18 = $r0.19, $r0.17
        c0      mpylu $r0.17 = $r0.19, $r0.17
        c0      asm,105 $r0.0, $r0.0
        c0      asm,106 $r0.0, $r0.0
        c0      asm,107 $r0.0, $r0.0
        c0      asm,116 $r0.0, $r0.0
    ;;
        c0      ldw $r0.19 = 0[$r0.11]
        c0      add $r0.14 = $r0.20, $r0.14
        c0      asm,102 $r0.0, $r0.0
        c0      asm,103 $r0.0, $r0.0
        c0      asm,104 $r0.0, $r0.0
        c0      asm,105 $r0.0, $r0.0
        c0      asm,106 $r0.0, $r0.0
        c0      asm,107 $r0.0, $r0.0
        c0      asm,116 $r0.0, $r0.0
    ;;
```

Source: author

In order to measure *nop after nop* sequences, we have extended LLVM to print *nop* for all lanes not used in the bundle. Figure 5.2 shows an assembly code where lane-specific *nops* are included (they are represented by the *asm,10X* instructions, where *X* is the lane number). They are implemented in the same fashion as SMM instructions, through the addition of customized instructions within HP Platform. In the example, lanes 5 to 7 have *nop after nop* sequences. The second step consisted in adding lane-specific *nop* counters to the VEX Simulator, for computing the total of sequences for each lane. For every execution of *nop*, a callback to a library occurs (alike in SMM instructions), and counters can be updated. The last instruction in bundles *asm,116 $r0.0, $r0.0* guarantees that counters are updated in every cycle.

5.2.4 Energy-delay Product (EDP)

Techniques to improve performance might have a negative impact on energy consumption, thus EDP comes into play as a metric to evaluate the quality of a system. We have also evaluated full-system EDP for the selected benchmarks. These results are the product of performance (execution time) and energy consumption.

5.2.5 Benchmark Selection

For our experiments, we have selected 9 benchmarks: a discrete Fourier transform (DFT), an edge detection algorithm (edge), a Fourier inverse transform algorithm (itver2), three matrix multiplication benchmarks with different matrix sizes (m10x10, m16x16 and m32x32), the sum of absolute differences (SAD), the KMP string matching, and an x264 video encoder algorithm. These benchmarks provide data reuse and regular access patterns, which allow the use of our technique. The choice of using three matrix multiplication benchmarks comes from how our technique may change performance depending on the workload. LLVM generates different intermediate code for each benchmark, mainly due to different choices regarding loop unrolling.

5.3 Experimental Results

5.3.1 Performance

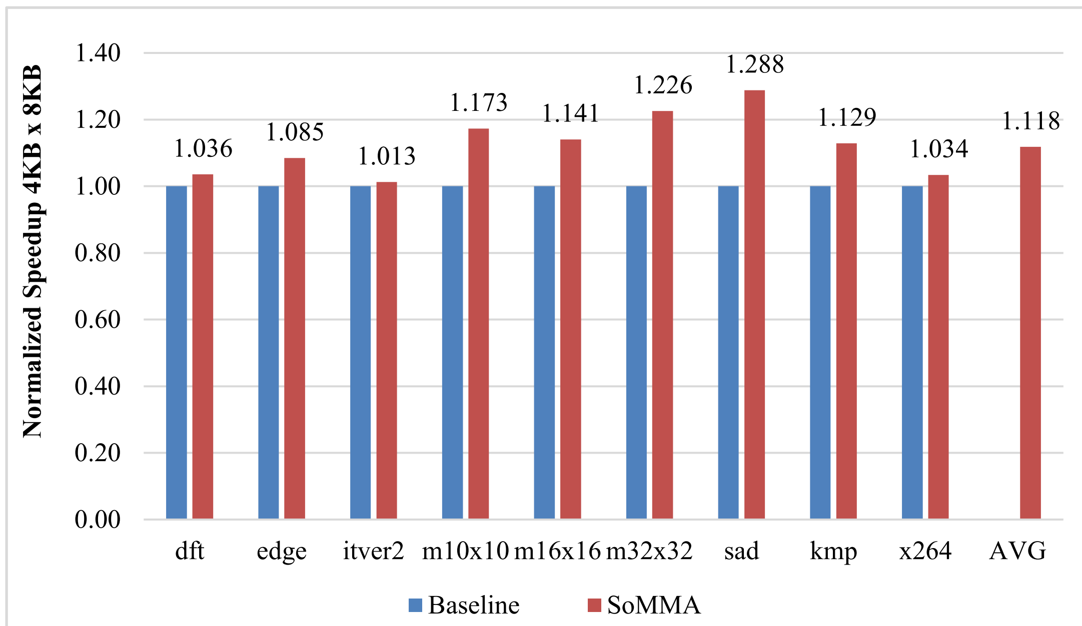
Figure 5.3 shows the speedup of the modified processors normalized by their respective baseline counterparts. Benchmarks dft, x264 and itver2 can provide good parallelism even for a single-port system, thus SoMMA was not capable of significantly improving performance for those benchmarks. We observe an average speedup of 1.118x and 1.121x in 4KB vs. 8KB and 16KB vs. 32 KB comparisons, respectively.

Performance improvements in SoMMA stem from allowing parallel access to data by using SMM allocation, as depicted in Figure 3.2. However, one could point out that such improvements are not significant considering the use of eight SMMs in parallel in comparison to the baseline, where a single-ported memory system is employed. There are two elements that help explaining these results:

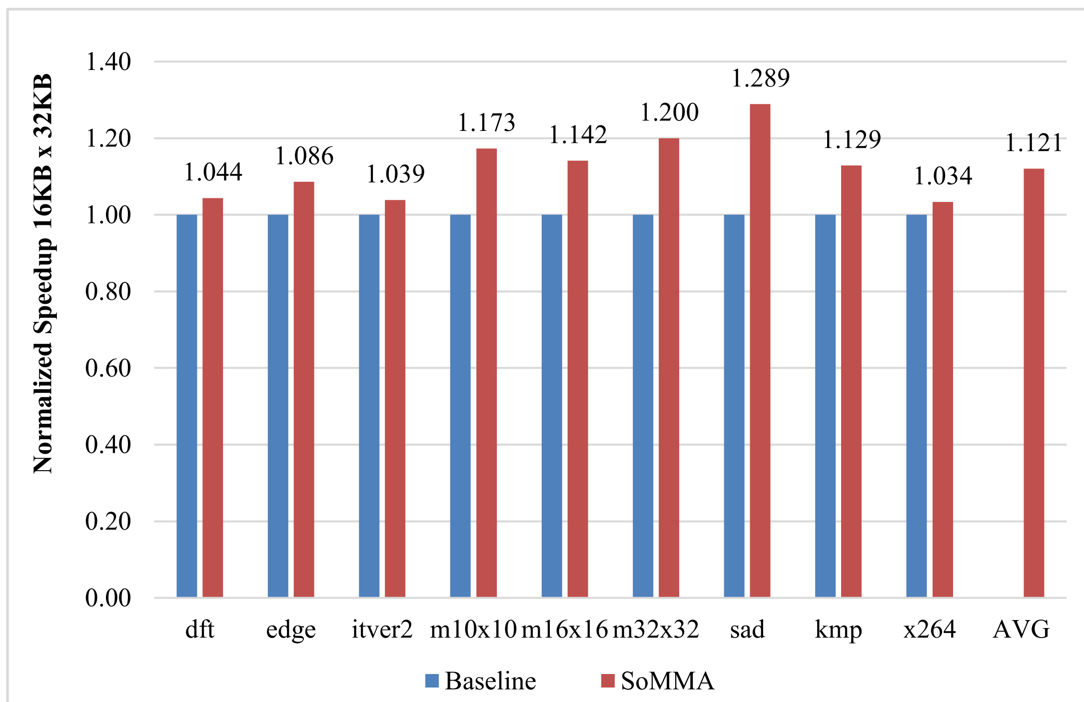
- Preamble insertion: because SoMMA uses memories controlled by software, a previous placement of data inside SMMs needs to happen, as explained in Section 4.2.2. The impact of the preamble code within execution time of applications is illustrated in Table 5.2. We notice an average of 7.6% increase in execution time, with two applications standing out from the rest, m10x10 and x264, with increases of 22.2% and 18.4%, respectively. Benchmarks *edge* and

Figure 5.3 – Normalized Speedup

(a) SoMMA processor with 4KB of DCache x Baseline with 8KB of DCache



(b) SoMMA processor with 16KB of DCache x Baseline with 32KB of DCache.



Source: author

kmp require no preamble code. Nonetheless, preamble code must be inserted whenever needed for correct execution

- Though data are fetched in parallel in SMMs, applications still use the single-ported cache to access part of the applications' data. Hence, we are still limited, at a smaller scale, to the single port, as applications process data that come from

Table 5.2 – Execution time increase when adding preamble code

Benchmark	Execution Time Increase
dft	1,032
edge	1,000
itver2	1,033
m10x10	1,222
m16x16	1,045
m32x32	1,097
sad	1,119
kmp	1,000
x264	1,184
Avg	1,076

Source: author

both types of memory. SMM leverages data reuse, storing data that can be reused later, while data cache will be used to fetch data that is not reused later. As an example, *kmp* stores in the SMMs two variables: the word to be searched, and the table used to backtrack the search when a mismatch happens, i.e., variables that are reused during the execution. The text, on the other hand, will be fetched through the data cache system since no data reuse occurs.

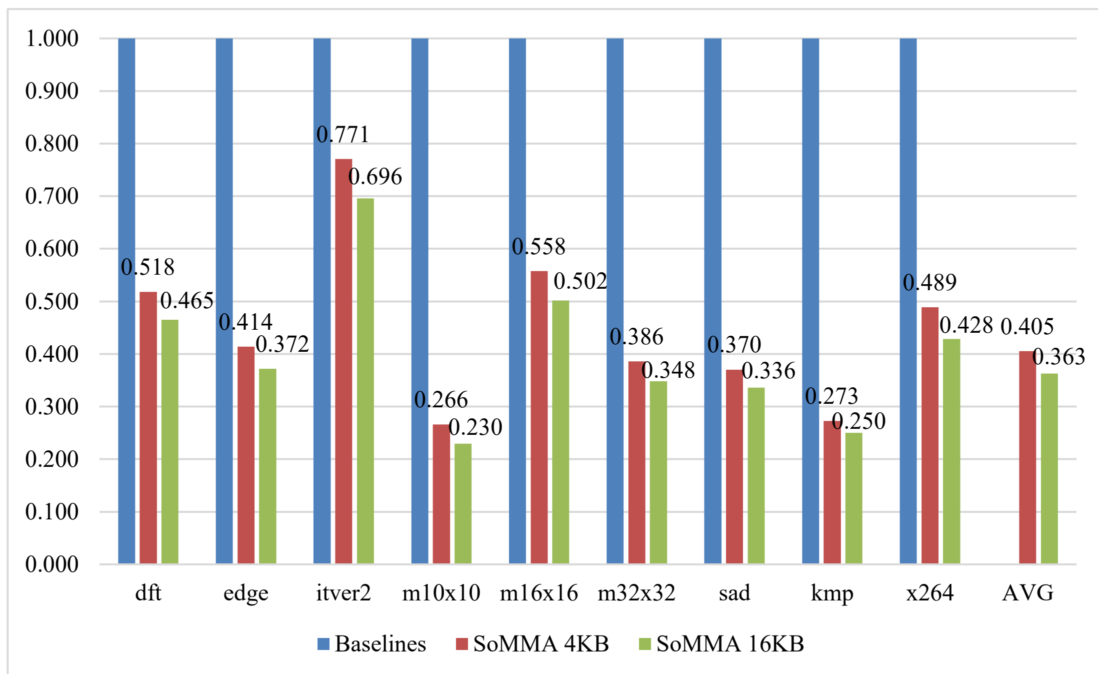
5.3.2 Energy Consumption

5.3.2.1 Dynamic energy consumption

Figure 5.4 illustrates this comparison. SoMMA 4KB and SoMMA 16KB consume 59.5% and 63.7% less energy in the data memory system compared to their baselines, respectively. Energy gains stem from two main reasons: each SMM access consumes substantially less than L1 accesses; and the L1 size reduction allows L1 accesses to consume less as well.

Figure 5.5 shows that, even though the total number of accesses to data memory hierarchy (either SMM or data cache) was increased in SoMMA, most accesses to the L1 were replaced by SMM accesses. Due to the differences in individual energy costs per access between caches and SMMs, depicted in Table 5.1, SoMMA can reduce dynamic energy in the data memory hierarchy significantly.

Figure 5.4 – Normalized dynamic energy comparison. Baselines have 8KB and 32KB of data cache when compared to SoMMA 4KB and SoMMA 16KB, respectively.

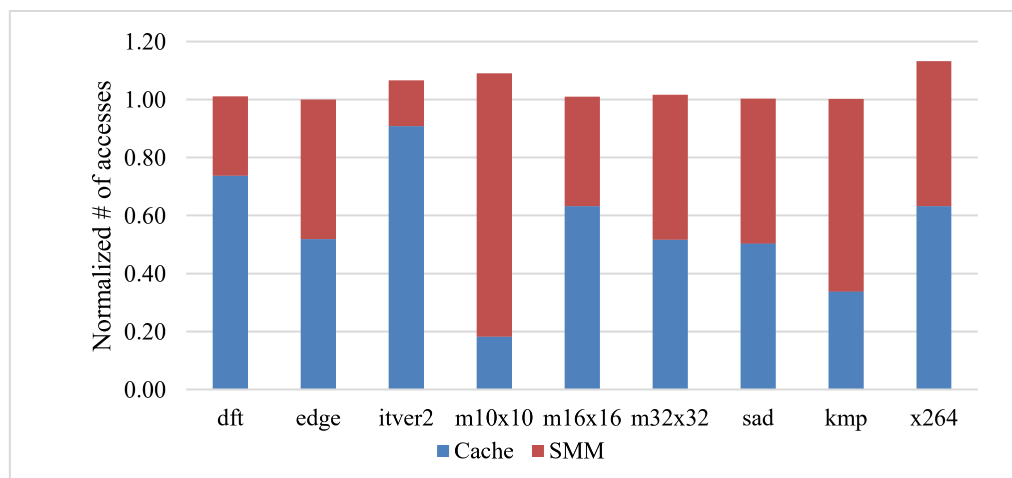


Source: author

5.3.2.2 Full-system comparison

Figure 5.6 depicts the full-system energy reduction, showing the normalized energy consumption for each component of the system (DRAM, processor, ICache, and Data

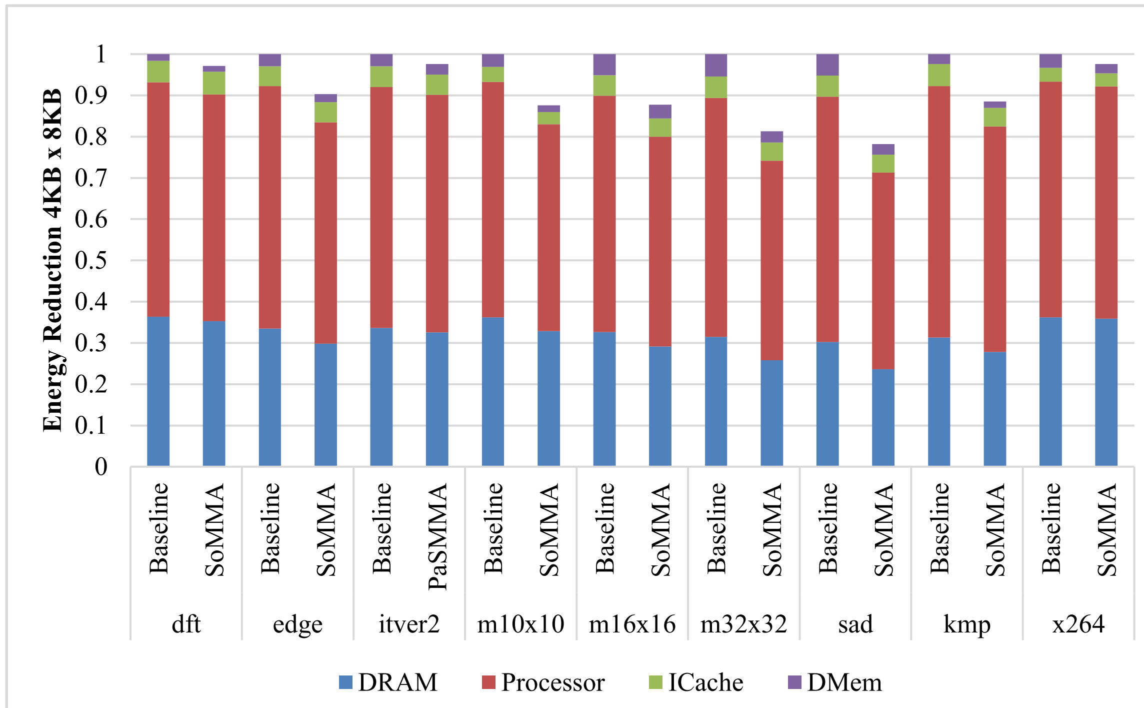
Figure 5.5 – Normalized number of accesses for the SoMMA processors in comparison to the baseline



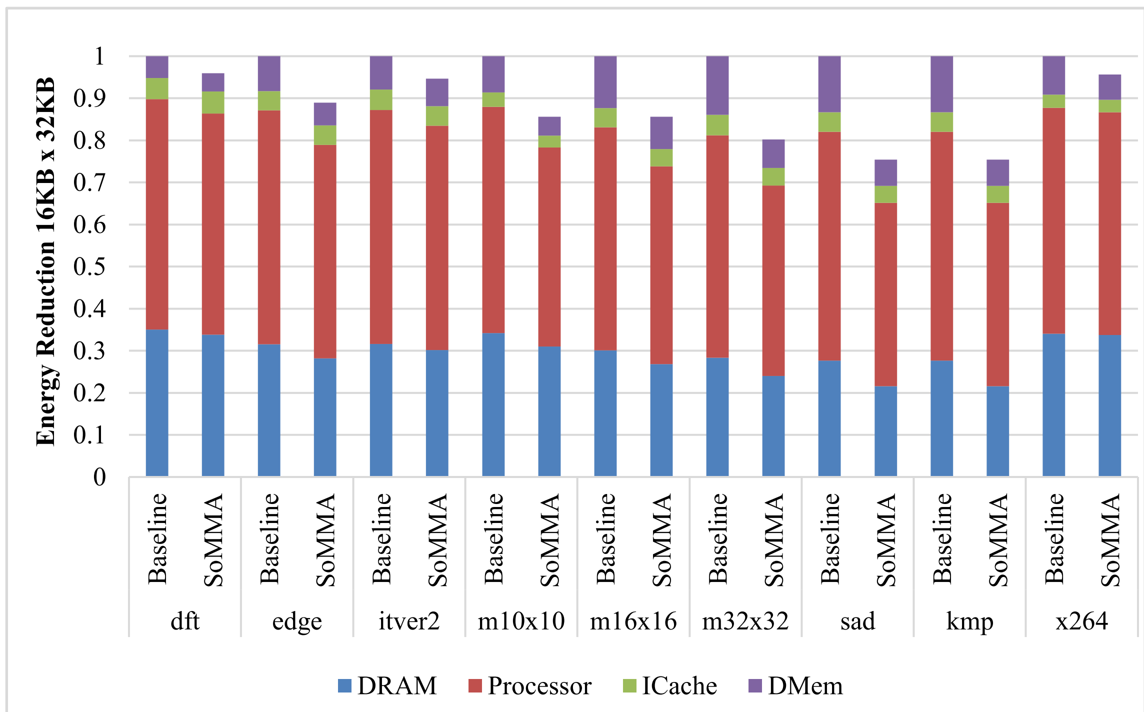
Source: author

Figure 5.6 – Normalized Energy in a full-system configuration

(a) SoMMA processor with 4KB of DCache x Baseline with 8KB of DCache,



(b) SoMMA processor with 16KB of DCache x Baseline with 32KB of DCache



Source: author

Memory). The Data memory (DMem) component consists of either DCache and SMMs (in SoMMA processors), or DCache (in baselines).

We have reduced energy consumption for almost every component in the applications. The impact of the preamble code can also be seen in the ICache comparison. Its insertion increased ICache accesses and misses in SoMMA processors, though without having a significant impact on the overall energy consumption. We also note how little influence the DMem dynamic energy consumption has in the full-system comparison, accounting for less than 5% of the overall value of energy consumption. Nonetheless, SoMMA consumes less energy than baselines due to two factors:

- Performance impacts the overall gains of SoMMA in energy consumption as it influences static energy on all types of memories and the processor.
- Main memory accesses also collaborate for reducing energy consumption. Static energy was improved due to better execution time, and dynamic energy was improved due to reduction in cache misses, detailed in section 5.3.4.

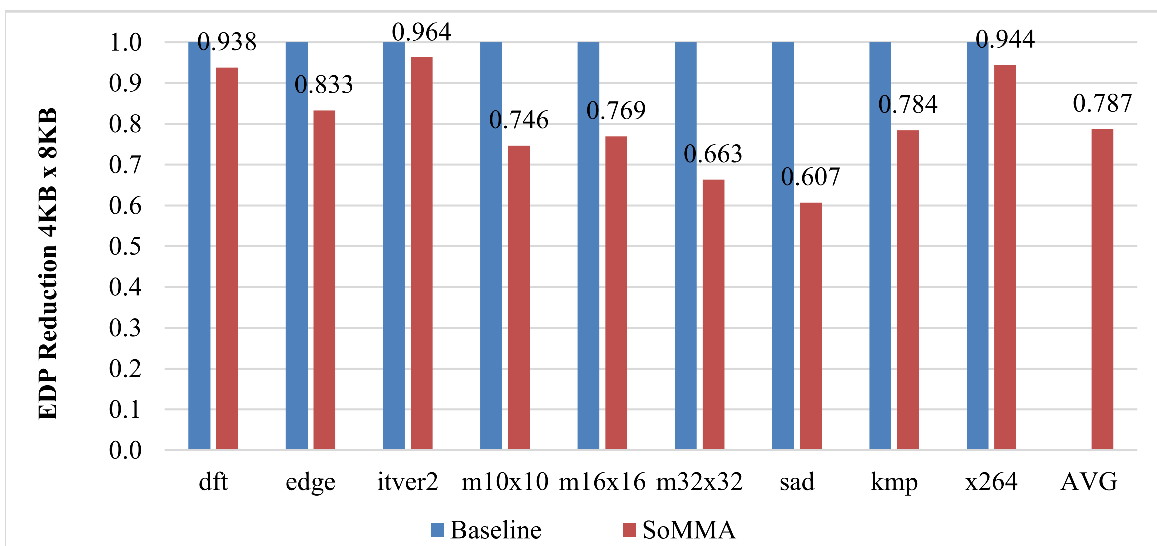
Results show an energy reduction of 11% when comparing a SoMMA processor with 4KB of DCache (and 4KB of SMMs) and the baseline with 8KB of DCache, and 12.8% in the comparison of an SoMMA processor with 16KB of DCache (and 16KB of SMMs) and a baseline with 32KB of DCache. Moreover, we notice how most of the energy consumption stems from main memory and processors.

5.3.3 Energy-delay Product (EDP)

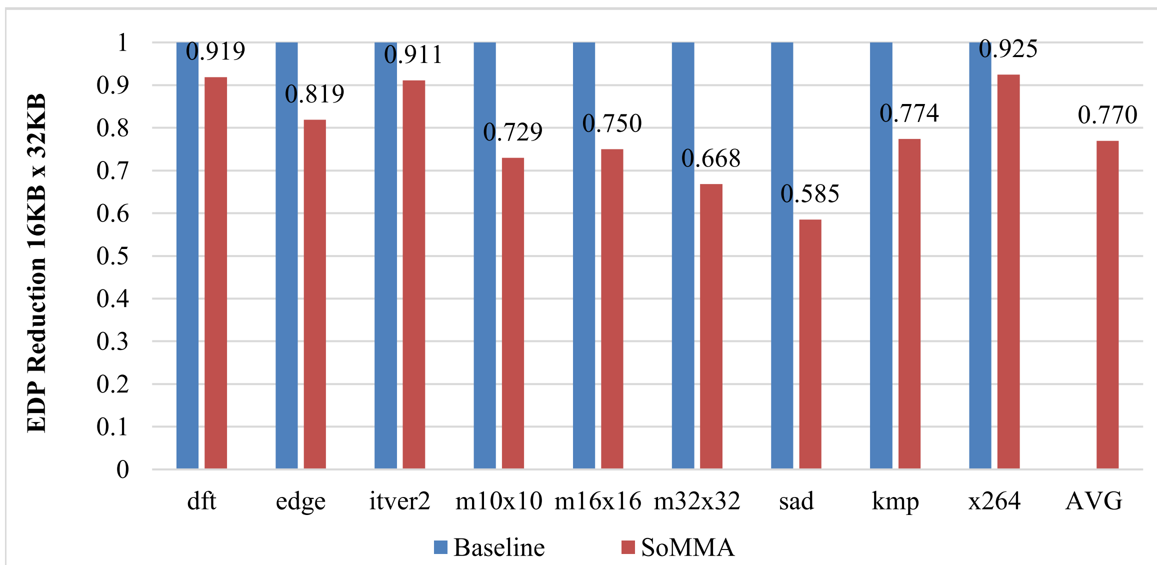
Figure 5.7 presents the comparison between processors, showing that SoMMA is more effective when it comes to EDP, because we show better results for both performance and energy consumption. We have reduced EDP by a factor of 21.3% in our 4KB vs 8KB comparison, and 23% when comparing SoMMA processor with 16KB of DCache (and 16KB of SMMs) and a baseline with 32KB of DCache.

Figure 5.7 – EDP reduction in a full-system configuration

(a) SoMMA processor with 4KB of DCache x Baseline with 8KB of DCache



(b) SoMMA processor with 16KB of DCache x Baseline with 32KB of DCache.



Source: author

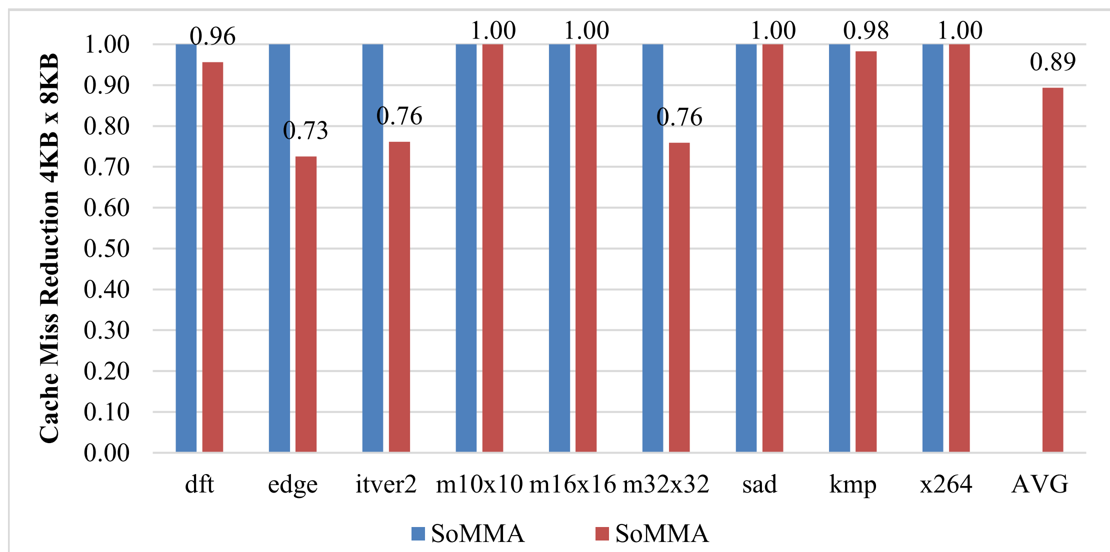
5.3.4 Data Cache Misses

In order to explain results in energy consumption, we can analyze the number of data cache misses between SoMMA and baseline processors. Although we consider SoMMA processors with half the data cache size of baseline processors due to the presence of SMMs to maintain area equivalence, SoMMA leverages data reuse to reduce data cache misses in many of the benchmarks.

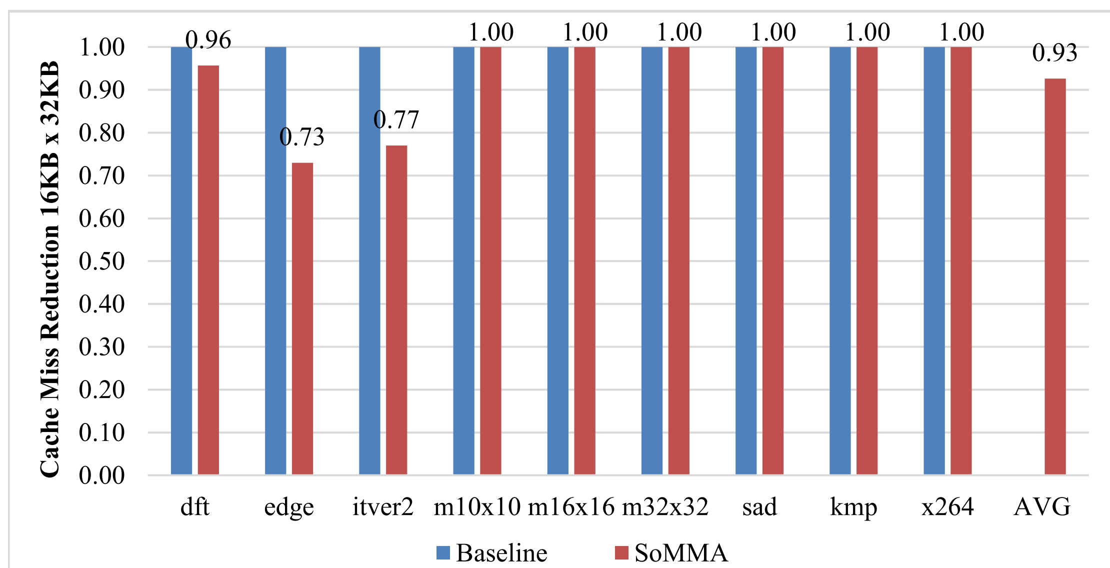
Figure 5.8 illustrates the data cache miss comparison between processors. It is noticed how some benchmarks, namely m10x10, m16x16, sad, and x264, can accommodate all

Figure 5.8 – Data cache miss reduction

(a) SoMMA processor with 4KB of DCache x Baseline with 8KB of DCache.



(b) SoMMA processor with 16KB of DCache x Baseline with 32KB of DCache



Source: author

application data in cache, leading to the same number of data cache misses in all processors. Applications that have a larger working set have reduced data cache misses in SoMMA processors, as many of those requests are replaced by SMM accesses and there is no need to accommodate such data into the cache. In our 4KB vs. 8KB comparison, five out of nine benchmarks showed a reduction in cache misses. As caches get larger, the difference between SoMMA and baseline narrows to only three cases. These results show that our software-based managing of data can exploit locality that traditional caching algorithms miss, thereby reducing overall miss rates.

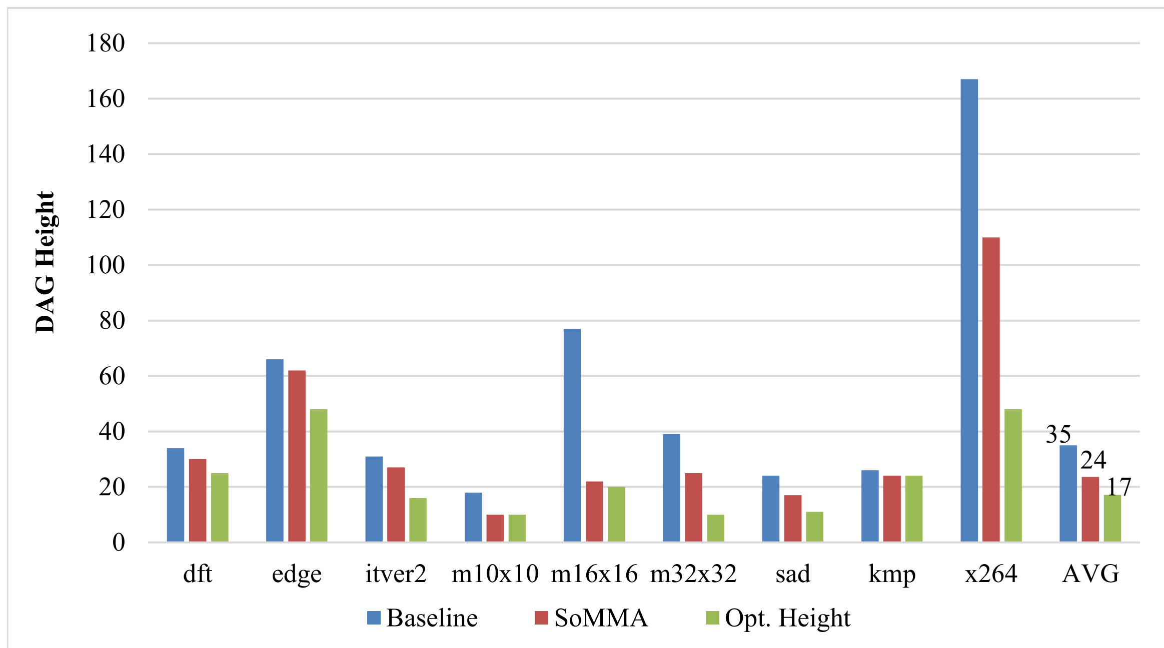
5.3.5 DAG Analysis

SoMMA has the advantage over other methods through the expansion of the usability of SMMs to handle data in parallel, allowing multi-issue processors to exploit a higher ILP. DAG analysis can be used to explain why SoMMA performs better than baselines when it comes to performance improvements. In these DAGs, vertices are instructions and edges indicate dependencies. Thus, the compiler's scheduler must accommodate the instructions in the lanes respecting the dependencies and each lane's capabilities. Once scheduled, the height of the DAG of a basic block indicates how many very long instruction words will have to be executed. When analyzing the height of the basic blocks, especially those that are part of the applications' kernels, we observe how our approach compresses DAG heights, which permits a faster performance.

Figure 5.9 shows the DAG heights of the most executed BB for every benchmark, which corresponds to nearly 90% of the applications' execution times. The baseline and SoMMA processors were set to work in a 24-issue configuration (maximum supported by the framework) and with a very large register file (256 registers), in order to minimize in number of issues and registers. This experiment's goal is to assess the proximity between our technique and the ideal processor with unlimited resources (i.e. maximum number of load/store units and any other operation) in comparison to the baseline. DAG heights are calculated according to the resources available for each processor, and the optimized height is calculated by setting up a processor with a number of resources of any kind to the maximum.

We can see how applications compiled with SMMs generate DAGs with shorter heights, enabling more opportunities of parallelism. SoMMA beats the baseline in all situations, approaching the optimal average height. The average height for the baseline is 35, while

Figure 5.9 – DAG Heights for the main BBs with their optimized values



Source: author

SoMMA provides an average DAG height of 24, with the optimal average DAG height of 17. Note that DAG height differences between baseline and SoMMA are not directly translated to performance improvements. Resource-limited processors can still reschedule instructions in order to minimize DAG heights. Nevertheless, SoMMA generates shorter DAGs that approach the minimum DAG heights possible.

5.3.6 Potential for improvements: Finite Impulse Response (FIR) Filter Case

The performance increase observed in SoMMA processors averaged around 12% over baselines, showing that combining software-managed memories and the data cache can open up new possibilities of parallelism. SoMMA provides a very simple approach to handle multiple memory locations at once, making it possible for users to handwrite their own kernel programs. We demonstrate through a finite impulse response (FIR) filter application how SoMMA can be used for manual code writing, showing that a fine-tuned compiler can potentially tackle other application domains.

5.3.6.1 Algorithm and Data placement

FIR is an extremely parallelizable application with few control-flow operations, a well-known and vastly used DSP application, providing great possibilities of parallelism. Due to how data are consumed, however, FIR requires a more refined and complex data replacement strategy than the ones proposed in Figure 4.5, which is not yet available at compiler. Thus we detail a handwriting FIR code that uses SoMMA and is able to provide very high parallelism.

Figure 5.10 shows a small snippet of C code for a filter application to calculate 256 output values for a 15th -order filter. We observe how the inner loop can easily be unrolled to explore ILP by executing multiple iterations at the same time. Another important quality of filters is that the number of data needed for calculating one output value is proportional to its order. For our 15th order filter, we need 32 values to produce one output: 16 coefficients and 16 input samples. Note, however, that only one datum, namely the latest input sample, was not used in the previous iteration of the outer loop. Due to data reusability, we can use SMMs to store both input and coefficient values for each outer loop iteration.

Figure 5.11 shows the placement of inputs and coefficients on the SMMs for four distinct outer loop iterations. The four depicted cases represent four consecutive iterations of the outer filter loop, and are the only required cases for an 8-issue processor. Coefficients are stored in the first 4 SPMs, while input values are stored in the SPMs 4 through 7. Colors represent inner loop iterations with an unrolled factor of 4, i.e., positions with the same color are fetched in parallel. The fetched data can then be element-wise multiplied and accumulated. Considering the 15th order filter and an 8-issue processor, after four parallel data fetches and multiplication rounds (represented by the four colors in Figure 5.11), one output sample is produced. Figure 5.11 also depicts the register allocation method for the application. The same registers are used to multiply one another, e. g., *r1* always multiplies *r8*, *r2* multiplies *r7*, and

Figure 5.10 – Small snippet of C-code for a FIR filter

```
// c[] = filter coefficients
// x[] = input values
// y[] = output values
for(int j = 0; j < 256; ++j) {
   int s = 0;    // s = accumulator
   for (int I = 0; I < 16; ++i) {
      s += c[i] * x[i+j];
   }
   Y[j] = s;
}
```

Source: author

Figure 5.11 – Data placement on the SMMs for FIR

Case 1

Memory address \ SMM	0	1	2	3	4	5	6	7
0	c_0 (r_1)	c_1 (r_2)	c_2 (r_3)	c_3 (r_4)	x_i (r_8)	x_{i-15} (r_5)	x_{i-14} (r_6)	x_{i-13} (r_7)
1	c_4 (r_1)	c_5 (r_2)	c_6 (r_3)	c_7 (r_4)	x_{i-12} (r_8)	x_{i-11} (r_5)	x_{i-10} (r_6)	x_{i-9} (r_7)
2	c_8 (r_1)	c_9 (r_2)	c_{10} (r_3)	c_{11} (r_4)	x_{i-8} (r_8)	x_{i-7} (r_5)	x_{i-6} (r_6)	x_{i-5} (r_7)
3	c_{12} (r_1)	c_{13} (r_2)	c_{14} (r_3)	c_{15} (r_4)	x_{i-4} (r_8)	x_{i-3} (r_5)	x_{i-2} (r_6)	x_{i-1} (r_7)

Case 2

Memory address \ SMM	0	1	2	3	4	5	6	7
0	c_0 (r_1)	c_1 (r_2)	c_2 (r_3)	c_3 (r_4)	x_{i-1} (r_7)	x_i (r_8)	x_{i-15} (r_5)	x_{i-14} (r_6)
1	c_4 (r_1)	c_5 (r_2)	c_6 (r_3)	c_7 (r_4)	x_{i-13} (r_7)	x_{i-12} (r_8)	x_{i-11} (r_5)	x_{i-10} (r_6)
2	c_8 (r_1)	c_9 (r_2)	c_{10} (r_3)	c_{11} (r_4)	x_{i-9} (r_7)	x_{i-8} (r_8)	x_{i-7} (r_5)	x_{i-6} (r_6)
3	c_{12} (r_1)	c_{13} (r_2)	c_{14} (r_3)	c_{15} (r_4)	x_{i-5} (r_7)	x_{i-4} (r_8)	x_{i-3} (r_5)	x_{i-2} (r_6)

Case 3

Memory address \ SMM	0	1	2	3	4	5	6	7
0	c_0 (r_1)	c_1 (r_2)	c_2 (r_3)	c_3 (r_4)	x_{i-2} (r_6)	x_{i-1} (r_7)	x_i (r_8)	x_{i-15} (r_5)
1	c_4 (r_1)	c_5 (r_2)	c_6 (r_3)	c_7 (r_4)	x_{i-14} (r_6)	x_{i-13} (r_7)	x_{i-12} (r_8)	x_{i-11} (r_5)
2	c_8 (r_1)	c_9 (r_2)	c_{10} (r_3)	c_{11} (r_4)	x_{i-10} (r_6)	x_{i-9} (r_7)	x_{i-8} (r_8)	x_{i-7} (r_5)
3	c_{12} (r_1)	c_{13} (r_2)	c_{14} (r_3)	c_{15} (r_4)	x_{i-6} (r_6)	x_{i-5} (r_7)	x_{i-4} (r_8)	x_{i-3} (r_5)

Case 4

Memory address \ SMM	0	1	2	3	4	5	6	7
0	c_0 (r_1)	c_1 (r_2)	c_2 (r_3)	c_3 (r_4)	x_{i-3} (r_5)	x_{i-2} (r_6)	x_{i-1} (r_7)	x_i (r_8)
1	c_4 (r_1)	c_5 (r_2)	c_6 (r_3)	c_7 (r_4)	x_{i-15} (r_5)	x_{i-14} (r_6)	x_{i-13} (r_7)	x_{i-12} (r_8)
2	c_8 (r_1)	c_9 (r_2)	c_{10} (r_3)	c_{11} (r_4)	x_{i-11} (r_5)	x_{i-10} (r_6)	x_{i-9} (r_7)	x_{i-8} (r_8)
3	c_{12} (r_1)	c_{13} (r_2)	c_{14} (r_3)	c_{15} (r_4)	x_{i-7} (r_5)	x_{i-6} (r_6)	x_{i-5} (r_7)	x_{i-4} (r_8)

Source: author

so on. That way, the logic of the convolution is satisfied. The preamble code, as described in a previous section, will fetch all coefficients and the input for the first iteration. Other iterations will only need to fetch one input value, since all other values are already in memory.

Note that, for each case, there is always one position of each color in each lane, meaning that all elements can be fetched in parallel, and all SMMs are used. Four cases are required because the algorithm needs to emulate a shift on the input buffer. The difference between the current iteration and the previous is the presence of a new datum that replaces the previous oldest datum and that is multiplied by the first coefficient. We handle fetches from SMMs through a software-managed circular buffer that keeps track of which data should be fetched and where the next input datum should be placed on the memories. For example, iteration 0 will use case 1. Iteration 1 will need to fetch one new value from memory and store in SMM 5 in the position assigned at the circular buffer. Other iterations will follow these same procedures. Note that this approach is general for circular buffers, and can be used in any application that needs this kind of data structure.

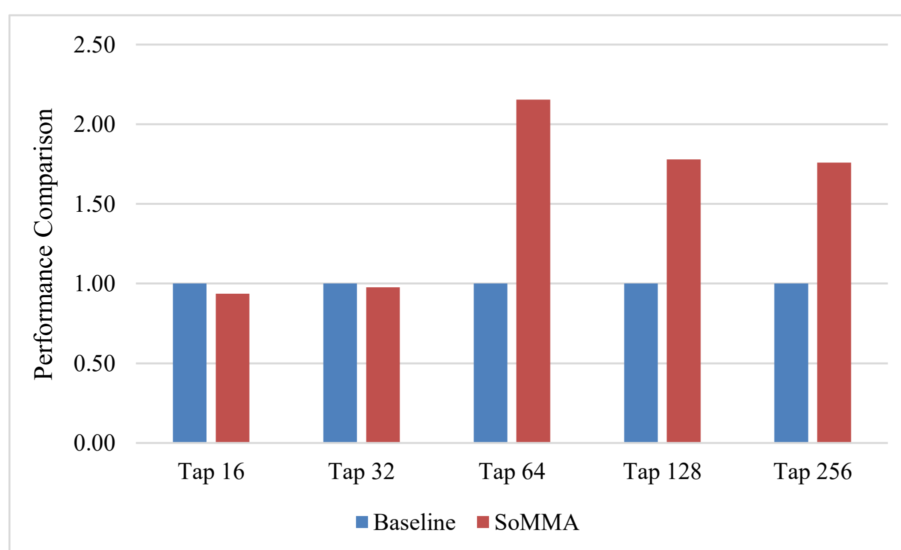
Moreover, the ILP improvement observed when compared to a regular single ported processor is noteworthy. Once the data is on the SMMs, all memory operations can be performed in parallel, since each internal memory is independent from one another. When using only a regular single-ported hierarchy, multiple load instructions would limit the ILP, preventing the possibility of performing parallel multiplications.

5.3.6.2 Performance Results

FIR was simulated with 256 input values and taps of 16 (FIR 16), 32 (FIR 32), 64 (FIR 64), 128 (FIR 128), and 256 (FIR 256) values. Figure 5.12 illustrates the speedup comparison between SoMMA and the baseline processors. Due to the negligible performance difference between 4KB vs 8KB and 16KB vs. 32KB configuration, we only show the comparison from the latter. Results show that SoMMA has a speedup of 1.75x for large number of taps, achieving up to 2.15x for Tap 64. Small tap sizes do not take advantage of the technique in comparison with the baseline, because the number of taps is sufficiently small to always be stored in the register file, therefore, there is no need for fetching these values from memory at every iteration. The performance slow-down observed in Tap 16 and Tap 32 come from the preamble code needed to accommodate coefficients and input data.

Moreover, the possibility of storing both input data and coefficients in SMMs and having only one datum to be fetched at every iteration source the performance improvements for large number of taps. We also notice that such behavior is different from previously analyzed benchmarks, that operated on both SMMs and data cache at a similar proportion. A fine-tuned compiler, that can understand and generate data placement for FIR-like benchmarks, is part of our future works enabling new possibilities of improvements to SoMMA.

Figure 5.12 – Performance Speedup for a handwritten FIR in SoMMA



Source: author

6 CONCLUSIONS AND FUTURE PERSPECTIVES

In this dissertation, we have presented SoMMA, a software-managed memory architecture for embedded multi-issue processors that combines software-managed memories and the data cache. Its main goal is to help mitigating the ever-going struggle between memory bandwidth and processor performance, by creating a layer of memories that can provide a higher bandwidth in comparison to cache-only based systems. SoMMA leverages applications' data reusability and the low cost of read/write operations of SMMs to avoid frequent access to data cache. It aims at providing reductions in energy end EDP while still offering a better ILP through the use of multiple software-managed memories in parallel. We have developed an LLVM-based backend for a VLIW processor that analyzes and transforms code to operate such memories in cooperation with the memory hierarchy.

Through a series of experiments, SoMMA showed to provide better performance, energy and also EDP in comparison to cache-only processors. SoMMA processors best baselines in average speedups of 1.118x and 1.121x in a set of benchmarks for the ρVEX processor. Energy consumption was reduced to 89% and 87.2%, while EDP showed reductions of 21.3% and 23%, all within experiments of full-system configurations with equivalent area. When considering the dynamic energy consumption in the data memory hierarchy solely, SoMMA showed reductions of 59.5% and 63.7% within the same experiments.

Moreover, data cache miss analysis shows that SoMMA can also reduce the number of cache misses even when halving cache size, demonstrating that SoMMA is effective when data reuse is presented in the application. Lastly, a DAG analysis experiment demonstrates that our solution bests DAG heights in baselines by a margin of 31.4%, and generates shorter DAG that approach the minimum DAG heights possible.

6.1 Future works

The work developed during this dissertation enlightens new possibilities for researchers. The algorithm for SoMMA currently works with global variables. Extending SoMMA to support stack and heap-allocated variables means widening the applicability of the architecture to new benchmarks, making SoMMA a more generic solution to reduce energy consumption and EDP in multi-issue processors.

Additionally, we demonstrate how SoMMA can be used through a VLIW processor, though other ILP-capable processors could also be considered. The rapid emerging of RISC-V

ISA (WATERMAN *et al.*, 2014), an open-source ISA for education, research and also industrial development, and its design to support extensive customization and specialization of new instructions in the ISA, would allow SoMMA to be expanded to RISC-V superscalar processors (ASANOVIC; PATTERSON; CELIO, 2015). Still in the category of ILP-capable processors, SoMMA could potentially be explored in SIMD processors or GPUs, as data can be placed at SMMs and accessed in a uniform fashion.

Section 5.3.6 also shows potential for improvements that can be exploited by tuning the compiler with new SMM data placements. We have seen the great performance improvements observed with a FIR benchmark when a more complex SMM placement can be performed. Future works on the compiler to generate new SMM data placements are envisioned by taking into consideration loop optimization techniques such as loop tiling, as well as exploring new instruction scheduling algorithms to make better use of SMMs. Additionally, SoMMA could potentially be combined with speculative execution and memory prefetching techniques for further improvements.

6.2 Publications

During the course of this dissertation, the following publications were accomplished by the author.

JOST, T.; NAZAR, G.; CARRO, L. Improving Performance in VLIW Soft-core Processors through Software-controlled ScratchPads. **Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2016 International Conference on**, 2016

JOST, T.; NAZAR, G.; CARRO, L. An Energy-Efficient Memory Hierarchy for Multi-Issue Processors. **Design, Automation & Test in Europe Conference & Exhibition (DATE)**, 2017.

JOST, T. T.; NAZAR, G. L.; CARRO, L. Scalable memory architecture for soft-core processors. **The IEEE International Conference on Computer Design (ICCD)**, 2016. p. 396–399.

REFERENCES

ADLER, M. *et al.* Leap scratchpads: automatic memory and cache management for reconfigurable logic. In: INTERNATIONAL SYMPOSIUM OF FIELD PROGRAMMABLE GATE ARRAYS, 19th, 2011. **Proceedings.** ACM/SIGDA, p. 25–28.

AHO, A. V *et al.* **Compilers: Principles, Techniques, and Tools (2Nd Edition)**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

ANGIOLINI, F. *et al.* A Post-compiler Approach to Scratchpad Mapping of Code. In: INTERNATIONAL CONFERENCE ON COMPILERS, ARCHITECTURE, AND SYNTHESIS FOR EMBEDDED SYSTEMS, 2004. **Proceedings.** New York, NY, USA: ACM, 2004. p. 259–267.

APPLE INC. LLVM Compiler Overview. [s.d.]. Available at: < https://developer.apple.com/library/content/documentation/CompilerTools/Conceptual/LLVM CompilerOverview/index.html>. Accessed: July 10, 2017

ASANOVIC, K.; PATTERSON, D. A.; CELIO, C. **The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor**. [S.l.]: [s.n.], 2015.

AVISSAR, O.; BARUA, R.; STEWART, D. An optimal memory allocation scheme for scratch-pad-based embedded systems. New York, NY, USA: **ACM Transactions on Embedded Computing Systems**, nov. 2002. v. 1, n. 1, p. 6–26.

BAJWA, H.; CHEN, X. Low-Power High-Performance and Dynamically Configured Multi-Port Cache Memory Architecture. In: INTERNATIONAL CONFERENCE ON ELECTRICAL ENGINEERING, 2007. **Proceedings.** 2007. p. 1–6.

BETKAOUI, B.; THOMAS, D. B.; LUK, W. Comparing performance and energy efficiency of FPGAs and GPUs for high productivity computing. In: INTERNATIONAL CONFERENCE ON FIELD-PROGRAMMABLE TECHNOLOGY, 2010. **Proceedings**. IEEE, p. 94–101.

BRANDON, A.; WONG, S. Support for dynamic issue width in VLIW processors using generic binaries. In: DESIGN, AUTOMATION TEST IN EUROPE CONFERENCE EXHIBITION, 2013. **Proceedings.** 2013. p. 827–832.

CHANDRA, R. **Parallel programming in OpenMP**. [S.l.]: Morgan Kaufmann, 2001.

CHANDRASEKAR, K. *et al.* DRAMPower: Open-source DRAM Power & Energy Estimation Tool. [S.l.], [s.d.]. Available at: <http://www.drampower.info>. Accessed: August 8, 2017

CHE, W.; CHATHA, K. S. Scheduling of stream programs onto SPM enhanced processors with code overlay. In: SYMPOSIUM ON EMBEDDED SYSTEMS FOR REAL-TIME MULTIMEDIA, 2011. **Proceedings.** IEEE, 2011. p. 9–18.

CODRESCU, L. Qualcomm Hexagon DSP: An architecture optimized for mobile multimedia and communications. In: HOT CHIPS SYMPOSIUM, 2013. **Proceedings.** IEEE, 2013. p. 1–

23.

COOPER, K.; TORCZON, L. **Engineering a compiler**. [S.l.]: Elsevier, 2011.

CYTRON, R. *et al.* Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. New York, NY, USA: **ACM Trans. Program. Lang. Syst.**, out. 1991. v. 13, n. 4, p. 451–490.

DEHNERT, J. C. The Transmeta Crusoe: VLIW Embedded in CISC. In: KRALL, A. (Org.). **Software and Compilers for Embedded Systems: 7th International Workshop, SCOPES 2003, Vienna, Austria, September 24-26, 2003. Proceedings**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, p. 1.

DENNARD, R. H. *et al.* Design of ion-implanted MOSFET's with very small physical dimensions. **IEEE Journal of Solid-State Circuits**, out. 1974. v. 9, n. 5, p. 256–268.

EVANS, J. S.; TRIMPER, G. L. **Itanium Architecture for Programmers: Understanding 64-bit Processors and EPIC Principles**. [S.l.]: Prentice Hall PTR, 2003.

FAIRCHILD. Fairchild F3850 - Central Processing Unit (CPU). 1975.

FARABOSCHI, P. *et al.* Lx: A Technology Platform for Customizable VLIW Embedded Processing. New York, NY, USA: **SIGARCH Comput. Archit. News**, 2000. ISCA '00. v. 28, n. 2, p. 203–213.

FISHER, J. A. Trace Scheduling: A Technique for Global Microcode Compaction. Washington, DC, USA: **IEEE Trans. Comput.**, jul. 1981. v. 30, n. 7, p. 478–490.

FLACHS, B. *et al.* The microarchitecture of the synergistic processor for a cell processor. **IEEE Journal of Solid-State Circuits**, jan. 2006. v. 41, n. 1, p. 63–70.

FORUM, M. P. **MPI: A Message-Passing Interface Standard**. Knoxville, TN, USA: University of Tennessee, 1994.

GEUSKENS, B.; ROSE, K. **Modeling microprocessor performance**. [S.l.]: Springer Science & Business Media, 2012.

HENNESSY, J. L.; PATTERSON, D. A. **Computer Architecture, Fifth Edition: A Quantitative Approach**. 5th. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

HEWLETT-PACKARD LABORATORIES. VEX Toolchain. [S.l.], [s.d.]. Avaiable at: <http://www.hpl.hp.com/downloads/vex/>. Accessed: August 9, 2017

KADRIC, E.; LAKATA, D.; DEHON, A. Impact of Memory Architecture on FPGA Energy Consumption. In: INTERNATIONAL SYMPOSIUM ON FIELD PROGRAMMABLE GATE ARRAYS, 2015. **Proceedings.** ACM, p. 146–155.

KARTHIK CHANDRASEKAR. **High-Level Power Estimation and Optimization of DRAMs**. [S.l.]: Delft University of Technology, 2014.

KIRK, D. NVIDIA Cuda Software and Gpu Parallel Computing Architecture. In:

INTERNATIONAL SYMPOSIUM ON MEMORY MANAGEMENT, 2007. **Proceedings.** New York, NY, USA: ACM, 2007. p. 103–104.

KOMURAVELLI, R. *et al.* Stash: Have your scratchpad and cache it too. In: INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 2015. **Proceedings.** ACM/IEEE, p. 707–719.

KUCK, D. L. **Structure of Computers and Computations**. New York, NY, USA: John Wiley & Sons, Inc., 1978.

LAFOREST, C. E.; STEFFAN, J. G. Efficient multi-ported memories for FPGAs. In: INTERNATIONAL SYMPOSIUM ON FIELD PROGRAMMABLE GATE ARRAYS, 2010. **Proceedings.** ACM, p. 41–50.

LATTNER, C.; ADVE, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: INTERNATIONAL SYMPOSIUM ON CODE GENERATION AND OPTIMIZATION: FEEDBACK-DIRECTED AND RUNTIME OPTIMIZATION, 2004. **Proceedings.** Washington, DC, USA: IEEE Computer Society, 2004. p. 75--.

LI, S. *et al.* CACTI-P: Architecture-level Modeling for SRAM-based Structures with Advanced Leakage Reduction Techniques. In: INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN, Piscataway, NJ, USA, 2011. **Proceedings.** IEEE Press, 2011. p. 694–701.

LOPES, B. C.; AULER, R. **Getting Started with LLVM Core Libraries**. [S.l.]: Packt Publishing, 2014.

LOWNEY, P. G. *et al.* The multiflow trace scheduling compiler. **The Journal of Supercomputing**, maio. 1993. v. 7, n. 1, p. 51–142.

MALAZGIRT, G. A. *et al.* Application specific multi-port memory customization in FPGAs. In: INTERNATIONAL CONFERENCE ON FIELD PROGRAMMABLE LOGIC AND APPLICATIONS, 2014. **Proceedings.** 2014, p. 1–4.

MENG, C. *et al.* Efficient memory partitioning for parallel data access in multidimensional arrays. In: DESIGN AUTOMATION CONFERENCE, 2015. **Proceedings.** ACM, 2015. p. 160.

MOAZENI, M.; BUI, A.; SARRAFZADEH, M. A memory optimization technique for software-managed scratchpad memory in GPUs. In: SYMPOSIUM ON APPLICATION SPECIFIC PROCESSORS, 2009. **Proceedings.** IEEE, 2009. p. 43–49.

PANDA, P. R.; DUTT, N. D.; NICOLAU, A. Efficient Utilization of Scratch-Pad Memory in Embedded Processor Applications. In: EUROPEAN CONFERENCE ON DESIGN AND TEST, 1997. **Proceedings.** Washington, DC, USA: IEEE Computer Society, 1997. p. 7--.

PATTERSON, D. A.; HENNESSY, J. L. **Computer Organization and Design, Fifth Edition: The Hardware/Software Interface**. 5th. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013.

PETTIS, K.; HANSEN, R. C. Profile Guided Code Positioning. In: INTERNATIONAL CONFERENCE ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION. **Proceedings.** New York, NY, USA: ACM, 1990. p. 16–27.

PUSCHNER, P.; BURNS, A. Guest Editorial: A Review of Worst-Case Execution-Time Analysis. **Real-Time Systems**, 2000. v. 18, n. 2, p. 115–128.

SARTOR, A. L. *et al.* Exploiting Idle Hardware to Provide Low Overhead Fault Tolerance for VLIW Processors. New York, NY, USA: **J. Emerg. Technol. Comput. Syst.**, 2017. v. 13, n. 2, p. 13:1--13:21.

SARTOR, A. L.; WONG, S.; BECK, A. C. S. Adaptive ILP control to increase fault tolerance for VLIW processors. In: INTERNATIONAL CONFERENCE ON APPLICATION-SPECIFIC SYSTEMS, ARCHITECTURES AND PROCESSORS, 2016. **Proceedings.** IEEE, 2016, p. 9–16.

STEINKE, S. *et al.* Assigning program and data objects to scratchpad for energy reduction. In: DESIGN, AUTOMATION AND TEST IN EUROPE CONFERENCE AND EXHIBITION, 2012. **Proceedings.** IEEE, 2002. p. 409–415.

TATSUMI, Y.; MATTAUSCH, H. J. Fast quadratic increase of multiport-storage-cell area with port number. **Electronics Letters**, 1999. v. 35, n. 25, p. 2185–2187.

TERECHKO, A. S.; S, T. A. **Clustered VLIW Architectures: a Quantitative Approach**.

TOMIYAMA, H.; YASUURA, H. Optimal code placement of embedded software for instruction caches. In: EUROPEAN DESIGN AND TEST CONFERENCE, 1996. **Proceedings.** IEEE: 1996. p. 96–101.

UDAYAKUMARAN, S.; DOMINGUEZ, A.; BARUA, R. Dynamic allocation for scratch-pad memory using compile-time decisions. **ACM Transactions on Embedded Computing Systems (TECS)**, 2006. v. 5, n. 2, p. 472–511.

VERMA, M.; MARWEDEL, P. Overlay techniques for scratchpad memories in low power embedded processors. **Very Large Scale Integration (VLSI) Systems, IEEE Transactions on**, 2006. v. 14, n. 8, p. 802–815.

VERMA, M.; WEHMEYER, L.; MARWEDEL, P. Cache-aware scratchpad allocation algorithm. In: DESIGN, AUTOMATION AND TEST IN EUROPE CONFERENCE, 2004. **Proceedings.** IEEE, 2004. p. 21264.

WANG, Z.; GU, Z.; SHAO, Z. WCET-Aware Energy-Efficient Data Allocation on Scratchpad Memory for Real-Time Embedded Systems. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, nov. 2015. v. 23, n. 11, p. 2700–2704.

WATERMAN, A. *et al.* The risc-v instruction set manual, volume i: User-level isa. **CS Division, EECE Department, University of California, Berkeley**, 2014.

WEISZ, G.; HOE, J. C. CoRAM++: Supporting data-structure-specific memory interfaces for FPGA computing. In: INTERNATIONAL CONFERENCE ON FIELD PROGRAMMABLE LOGIC AND APPLICATIONS, 2015. **Proceedings.** IEEE, 2015. p. 1–8.

WHITHAM, J.; AUDSLEY, N. Using trace scratchpads to reduce execution times in predictable real-time architectures. In: REAL-TIME AND EMBEDDED TECHNOLOGY AND APPLICATIONS SYMPOSIUM, 2008. **Proceedings.** IEEE, 2008. p. 305–316.

WINTERSTEIN, F. *et al.* MATCHUP: memory abstractions for heap manipulating programs. In: INTERNATIONAL SYMPOSIUM ON FIELD-PROGRAMMABLE GATE ARRAYS, 2015. **Proceedings.** ACM, 2015. p. 136–145.

WOLFE, M. **The OpenACC application programming interface**. Version.

WONG, S.; AS, T. VAN; BROWN, G. ρ-VEX: A reconfigurable and extensible softcore VLIW processor. In: INTERNATIONAL CONFERENCE ON FIELD-PROGRAMMABLE TECHNOLOGY, 2008. **Proceedings.** IEEE, 2008. p. 369–372.

WULF, W. A.; MCKEE, S. A. Hitting the Memory Wall: Implications of the Obvious. New York, NY, USA: **SIGARCH Comput. Archit. News**, mar. 1995. v. 23, n. 1, p. 20–24.

YANG, H.-J. *et al.* Scavenger: Automating the construction of application-optimized memory hierarchies. In: INTERNATIONAL SYMPOSIUM ON FIELD PROGRAMMABLE LOGIC AND APPLICATIONS, 2015. **Proceedings.** IEEE, 2015. p. 1–8.