UNIVERSIDADE DE LISBOA
INSTITUTO SUPERIOR TÉCNICO

Energy-Efficient Computing: Adaptive Structures and Data Management

Nuno Filipe Simões Santos Moraes Neves

Supervisor: Doctor Nuno Filipe Valentim Roma
Co-Supervisor: Doctor Pedro Filipe Zeferino Aidos Tomás

Thesis approved in public session to obtain the PhD Degree in
Electrical and Computer Engineering

Jury final classification: Pass with Distinction

2019

UNIVERSIDADE DE LISBOA
INSTITUTO SUPERIOR TÉCNICO

Energy-Efficient Computing: Adaptive Structures and Data Management

Nuno Filipe Simões Santos Moraes Neves

Supervisor: Doctor Nuno Filipe Valentim Roma
Co-Supervisor: Doctor Pedro Filipe Zeferino Aidos Tomás

Thesis approved in public session to obtain the PhD Degree in
Electrical and Computer Engineering

Jury final classification: Pass with Distinction

Jury

Chairperson: Doctor Isabel Maria Martins Trancoso, Instituto Superior Técnico, Universidade de Lisboa

Members of the Committee:

Doctor João Manuel Paiva Cardoso, Faculdade de Engenharia, Universidade do Porto
Doctor Horácio Cláudio de Campos Neto, Instituto Superior Técnico, Universidade de Lisboa
Doctor Arnaldo Silva Rodrigues de Oliveira, Universidade de Aveiro
Doctor Nuno Filipe Valentim Roma, Instituto Superior Técnico, Universidade de Lisboa
Doctor Ricardo Miguel Ferreira Martins, Instituto Superior Técnico, Universidade de Lisboa

FUNDING INSTITUTION

Fundação para a Ciência e a Tecnologia (FCT)

2019

Acknowledgements

The work presented in this thesis would not be possible without the support of many people and institutions.

Firstly, I would like to thank my Ph.D. supervisors Pedro Tomás and Nuno Roma for the tremendous amount of support they gave to me during all these years. This support does not only cover their invaluable guidance and dedication to the entire work, but also all the personal aspects by teaching me how to become a better researcher and constantly challenging me and pushing me out of my comfort zone. Thank you for your trust, your patience, and all the opportunities you gave (and are still giving) me over all these years.

Furthermore, I would like to thank Fundação para a Ciência e a Tecnologia for the financial support under the Ph.D. grant SFRH/BD/100697/2014, project UID/CEC/50021/2019 and project HAnDLE (ref. PTDC/EEI-HAC/30485/2017). I would also like to thank INESC-ID for hosting me, for providing all the conditions necessary for performing my work in the best way, and for the financial support required for the publications and international conferences.

To all the members of the SiPS group for the solidarity and teamwork, as well as to Ana, Aurélia and Elisabete for the provided administrative support. I would like to thank all my closest colleagues João, Nuno, Paulo, Pedro, Roger, Tiago, and Professor Leonel, for their partnership, collaboration and all the good times throughout the years.

To all the co-authors of my publications and people I had the pleasure to work with, especially to Adrien Mussio, Fabien Gonçalves, Henrique Mendes, Nuno Horta, Ricardo Chaves, and Rui Neves. Thank you for your help, recommendations and provided scientific background.

To all my friends, namely Duarte, Gonçalo, João, Pedro and Sónia for their friendship and support from as far back as I can remember.

To all my family and relatives for the incentive they provided during these five years, especially to my mother, Maria de Lurdes, and my father, José.

To Joana for all the support, patience and comprehension during the most difficult times of this journey, for the friendship and for the unconditional love.

Finally, to all of those whom I do not mention here but that directly or indirectly have contributed and helped me to achieve my goals.

Abstract

Computing efficiency is often regarded as the most challenging goal to achieve cost-effective exascale computing. While the computing market has driven the research for heterogeneous and specialized many-core architectures, several issues must still be addressed to attain the efficiency goals established by the High-Performance Computing (HPC) community. In particular, the processing performance of current computing systems is often conditioned by power constraints and thermal dissipation, forcing most of the chip area to be dimmed or powered down during long periods of time. This issue limits the achievable computing performance, by making the utilization of the available chip area ineffective. Furthermore, most of the research has been focused on the performance-energy efficiency of new processing infrastructures, with the underlying data communication subsystems relying on conventional multi-level cache structures, to mitigate the data access latencies. However, these structures struggle when the application dataset is very large and does not fit in the cache memory, or in the presence of complex memory access patterns, where data-locality cannot be efficiently exploited. This results in costly contention issues that tend to degrade the data transfer throughput and, in turn, limit the system's achievable performance. To tackle all these issues, this dissertation proposes new adaptive computing mechanisms to cope with the efficiency demands of the next generations of computing systems. The research is initially focused on compile-time memory access pattern analysis and code transformations, to enable data streaming mechanisms in conventional cache-based infrastructures. Such mechanisms allow the exploitation of hybrid data communication schemes, targeting runtime data movement adaptation and memory throughput maximization. The devised mechanisms showed to reduce data transfer overheads with efficient data fetching, reutilization, and management techniques, in turn countering the contention that is usually observed in shared interconnections. The same application analysis principle is also exploited for adaptable processing acceleration, through runtime hardware adaptation. It is explored the viability of using partial reconfiguration to balance the raw performance with the corresponding energy consumption, according to the runtime context. The resulting adaptable processing framework allows for significant performance gains and energy consumption reductions, resulting in increased computing efficiency.

Keywords

Computing Efficiency, Compiler-Time Analysis, Data Streaming, Adaptable Data Communication, Reconfigurable Computing

Resumo

A eficiência da computação é muitas vezes reconhecida como o objetivo mais desafiante para se alcançar uma capacidade de computação em exa-escala. Para atingir esse objetivo, têm sido desenvolvidas inúmeras arquiteturas multiprocessador heterogéneas, com diversos níveis de especialização. No entanto, existem ainda várias questões que terão de ser convenientemente abordadas para alcançar as metas de eficiência estabelecidas pela comunidade de computação de elevado desempenho. Em particular, o desempenho dos sistemas de computação atuais é condicionado por restrições de energia e dissipação térmica, forçando a maior parte dos recursos do sistema a estarem desligados durante uma parte significativa do tempo. Este problema torna a utilização dos recursos disponibilizados pelo sistema pouco eficiente, limitando, assim, o desempenho computacional que é realmente atingível. Além disso, a maior parte da investigação feita nos últimos anos tem sido principalmente focada em novas infraestruturas de processamento, e no seu desempenho e eficiência energética. Em compensação, os subsistemas de comunicação de dados são geralmente suportados em estruturas de cache convencionais, de forma a mitigar a latência de acesso aos dados. No entanto, estas estruturas continuam a não conseguir lidar com conjuntos de dados com dimensão superior à capacidade da memória cache, nem com padrões complexos de acesso à memória, em que a localidade de dados não é eficientemente explorada. Isto resulta em problemas de contenção que diminuem a taxa de transferência de dados, e, por sua vez, limitam o desempenho do sistema. De modo a abordar estes problemas, esta dissertação propõe novos mecanismos de computação adaptativos para alcançar os níveis de eficiência necessários para as próximas gerações de sistemas de computação. A investigação é inicialmente focada na análise de padrões de acesso à memória em tempo de compilação e em transformações de código, de modo a criar um suporte para a utilização de mecanismos de manipulação de cadeias de dados em infraestruturas de cache convencionais. Os mecanismos desenvolvidos permitem a exploração de esquemas híbridos de comunicação, permitindo uma adaptação ao nível do esquema de transferência de dados em tempo de execução e uma maximização da largura de banda da memória. Os mecanismos propostos mostraram ser capazes de mitigar penalizações associadas à transferência de dados tirando partido de técnicas eficientes de aquisição, reutilização e gestão de dados, contrariando, por sua vez, os problemas de contenção que são normalmente observados em barramentos de comunicação partilhados.

O mesmo princípio de análise de aplicações é também explorado no contexto da aceleração do processamento, através da adaptação da arquitetura em tempo de execução. Para o efeito, é explorada a viabilidade da utilização de mecanismos de reconfiguração parcial para balancear o desempenho do sistema com o seu consumo de energia, de acordo com o contexto de execução. As estruturas de processamento adaptáveis propostas permitem atingir ganhos de desempenho significativos e consequentes reduções de consumo de energia, resultando, assim, numa eficiência de computação elevada.

Palavras-Chave

Computação Eficiente, Análise em Tempo de Compilação, Manipulação de Cadeias de Dados, Comunicação de Dados Adaptativa, Computação Reconfigurável

Contents

1 Introduction 1

1.1 Motivation and Objectives . . . 3

1.2 Contributions . . . 5

1.3 Outline . . . 7

2 Background and State-of-the-Art 9

2.1 Overview of Modern Computing Systems . . . 10

2.2 Data Communication Schemes and Paradigms . . . 13

2.2.1 Compiler Static Analysis and Memory Access Optimization . . . 14

2.2.2 Data Prefetching Techniques . . . 16

2.2.3 Data Streaming Architectures . . . 20

2.2.4 Discussion . . . 22

2.3 Adaptive Computing . . . 23

2.3.1 Power Supply Management . . . 24

2.3.2 Dynamically Reconfigurable Systems . . . 24

2.3.3 Dedicated Programming Frameworks and Execution Optimization . . . 28

2.3.4 Reconfigurable Communication Systems . . . 30

2.3.5 Discussion . . . 31

2.4 Summary . . . 32

3 Data-Pattern Analysis and Stream Transformations 33

3.1 Modeling of Complex Data-Patterns . . . 35

3.1.1 Affine Mathematical Model for Data Indexing . . . 37

3.1.2 Memory Access Description Specification . . . 39

3.2 Compile-Time Stream Code Generation . . . 42

3.2.1 Compiler Module Overview . . . 43

3.2.2 Context Representation Language . . . 46

3.2.3 Stream Code Generation . . . 49

3.3 Preliminary Experimental Evaluation . . . 53

3.3.1 Methodology . . . 53

3.3.2 Data-Pattern Encoding Efficiency 56
3.3.3 Source Code Reduction Evaluation 58
3.3.4 Discussion 59
3.4 Summary 60

4 Data Stream Communication 61
4.1 Data Stream Generation 63
4.1.1 Data Stream Controller Architecture 63
4.1.2 Streaming Infrastructure Interface and Programming 67
4.2 Data Stream Prefetching 67
4.2.1 Case Study A: Stream Prefetching on GPGPUs 68
4.2.2 Case Study B: Data Streaming on Modern General-Purpose CPUs 75
4.2.3 Discussion 83
4.3 In-Cache Stream Communication Paradigm 84
4.3.1 In-Cache Stream Controller 85
4.3.2 Communication Infrastructure and Protocol 89
4.3.3 Memory-Aware Data Stream Generation 91
4.4 In-Cache Streaming Evaluation 93
4.4.1 Methodology 94
4.4.2 Hardware Resources Overhead 96
4.4.3 Stream Generation Efficiency and Main Memory Throughput 98
4.4.4 Prototype Evaluation 99
4.5 Summary 103

5 Adaptive Processing Structures 105
5.1 Reconfigurable Many-Core Accelerator 107
5.1.1 Many-Core Processing Infrastructure 108
5.1.2 Data Access and Interconnection Networks 109
5.1.3 Hypervisor and Accelerator Management 110
5.1.4 Reconfiguration Engine 111
5.1.5 Implementation Considerations and Constraints 113
5.1.6 Accelerator Configuration and Implementation 113
5.2 Reconfiguration Management 116
5.2.1 Runtime Learning and Reconfiguration Policies 117
5.2.2 Compile-time Modeling and Optimization 119
5.3 Experimental Validation 129
5.3.1 Methodology 129
5.3.2 Case Study A: Learning-based Automatic Reconfiguration 131

5.3.3 Case Study B: Compiler-Assisted Reconfiguration 135
5.3.4 Case Study C: Resource Management for Dynamic Workloads 138
5.4 Summary 143

6 Conclusions and Future Work 145
6.1 Conclusions 146
6.2 Main Contributions 147
6.3 Future Work 148

Bibliography 151

Publications 165

List of Figures

2.1 Examples of current multi- and many-core system organizations 10
2.2 GRAPHITE for GCC 14
2.3 Polly for LLVM IR 15
2.4 Common spatial prefetching mechanisms. 16
2.5 AMPM Prefetcher 17
2.6 VLDP Prefetcher 18
2.7 Offset Prefetchers 19
2.8 APMC Stream Controller 21
2.9 HotStream Framework DFC 22
2.10 Plasticine CGRA compute unit 26
2.11 ReMAP Architecture 27

3.1 Stream controller topologies 34
3.2 Example affine representation 38
3.3 Example affine representation of indirect memory accesses 39
3.4 Context Descriptor specification 40
3.5 *Access Descriptor* encoding for a triangular matrix access 41
3.6 *Access Descriptor* encoding for indirect memory indexation 42
3.7 Clang AST example 43
3.8 Overview of the compilation tool 44
3.9 Compilation tool translations and code transformations 45
3.10 Context Representation Language reference 46
3.11 Example of an AST translation to CRL 49
3.12 CRL to descriptor representation translation 50
3.13 Indirection representation and descriptor encoding 51
3.14 Stream code transformation illustration 52
3.15 Considered synthetic data-patterns 57
3.16 Context descriptor efficiency 58

4.1 Data Stream Controller architecture. 64

4.2 GPU architecture and memory hierarchy overview. 69

4.3 Application profiles 70

4.4 Stream prefetching mechanism integration 71

4.5 GPU benchmark results 74

4.6 CPU integration of the proposed DSC 75

4.7 L1 hit rate comparison 80

4.8 IPC comparison in absolute values. 80

4.9 Overall speedup comparison 81

4.10 Speedup breakdown in data streaming and code reduction 81

4.11 In-cache stream communication infrastructure overview 84

4.12 Cache memory hybridization 86

4.13 In-cache stream controller architecture 87

4.14 Message-passing protocol 89

4.15 Memory access optimization illustration 92

4.16 Reorder buffer architecture 93

4.17 Main memory throughput and memory access latency 99

4.18 Comparison of the `ICS-SMC` with the `BASE`, AMPM and `uICS` setups 100

5.1 Reconfigurable many-core accelerator overview 107

5.2 Internal block diagram of a processing cluster 109

5.3 Accelerator Controller 111

5.4 Reconfiguration engine architecture 112

5.5 Execution plan example 120

5.6 Example of a system specification 121

5.7 Example of an architecture model 122

5.8 NSGA-II extension 123

5.9 Individual encoding 126

5.10 DSE operators 127

5.11 Example of an execution queue 129

5.12 Real-time adaptation of the processing architecture A 132

5.13 Real-time adaptation of the processing architecture B 132

5.14 System real-time adaptation - Maximum performance 133

5.15 System real-time adaptation - Power ceiling 134

5.16 System real-time adaptation - Minimum performance 135

5.17 Arithmetic benchmark application model 136

5.18 Arithmetic benchmark architecture configuration 136

5.19 Objective function interaction in the Pareto front 138

5.20 Execution-time optimization plan 139

5.21 Low peak power dissipation optimization plan 139
5.22 Biological sequence alignment system model 141
5.23 Relaxed execution plan 143
5.24 Intensive execution plan 143

List of Tables

3.1 Considered standard benchmarks and kernels 55
3.2 Characterization of the considered synthetic kernels 57

4.1 GPGPU-Sim configuration and benchmark set 73
4.2 Adopted x86-based CPU configuration. 77
4.3 Considered benchmarks and kernels for the evaluation setup. 78
4.4 Configurations for the reference prefetching and proposed DSC setups. 79
4.5 Benchmark applications and datasets 96
4.6 Resource usage 97
4.7 Address generation rate and descriptor size 97

5.1 Characterization of each type of core 115
5.2 Characterization of each cluster type 115
5.3 Characterization of the accelerator’s supporting infrastructure 115
5.4 Execution time and energy results 135
5.5 Arithmetic kernel benchmark characterization 137
5.6 Biological sequence alignment case study 141

List of Algorithms

1 Execution Time Optimization 118

2 Power-Ceiling Algorithm 118

3 Minimum Assured Performance Algorithm 119

4 Mapping Validation/Repair Operator 127

5 Execution Plan Definition Operator 128

List of Acronyms

AGU	Address Generation Unit
ALU	Arithmetic and Logic Unit
AMPM	Access Map Pattern Matching
APMC	Advanced Pattern-based Memory Controller
ASIC	Application Specific Integrated Circuit
ASIP	Application Specific Instruction-set Processor
AST	Abstract Syntax Tree
CCPO	Clock Cycles per Operation
CGRA	Coarse-Grained Reconfigurable Array
CLB	Configurable Logic Block
CPU	Central Processing Unit
CRL	Context Representation Language
CTA	Cooperative Thread Array
DFC	Data-Fetch Controller
DMA	Direct Memory Access
DSC	Data Stream Controller
DSE	Design Space Exploration
DSP	Digital Signal Processor
DVFS	Dynamic Voltage and Frequency Scaling
EA	Evolutionary Algorithm
EDP	Energy-Delay Product

FIFO	First-In First-Out
FP	Floating-Point
FPGA	Field-Programmable Gate Array
FPU	Floating-Point Unit
FSM	Finite-State Machine
GCC	GNU Compiler Collection
GHB	Global History Buffer
GPP	General Purpose Processor
GPU	Graphics Processing Unit
HPC	High-Performance Computing
IC	Integrated Circuit
ICAP	Internal Configuration Access Port
ICS	In-Cache Stream
ISA	Instruction Set Architecture
ISB	Irregular Stream Buffer
IoT	Internet-of-Things
IPC	instructions-per-cycle
IR	Intermediate Representation
LRU	Least Recently Used
LUT	Look-Up Table
MCU	Modifier Chain Unit
MOEA	Multi-Objective Evolutionary Algorithm
MOO	Multi-Objective Optimization
MSHR	Miss Status Hold Register
NoC	Network on Chip
NSGA-II	Non-dominated Sorting Genetic Algorithm II

NVM	Non-Volatile Memory
OPS	operations-per-second
PARE	Power-Aware pRefetch Engine
PC	Program Counter
PE	Processing Element
RISC	Reduced Instruction Set Computer
SIMD	Single-Instruction Multiple-Data
SIMT	Single-Instruction Multiple-Thread
SM	Streaming Multiprocessor
SMC	Stream Management Controller
SoC	System-On-Chip
TLB	Translation Lookaside Buffer
VLDP	Variable Length Delta Prefetcher
VLIW	Very Long Instruction Word

1

Introduction

Contents

1.1 Motivation and Objectives 3
1.2 Contributions 5
1.3 Outline 7

The ever-increasing demand for computational processing power has reached a critical point in the past decade since the straightforward scaling of current technology is no longer viable [1]. Moreover, driven by the Internet-of-Things (IoT) and Cloud Computing era, an ever-increasing number of computing platforms, ranging from battery-powered mobile devices to supercomputing clusters, has been hitting the limits of performance and energy efficiency. In fact, computing efficiency has been recognized as the most challenging goal to achieve cost-effective *exascale computing* by 2020 (as established by the High-Performance Computing (HPC) community) [2]. Accordingly, the scientific community recognizes that the research on new node/chip-level architectures, circuitry management mechanisms and memory packaging technologies, represents promising strategies that can still contribute an estimated 5x improvement in performance and energy efficiency [3].

To attain such a goal, the computing market has been pushing the research on alternative heterogeneous and high-performance many-core processing systems, that combine multiple architectures (with different degrees of parallelism and specialization) to improve the system's processing efficiency. By deploying resource and power management mechanisms, they can balance raw performance and energy consumption according to the aimed execution context. However, processing systems are still struggling to provide the required levels of computing efficiency and reach the *exascale* goal. In fact, their attainable performance is still limited by energy consumption constraints and by the adverse impact of stalls resulting from long memory access latencies.

Specifically, computing throughput is still limited in many computing systems by power supply management mechanisms that turn off parts of the device to save energy, effectively reducing the amount of available processing resources. Moreover, their general purpose nature imposes a processing throughput limit, caused by a gap between flexibility and specialization to the application processing requirements. On the other hand, most systems are still relying on conventional cache-based memory subsystems. However, although they have been widely exploited to reduce the impact of long memory access latencies, the limitations and contention issues of cache structures are well-known [4], as their performance is bound by the characteristics of the application data access pattern.

Accordingly, knowing an application's characteristics (e.g., type of computing operations, the degree of parallelism or memory access patterns) can provide the necessary insights to make its execution as efficient as possible. Such a premise has long been exploited by compilation tools to optimize machine code according to the features of the target architecture [5–10] and by power management and performance throttling mechanisms to reduce energy consumptions [11–15]. It is also the basis for memory access optimization mechanisms (such as data prefetchers [16–24]) that analyze the data indexing behavior of a running application to predict future accesses. However, such solutions are mostly used to mitigate execution and energy overheads or costly data operations and are bound by the general purpose computing nature of off-the-shelf systems.

On the other hand, while application-specific approaches can be tailored according to an application's characteristics to provide the necessary computing efficiency, their inherent loss in flexibility makes them unsuited for the diverse HPC market.

One viable approach to bridge this performance-efficiency gap is the introduction of runtime adaptation in general purpose multi/many-core infrastructures. Specifically, by performing analyses of the application at compile-time and/or at runtime, it is possible to extract its computational requirements and data access characteristics and adapt its execution by: *i)* performing independent data acquisition operations according to its memory access pattern; *ii)* adapt the data communication scheme between the system's processing cores and the main memory; *iii)* perform hardware adaptation of the cores according to the type of executed operations; and *iv)* manage the type and amount of active processing resources according to the system's execution context.

Accordingly, by demarking themselves from off-the-shelf approaches that solely try to hide or mitigate known performance limitations in conventional systems, alternative solutions can be designed that exploit the adaptation of a processing system and modify its infrastructure to meet the running application requirements, without eliminating the system's general purpose computing capabilities. Such solutions can be attained by the combination of compile-time application analysis and optimization tools and the introduction of dynamically reconfigurable architectures and adaptable data communication structures in conventional processing systems. Such structures can also be paired with intelligent execution management modules that leverage the information provided by the compilation tools, with the goal of increasing the system's processing throughput and reducing its energy consumption, in turn increasing the overall computing efficiency.

1.1 Motivation and Objectives

Embracing the energy-efficiency constraints and the current HPC demands, multi-core heterogeneous systems are usually composed of differently balanced architectures to maintain higher levels of performance available while providing low-power execution contexts [12, 15]. Despite being now widely deployed in mobile platforms, such systems still rely on mechanisms based on task migration and aggressive voltage scaling and power-gating. As such, they may require that some (usually most) of the transistors remain dimmed or powered down most of the time, due to the observed divergence between device-level energy efficiency and transistor density [1], in turn pushing the dark silicon problem in current computing systems. Such a problem adds an inefficient utilization of the available processing resources on top of the overheads that are also imposed by task migration mechanisms.

Conversely, application-specific accelerators [25–28] have shown to be capable of leading to high application acceleration with low power supply demands. Such an efficiency often results from the significantly lower architectural footprints and low power dissipation characteristics of such structures, when compared to conventional processing systems. However, the provided specialization often incurs in a loss in general purpose computing capabilities. Moreover, each system must be tailored to a particular application domain and applications must be carefully dimensioned to exploit as much performance-energy efficiency as possible [28].

However, the actual throughput that is offered by modern computing systems is currently limited by the power/performance impact of data transfers and general data indexing in the memory subsystem. This is a result of the standard adoption of conventional local (and often multi-level) cache structures to avoid high memory access latencies. Although they can significantly reduce latency in the presence of computationally intensive applications, they struggle when the application is bound by the characteristics of its memory accesses. Hierarchical cache structures especially struggle when the application dataset is large and does not fit in one or more cache levels, or in the presence of complex memory access patterns, where data-locality cannot be efficiently exploited.

Typically, the first steps to mitigate the impact of long memory access latencies are performed at compile-time. In particular, compilers have long utilized specialized tools to reorganize an application's sequence of instructions and hide memory access latency behind computation [29, 30]. The deployed algorithms statically analyze the original code and optimize it, with the goal of bridging the gap between application code characteristics and the capabilities of the target processing architecture and memory subsystem [5–10].

By following a similar principle, data prefetching techniques [16–19, 19–24, 31–33] are designed to deal with the intrinsic characteristics of the memory access patterns at runtime. In particular, they analyze an application's sequence of memory requests to predict future accesses and obtain data ahead of time, hence mitigating the impact of long memory access latencies. However, despite their throughput improvements, the incremental gains provided by each new generation of prefetchers are becoming limited. This is mostly because prediction inaccuracies can still occur when dealing with complex data patterns, which affect the memory access coverage. On the other hand, they must rely on costly runtime memory access monitoring, which can result in added time-consuming penalties.

Furthermore, to achieve higher prediction accuracies, such data prefetching schemes not only require larger amounts of hardware resources to implement the predictive algorithm but also added memory space to trace recent accesses [20, 22–24]. Moreover, when applied to processing systems with large numbers of physical cores, the prefetching structures can suffer from the same increased contention that occurs with conventional cache-based systems, resulting from a flood of prefetching requests in the memory subsystem [16].

Accordingly, by considering that application-specific architectures have limited use in a general purpose HPC context, **adaptive computing systems** (exploiting hardware reconfiguration and dynamic data communication schemes) represent a promising approach to provide the necessary mechanisms to bridge that gap and to push the limits of **computing efficiency**. Naturally, the viability of such adaptive systems implies that the running application is thoroughly analyzed by compiler tools that can extract information regarding its computational requirements (including, but not limited to, types of processing operations, instruction and data parallelism, data communication schemes and memory access patterns) and provide it to the system's processing infrastructure, so it can adapt itself to the execution context and improve its overall performance and energy efficiency.

Acknowledging that conventional processing systems and memory organizations are hardly coping with the throughput demands of current applications and with the challenges recognized by the HPC community, the main objective of this thesis is the investigation of new adaptive computing techniques and mechanisms (acting both at compile-time and runtime) to provide the means to improve the performance and energy efficiency of future generations of computing systems.

1.2 Contributions

To attain the described objectives, the presented research is initially focused on compile-time memory access pattern analysis and encoding, combined with code transformations, to enable data stream communication in conventional cache-based infrastructures. Such mechanisms allow the exploitation of new data communication and stream prefetching schemes, resulting in runtime data movement adaptation and memory throughput maximization. The same principle of compile-time application analysis and optimization is also explored for reconfigurable processing acceleration through a comprehensive study on runtime architecture adaptation. In the particular context of Field-Programmable Gate Array (FPGA) implementation technologies, it is explored the viability of using partial reconfiguration to increase the processing efficiency. This mechanism is used as an alternative to voltage/frequency scaling and power/clock-gating mechanisms to deploy throughput-energy balancing mechanisms to improve overall computing efficiency.

Accordingly, the contributions of this Thesis can be summarized in the following paragraphs, together with the enumeration of the resulting publications in peer-reviewed journals and conferences where these contributions were made available to the scientific community:

- A memory access description specification, based on a multi-level affine model that allows the encoding of data dependencies between accesses. This specification is capable of efficiently encoding memory access patterns through improved description flexibility and efficiency, providing support for arbitrarily complex deterministic access patterns and indirect memory accesses;
    - N. Neves, P. Tomás, and N. Roma, "Efficient data-stream management for shared-memory many-core systems," in 25th International Conference on Field Programmable Logic and Applications (FPL). IEEE, 2015, pp. 508–515

  - N. Neves, P. Tomás, and N. Roma, “Adaptive in-cache streaming for efficient data management,” IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 25, no. 7, pp. 2130–2143, March 2017
  - N. Neves, P. Tomás, and N. Roma, “Stream data prefetcher for the gpu memory interface,” The Journal of Supercomputing, vol. 74, no. 76, pp. 2314–2328, June 2018

- A compile-time static analysis tool to identify, describe and encode the application’s memory access pattern using the proposed representation. As a result of this data access encoding, the explicit representation of the memory access pattern makes the corresponding data indexation and address calculation redundant and unnecessary. Accordingly, the compilation tool also performs a code transformation pass that replaces the array subscript indexation of each encoded data load with a stream reference (represented by a pointer), ultimately resulting in a reduced number of instructions and accelerating the execution of the code;
- A new In-Cache Stream (ICS) communication model supporting both *memory-addressed* and *packed-stream* data accesses, as well as adaptive mixed *memory-addressed*/*packed-stream* accesses, specially suited for applications composed of compile-time predictable (described as a set of streams) and non-predictable memory access patterns. The physical implementation of the ICS model also deploys several efficient memory access optimization techniques, such as bandwidth optimization, data reorganization and reutilization, and in-time stream manipulation;
  - N. Neves, A. Mussio, F. Gonçalves, P. Tomás, and N. Roma, “In-cache streaming: Morphable infrastructure for many-core processing systems,” in Euro-Par 2016: Parallel Processing Workshops. Springer International Publishing, 2017, pp. 775–787
  - N. Neves, P. Tomás, and N. Roma, “Adaptive in-cache streaming for efficient data management,” IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 25, no. 7, pp. 2130–2143, March 2017
- A reconfigurable hundred-core heterogeneous architecture, capable of adapting its processing characteristics according to runtime application requirements. This architecture is particularly suited to take advantage of the partial dynamic reconfiguration capabilities of modern FPGA devices. The system is capable of autonomously determining and reconfiguring each processing core to the most suitable architecture through a set of runtime optimization policies to balance the number and micro-architecture of the instantiated cores, according to instantaneous application and system requirements and constraints;
  - N. Neves, H. Mendes, R. J. Chaves, P. Tomás, and N. Roma, “Morphable hundred-core heterogeneous architecture for energy-aware computation,” IET Computers & Digital Techniques, vol. 9, no. 1, pp. 49–62, 2015
  - N. Neves, P. Tomás, and N. Roma, “Host to accelerator interfacing framework for high-throughput co-processing systems,” in XI Jornadas sobre Sistemas Reconfiguráveis

(REC), 2015, pp. 31–38

- A compile-time analysis and optimization tool for efficient scheduling and mapping of processing tasks into reconfigurable architectures. The conceived tool is based on a combination of Multi-Objective Optimization (MOO) and Design Space Exploration (DSE) techniques and is capable of deriving multiple sets of adaptive task mapping plans, which establish different compromises between the application's performance, system power consumption, and energy efficiency.
  - N. Neves, R. Neves, N. Horta, P. Tomás, and N. Roma, "Multi-objective kernel mapping and scheduling for morphable many-core architectures," Expert Systems with Applications, vol. 45, pp. 385–399, 2016

1.3 Outline

The dissertation is organized in the following chapters. After this introductory part, Chapter 2 presents a revision of the current state-of-the-art and technical background in the research domains related to application analysis, data prefetching, data-stream and adaptive communication techniques, and reconfigurable processing systems. The new memory access description specification is detailed in Chapter 3, together with the conceived static analysis and code transformation tools. Chapter 4 describes the proposed data streaming mechanisms and their implementation and evaluation in a Graphics Processing Unit (GPU), in a General Purpose Processor (GPP) and an FPGA accelerator, together with comprehensive comparisons with state-of-the-art solutions. Chapter 5 presents a study on dynamically adaptable processing architectures and describes the implementation of a fully functional reconfigurable accelerator prototype. The study is complemented with the implementation of a DSE compile-time optimization tool to support the task scheduling and mapping into reconfigurable hardware. Finally, a discussion of the achieved contributions is presented in Chapter 6, together with the enumeration of some possible future research guidelines.

2

Background and State-of-the-Art

Contents

2.1 Overview of Modern Computing Systems 10

2.2 Data Communication Schemes and Paradigms 13

2.2.1 Compiler Static Analysis and Memory Access Optimization 14

2.2.2 Data Prefetching Techniques 16

2.2.3 Data Streaming Architectures 20

2.2.4 Discussion 22

2.3 Adaptive Computing 23

2.3.1 Power Supply Management 24

2.3.2 Dynamically Reconfigurable Systems 24

2.3.3 Dedicated Programming Frameworks and Execution Optimization 28

2.3.4 Reconfigurable Communication Systems 30

2.3.5 Discussion 31

2.4 Summary 32

This chapter provides a brief overview of the currently prominent computing system architectures, followed by a review of the most important state-of-the-art approaches in the areas of reconfigurable computing architectures and data communication techniques and infrastructures.

2.1 Overview of Modern Computing Systems

Since the decay of the single-core General Purpose Processor (GPP) era that chip power consumption has been regarded as one of the most pressing concerns in the development of modern High-Performance Computing (HPC) architectures. Acknowledging that raw processing power inevitably leads to high energy consumption, early solutions relied on the complexity reduction of general-purpose processing architectures and their replication in multi-core infrastructures (depicted in Fig. 2.1.A), allied with voltage and operating frequency management techniques (e.g., power-gating or Dynamic Voltage and Frequency Scaling (DVFS)). However, such a lower structural complexity led to reduced processing capabilities per core, thus limiting the achievable core performance as a trade-off for lower energy consumptions.

Accordingly, several current solutions try to surpass this per-core lower performance with increased levels of parallelism. This is often done by deploying infrastructures with large numbers of processing cores, either on the same chip (such as Graphics Processing Units (GPUs), as depicted in Fig.2.1.D) or by interconnecting several processors (such as data-center clusters). However, the increased number of processing units incurs, by itself, on added energy consumption in the whole system. This is due to the amount of logic required to deploy and control massively parallel architectures and the underlying data communication infrastructures.

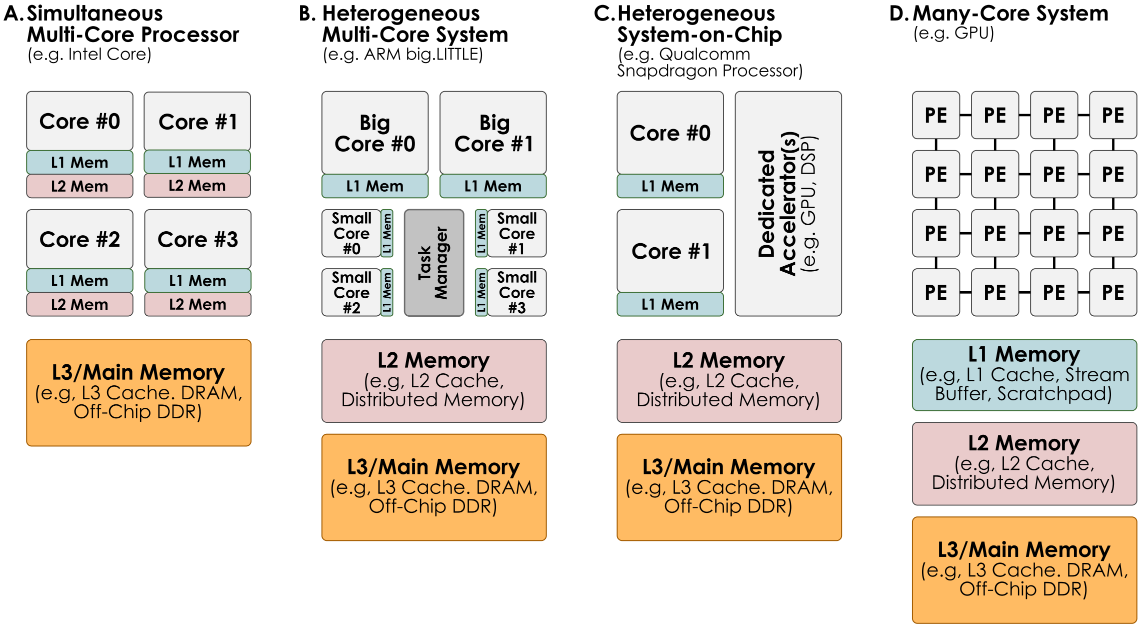

A. Simultaneous Multi-Core Processor
(e.g. Intel Core)
Core #0
Core #1
Core #2
Core #3
L1 Mem
L2 Mem
L3/Main Memory
(e.g, L3 Cache. DRAM, Off-Chip DDR)
B. Heterogeneous Multi-Core System
(e.g. ARM big.LITTLE)
Big Core #0
Big Core #1
L1 Mem
Small Core #0
Small Core #1
Small Core #2
Small Core #3
Task Manager
L2 Memory
(e.g, L2 Cache, Distributed Memory)
L3/Main Memory
(e.g, L3 Cache. DRAM, Off-Chip DDR)
C. Heterogeneous System-on-Chip
(e.g. Qualcomm Snapdragon Processor)
Core #0
Core #1
L1 Mem
Dedicated Accelerator(s)
(e.g. GPU, DSP)
L2 Memory
(e.g, L2 Cache, Distributed Memory)
L3/Main Memory
(e.g, L3 Cache. DRAM, Off-Chip DDR)
D. Many-Core System
(e.g. GPU)
PE
L1 Memory
(e.g, L1 Cache, Stream Buffer, Scratchpad)
L2 Memory
(e.g, L2 Cache, Distributed Memory)
L3/Main Memory
(e.g, L3 Cache. DRAM, Off-Chip DDR)

Figure 2.1: Examples of current multi- and many-core system organizations.

To circumvent this issue, instead of massively parallelizing architectures to increase raw performance, heterogeneous systems with different co-existing architectures have been regarded as viable solutions. Such systems are often based on combinations of both high-performance and low-power general-purpose architectures or on the deployment of System-On-Chips (SoCs) combining several different dedicated architectures on a single chip, as depicted in Figs. 2.1.B and 2.1.C. While the first approach allows a precise balance between processing performance and energy consumption, the latter acknowledges that specialized architectures lead to straightforward application accelerations (with low power dissipation) [25]. This results in significantly lower energy consumptions, often aided by the lower architectural footprint and power dissipation characteristics of those processing structures. Nevertheless, despite the broad range of dedicated architectures that can be deployed in heterogeneous systems (e.g., GPPs, Digital Signal Processors (DSPs), GPUs, mobile communication or localization modules), such a specialization often incurs in a loss in general-purpose computing capabilities. Moreover, each system must be tailored to a particular application domain, and each task has to be scheduled and migrated to its corresponding dedicated architecture. In fact, since each computing unit is usually best-suited to run a specific type of application, the amount of existing dedicated architectures and parallelism require a careful balance to avoid long module power-down times [1].

The recent reemergence of reconfigurable systems [41–44] has driven the development of alternative mechanisms to deploy dynamic processing specialization while achieving a more efficient resource utilization. Instead of turning off specific processors (or processor components), leaving part of the device unused, reconfigurable architectures reuse the same resource area to provide different dedicated processing structures and schemes to better suit several applications and system constraints. Hence, they can provide the means to make the execution as efficient as possible, both in terms of performance and energy consumption [45–48]. Furthermore, by adopting such structures, it is possible to achieve throughput and efficiency levels similar to those offered by application-specific accelerators [49], without losing general-purpose computing capabilities.

However, although reconfigurable architectures can provide higher levels of processing efficiency, the design of such architectures is usually focused on the processing blocks and their adaptation, often neglecting the power/performance impact of data transfers and general data indexing in the memory subsystem. A standard approach to tackle this problem is to only rely on conventional local (and often multi-level) cache structures (as depicted in Fig. 2.1) to avoid high memory access latencies. However, as the number of cores on a cache-coherent system increases, its contention and overall energy consumption tend to grow. As a result, it can even reach a point where the addition of more cores is no longer useful [4], hence limiting the system's achievable processing throughput. Moreover, such structures struggle when the application dataset is very large and does not fit in the cache, or in the presence of complex memory access patterns, where data-locality cannot be efficiently exploited.

Nevertheless, some adaptive schemes targeting energy efficiency have already been deployed in the communication infrastructure [50–52]. However, they mostly focus on the management of memory resources, to reduce energy consumption through architectural adaptation. Although it has been proved viable for processing infrastructures, the communication infrastructures struggle to efficiently exploit the advantages of adaptive mechanisms (e.g., attained by dynamic reconfiguration or gating mechanisms). This is mostly due to the fact that data communication in many-core infrastructures is already itself constrained in what concerns data transfer latency, mainly resulting from its high contention and request concurrency. Hence, the potential overheads that would result from adapting the communication infrastructure not only would impose additional latency constraints but also would require a careful synchronization between the data transfers and the adaptation process, to avoid data losses and coherency/consistency issues.

Conversely, data prefetching techniques have recently reemerged [18–21] as viable solutions to optimize the sequence of issued memory requests and to mitigate the impact of long memory access latencies. Several methods have been designed to deal with the intrinsic characteristics of the memory access patterns, such as reduced data-locality [21, 31], complex access patterns [17, 19, 32, 33] or large datasets that do not fit in the cache [53]. Moreover, due to their successful memory access improvements, a wide range of device classes has been targeted, from typical multi-core GPP architectures [17] to GPU devices [16]. In fact, this technology has evolved to a point where the main concern is no longer memory access pattern detection and prediction, but instead the timeliness and effectiveness of the prefetching procedure. This led to the emergence of new prefetchers [20, 22–24] that combine multiple hardware modules, with different data fetch granularities and prediction heuristics, across different cache levels.

However, despite the improved throughput, the gains provided by each new generation of prefetchers are becoming limited. In particular, depending on the complexity of the prefetcher, prediction inaccuracies can still occur when dealing with complex data patterns, which affect the memory access coverage and can result in time-consuming penalties. Hence, instead of mitigating memory access latencies, such drawbacks can result in added pressure to the memory subsystem. On the other hand, if a prefetcher deploys more sophisticated techniques to achieve higher prefetching accuracies, it not only utilizes larger amounts of hardware resources to implement the predictive algorithm, but it also requires added memory space to trace recent accesses [20, 22–24]. Furthermore, when applied to processing systems with high levels of parallelism (i.e., with large numbers of physical cores), the prefetching structures tend to require precise synchronization mechanisms and intensity balancing [16]. Otherwise, they can suffer from the same increased contention that occurs with conventional cache-based systems, resulting from a flood of prefetching requests in the memory subsystem.

To circumvent such issues, stream-based communication schemes have been regarded as a viable alternative to cache-based systems and pure-prefetching structures [26, 34, 54]. Instead of having a processing core performing its main memory requests, they are based on the principle that it is possible to exploit the memory access pattern to transparently buffer and transmit (*stream*) the requested data to the corresponding core. Hence, by explicitly decoupling the data indexation/communication and processing infrastructures, data transfers can effectively be hidden behind computation, in turn decreasing the data transfer path and counteracting most of the contention observed in the shared communication structures.

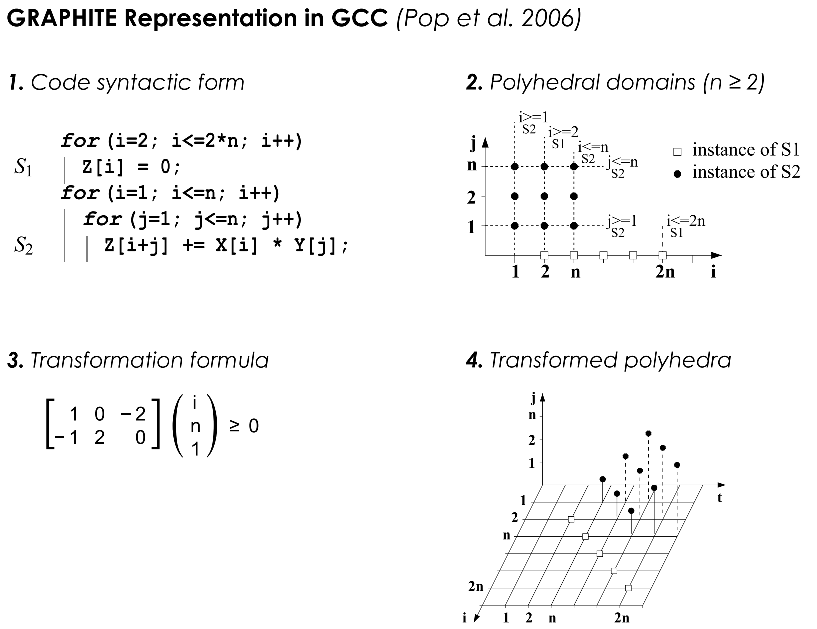
However, due to their application-specific nature, streaming architectures have only been deployed in custom dedicated accelerators [26, 27, 34]. Moreover, they typically require the programmer to manually encode the memory access pattern corresponding to each data stream. Hence, without proper compilation tools, its viability becomes limited for general purpose contexts. Furthermore, since an accurate memory access pattern description is limited to applications with arbitrarily complex, but still deterministic, memory access patterns, purely stream-based infrastructures can hardly deal with certain types of applications. As an example, they struggle in the presence of pointer-based data structures or dynamic indexing procedures, where the memory access is either non-deterministic or generated at runtime. As such, they need to be combined with conventional communication schemes to efficiently handle such cases without relying on costly runtime memory access monitoring [54] and heuristic predictions [18].

2.2 Data Communication Schemes and Paradigms

While performance and efficiency are typically sought by improving the system's processing infrastructure, by making the computing architecture as fast and efficient as possible, the offered throughput is still often limited by the impact of data transfers and general data indexing. This is because the computing architecture in most systems can perform elementary operations much faster than the time that is required to obtain data from the main memory. This data acquisition latency is particularly high when off-chip memory modules are accessed and can lead to considerable performance losses.

Typically, a preliminary mitigation of the impact of costly memory accesses is performed at compile-time, by analyzing the sequence of instructions of an application. According to the target architecture, the compiler performs instruction reorganizations to try to hide the high latency of memory accesses behind other computational instructions, hence promoting instruction-level and data-level parallelism.

On the other hand, at the hardware level, different communication paradigms have been adopted to minimize the power/performance impact of the data management subsystem. In particular, besides straightforward and aggressive data prefetching schemes, often associated with high-energy consuming memory/cache hierarchies, more efficient and sophisticated stream-



GRAPHITE Representation in GCC *(Pop et al. 2006)*

1. *Code syntactic form*

```
    for (i=2; i<=2*n; i++)
S1  | Z[i] = 0;
    for (i=1; i<=n; i++)
    | for (j=1; j<=n; j++)
S2  | | Z[i+j] += X[i] * Y[j];
```

2. *Polyhedral domains (n ≥ 2)*

3. *Transformation formula*

$$\begin{bmatrix}1 & 0 & -2\\ -1 & 2 & 0\end{bmatrix}\begin{pmatrix}i\\ n\\ 1\end{pmatrix} \geq 0$$

4. *Transformed polyhedra*

Figure 2.2: Illustration of the compiler intermediate representation used by GRAPHITE [6] for polyhedral analysis.

based communication systems have been explored. This section presents a review of the most prominent compiler tools and data prefetching and streaming solutions.

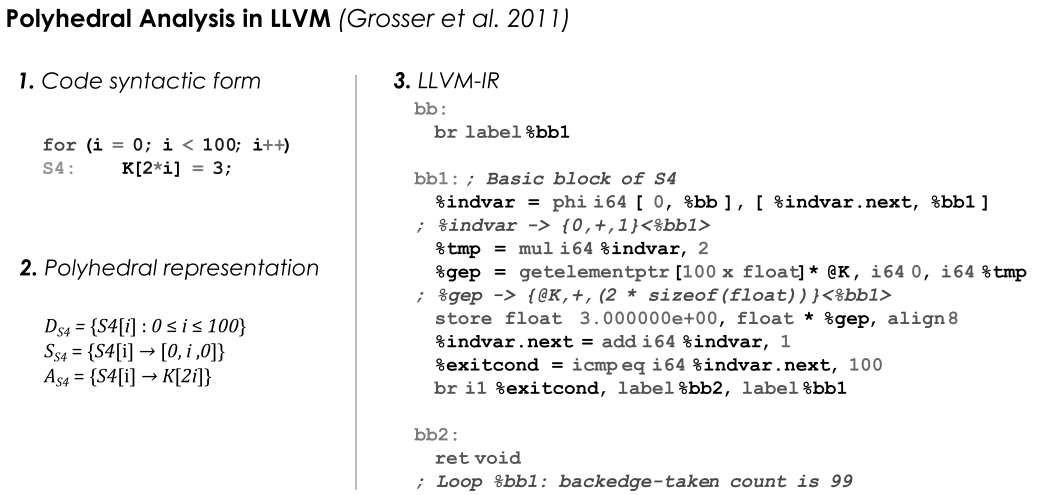
2.2.1 Compiler Static Analysis and Memory Access Optimization

Historically, compilation tools have been used to optimize the sequence of computational operations and hide memory access latency behind computation, bridging the gap between application code characteristics and the capabilities of the target processing architecture. Sophisticated compiler tools often rely on static code analysis [5–7] to extract application information regarding instruction/data dependencies, memory access patterns and/or critical instructions [8]. With this information, these tools subsequently apply code transformations to perform data access reorganization [8, 10] or code optimizations [9], with the goal of increasing instruction-level parallelism to hide the memory access latency. Moreover, the information provided by the compiler is also often used at runtime for assisted execution [8] and/or assisted data acquisition schemes [18, 53].

2.2.1.A Polyhedral Analysis

Compilers have long adopted the polyhedral model to represent nested-loop programs [30]. It is based on the assumption that each loop iteration (within a nested loop) can be described as a lattice point inside mathematical objects called polyhedra (see Fig. 2.2-2). Such a representation allows the compiler to perform affine transformations over the original code and generate optimized loop nests according to an optimization goal (such as vectorization or parallelization).

In an early approach, Pop et al. [6] recognized that polyhedral analysis and loop nest trans-



Polyhedral Analysis in LLVM *(Grosser et al. 2011)*

1. *Code syntactic form*

```
for (i = 0; i < 100; i++)
S4:    K[2*i] = 3;
```

2. *Polyhedral representation*

$D_{S4} = \{S4[i] : 0 \leq i \leq 100\}$
$S_{S4} = \{S4[\mathrm{i}] \rightarrow [0, i, 0]\}$
$A_{S4} = \{S4[\mathrm{i}] \rightarrow K[2i]\}$

3. *LLVM-IR*

```
bb:
  br label %bb1

bb1: ; Basic block of S4
  %indvar = phi i64 [ 0, %bb ], [ %indvar.next, %bb1 ]
; %indvar -> {0,+,1}<%bb1>
  %tmp = mul i64 %indvar, 2
  %gep = getelementptr [100 x float]* @K, i64 0, i64 %tmp
; %gep -> {@K,+,(2 * sizeof(float))}<%bb1>
  store float 3.000000e+00, float * %gep, align 8
  %indvar.next = add i64 %indvar, 1
  %exitcond = icmp eq i64 %indvar.next, 100
  br i1 %exitcond, label %bb2, label %bb1

bb2:
  ret void
; Loop %bb1: backedge-taken count is 99
```

Figure 2.3: Illustration of the compiler intermediate representation used by Polly [5] for polyhedral analysis.

formations are affordable approaches in production compilers. They proposed the GRAPHITE intermediate representation for the GCC compiler [29] (see Fig. 2.2). It combined static analysis with code transformations to effectively balance compilation time reduction and analysis precision. Inspired by this early work, Grosser et al. [5] applied the same principle to the LLVM compiler [30] (see Fig. 2.3), by recognizing that polyhedral techniques were limited to simple programs and specific programming languages. Accordingly, they proposed Polly to automatically detect and transform relevant parts of each program in a language-independent and syntactically transparent way. It deployed an advanced data dependency analysis and support for external optimizers and most common programming languages and compilation targets (such as Central Processing Units (CPUs), GPUs and hardware descriptions).

2.2.1.B Parallel Code Optimization

Other tools have also addressed the increased data transfer contention that characterizes multi-core processors. Majo et al. [7] presented a study showing that many loop-parallel programs include mutually incompatible data access patterns that can result in a high fraction of costly memory accesses. They proposed a set of language-level primitives for memory allocation and loop scheduling, that are used together with simple program-level transformations to eliminate mutually incompatible access patterns from OpenMP-style parallel programs. Similarly, Kiriansky et al. [10] recognized that traditional compiler cache optimizations have not been sufficiently aggressive to overcome the poor scaling of memory bandwidth in parallel applications. Accordingly, they introduced the *milk* language extension to allow programmers to annotate memory-bound loops for efficient parallelization with OpenMP. They used intermediate data structures to transform random indirect memory references into batches of efficient sequential memory accesses.

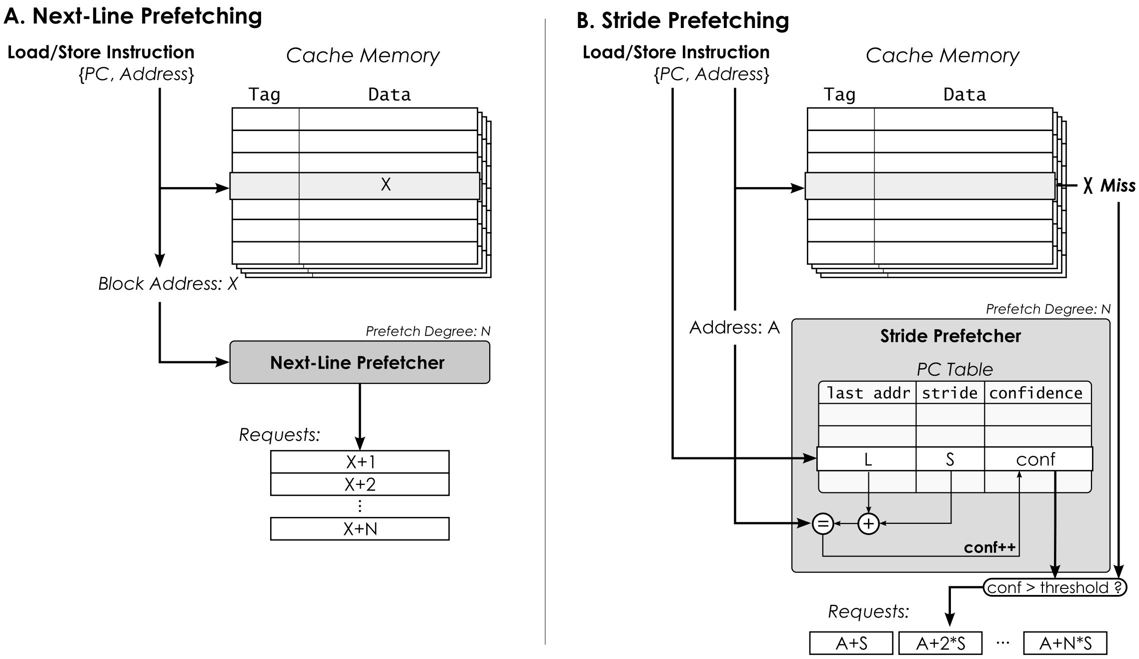


Figure 2.4: Common spatial prefetching mechanisms.

2.2.2 Data Prefetching Techniques

With the existing contention issues of conventional cache-based systems and shared communication infrastructures, prefetching had a strong reemergence in the past decade with the exploration and combination of several new sophisticated techniques. Such methods are particularly designed to deal with the intrinsic characteristics of particular types of applications, such as reduced data-locality and complex memory access patterns, memory-bound kernels or very large datasets that do not fit in the local memories.

Several solutions have been designed, by trading-off hardware complexity (area resources and power dissipation), prefetching intensity (degree of fetch requests to future addresses), accuracy (rate of correct fetch requests) and coverage (amount of prefetched data in the total dataset). Hence, to maximize the accuracy and the attained coverage (and consequently, the system's throughput), they must rely on complex dedicated modules that dynamically analyze the most recent memory access patterns and try to predict future accesses based on heuristic algorithms. A selected list of techniques is presented in the next paragraphs.

2.2.2.A Spatial Prefetching

Just as caches, the simpler and most common prefetching solutions, take advantage of the memory access spatial locality. By following this principle, sequential prefetchers try to preempt data accesses by populating the cache with several lines ahead of the most recently accessed address. In this category, *next-line prefetchers* (see Fig. 2.4.A) work by aligning a requested address to its corresponding block (cache line) address and then requesting the next *N* subsequent

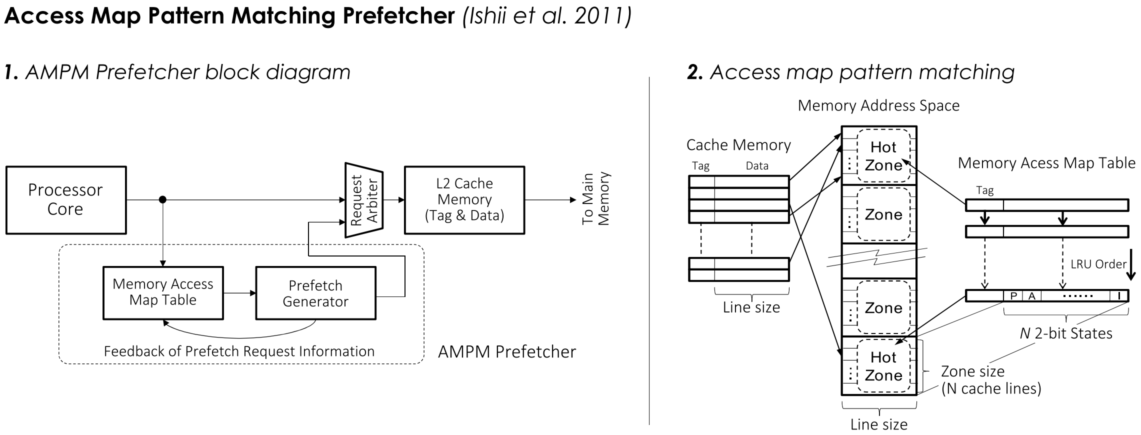


Figure 2.5: Temporal-based prefetching mechanism of the AMPM prefetcher [20].

lines (prefetch degree) from the memory hierarchy level.

Naturally, being this a hardly practical approach, the slightly more efficient *stride prefetchers* analyze the most recently requested addresses and calculate the difference between them (stride) to predict future accesses (see Fig. 2.4.B). To infer the correct sequence of addresses, these prefetchers typically rely on the use of prediction table structures indexed by the instruction address (Program Counter (PC)). As such, for each load/store instruction they register the last accessed address, the difference between subsequently accessed addresses (stride) and a confidence value (incremented every time the difference between addresses matches the calculated stride). When the confidence value reaches a predefined threshold the prefetcher issues requests to the next *N* strided addresses (as depicted in Fig. 2.4.B).

2.2.2.B Temporal Prefetching

Since memory access patterns can present characteristics more complex that strided address sequences [20], spatial prefetching approaches are often enhanced with temporal-locality-based heuristic prediction schemes to increase the achievable prefetch coverage. By following that principle, Ishii et al. [20] proposed the Access Map Pattern Matching (AMPM) prefetcher (see Fig. 2.5) to identify hot zones in memory and infer multiple patterns in the access stream. To make accurate predictions, it uses a bitmap-like data structure to hold the information of memory accesses occurred in the recent past. Next, it divides the memory address space into memory regions of a fixed size and relates the memory access map to each region. Finally, it makes use of pattern matching logic for inferring memory access patterns in each mapped region, detecting all possible strides at the same time. With the detected patterns, it predicts future memory accesses independently of the order in which they are observed, in turn attaining a high prefetch coverage.

By recognizing that temporal address correlation often allows the detection of complex memory access patterns with higher accuracy, Global History Buffers (GHBs) [31, 33] were introduced to represent a temporal address history. They hold the most recent memory accesses in a First-In

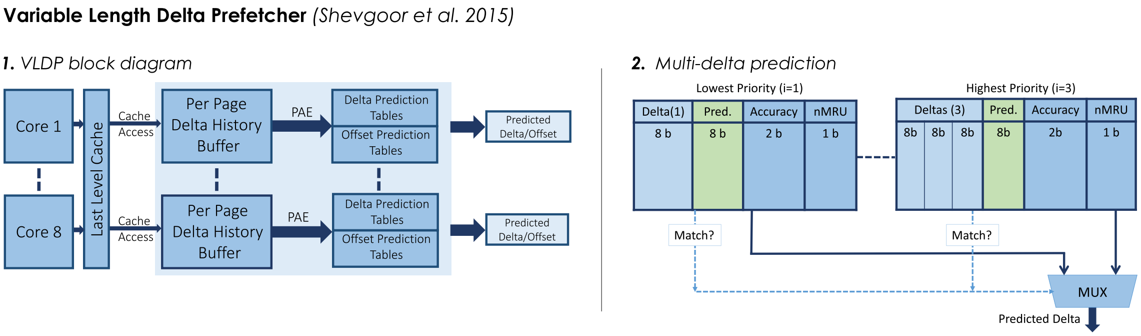


Figure 2.6: Temporal-based prefetching mechanism of the VLDP prefetcher [32].

First-Out (FIFO) buffer and use a linked list to store the sequence of accessed addresses. However, although being more efficient than prediction tables, in some cases it may involve a large number of sequential accesses and modifications to the linked list, resulting in long latency in the prediction of the addresses to be prefetched.

To mitigate such issues, GHBs are often combined with other prediction structures, allowing researchers to efficiently exploit the history of accesses to calculate accurate future accesses [21, 32]. As an example, Shevgoor et al. [32] proposed the Variable Length Delta Prefetcher (VLDP). It maintains local delta (stride) histories for each physical page accessed by a given workload in multiple delta history buffers with different lengths. When a prefetching opportunity is detected in one of the tracked pages, its delta history is used to perform lookups in delta prediction tables (see Fig. 2.6) to infer the correct prefetches to issue. The utilization of multiple prediction tables (corresponding to different history lengths), allows the VLDP to maximize prefetch coverage and accuracy, by preferring to use long histories to make accurate prefetches and using shorter histories to fill in the gaps when long histories are unavailable.

2.2.2.C Irregular and Correlation Prefetching

Since spatial and temporal prefetching rely on the assumption that accesses maintain a certain regularity over time, they fall short when the memory access is not sequential or presents some degree of irregularity. Hence, to deal with such situations, temporal and spatial correlation heuristics have been deployed in some prefetchers. They usually deploy probabilistic and automatic learning algorithms that analyze the recent address history and, instead of calculating differences, identify regions of interest in the memory to be prefetched. As an example, in [55] it is introduced a context-based memory prefetcher, which approximates spatial and temporal locality by applying reinforced learning methods over system and code attributes, which provide valuable hints for memory access prediction.

Recently, Pugsley et al. [23] introduced a new category of offset prefetchers, as a generalization of the classic next-line prefetching, supported by a sandbox method to dynamically select the prefetch offset (see Fig. 2.7.A). It works on the principle of validating the accuracy of the

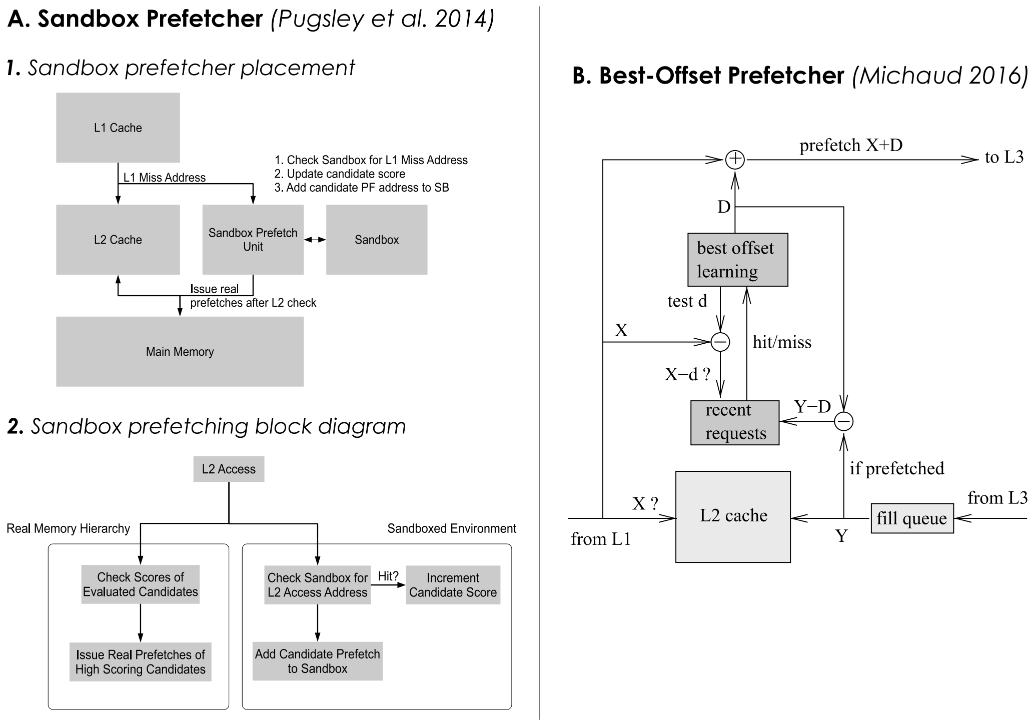

A. Sandbox Prefetcher (Pugsley et al. 2014)
1. Sandbox prefetcher placement
L1 Cache
L1 Miss Address
1. Check Sandbox for L1 Miss Address
2. Update candidate score
3. Add candidate PF address to SB
L2 Cache
Sandbox Prefetch Unit
Sandbox
Issue real prefetches after L2 check
Main Memory
2. Sandbox prefetching block diagram
L2 Access
Real Memory Hierarchy
Sandboxed Environment
Check Scores of Evaluated Candidates
Issue Real Prefetches of High Scoring Candidates
Check Sandbox for L2 Access Address
Hit?
Increment Candidate Score
Add Candidate Prefetch to Sandbox
B. Best-Offset Prefetcher (Michaud 2016)
prefetch X+D
to L3
D
best offset learning
test d
X
hit/miss
X−d ?
recent requests
Y−D
if prefetched
from L3
X ?
L2 cache
fill queue
from L1
Y

Figure 2.7: Schematic views of (A) the Sandbox Prefetcher [23] and (B) the Best-Offset Prefetcher [22].

prefetchers in a safe, sandbox environment, where neither the real cache nor memory bandwidth is disturbed. To do so, the proposed method issues ”fake” prefetch requests by recording them in a bloom filter structure and calculating their accuracy by checking if a subsequent access hits in the bloom filter. The actual prefetch candidates are then deployed in the real memory hierarchy only if they prove that they can accurately prefetch useful data.

Following the same principle, Michaud [22] proposed the Best-Offset Prefetcher. It was designed with prefetching timeliness in mind and implements an offset selection mechanism that dynamically sets the prefetching offset depending on the application behavior to maximize prefetch coverage (see Fig. 2.7.B). This mechanism relies on a best-offset learning algorithm that tries to find the best prefetch offset by testing several different offsets. It does so by recording the base address of prefetch requests that have been completed in a recent requests table and inferring the correct timeliness for a prefetch candidate based on how recently a cache line was accessed.

Due to its detachment from spatial and temporal locality, address correlation is also adopted to deal with irregular applications (such as data-based accesses, graph- and list-oriented and pointer-based applications). To deal with the irregular nature of graph structures, Zhang et al. [17] proposed a worklist-directed prefetching technique, that takes advantage of the processing system’s knowledge of upcoming work items to issue prefetch requests accurately. The prefetcher is paired with a credit-based system to improve prefetch timeliness and prevent cache thrashing. Jain et al. [19] proposed a structural address space that translates arbitrary pairs of correlated physical addresses into consecutive addresses. This allows their proposed Irregular Stream Buffer (ISB) to organize prefetching meta-data so that it is simultaneously temporally and spatially ordered.

2.2.2.D Compiler-aided Prefetching

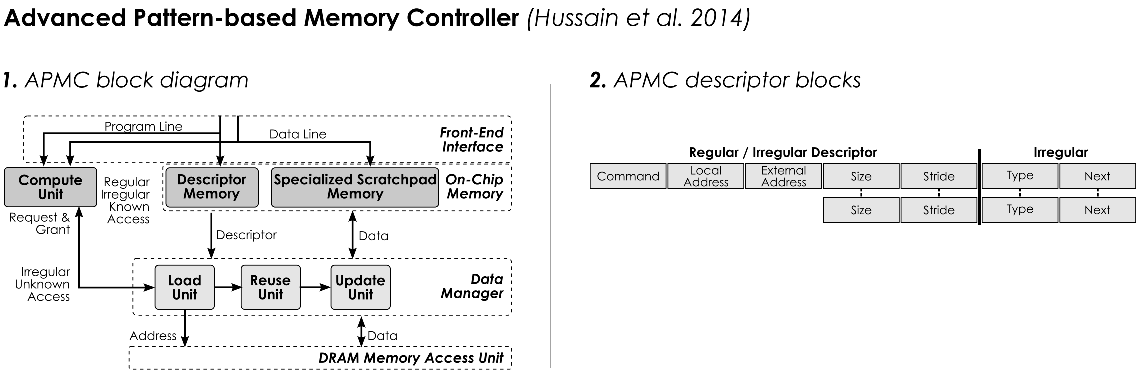
Although predictive approaches allow a high abstraction of the application from the prefetching procedure, they can fall short when the application is characterized by complex memory access patterns. Moreover, they impose an increased amount of resources, often related to the adopted level of prefetching aggressiveness [18, 23].

Hence, alternative solutions have been considered [18, 53] that rely on compile-time procedures, where the compiler pre-analyzes the code and tries to extract/model the application memory access pattern. Such information is then fed to on-chip prefetching modules, mitigating the effort of their prediction procedures. By eliminating unnecessary requests, these approaches reduce the prediction overheads while also improving the accuracy of the prefetchers. They also lead to far simpler hardware structures, since no runtime analysis is performed, in turn resulting in lower-footprint and more energy efficient controllers, at the cost of an increased pre-processing effort.

Ebrahimi et al. [53] explored such an approach, through a hardware/software cooperative technique for the prefetching of linked data structures. They proposed a compiler-guided prefetch filtering mechanism that informs the hardware about which pointer addresses to prefetch, and a prefetch throttling mechanism that relies on runtime feedback to manage the operation of multiple prefetching architectures. Another example is the Power-Aware pRefetch Engine (PARE) [18], a compiler-assisted prefetching mechanism, which utilizes compiler information to selectively apply different hardware prefetching schemes based on predicted memory access patterns. It also applies filtering techniques at compile-time and at runtime to reduce the number of accesses to be prefetched by hardware, consequently reducing the L1 cache related energy overhead due to prefetching.

2.2.3 Data Streaming Architectures

Despite the high accuracy provided by current prefetching approaches, their predictive nature and necessary runtime monitoring impose inevitable inaccuracies and delays in the memory access infrastructure. In particular, while the prediction delays are inevitable to maximize accuracy and coverage in such scenarios, an increased amount of resources is required, often increasing the system's overall energy consumption [18]. Moreover, even the most sophisticated techniques are not well-suited to processing systems with large numbers of processing cores, due to possible contention introduced in the shared interconnections caused by massive on-the-fly prefetch requests. Such contention may even degrade the system's throughput to the point where the


Advanced Pattern-based Memory Controller *(Hussain et al. 2014)*

1. *APMC block diagram*

2. *APMC descriptor blocks*

Figure 2.8: Overview of the APMC [54] (1) stream controller and (2) pattern descriptor representation.

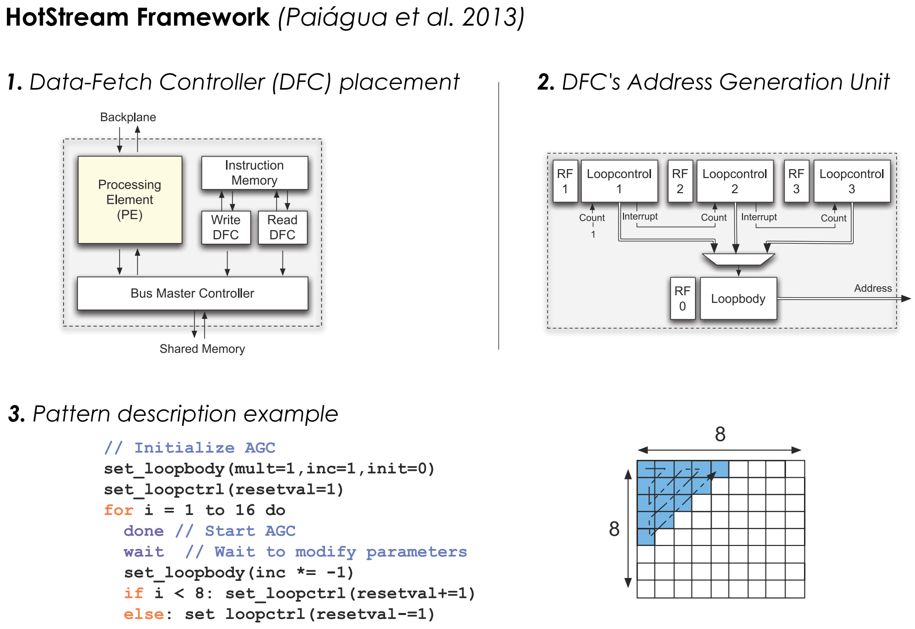
benefits of the prefetching mechanism become obsolete.

Alternatively, by relying on data streaming approaches it is possible to deploy data acquisition mechanisms without directly relying on aggressive predictive schemes and structures. Such approaches essentially reverse the conventional request-based memory access process by autonomously generating the indexing of the memory and transparently sending data to the processing system. This results in a significant mitigation of the contention in the communication infrastructures and also allows exploiting runtime data manipulation mechanisms (e.g., on-the-fly data reorganization and reutilization). Moreover, far simpler hardware structures are required, since no runtime analysis is performed, in turn resulting in controller structures with lower area footprints and higher energy efficiency.

Accordingly, several data streaming techniques have been proposed to improve the throughput of the data management infrastructure. Park et al.[56] formally studied the problem of data fetching from an external memory to a Field-Programmable Gate Array (FPGA) in the context of stream-computing. Acknowledging that proper data re-utilization mechanisms are fundamental in FPGA-based systems, the work was further extended to support the scheduling of simple data operations (e.g., stripping, splitting or merging) in the context of multi-processor systems [57].

Meanwhile, Hussain et al.[54] proposed the APMC (see Fig. 2.8), which supports regular 1D, 2D and 3D data-fetching mechanisms, such as scatter-gather and strided accesses. A dedicated scheduler, allied to an intelligent memory manager is also integrated into the controller, to facilitate the implementation of elaborated data movements, as well as simple computational tasks. However, while the APMC represents a step forward towards the streaming of complex patterns, it was designed for moving large and regular data chunks and falls short with irregular or complex memory indexing. Irregular and pointer-based accesses are managed through runtime memory access analysis and recording (similar to prefetching mechanisms).

The issue of complex data pattern generation and efficient data manipulation schemes was addressed by Paiágua et al. [26]. They proposed the HotStream Framework, which relies on a



```
// Initialize AGC
set_loopbody(mult=1,inc=1,init=0)
set_loopctrl(resetval=1)
for i = 1 to 16 do
  done // Start AGC
  wait  // Wait to modify parameters
  set_loopbody(inc *= -1)
  if i < 8: set_loopctrl(resetval+=1)
  else: set loopctrl(resetval-=1)
```

Figure 2.9: Overview of the HotStream Framework [26] (1) data-fetch controller, (2) address generation unit and (3) pattern description language.

micro-coded approach to program memory access patterns directly to on-chip Data-Fetch Controllers (DFCs) (see Fig. 2.8). Each DFC is composed of an address generation controller (that generates up to three-dimensional patterns) and managed by a custom micro-controller that combines multiple patterns to generate higher dimensional address sequences. The DFCs also allow the deployment of several data reutilization techniques that aim at maximizing the main memory bandwidth. Although the considered programmable approach eases the description of higher dimensional data patterns without compromising the address issue rate, it still struggled with many complex data access patterns.

Hence, although these streaming approaches provided a step forward in data transfer efficiency, they have only been successfully deployed in custom dedicated accelerators. Moreover, they still lack in viability since they require the programmer to encode the memory access pattern of each application manually.

2.2.4 Discussion

The adoption of conventional cache hierarchies in current computing systems still imposes important limitations to the offered performance. Although they are capable of reducing the memory access latency in several advantageous scenarios, their performance is bound by the characteristics of the running application and the topology of the processing system, specifically:

- **Poor memory access spatial and temporal locality** can result in high numbers of early cache line evictions and consequently increased miss rates. Such scenarios typically occur when the application is characterized by large datasets that do not fit in the cache memory or by complex memory access patterns;

- **Large numbers of processing cores** in a cache-coherent system tend to increase its contention and the overall energy consumption, resulting not only from increased on-the-fly memory requests but also from the complexity of the underlying shared data transfer infrastructures.

While prefetching techniques have shown success in mitigating the adverse characteristics of running applications, and are capable of providing high data acquisition timeliness and effectiveness [22, 23], they are reaching throughput limits caused by:

- **Monitoring and prediction delays** caused by the memory access analysis that must be performed during application execution to predict future accesses;
- **Prediction inaccuracies** that result in unnecessary data acquisition requests, and can in turn flood the shared communication structures.

Recent studies showed that data streaming is a viable alternative to conventional memory access hierarchies in application-specific architectures [26, 54]. Not only do they detach the memory access generation procedure from the processing infrastructure, but they also provide efficient data transfer, reutilization, and manipulation schemes, that are not supported in conventional systems. However, their applicability to general purpose systems is still limited to:

- **Deterministic representations of the memory accesses**, with limited data-pattern description complexity, and providing no support for the irregular accesses and pointer-based accesses that characterize several HPC applications;
- **Manual coding of the data streaming scheme** resulting from the lack of proper compilation tools that can extract and encode an application's memory access pattern and data communication scheme.

Hence, to effectively allow the deployment of data streaming in general purpose systems, it is necessary to take a step further from typical static analysis tools that solely analyze memory access patterns and perform code optimization at compile-time [5–7, 9, 10]. This can be done by combining memory access pattern analysis with new dedicated representations that can encode arbitrarily complex sequences of addresses.

2.3 Adaptive Computing

To avoid the disadvantages inherent to power-hungry high-end general purpose processing systems or low-power and application-specific dedicated architectures, several systems have recently adopted adaptive structures to simultaneously handle broader ranges of applications and workloads. Partly driven by the increased integration levels that were observed over the past decade, and by the consequent adoption of heterogeneous systems and new emerging technologies, new techniques are being explored that combine the advantageous characteristics of several specific or dedicated approaches. In fact, while conventional approaches become less viable to cope with the current performance-efficiency demands, several early solutions, often disregarded due to the drawbacks of early technology (e.g., partial reconfiguration) are being revisited and adapted to the current HPC context.

2.3.1 Power Supply Management

Several circuit management mechanisms have been proposed to tackle the increasing demand for energy efficient platforms. This includes turning off parts of the processor [11, 13] or using dynamic voltage and frequency scaling approaches [14, 15] to decrease energy consumption whenever the computational requirements decrease. Recently, researchers have also turned to multi-core heterogeneous systems composed of high performance *big* cores and low-power *small* cores to reduce the power consumption of the whole system, while providing adaptive processing capabilites [12, 15]. These systems typically exploit a common Instruction Set Architecture (ISA) among all the cores, to facilitate task migration from the *big* to the *small* cores (and vice-versa), which allows for fast and efficient switching between high-performance and low-power scenarios, depending on the application requirements and constraints.

However, to adequately exploit the existing resources on heterogeneous multi-core systems, complex monitoring, task migration, and scheduling procedures need to be performed, which also depend on the execution profile. In [58], key metrics are identified to characterize the application under execution, including the core type that best suits its resource requirements. A combination of static analysis and runtime scheduling is proposed in [59], to achieve an energy efficient scheduling on Intel's QuickIA heterogeneous platform [60]. The utilization of statistical approaches has also been explored in [61], where a performance impact estimation is used as a mechanism to predict which workload-to-core mapping is likely to provide the best performance.

2.3.2 Dynamically Reconfigurable Systems

Although the adoption of heterogeneous computing architectures has proved advantageous in many scenarios, further energy savings can be attained by adapting the computing architecture/communication topology to the characteristics of running multi-threaded applications. Some possible approaches are either the application of network reconfiguration or adaptive core interconnection topologies [47, 62], changing the cache configuration [63], or performing core morphing [48, 64, 65]. Such solutions have been significantly aided by the recent advances in FPGA [41, 42] and Coarse-Grained Reconfigurable Array (CGRA) [43, 44] technologies. These systems rely on fast dynamic partial reconfiguration, which provides the possibility to reconfigure a selected region of the device, while the remaining logic continues to operate in an uninterrupted way [66].

Despite the similarities between their reconfiguration processes, different advantages (and drawbacks) make FPGAs and CGRAs better-suited for particular execution scenarios. The main functional difference between these technologies lies in the granularity and diversity of their basic reconfigurable elements. Specifically, FPGAs provide bit-level reconfigurability, with a wide variety of configurable resources (e.g., BRAMs, Look-Up Tables (LUTs), Configurable Logic Blocks (CLBs)) and embedded elements (e.g., clocking, full CPUs). On the other hand, CGRAs provide a data-path-level reconfigurability, with a more restrictive and custom-tailored range of configurable Processing Elements (PEs) (e.g., Arithmetic and Logic Units (ALUs), Floating-Point (FP) units) embedded in SoCs, containing hardwired custom communication networks and memory subsystems.

2.3.2.A Coarse-Grained Reconfigurable Arrays

Given the inefficiency of the reconfiguration procedures and resource limitations of early FPGA technologies, the first viable reconfigurable architectures were deployed in CGRAs. Contrarily to the bit-level reconfigurability of FPGAs, CGRAs provide data-path-level reconfigurability. Their reconfigurable units include sets of predefined functional units and complex operators that can be interconnected to create custom operations.

One of the earlier CGRAs was the MorphoSys [67]. It comprised a reconfigurable processing unit organized in a Single-Instruction Multiple-Data (SIMD) fashion as an array of reconfigurable cells with word-level granularity. A specialized streaming buffer handled the data transfers between external memory and the array. Reportedly, it achieved Application Specific Integrated Circuit (ASIC)-level performance with a single-cycle reconfiguration latency. On a different approach, the ADRES framework [68] deployed a flexible Very Long Instruction Word (VLIW) processor architecture template organized in an array of tightly coupled configurable processing cells. Its architecture allowed the dynamic reconfiguration to be performed without stalling the array. Data input from various possible sources was done with the help of configurable port sides in the functional units.

While early CGRAs were critical in demonstrating the viability of reconfigurable architectures for high-performance processing, it was the PACT XPP-III [69] processor that presented the most significant step forward in reconfigurable processors. The XPP-III [69] is a strictly modular and hierarchical design architecture that combines data stream processing in an array with runtime reconfiguration mechanisms. Each configuration is a parallel computation module derived from the data-flow graph of a given application. The array provides high parallel processing performance for typical stream-based applications, resulting from its run-time reconfiguration mechanisms that allow entire applications to be configured and run independently on different parts of the array.

Despite the breakthroughs that were achieved at the time, only recently have CGRAs began to reemerge, as a result of the recent advances in integration and production technology. One example is the HARP [44] integrated platform that combines multiple CGRAs over a Network on

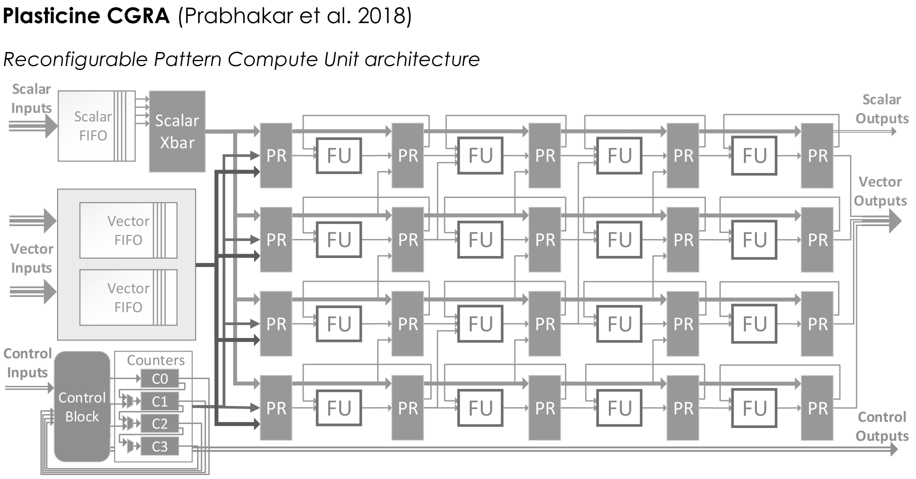


Figure 2.10: Architecture of a reconfigurable compute unit from the Plasticine CGRA [43].

Chip (NoC), where each CGRA is scaled and tailored for a specific application. Its loosely-coupled architecture was designed to maximize the number of instantiated reconfigurable PEs, by placing several heterogeneous and homogeneous accelerators around a CPU core. The processor is used to supervise and control the PEs for independent and simultaneous execution of multiple kernels over the platform. Recently, Prabhakar et al. [43], proposed the Plasticine reconfigurable accelerator architecture designed to execute parallel patterns efficiently. It deploys a 2D array of reconfigurable units (see Figure 2.10), including: *i)* compute units, that exploit fine-grained SIMD and pipeline parallelism; *iii)* memory units, to exploit data-locality using banked scratchpad memories with configurable address logic; *iii)* pipelined switches that implement control networks with multiple granularities; and *iv)* address generators and coalescing units to efficiently perform data accesses to DRAM.

2.3.2.B FPGA-based Reconfigurable Accelerators

Motivated by the same technology advances that drove the reemergence of CGRAs, several high-performance and adaptive many-core heterogeneous systems have been proposed to exploit the reconfigurable capabilities of FPGAs.

Early solutions envisaged the simultaneous instantiation of multiple architecture configurations in the device. As an example, Garcia et al. [70] proposed a reconfigurable multiprocessor system that allows multiple configurations to coexist by using reconfigurable coprocessors with multiple cores. Caspi et al. [71] proposed a design that incorporates a single CPU and multiple reconfigurable computing blocks, with data streams being transferred between the blocks over a dedicated interconnect. On a lower level, Chen et al. [72] proposed the inclusion of reconfigurable ISA support in multi-core processors, allowing the exploitation of adaptive fine-grained and coarse-grained parallelized architectures.

Some unconventional approaches have also been proposed that exploit the reconfiguration

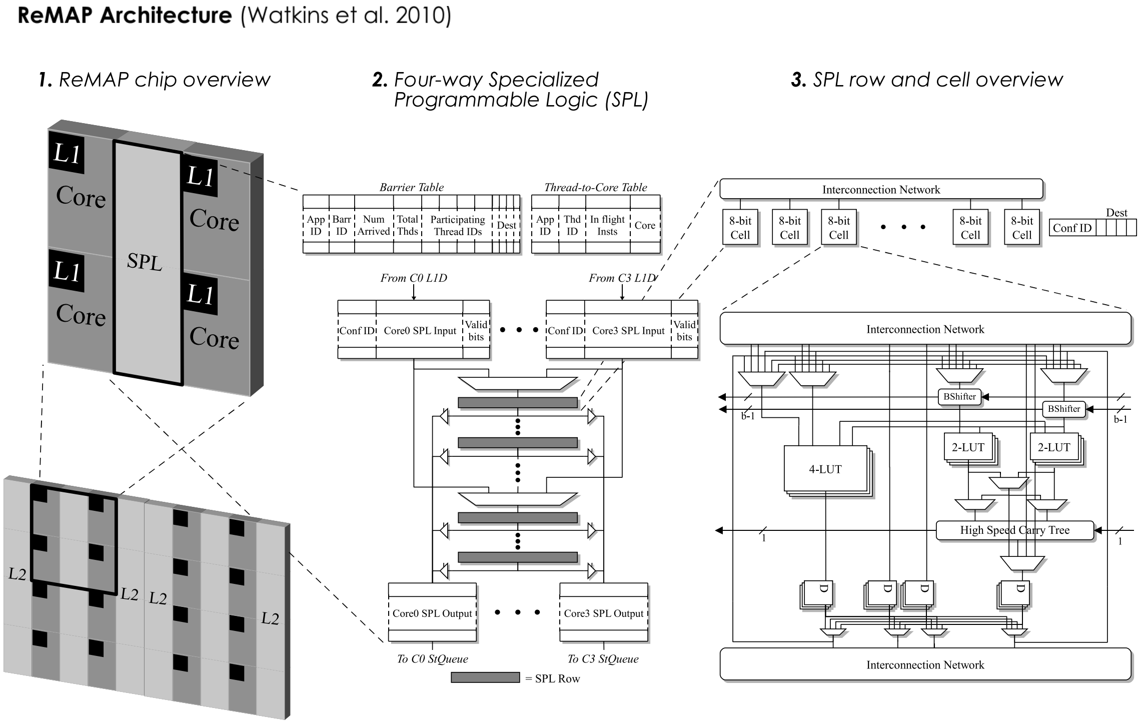


Figure 2.11: Architecture of the ReMAP heterogeneous accelerator [73].

capabilities of FPGA fabric in particular manners. One of such examples is the ReMap system [73], which integrates reconfigurable computation modules within an accelerator's interconnection buses (see Fig. 2.11). Hence, the proposed approach provides acceleration by performing computation operations to the data, while it is being transferred between the accelerators components.

Despite the attainable levels of architecture specialization, dynamic reconfiguration has also been used to optimize the processing system and save energy. To validate this idea, Lorenz et al. [74] compared the energy that is required in the reconfiguration process with the potential savings that are introduced by a dynamic and adaptive change of the computing units of the processing system.

Recently, Ordaz et al. [42] proposed an architecture comprising a soft dual-processor system augmented with a runtime reconfigurable SIMD engine. The deployed engine is automatically split into multiple lanes according to the computational requirements of the running application. The SIMD engine is integrated into the dual-processor system through a small set of special instructions to invoke SIMD operations at runtime.

2.3.2.C FPGAs vs. CGRAs

Their fundamental technological difference makes FPGAs the most flexible reconfigurable devices available today. In fact, any application can be designed and instantiated on a FPGA device allowing, in particular cases, higher throughputs with lower energy consumptions, when compared to static implementations in Integrated Circuit (IC) technology (as recently reported in[49]). However, computationally intensive applications can incur in significant overheads in terms of area and latency, resulting from the bit-level routing penalties imposed by the implementation of complex operations.

On the other hand, at the expense of some flexibility, CGRAs provide a set of predefined datapaths and complex operators that can be interconnected to create custom operations. Despite occupying a larger area than a CLB, the wider multiple-bit computing elements allow the total number of instantiated elements to be much lower than in FPGAs, in turn requiring a lower global area for routing. Furthermore, communication networks and memory access infrastructures are typically hardwired, making such resources extremely efficient when compared to FPGAs. This is mainly due to the fact that FPGAs require an explicit coding of the communication between operators and the memory elements, typically occupying a significant percentage of the available resources, due to routing and chained multiplexing.

Regarding market applicability, the viability of both approaches as co-processing devices increased significantly over the last decade [41]. The integration of CGRAs in heterogeneous SoCs as co-processors [75] is no different than any other custom module. Similarly, FPGAs are already supplied with embedded processor cores connected to the reconfigurable fabric [76]. However, both approaches still require a non-negligible development effort per application. In fact, despite the emergence of new automated design tools (e.g., Xilinx Vivado [77], Altera SDK for OpenCL [78], Maxeler Technologies' MaxCompiler [79]) that automatically translate applications to low-level hardware, the hardware implementation must still be carefully described and optimized to maximize an application's throughput, requiring a non-negligible designer's effort.

2.3.3 Dedicated Programming Frameworks and Execution Optimization

Although several programming frameworks have already been developed with the particular purpose of exploiting GPUs for general purpose computation (e.g., CUDA and OpenCL), not much attention has been given to reconfigurable hardware accelerators, especially those deployed on reconfigurable technology.

Conversely, several solutions have been proposed to manage and optimize the execution of heterogeneous many-core processing systems. Some examples comprise dynamic task scheduling [80–82], high-level and system-level synthesis approaches [83–85], application mapping tools for multi-processor [86, 87] and heterogeneous systems [88], and routing and communication topology optimization mechanisms [89]. However, these solutions do not take advantage of the runtime reconfiguration/adaptation capabilities of the underlying hardware.

2.3.3.A Design Space Exploration

Although the above-mentioned tools [77–79] provide the means for automatic task/kernel translation into low-level hardware implementations and for mapping them to custom hardware, the runtime reconfiguration of the architectures is usually not taken into account. To address this lack of support, several Design Space Exploration (DSE) algorithms have been proposed to specifically target dynamically reconfigurable platforms [48].

In the context of hardware implementations, DSE has been used to perform systematic analyses of several design options (e.g., type and number of components, the degree of architecture specialization, interconnection topologies) and parameterization choices (e.g., power, performance, cost), according to a given optimization goal. This is usually done by trading off design complexity and parameter weight through an exploration process that results in several differently balanced system designs.

As an example, Miramond et al. [90] proposed a tool that defines different contexts for reconfigurable circuits, which are subsequently switched through partial reconfiguration. The underlying tasks are then assigned to them through spatial and temporal partitioning, by using a local search algorithm. Meanwhile, Czarnecki et al. [91] proposed the utilization of conditional task graphs to model mutually exclusive tasks. According to this algorithm, all tasks are initially assigned to only one GPP module, and new solutions are produced using iterative improvement methods.

2.3.3.B Evolutionary Algorithms and Multi-Objective Optimization

While DSE is a well-known system design approach, the number of possible design choices and parameter variations that characterize complex system implementations [90, 91] may lead to inefficient optimization procedures. To tackle this issue, Multi-Objective Optimization (MOO) methods have been regarded as viable approaches to deliver highly optimized DSE solutions.

MOO problems are designed with the goal of optimizing a set of objective functions, to which it is not possible to find a single solution that simultaneously minimizes (or maximizes) all the objectives. The resulting set of solutions, denoted as the *Pareto optimal front*, contains the points representing the best trade-off solutions in the objective space. Typically, a multi-objective optimization problem is defined as:

$$\begin{aligned} &minimize \;\; f(x) = (f_1(x), ..., f_k(x)) \\ &subject\ to \;\; x \in X \quad , \end{aligned} \tag{2.1}$$

where x is a solution, X is a set of feasible solutions, and $f(x)$ is an objective function vector that maps a solution vector x in the decision space to a point in the objective space. A solution is in the Pareto front, and is called *non-dominated* (by any other solution), if none of its objective functions' values can be improved without degrading the others [92], representing a best-compromise solution in the objective space. Formally, given two solutions x and v, the Pareto dominance property between them (i.e., x *dominates* v) is defined as:

$$x \prec v \ iff \ f_i(x) \leq f_i(v), \forall_i \in \{1, ..., k\} \wedge \exists i \in \{1, ..., k\} \ s.t. \ f_i(x) < f_i(v). \tag{2.2}$$

A wide variety of methodologies has been proposed to solve MOO problems [92]. Among the most used are the Multi-Objective Evolutionary Algorithms (MOEAs), since these algorithms are particularly well-suited for simultaneously dealing with solutions in large search spaces. Specifically, evolutionary approaches are designed by encoding each potential solution as a chromosome of an individual in a given population. Hence, the set of *non-dominated* solutions that define the Pareto front is obtained by evolving the population through a particular number of generations. The population, itself, is evolved by applying operations based on nature's evolution mechanisms, such as selection, reproduction, and mutation.

In particular, one of the most well-established MOEAs is the Non-dominated Sorting Genetic Algorithm II (NSGA-II) [93], characterized by a very fast convergence and a good spread of solutions near the Pareto front. This is achieved by including a fast non-dominated sorting approach, followed by a selection operator that, together, create a mating pool by combining the parent and offspring populations and by subsequently selecting the best (on fitness and spread). Furthermore, the NSGA-II algorithm is especially well-suited for solving problems with multiple constraints [93].

Accordingly, since a vast number of mapping and architectural constraints are inherently imposed in a DSE problem, it can be regarded as a constrained MOO problem [92]. As an example, based on the application's translation to conditional task graphs, Shang et al. [94] proposed an Evolutionary Algorithm (EA) to determine the number of used resources and to assign them with tasks, followed by a re-mapping and scheduling algorithm that makes use of the dynamic reconfiguration capabilities of FPGAs. Wildermann et al. [95] proposed an algorithm that targets multi-mode systems. It is based on the notion that different operating modes can share the same hardware resources through partial reconfiguration. A symbolic encoding combines a SAT solver and a MOEA is used to perform the system synthesis for allocation, binding, and placement of partially reconfigurable modules, both temporally and spatially.

In a more recent approach, Grigore et al. [96] tackled the runtime placement problem in FPGAs. They proposed a model to place rectangular modules on a datapath with predetermined communication constraints, that takes into account the resource requirements of running tasks and minimizes fragmentation. The model uses string matching to compute all possible placement positions of any given rectangular module, providing information for the runtime placer, resulting in faster execution.

2.3.4 Reconfigurable Communication Systems

Although adaptive structures have been exploited to improve the efficiency of the processing infrastructure in heterogeneous systems, the communication infrastructure is usually kept with non-reconfigurable and generic structures (e.g., NoC or shared-bus structures). This is mostly because the straightforward addition of reconfiguration procedures would incur in unsurmountable latency and energy consumption overheads to the already inherent overheads of currently established communication infrastructures.

Nonetheless, there are still some cases where energy-efficiency is targeted with the adaptation of the communication subsystem (e.g., cache structures, local memories, network/bus topologies). As an example, Weixun et al. [63] deployed a technique that combines dynamic reconfiguration of private caches and partitioning of the shared cache, to minimize the cache hierarchy energy consumption. Meanwhile, Sundararajan et al. [50] proposed a cache architecture that allows the dynamic reconfiguration of both its size and associativity, whose best configuration for a given application is dynamically predicted with the aid of a decision-tree machine learning model. Similarly, El-Moursy et al. [97] proposed a v-set cache design, targeting an adaptive and dynamic utilization of cache blocks for shared LLCs in multi-core environments.

The combination of multiple memory technologies has also been explored to increase the system's data communication efficiency. Chen et al. proposed a reconfigurable hybrid last-level cache design, where different memory technologies (SRAM and Non-Volatile Memory (NVM)) are unified at the same cache level to form a hybrid design. Moreover, power gating circuitry is also introduced to allow adaptive powering on/off of SRAM/NVM sub-arrays at the way level. To increase the cache performance of a multi-core processor, Kalokerinos et al. [52] proposed a hybrid architecture comprising a local scratchpad memory that can be partly configured at runtime. This scratchpad memory operates as a local second level cache, providing unified hardware support for both implicit and explicit communication. The communication is handled with an integrated network interface that deploys virtualized Direct Memory Access (DMA) procedures.

Although adaptable, all these approaches are mostly focused on balancing the available memory and interconnection resources, and they still incur in inevitable overheads resulting from the reconfiguration process. Moreover, they do not directly address the communication process itself, meaning they still struggle when dealing with communication concurrency and contention or with data locality issues resulting from the characteristics of running applications.

2.3.5 Discussion

While heterogeneous systems took a step further in providing new levels of computing specialization, there is still a performance limit caused by an inefficient utilization of hardware resources. In particular:

- **Task migrations between processing components** result in inevitable delays that degrade the system's throughput;
- **Power supply management and performance throttling** can result in a significant amount of resource being powered-down most of the time, making their utilization ineffective.

Alternatively, partial reconfiguration can provide the necessary mechanisms to improve the utilization of processing resources and simultaneously improve the processing infrastructure's computing efficiency. However, it is necessary to acknowledge that the existing state-of-the-art is still focused on custom dedicated accelerators and lacks the adequate responsiveness and adaptability to a general-purpose context, specifically:

- **The reconfiguration process** imposes non-negligible time overheads and power dissipation that can significantly impact its the performance and energy consumption, and also result in inevitable delays that degrade the system's throughput;
- **Application-to-accelerator mapping tools** lack support in what concerns the reconfigurability of the processing hardware, in turn decreasing its viability.

On the other hand, while new levels of adaptation could also be provided in the memory subsystem, partial reconfiguration is not a viable approach in this area, due to the significant impacts in performance that would occur from constantly stopping the communication channels to perform reconfiguration. Nonetheless, new adaptive techniques can still be investigated to address the performance and energy consumption limitations of conventional data communication schemes. In particular, they can explore the combination of different data communication paradigms and data fetching mechanisms (such as cache hierarchies, data streaming and prefetching), to provide new levels of adaptation in new hybrid data management infrastructures.

2.4 Summary

This chapter presented a brief overview of the current limitations imposed on computing systems by power supply management mechanisms and by the adoption of conventional cache hierarchies. Possible solutions were also highlighted that can counteract the identified limitations. In particular, it was recognized that adaptive computing systems (in the format of hardware reconfiguration and dynamic data communication schemes) can provide the necessary mechanisms to bridge that gap and to push the limits of computing efficiency. Accordingly, a brief review of the state-of-the-art was presented regarding the areas of heterogeneous computing systems, reconfigurable architectures, data prefetching mechanisms, data streaming architectures, and compilation tools for memory access analysis and task mapping and scheduling.

3

Data-Pattern Analysis and Stream Transformations

Contents

3.1 Modeling of Complex Data-Patterns 35
3.1.1 Affine Mathematical Model for Data Indexing 37
3.1.2 Memory Access Description Specification 39
3.2 Compile-Time Stream Code Generation 42
3.2.1 Compiler Module Overview 43
3.2.2 Context Representation Language 46
3.2.3 Stream Code Generation 49
3.3 Preliminary Experimental Evaluation 53
3.3.1 Methodology 53
3.3.2 Data-Pattern Encoding Efficiency 56
3.3.3 Source Code Reduction Evaluation 58
3.3.4 Discussion 59
3.4 Summary 60

As it was described in the previous chapters, the conventional communication schemes equipping most general-purpose systems require the processing cores to handle the complete memory access procedure (including address generation and memory access requests). Moreover, they rely on cache hierarchies to handle the corresponding data transfers as efficiently as possible. In contrast, stream-based mechanisms rely on dedicated on-chip modules (stream controllers) that perform both address generation and handle data transfers directly to/from memory, independently of the system's processing cores. Moreover, they deploy stream buffering structures in the communication infrastructure that allow the exploitation of unique data manipulation techniques (such as on-the-fly data reutilization and reorganization).

Accordingly, such dedicated stream controllers can be deployed close to the processing cores [26] (see Fig. 3.1.A), where aside from providing a functionality similar to that of a prefetcher, also allow the exploitation of point-to-point communication schemes between the system's cores. Alternatively, these controllers can also be deployed closer to the system's main memory [54] (see Fig. 3.1.B). This option also allows a significant mitigation of the memory access concurrency (typical to multi/many-core systems) by sending data directly from memory and allowing the

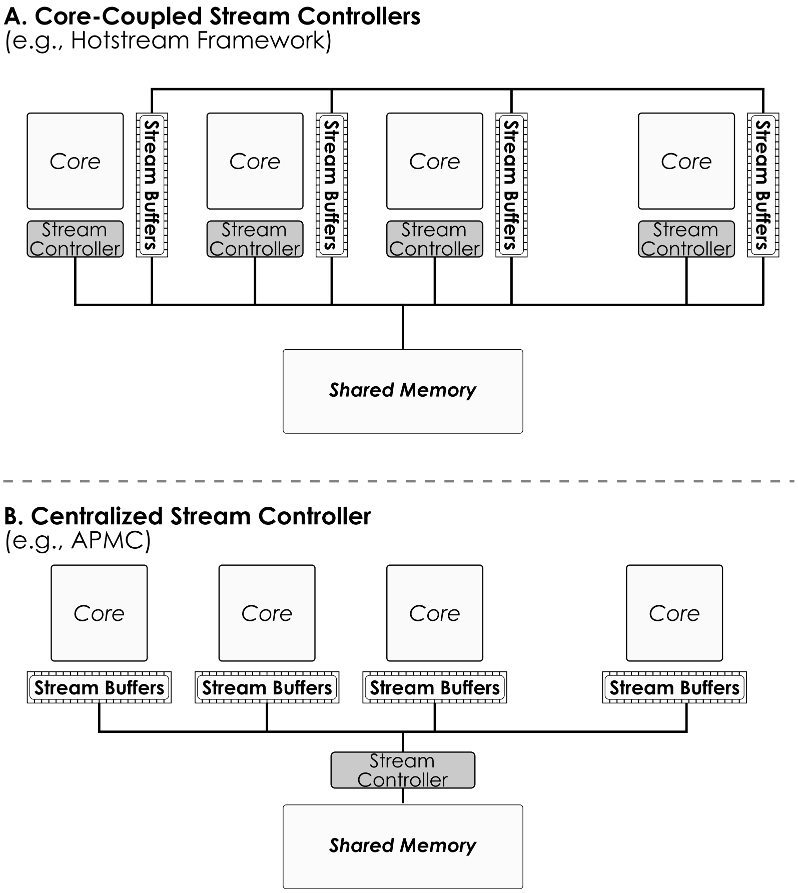


Figure 3.1: Stream controller topologies conventionally adopted by data streaming systems, with modules close to the system's processing cores (A - e.g., Hotstream Framework [26]) or close to the main memory (B - e.g., APMC [54]).

exploitation of alternative data communication schemes (e.g., broadcast data transfers) [34].

Independently of the adopted topology, stream controllers must rely on accurate representations of the data access pattern to perform the correct sequence of memory accesses and generate data streams for the system's processing cores [26, 34, 54]. However, existing representations have been limited to regular data patterns and, since compiler support is usually not provided, they require the programmer to manually describe the access sequence.

Accordingly, the data streaming scheme that is proposed in this dissertation (further discussed in Chapter 4) is based on the notion that an application memory access pattern can be explicitly extracted during compilation and used, at runtime, to generate data streams directly from memory. This is done by taking a step further from typical static analysis tools. In particular, while these tools have been used to analyze memory access patterns and perform code optimization at compile-time, there is an opportunity to extend this functionality by explicitly exposing and extracting the access pattern to provide support for data streaming. The extracted patterns can then be encoded and embedded in the application code, and subsequently sent to the on-chip stream controllers to autonomously perform the corresponding sequence of memory accesses, during execution.

Accordingly, this chapter presents and evaluates the developed compilation tools and techniques that give support to generic data stream communication schemes, namely:

- A **memory access description specification** based on an affine mathematical model. This specification is capable of representing arbitrarily complex deterministic access patterns and indirect memory accesses, to enable data stream communication schemes.
- A **compile-time static analysis tool** that examines a region of application source code to extract and encode its memory access sequence with the defined description specification.
- A **code transformation mechanism** that leverages the resulting detachment of memory accesses from computation, to perform code reductions and accelerate its execution. In particular, it replaces the array subscript indexation of each encoded memory access with a stream reference, eliminating redundant data indexation and address calculation.

The proposed compilation tool is validated through a preliminary experimental evaluation, by comparing the efficiency of the memory access pattern encoding with existing state-of-the-art representations, and by assessing its stream code transformation capabilities. The chapter is concluded with a brief discussion of the main advantages and limitations of the presented techniques.

3.1 Modeling of Complex Data-Patterns

The performance of any processing architecture is directly related to the characteristics of the applications that it executes. Based on such knowledge, computing frameworks (ranging from compilation tools to computing architectures) are built by recognizing the most common features found across a wide range of general-purpose applications. In particular, special attention is often given to the complexity of the data access sequence and memory access latency, as they can limit the throughput provided by the processor resources.

Accordingly, most compilation tools are designed to exploit specific features of the processing architectures (e.g., out-of-order execution) to hide the latency of such data accesses. For instance, they try to reorganize the instructions from the original code to allow concurrent data fetching and computing operations, effectively trying to hide the latency of accessing the memory behind the computation performed by the processor.

On the other hand, the typical architecture of a cache memory is based on a more in-depth insight of the characteristics of general data accesses. Specifically, caches have long been consolidated around the principle of data locality, which characterizes the behavior of an application memory access pattern. This principle is based on the notion that a given memory region (e.g., a set of contiguous memory locations - **spatial locality**) is frequently accessed (**temporal locality**). Hence, caches are designed by embracing the common characteristics found in general-purpose applications, where they typically assume that memory accesses maintain a regular pattern over time. Naturally, with that sole assumption as its basis, when the memory access pattern is characterized by poor data locality, this model falls short and data cannot be efficiently maintained close to the processor.

By following a locality principle similar to caches, prefetching solutions anticipate the loading of data in the cache lines by analyzing the most recently accessed addresses ($y_{current}$) and calculating the distance between them (*stride*), to predict the sequence of future accesses (up to a given *degree*). The most common solutions rely on a unidimensional model based on near-temporal and near-spatial locality:

$$y(degree) = y_{current} + degree \times stride \tag{3.1}$$

While this model is capable of predicting a wide range of memory access sequences, its unidimensional representation can result in inaccuracies when the complexity of the pattern of accessed addresses increases. Hence, to provide accurate representations with higher levels of complexity, a more in-depth model is required.

Accordingly, the proposed static analysis tool (further detailed in Section 3.2) is based on a formal mathematical model to capture and describe deterministic memory access patterns. This is done by using a representation of the exact sequence of addresses generated by the application, for a specific data access. The mathematical model is used to define a new memory access pattern descriptor specification that can be used to generate data streams for each data access. To do so, the proposed specification mirrors the typical for-loop encoding scheme for array indexing. Consequently, it allows capturing nested loop indexes and loop- or data-dependent (indirect) index dynamic ranges, where the above-mentioned models can fall short.

3.1.1 Affine Mathematical Model for Data Indexing

Independently of their application domain, many applications are characterized by data accesses with complex memory address patterns that can be represented by an *n*-dimensional (*n*-D) affine function [98] (or by a set of such functions). According to this principle, the following affine function can be used to describe any deterministic *n*-dimensional address sequence:

$$\begin{aligned} y(X) &= y_{base} + \sum_{k=0}^{dim_y} x_k \times stride_k \\ \textbf{with} \quad & x_k \in \left[\alpha_k,\ \beta_k\right], \\ & X = \{x_0, \ldots, x_{dim_y}\} \quad , \end{aligned} \tag{3.2}$$

where the memory address sequence $y(X)$ is generated by the sum of a base address (or offset) value y_{base}, with dim_y pairs of increment variables (or indexes) x_k and *stride*$_k$ multiplication factors. Each increment variable x_k is represented by a integer range, with limits α_k and β_k.

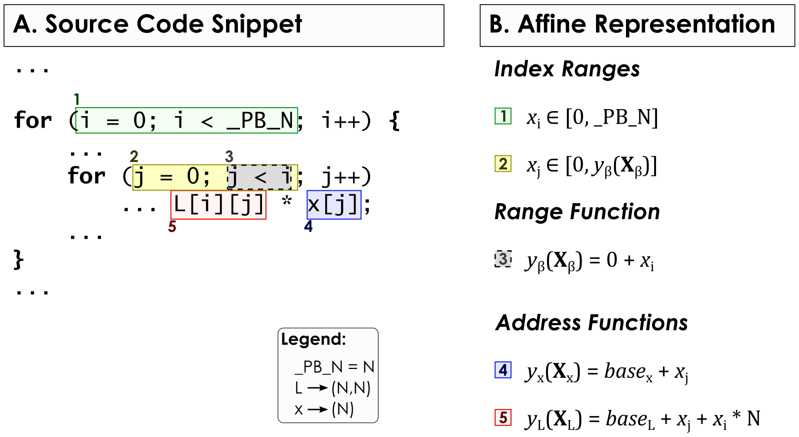
Since this representation, by itself, allows indexing a significant amount of deterministic access patterns, it has been commonly used in description specifications deployed by data-fetch controllers, including Direct Memory Access (DMA) engines, although typically restricted to 2-D patterns (dim_y=2).

3.1.1.A Multi-level Affine Representation

The *n*-dimensional model defined in Eq. 3.2 can be further extended to describe more complex patterns by performing the composition of multiple affine functions. This is done by determining the base address (y_{base}), the set of $stride_k$ factors and/or the upper and lower bounds (α_k and β_k) of each increment variable by means of another affine function (as depicted in the represented index ranges in Fig. 3.2). Accordingly, it is proposed the following extended model to describe the deterministic address sequence of a given memory access:

$$\begin{aligned} y(X_y) &= y_{base}(X_{y_{base}}) + \sum_{k=0}^{dim_y} x_k \times y_{stride_k}(X_{stride_k}) \\ \textbf{with} \quad & x_k \in \left[\alpha_k(X_{\alpha_k}),\ \beta_k(X_{\beta_k})\right], \\ & X_v = \{x_0, \ldots, x_{dim_v}\}, \\ & v \in \{y, y_{base}, stride_k, \alpha_k, \beta_k\} \quad , \end{aligned} \tag{3.3}$$

where the values calculated by each $y_v(X_v)$ function are used as parameters in the $y(X_y)$ function to calculate the final address sequence. Furthermore, since there is no limit on the number of composition levels nor on the dimensionality of each function, the complexity of the modeled deterministic address sequences is only limited by the representation languages that adopt this model. Naturally, the representation of higher levels of complexity may require larger memory storage space to store the representation and more complex address generation units, to decode the representation and calculate the correct sequence of memory addresses.



```
...
for (i = 0; i < _PB_N; i++) {
    ...
    for (j = 0; j < i; j++)
        ... L[i][j] * x[j];
    ...
}
...
```

Index Ranges

1 $x_i \in [0, \_PB\_N]$

2 $x_j \in [0, y_\beta(\mathbf{X}_\beta)]$

Range Function

3 $y_\beta(\mathbf{X}_\beta) = 0 + x_i$

Address Functions

4 $y_x(\mathbf{X}_x) = base_x + x_j$

5 $y_L(\mathbf{X}_L) = base_L + x_j + x_i * \mathrm{N}$

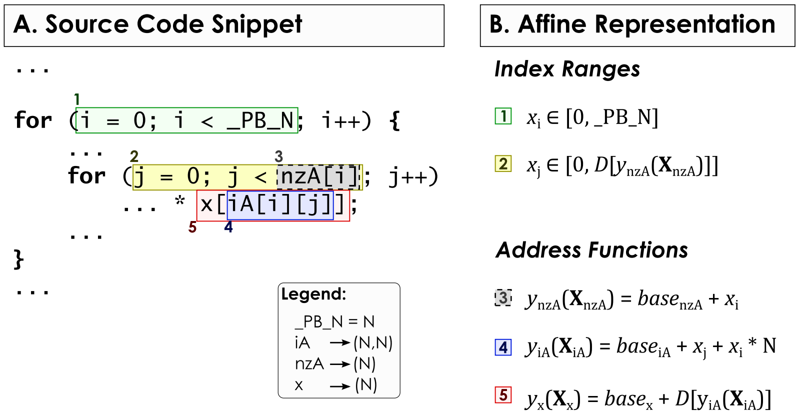
Figure 3.2: Example affine representation of the memory accesses in a code snippet of the `trisolv` benchmark, from the Polybench [99] suite.

3.1.1.B Memory Access Indirection Representation

Conventionally, affine representations are limited to deterministic memory address sequences and cannot be directly applied to patterns characterized by indirect memory accesses (i.e., in the format A[B[i]]). This is due to the fact that it is usually impossible to know (at compile-time) to what kind of pattern a particular indirect reference will lead to, since the access pattern depends on values in the indexed arrays, which are only known at runtime. While the proposed model can only represent exact deterministic patterns, by taking the composition of affine functions a step further, it is possible to derive an alternative representation of Eq. 3.3 to support indirect memory access patterns. In particular, instead of assigning the values calculated by each $y_v(X_v)$ function to the parameters of the $y(X_y)$ function, it is possible to use them to generate address sequences and then assign the corresponding data to the parameters (as depicted in Fig. 3.3):

$$\begin{aligned} y(X_y) = D\big[y_{base}(X_{y_{base}})\big] + \sum_{k=0}^{dim_y} x_k \times D\big[y_{stride_k}(X_{stride_k})\big] \\ \textbf{with} \quad x_k \in \big[D\big[\alpha_k(X_{\alpha_k})\big],\ D\big[\beta_k(X_{\beta_k})\big]\big], \\ X_v = \{x_0, \ldots, x_{dim_v}\}, \\ v \in \{y, y_{base}, stride_k, \alpha_k, \beta_k\} \quad , \end{aligned} \tag{3.4}$$

where each $D\big[y_v(X_v)\big]$ function represents the data values corresponding to the sequence of addresses generated by each $y_v(X_v)$ function. The data itself is used as the value assigned to a parameter in the $y(X_y)$ function to calculate the final address sequence (see Fig. 3.3). Accordingly, although a data-dependent memory access pattern can only be completely calculated at runtime, the represented relation can still enable a processor-independent data stream generation procedure. As a result, the derived model adds a previously inexistent support in data streaming systems to a wide range of irregular applications (such as graph computations and sparse linear algebra).



```
...
for (i = 0; i < _PB_N; i++) {
    ...
    for (j = 0; j < nzA[i]; j++)
        ... * x[iA[i][j]];
    ...
}
...
```

1 $x_i \in [0, \_PB\_N]$

2 $x_j \in [0, D[y_{nzA}(\mathbf{X}_{nzA})]]$

3 $y_{nzA}(\mathbf{X}_{nzA}) = base_{nzA} + x_i$

4 $y_{iA}(\mathbf{X}_{iA}) = base_{iA} + x_j + x_i * N$

5 $y_x(\mathbf{X}_x) = base_x + D[y_{iA}(\mathbf{X}_{iA})]$

Figure 3.3: Example affine representation of indirect memory accesses in a code snippet of the `spmv` kernel, adapted from the HPCG [100] benchmark.

3.1.2 Memory Access Description Specification

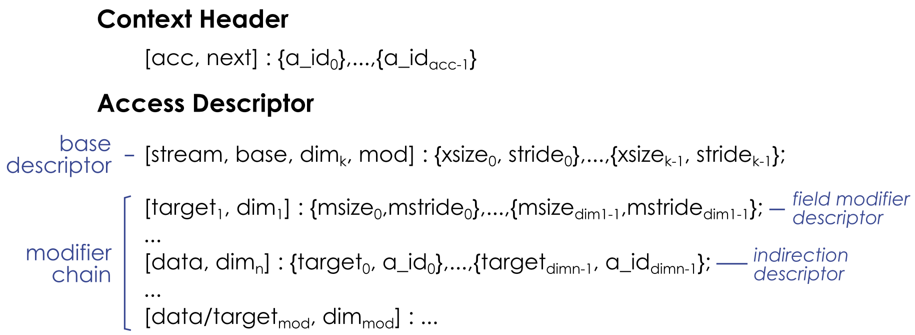
Having defined the previously-presented mathematical model, it is necessary to encode it with a representation that can be interpreted by the address generation modules of a stream-based system. To do so, it is only necessary to store the variables (parameters) and the interactions between affine functions, represented by Eqs. 3.3 and 3.4, in a predefined format that allows the system to perform an independent calculation of the required address sequences.

Hence, it is proposed a memory access *Context Descriptor* specification (presented in Fig. 3.4), which provides a for-loop context data access encoding and encapsulation. It is composed of a top-level *Context Header*, which indicates the number (`acc`) and memory locations ($\texttt{a\_id}_{acc}$) of a set of *Access Descriptors*, each describing one memory access pattern (represented by the $y(X_y)$ function in Eqs. 3.3 and 3.4). It also contains a reference to a subsequent *Context Header* (`next`), allowing multiple descriptor contexts to be described and solved in sequence.

3.1.2.A Base Descriptor Encoding

The *Access Descriptor* defines the data access pattern represented by a $y(X_y)$ function by means of: *i)* an header tuple, containing a stream pointer (`stream`), the array base address (`base`), the descriptor dimensionality ($\texttt{dim}_k$), and the number of modifier chains (`mod`) that represent the function compositions as in Eqs. 3.3 and 3.4 (see below); and *ii)* pairs of $\texttt{xsize}_k$ and $\texttt{stride}_k$ fields, representing the x_k range (in number of iterations) and *stride*$_k$, respectively.

The encoded $\{\texttt{xsize}_k, \texttt{stride}_k\}$ pairs have an implicit hierarchy where the rightmost pair has the higher position. This forces a fixed order of pair iterations to generate the correct sequence of memory addresses. Accordingly, each pair is fully iterated (once) for each instance of the pair in the upper position of the hierarchy level. The descriptor solving is further detailed in the following sections.



Context Header

[acc, next] : {a_id$_0$},...,{a_id$_{acc-1}$}

Access Descriptor

base descriptor – [stream, base, dim$_k$, mod] : {xsize$_0$, stride$_0$},...,{xsize$_{k-1}$, stride$_{k-1}$};

modifier chain

[target$_1$, dim$_1$] : {msize$_0$,mstride$_0$},...,{msize$_{dim1-1}$,mstride$_{dim1-1}$}; – field modifier descriptor

...

[data, dim$_n$] : {target$_0$, a_id$_0$},...,{target$_{dimn-1}$, a_id$_{dimn-1}$}; — indirection descriptor

...

[data/target$_{mod}$, dim$_{mod}$] : ...

Figure 3.4: Context Descriptor specification.

3.1.2.B Affine Composition and Data-Dependency Encoding

Since each x_k range and *stride*$_k$ can be delimited by the composition of multiple affine functions (see Eq. 3.3), the *Access Descriptor* provides an optional modifier chain, composed of multiple descriptors, in which the header indicates the target field ($\texttt{target}_{mod}$) to be modified (see Fig. 3.4). The modifier chain applies a field modifier descriptor to the target field every time the corresponding pair is completely iterated. As an added functionality, if multiple modifier descriptors target the field, they are applied in sequence, after each modifier descriptor is completely iterated. Accordingly, the *Access Descriptor* is only completely iterated after each modifier descriptor completes its own iteration.

Similarly, data dependencies (as represented by Eq. 3.4) between *Access Descriptors* can also be encoded with the modifier chain. This is done by creating a producer-consumer-like relation between *Access Descriptors* within a context, where the data obtained with the address sequence generated by a descriptor is used as a parameter in another descriptor.

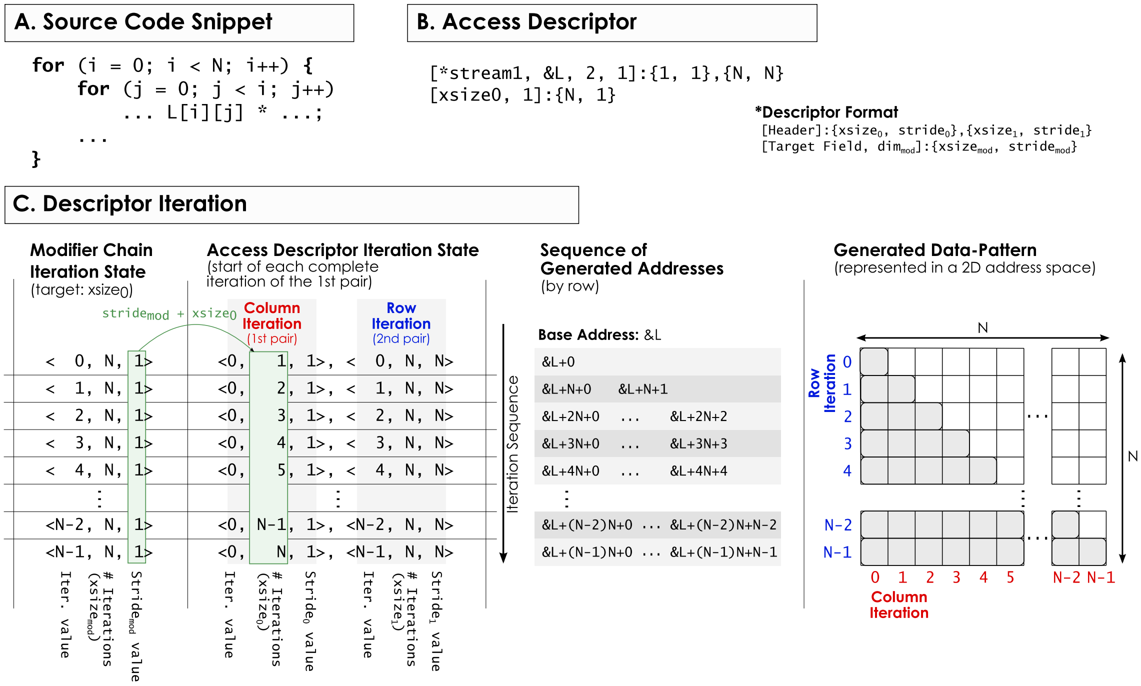
Accordingly the modifier chain provides an alternative indirection descriptor (encoded in the consumer descriptor), composed of pairs of target fields ($\texttt{target}_n$) and descriptor identifications ($\texttt{a\_id}_n$) (see Fig. 3.4). Hence, the indirection descriptor applies the encoded data dependencies similarly to a field modifier descriptor, where the actual data from the stream generated by a $\texttt{a\_id}_n$ descriptor is used to modify the $\texttt{target}_n$ field in the *Access Descriptor*.

3.1.2.C Descriptor Encoding and Resolving Examples

The following paragraphs detail two memory access pattern examples, illustrating the utilization of the *Access Descriptor* modifier chain for the encoding of deterministic patterns with higher levels of complexity. A pattern with indirect memory accesses is also illustrated.

Fig. 3.5 depicts the resolving procedure for an *Access Descriptor* encoding a triangular N×N matrix access pattern. The base descriptor (see Fig. 3.5.B) is composed of two $\{\texttt{xsize, stride}\}$ pairs, where the first ($\{\texttt{1,1}\}$) describes contiguous row accesses (initially with size 1) and the second ($\{\texttt{N,N}\}$) applies a stride (N - matrix width) to skip to the next row (N times). The descriptor is also encoded with a modifier chain comprising a single field modifier descriptor - [xsize0]:{N,1}. It works by adding its own stride value to the xsize field of the first pair in the base descriptor each time it is completely iterated (see Fig. 3.5.C). As a result, each time the base descriptor iterates to a new row, the number of contiguous accesses is increased by 1, hence producing a triangular scan pattern (see Fig. 3.5.C - right).

In Fig. 3.6 it is depicted the resolving procedure for a set of *Access Descriptors* encoding a simple indirect memory access pattern, where the data obtained by accessing an *y* array is used to subsequently index an *x* array. Accordingly, the encoding is composed of two descriptors, both composed of a single {xsize, stride} pair, to represent the accesses to the *x* and *y* arrays (see Fig. 3.6.B). The accesses to *y* (a_1 descriptor) are encoded by a single pair ({N,1}), that is used to generate a contiguous sequence (stride=1) of N addresses. On the other hand, the accesses to *x* (a_2 descriptor) are encoded with a dummy pair ({-,1}) and a modifier chain comprising an indirection descriptor - [data]:{xsize0,a1}. This indirection descriptor is used to represent a data dependency between the xsize0 field of the a_2 base descriptor and the data obtained from the addresses generated by descriptor a_1. As a result, each time data is obtained from array *y* (through descriptor a_1), the corresponding value is used by descriptor a_2 as an offset to calculate a single address, hence producing an irregular scan pattern of array *x* (see Fig. 3.6.B-right).


A. Source Code Snippet

```
for (i = 0; i < N; i++) {
    for (j = 0; j < i; j++)
        ... L[i][j] * ...;
    ...
}
```

B. Access Descriptor

```
[*stream1, &L, 2, 1]:{1, 1},{N, N}
[xsize0, 1]:{N, 1}
```

*Descriptor Format

```
[Header]:{xsize0, stride0},{xsize1, stride1}
[Target Field, dimmod]:{xsizemod, stridemod}
```

C. Descriptor Iteration

Modifier Chain Iteration State (target: xsize0)

Access Descriptor Iteration State (start of each complete iteration of the 1st pair)

```
<  0, N, 1>     <0,   1, 1>, <  0, N, N>
<  1, N, 1>     <0,   2, 1>, <  1, N, N>
<  2, N, 1>     <0,   3, 1>, <  2, N, N>
<  3, N, 1>     <0,   4, 1>, <  3, N, N>
<  4, N, 1>     <0,   5, 1>, <  4, N, N>
     ⋮                     ⋮
<N-2, N, 1>     <0, N-1, 1>, <N-2, N, N>
<N-1, N, 1>     <0,   N, 1>, <N-1, N, N>
```

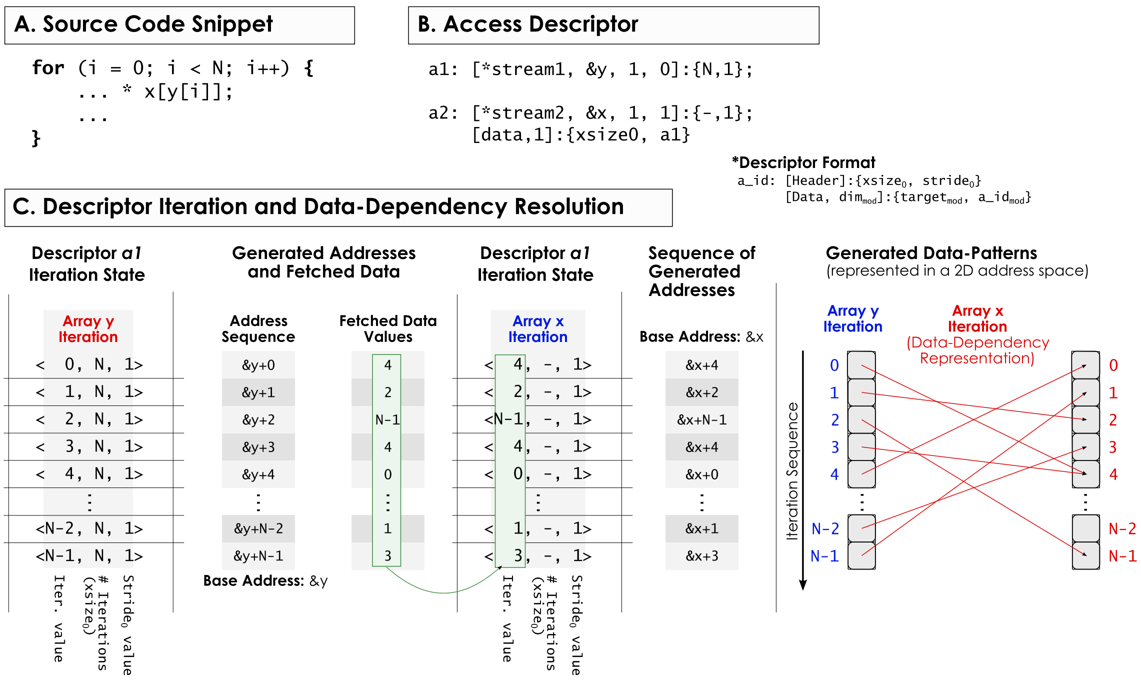
Sequence of Generated Addresses (by row)

Base Address: &L

```
&L+0
&L+N+0       &L+N+1
&L+2N+0      ...      &L+2N+2
&L+3N+0      ...      &L+3N+3
&L+4N+0      ...      &L+4N+4
  ⋮
&L+(N-2)N+0  ...  &L+(N-2)N+N-2
&L+(N-1)N+0  ...  &L+(N-1)N+N-1
```

Generated Data-Pattern (represented in a 2D address space)

Figure 3.5: *Access Descriptor* encoding a triangular matrix access. Notice how the modifier chain increases the number of accessed matrix columns after each full iteration of the first pair in the base descriptor, while the second pair adds a stride value to a subsequent matrix row.



A. Source Code Snippet

```
for (i = 0; i < N; i++) {
    ... * x[y[i]];
    ...
}
```

B. Access Descriptor

```
a1: [*stream1, &y, 1, 0]:{N,1};

a2: [*stream2, &x, 1, 1]:{-,1};
    [data,1]:{xsize0, a1}
```

*Descriptor Format

```
a_id: [Header]:{xsize0, stride0}
      [Data, dimmod]:{targetmod, a_idmod}
```

C. Descriptor Iteration and Data-Dependency Resolution

Descriptor a1 Iteration State Array y Iteration	Address Sequence	Fetched Data Values	Descriptor a1 Iteration State Array x Iteration	Sequence of Generated Addresses
< 0, N, 1>	&y+0	4	< 4, -, 1>	&x+4
< 1, N, 1>	&y+1	2	< 2, -, 1>	&x+2
< 2, N, 1>	&y+2	N-1	<N-1, -, 1>	&x+N-1
< 3, N, 1>	&y+3	4	< 4, -, 1>	&x+4
< 4, N, 1>	&y+4	0	< 0, -, 1>	&x+0
⋮	⋮	⋮	⋮	⋮
<N-2, N, 1>	&y+N-2	1	< 1, -, 1>	&x+1
<N-1, N, 1>	&y+N-1	3	< 3, -, 1>	&x+3

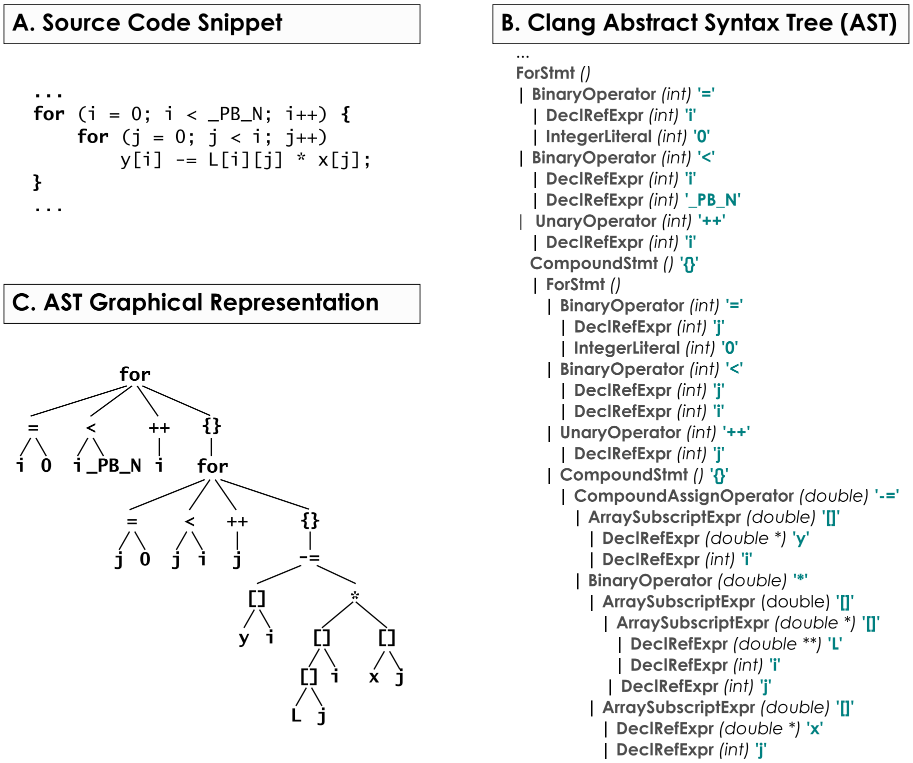
Figure 3.6: *Access Descriptor* encoding an indirect memory indexation. Notice how the data obtained by solving the descriptor corresponding to array *y* is used as a parameter to calculate the sequence of addresses of the descriptor corresponding to array *x*.

3.2 Compile-Time Stream Code Generation

Any stream-based communication scheme must rely on some kind of dedicated memory access representation. However, the utilization of such a dedicated approach imposes limitations to the applicability of data-streaming to general purpose contexts. This is mainly due to the fact that each data stream must be manually coded through specific language interfaces [26, 34, 54], resulting from the lack of compiler support to automatically encode data streams and embed them in application code.

To tackle such a limitation, it is proposed the extension of conventional static analysis tools, commonly deployed in compilers. In general, these tools are used to analyze the application code and perform optimizations according to computational and memory access dependencies, with the goal of promoting instruction- and data-level parallelism during the code's execution. During this optimization process, all the information regarding the application memory access pattern (including indexing variable dependencies, data structure dimensionality, and memory address calculation) is completely exposed to the compilation tool through a syntactic representation, usually named Abstract Syntax Tree (AST).

The AST of an application is typically generated by the compiler front-end. Initially, the code is passed through a preprocessor, which expands all the language macros and performs lexical analysis. Then, a parser breaks-up the code in its individual syntactic elements and generates the corresponding AST (see Fig. 3.7). The compiler then uses this structure for multiple purposes,



A. Source Code Snippet

```
...
for (i = 0; i < _PB_N; i++) {
    for (j = 0; j < i; j++)
        y[i] -= L[i][j] * x[j];
}
...
```

C. AST Graphical Representation

B. Clang Abstract Syntax Tree (AST)

```
...
ForStmt ()
| BinaryOperator (int) '='
  | DeclRefExpr (int) 'i'
  | IntegerLiteral (int) '0'
| BinaryOperator (int) '<'
  | DeclRefExpr (int) 'i'
  | DeclRefExpr (int) '_PB_N'
|  UnaryOperator (int) '++'
  | DeclRefExpr (int) 'i'
  CompoundStmt () '{}'
  | ForStmt ()
    | BinaryOperator (int) '='
      | DeclRefExpr (int) 'j'
      | IntegerLiteral (int) '0'
    | BinaryOperator (int) '<'
      | DeclRefExpr (int) 'j'
      | DeclRefExpr (int) 'i'
    | UnaryOperator (int) '++'
      | DeclRefExpr (int) 'j'
    | CompoundStmt () '{}'
      | CompoundAssignOperator (double) '-='
        | ArraySubscriptExpr (double) '[]'
          | DeclRefExpr (double *) 'y'
          | DeclRefExpr (int) 'i'
        | BinaryOperator (double) '*'
          | ArraySubscriptExpr (double) '[]'
            | ArraySubscriptExpr (double *) '[]'
              | DeclRefExpr (double **) 'L'
              | DeclRefExpr (int) 'i'
            | DeclRefExpr (int) 'j'
          | ArraySubscriptExpr (double) '[]'
            | DeclRefExpr (double *) 'x'
            | DeclRefExpr (int) 'j'
```

Figure 3.7: Example of an AST representation generated by the Clang front-end, from the LLVM compiler [30].

such as creating symbol tables, performing type checking and generating machine code.

While most analysis and optimization procedures are performed in the mid and later stages of the compilation chain, there is an opportunity to perform an earlier in-depth analysis of the memory accesses through the AST. In particular, it is possible to extract them and perform their encoding with the proposed representation. Moreover, since the AST is an exact representation of the application source code, it is also possible to deploy code transformation mechanisms to automatically inject data streaming directives.

3.2.1 Compiler Module Overview

To explore the described features of the AST representation, the proposed compilation tool (depicted in Fig. 3.8) was built on the Clang LibTooling library [101], from the LLVM [30] framework. This framework was selected not only because it has been increasingly adopted by the scientific community, but also due to the available amount of open-source and community-based compilation tools. In particular, the LibTooling library provides a C++ interface that provides full control over the compiler's front-end resources. Naturally, since the envisaged tool is proposed to take action at the level of the compiler AST, it could equally be implemented in any compilation toolchain that makes use of this type of representation.

Accordingly, the conceived compilation method operates in five consecutive phases: *i)* target

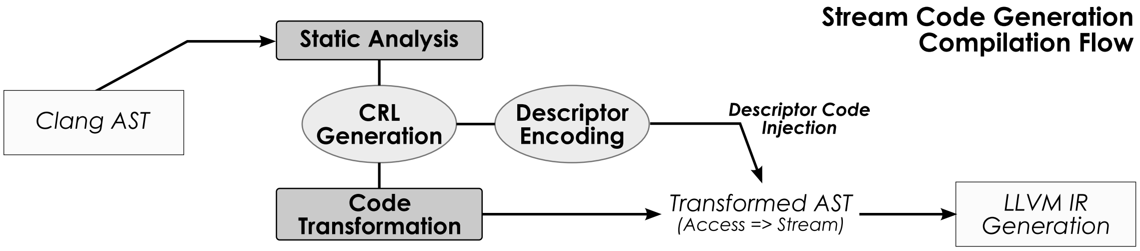


Figure 3.8: Overview of the compilation tool for data streaming methods, comprising code analysis and transformation modules.

code region extraction; *ii)* code translation to AST; *iii)* AST static analysis and translation to an intermediate memory access representation; *iv)* AST code transformation to data streaming; and *v) Context Descriptor* generation and code injection. Fig. 3.9 depicts an example illustrating all the code transformations and translations performed between representations.

In the first stage, the tool makes use of a code annotation scheme based on typical pragma directives. Hence, by making use of the pragma handling routines from the LibTooling library, it is possible to select any region-of-interest in the source code to be analyzed, by delimiting it with the pragma directive (see Fig. 3.9.A):

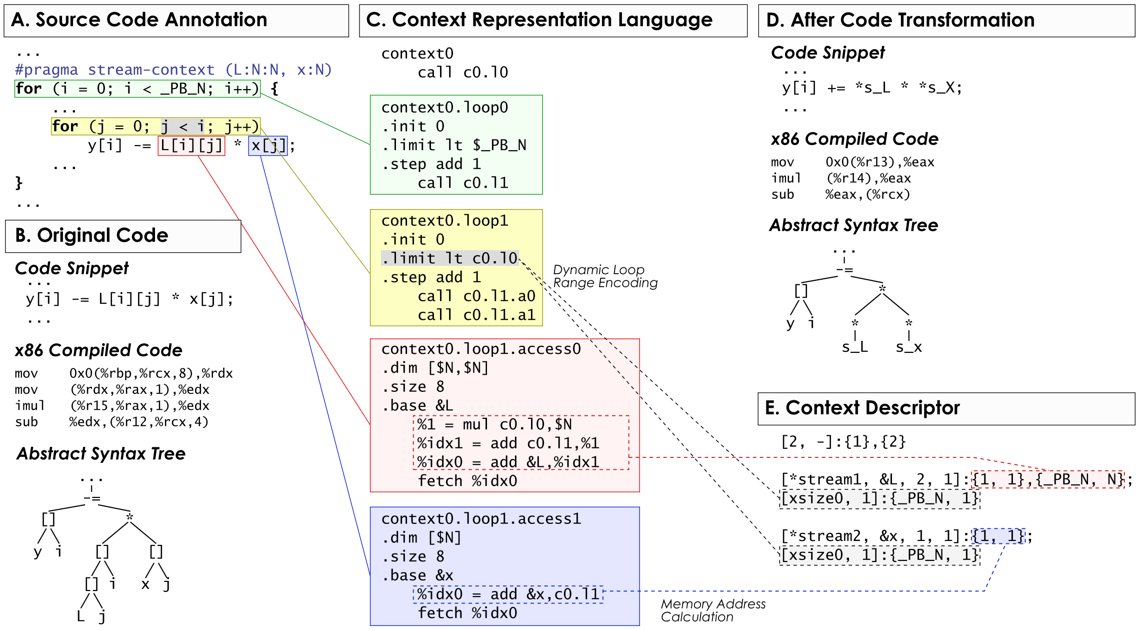
```
#pragma stream-context (var:size[:size], ...)
```

With such an interface, the tool can be configured and made aware of the target array variables (and their size, per dimension) for data stream description.

After being identified, the selected code is transformed into a Clang translation unit, which is passed to the front-end tools to generate the corresponding AST (see Fig. 3.9.B bottom). At this point, the compilation flow slightly diverges from the typical compiler, which would proceed to generate LLVM Intermediate Representation (IR) code, followed by compiled machine code.

Although the proposed compilation tool could be implemented as an LLVM IR optimization pass, the Clang AST provides a much more exposed representation of the loop context for a given data access, and it is implemented by a *n*-ary tree data structure. This allows a straightforward code parsing and analysis with known tree scanning algorithms. Moreover, it allows the implementation of simple application code manipulation procedures through tree node injection before generating the IR code.

Hence, contrasting to LLVM, the tool generates its own internal memory access high-level representation, through a dedicated Context Representation Language (CRL), further described in Section 3.2.2 (see also Fig. 3.9.C). This representation is generated through a direct translation of the region-of-interest source code, by parsing the Clang AST with a typical depth-first tree analysis, which will be later used as a reference to encode the memory access pattern of each access. Hence, the context of each access is translated to CRL by gathering all its information, including all the dependencies for the address generation, such as nested loop context (providing



```
...
#pragma stream-context (L:N:N, x:N)
for (i = 0; i < _PB_N; i++) {
    ...
    for (j = 0; j < i; j++)
        y[i] -= L[i][j] * x[j];
    ...
}
...
```

```
...
y[i] -= L[i][j] * x[j];
...
```

```
mov     0x0(%rbp,%rcx,8),%rdx
mov     (%rdx,%rax,1),%edx
imul    (%r15,%rax,1),%edx
sub     %edx,(%r12,%rcx,4)
```

```
context0
    call c0.l0

context0.loop0
.init 0
.limit lt $_PB_N
.step add 1
    call c0.l1

context0.loop1
.init 0
.limit lt c0.l0
.step add 1
    call c0.l1.a0
    call c0.l1.a1

context0.loop1.access0
.dim [$N,$N]
.size 8
.base &L
    %1 = mul c0.l0,$N
    %idx1 = add c0.l1,%1
    %idx0 = add &L,%idx1
    fetch %idx0

context0.loop1.access1
.dim [$N]
.size 8
.base &x
    %idx0 = add &x,c0.l1
    fetch %idx0
```

```
...
y[i] += *s_L * *s_X;
...
```

```
mov     0x0(%r13),%eax
imul    (%r14),%eax
sub     %eax,(%rcx)
```

```
[2, -]:{1},{2}

[*stream1, &L, 2, 1]:{1, 1},{_PB_N, N};
[xsize0, 1]:{_PB_N, 1}

[*stream2, &x, 1, 1]:{1, 1};
[xsize0, 1]:{_PB_N, 1}
```

Figure 3.9: Depiction of the several translations and code transformations performed by the proposed compilation tool, for a code snippet of the `trisolv` benchmark, from the Polybench [99] suite.

temporal information), indexing ranges and data dimensionality (for address calculation) and data-dependent access hierarchies (for indirect memory access representation).

After parsing the initial AST, it is performed a transformation pass that modifies the AST subtree of each extracted memory access into a data stream access (see Fig. 3.9.D and Section 3.2.3). This is possible because the memory address sequence of each access will be encoded and subsequently handled by on-chip stream generation modules, making the corresponding application code redundant. Such a transformation results in an explicit detachment of the memory addressing generation procedure from the computational operations in the application source code and a consequent reduction in the number of instructions per loop. These transformations are performed by creating an array with a stream reference per extracted access, and by transforming the usual array n-dimensional indexation (`idx`) with a single access to a stream reference (see Fig. 3.9.D), such as:

```
<array_name>[idx0]..[idxN] → *stream_<name>
```

The addressing sequence of each memory access in the generated CRL is then translated to the proposed *Context Descriptor* specification and tagged with the stream address corresponding to each memory access. This translation is performed by extracting the access pattern from the CRL and by encoding the descriptor's parameters that will allow the resolution of the address sequence of each memory access (see Fig. 3.9.E).

Finally, the encoded *Context Descriptors* are embedded in the original code by injecting a set of data structures and initialization directives. This solution allows information that can only be obtained at runtime (such as the base address of the described array variables) to be sent to (or captured by) dedicated on-chip data streaming modules (later discussed in Chapter 4). From this point on, the control is returned to the LLVM compiler and the typical compilation flow is resumed.

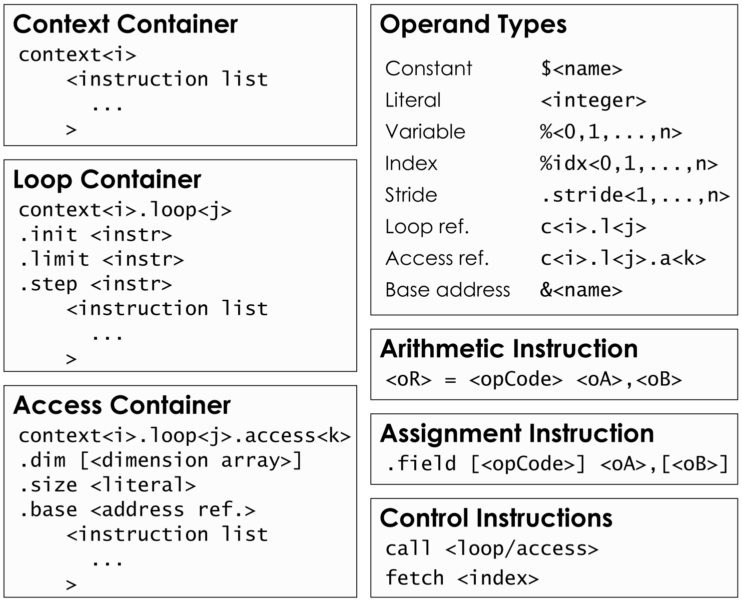
3.2.2 Context Representation Language

One of the main features of the implemented static analysis tool is the developed Context Representation Language (CRL), specifically designed to gather and represent the information that is required to calculate the address sequence of a given memory access. The language is built via a direct translation of the Clang AST (detailed below). It is composed of a basic instruction set and three container structures: *context*, *loop* and *access* (see Fig. 3.10). Each container is composed of a unique identification, a header with a set of configuration fields (specific to each container type), and a body with a list of instructions. An example of a CRL representation is depicted in Fig. 3.9.C.

3.2.2.A Context, Loop and Access Containers

The *context container* is the top level container of the CRL. It serves as the entry point for the encoding and calls the top-level *loop containers*. Hence, a *context container* encapsulates all the information concerning the indexing variables, base addresses, dimensionality and size of the corresponding array variables within a nested loop. This information allows strides defined in different dimensions to be inferred and multiplied by indexing variables, in order to calculate each sequence address based on the model defined in Eqs. 3.3 and 3.4.

All the memory accesses to a given array variable (in the same context) are encapsulated in



Context Container

```
context<i>
    <instruction list
      ...
    >
```

Loop Container

```
context<i>.loop<j>
.init <instr>
.limit <instr>
.step <instr>
    <instruction list
      ...
    >
```

Access Container

```
context<i>.loop<j>.access<k>
.dim [<dimension array>]
.size <literal>
.base <address ref.>
    <instruction list
      ...
    >
```

Operand Types

Constant	`$<name>`
Literal	`<integer>`
Variable	`%<0,1,...,n>`
Index	`%idx<0,1,...,n>`
Stride	`.stride<1,...,n>`
Loop ref.	`c<i>.l<j>`
Access ref.	`c<i>.l<j>.a<k>`
Base address	`&<name>`

Arithmetic Instruction

```
<oR> = <opCode> <oA>,<oB>
```

Assignment Instruction

```
.field [<opCode>] <oA>,[<oB>]
```

Control Instructions

```
call <loop/access>
fetch <index>
```

Figure 3.10: Context Representation Language reference.

an *access container*. Before each access is generated by a `fetch` instruction, it is preceded by a set of arithmetic instructions that describe the operations required to calculate its address. Since each memory access sequence is modeled through an affine function of the base address, index and stride, each of these variables is individually expressed as a function of the current context (see Fig. 3.9.C).

Accordingly, the *access container* configuration header gathers all the necessary information about the considered array in three fields (see Fig. 3.10): *i)* a dimension vector (`.dim`), that stores the size of each dimension of the array, either indicated by integer values or named constants; *ii)* the size of the array data type (`.type`); and *iii)* the array base address (`.base`), indicated by an address reference.

Finally, each loop in the represented region of code is encoded by a *loop container* (see Fig. 3.10). The container body encapsulates `call` control instructions to the lower level loop and access containers and any other necessary arithmetic instructions. In the particular case when a field cannot be described by a single assignment instruction, additional instructions are added to the upper-level container before the corresponding call. The container also provides a header that stores the range of the loop iteration variable (typically used as an indexing variable for array structures). It includes: *i)* the variable initialization (`.init`); *ii)* the iteration limit value (`.limit`); and *iii)* the iteration step (`.step`). With such an encoding, the loop iteration variable is completely embedded in the container. Thus, when using the variable in a subsequent instruction (e.g., for a memory address calculation), the corresponding operand is, itself, a reference to the loop container (`c<i>.l<j>`).

3.2.2.B Instruction Set

To implement all the required calculations and control directives, the developed CRL provides a simple instruction set, divided in three instruction types: *arithmetic*, *assignment* and *control*. Depending on their type, these instructions support up to three operands, as depicted in Fig. 3.10, where it is also shown a summary of the supported operands.

The considered set of arithmetic instructions provides a significant range of three-operand integer operations. Each instruction is encoded in a `oR = opCode oA,oB` format, where `oR` is the destination operand, `oA` and `oB` are the source operands and `opCode` is the arithmetic operation.

To facilitate the representation of the initialization, range/conditions and step parameters typically coded in loop headers, a set of assignment instructions were also included to enable a composed configuration of container fields. As such, they are encoded with one or two source operands and an optional operation code (representing an integer arithmetic operation or an inequality binary operator), in the format `.field [opCode] oA[,oB]`, where `.field` is the container field. The container fields also provide a convenient encoding for the initialization, condition, and step of each iteration variable, supporting the assignment of literal values, variables, conditions, and instructions, to each field.

Finally, the control flow between containers and memory access operations is encoded with control instructions. The encoding is provided in the formats `call <ref>` and `fetch %idx<n>`, respectively, where `<ref>` is either a loop container reference (`c<i>.l<j>`) or an access container reference (`c<i>.l<j>.a<k>`), and `%idx<n>` is an index variable (see Fig. 3.10).

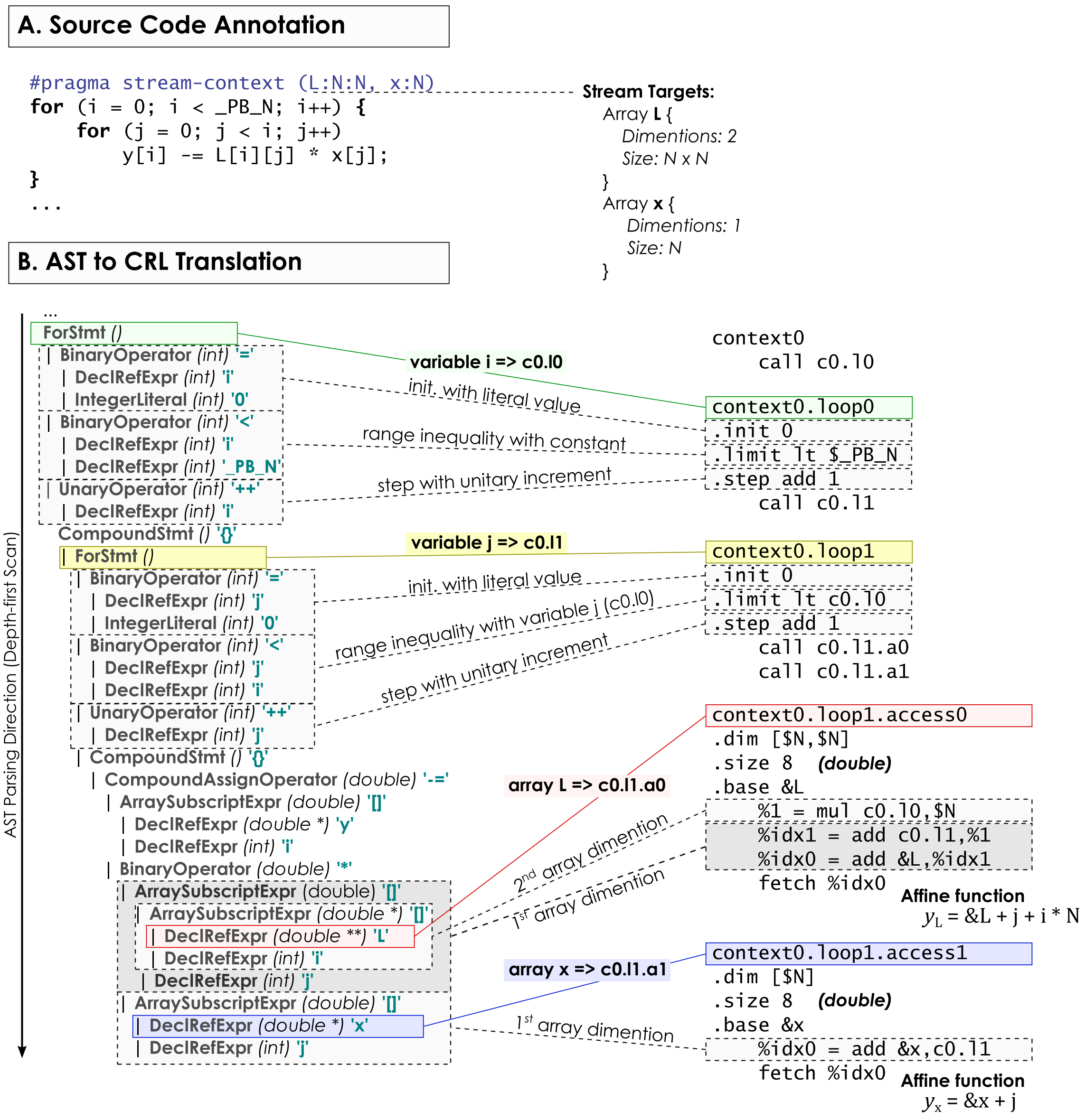
3.2.2.C Abstract Syntax Tree translation to Context Representation Language

The proposed compiler tool triggers the generation of a CRL representation upon the detection of a `#pragma stream-context` macro in the source code. At this point, the list of target array names, dimensionality, and sizes is extracted from the macro and stored (see Fig. 3.11.A). After the front-end generates its AST representation, the proposed tool performs a depth-first scan of the sub-tree corresponding to the annotated region of code to perform its translation to CRL. As depicted in Fig. 3.11, CRL components are generated as they are read during the scan, starting by the creation of a *context container* element to encapsulate the full representation.

When a for-loop statement is detected in the AST, a new *loop container* is created, and its corresponding iteration variable name is stored. The container's `.init`, `.limit` and `.step` header fields are filled in by interpreting the three subsequent sub-trees, corresponding to the initialization, stopping condition and increment expressions of the for-loop statement. Next, a `call` control instruction to the new container is inserted in the upper-level container corresponding to the context where the for-loop statement was detected (see the nested loop translation depicted in Fig. 3.11.B). The control instruction is preceded by additional instructions that may be required to represent the expressions assigned to the new container's header fields.

Upon the detection of a targeted array name, a new *access container* is created and a corresponding *call* instruction inserted in the upper-level container. The new container's `.dim` and `.size` header fields are initialized with the information extracted from the initial pragma directive and the `.type` field is assigned with the size (in bytes) corresponding to the array's data type (exposed in the AST node). Next, a set of instructions is inserted to encode the address generation for the array access. This is done by adding the stride of each array dimension (k) (implicitly defined by $stride_k = dim_1 \times \cdots \times dim_{k-1}$, with $stride_1 = 1$ and dim_k the width of the array in the corresponding dimension) to its assigned iteration value, represented by a *loop container* reference (see the accesses translation depicted in Fig. 3.11.B)). The generated set of instructions is then followed by a `fetch` instruction to represent the actual memory access. Naturally, multiple accesses to the same array variable can be scattered across the selected region of code. To simplify the translation, accesses that are performed (to one array variable) in the same loop context are grouped in the same *access container*, while accesses in different contexts require the generation of new containers.

The CRL translation stops when the sub-tree corresponding to the annotated region of code



A. Source Code Annotation

```
#pragma stream-context (L:N:N, x:N)
for (i = 0; i < _PB_N; i++) {
    for (j = 0; j < i; j++)
        y[i] -= L[i][j] * x[j];
}
...
```

```
Stream Targets:
  Array L {
    Dimentions: 2
    Size: N x N
  }
  Array x {
    Dimentions: 1
    Size: N
  }
```

B. AST to CRL Translation

```
...
ForStmt ()
| BinaryOperator (int) '='
  | DeclRefExpr (int) 'i'
  | IntegerLiteral (int) '0'
| BinaryOperator (int) '<'
  | DeclRefExpr (int) 'i'
  | DeclRefExpr (int) '_PB_N'
| UnaryOperator (int) '++'
  | DeclRefExpr (int) 'i'
  CompoundStmt () '{}'
  | ForStmt ()
    | BinaryOperator (int) '='
      | DeclRefExpr (int) 'j'
      | IntegerLiteral (int) '0'
    | BinaryOperator (int) '<'
      | DeclRefExpr (int) 'j'
      | DeclRefExpr (int) 'i'
    | UnaryOperator (int) '++'
      | DeclRefExpr (int) 'j'
    | CompoundStmt () '{}'
      | CompoundAssignOperator (double) '-='
        | ArraySubscriptExpr (double) '[]'
          | DeclRefExpr (double *) 'y'
          | DeclRefExpr (int) 'i'
        | BinaryOperator (double) '*'
          | ArraySubscriptExpr (double) '[]'
            | ArraySubscriptExpr (double *) '[]'
              | DeclRefExpr (double **) 'L'
              | DeclRefExpr (int) 'i'
            | DeclRefExpr (int) 'j'
          | ArraySubscriptExpr (double) '[]'
            | DeclRefExpr (double *) 'x'
            | DeclRefExpr (int) 'j'
```

```
context0
    call c0.l0
context0.loop0
.init 0
.limit lt $_PB_N
.step add 1
    call c0.l1
context0.loop1
.init 0
.limit lt c0.l0
.step add 1
    call c0.l1.a0
    call c0.l1.a1
context0.loop1.access0
.dim [$N,$N]
.size 8  (double)
.base &L
    %1 = mul c0.l0,$N
    %idx1 = add c0.l1,%1
    %idx0 = add &L,%idx1
    fetch %idx0
context0.loop1.access1
.dim [$N]
.size 8  (double)
.base &x
    %idx0 = add &x,c0.l1
    fetch %idx0
```

Affine function
$y_L = \&L + j + i * N$

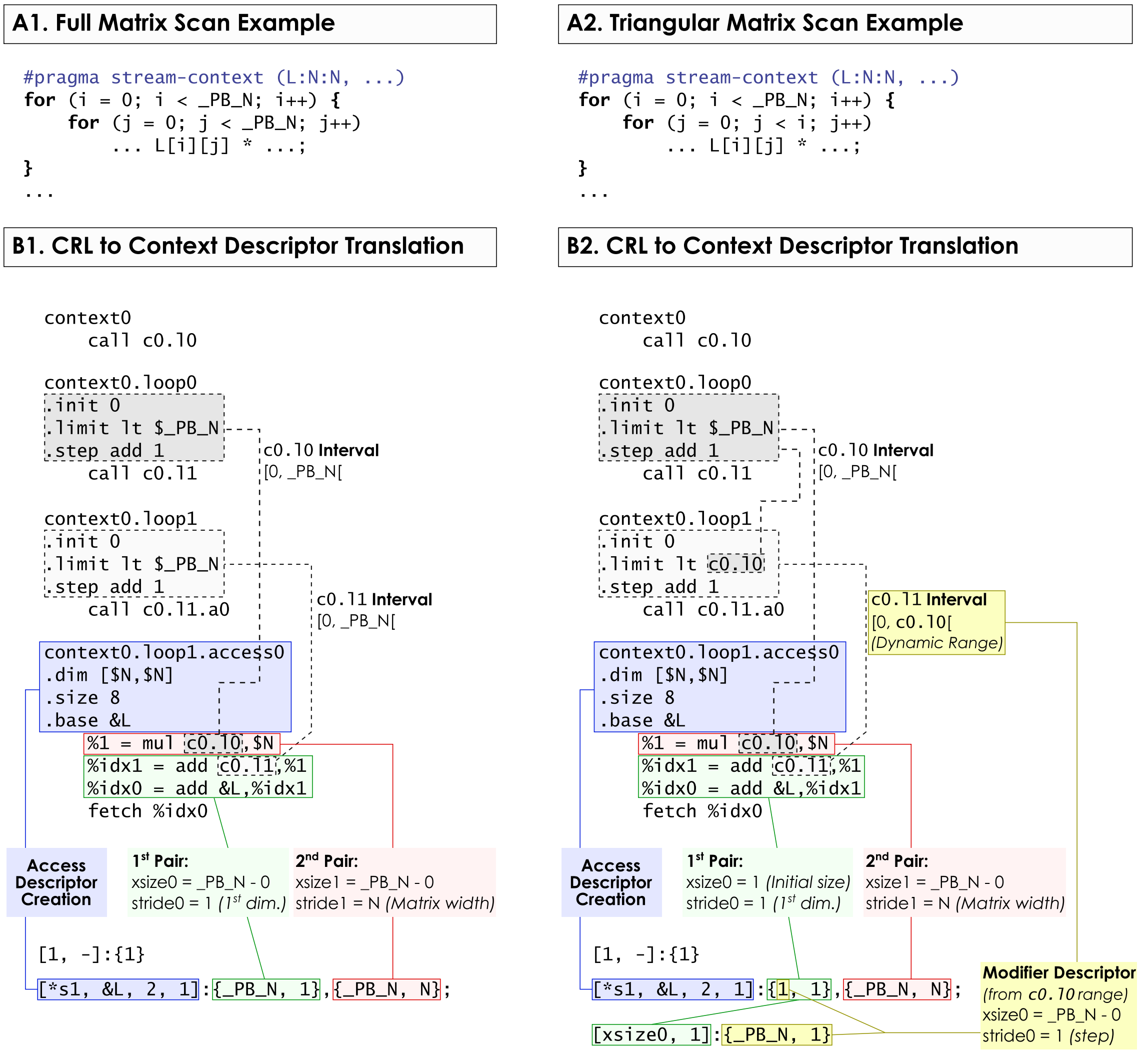
Affine function
$y_x = \&x + j$

Figure 3.11: Example of an AST translation to CRL, for a code snippet of the trisolv benchmark, from the Polybench [99] suite.

is completely scanned. At this point, the compiler tool proceeds to the generation of stream code, through the translation of the CRL to the proposed descriptor specification.

3.2.3 Stream Code Generation

The proposed compilation tool leverages the information that was gathered in the CRL to perform the final stream code generation step. It comprises: *i)* the translation of the CRL into the devised descriptor specification; and *ii)* appropriate code transformations to inject the encoded descriptors and eliminate redundant data indexation.



A1. Full Matrix Scan Example

```
#pragma stream-context (L:N:N, ...)
for (i = 0; i < _PB_N; i++) {
    for (j = 0; j < _PB_N; j++)
        ... L[i][j] * ...;
}
...
```

A2. Triangular Matrix Scan Example

```
#pragma stream-context (L:N:N, ...)
for (i = 0; i < _PB_N; i++) {
    for (j = 0; j < i; j++)
        ... L[i][j] * ...;
}
...
```

B1. CRL to Context Descriptor Translation

```
context0
    call c0.10

context0.loop0
.init 0
.limit lt $_PB_N
.step add 1
    call c0.11

context0.loop1
.init 0
.limit lt $_PB_N
.step add 1
    call c0.11.a0

context0.loop1.access0
.dim [$N,$N]
.size 8
.base &L
    %1 = mul c0.10,$N
    %idx1 = add c0.11,%1
    %idx0 = add &L,%idx1
    fetch %idx0

[1, -]:{1}
[*s1, &L, 2, 1]:{_PB_N, 1},{_PB_N, N};
```

B2. CRL to Context Descriptor Translation

```
context0
    call c0.10

context0.loop0
.init 0
.limit lt $_PB_N
.step add 1
    call c0.11

context0.loop1
.init 0
.limit lt c0.10
.step add 1
    call c0.11.a0

context0.loop1.access0
.dim [$N,$N]
.size 8
.base &L
    %1 = mul c0.10,$N
    %idx1 = add c0.11,%1
    %idx0 = add &L,%idx1
    fetch %idx0

[1, -]:{1}
[*s1, &L, 2, 1]:{1, 1},{_PB_N, N};
[xsize0, 1]:{_PB_N, 1}
```

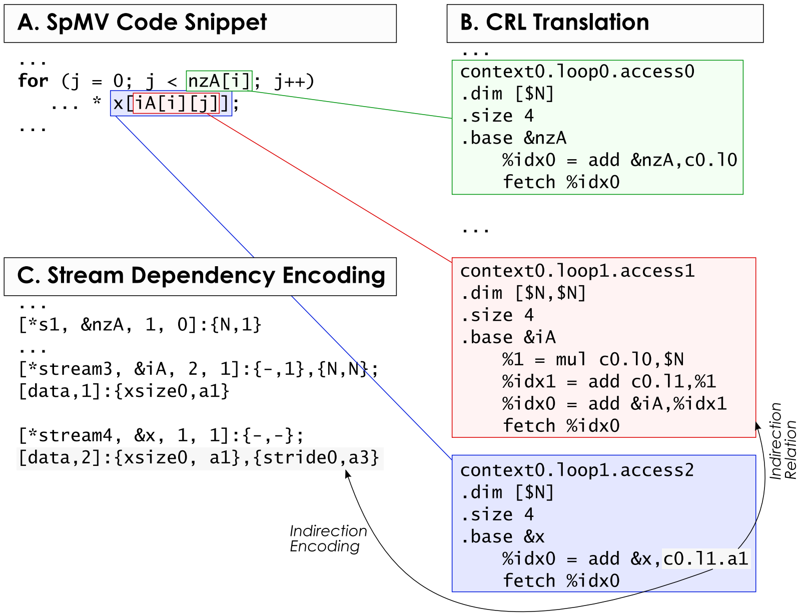
Figure 3.12: CRL to descriptor representation translation by considering two examples depicting: a full matrix scan (A1, B1) and a triangular matrix scan (A2, B2). In the second example (A2, B2), the dependency between the range of the inner loop and the iteration of the outer loop is encoded with a modifier chain, for the `xsize0` parameter of the *Access Descriptor*.

3.2.3.A Context Representation Language to Descriptor Representation Translation

The defined CRL encapsulates all the necessary information to generate the data stream encoding with the proposed descriptor specification. Accordingly, each descriptor is encoded by parsing the CRL representation and by considering each parameter required by the calculation of each memory address, according to the functions defined in Eqs. 3.3 and 3.4 (see Fig. 3.12).

The translation procedure is conducted by tracing each *access container* in the CRL, corresponding to each data access extracted from the original code. The CRL is then parsed by identifying each `fetch` instruction in the container and backtracking the instructions preceding it, in order to build its corresponding descriptor.

Accordingly, when an *access container* is identified, it is created a new *Access Descriptor* and



A. SpMV Code Snippet

```
...
for (j = 0; j < nzA[i]; j++)
   ... * x[iA[i][j]];
...
```

B. CRL Translation

```
...
context0.loop0.access0
.dim [$N]
.size 4
.base &nzA
    %idx0 = add &nzA,c0.l0
    fetch %idx0

...

context0.loop1.access1
.dim [$N,$N]
.size 4
.base &iA
    %1 = mul c0.l0,$N
    %idx1 = add c0.l1,%1
    %idx0 = add &iA,%idx1
    fetch %idx0

context0.loop1.access2
.dim [$N]
.size 4
.base &x
    %idx0 = add &x,c0.l1.a1
    fetch %idx0
```

C. Stream Dependency Encoding

```
...
[*s1, &nzA, 1, 0]:{N,1}
...
[*stream3, &iA, 2, 1]:{-,1},{N,N};
[data,1]:{xsize0,a1}

[*stream4, &x, 1, 1]:{-,-};
[data,2]:{xsize0, a1},{stride0,a3}
```

Figure 3.13: Indirection representation and descriptor encoding for a code snippet of the `spmv` kernel, adapted from the HPCG [100] benchmark.

its header is initialized with the information stored in the container header (see Figs. 3.12.B1 and B2). Next, whenever a `fetch` instruction is identified in the container, it is performed a simple dependency analysis between the operands of each preceding instruction and the container headers that comprise the data access context. Naturally, the dependency path between operands provides an implicit representation of the functions from Eqs. 3.3 and 3.4. Hence, each pair of the *Access Descriptor* is encoded by extracting the parameters corresponding to each dimension (see Fig. 3.12.B1). In particular, each `xsize` value is calculated according to the range of the indexation variable (represented by a reference to a *loop container*) corresponding to the parameter's dimension. On the other hand, `stride` values are calculated during the generation of the CRL (as described in Section 3.2.2.C) and are simply copied to the corresponding parameter.

In the example from Fig. 3.12.B1 (representing a common 2D matrix scan), `xsize0` and `xsize1` parameters are simply generated by calculating the size (`.limit` - `.init`) of the ranges from `c0.l1` and `c0.l0`, respectively. However, in the example from Fig. 3.12.B2 (representing a triangular matrix scan), there is a dependency between the `c0.l1` range `.limit` value and the `c0.l0` range. This is a result of the dependency between the indexation variables in the nested loop from the original code (see Fig. 3.12.A2). Accordingly, since the `.limit` parameter of `c0.l1` is defined by the `c0.l0` range, the `xsize0` value is initialized to 1 (corresponding to the initial `c0.l1` interval size) and it is created a field modifier descriptor (targeting `xsize0`) with a pair encoding the range of `c0.l0` (see Fig. 3.12.B2).

In the presence of indirect memory accesses (e.g., A[B[i]], $i \in \{0, 1, ..., N\}$), the dependency analysis performed on an *access container* (A), leads to another *access container* (B) (see

Fig. 3.13.B). In such cases, both containers are encoded with their corresponding *Access Descriptors* and an indirection descriptor is added to the modifier chain of descriptor A, indicating the data-dependency between accesses (see Fig. 3.13.C).

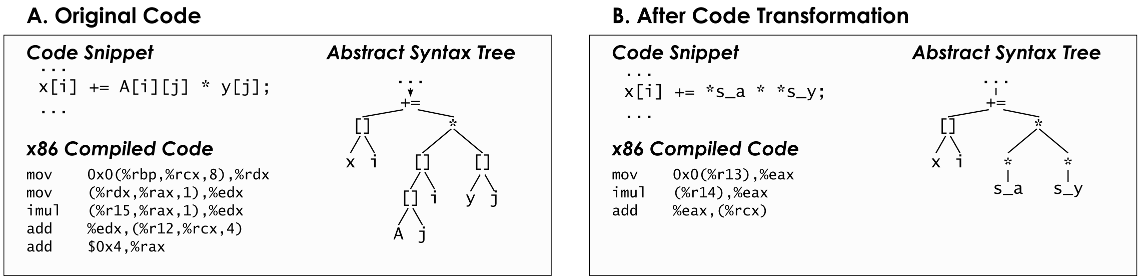
3.2.3.B Automatic Code Transformation

With the implicit detachment of the memory addressing and the data access phases, array indexing instructions in the original source code become redundant. Consequently, they can be eliminated by transforming the former array data accesses to stream accesses. The proposed code transformation pass performs this task in two steps. Initially, it injects a routine that provides the source code for allocating an array of stream references, corresponding to each transformed memory access. This provides a simple interface that reserves a unique pointer reference to each stream access.

Next, the devised pass performs the transformation of each extracted data access from array subscript indexation to stream access. Such a procedure relies on the fact that an access to an *n*-dimensional array structure is represented in the Clang AST by a sub-tree, where the root node represents the array subscript operator (i.e., `[.]`) for the array's first dimension (see Fig. 3.14). Accordingly, to transform each extracted data access to a data stream reference, it is only necessary to replace the subscript sub-tree with a modified sub-tree representing a pointer expression, as depicted in Fig. 3.14. To do so, the original AST is searched by tracing each data access captured by the CRL and performing the corresponding *in-situ* transformation.

3.2.3.C Descriptor Code Injection

After encoding the targeted memory accesses with the devised descriptor specification, it is necessary to provide appropriate mechanisms so that they can be uploaded to on-chip stream controllers, at runtime. To account for this step, the proposed compilation tool deploys a set of uploading routines that were devised to support the most commonly available stream management topologies. In particular, it considers topologies *i)* where there is a direct communication between the processing cores and the stream controllers via memory-mapped interfaces (latter discussed



```
 ...
 x[i] += A[i][j] * y[j];
 ...
```

```
mov     0x0(%rbp,%rcx,8),%rdx
mov     (%rdx,%rax,1),%edx
imul    (%r15,%rax,1),%edx
add     %edx,(%r12,%rcx,4)
add     $0x4,%rax
```

```
 ...
 x[i] += *s_a * *s_y;
 ...
```

```
mov     0x0(%r13),%eax
imul    (%r14),%eax
add     %eax,(%rcx)
```

Figure 3.14: Illustration of the stream code transformation for an example code snippet, together with the resulting generated code (compiled for the x86 instruction set).

in Section 4.2); and *ii)* where there is a dedicated stream management mechanism, independent from the processing infrastructure (latter discussed in Section 4.3).

When a direct communication between the system's cores and the streaming infrastructure is available (typically in core-coupled (distributed) stream controller topologies), it is possible to upload the required descriptor code directly from a core to a stream controller (through a dedicated or memory-mapped interface). Accordingly, the proposed compilation tool injects specific code (in the application code) for data streaming configuration and initialization. To do so, the generated descriptors are embedded in the original source code, encoded by a set of dedicated data structures. It is also injected the required source code to initialize the encoded descriptors with information that is only available at runtime (such as array base addresses and loop ranges). Next, the proposed compilation tool injects a routine that transfers the initialized descriptors to the stream controller, by placing a set of inline store instructions that send the descriptor data through a dedicated/memory-mapped interface.

On the other hand, when the streaming infrastructure is managed by a dedicated mechanism (typically used in centralized stream controller topologies, i.e., deployed close to the main memory), it is assumed that there is a runtime environment (and management hardware) that handles the descriptor initialization and loading procedures (as in [26, 54]). In such cases, it is only necessary to provide a mechanism that allows the descriptor code to be directly sent to the runtime management modules. To do so, the compilation tool embeds the encoded descriptor code in the final stages of the compilation flow, where it is placed in a dedicated code section (later linked to the final compiled program). Such an approach allows the uploading of the descriptor code into the stream management infrastructure, in parallel with the compiled application code to the system's processing cores.

3.3 Preliminary Experimental Evaluation

This section presents a preliminary evaluation of the devised memory access pattern descriptor specification and the proposed compilation tool.

3.3.1 Methodology

The experimental evaluation is performed in two stages. Initially, it is demonstrated the data-pattern encoding capabilities of the proposed descriptor specification. This was done by comparing its representation efficiency with existing data-pattern representations, deployed by the HotStream framework [26] and by the Xilinx AXI DMA [102] controller (which exploits a representation similar to that of the APMC [54]). To provide an initial assessment regarding encoding efficiency, a set of synthetic micro-benchmarks was specifically developed to guarantee representative data-pattern complexities found in different application domains. Hence, each micro-benchmark isolates a different data pattern sample, allowing to eliminate the effects from code-related overheads.

After this initial validation, the compilation tool was also evaluated by assessing its memory access pattern extraction and code reduction capabilities. Contrarily to the previous evaluation, in this case, full benchmark applications (and not data-pattern samples) were adopted to validate the proposed tool regarding its capability to operate over real-world applications. This was done by running the tool over a selection of applications from standard benchmark suites.

3.3.1.A Context Descriptor Configuration

For this specific evaluation, each parameter of the proposed descriptor specification was configured by assuming a 32-bit physical addressing space, resulting in the following dimensions:

- *Context Header*:
  - Header - `acc`: 8 bits, `next`: 16 bits;
  - Descriptor References - $\texttt{a\_id}_{acc}$: 16 bits.
- *Access Descriptor*:
  - Header - `stream`: 32 bits, `base`: 32 bits, `dim`: 8 bits, `mod`: 8 bits;
  - Pairs - $\texttt{xsize}_k$: 32 bits, $\texttt{stride}_k$: 32 bits.
- *Field Modifier Descriptor*:
  - Header - `target`: 16 bits, `dim`: 8 bits;
  - Pairs - $\texttt{xsize}_k$: 32 bits, $\texttt{stride}_k$: 32 bits.
- *Indirection Descriptor*:
  - Header - `data`: 16 bits, `dim`: 8 bits;
  - Pairs - $\texttt{target}_n$: 16 bits, $\texttt{a\_id}_n$: 32 bits.

The considered state-of-art approaches were configured with similar parameter sizes to provide a fair comparison. In particular, while the AXI DMA [102] scatter-gather descriptor already comprises 32-bit parameters, the Hotstream framework [26] instruction set was configured with 32-bit operands (instead of the default 16-bit configuration).

3.3.1.B Synthetic Data-Pattern Samples

To evaluate the data-pattern encoding efficiency of the proposed descriptor specification, a set of synthetic memory access patterns was constructed. The goal is to provide a representative subset of deterministic patterns that not only are commonly found in applications from several domains, but that can also be hard to describe with a lower dimensionality representation, namely:

- *Linear* and *Tiled* patterns, which are commonly found in most array computations, such as table/array-based algorithms or vector/matrix arithmetics;
- *Diagonal* patterns, which are characteristic to bioinformatics applications, for instance in biological sequence alignment algorithms [28];

Table 3.1: Considered standard benchmarks and kernels for the preliminary evaluation setup.

C-Polybench [99]	`2mm`	Multiple Matrix Multiplications
	`cov`	Covariance Computation
	`mvt`	Matrix-Vector Product and Transpose
	`seidel`	2D Seidel Stencil
	`syr2k`	Symmetric Rank-2k Update
	`trisolv`	Dense Triangular Solver
HPCG [100]	`spmv`(*)	Sparse Matrix-Vector Multiplication
	`symgs`(*)	Symmetric Gauss-Seidel (Sparse Triangular Solvers)
Rodinia [105]	`path`	PathFinder - 2D Shortest Path
	`srad`(*)	SRAD Diffusion Method

(*) Benchmarks characterized by indirect memory accesses.

- *Zig-Zag* and *Greek Cross* patterns, which are common to specific multimedia application phases, such as entropy encoding [103] and motion estimation algorithms [104], respectively.

These patterns were devised by also taking into account their support in the considered state-of-the-art approaches. As such, despite being support by the proposed specification, indirect memory access patterns were not considered in this set of synthetic samples (since the other approaches do not support their representation). The considered patterns (and corresponding datasets) are presented in the following pages and depicted in Fig. 3.15.

3.3.1.C Standard Benchmark Samples

The proposed compilation tool was evaluated with a set of memory access patterns sampled from a selection of benchmarks from the C-Polybench [99] and Rodinia [105] suites and kernels from the HPCG [100] benchmark (see in Table 3.1). All benchmarks were specifically selected (and compiled for the Intel x86 Instruction Set Architecture (ISA) [106]) to provide a representative set of memory access patterns and kernels present in current High-Performance Computing (HPC) applications. They were categorized as follows:

- **Polyhedral Loop Computation:** Nested loop computations in the affine domain are all-around. Their deterministic nature is particularly suited for memory access pattern description and data streaming. To represent this class of applications, a subset of kernels was selected from the C-Polybench [99] suite, comprising different combinations of pattern complexity, data reutilization, and dataset dimensionality;
- **Sparse Linear Algebra:** Sparse linear algebra is viewed as an important class of algorithms present in most HPC applications. They are usually represented in Compressed Sparse Row (CSR) format, requiring operations between sparse and dense arrays to be implemented through memory access indirection (i.e., `A[B[.]]`). This class of applications was considered by adapting the sparse matrix-vector multiplication kernel and the symmet-

ric Gauss-Seidel method (two consecutive sparse triangular solvers) from the HPCG [100] benchmark;

- **Scientific Benchmarks:** Several scientific application domains are regarded as particularly computationally demanding. To represent this class of applications, the SRAD diffusion method (used in ultrasonic and radar imaging) and the PathFinder algorithm (used to find the shortest path of a 2D grid) were selected from the Rodinia [105] suite. These benchmarks were chosen by considering the data access indirection present in SRAD, and the large data sets and high data reutilization that characterize PathFinder.

3.3.2 Data-Pattern Encoding Efficiency

The considered set of synthetic data-patterns was encoded with the proposed descriptor specification. By analyzing the encodings (see Fig. 3.15), it is possible to ascertain the capabilities of the proposed specification to describe data-patterns based on array structures with different dimensionality levels (see Fig. 3.15.A, B and E). For such patterns, it exploits their inherent hierarchical regularity to describe them with a single descriptor (with a number of pairs adjusted to the required dimensionality). Examples of such a characteristic are the *Tiled* and *Greek Cross* patterns, where matrix- and cross-patterns of 2D tiles, respectively, are repeated across the described memory region (see Fig. 3.15.B and E).

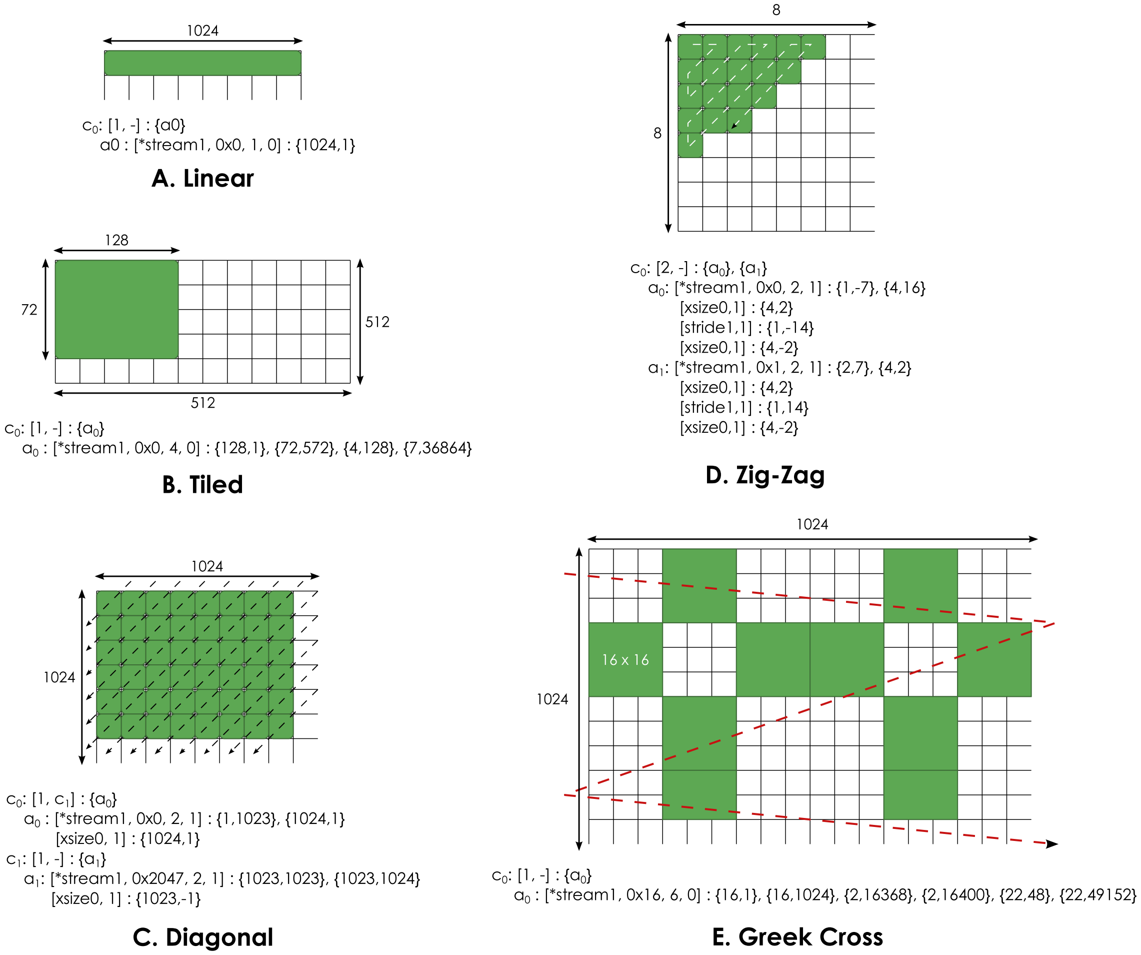
Such capabilities are highlighted in Table 3.2, which presents the required code size to represent the devised set of synthetic data-patterns and compares it with the considered state-of-the-art approaches (with their corresponding description encoding). While the simpler *Linear* and *Tiled* patterns are described with a code size similar to that of the AXI DMA [102] (APMC [54]), the description code from the HotStream framework [26] requires about two times more memory space. This results from an overhead imposed by initialization instructions that are required by HotStream [26] to describe the data-patterns, which in turn are simply coded by tuples of parameters in the proposed specification and the AXI DMA [102] (or similarly in the APMC [54]). Furthermore, despite its greater dimensionality, the *Greek Cross* pattern requires code sizes that are 4.23× and 3619× smaller than the HotStream [26] and AXI DMA [102]/APMC [54] approaches. This is mainly due to the fact that the compared approaches are designed to efficiently describe patterns with a dimensionality up to 3D (or tiled) [26, 54], and fall short in the presence of higher dimensional patterns. In the particular case of HotStream [26], the description requires the addition of control instructions to repeat the cross-pattern in the represented memory region. On the other hand, the AXI DMA [102] requires a large list of scatter-gather descriptors to represent the multiple dimensions of the *Greek Cross* pattern.

The proposed specification also provides advantages when the described pattern is characterized by a higher complexity, as it can be ascertained by analyzing the encodings of the *Diagonal* and *Zig-Zag* patterns (see Fig. 3.15.C and D). In particular, it makes use of the modifier chain to

Table 3.2: Characterization of the considered synthetic kernels and the corresponding description code size (in bytes) for the proposed specification and comparison with the considered state-of-the-art representations.

Pattern Type	Applications/ Data Structures	Pattern Length (# words)	Proposed	Hotstream [26]	AXI DMA [102]
			Size (bytes)	Size (bytes)	Size (bytes)
Linear	Array, Table	1024	23	48	32
Tiled	Arithmetic, Matrix	128×72*	47	80	32
Diagonal	Bioinformatics [28]	1024×1024	84	88	65k
Zig-Zag	Entropy Encoding [103]	8×8	125	132	480
Greek Cross	Diamond Search [104]	1024×1024	63	264	228k

* Within a memory block of 512×512



```
c0: [1, -] : {a0}
  a0 : [*stream1, 0x0, 1, 0] : {1024,1}
```

A. Linear

```
c0: [1, -] : {a0}
  a0 : [*stream1, 0x0, 4, 0] : {128,1}, {72,572}, {4,128}, {7,36864}
```

B. Tiled

```
c0: [1, c1] : {a0}
  a0 : [*stream1, 0x0, 2, 1] : {1,1023}, {1024,1}
       [xsize0, 1] : {1024,1}
c1: [1, -] : {a1}
  a1: [*stream1, 0x2047, 2, 1] : {1023,1023}, {1023,1024}
       [xsize0, 1] : {1023,-1}
```

C. Diagonal

```
c0: [2, -] : {a0}, {a1}
  a0: [*stream1, 0x0, 2, 1] : {1,-7}, {4,16}
       [xsize0,1] : {4,2}
       [stride1,1] : {1,-14}
       [xsize0,1] : {4,-2}
  a1: [*stream1, 0x1, 2, 1] : {2,7}, {4,2}
       [xsize0,1] : {4,2}
       [stride1,1] : {1,14}
       [xsize0,1] : {4,-2}
```

D. Zig-Zag

```
c0: [1, -] : {a0}
  a0 : [*stream1, 0x16, 6, 0] : {16,1}, {16,1024}, {2,16368}, {2,16400}, {22,48}, {22,49152}
```

E. Greek Cross

Figure 3.15: Considered synthetic data-patterns (shown using two-dimensional memory regions representations) and their corresponding descriptor encoding.

dynamically change the length of each diagonal scan in the described memory region. This approach is effectively similar to that of the HotStream [26] representation, where additional control instructions are used to modify the same parameters. As a result, both approaches incur in similar description code sizes, with the proposed specification requiring slightly less memory space (see Table 3.2). On the other hand, the AXI DMA [102] requires a list with one descriptor per diagonal to represent the *Diagonal* and *Zig-Zag* patterns. As a result of the application of the modifier chain, the proposed specification requires 3.9× less memory space, when compared to the AXI DMA [102] (APMC [54]).

3.3.3 Source Code Reduction Evaluation

The compilation tool was also evaluated in terms of code transformation and memory access encoding efficiency (see Fig. 3.16), when considering a set of benchmarks from standard suites. The observed results show a clear relation between *i)* the complexity of the address calculation on memory accesses converted to stream references; *ii)* the size of the corresponding descriptor representation; and *iii)* the consequent code reductions. This is particularly visible in the considered applications from the polyhedral domain, where such a relation is observed with a direct proportionality between each of the named factors. Such a characteristic is a direct consequence of the source code analysis that is performed by the proposed compilation tool, which follows the typical for-loop organization for array indexation.

Accordingly, Fig. 3.16 shows that only 40% of the loads in the `cov` benchmark were targeted to be converted to streams; however, since they are mostly matrix accesses, the extraction of address calculation from the code results in more than 13% code size reduction. On the other hand, the `seidel` benchmark is characterized by a reduced percentage of address calculation instructions, resulting in only 5% code reduction. Moreover, the named differences in address calculation complexity also result in a smaller descriptor size for the `seidel` benchmark, when compared to `cov`.

As it can also be observed, benchmarks with indirect memory accesses (`spmv`, `symgs` and `srad`) take the most advantage of the code transformations. The conversion of indirect memory accesses (in the format `A[B[.]]`) to single pointer references (e.g., `*stream`) eliminates memory accesses (from the code perspective), resulting in a significant amount of code reduction (up to 23%). Naturally, such a reduction imposes a larger descriptor size (see Fig. 3.16), due to the necessary inter-stream dependency encoding. In the particular case of the `srad` kernel (with the

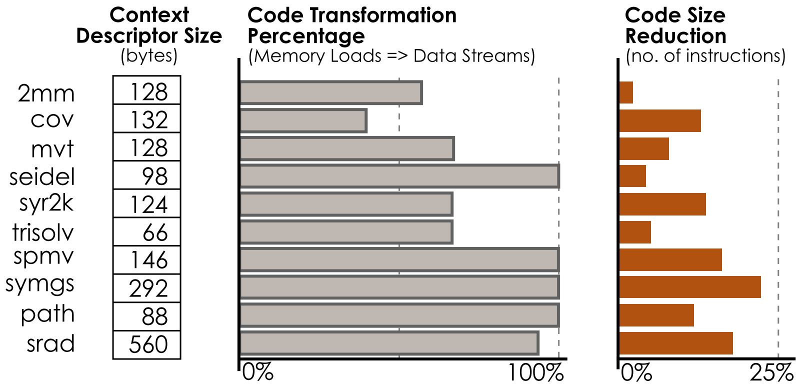


Figure 3.16: Context descriptor size, percentage of streamed accesses and resulting code reduction.

highest descriptor size), despite a conversion of 90% of the loads to data streams, there is a lower impact in the reduction of code size. This is mainly due to the fact that the `srad` kernel is highly compute-bound and most of the code performs complex computing operations, rather than memory indexing and load operations.

3.3.4 Discussion

The devised preliminary evaluation showed that the proposed memory access pattern description specification takes a step further from existing data-pattern representations for stream-based systems. It provides the necessary mechanisms to encode patterns with high dimensionality and complexity. It does so while leading to an overall reduction of the required amount of memory for storing the data access pattern when compared with the other state-of-art approaches. Moreover, as a result of its underlying mathematical model, it is capable of chaining multiple memory access descriptors to represent indirect memory access patterns, hence providing an hardly available support for a significant set of irregular memory access patterns.

Despite the wide range of proposed functionalities that were previously not supported by conventional stream representations, there are still several types of memory access patterns that were not considered. In particular, pointer-based and other similar accesses could also take advantage of the indirection encoding capabilities of the proposed specification. In fact, as long as a data pattern is implemented by an identifiable memory access structure (such as the indirect A[B[.]] format, even if the actual address sequence is data-dependent and can only be entirely generated at runtime (i.e., an irregular pattern), it could potential be supported by the proposed specification (with proper extensions). Similarly, specific extensions could also be considered to support the conditional generation of portions of memory access patterns, particularly useful to encode graph-based (or similar) applications.

The performed evaluation also showed the capabilities of the proposed compilation tool to automatically extract and encode memory access patterns for generic stream-based systems. The observed results not only showed its viability to be integrated into state-of-the-art compilers but also highlighted its capabilities to generate and optimize data-streaming code from conventional application code.

The proposed code transformation mechanism acts over the compiler's AST to extract, encode and convert the memory accesses to data streams. With the proposed approach, the source code is completely modified at the compiler's front-end. As such, the resulting pointer (stream) references are interpreted throughout the compilation chain as common load instructions, while the encoded descriptor code is viewed as conventional data structures. This enables a natural compatibility with any existing coding directives that can be interpreted by the remaining phases of the compiler chain.

However, it is still possible to take a step further to take advantage of the set of optimizations performed by the compiler in subsequent phases. Accordingly, future implementations of the compilation tool may consider moving the proposed memory access pattern analysis to the compiler's optimization phases over its intermediate representation (e.g., LLVM IR). Such an approach would allow the combination of the proposed data stream generation and code transformation mechanisms with other optimization phases, such as vectorization or automatic parallelization passes, in turn enabling possible gains during the execution in a wide range of processing architectures.

3.4 Summary

This chapter presented the proposed compile-time analysis and code transformation methods, aimed at providing support to stream-based communication infrastructures and techniques. The devised stream code generation tools rely on a new memory access pattern description specification, capable of efficiently representing deterministic data access sequences, with arbitrary complexity. Its underlying affine mathematical model also allows other alternative representations, where multiple data access patterns can be chained to represent indirect memory accesses.

The proposed access description specification is exploited by a static analysis tool to extract and encode memory access patterns in pre-annotated regions of application source code. To do so, the tool leverages a specially devised internal representation, that gathers all the information regarding the calculation of each memory access, to generate the corresponding data streaming descriptors. Finally, a code transformation pass modifies all the encoded accesses to stream references, thus eliminating redundant data indexation code and reducing the number of compiled instructions.

Finally, a preliminary assessment was performed to validate the proposed methods. The proposed descriptor specification showed to be capable of efficiently encoding memory access patterns with different complexities, while requiring reduced description code sizes, when compared to other state-of-the-art stream representations. On the other hand, the proposed compilation tool was validated with a set of standard benchmark applications, where it showed effective to exploit the developed automatic memory access pattern extraction and code reduction capabilities.

4

Data Stream Communication

Contents

4.1 Data Stream Generation 63

4.1.1 Data Stream Controller Architecture 63

4.1.2 Streaming Infrastructure Interface and Programming 67

4.2 Data Stream Prefetching 67

4.2.1 Case Study A: Stream Prefetching on GPGPUs 68

4.2.2 Case Study B: Data Streaming on Modern General-Purpose CPUs 75

4.2.3 Discussion 83

4.3 In-Cache Stream Communication Paradigm 84

4.3.1 In-Cache Stream Controller 85

4.3.2 Communication Infrastructure and Protocol 89

4.3.3 Memory-Aware Data Stream Generation 91

4.4 In-Cache Streaming Evaluation 93

4.4.1 Methodology 94

4.4.2 Hardware Resources Overhead 96

4.4.3 Stream Generation Efficiency and Main Memory Throughput 98

4.4.4 Prototype Evaluation 99

4.5 Summary 103

The data stream communication mechanisms proposed in this thesis were envisaged as an alternative to the sole utilization of conventional cache hierarchies and predictive data prefetching approaches in general-purpose computing systems. The key idea is to allow the system's processing cores and shared interconnections to view the application dataset as temporally structured sequences of data blocks (*data streams*). It achieves this goal by detaching data access sequences from the physical memory address where they are stored, and by using the memory access pattern descriptor specification proposed in Chapter 3. This allows the required data to be organized and transferred to/from the main memory without relying on conventional address correlation premises, such as those that are on the basis of caches and prefetchers. With such an approach, it is not only possible to mitigate memory access latency issues resulting from poor data-locality, but it also eliminates delays caused by prefetching prediction inaccuracies and monitoring overheads.

Furthermore, since data is *streamed* to the processing cores in the exact sequence in which it needs to be processed, several advantages can be exploited across the memory access subsystem. In particular, shared interconnections and intermediate data buffering mechanisms can rely on modules with lower hardware complexity. This is possible because data is organized in an implicit temporal structure that does not require the explicit storage and post-indexation provided by cache memories. Moreover, after being generated, a data stream only needs to be conducted from one system component to another in a point-to-point data transfer scheme, instead of the request-based paradigms of conventional cache hierarchies. Furthermore, alternative data transfer mechanisms (such as data broadcasting) can also be used to mitigate the contention issues that occur in shared communication infrastructures of systems with large numbers of processing cores.

Accordingly, this chapter presents and evaluates a newly proposed data stream communication paradigm for general-purpose computing systems that is supported by the compilation tool proposed in Chapter 3. The designed communication infrastructures were deployed in computing systems *i)* with conventional cache hierarchies where stream generation modules are placed close to the processing cores (as stream prefetchers); and *ii)* with dedicated communication infrastructures where stream generation modules are placed closer to the system's main memory. Hence, the devised data streaming mechanisms comprise:

- **A dedicated Data Stream Controller (DSC)** designed to index memory access patterns described by the descriptor specification proposed in Chapter 3;
- **Data stream prefetching mechanisms** that deploy the DSC close to the processing cores of conventional Graphics Processing Unit (GPU) and General Purpose Processor (GPP) systems, as an alternative to the utilization of predictive prefetching schemes;
- **An In-Cache Stream (ICS) paradigm** that not only deploys the DSC close to the main memory, but it simultaneously exploits a conventional cache-coherent memory hierarchy

to implement the data streaming communication schemes, by allowing the data transfer infrastructure to cooperatively exploit both memory-address-based and stream-based communication paradigms.

The proposed stream prefetching mechanisms were thoroughly evaluated by simulating the corresponding infrastructure in GPU and Central Processing Unit (CPU) architectures. To do so, two different case studies were devised to provide a comprehensive validation of the proposed mechanisms in real processing systems and their comparison to state-of-the-art prefetching solutions. On the other hand, the proposed ICS paradigm was implemented in a many-core accelerator, prototyped in a Field-Programmable Gate Array (FPGA) device. The implemented infrastructure and its components were thoroughly evaluated and compared to other state-of-the-art solutions.

4.1 Data Stream Generation

The central components of any data streaming infrastructure are the stream generation units. Accordingly, a dedicated DSC was designed to index memory access patterns encoded with the descriptor specification proposed in Chapter 3. It implements an efficient descriptor decoding architecture that can deploy a single-cycle per address generation throughput, with minimal hardware resource requirements. Its architecture was devised to be as independent as possible from the topology of the underlying data streaming infrastructures. Accordingly, its deployment is envisaged both as a stream prefetcher (when paired with a processing core or a cache memory) or as a dedicated stream generation controller (when deployed close to the main memory).

4.1.1 Data Stream Controller Architecture

The proposed DSC's components (depicted in Fig. 4.1) work together to automatically perform data fetching and stream generation. Its architecture was specifically designed to resolve memory access patterns encoded with the *Context Descriptor* representation provided by the proposed specification. This is done according to the resolving procedure described in Section 3.1.2. To do so, the architecture is divided into three main sub-modules, namely: *i)* a *Context Controller*, responsible for managing the descriptor resolving procedure according to the sequence encoded in each *Context Header*; *ii)* an Address Generation Unit (AGU), responsible for generating the sequence of memory addresses encoded by an *Access Descriptor*; and *iii)* a Modifier Chain Unit (MCU), responsible for modifying a descriptor according to its modifier chain (if available). The descriptor representation to be resolved is maintained in a scratchpad *Descriptor Memory* (see Fig. 4.1.A). This memory module is accessed by the *Context Controller* (to obtain the descriptors to be resolved) and is loaded through a dedicated communication interface, according to the adopted topology of the underlying data streaming infrastructure.

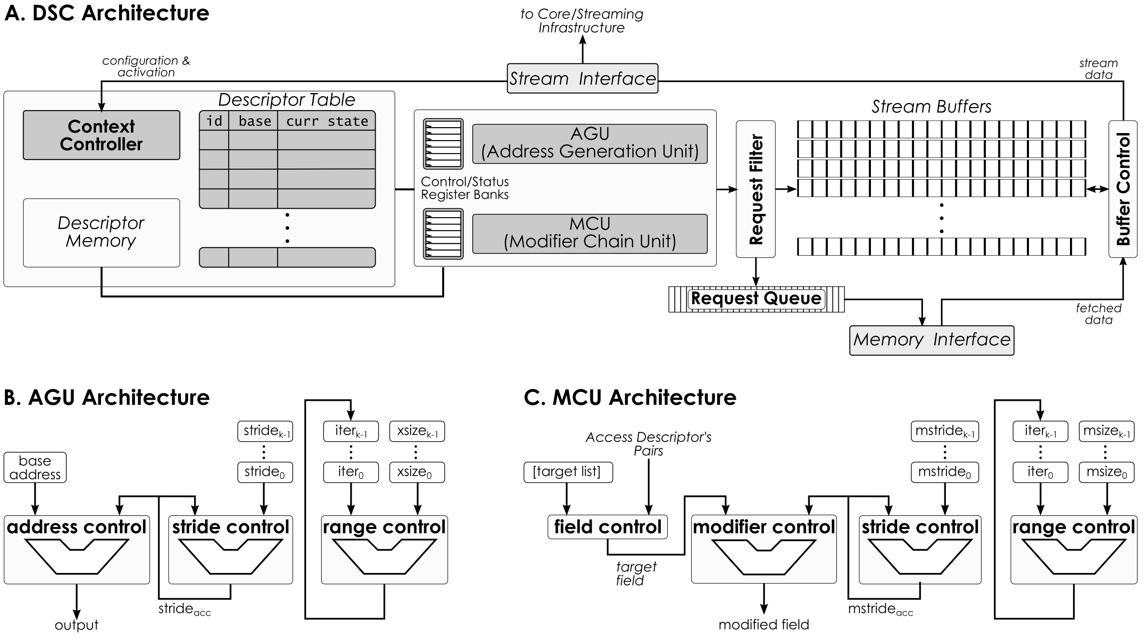


Figure 4.1: Data Stream Controller architecture.

To correctly manage the resolving procedure, the *Context Controller* maintains the iteration state of the complete descriptor representation in a dedicated *Descriptor Table* (see Fig. 4.1.A) and loads each *Access Descriptor*, and corresponding modifier chains, to the AGU and MCU modules through dedicated register banks. As it is also depicted in Fig. 4.1.A, the three functional units are complemented with several *Stream Buffers* and data communication interfaces that not only allow the DSC to be configured, but also facilitate the acquisition of data blocks (according to the calculated sequence of memory addresses) and their organization and communication in data streams.

4.1.1.A Descriptor Resolving Architecture

As described in Chapter 3, the proposed descriptor specification is composed of *Access* and *Field Modifier Descriptors* that can have a variable dimensionality. Despite the complexity of the proposed representation, that could lead to costly hardware structures to implement the resolving procedure, the devised architecture footprint was kept as low as possible. As such, it is solely based on three adders and a set of control and status registers (see Fig. 4.1.A), that are used to store a single descriptor's parameters and its current iteration state.

Accordingly, both the AGU and MCU functional units rely on a similar resolving architecture that comprises three parallel functional blocks (see Figs. 4.1.B and C), each composed of an adder and a set of registers. They are responsible for iterating each $\{\mathtt{xsize}_k,\mathtt{stride}_k\}$ pair and for applying the result to the descriptor base address or a target modifying field (refer to the resolving procedure described in Chapter 3).

Specifically, each functional unit is composed of a *stride control* block that is used to successively accumulate the $\texttt{stride}_k$ fields of each pair of the descriptor, in a $\texttt{stride}_{acc}$ value. On the other hand, a *range control* block is used to count the number of iterations of each pair ($\texttt{iter}_k$), according to the range size defined by the corresponding $\texttt{xsize}_k$ field. While both these blocks are common to the AGU and MCU, the remaining functional blocks are specific to each of the modules.

Accordingly, the AGU is also composed of an *address control* block (see Fig. 4.1.B). It is responsible for successively adding the accumulated $\texttt{stride}_{acc}$ value to the *Access Descriptor* base address, and to produce the final sequence of memory addresses corresponding to the encoded data stream.

Similarly, the MCU functional unit (see Fig. 4.1.C) is composed of two additional functional blocks. A *field control* block is responsible for selecting a field from the current *Access Descriptor* according to the target indicated in a *Field Modifier Descriptor*. The selected field is sent to a *modifier control* block, which performs the required modification operation by adding it to the accumulated $\texttt{mstride}_{acc}$ value (generated by the *stride control* and *range control* blocks). Additionally, in the presence of an *Indirection Descriptor*, the *field control* block notifies the *Context Controller* of a data-dependency between two *Access Descriptors* (indirect memory access). It does so by sending the corresponding target field and descriptor identification that represent the dependency.

Finally, the AGU and MCU functional units are paired with dedicated register banks (see Fig. 4.1.A), programmable by the *Context Controller*. They are used to maintain the field values and the iteration state of the descriptor being solved, by storing the corresponding $\{\texttt{xsize}_k\texttt{,}\texttt{stride}_k\}$ tuples, the accumulated $\texttt{stride}_{acc}$ values and the $\texttt{iter}_k$ iteration values of each pair.

4.1.1.B Stream Generation Management and Communication

The DSC's stream generation procedure is managed by the *Context Controller* unit (represented in Fig. 4.1.A). This central module starts working upon the loading of a descriptor representation to the *Descriptor Memory*. This is done through a dedicated stream interface (see Fig. 4.1.A), that can either be directly commanded by the processing core (e.g., via a memory-mapped interface) or managed by the streaming infrastructure (through a runtime environment and management hardware).

During its operation, the *Context Controller* makes use of a simple parameter matching architecture that extracts descriptor references from a *Context Header* and obtains the corresponding *Access Descriptors* from the *Descriptor Memory*. It also relies on the dedicated *Descriptor Table* to assign the resolving descriptors to the AGU and MCU modules in the correct order (indicated in the *Context Header*), and save their overall iteration state. Naturally, the parameter matching procedure of the *Context Controller* imposes an initial overhead to the whole descriptor resolving procedure. This a result of the necessary parsing of the first *Context Header*, which in turn encapsulates the first *Descriptor* of the encoded representation. However, this initialization overhead is kept low, requiring only 4 clock cycles (one per step), namely: `1.` read the first *Context Header* from the *Descriptor Memory*; `2.` parse the *Context Header*'s first *Access Descriptor* reference; `3.` read the corresponding *Access Descriptor* from the *Descriptor Memory*; and `4.` parse and load the *Access Descriptor* to the *Descriptor Table* and the AGU. After these initial clock cycles the remaining descriptor resolving procedure and *Context Header* parsing are performed completely in parallel (as described above), attaining a throughput of one address per cycle.

During the AGU and MCU descriptor resolving procedure, each generated address is initially loaded in a convenient set of *Stream Buffers* (see Fig. 4.1.A), each assigned to a stream reference (encoded in the *Access Descriptor*). Such a structure allows the DSC to keep track of the sequence of outstanding memory access requests for the data stream being generated. Accordingly, as soon as the corresponding data is fetched from memory, the generated address entries are filled and the stream is transferred to the communication infrastructure. This is either done as soon as the data is acquired (in a centralized streaming topology) or upon request (in a core-coupled topology).

Additionally, in the presence of data dependencies between streams (in the case of indirect memory accesses), the data stored in the *Stream Buffers* is used by the AGU to iterate the dependent *Access Descriptor* (refer to the resolving procedure in Section 3.1.2). The data dependence path is set by the *Context Controller*, according to the information provided by the dependent *Access Descriptor* modifier chain.

When the AGU generates a new address, it is passed through a specially devised *Request Filter* module (see Fig. 4.1.A). Its purpose lies with the fact that the size of the data blocks that are requested by a processing core is typically smaller than the data blocks that are transferred in the memory subsystem. As an example, while the size a common integer value (as viewed by the core) is usually 4 or 8 bytes, the size of a cache line is commonly in the order of tens of bytes [106]. Similarly, the data size of each block in a data stream can be smaller than the size of blocks that are transmitted in the underlying streaming infrastructure. Accordingly, the *Request Filter* module maintains the last data block that was previously fetched for each stream (in a set of registers). Hence, each generated address is served with data from the fetched block, by filling the corresponding entry in the *Stream Buffers*. When a newly generated address crosses the address range of the available data block, the *Request Filter* autonomously issues a new memory access for a new data block. The issued requests are inserted in a *Request Queue*, also implemented by a buffer structure, and later sent to the lower memory hierarchy (through the *Memory Interface* depicted in Fig. 4.1.A).

4.1.2 Streaming Infrastructure Interface and Programming

The compilation tool presented in Chapter 3 provides support for the deployment of data streaming infrastructures with different characteristics. As such, the proposed DSC provides a generic *Stream Interface* (see Fig. 4.1.A), allowing it to be deployed with different interfacing schemes, such as:

- **A memory-mapped interface** that can be used in cases where the DSC is directly controlled by the system's processing cores. In this case, the *Stream Interface* provides a memory-mapped connection to the core that uses it to send descriptor data and initiate the stream generation procedure, through a set of inline store instructions (injected by the compilation tool);
- **A controller interface** that can be used in cases where the DSC is controlled by the system's runtime environment and management hardware. In this case, the *Stream Interface* is connected to the data streaming infrastructure and simply receives and stores descriptor data sent by the system's management facilities, as specified by the compilation tool.

With the considered interfacing schemes and the provided compiler support for data streaming, the DSC can be deployed in a wide range of computing systems (from dedicated accelerators to general-purpose systems). Accordingly, the remaining sections of this chapter will describe its deployment as *i)* a stream prefetcher, in computing systems with GPUs and GPPs; and *ii)* as a stream management controller, in a dedicated accelerator.

4.2 Data Stream Prefetching

As it was described in the previous chapters, the current prefetching technology is reaching a throughput limit caused by unavoidable memory access monitoring delays and prediction inaccuracies. Such limitations not only degrade the performance of the whole processing system but can also cause higher energy consumptions associated with an inefficient memory access procedure.

Accordingly, the proposed DSC offers an opportunity to deploy alternative compiler-assisted prefetching mechanisms that are not bound by monitoring and prediction overheads. Naturally, this is done by relying on the memory access pattern extraction and encoding mechanisms provided by the proposed compilation tool, to eliminate data fetching inaccuracies and to acquire the exact sequence of data required by the system's processing cores. Consequently, the autonomous stream generation procedure deployed by the proposed DSC eliminates the need for costly memory access monitoring and prediction algorithms. As such, the combination of these advantages can be exploited to take a step further from the utilization of predictive prefetchers and offer new levels of performance and energy efficiency.

To validate this hypothesis, GPU and GPP stream prefetching mechanisms are herein proposed that exploit the DSC with different integration levels, namely:

- **In a general-purpose GPU architecture**, to reduce the high contention of its massively-parallel communication infrastructure. The devised mechanism exploits the DSC to deploy stream prefetchers that automatically feed the data to the L1 caches of each Streaming Multiprocessor (SM). This is done according to memory access sequences encoded with the proposed descriptor specification.
- **In a general-purpose x86-based processor**, as an alternative to pure prefetching mechanisms. The DSC is deployed as an autonomous data fetching controller that works independently of the L1 cache and feeds data directly to the core (upon request). The provided autonomous data acquisition procedure is also combined with the stream code transformations provided by the proposed compilation tool to account for the throughput saturation that has been achieved by pure prefetching approaches.

The proposed implementations are described and evaluated in the following paragraphs through two individual case studies.

4.2.1 Case Study A: Stream Prefetching on GPGPUs

The designed GPU stream prefetching mechanism relies on the integration of the proposed DSC in the memory interface of the NVIDIA Fermi^{TM} GPU (GTX480) architecture [107], implemented on the GPGPU-Sim simulator [108]. Although this architecture was selected due to its support by the GPGPU-Sim [108] simulator, the proposed mechanism is architecture-independent and can be straightforwardly implemented in recent NVIDIA architectures [109]. This is mainly due to the fact that none of the features that have been introduced in recent NVIDIA GPUs [109] affect or compromise the proposed integration in their memory hierarchy.

The implemented prefetching mechanism makes use of the data fetch handling resources from the L1 data cache of each SM, by independently prefetching the required data for each issued Cooperative Thread Array (CTA) (as known as CUDA block). It also transparently intercepts the cache miss memory requests, serving them either with buffered prefetched data or merging them with outstanding prefetch requests. Such an approach allows the straightforward deployment of an autonomous prefetching scheme that does not rely on complex monitoring and feedback-based control substructures, in turn eliminating detection overheads and reducing the amount of contention and the pressure over the memory subsystem.

4.2.1.A GPGPU Architecture and Memory Subsystem

The rather consolidated massively parallel computing structure of current GPUs (depicted in Fig 4.2) is based on a Single-Instruction Multiple-Thread (SIMT) execution model, where a massive amount of threads is launched at each SM, in groups of CTAs. The threads of each CTA are then executed in fixed-sized batches (named *warps*), in which all threads simultaneously execute the same instruction. To deal with the long memory access (and inherent execution) latencies, warps are switched and enqueued while data dependencies and outstanding memory requests are resolved. This is accomplished by increasing the amount of on-the-fly parallelism, to hide the instruction execution latency behind the computation of other threads.

However, the SIMT execution paradigm can only partially mitigate the resulting pressure in the main memory subsystem. Due to the long access times of the global memory, the SIMT architecture and the warp switching mechanisms, by themselves, can hardly mitigate the imposed overheads. In the particular case of the NVIDIA Fermi^{TM} architecture [107], each SM integrates a set of L1 CTA-private caches (constant, texture and data) and a local shared memory used for inter-CTA communication. Due to the high volume of simultaneous requests, cache misses are registered in dedicated Miss Status Hold Registers (MSHRs) and scheduled to the subsequent memory level [107]. Furthermore, to efficiently manage the available memory bandwidth, requests to contiguous memory positions are grouped in a special coalescing unit before they are sent to the L1 caches [107].

Furthermore, due to a large amount of concurrently executing threads, the commonly adopted CPU-like cache structures are not well-suited to address the complexity and demand of the memory access patterns of some High-Performance Computing (HPC) kernels. In fact, the presence of such structures can even lead to an increased contention to move data from the main memory to caches, often degrading (instead of improving) the resulting performance. Moreover, given the number of simultaneous requests for distinct memory regions, data locality is often poorly exploited, resulting in low cache hit rates and increased access latencies. This is supported by

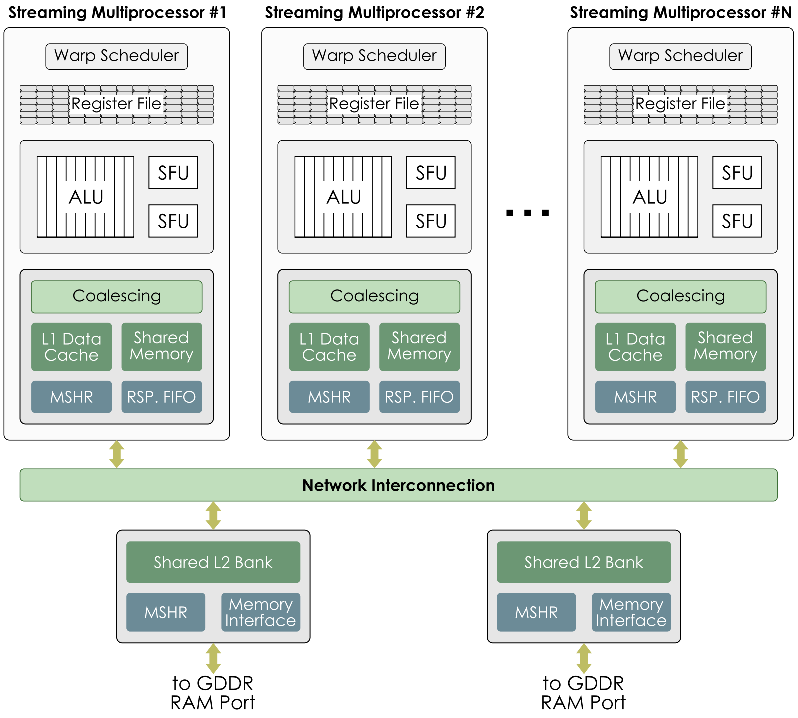


Figure 4.2: GPU architecture and memory hierarchy overview.

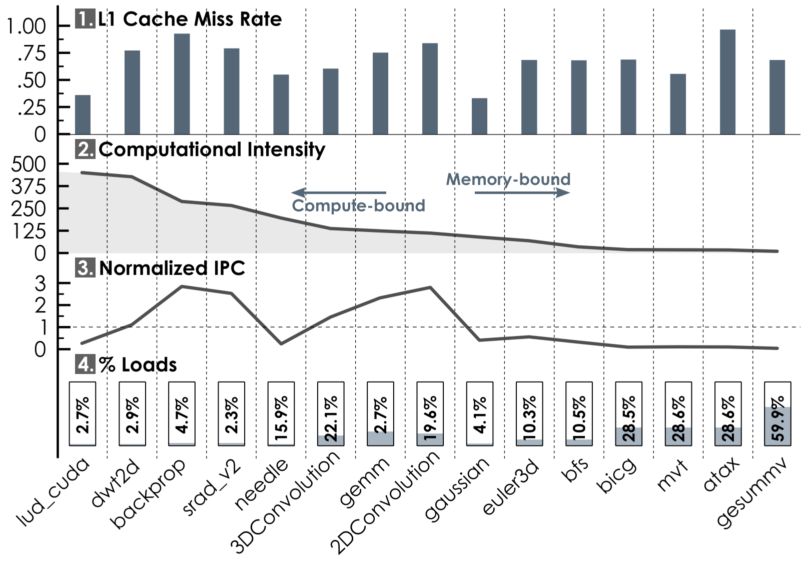


Figure 4.3: Application profiles (cache miss rate, computational intensity, normalized IPC and percentage of issued load instructions) for subsets of the Rodinia [105] and Polybench [99] benchmark suites.

the observed L1 cache miss rates (up to 94%), and by its impact on the system's performance (shown in Fig. 4.3 for representative subsets of the Rodinia [105] and Polybench [99] benchmark suites).

Besides the thread management, scheduling and profiling techniques [110, 111], only the adoption of prefetching techniques has proven to successfully deal with such drawbacks [16, 112–114]. In particular, the straightforward introduction of prefetching mechanisms to execute memory-bound applications (with lower computational intensity - see Fig. 4.3.2) inherently allows remarkable improvements of the performance and cache efficiency. However, cache inefficiencies resulting from complex memory access patterns (namely, poor spatial and temporal locality) have only been significantly mitigated with aggressive predictive techniques and software-aided control, monitoring and feedback mechanisms to manage the prefetching procedure [16, 112, 113].

Alternatively, the proposed stream prefetching mechanism aims at mitigating the data-locality and contention issues present in the GPU memory access subsystem, without requiring costly prediction overheads and prefetching inaccuracies. It does so by encoding the memory access pattern of a given kernel (with the proposed specification), and by automatically obtaining the required data (with the proposed DSC) without interfering with the remaining memory hierarchy. Such an approach eliminates prefetching prediction inaccuracies and monitoring delays, and avoids the deployment of complex decision control structures.

4.2.1.B Implementation Considerations

To deploy the proposed specification in a GPU prefetching mechanism, several approaches can be considered regarding its granularity and the context level at which it is applied. In fact, it

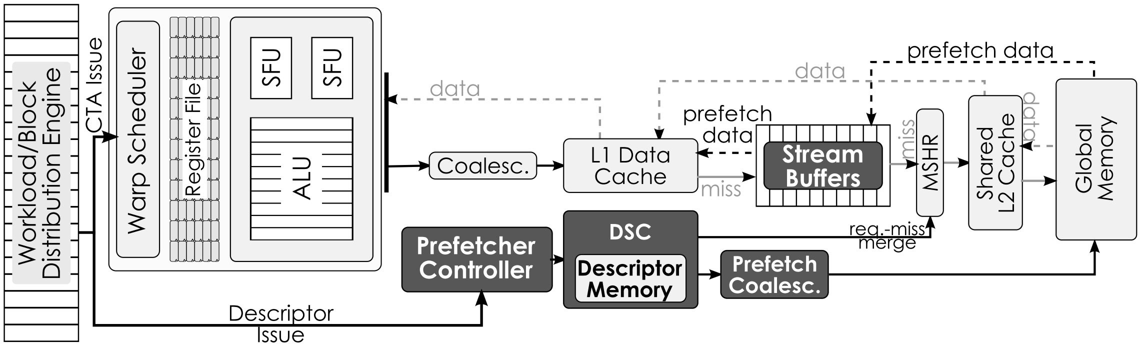


Figure 4.4: Integration of the stream prefetching mechanism in the SM's L1 memory sub-hierarchy and prefetecher architecture.

is common to exploit prefetching techniques at a thread-level granularity [16], which in practice incurs in a warp-level prefetching, since all threads of a given warp simultaneous perform the memory accesses. However, although such a fine-grained prefetching allows data to be fetched in the same order in which warps are scheduled, it also requires active complex control and monitoring infrastructures (including modifications to the scheduler itself [112, 113]).

On the other hand, if a more coarse-grained approach is considered, other transparent and fully autonomous prefetching mechanisms can be explored. Accordingly, a CTA-level prefetching granularity is adopted, targeting the deployment of a low-profile and transparent data fetching scheme. Hence, all memory access descriptors are encoded with a CTA-level granularity, with the valid assumption that their corresponding warps are initially scheduled in order and switched in a round-robin fashion [108, 115]. Hence, data can be made available as quickly as possible by designing the prefetching mechanism in such a way that it fetches and buffers data as soon as a CTA is launched.

4.2.1.C DSC Integration in the GPU Architecture

The devised stream prefetching mechanism was implemented and integrated in parallel with the existing L1 data caches (see Fig. 4.4). The prefetcher is responsible for issuing prefetch requests, encoded with the proposed descriptor specification. It also intercepts and merges cache miss requests with prefetch requests and serves them with prefetched data. Its architecture comprehends: *i)* a dedicated controller module, responsible for accepting incoming descriptors and for initializing them with CTA configuration parameters, as well as managing the prefetcher operation; *ii)* the proposed DSC, to solve the descriptor specification and to generate its corresponding memory access sequence; and *iii)* a prefetch request coalescing unit, entirely similar to the original one that is adopted in the SM's caches.

When a CTA is issued on a SM by the GPU's workload distribution engine (the GigaThread Engine, in NVIDIA GPUs [107]), its descriptor code is separated from the kernel code and sent to the stream prefetcher controller, together with the configuration parameters of the CTA (depicted in Fig. 4.4). Upon their reception, the controller initializes the descriptors with the parameter values and stores them in a dedicated scratchpad memory.

According to the GPU's SIMT execution model, the data that is processed by each thread in a CTA can be indexed by using predefined system variables containing the corresponding thread and CTA identifiers. Hence, although the assigned kernel code is the same, each CTA initializes the values of such variables with its runtime identification parameters, thus defining the memory regions that each thread will access. By following the same principle, the fields of the descriptor specification can be initially encoded by taking advantage of the same CTA identification parameters, since the memory access pattern of a given kernel is essentially the same for all of its CTAs.

In turn, the stream prefetcher is itself integrated with the conventional GPU cache hierarchy. Hence, the generated memory address requests are coalesced by a dedicated prefetch coalescing unit and sent to the global memory. Moreover, the stream requests issued to the global memory bypass the L2 cache banks, to avoid the introduction of more contention and to not interfere with the coherency policy of the shared cached data. Upon reception of the requested data streams, they are stored in the DSC's *Stream Buffers* (separately depicted in Fig. 4.4), which work in parallel with the L1 data cache. Their utilization allows the storage of prefetched data obtained before it is needed by the processing infrastructure. As a result, it avoids filling up the cache memory with prefetched data, which could otherwise incur in premature cache line eviction, and ultimately degrade the overall performance.

Accordingly, the requests to identical memory regions issued either by the stream prefetcher or by miss requests from the L1 cache, are merged and registered in the MSHR. This way, requests issued by the cache can be checked, by the stream prefetcher, against the data already stored in the *Stream Buffers*. In case the required data is present, it is immediately copied to the L1 cache and sent to the SM. Otherwise, the miss request is either merged to an outstanding prefetch request in the MSHR or directly registered and sent to the L2 cache.

4.2.1.D Methodology

As was previously stated, the devised GPU stream prefetching mechanism was integrated in the NVIDIA Fermi$^{\text{TM}}$ GPU (GTX480) architecture [107], allowing to compare to the conventional baseline model implemented in GPGPU-Sim [108] (version 3.2.2). The adopted simulator configuration is detailed in Table 4.2, together with the considered subsets of the Rodinia [105] and Polybench [99] benchmark suites. Power consumption was estimated with the GPUWattch [116] tool. All the benchmarks were fully simulated for the baseline and stream prefetching architectures. The obtained results are shown in Fig. 4.5, with the considered benchmarks ordered by computational intensity (see Section 4.2.1.A).

Table 4.1: GPGPU-Sim configuration for a NVIDIA Fermi^{TM} architecture model; adopted Rodinia [105] and Polybench [99] benchmark applications and datasets.

SIMT core	16 cores, SIMT width=32, 5-Stage Pipeline, 1.4GHz
Core Resources	48KB scratchpad, 32768 registers, 32 MSHRs, 1536 threads, 48 warps
L1 Cache	32KB/core, 4-way, 128B line, coalescing enabled
L2 Cache	8 banks, 128KB/bank, 16-way, 128B line
Scheduling Policy	LRR warp scheduling, round-robin CTA scheduling
DRAM Model	FR-FCFS Scheduling, 924 MHz, 6 GDDR5 MCs, BW=8Bytes/Cycle
GDDR5 Timing	t_{CL}=12 ns, t_{RP}=12 ns, t_{rC}=40 ns, t_{RAS}=28 ns, t_{RCD}=12 ns, t_{RRD}=6 ns

APPLICATION	BENCHMARK SUITE	INPUT SIZE
lud_cuda	Rodinia [105]	256
dwt2d	Rodinia [105]	1024*1024
backprop	Rodinia [105]	65536
srad_v2	Rodinia [105]	2048*2048
needle	Rodinia [105]	2048
3DConvolution	Polybench [99]	256*256*256
gemm	Polybench [99]	512*512*2
2DConvolution	Polybench [99]	4096*4096
gaussian	Rodinia [105]	512
euler3d	Rodinia [105]	97K
bfs	Rodinia [105]	1M nodes
bicg	Polybench [99]	16M
mvt	Polybench [99]	16M
atax	Polybench [99]	16M
gesummv	Polybench [99]	16M

4.2.1.E Experimental Evaluation

The charts presented in Figs.4.5.A and 4.5.B depict the attained L1 data cache efficiency and the attained performance speedup. As it can be observed, a significant cache hit-rate improvement is obtained (between 34% and 82%), resulting in a clear relation between the performance and cache efficiency improvements, and the computational intensity of each considered application (see also Fig. 4.3). Moreover, the measured gains reflect the allied capabilities that are provided by the stream prefetcher to mitigate the cache performance degradation effects of memory-bound applications and complex memory access behaviors. In particular, it shows hit-rate improvements between 68% and 82% for the `euler3d`, `gaussian`, `atax`, `bicg`, `gesummv` and `mvt` memory-bound applications. The exception concerns the `bfs` benchmark, where a smaller improvement is observed (58%), due to its irregular data access nature [105]. Although slightly less effective for the remaining compute-bound applications, the prefetching mechanism still allows a significant average cache hit rate improvement of 61.3%.

Furthermore, from the chart presented in Fig. 4.5.B, it is possible to ascertain that the stream prefetching mechanism allows performance speedups (against the baseline setup) as high as 9.2x (in memory-bound applications). In general, the attained performance gains range from 2.8x and 3.1x, measured for the `3DConvolution` and `euler3d`, up to the higher speedups of 5.7x and 9.2x, in the `bicg` and `gesummv` applications, respectively. These values show the ability of the proposed stream prefetching mechanism to hide long memory accesses (and mitigate their performance degradation). Moreover, it is shown that even though computationally intensive applications inherently tend to masquerade the impact of the communication infrastructure on the global application performance, they can still achieve a significant performance improvement when aided by the implemented prefetching mechanism.

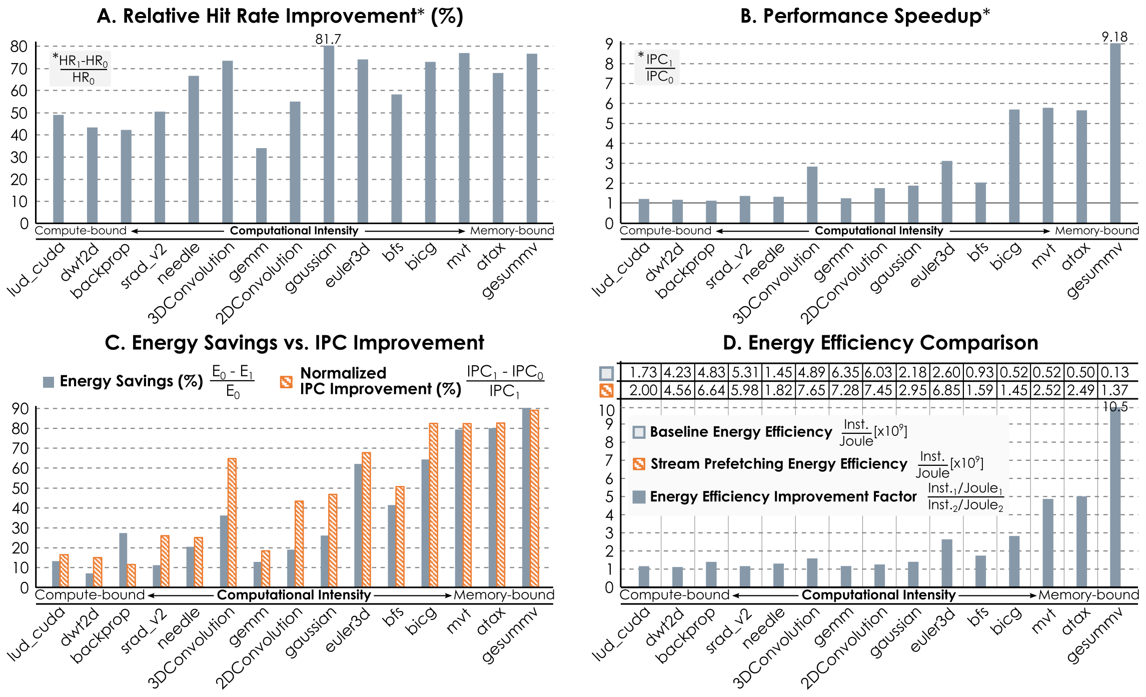


Figure 4.5: Relative L1 cache hit-rate improvement, resulting performance speedup, energy savings and energy efficiency for the adopted benchmark set.

To complete the evaluation of the proposed GPU stream prefetching mechanism, an energy efficiency study was also conducted regarding the adopted set of benchmarks. From the results presented in Fig. 4.5.C, it is possible to ascertain that the energy consumption reduction is directly related to the attained performance gains (i.e., resulting from the execution time reduction). As a result, the proposed mechanism allows energy consumption reductions from 7% up to 90%, corresponding to the most compute-bound and memory-bound applications, respectively. This is inherently reflected in the considered performance-energy consumption efficiency metric (Instructions/Joule), presented in the table and chart depicted in Fig. 4.5.D. It shows that the devised prefetching mechanism increases the efficiency of all the considered benchmark applications. As expected from the previously presented performance gains, a significant efficiency improvement is observed for memory-bound applications, leading to a maximum of 10.3x efficiency improvement in the `gesummv` application. The impact of the prefetching is also highlighted for the most computationally intensive applications, incurring in at least 1.2x energy efficiency improvements, in the `dwt2d` benchmark.

Hence, the obtained results showed that the sole exploitation of the proposed DSC as a stream prefetcher offers significant performance gains. In particular, it was able to account for the high contention imposed by the massively parallel processing architecture of GPUs, in turn improving its overall throughput and energy efficiency. Nonetheless, despite the attained gains, the proposed GPU stream prefetching mechanism only makes use of the DSC to populate the system's L1 caches with the required data. As such, further improvements can still be achieved by combining the DSC stream generation capabilities with the code optimization mechanisms provided by the proposed compilation tool (see Section 4.2.2).

4.2.2 Case Study B: Data Streaming on Modern General-Purpose CPUs

The stream prefetching approach discussed in the first case study improves the computing system's performance and efficiency by straightforwardly fetching and buffering data as fast as possible (later serving it to the L1 cache and processing cores upon request). However, the proposed prefetching mechanism is still integrated with the L1 cache memory and its miss handling control logic. In fact, while it does not require any prediction and monitoring logic to deploy the prefetching procedure, the devised integration can still potentially interfere with the L1 cache performance.

Conversely, this second case study considers a complete detachment of the data acquisition procedure from the L1 cache memory. It does so by deploying a full data streaming mechanism that works in parallel with the first level of the memory hierarchy of an x86-based CPU [106] (see Fig. 4.6), implemented in the Gem5 simulator [117].

The considered detachment is accomplished by connecting the *Stream Buffers* of the DSC directly to a processing core, through a memory-mapped interface configuration, supported by the compilation tool proposed in Chapter 3. Hence, prefetched data is fed to the processing core instead of populating the L1 cache. As a result, cache evictions caused by prefetched data are eliminated, in turn resulting in an implicit timeliness of data acquisition.

In accordance, an in-depth assessment of the viability of the proposed CPU data streaming

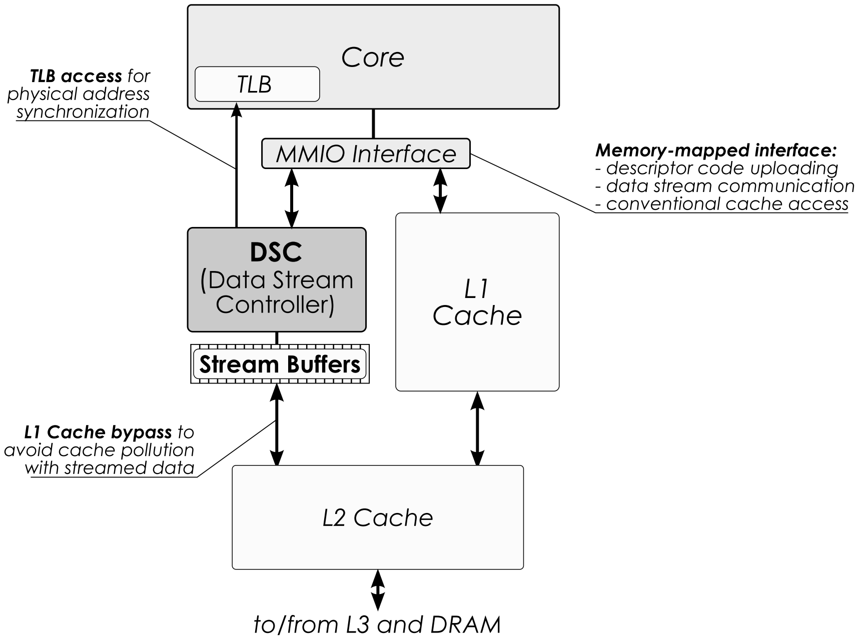

Core
TLB
TLB access for physical address synchronization
MMIO Interface
Memory-mapped interface:
- descriptor code uploading
- data stream communication
- conventional cache access
DSC
(Data Stream Controller)
L1 Cache
Stream Buffers
L1 Cache bypass to avoid cache pollution with streamed data
L2 Cache
to/from L3 and DRAM

Figure 4.6: Integration of the proposed DSC in a modern CPU memory subsystem.

approach as an alternative to predictive prefetching mechanisms is herein presented, by comparing it with two of the most prominent prefetchers in the state-of-the-art.

Meanwhile, the following subsection presents a brief background overview of CPU memory subsystems. Despite already discussed in detail in Chapter 2, it is important to better contextualize the implementation described in this case study.

4.2.2.A Current Prefetching Approaches

Memory access latency often limits the application performance on modern CPUs. In an initial attempt to account for such limitations, typical compile-time tools try to exploit some processor features (e.g., out-of-order execution) to reorganize data accesses, promoting concurrent data fetch and computation, and hiding the latency of data accesses. Nonetheless, while such optimizations aim at counteracting the limitations of the memory access chain, there is a performance gap that often can only be leaned by deploying dedicated data fetch hardware modules, such as prefetchers.

Accordingly, data prefetching methods are designed to deal with the intrinsic characteristics of the application memory access patterns that are most commonly encountered in general-purpose CPUs. In fact, this technology has evolved to a point where the main concern is no longer the prefetcher's capability to detect and predict memory access patterns, but instead the timeliness and effectiveness of the data acquisition procedure. This led to the emergence of new prefetchers [20, 22–24] that combine multiple hardware modules, with different data fetch granularities and prediction heuristics, across different cache levels. However, despite the offered improved throughput, resulting from a high accuracy and coverage of the data access prediction, the gains provided by each new generation of prefetchers are becoming limited.

As a consequence, alternative mechanisms are still necessary to overcome the throughput saturation that is being reached with current state-of-the-art prefetching technology. Accordingly, the proposed CPU data streaming mechanism aims at offering new levels of performance and efficiency by providing a two-fold advantage over predictive prefetching approaches. In particular, it not only eliminates delays in the memory access stream, resulting from possible prediction inaccuracies and monitoring overheads, but it also offers an extra acceleration at the level of the application code, through the stream code transformations provided by the compilation tool.

4.2.2.B Data Streaming Integration in an x86-based Architecture

The flexible interface that is provided by the proposed DSC makes its integration in a modern CPU memory subsystem entirely straightforward. Accordingly, for this case study, it is adopted a CPU architecture and memory chain similar to the one equipping the Skylake microarchitecture, based on the information released by Intel [106] (as detailed in Table 4.2). The adopted architecture topology assumes a single processing core connected to a 3-level cache hierarchy and a

Table 4.2: Adopted x86-based CPU configuration.

	CPU
Frequency	3 GHz
Core Model	x86-64, Out-of-Order

	Cache Configuration
Cache line Size	64 bytes
L1 I/D Cache	32 KB, 8-way, 4-cycle latency
L2 Cache	256 KB, 8-way, 20-cycle latency
L3 Cache	2 MB, 16-way, 36-cycle latency

	Main Memory
Size	4096 MB
DRAM Model	Micron MT41J512M8 11-11-11 DDR3-1600 8 banks/rank, 2 ranks/MC, t_{RCD},t_{RP},t_{CL}=13.75 ns, t_{CK}=1.25 ns

DRAM main memory module.

To integrate the DSC in the adopted architecture (see Fig. 4.6), collocated with the L1 data cache. The DSC generates the memory accesses extracted by the compilation tool and encoded with the proposed descriptor specification. Hence, it becomes solely responsible for fetching and storing the encoded data streams and for serving them (upon request) to the processor. Consequently, to provide the aimed detachment of the L1 cache memory, it is bypassed during the data stream acquisition and access procedures, avoiding cache pollution and early evictions resulting from poor data fetching timeliness.

Accordingly, the DSC's integration is realized by directly connecting it to the processing core (through a memory-mapped configuration, as described in Section 4.1.2) and to the lower cache hierarchy level (L2), as depicted in Fig. 4.6.

With such a configuration, the descriptor code generated by the compilation tool is sent to the DSC through a set of inline stores (as described in Chapter 3). Hence, the DSC is only required to accept the sent descriptor code and store it in the local *Descriptor Memory* (see Section 4.1.1). After the reception is finalized, the DSC starts working by generating the memory addresses for each stream and by fetching and buffering the corresponding data.

To provide an effective interface that allows the core to transparently perform data stream accesses, the DSC snoops the core's memory access channel to detect requests to the stream references assigned to each descriptor (see Chapter 3). Hence, when a request to a data stream is detected, the memory request is intercepted and directly served with data from the corresponding stream.

Notwithstanding, to deploy the DSC as a fully autonomous data streaming mechanism a final consideration must be taken in to account. It lies with the fact that in the CPU core, the application's data structures are usually allocated over a contiguous virtual memory address range.

Table 4.3: Considered benchmarks and kernels for the evaluation setup.

C-Polybench [99]	`2mm`	Multiple Matrix Multiplications
	`cov`	Covariance Computation
	`mvt`	Matrix-Vector Product and Transpose
	`seidel`	2D Seidel Stencil
	`syr2k`	Symmetric Rank-2k Update
	`trisolv`	Dense Triangular Solver
HPCG [100]	`spmv`(*)	Sparse Matrix-Vector Multiplication
	`symgs`(*)	Symmetric Gauss-Seidel (Sparse Triangular Solvers)
Rodinia [105]	`path`	PathFinder - 2D Shortest Path
	`srad`(*)	SRAD Diffusion Method

(*) Benchmarks characterized by indirect memory accesses.

Similarly, the mathematical model that is at the base of the proposed descriptor specification also assumes a contiguous address space to represent memory access patterns. However, the DSC operates on the physical memory address space, where data may not be stored in contiguous physical pages. While typical prefetchers avoid this issue by stopping the address generation procedure and wait for the CPU to resynchronize the physical address offset, this is not possible with the considered detachment from the address generation in the application code that is introduced by the proposed streaming mechanism. Accordingly, for this implementation, the DSC was equipped with page crossing detection logic. It works by comparing each generated memory address with previous one to detect the crossing of a page address range. Hence, upon its detection, the DSC makes use of a dedicated channel (depicted in Fig. 4.6) to consult the CPU's Translation Lookaside Buffer (TLB) and obtain the page offset for the newly generated address.

4.2.2.C Methodology

The proposed DSC architecture was implemented and evaluated in the Gem5 simulator [117], with the system configuration detailed in Table 4.2. The implemented system was evaluated with the same selection of benchmarks adopted in the preliminary evaluation of the proposed compilation tool, described in Chapter 3. The selection of benchmarks from the C-Polybench [99] and Rodinia [105] suites, and the kernels extracted from the HPCG [100] benchmark, are once again conveniently listed in Table 4.3.

To validate the proposed data streaming mechanism, it was compared with a baseline stride prefetcher and two state-of-the-art prefetchers. Table 4.4 presents the considered configuration for each of these setups:

- **Baseline:** The considered baseline configuration represents the most established prefetching scheme, corresponding to a typical stride prefetcher, comprising a stride/confidence table indexed by the instruction Program Counter (PC);

Table 4.4: Configurations for the reference prefetching and proposed DSC setups.

Baseline Setup	AMPM Setup
L1 Stride Prefetcher	L1 stride prefetcher (Baseline)
16x4-entry PC table	L2 AMPM prefetcher [20]
Confidence threshold: 4	256-entry access map
Prefetch degree: 4	5.2KB storage

BO Setup	DSC Setup (proposed)
L1 stride prefetcher (Baseline)	Data Stream Controller
L2 Best-Offset prefetcher [22]	16-entry Descriptor Table
256-entry RR table	16 32x8-Byte Stream Buffers
4KB storage	1KB Descriptor Memory

- **AMPM Prefetcher:** This configuration relies on an implementation of the AMPM prefetcher [20], that combines a memory access map and hardware pattern matching to detect all possible patterns in parallel. A stride prefetcher (provided by the Gem5 simulator [117]) is paired with the L1 cache to perform fine-grained prefetching;
- **Best-Offset Prefetcher:** This configuration makes use of the Best-Offset prefetcher [22], which acts as a generalization of next-line prefetching. It implements a selection mechanism that dynamically sets the prefetching offset depending on the application behavior and accounting for prefetch timeliness. The stride prefetcher from the Gem5 simulator [117] is also paired with the L1 cache.

4.2.2.D Experimental Evaluation

The proposed data streaming mechanism was evaluated by first studying the impact of code transformation (previously discussed in Chapter 3) in the number of issued instructions by the processor. Next, it was conducted a careful analysis of the DSC data streaming performance, by comparing its operation with the considered prefetching setups.

The impact of each prefetching method in the L1 data cache hit rate is shown in Fig. 4.7. For most polyhedral applications, the baseline stride prefetcher (`BASE`) is already able to provide high hit rates, due to the regularity of the memory accesses. Nonetheless, the `AMPM` and `BO` are still able to improve the cache performance, thanks to a high prefetching coverage. This, however, is not the case when the memory access complexity increases, as it can be observed in the `cov` and `mvt` benchmarks, where the datasets are large and require a significant amount of data reutilization; and in `spmv` and `symgs`, where the memory accesses present irregularities due to indirection.

On the other hand, when considering the proposed DSC, it is possible to ascertain that it takes advantage over the other prefetching methods through its memory access generation procedure. In particular, since the data stream acquisition initiates before the execution of the corresponding request, data is promptly available ahead of time. Moreover, the ability to exactly describe the sequence of addresses provides an important mitigation of data locality issues. This is highlighted

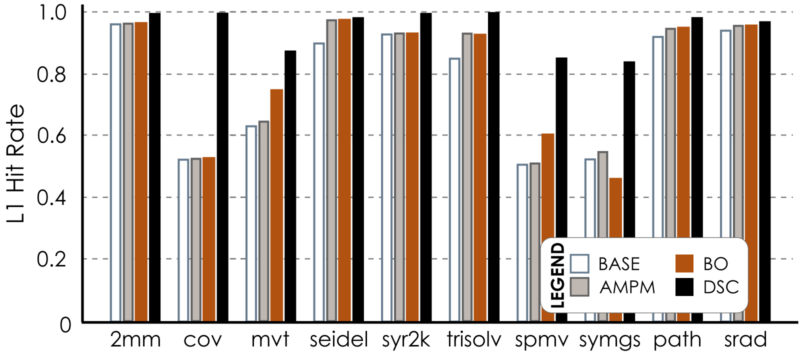


Figure 4.7: L1 hit rate comparison. For the proposed DSC, it is considered both the hit rates of the data cache and stream buffer.

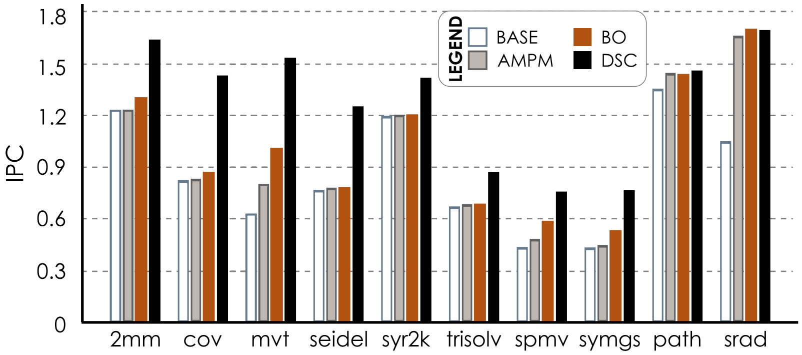


Figure 4.8: IPC comparison in absolute values.

by the observed average hit rate of 95% with the DSC, when compared to the average 79% and 80% hit rates observed in the AMPM and BO setups, respectively.

Hence, as a result of the attained memory access optimization, there is a direct impact on the performance of the processor. This is observable in Fig. 4.8 that shows the processor's average instructions-per-cycle (IPC) rate that is observed for each considered setup. Despite an actual reduction of the number of issued instructions, the proposed scheme allows for significant IPC increases of up to 2.4x, when compared to the baseline setup (BASE), and 1.7x, when compared with the AMPM and BO setups.

The acceleration provided by the reduction of code is particularly evident in the six polyhedral benchmarks (Fig. 4.9). Since the memory access patterns are easily detected by the BASE stride prefetcher, the improvements provided by the AMPM and BO are limited to their ability to move data to the L2 cache in a more timely manner (as it occurs in *mvt* due to its poorer data locality). However, due to the elimination of redundant array indexation (accounting for up to 40% of the achieved speedup - see Fig 4.10), the proposed data streaming is capable of further boosting the performance in such cases up to 2.63x over the BASE setup.

The advantages of data streaming are also reflected in the overall system performance, as shown in the speedup charts from Fig. 4.9. This is particularly evident when the memory access

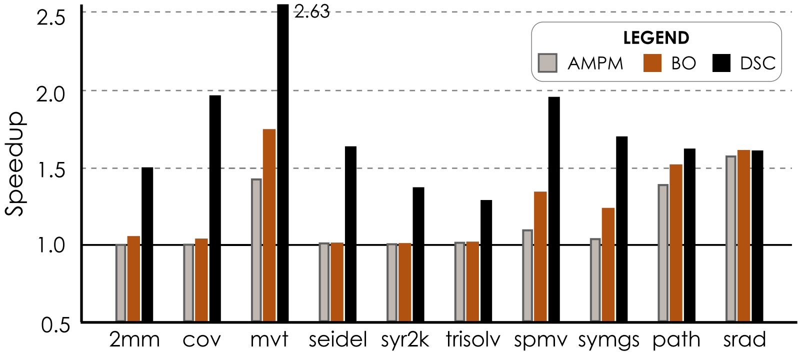


Figure 4.9: Overall speedup comparison (using the baseline setup as reference).

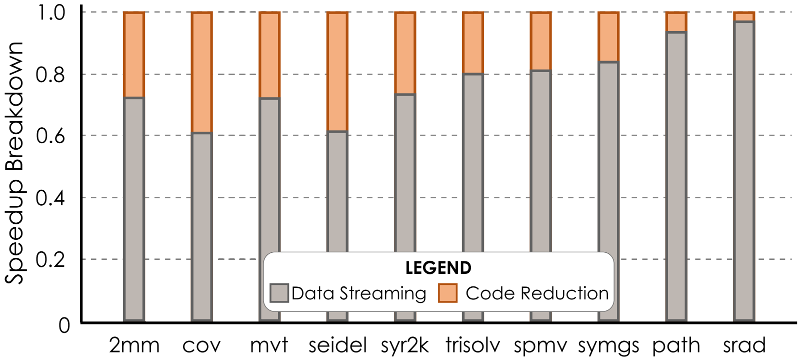


Figure 4.10: Breakdown of the contributions of data streaming and code reduction, for the attained performance gains.

pattern is characterized by poor spatial locality across the whole dataset, such as in `2mm` and `cov` benchmarks, where the dataset is composed of multiple large matrices. While the `AMPM` and `BO` prefetchers can easily detect the pattern for each access and feed the L2 cache, the dataset size inherently result in a large amount of L1 evictions. However, the DSC is capable of fetching and buffering data streams ahead of time, resulting in 1.5x and 2x speedups over the other setups, respectively in `2mm` and `cov`.

Performance gains are also evident in the presence of reutilization of datasets larger than the L1 cache capacity. Such is the case of the `mvt` and `path` benchmarks, where a large dense matrix is read multiple times. It is also possible to observe the gains resulting from the coarse data movement into the L2 cache with the `AMPM` and `BO` prefetchers, making it available for reutilization. Nonetheless, the data acquisition timeliness of the DSC still provides a performance boost of 10% for `path`, when compared to the other prefetchers. In the case of `mvt`, the matrix is also accessed in transposed order, resulting in L1 data-locality-related issues. However, the combination of the pattern description and code reduction provided by the proposed data streaming mechanism results in 1.9x and 1.5x speedups (in `mvt`) over the `AMPM` and `BO` setups, respectively.

Finally, the results obtained with the `spmv` and `symgs` benchmarks show the capability of the proposed data streaming mechanism to deal with indirect memory accesses. While the `BO` prefetcher's correlation heuristics provide visible performance improvements when compared to the base stride and AMPM prefetchers, it is still limited by the irregularity of the data accesses in spmv and symgs. However, the DSC is capable of producing the exact sequence of addresses (after indirection) ahead of time and without polluting the L1 with unnecessary data. Such an advantage (accounting for about 80% of the achieved speedup - see Fig. 4.10), when combined with the performed code reductions, results in 45% and 37% performance increases for the spmv and symgs, when compared to the BO setup.

Overall, the obtained results show that the combination of code reduction and efficient data acquisition provides a two-fold improvement that directly impacts the system performance. Not only there is a straightforward acceleration of the code, but there is also an increased memory access throughput. This is evident when initially comparing the obtained speedup for the considered AMPM [20] and BO [22] setups, using the BASE stride prefetcher as the reference. It is possible to observe that they already provided significant performance improvements. This is a direct result of their high prefetching coverage and accuracy and, in the particular case of BO, of the timeliness of the prefetching procedure [22].

When compared with these approaches, the proposed DSC, by itself, provides an increased memory access throughput (resulting from the exact data acquisition and implicit timeliness of the data streaming), matching and improving the performance over the other setups. However, the offered performance is further stretched through a straightforward acceleration arisen by the performed code reductions, providing further 20% improvements (see Fig 4.10) over predictive prefetching.

The performance gains offered by the proposed setup are emphasized by the attained speedups ranging from 1.3x to 2.6x when using BASE as the reference. Such gains represent a 40% and 30% performance increase (on average) over the AMPM and BO setups, respectively.

4.2.2.E Resource Overheads and Limitations

Despite the performance gains obtained by the proposed DSC, it requires simpler hardware structures when compared to other prefetchers. While the amount of storage for data streams (4KB stream buffering) and pattern description (1KB descriptor memory) is similar to the storage required by AMPM (5.2KB) and BO (4KB), the DSC requires lower logic complexity. In particular, the DSC only requires 6 adders and two register banks for the descriptor resolving architecture and a comparator to detect page crossing in the address generation procedure. Contrarily, the AMPM prefetcher [20] requires logic to match up to 256 stride patterns to find prefetch candidates on each L2 access. On the other hand, the Best-Offset prefetcher [22] requires adder logic to compute the position of a cacheline inside a page, while the recent request table is accessed through a hash function.

In what concerns the performance limitations, it is possible to observe that the performance attained by the DSC is only similar to that of the considered state-of-the-art prefetchers in the particular case of the `srad` benchmark. This results from the fact that the dataset used by this benchmark is small enough to fit in the L1 cache, and the data accesses performed by the DSC are done directly to the L2, in turn increasing the access latency. While this impact is mitigated by code reduction and data stream pre-acquisition, there is still room for improvement. In accordance, future implementations can consider a snoop-like access to the L1 cache tags (i.e., without causing demand misses) to directly copy data from the L2 and speedup the data access. Such an approach can further improve the offered gains, especially for applications that are characterized by high data reutilization.

4.2.3 Discussion

The devised case studies showed that the proposed data streaming mechanisms are viable alternatives to conventional predictive prefetching mechanisms, in off-the-shelf general-purpose computing systems.

In particular, the conducted experimentations for the GPU implementation demonstrated that the autonomous data stream generation procedure deployed by the DSC, by itself, allows a significant improvement of the data transfer procedure. This is a direct result both of the pre-acquisition of data before it is requested by the system's processing cores, and of the fact that it does not require the adoption of inaccurate prefetching prediction heuristics and monitoring delays. Moreover, the obtained results also showed that the deployed preemptive data fetching procedure is capable of mitigating the high contention that characterizes the massively-parallel memory access structure of a GPU device. As a consequence, it is possible to offer new levels of computing throughput, resulting in an overall energy efficiency improvement of the whole system.

Despite the advantages offered by the initial stream prefetching implementation, the proposed CPU approach took a step further, by deploying a fully functional data streaming mechanism detached from the system's cache hierarchy. The devised mechanism exploits the combination of the DSC and the code transformations of the compilation tool proposed in Chapter 3, to deploy a viable alternative to predictive prefetching approaches. In fact, the obtained results for this implementation showed that the proposed mechanism can outperform two of the most prominent state-of-the-art prefetchers. This results from the provided two-fold improvement that directly impacts the whole system's performance. Specifically, it is attained a straightforward acceleration of the application code (in the processing core) and an increased memory access throughput (offered by the DSC).

Nonetheless, while the proposed mechanisms (based on the DSC) already provide significant performance and efficiency gains, their implementation in off-the-shelf devices is bound by the conventional memory access infrastructures of such systems. Consequently, by designing a dedicated data streaming infrastructure, it is still possible to take a step further from the sole exploitation of the DSC. Specifically, it can be considered the deployment of additional data streaming mechanisms and alternative data transfer schemes, that could not otherwise be deployed in the consolidated communication infrastructures of conventional computing systems.

4.3 In-Cache Stream Communication Paradigm

The proposed deployment of the DSC in stream prefetching mechanisms showed to be a viable alternative to conventional predictive prefetching. However, as it was discussed in the previous section, it is still possible to offer further gains in performance and efficiency if the DSC is placed close to the main memory, supported by a communication infrastructure that exploits more sophisticated data streaming mechanisms. In particular, memory bandwidth and data transfer efficiency can be maximized with specialized stream manipulation techniques (such as data reorganization). On the other hand, it is possible to tackle the contention present in highly parallel architectures (only partly mitigated by the preemptive data acquisition of the proposed stream prefetching) with the utilization of broadcast and point-to-point communication operations to manage the data flow between the main memory and the system's processing cores. Naturally, such techniques and mechanisms must be deployed without losing general-purpose capabilities.

Accordingly, this section proposes a new In-Cache Stream (ICS) communication paradigm that deploys a full data streaming communication scheme in a conventional cache-coherent memory hierarchy, by allowing the infrastructure to exploit both conventional *memory-addressed* and *packed-stream* data accesses, simultaneously and cooperatively. To achieve such a goal, the proposed ICS communication paradigm was implemented in a custom many-core accelerator

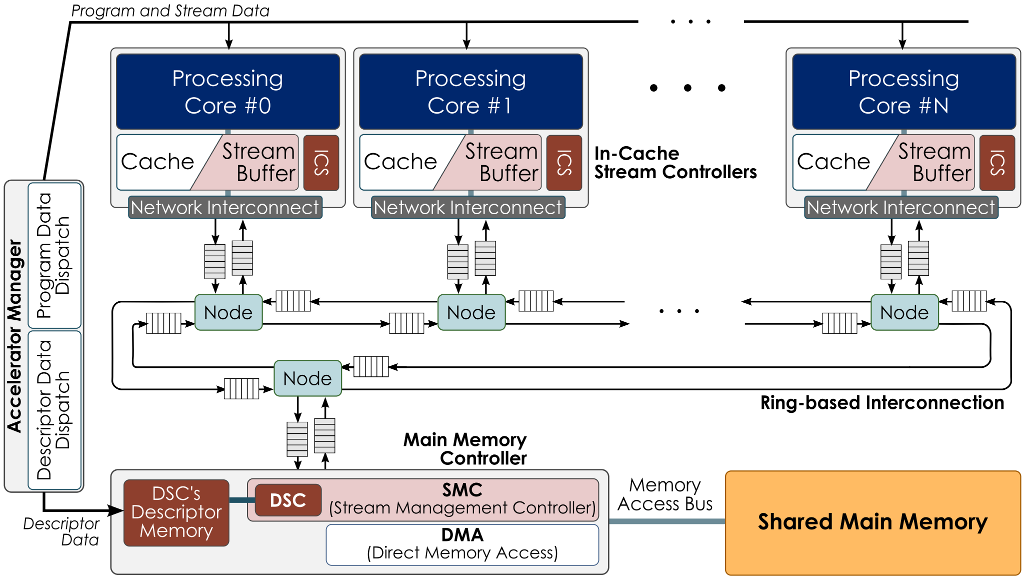


Figure 4.11: In-Cache Stream communication infrastructure overview, comprising the ICS controllers connected to each core, the SMC (based on the DSC) and a dedicated data transfer interconnection.

(depicted in Fig. 4.11) that deploys a scalable processing infrastructure, composed of:

- **Core-coupled ICS controllers**, that allow the seamless adaptation of local cache memories to provide support for data streaming;
- **A dedicated data transfer interconnection**, which deploys point-to-point and broadcast communication schemes, and that simultaneously supports memory-address-based and stream-based data communication through a dedicated message-passing protocol;
- **A hybrid main memory controller**, comprising a *memory-addressed* Direct Memory Access (DMA) module and a *packed-stream* memory-aware Stream Management Controller (SMC), based on the proposed DSC. The SMC deploys a set of memory bandwidth optimization and data reutilization and reorganization techniques, through in-time stream manipulation.

The accelerator's execution is controlled by a centralized hardware manager, which is responsible for assigning to each processing core the program and descriptor data generated by the compilation tool proposed in Chapter 3. It does so by dispatching the required descriptor data to the SMC and the stream references assigned to each ICS controller, together with the compiled program (see Fig. 4.11). The communication between the manager and each component is performed via a direct unidirectional channel.

4.3.1 In-Cache Stream Controller

The key idea of the proposed ICS paradigm is to allow each of the system's processing cores to seamlessly switch their local communication scheme between conventional *memory-addressed* and *packed-stream* data access paradigms. In particular, to avoid a mutually exclusive adoption of the two paradigms, which could otherwise result in potential performance penalties in non-pure streaming applications, a dedicated ICS controller (placed close to the core) was developed that allows the utilization of the set-associative ways of each local cache memory to perform data streaming to/from the core. With this approach, it is possible to support both data-access types not only separately, but also simultaneously, by balancing the ratio of set-associative ways and buffers in the cache memory.

Hence, in the devised ICS paradigm, each core views its local *n*-way set-associative cache memory as a set of m cache ways plus $n{-}m$ stream buffers, each capable of holding multiple streams (as depicted in the example of Fig. 4.12). Such a topological reorganization, together with the adoption of a switched control structure, allows the communication infrastructure to instantly adapt the cache memory according to the instantaneous requirements of a running application. Moreover, it conveniently supports mixed scenarios composed of compile-time predictable and non-predictable (or runtime generated) memory access patterns.

Accordingly, each cache memory is simultaneously managed by two independent modules (see Fig. 4.13): a *cache controller* and a *stream controller*. The default *memory-addressed* com-

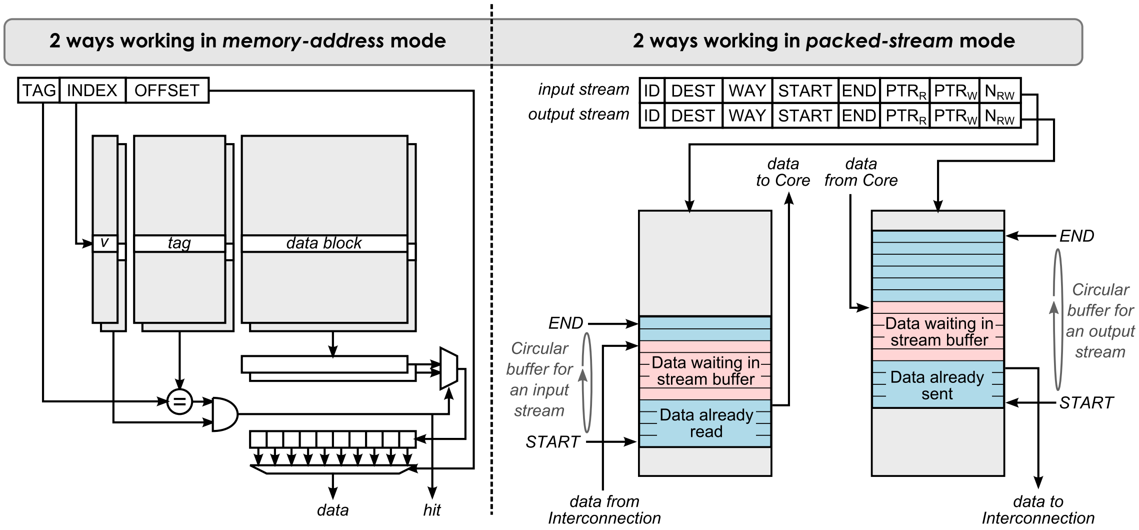

2 ways working in memory-address mode
2 ways working in packed-stream mode
TAG
INDEX
OFFSET
v
tag
data block
=
data
hit
input stream
output stream
ID
DEST
WAY
START
END
PTR$_R$
PTR$_W$
N$_{RW}$
data to Core
data from Core
END
START
Circular buffer for an input stream
Data waiting in stream buffer
Data already read
Circular buffer for an output stream
Data already sent
data from Interconnection
data to Interconnection

Figure 4.12: Example configuration of a 4-way set associative memory after a control switch, becoming configured to use 2 ways for conventional memory-address mode and 2 ways for stream mode.

munication paradigm is assured by the *cache controller*, by using any arbitrary replacement and write policies. On the other hand, the *stream controller* is used to conveniently adapt and reuse the resources of an *n*-way set-associative cache memory to implement the buffering structures required by a *packed-stream* communication.

At any instant, the cache memory configuration is defined by a simple register, shared by both controllers, indicating which ways are accessible (or owned) by each controller. The register is modified when a new data stream is received (either from the core or the underlying interconnection), according to the information stored in a dedicated *Stream Table* (programmed by the accelerator's manager). Accordingly, the cache memory is configured by default with each way owned by the *cache controller*, and, upon request, the ownership of individual ways can be changed to the *stream controller*. The control switch for each way is performed transparently from each of the controllers, eliminating any unnecessary waiting times that could otherwise degrade performance.

The ICS controller communicates with its assigned processing core through a memory-mapped interface (see Fig. 4.13) that redirects data accesses either to the *cache controller* or to the *stream controller*. On the other hand, the communication with the remaining communication infrastructure is assured by a *Message Dispatcher* module (see Fig. 4.13), which transparently handles the communication of the data into, or out of, the ICS controller, through a buffered interface.

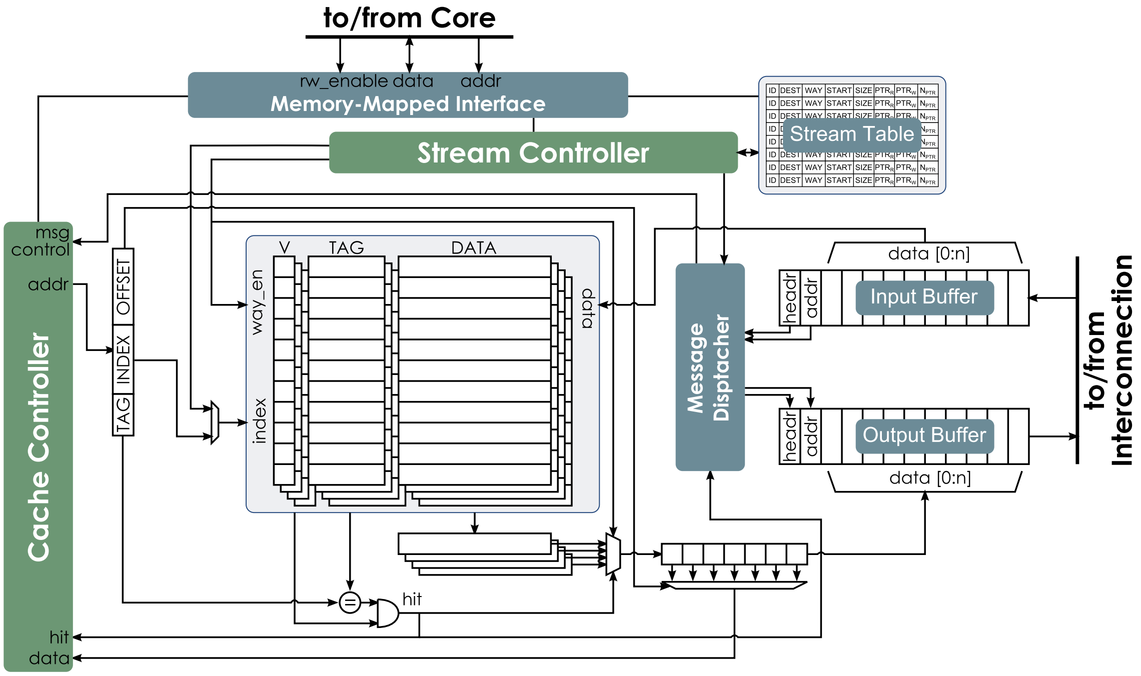


Figure 4.13: ICS controller architecture. The *cache controller* and the *stream controller* (supported by the information stored in the *Stream Table*) perform an exclusive access to an n-way set-associative cache memory depending on the requests received from the core and the communication infrastructure.

4.3.1.A Cache Controller

The conventional *memory-addressed* operation is the default communication paradigm and works independently of the underlying cache protocol (see Fig. 4.13), hence supporting any conventional replacement and write policies. Notwithstanding, for the herein presented design, the ICS *cache controller* was prototyped using a simple hardware structure. It deploys a write-through-invalidate, write no-allocate snooping protocol (to limit the number of coherency-related messages in the interconnection) on the *n*-way set-associative cache memory, managed by a Pseudo-Least Recently Used (LRU) replacement policy (implemented by a binary decision tree algorithm, supported by a with a register with *n*-1 bits per cache index).

According to the adopted protocol, only one *valid* state bit (together with the *tag* field) is required per cache line and the cache access time is limited to two clock cycles (disregarding cache miss penalties). Hence, upon a core request, the cache memory is immediately accessed. The hit/miss analysis is performed in the second clock cycle, and the hit/miss-related action is taken according to the adopted cache protocol.

As a result of the adopted write-through-invalidate and write no-allocate protocol, the core requests are only answered with a *wait* state when there is a read miss, until the required data is fetched from the memory subsystem. At a write miss scenario, the written data block is immediately sent to the main memory and it is followed by an invalidation broadcast, requiring no return information to the *cache controller*, thus minimizing the waiting times and the number of on-the-fly messages in the communication infrastructure.

4.3.1.B Stream Controller

To reuse the resources of the *n*-way set-associative cache memory for stream-based communication, the cache memory access mechanism has to be conveniently adapted. Instead of relying on conventional memory-addressed accesses, in this operation mode, each associative way is regarded as an independent buffering structure that is accessed by a dedicated set of read and write pointers (which identify the memory region where a stream is stored). This transforms the *n*-way set-associative memory in m independent stream buffers, each capable of storing multiple streams, while allowing the remaining $n-m$ ways to be accessed using traditional *memory-address* load/store operations (see Fig. 4.12).

Naturally, contrarily to the default memory-addressed communication, the stream-based paradigm requires a set of auxiliary data structures (stored in the programmable *Stream Table*), to accommodate the information and the state of every stream currently stored and handled by the controller. Each table entry (depicted in Fig. 4.13) comprises: *i)* a unique stream identifier, corresponding to a stream reference encoded in the proposed descriptor specification (detailed in Chapter 3); *ii)* the way used for buffering the stream; *iii)* pointers to the start and end of the buffering region within the way; *iv)* pointers for identifying the core local read/write positions in the inbound/outbound stream, which effectively allow implementing a circular read/write buffer between the region defined by the stream start and end pointers; *v)* a stream destination, represented the corresponding component (core or memory) identification; and *vi)* a read/write pointer for identifying the current read/write position for the Message Dispatcher module (see Fig. 4.13).

Hence, whenever a read/write request is performed in the core interface for a given stream reference, the local cache memory is accessed according to the information depicted in the *Stream Table*, with the consequent update of its read/write pointers. The interface with the communication infrastructure is performed by accepting the incoming stream messages and by storing them in the local memory (according to the targeted set-associative way and pointer-values in the *Stream Table*) and by updating the write pointer. Outgoing streams are automatically sent (according to the destination stored in the *Stream Table*) as soon as they become available. Moreover, upon their transmission, stream data is maintained in the local memory until it is overwritten by new streams, allowing the data to be reused by the core. As a result, depending on the assignment of each stream, data can be made persistent throughout the whole execution.

4.3.1.C Communication Interfaces

The communication between the ICS controller and the processing core is assured by a dedicated memory-mapped interface (see Fig. 4.13). It provides a generic and straightforward request

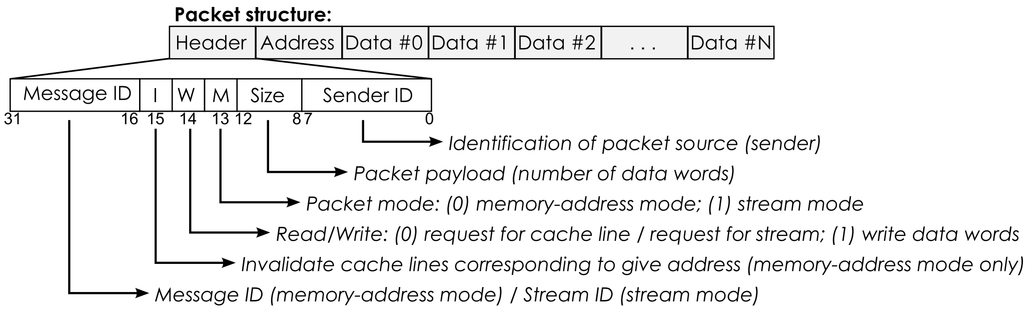

Packet structure:
Header
Address
Data #0
Data #1
Data #2
. . .
Data #N
Message ID
I
W
M
Size
Sender ID
31
16
15
14
13
12
8
7
0
Identification of packet source (sender)
Packet payload (number of data words)
Packet mode: (0) memory-address mode; (1) stream mode
Read/Write: (0) request for cache line / request for stream; (1) write data words
Invalidate cache lines corresponding to give address (memory-address mode only)
Message ID (memory-address mode) / Stream ID (stream mode)

Figure 4.14: Message-passing packet-format, including the header specification.

redirection mechanism that distinguishes between memory-addressed and stream-based read-/write requests. It does so by comparing the requested address with the stream references stored in the *Stream Table*. Accordingly, each request to the cache address space is redirected to the *cache controller*, in the conventional memory-addressed communication paradigm. On the other hand, requests for the stream address space are handled by the *steam controller* that accesses the local memory according to the accessed stream's read/write pointers (stored in the *Stream Table*).

Additionally, the ICS controller interfaces with the communication infrastructure by using two input/output register-based buffers, managed by a *Message Dispatcher* module. Each buffer is capable of accommodating a complete message to/from the interconnection (see next section), allowing cache lines and streams to be split into message packets for an output message and joined in a single word from an input message. Such an approach simplifies the transmission of cache lines and streams with the adopted message protocol. Depending on the assigned message type (see the protocol in the following section), incoming messages are redirected by the *Message Dispatcher* either to the *cache controller* or the *stream controller*. Conversely, outgoing messages are generated by the *Message Dispatcher* according to requests performed by the controllers.

4.3.2 Communication Infrastructure and Protocol

Since the ICS infrastructure exploits both *memory-addressed* and *packed-stream* data accesses, it is necessary to provide a data transfer mechanism between the system's components that is independent of the data access paradigm. Accordingly, to abstract the underlying data accesses, a simple message-passing protocol was adopted. It consists of a `header`, an optional memory `address` and a number of `data` words that, at most, add up to the size of a cache line (see Fig. 4.14). As referred above, each ICS controller provides a message dispatching module that is responsible for packing and unpacking data blocks according to the defined message format.

4.3.2.A Interconnection Architecture Considerations

Naturally, such a message-passing protocol could be deployed in any type of interconnection bus. However, to deploy the aimed point-to-point and broadcasting data communication schemes, it is necessary to adopt a dedicated data transfer interconnection that is capable of efficiently deploying such schemes (both in terms of latency and hardware complexity). While a shared crossbar-switch-type bus, providing a direct interconnection between all components, might be an obvious choice in terms of functionality, the amount of connection paths tends to grow with a power of 2 of the number of connections (all components are connected between them). Alternatively, in the presence of high numbers of connected components, two-dimensional networks-on-chip (e.g., grids or meshes) could be viewed as viable solutions. However, despite requiring a lower hardware complexity per node (component), they usually rely on costly packet routing protocols that require complex control structures.

Accordingly, it is possible to achieve a viable trade-off by deploying a ring-based interconnection. While such a structure is capable of interconnecting all the system's components (to deploy the aimed data transfer schemes), the packet transmission procedure is simply implemented by a test-and-forward architecture (in each node/component). Furthermore, by making the communication bidirectional, it is possible to deploy an interconnection with a maximum latency proportional to half of the number of connected nodes/components.

As a result of the adopted interconnection architecture, each of its nodes is assigned with a unique identification and is responsible for routing the incoming messages to/from its two adjacent nodes (right and left) and to/from its connected component. To attain such an objective, a completely pipelined architecture ensures that each packet requires a single clock to be analyzed and forwarded. This results in a minimum packet communication latency of a number of clock cycles given by the distance between the sender and receiver nodes. To overcome any contention caused by arriving packets, a simple round-robin priority function was devised that rotates the priority between channels upon the completion of a message transmission. Moreover, each component interfaces with the communication infrastructure through two input/output register-based buffers, further mitigating access contention through intermediate buffering.

4.3.2.B Data Transfer Mechanism

Communication between the interconnection's nodes is achieved through an AXI-Stream protocol [118] interface. The addressing is defined by a modified `tdest` signal, where the most-significant-bit is used to signal a broadcast message, and the remaining bits represent the identification of the receiver. Furthermore, a routing mechanism is responsible for sending the received message packets to their appropriate destination. This is done by comparing the message destination address with the node own physical address. Hence, whenever the message is directed to the node (itself), it is immediately forwarded to the connected component. Otherwise, it is merely forwarded to the next adjacent node, maintaining the same communication direction. In the particular case of a broadcast message, the packets are simultaneously sent both to the connected component and to the adjacent node (the broadcast message is evicted as soon as it reaches the node where it was originally sent from).

4.3.3 Memory-Aware Data Stream Generation

In a stream-based communication environment, each component must be capable of generating/processing its data streams. Although each core can manage the flow of its data streams with the aid of the ICS controllers, streams fetched from the main memory require a dedicated structure to handle the distinction between a *memory-addressed* access and a *stream*-based communication. This is accomplished through a dedicated main memory controller, composed of *i)* a low-profile address-based DMA controller, to perform address-based memory operations upon memory access requests performed by the cores; and *ii)* a memory-aware SMC, which automatically generates and saves the streams, respectively by fetching/storing data from/to the main memory.

The SMC itself relies on the DSC architecture to generate memory access patterns and on a *Stream Table* (similar to that of the ICS controller). Hence, its operation is managed according to the descriptor data and stream information that is sent by the hardware manager at the beginning of the execution. Accordingly, the stream generation procedure is performed automatically upon reception of the descriptor data. The stream storing procedure (for streams generated by the ICS controllers), on the other hand, is automatically performed upon the reception of a stream from the data transfer interconnection.

With the goal of optimizing the memory accesses for stream generation, the SMC deploys two specialized modules that maximize the main memory bandwidth (see Fig. 4.15.B), specifically:

- **A dedicated burst controller** used to take advantage of the main memory burst capabilities, to maximize its throughput;
- **A reorder buffer** that allows the fetching and intermediate buffering of coarse-grained memory regions where the data-patterns generated by the DSC are contained, to avoid costly data accesses to non-contiguous memory blocks.

4.3.3.A Memory Burst Controller

Despite being efficient when dealing with fast-access local memories (e.g., BRAMs), address generation units often struggle to perform requests to long-latency external memories (e.g., DDR3 memories). This is mainly because these memories usually impose costly latency overheads (up to tens of clock cycles per request). To mitigate such delays and increase the throughput, most memories offer the ability to burst data transfers of contiguous memory regions. Accordingly, the DSC's AGU was duplicated and coupled with a specially devised burst controller (see Fig.4.15.B)

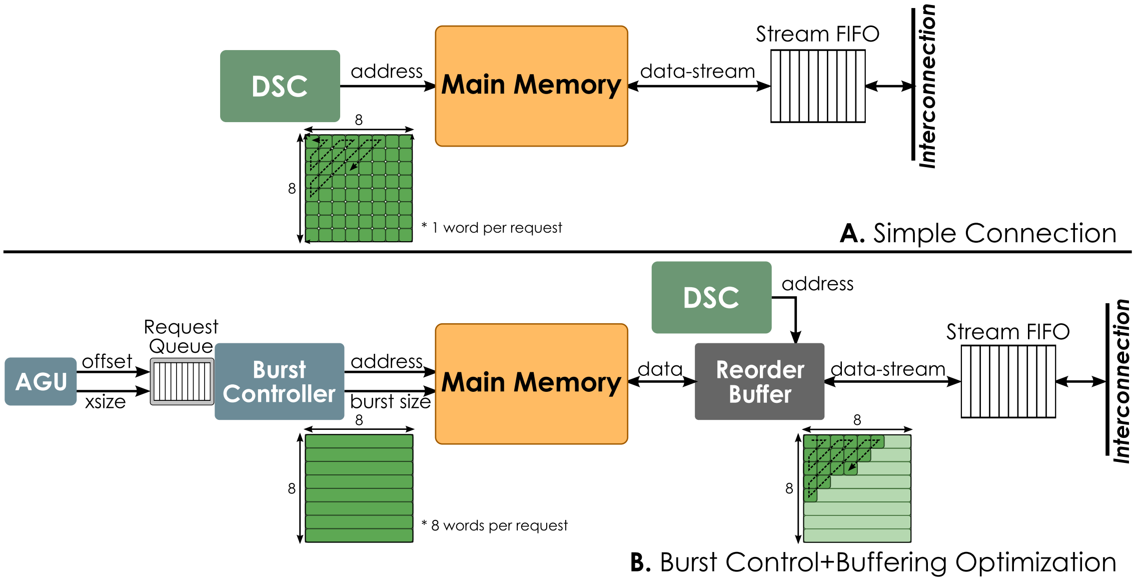


Figure 4.15: Memory requests performed by the DSC for the zig-zag pattern descriptor, when directly connected to the main memory (A) and to the burst controller and reorder buffer optimizations (B).

that generates and manages burst requests to the main memory, based on the descriptor being resolved.

However, instead of sending each address (indexed by a given descriptor), the AGU sends the descriptor's base address and the first pair tuple to the burst controller, indicating a contiguous memory region to be accessed (see Fig 4.15.B). The burst controller is then responsible for splitting the request in one or more burst requests (depending on the maximum burst length supported by the memory and its communication protocol), which is performed by a straightforward register-based increment/subtract architecture.

4.3.3.B Reorder Buffer Optimization and Stream Manipulation

Although the burst controller, by itself, could increase the main memory throughput for most regular access patterns, dealing with complex patterns, characterized by poor data locality, still poses a difficult challenge. One example is a zig-zag pattern (discussed in the preliminary evaluation presented in Chapter 3). Due to its diagonal scanning pattern, the direct application of the burst controller would not improve the resulting memory throughput. However, these complex patterns are usually contained in regular memory regions that can be fetched from the main memory and temporarily buffered in fast-access memory structures. In particular, a zig-zag pattern can be scanned in N×N data blocks, and each block can be individually fetched and stored in a specially devised reorder buffer, which can then be accessed by the DSC with a zig-zag descriptor, as depicted in Fig 4.15.B.

To offer such functionality, each stream can be regarded as a block of contiguous data, where

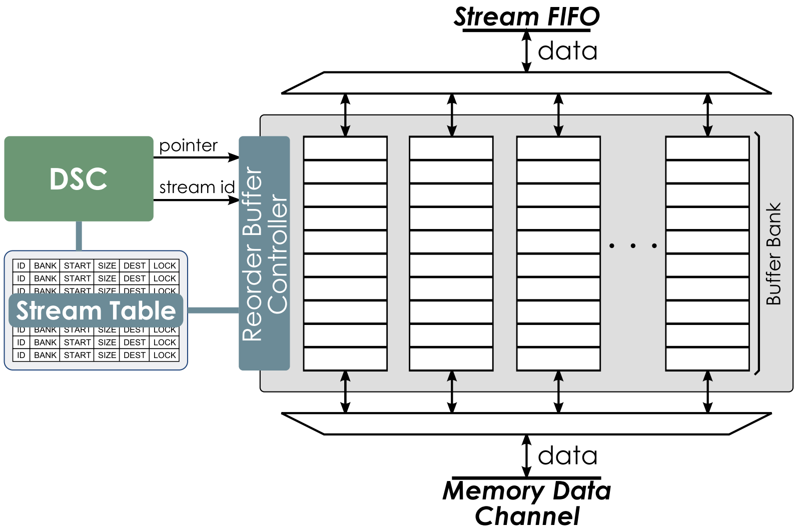


Figure 4.16: Reorder buffer architecture and its connection to the DSC and *Stream Table*.

the data block is stored starting at memory address (offset) `0x0`. By adopting such an analogy, it is possible to extract a particular data block from a previously obtained data stream by only adding a stream start pointer to an offset generated by the DSC. This allows applying the descriptor specification to straightforwardly extract sub-patterns from on-the-fly data streams. Furthermore, it is possible to easily implement run-time stream manipulation operations (e.g., stream splitting and merging) based on the descriptor specification, as well as rerouting operations over the flowing streams between the cores and memory.

Hence, the DSC is used to access the new reorder buffer (obtaining new data blocks as soon as they become available in the buffer), while the duplicated AGU is used to fetch the data blocks from memory, aided by the burst controller (see Fig.4.15). To be properly accessed by the DSC, the devised reorder buffer (depicted in Fig. 4.16) is composed of a set of stream buffer banks, each able to store multiple streams. In fact, its control architecture is similar to that of the *stream controller* from the ICS controller. It works by relying on the information stored in the SMC's *stream table* to perform the required stream manipulation and generate the final data streams.

4.4 In-Cache Streaming Evaluation

This section presents an in-depth evaluation of the proposed ICS communication paradigm, support by an implementation of the devised accelerator on an XC7VX485T Virtex-7 FPGA device. Such an implementation allowed the assessment of a fully functional prototype (instead of relying on simulation tools), in terms of hardware resources, timing measurements, and power supply requirements.

4.4.1 Methodology

The experimental evaluation is performed in three stages. Initially, it is performed a hardware resource study depicting the area overheads of each individual component of the accelerator. Next, it is assessed the address generation efficiency of the proposed DSC (also compared with the same state-of-the-art solutions adopted in the preliminary evaluation presented in Chapter 3) and the resulting memory throughput optimization of its deployment by the SMC. The evaluation is concluded with a thorough assessment of the implemented prototype regarding the performance and energy efficiency improvements, and its comparison with an implementation of the state-of-the-art Access Map Pattern Matching (AMPM) [20] prefetcher.

For the hardware evaluation, all the required Synthesis and Place&Route procedures were performed using Xilinx ISE 14.5. The power consumption of each of the system's components was estimated with the Xilinx Power Estimation toolchain, and the DDR3 memory power consumption was calculated according to the vendor's guidelines and estimation tool [119].

4.4.1.A Accelerator Implementation and Prototyping Framework

The accelerator implementing the proposed ICS paradigm (see Fig. 4.11) was configured with a computing infrastructure composed of a variable amount of cores (ranging between 1 and 64). Each core comprises an MB-LITE [120] processor modified to support vector instructions [28] and a private scratchpad for program data.

The ICS infrastructure prototype was implemented in a Xilinx VC707 development board, equipped with an XC7VX485T Virtex-7 FPGA and a 1GB DDR3-1600 SODIMM memory module (MT8JTF12864HZ-1G6G1). The DDR3 module is accessed through a Xilinx IP MIG controller, comprising an AXI4-Full interface with a 20 clock cycle overhead per transfer request (independently of its length). The AXI4 protocol [121] allows burst transfers of up to 256 words and an attainable bus throughput of 400MB/s per channel direction (32-bit words @ 100MHz), resulting in a maximum theoretical throughput of 800MB/s.

4.4.1.B Experimental Setups and Configurations

The preliminary evaluation of the proposed descriptor specification presented in Chapter 3 was complemented with an evaluation of the address generation efficiency of the DSC, by comparing it with the Hotstream framework's Data-Fetch Controller (DFC) [26] and the Xilinx AXI DMA [102] controller (illustrating the performance of the Advanced Pattern-based Memory Controller (APMC) [54]).

To evaluate the full ICS implementation, the following accelerator configuration setups were considered:

- `BASE`**:** The considered baseline setup comprises a configuration with a conventional cache-based system (i.e., without the ICS controllers and the SMC), with the main memory access solely managed by the low-profile DMA controller;

- `uICS`: This configuration comprises an ICS infrastructure with the SMC's DSC directly connected to the main memory (i.e., without the burst controller and reorder buffer);
- `ICS-SMC`: This setup implements the full ICS infrastructure with the complete memory-aware SMC;
- `AMPM`: The final configuration considers the inclusion of the state-of-the-art AMPM [20] in the `BASE` setup (paired with each cache). It relies on an idealized behavioral model implemented and simulated according to the architecture description provided in [20].

All the considered infrastructure setups were connected to the DDR3 controller through an AXI4-Full bus interconnection (memory access bus in Fig 4.11). Accordingly, the main memory controller for the `BASE`, `AMPM` and `uICS` setups was configured to perform cache-line-sized burst transfers. On the other hand, for the `ICS-SMC`, the burst controller was set to use a maximum burst length of 256 words (the same as the AXI4 protocol), meaning that it splits the requested data transfers in individual 256-word burst requests.

The proposed descriptor specification was configured with the same parameter sizes adopted in the preliminary evaluation presented in Chapter 3, specifically:

- *Context Header*:
  - Header - `acc`: 8 bits, `next`: 16 bits;
  - Descriptor References - $\texttt{a\_id}_{acc}$: 16 bits.
- *Access Descriptor*:
  - Header - `stream`: 32 bits, `base`: 32 bits, `dim`: 8 bits, `mod`: 8 bits;
  - Pairs - $\texttt{xsize}_k$: 32 bits, $\texttt{stride}_k$: 32 bits.
- *Field Modifier Descriptor*:
  - Header - `target`: 16 bits, `dim`: 8 bits;
  - Pairs - $\texttt{xsize}_k$: 32 bits, $\texttt{stride}_k$: 32 bits.
- *Indirection Descriptor*:
  - Header - `data`: 16 bits, `dim`: 8 bits;
  - Pairs - $\texttt{target}_n$: 16 bits, $\texttt{a\_id}_n$: 32 bits.

Similarly, the representations of the considered Hotstream framework [26] and the AXI DMA [102] approaches were also configured with the same 32-bit parameter sizes as in Chapter 3.

Instead of adopting a 3-level cache hierarchy (as in Section 4.2.2), it was considered a configuration closer to those typically deployed in embedded systems [122]. Accordingly, each core is integrated with an 8KB 4-way set-associative data cache memory with 64-Byte cache lines, deploying a write-through-invalidate, write no-allocate snooping protocol, managed by a binary-tree-based pseudo-LRU replacement policy. The SMC was configured with a 2KB descriptor memory (scratchpad), while the reorder buffer comprises a 4KB, 32-bit line, direct-mapped mem-

Table 4.5: Adopted benchmark applications and their corresponding datasets, data access patterns and communication characteristics.

BENCHMARK	INPUT SIZE	DATA PATTERNS	COMM./DATA MANAGEMENT[1]
Blocked Matrix Multiplication	128×128 8×8 blocks	*Tiled*	Hybrid, Broadcast Data Reuse
Biological Sequence Alignment [123]	128× 1024 queries 4096 reference	*Diagonal* *Striped*	Stream, core-core, Broadcast Data Reorganization
Histogram Equalization	256×256 image	*Tiled* *Linear*	Hybrid, core-core, Broadcast, Data Reuse
2D Poisson Equation Solver[2] [124]	128×128 grid 100 iterations	*Tiled* *Linear*	Stream, core-core Data Pipelining
Convolutional Neural Network[3] [125]	32×32 image 142× 5×5 kernels	*Tiled*	Stream, core-core, Broadcast Data Reuse+Pipelining

[1] As used in the In-Cache Stream setups
[2] Jacobi parallel iteration method (striped decomposition)
[3] For image classification

ory structure. The AMPM prefetcher was configured with a prefetch degree of 4 (four prefetch requests per memory access), which provides the best prefetching performance while avoiding over-prefetching, as reported in [20].

4.4.1.C Workloads

To evaluate the DSC data-pattern generation efficiency and the SMC memory throughput optimization capabilities, the same set of memory access pattern samples used in the preliminary evaluation presented in Chapter 3 was used, namely: *Linear*; *Tiled*; *Diagonal*; *Zig-Zag*; and *Greek Cross*. For the SMC evaluation, it was also considered an additional *Striped* pattern, representing a variation of the *Diagonal* pattern, where all diagonals are split and processed in a stripped (or banded) approach.

To evaluate the gains offered by the data-transfer and communication capabilities of the ICS infrastructure, benchmarks were implemented based on real applications (see Table 4.5), namely: *i)* a standard block-based matrix multiplication kernel; *ii)* a biological sequence alignment application [123]; *iii)* an histogram equalization kernel; *iv)* a 2D Poisson equation parallel solver [124]; and *v)* a convolutional neural network for image recognition [125]. Each application was particularly considered to provide a comprehensive evaluation of the ICS infrastructure with different combinations of memory access patterns and communication schemes (see Table 4.5).

4.4.2 Hardware Resources Overhead

The obtained results of the implementation of the ICS infrastructure and accelerator in the considered Virtex-7 FPGA device are presented in Table 4.6. Despite the added versatility of the offered streaming capabilities, the results obtained for the devised ICS controller represent a 25%

Table 4.6: Resource usage for each component of the ICS communication infrastructure.

	Available Resources	Baseline Cache Ctrl.[2]	In-Cache Stream Ctrl.[2]	DSC Architecture	Memory-Aware SMC (w/ DSC)	Ring Node	Modified MB-LITE [28]
Slices	75,900	1896 (2.5%)	2370 (3.1%)	605 (.8%)	852 (1.1%)	155 (.2%)	1753 (2.3%)
LUTs	303,600	3602 (1.4%)	4367 (1.4%)	1466 (.5%)	1666 (.5%)	297 (.1%)	4566 (1.5%)
Registers	607,200	365 (.1%)	1176 (.2%)	19 (.1%)	991 (.2%)	164 (.1%)	1013 (.2%)
BRAM	3,090	16 (.5%)	16 (.5%)	7 (.2%)	9 (.3%)	2 (.1%)	6 (.2%)
Static Power[1]		210.42					
Dynamic Power*[1]		89.3	91.2	34.1	43.2	10.3	138

* @100 MHz

[1] Power consumption values displayed in mW

[2] Considering an 8KB 4-way set-associative memory w/ 64B cache lines

increase in hardware resource slices (corresponding to a 20% increase in the number of LUTs), when compared to the baseline cache controller. This difference is a result of the additional logic that is required by the included stream controller and stream table structures. On the other hand, the register utilization is increased by approximately three times, mostly due to the fact that the ICS controller implements its buffering structures with registers. Since both the baseline cache and the ICS architectures adopt 8KB 4-way set-associative cache memories, their required amount of BRAMs is the same. Overall, each ICS controller only occupies an area corresponding to 3% of the FPGA fabric, which, when combined with a modified MB-LITE [28] processor core, together result in a total occupation of 5.4% of the available hardware resource slices.

Regarding the proposed SMC, it is possible to observe a slight increase in hardware resources (when compared to the DSC), resulting from the inclusion of the burst controller and reorder buffer structures. These two structures only impose a utilization of 248 additional resource slices (corresponding to a 10% increase in LUTs) and two extra BRAMs for the reorder buffer.

The observed results for the base ring node architecture support the fact that the adopted data transfer interconnection is inherently scalable in what concerns its hardware footprint and operating frequency. Overall, each node only requires up to 0.2% (155 slices, corresponding to 297 LUTs) of the FPGA's resources. Accordingly, the adopted interconnection can be efficiently used to support a considerable number of processing cores, being the limiting factor the increased communication latency between nodes.

4.4.3 Stream Generation Efficiency and Main Memory Throughput

Table 4.7 presents the efficiency of the proposed DSC to resolve the data-patterns encoded with the proposed descriptor specification, when compared with the considered state-of-the-art approaches (with their corresponding description encoding).

By analyzing the values presented in Table 4.7, it is clear that the DSC architecture, by itself (not accounting for the SMC burst control), provides a steady performance of one address-per-cycle generation rate, whereas the remaining considered state-of-the-art pattern generation structures show a significant performance degradation for high complex patterns. Such is the case of

Table 4.7: Address generation rate and descriptor size (in bytes) for the proposed DSC and comparison with the considered related work approaches.

Pattern Type	Pattern Length (# words)	DSC			DFC [26]		AXI DMA [102]	
		Size (bytes)	Addr/cycle		Size (bytes)	Addr/cycle	Size (bytes)	Addr/cycle
			No Burst	Burst[2]				
Linear	1024	23	1	256	48	1	32	0.96
Tiled	128×72[1]	47	1	128	80	0.99	32	1
Diagonal	1024×1024	84	1	1	88	1	65k	1
Zig-Zag	8×8	125	1	1	132	0.71	480	0.63
Greek Cross	1024×1024	63	1	16	264	0.89	228k	1

[1] Within a memory block of 512×512 [2] Assuming a maximum burst length of 256 words (ref. AXI4 Protocol [121])

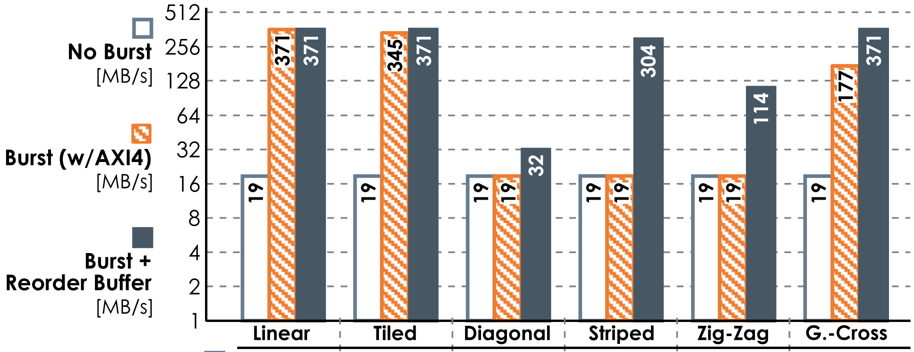
the *Zig-Zag* pattern where the HotStream DFC [26] and the AXI DMA [102] (APMC [54]) are only capable of offering 0.71 and 0.63 addresses-per-cycle generation rates, respectively.

Besides the achievable address generation efficiency, the benefits of the burst controller and reorder buffering optimizations to the main memory access were also evaluated. For such purpose, it was measured the throughput and average memory access latency (measured in cycles per byte) of the considered DDR3 memory with the adopted evaluation patterns for both optimization setups and compared to a direct connection between the DSC and the main memory, where only a single word is transferred per burst.

The obtained results for an SMC implementation (operating at 100 MHz) are shown in Fig. 4.17. From these results, it is clear that the most regular access patterns (with larger contiguous data regions) take the most advantage of the burst controller. In particular, for the *Linear* and *Tiled* patterns it is possible to reduce the average latency by as much as 20x, resulting in a throughput of up to 371 MB/s [1], when compared to a direct DSC connection. On the other hand, for the *Greek-Cross* pattern, the achieved throughput was limited at 177 MB/s (only 9x average latency reduction), due to its smaller 16-word burst lengths. The remaining patterns have no contiguous data regions and as such cannot take advantage of the burst controller by itself.

However, some substantial improvements are observed when adding the reorder buffer optimization. In particular, it is possible to increase the throughput of the *Greek-Cross* pattern to 371 MB/s by fetching the 48×48 blocks before applying the crossed pattern. Similarly, some significant benefits can be observed for the *Diagonal* and *Zig-Zag* patterns. For the *Zig-Zag* pattern, it is possible to fetch each 8×8 block before applying the *Zig-Zag* scan (reducing the average latency by 6x), hence increasing the effective throughput to 90 MB/s. Conversely, for the *Diagonal* pattern, only a slight throughput increase of 13MB/s is observed, due to the limited size (4KB) of the adopted reorder buffer, which provides limited data reuse opportunities (even when using a tiling procedure) when the diagonal size is too large. In fact, at the larger diagonal, the throughput becomes effectively the same as with a direct connection to the memory. A possible optimization for this pattern is the adoption of a stripped (or banded) processing approach, where *n*×1024

[1] This throughput value is close to the AXI4 bus theoretical maximum, which is 400MB/s for a single channel direction.



Avg. Memory Access Latency [cycles/Byte]	Linear	Tiled	Diagonal	Striped	Zig-Zag	G.-Cross
□	5.25	5.25	5.25	5.25	5.25	5.25
▨	0.27	0.29	5.25	5.25	5.25	0.56
■	0.27	0.27	3.10	0.33	0.88	0.27

Figure 4.17: Main memory throughput and average memory access latencies with the burst controller and reorder buffer optimizations for the considered data-patterns.

stripes are fetched and processed in smaller diagonals. With such an optimization, it becomes possible to take full advantage of the burst controller and reorder buffer, resulting in an overall throughput increase. For example, with a configuration of 16×1024 stripes, the average memory access latency is reduced by 16x, and a maximum throughput of 304 MB/s is achieved (see Fig. 4.17, column *Striped*).

4.4.4 Prototype Evaluation

The considered setups were evaluated in terms of: *i)* data communication and memory access time reduction; *ii)* performance speedup scalability for each setup against its own single-core configuration; *iii)* performance speedup and total energy savings against the reference `BASE` setup; and *iv)* performance-energy efficiency (in the form of an Energy-Delay Product (EDP) study).

4.4.4.A Performance Evaluation

The obtained results for the considered evaluation benchmarks are depicted in the charts of Fig. 4.18, by considering a variable number of cores. In particular, the bar plots on the top row represent the ratio of the communication time over the total execution time for each setup. Although the `AMPM` setup shows significant data communication overhead reductions, when compared to the `BASE` reference, they are mostly present in configurations with fewer cores (only up to 8 in most cases). Hence, despite their efficient capabilities, this type of prefetching mechanisms is not particularly suited for many-core systems, since they cannot cope with the increase in memory access traffic and bus contention of larger systems.

Conversely, as it can be observed for most of the considered benchmarks, the `uICS` setup provides mechanisms that target the mitigation of such drawbacks, allowing a significant data communication overhead reduction when compared to the `AMPM` setup. In particular, the offered

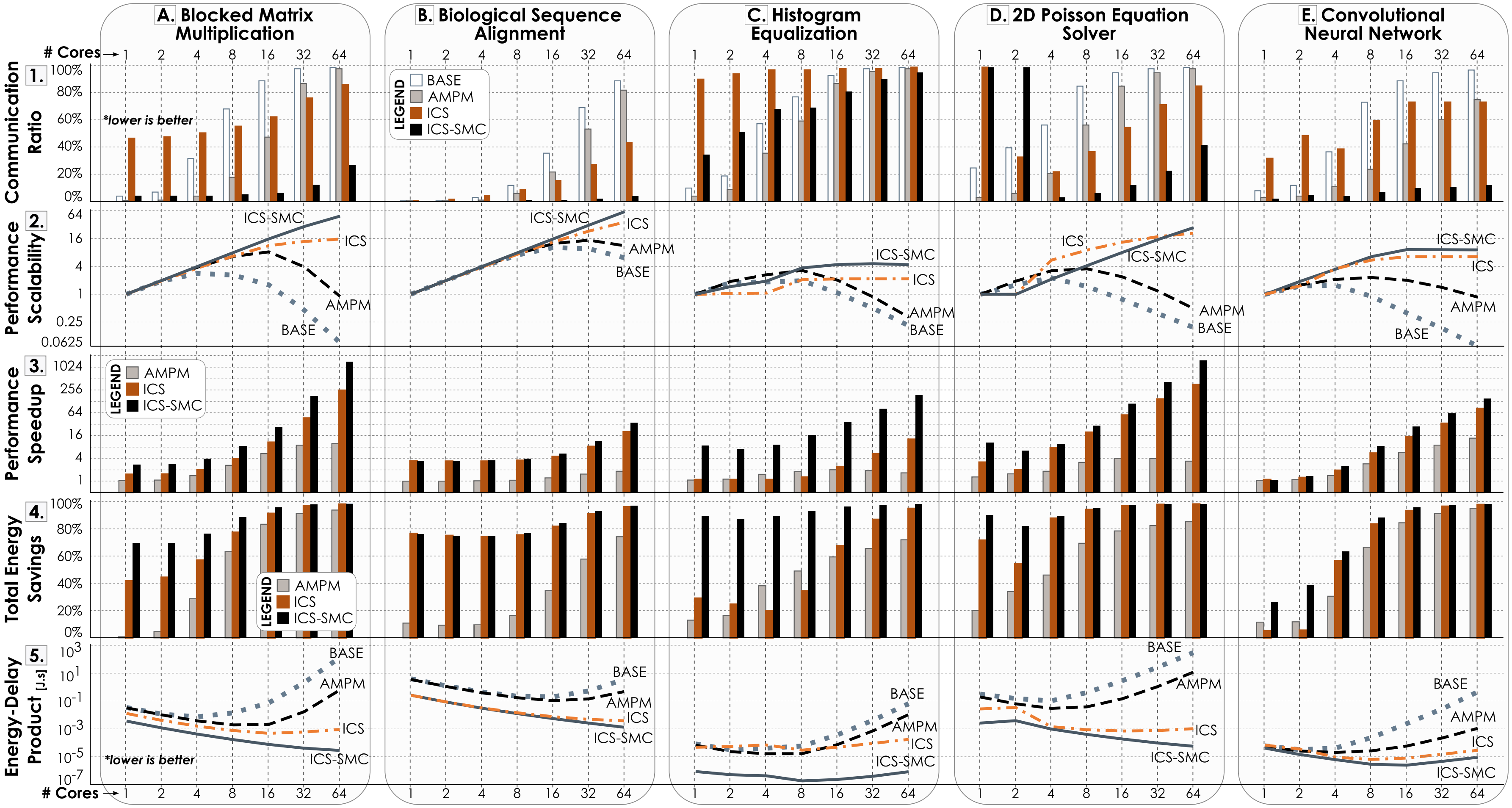


Figure 4.18: Comparison of the proposed ICS-SMC with the BASE, AMPM and uICS setups in: *i)* communication and execution time ratio improvement (top row); *ii)* performance speedup scalability (second row) for each setup against its own single-core configuration; *ii)* execution time speedup (third row) and the total energy savings (fourth row) against the BASE reference; and *iv)* energy-delay product (lower is better) (bottom row).

data streaming, broadcasting, and data reutilization capabilities allow a significant reduction of the total number of main memory accesses, therefore decreasing the observed contention in the interconnections. This is particularly evident with higher numbers of cores, where increased levels of data reutilization and inter-core communication are possible, due to the added amount of local memory resources available in the infrastructure. In fact, as it can be observed in the performance scalability charts in the second row of Fig. 4.18, the AMPM setup with a number of cores higher than 16 (8 in the Poisson equation solver benchmark) incurs in a high communication contention. This, in turn, results in a noticeable performance degradation (even resulting in performance slowdowns). Contrarily, the ICS-SMC setup is capable of fully taking advantage of its capabilities to mitigate contention and to increase the overall performance, even with a number of cores as high as 64. For the histogram equalization (Fig. 4.18.C) and Poisson equation solver (Fig. 4.18.D), it is possible to observe a sudden increase in performance when the entire dataset fits in the local memories. Specifically, it starts being directly communicated between cores, avoiding unnecessary communication with the main memory.

Despite the increased levels of communication efficiency of the uICS setup, an even higher data communication efficiency is observed for the ICS-SMC architecture, as it can be seen from the top two rows of Fig. 4.18. In fact, a significant communication-execution time ratio reduction is observed, when compared to the AMPM and uICS setups, resulting from the memory bandwidth optimization techniques of the SMC (burst controller and reorder buffer). This allows a greater mitigation of the communication overhead resulting from costly main memory accesses, which is particularly evident when observing the communication/execution time ratio improvement for the memory-bound matrix multiplication application (Fig. 4.18.A).

Such a significant memory access optimization and the observed performance scalability are directly reflected on the total execution time speedups (against the BASE reference) presented in the bar plots of the third row of Fig. 4.18. As it could be expected, the AMPM is only capable of partly mitigating the contention and memory traffic with fewer cores, achieving an average 4x speedup for the considered benchmarks. Contrarily, the streaming optimizations of the uICS setup already allow significant performance scalability improvements, not showing the slowdowns observed in the BASE and AMPM setups, with configurations with more than 8 cores. This, in turn, results in average 32x execution time speedups for the uICS setup, when compared to the BASE setup, corresponding to a 10x improvement over the AMPM.

However, when combining the ICS architecture with the SMC's memory-aware techniques, the ICS-SMC setup consistently provides performance speedup levels with no noticeable performance loss for most of the considered benchmarks. In fact, the only exceptions are the histogram equalization (Fig. 4.18.C) and the convolutional neural network (Fig. 4.18.E) benchmarks. In these cases, no added gain is observed with a 64-core configuration (when compared to a 32-core configuration), due to computation- and synchronization-related overheads. Such an improved communication efficiency allows average performance speedups of 126.7x against the BASE setup, corresponding to average 54x and 13x improvements over the AMPM and uICS setups, respectively. Moreover, maximum performance speedups of up to 1500x are observed for the ICS-SMC setup in the matrix multiplication (Fig. 4.18.A) and Poisson equation solver (Fig. 4.18.D) benchmarks.

4.4.4.B Energy Savings Evaluation

The observed performance increase and data transfer overhead reduction directly impact the total energy consumption of the whole system, as it is shown in the energy savings attained by the ICS-SMC when compared to the other setups (bar plots from the fourth row of Fig. 4.18). From the charts, it is clear that, besides the discussed performance improvements and the efficient data management of the offered streaming and broadcasting techniques, the considered main memory access optimizations also allow a significant reduction of the system's total energy consumption. Such savings (averaging 91% with 64-core setups) are directly related not only to the obtained execution time speedups but also to the significant decrease of the bus communication (resulting from to the offered broadcast capabilities), and of the number of accesses to the main memory (accounted in the ratios of the top row bar plots for data transfers reduction), in turn significantly reducing its total energy consumption.

4.4.4.C Performance-Energy Efficiency Evaluation

To gather all the observed results in a single metric, an additional performance-energy efficiency study was performed. In this case, it was used an EDP metric, calculated by multiplying the total energy consumption of a given setup by the corresponding execution time of each application. The line plots in the bottom row of Fig. 4.18 represent the measured EDP for each setup. By keeping in mind that lower values represent a higher efficiency, the measured results reflect the attained lower energy consumption and the improved scalability of the ICS-SMC setup, when compared to the uICS setup. As it could be expected, the poorer performance scalability and higher energy consumptions observed in the BASE and AMPM setups also result in a poorly scalable EDP with the increase of the number of cores, due to the increased contention in its shared interconnections.

On the contrary, the measured EDP values for the ICS-SMC highlight its data prefetching, reutilization, efficient communication capabilities, and memory access optimization. This is supported by their combined ability to mitigate the contention to the shared structures, and a consequent data transfer overhead reduction, which in turn not only results in higher communication throughputs but also in lower energy consumptions. In fact, thanks to the memory-aware SMC, it is possible to observe energy efficiency improvements as high as 245x when compared to the uICS setup.

4.5 Summary

This chapter proposed several data streaming approaches as viable alternatives to conventional cache-based hierarchies and predictive prefetching mechanisms. The presented approaches rely on a dedicated DSC that leverages the memory access pattern descriptor specification and the compilation tool proposed in Chapter 3 to offer an autonomous data acquisition and stream generation procedure.

The proposed DSC was initially deployed close to the processing cores of conventional GPU and GPP systems, as an alternative to the utilization of predictive prefetching schemes. The combination of the efficient data streaming provided by the DSC and the code reductions performed by the compilation tool showed that the proposed stream prefetching mechanisms are capable of outperforming state-of-the-art prefetchers and offer significant performance and efficiency gains in general-purpose systems.

While by itself the proposed stream prefetching already provided significant gains, a step further was taken by deploying the DSC in a dedicated data streaming infrastructure. Accordingly, it was proposed a new In-Cache Stream (ICS) communication model that deploys a full data streaming communication paradigm in a conventional cache-coherent hierarchy. Its main feature is the capability for seamlessly switching the communication paradigm between *memory-addressed* and *packed-stream* data accesses. This is done through a dedicated ICS controller that takes advantage of the usual *n*-way set-associative cache memory organization, by making each way individually usable as a stream buffer. The DSC itself is deployed by a memory-aware SMC, that also leverages a burst controller to optimize the main memory bandwidth, along with a reorder buffer to exploit data reorganization and reutilization techniques through in-time stream manipulation.

An extensive experimental evaluation showed that the ICS infrastructure provides significant gains over current state-of-art solutions in the considered benchmark applications. The obtained results highlight the capabilities of the ICS infrastructure to significantly reduce the data transfer overheads with the deployed data streaming, data reutilization, and communication management techniques, mitigating the contention in the shared interconnections. Specifically, significant performance speedups and total energy savings are observed, which result in overall processing energy efficiency improvements as high as 245x.

5

Adaptive Processing Structures

Contents

5.1 Reconfigurable Many-Core Accelerator 107
5.1.1 Many-Core Processing Infrastructure 108
5.1.2 Data Access and Interconnection Networks 109
5.1.3 Hypervisor and Accelerator Management 110
5.1.4 Reconfiguration Engine 111
5.1.5 Implementation Considerations and Constraints 113
5.1.6 Accelerator Configuration and Implementation 113
5.2 Reconfiguration Management 116
5.2.1 Runtime Learning and Reconfiguration Policies 117
5.2.2 Compile-time Modeling and Optimization 119
5.3 Experimental Validation 129
5.3.1 Methodology 129
5.3.2 Case Study A: Learning-based Automatic Reconfiguration 131
5.3.3 Case Study B: Compiler-Assisted Reconfiguration 135
5.3.4 Case Study C: Resource Management for Dynamic Workloads 138
5.4 Summary 143

The previous chapters explored the idea that knowing an application's memory access pattern before its execution allows the system to adapt and optimize its data transfer scheme (either through data prefetching or dedicated data streaming mechanisms), in turn increasing its effective throughput.

In this chapter, that same notion is further extended by exploring the application's computational characteristics (such as the type of computing operations or the amount of data parallelism) to dynamically specialize the system's processing architecture at runtime, with the goal of increasing its performance and energy efficiency. Accordingly, it is proposed the exploitation of architectural partial reconfiguration as an alternative to the power supply management and performance throttling mechanisms deployed in current heterogeneous systems.

As discussed in Chapters 1 and 2, most heterogeneous systems try to provide some degree of specialization by co-deploying multiple specialized processing components. However, they rely on the migration of application tasks between resources (to match their computational requirements), resulting in delays that can degrade performance. Additionally, while tasks are executed in specialized modules, the remainder of the circuit is usually powered-down (or made idle) to save energy, which results in a rather ineffective resource utilization.

Contrarily, partial reconfiguration allows a dynamic adaptation of the processing architecture itself to the characteristics of the task during its execution and according to runtime system constraints (e.g., performance requirements, operation complexity or power dissipation limits). This can be done by analyzing the application's computational requirements (either before or during runtime) and utilize such information to reconfigure a selected region of the system's processing infrastructure at runtime (e.g., by specializing functional units or by dimensioning the amount of processing parallelism), while the remaining logic continues to operate uninterrupted. Such an approach allows a convenient computing specialization and resource tuning, by instantaneous balancing the processing throughput and the energy consumption. Moreover, it eliminates the need for task migration, since the architecture of each processing unit is, itself, adapted on-the-fly. As such, tasks do not have to be moved to specialized modules (better-suited for their computational requirements) or to low-power units (to save energy), has it commonly occurs in heterogeneous architectures.

With the goal of validating this hypothesis, this chapter presents a study on the exploitation of compile-time and runtime application modeling to support partial reconfiguration in the context of general-purpose accelerators. The presented study comprises:

- The design of a **reconfigurable many-core accelerator** architecture, particularly devised to enable the runtime reconfiguration of its processing infrastructure. The designed architecture was implemented targeting a Field-Programmable Gate Array (FPGA) device;
- A **reconfiguration engine** that leverages the partial reconfiguration technology of the prototyping FPGA device to adapt the accelerator's processing infrastructure at runtime;

- A programmable **Hypervisor** mechanism that monitors the execution of a task to learn its computational requirements and trigger the reconfiguration process accordingly. This is done by deploying programmed reconfiguration policies that allow the system to balance its performance and energy consumption (in different execution contexts);
- A **compile-time optimization algorithm**, that leverages the application and system models to provide the mapping and scheduling of tasks to processing resources and plan their reconfiguration (through the Hypervisor).

Finally, the study is complemented with a set of experiments to demonstrate and validate the accelerator's adaptation capabilities, when supported by *i)* runtime task monitoring and analysis; *ii)* compile-time system modeling for the reconfiguration optimization and task scheduling; and *iii)* compile-time system modeling for resource balancing according to dynamic workloads.

5.1 Reconfigurable Many-Core Accelerator

The study presented in this chapter relies on the implementation of a dedicated reconfigurable many-core acceleration infrastructure. It was specifically designed to demonstrate the advantages of exploiting partial reconfiguration in a general-purpose computing context. Despite comprising a generic computing architecture, it was constructed by targeting its implementation on an FPGA device, as a way of making use of its partial reconfiguration technology. Moreover, being this study an exploratory exercise, the design of the accelerator was only focused on the processing architecture and its adaptation while leaving the deployment of the remaining supporting infrastructures (such as data communication and accelerator management) to be done with straightforward architectures.

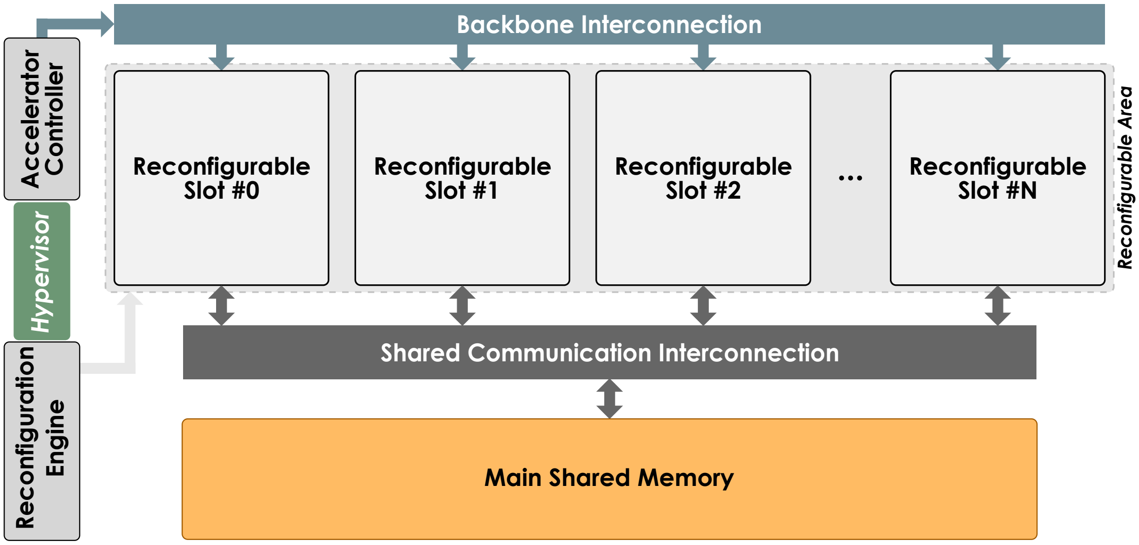


Figure 5.1: Reconfigurable many-core accelerator overview.

Accordingly, the devised many-core accelerator consists of a dense processing infrastructure, organized in several reconfigurable computing clusters, each composed of multiple processing cores. The adaptation of the processing infrastructure is assured by reconfiguring each cluster to modify the number of instantiated processing cores and their architecture. To do so, part of the FPGA fabric was divided in a set of *reconfigurable slots* where clusters can be instantiated. As such, while a static (non-reconfigurable) FPGA implementation is usually encoded in a single bitstream to program the full device, to make use of partial reconfiguration, a set of partial bitstreams is generated for each possible configuration for the defined reconfigurable slots.

The remaining fabric is reserved for the accelerator's supporting infrastructures which comprise: *i)* a *reconfiguration engine* that performs the actual partial reconfiguration of the accelerator's reconfigurable slots; *ii)* the *Hypervisor* module that is responsible for assigning tasks to processing cores, monitoring their execution and scheduling the reconfiguration process; and *iii)* an *accelerator controller*, responsible for interfacing the processing infrastructure and the Hypervisor. The processing infrastructure itself is connected to the *accelerator controller* through a dedicated *backbone communication structure*, and to a main shared memory via a *shared communication bus* (see Fig. 5.1).

The accelerator's reconfigurable infrastructure was implemented by defining a set of general-purpose processing core architectures with varying computing capabilities and complexity. Accordingly, each reconfigurable slot was carefully dimensioned by profiling the considered architectures in terms of throughput, power dissipation, and area resources. To do so, and to provide a fully functional reconfigurable platform, the designed accelerator and its components were implemented on a Xilinx Virtex-7 FPGA device.

5.1.1 Many-Core Processing Infrastructure

To maximize the flexibility and versatility of the reconfigurable processing infrastructure, its architecture was designed by following a fully modular approach. Accordingly, each computing cluster was designed to provide a generic topology and organization (see Fig. 5.2), comprising multiple processing cores interconnected via: *i)* a local core interconnection (e.g., a shared bus) that interfaces each core to the accelerator's shared communication interconnection (and to the main memory - see Fig. 5.1), through a dedicated bridge; and *ii)* a local backbone interconnection, connected to the accelerator's own backbone interconnection (see Fig. 5.1), to allow a direct communication of each core with the accelerator's controller (and the Hypervisor). Finally, the cluster accommodates an optional local memory device (shared between the cores).

Each processing core was designed to be as independent as possible of its underlying general-purpose Processing Element (PE) architecture (see Fig 5.2). Accordingly, each core's PE is paired with a dedicated controller, a local scratchpad memory (for the program and local data private to the core), and an interface to the local cluster interconnection.

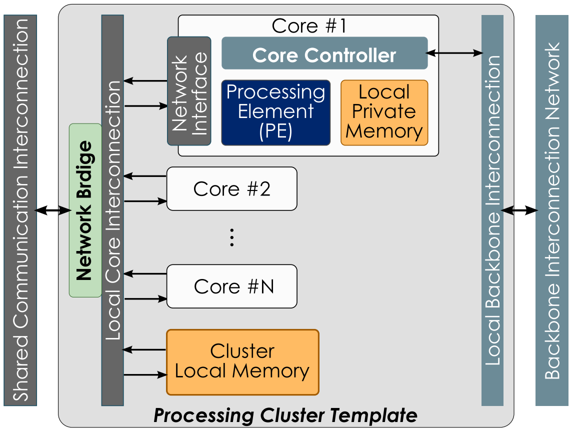

Shared Communication Interconnection
Network Bralge
Local Core Interconnection
Network Interface
Core #1
Core Controller
Processing Element (PE)
Local Private Memory
Core #2
Core #N
Cluster Local Memory
Processing Cluster Template
Local Backbone Interconnection
Backbone Interconnection Network

Figure 5.2: Internal block diagram of a processing core inside a cluster.

The adopted core controller ensures the PE's coordinated execution inside the processing infrastructure. Besides synchronizing the PE with the Hypervisor (through the accelerator controller, as detailed below), it also monitors its execution, while keeping a record of a set of performance counters. To accomplish such tasks, the core controller communicates with the Hypervisor to: *i)* load the local memory with program and data, and to receive a command to start the PE's execution; and *ii)* to notify the Hypervisor about the completion of PE's execution and to send back the information measured by the performance counters.

By default, the controller monitors the number of clock cycles that are required to execute a specific task. However, depending on the considered implementation, it is also possible to monitor other parameters, which are specifically customized based on the underlying architecture of the PE. Examples of additional counters include the number of specific operations (e.g., integer divisions, floating-point additions or multiplications), branch mispredictions or the identification of specific function calls (e.g., to detect operations that are performed through software libraries, instead of hardwired instructions). Hence, by providing this monitoring flexibility, it is possible to offer the Hypervisor with the necessary means to learn the characteristics of a running application/task and to adapt the processing architecture accordingly.

5.1.2 Data Access and Interconnection Networks

As it was referred before, the communication between all the accelerator's components in assured by the *backbone* and *shared communication* interconnections (see Figure 5.1).

The *backbone interconnection* was implemented by a simple communication structure based on the AXI-Stream Protocol [118]. Despite the already available Xilinx IP cores that implement AXI-Stream interconnections [121], it was implemented a lighter, but still fully-compliant and custom interconnection, since not all the protocol signals are required, and it is only necessary to create direct communication channels between the Hypervisor and each core. Moreover, this implementation was fulfilled with hardware resource overhead reduction in mind, since the custom module targets a much lower complexity than the original IP core.

The implemented interconnection provides a communication mechanism (with single-cycle latency) between up to 16 peripherals which, by the protocol, corresponds to 16 AXI-Stream master and 16 AXI-Stream slave ports. Consequently, this interconnection features two independent unidirectional channels: a one-to-many channel (with broadcast capabilities) and a many-to-one channel.

To allow a flexible access of each core to the main memory (see Fig. 5.1), the *shared interconnection* was designed by deriving its architecture from the implemented *backbone interconnection*. Hence, while maintaining the same base structure, it is possible to deploy an arbitrated shared-bus interconnection. This is achieved by including extra signals to transmit a memory address signal, and by adding dedicated memory and core interfaces, that see the unidirectional channels as a single bidirectional channel.

5.1.3 Hypervisor and Accelerator Management

The adaptation of the processing infrastructure is managed by a programmable Hypervisor mechanism (see Fig. 5.1). It comprises a software module executed in an MB-LITE [120] microprocessor, that is responsible for managing the reconfigurable hardware resources that are offered by the accelerator. Such a programmable management module aims at deploying execution optimization policies that allow the Hypervisor to determine the most adequate processing architecture to each application task under execution.

For such purpose, it receives and maintains a permanent record of the performance counters of each core. In brief (detailed later on), with this information, it calculates the actual operational and computational intensity of each running task and estimates the total execution time, power and energy consumption on the instantiated core architectures. After obtaining these values, it re-estimates the same metrics by considering the reconfiguration of a computing cluster to a better-suited architecture.

The Hypervisor's microprocessor communicates with the remaining management modules through a memory-mapped interface. Such an interface creates a straightforward communication channel with the accelerator controller (to assign tasks to the accelerator's processing cores and obtain performance values) and with the reconfiguration engine (to issue reconfiguration commands).

Accordingly, the interface between the Hypervisor and the processing infrastructure is assured by a dedicated accelerator controller. The controller's architecture (depicted in Fig. 5.3) comprises a set of *control* and *status registers*, a Finite-State Machine (FSM) (that handles requests from the Hypervisor and issues commands to the accelerator) and an interface to the backbone intercommunication.

Besides establishing the interface between the accelerator and the Hypervisor, the controller

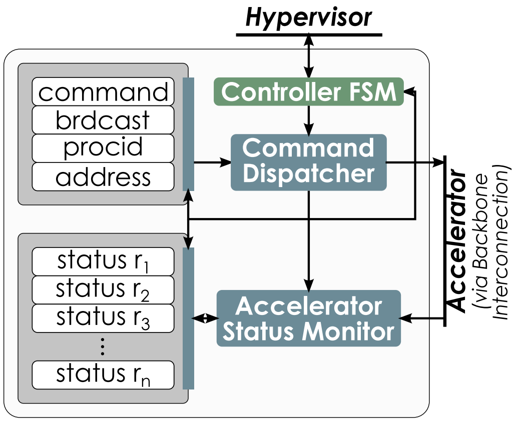


Figure 5.3: Accelerator Controller. A FSM is used to handle all requests from the host CPU, activate the command dispatcher that issues commands to the reconfigurable accelerator according to the control registers and report the accelerator status to the host.

is also responsible for managing and monitoring the processing cores execution and for loading the cores' local memories. The Hypervisor sends commands to the controller by configuring a set of *control registers*. A set of *status registers* (with a configurable number and size, according to the processing structure organization), is maintained by the controller, indicating the execution state of each processing core ('1' or '0', indicating running and stopped states). This way, the Hypervisor can monitor the cores' execution by reading the values in each register.

5.1.4 Reconfiguration Engine

To provide the required partial reconfiguration capabilities, a reconfiguration engine was designed by targeting the adopted Xilinx FPGA technology. It relies on a minimalistic controller (implemented by a FSM) that handles the following tasks: *i*) receiving reconfiguration commands issued by the Hypervisor; *ii)* initiating the reconfiguration port and reading the required partial bitstreams from a storage memory; *iii)* ensuring the correct transfer of each partial bitstream to the FPGA internal reconfiguration port; and *iv)* signaling the completion of the reconfiguration process.

For the Xilinx 7-Series technology, the developed reconfiguration engine (illustrated in Fig. 5.4) is composed of a Xilinx Internal Configuration Access Port (ICAP) [1] controller (AXI HWICAP), an on-board linear flash memory used as a bitstream memory, the reconfiguration controller and a dedicated Direct Memory Access (DMA) controller.

In this particular implementation, all the available bitstreams are preloaded to an on-board linear flash memory. This includes both the initial full system bitstream and the partial bitstream configurations. Although the accelerator's main memory could be used to accommodate the partial

[1] The ICAP is the device configuration port, with a 32-bit data port, supporting a writing bandwidth of up to 100MHz. It allows for the configuration process to be controlled by an entity instantiated in the device itself. This, in turn, allows the deployment of a custom low-profile reconfiguration engine, to provide full control over the reconfiguration process.

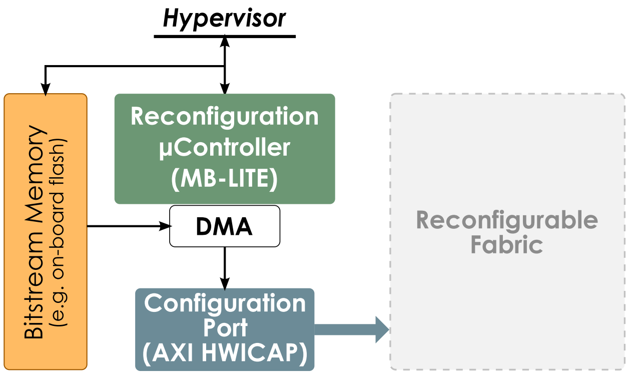


Figure 5.4: Architecture of the reconfiguration engine for a Xilinx 7-Series device.

bitstreams, it would potentially impact the computation throughput, since both the reconfiguration engine and the processing cores would be simultaneously accessing the same memory. Since the initial bitstream configuration is already automatically loaded from the FPGA's flash memory when the board is booted, it is also possible to use it to store the repository of partial bitstreams. This way, there is no communication interference between computation and reconfiguration.

To trigger the reconfiguration of each slot, the reconfiguration engine controller accepts a command from the Hypervisor containing an identification of the partial bitstream to be loaded to the reconfigurable fabric. Upon receiving the identification of the required configuration, the controller issues a read command to the flash controller to obtain the bitstream header. This header contains the information regarding the configuration bitstream size (corresponding to the number of bytes that need to be sent to the ICAP controller).

After this initial phase, a low-profile DMA controller (composed of a simple adder and control registers) transmits the partial bitstream from the flash memory to the ICAP port. To set up the DMA, it is only required to configure it with the bitstream size and its memory location. This is a necessary design option since the data transfer is performed by reading 16-bit words from the flash memory and by packing them into 32-bit words, before sending them to the ICAP controller. With such a DMA-based approach, it is possible to exploit the flash memory's burst mode, allowing a faster merge of the 16-bit words in 32-bit words. This results in much higher transfer rates and consequently lower reconfiguration latency.

According to device constraints, only one reconfiguration command can be issued at a time, since the previous reconfiguration must be concluded before a new reconfiguration command can be issued. Besides simplifying the reconfiguration procedure (avoiding the presence of reconfiguration command queues), it is worth noting that it does not significantly affect the resulting performance since a potential access contention to the flash memory would prevent greater reconfiguration throughputs.

5.1.5 Implementation Considerations and Constraints

To implement the reconfigurable processing infrastructure in the considered FPGA device, the distribution and amount of resources in each reconfigurable slot was carefully tuned to allow the instantiation of a significant range of processing core architectures. Moreover, attention was given to inherent overheads that result from the reconfiguration procedure (tied to the size of each reconfigurable area) and from possible execution synchronization issues that may arise from changing the processing hardware.

Furthermore, to attain the aimed modular approach of the reconfiguration procedure, it is necessary to constrain each *reconfigurable slot* to a specific and well-defined region of the reconfigurable fabric of the FPGA. As such, to ensure that the reconfiguration of each slot only changes a predefined region in the device, appropriate region delimitation had to be applied. This ensures that the reconfiguration engine can trigger the partial configuration of the fabric with no risk of the process affecting the on-going activity of the remaining accelerator's resources. Another limiting factor in the mapping of the slots to the configurable logic is the location of certain dedicated hardware modules/ports of the device (such as PCIe bridges, configuration ports or GPIO). Since these modules are implemented by hard-cores, they can only be allocated to specific regions of the device.

On the other hand, since a significant amount of cores can be deployed, a core-level reconfiguration granularity would incur on an undesired fragmentation of the reconfigurable logic. Specifically, not only the amount of individually reconfigurable zones would give rise to a significant reconfiguration overhead (the process would have to be repeated for each area), but their execution would also need to be individually stopped and restarted, in turn increasing the number of control messages throughout the system. To avoid such a situation, a more coarse-grained approach is considered, where the reconfigurable fabric is distributed in larger areas with equal size and amount of resources. This way, the reconfiguration can be performed to the level of a full slot, limiting the number of reconfiguration procedures and easing the system's control.

5.1.6 Accelerator Configuration and Implementation

This section presents the configuration and implementation of the devised accelerator using a Xilinx Virtex-7 FPGA (XC7VX485T) board. To do so, synthesis and place&route procedures were performed with Xilinx ISE 14.7, to implement the accelerator and assess the characteristics of each of its components, regarding their timing, resource area occupancy and power dissipation (estimated with the Xilinx Power Estimation tools).

The reconfigurable processing infrastructure was dimensioned by considering three homogeneous multi-core cluster configurations, based on three types of PE architectures with different computational and architectural complexities. Since each reconfigurable slot is equally dimensioned in terms of the provided FPGA resources, the topology of each type of cluster configuration will depend on a trade-off balance between core architecture complexity, the number of cores per cluster and the total number of clusters, as discussed in the following subsections.

5.1.6.A Processing Core Architectures

Similarly, to the Hypervisor, the three considered PE architectures were based on the 32-bit RISC MB-LITE [120]. This soft-core was selected due to its portable processing structure, with an implementation fully compliant with the well-known MicroBlaze Instruction Set Architecture (ISA) [126], which allows taking advantage of the GNU Compiler Collection (GCC) [29]. Moreover, the MB-LITE design requires few hardware resources and is configurable, being relatively easy to include custom modules.

Accordingly, the simplest PE architecture (*Type A*) that was deployed corresponds to the basic configuration of the MB-LITE core, i.e., with both the barrel shifter and multiplier deactivated. Without the presence of these structures, the architecture requires a much lower hardware footprint and presents a higher operating frequency [120]. The second architecture (*Type B*) includes the barrel shifter and the multiplier structures, corresponding to the full MB-LITE architecture. For the third considered architecture (*Type C*), a full MB-LITE architecture supporting a single-precision Floating-Point Unit (FPU) was considered. In this particular case, the considered 4-stage pipelined FPU is composed of three Xilinx IP Floating-Point Operator [127] cores. Since the MB-LITE core is compliant with the MicroBlaze ISA, the corresponding Floating-Point (FP) instruction opcodes [126] for addition, subtraction, multiplication, and compare were adopted. Moreover, although FPUs are only deployed by *Type C* PEs, the other two architecture types can still perform floating-point operations by resorting to the GCC software implementation. Naturally, the same situation occurs in *Type A* PEs regarding integer multiplication and barrel shifter operations.

To provide the Hypervisor with relevant profiling information about each core runtime performance, five monitoring events were measured during the execution, corresponding to the counts of clock cycles (`CLK`), integer multiplications (`MUL`), FP operations (`FP`) and calls to software-emulated integer multiplications (`SW_MUL`) and FP operations (`SW_FP`).

5.1.6.B Hardware Resource Analysis

After defining the adopted PE architectures, they were characterized by a hardware resource analysis to dimension the number of processing cores that can be instantiated in each cluster configuration. Table 5.1 presents the required hardware resources for each type of processing core. As expected, the resource overhead increases with the complexity of each architecture. Hence, the number of processing cores in each cluster configuration was dimensioned to ensure that the total occupied area of each cluster is approximately the same. This way, each *Type A* cluster contains 15 cores, each *Type B* cluster contains 12 cores, and each *Type C* cluster contains 8 cores (see Table 5.2).

Table 5.1: Characterization of each type of core, in terms of hardware resources and maximum operating frequency.

	Available	**Processing Cores**		
	Resources	*Type A*	*Type B*	*Type C*
Registers	607200	530 (<1%)	536 (<1%)	841 (<1%)
LUTs	303600	1132 (<1%)	1406 (<1%)	1932 (<1%)
Slices	75900	402 (<1%)	528 (<1%)	658 (<1%)
RAMB36E1	1030	4 (<1%)	4 (<1%)	4 (<1%)
RAMB18E1	2060	3 (<1%)	3 (<1%)	3 (<1%)
DSP48E1	2800	0 (0%)	3 (<1%)	7 (<1%)
Max Freq. [MHz]	-	211.1	114.7	113.4

Table 5.2: Characterization of each cluster type, in terms of hardware resources, maximum operating frequency and power consumption.

	Available	**Computing Clusters**		
	Resources	*15 × Type A*	*12 × Type B*	*8 × Type C*
Registers	607200	7655 (<1%)	6146 (<1%)	6569 (<1%)
LUTs	303600	16706 (5%)	16614 (5%)	15334 (5%)
Slices	75900	5959 (7%)	6244 (8%)	6530 (8%)
RAMB36E1	1030	60 (5%)	48 (4%)	32 (3%)
RAMB18E1	2060	45 (2%)	36 (<1%)	24 (<1%)
DSP48E1	2800	0 (0%)	36 (<1%)	56 (2%)
Max Freq. [MHz]	-	206.2	114.7	113.2
Static Power [mW]	-	210.9	210.7	209.9
Total Power@100 MHz [mW]	-	615.2	636.6	600.7

Table 5.3: Characterization of the supporting infrastructure of the accelerator, in terms of hardware resources, maximum operating frequency and power consumption.

	Registers	LUTs	RAMB36E1	RAMB18E1	DSP48E1
Hypervisor	536 (<1%)	1406 (<1%)	4 (<1%)	3 (<1%)	3 (<1%)
Reconfiguration Controller	530 (<1%)	1132 (<1%)	4 (<1%)	3 (<1%)	0 (0%)
ICAP Controller	742 (<1%)	546 (<1%)	1 (<1%)	1 (<1%)	0 (0%)
DMA	2891 (<1%)	2930 (<1%)	1 (<1%)	0 (0%)	0 (0%)
Total Power Dissipation [mW]	367.9				

By analyzing the power consumption of each cluster type, when considering an operating frequency of 100 MHz, it was observed a rather similar value for the three types (bottom of Table 5.2). This is explained by the fact that the three cluster types were properly defined to occupy the same amount of hardware resources.

The remaining components of the platform were also analyzed regarding the required hardware resources and power consumption. Hence, all the components that comprise the accelerator management and reconfiguration engine (see Fig. 5.1) were taken into account for the total power dissipation of the system. Table 5.3 presents the hardware resources required for the accelerator's supporting infrastructure. It can be observed that a low occupancy of about 5% was observed for the FPGA device, with a total power dissipation of 367.9 mW.

According to the obtained results, it is possible to implement a 7-cluster accelerator in the considered FPGA. In the whole, this represents a number of processing cores ranging from 56 to

105 in the implemented system.

5.1.6.C Reconfiguration Overhead

In what concerns the real-time adaptation of the system, it was observed an expected dependency between the reconfiguration time and the size of the partial bitstream that is loaded into the ICAP. Since the bitstreams for the three considered cluster configurations are approximately 2MBytes, a reconfiguration time of approximately 10ms is observed.

It was also considered an additional bitstream configuration corresponding to an empty cluster, which can be used to turn off a reconfigurable slot (e.g., to put a cluster on an idle state to save energy). This configuration bitstream file is only 460KBytes long, in turn requiring a reconfiguration time of approximately 2ms.

To calculate the total power that is used by the reconfiguration procedure, the estimated power that is consumed by the reconfiguration engine, when it is in its idle state, was subtracted from the power consumed by the same engine while performing a reconfiguration procedure. The obtained difference between these two estimations results in a reconfiguration power of about 44mW. Despite consuming significantly less power than what it is required by the actual processing clusters, this result can also be considered by the Hypervisor, when making decisions regarding the triggering of the reconfiguration procedure.

5.2 Reconfiguration Management

As described above, the designed reconfigurable accelerator is managed by a Hypervisor mechanism that is responsible for optimizing the accelerator's computing efficiency through instantaneous resource balancing. To do so, it can be programmed according to different task execution and cluster reconfiguration policies (e.g., to maximize raw acceleration or to minimize power dissipation). The main goal is to allow the Hypervisor to decide on the best possible action according to the application's characteristics and the system's execution context.

As an example, to accelerate the execution of a processing task, the Hypervisor may decide either to run it on an already instantiated cluster, to reconfigure the cluster to a better-suited architecture for the task, or to wait until a previous task completes its execution on an already suited cluster. On the other hand, to control energy consumption, the Hypervisor may decide to entirely disable a processing cluster, by reconfiguring it with an empty configuration (powering off the corresponding region on the FPGA device). Such cases may occur when there are not enough processing tasks to make use of the entire accelerator's resources (e.g., when the application has insufficient parallelism), allowing it to minimize the overall power dissipation; or when the power budget is exceeded by the active processing clusters, thus requiring parts of the reconfigurable area to be disabled.

To deploy such optimization procedures, the Hypervisor must be aware of the characteristics of the application and the available processing architecture configurations. While the architecture model of each processing cluster is predefined and can be hard-coded, the characteristics of the application can only be obtained through its analysis. In this study, two distinct approaches are considered to implement this analysis:

- **At runtime** (Section 5.2.1), by monitoring the application's execution in the accelerator and learning its computational requirements, through to the performance counters of each processing core. The gathered information is used by the Hypervisor to trigger the reconfiguration procedure according to different execution policies (e.g., maximum performance or energy saving contexts);
- **At compile-time** (Section 5.2.2), by pre-modeling the application and generating its mapping to the accelerator's cores with a specially devised optimization algorithm. The end result is a set of execution plans (optimized for different execution contexts) that are followed by the Hypervisor to schedule the execution of processing tasks and trigger cluster reconfiguration procedures.

Both approaches were built on the assumption that parallel applications can be discretized into multiple computing tasks (that perform the computation over a *chunk* of the dataset to be processed). The performance of each task in a particular core architecture is characterized according to a operations-per-second (OPS) metric, calculated according to the values measured by the core performance counters.

5.2.1 Runtime Learning and Reconfiguration Policies

The first considered approach characterizes each task by learning its computational requirements at runtime and then selecting the best-suited architecture to execute it. To do so, a set of reconfiguration policies is programmed in the Hypervisor to manage the accelerator's execution and reconfiguration, namely: *i)* overall minimization of the execution time; *ii)* maximization of the processing performance while establishing a ceiling for the total power dissipation; and *iii)* minimization of the energy consumption, while guaranteeing a minimum performance level.

The procedure presented in Algorithm 1 implements a runtime performance prediction routine. To decide when it is advantageous to reconfigure a given cluster, before executing a task, the algorithm performs two distinct steps. First, it searches for a configuration allowing for higher gains in performance, energy, or power consumption. This decision also takes into account the reconfiguration overhead. Then, if such configuration is found and if the required time to complete the required reconfigurations is lower than executing the task with the current configuration, the targeted cluster is reconfigured accordingly. It is worth noting that this algorithm can also be used for energy and power optimizations, by simply changing all time-based variables to energy variables.

Algorithm 1 Execution Time Optimization

```
Input: task, chunk, current_arch
Global Variables: reconf_time, inprogress_reconfs
Output: new_arch or 0

ops ← load_task_stats(task, current_arch)
curr_time ← ops × chunk

new_arch ← 0
time ← curr_time
for each arch do
    ops ← load_task_stats(task, arch)
    test ← ops × chunk

    if $test + reconf\_time < time$ then
        time ← test
        new_arch ← arch
    end if
end for

if ($new\_arch > 0$) and ($curr\_time < time + inprogress\_reconfs \times reconf\_time$) then
    new_arch ← 0
end if
return new_arch
```

Algorithm 2 Power-Ceiling Algorithm

```
Input: task, chunk, idle_list, max_power, rec_list
Global Variables: reconf_power, static_power

curr_power ← static_power
for each cluster do
    if $cluster.arch \neq EMPTY$ then
        curr_power ← curr_power + get_power(cluster, cluster.arch)
    end if
end for

for each $cluster \in idle\_list$ do
    if $curr\_power > max\_power$ then
        curr_power ← curr_power - get_power(cluster, cluster.arch)
        rec_list ← (cluster, EMPTY)
    end if
end for

cluster ← find_idle_or_off_cluster()
new_arch ← time_optimization(task, chunk, cluster.arch)
if ($new\_arch > 0$) and ($curr\_power + reconf\_power \leq max\_power$) then
    rec_list ← (cluster, new_arch)
end if
```

Accordingly, the procedure presented in Algorithm 2 adds an extra level of optimization to the previous algorithm, by introducing a dynamic power-ceiling constraint. This power constraint can change at runtime, depending on the dynamic requisites of the accelerator. Based on the total power budget of the system at a given time, the algorithm tries to turn off clusters that are inactive until the power constraint is met. Each cluster is turned off by reconfiguring it to an empty cluster, which turns off the FPGA logic in the corresponding reconfigurable slot. If the power budget is increased, an idle (or turned off) cluster is searched and analyzed with the algorithm presented in Algorithm 1 for a given task. Under this assumption, each cluster is only reconfigured if the power overhead for the reconfiguration procedure and the newly configured architecture does not violate the power ceiling constraint.

Algorithm 3 Minimum Assured Performance Algorithm

```
Input: task,chunk,idle_list,min_ops,rec_list
Global Variables: reconf_time

for each cluster do
    if $cluster.arch \neq EMPTY$ then
        $curr\_ops \leftarrow curr\_ops + load\_task\_stats(cluster.task, cluster.arch)$
    end if
end for

$cluster \leftarrow find\_idle\_or\_off\_cluster()$
if $cluster == NULL$ then
    $new\_arch \leftarrow time\_optimization(task, chunk, EMPTY)$
else
    $new\_arch \leftarrow time\_optimization(task, chunk, cluster.arch)$
end if

if $cluster \neq NULL$ and $new\_arch > 0$ then
    $test \leftarrow curr\_ops - load\_task\_stats(task, cluster.arch)$
    $ops \leftarrow load\_task\_stats(task, new\_arch)$
    $test \leftarrow test + (ops \times chunk + reconf\_time)/chunk$
    if $test \geq min\_ops$ then
        $rec\_list \leftarrow (cluster, new\_arch)$
        $curr\_ops \leftarrow curr\_ops - test$
    end if
end if

for each $cluster \in idle\_list$ do
    $ops \leftarrow load\_task\_stats(task, cluster.arch)$
    if $curr\_ops - ops < min\_ops$ then
        continue
    end if
    $curr\_ops \leftarrow curr\_ops - ops$
    $rec\_list \leftarrow (cluster, EMPTY)$
end for
```

Finally, the procedure presented in Algorithm 3 adds a new level of optimization, with a minimum assured performance policy. This algorithm tries to minimize the power consumption while maintaining a given minimum performance level. Initially, the algorithm presented in Algorithm 1 is executed to check for the performance requirements of the new task chunk. If reconfiguration is required for an idle cluster, the reconfiguration overhead and the future performance of the new cluster architecture is checked, and it only reconfigures such cluster provided that the minimum assured performance is met. Finally, the algorithm tries to turn off idle clusters to lower the total power consumption, as long as the minimum performance is met for the current task.

5.2.2 Compile-time Modeling and Optimization

The considered compile-time optimization approach assumes that the Hypervisor makes use of a set of optimized *execution plans*. These plans are generated at compile-time by a specially devised Design Space Exploration (DSE) methodology, which makes use of a Multi-Objective Optimization (MOO) algorithm to optimize the accelerator's execution in terms of performance, power dissipation, and energy consumption.

The generated execution plans comprise a time-ordered execution queue (see Fig. 5.5) containing: *i)* the mappings of the several application tasks to the different core architectures of the

TASK	ARCH	SLOT	REC	TIME
		⋮		
7	4	1	no	5
2	3	2	no	5
5	2	3	no	5
4	1	2	yes	13
3	1	1	yes	27
6	2	3	no	30
		⋮		

Figure 5.5: Execution plan example. If two tasks (`TASK` column) are mapped to different components (`ARCH` column) and subsequently assigned to the same reconfigurable slot (`SLOT` column), a reconfiguration must be triggered (`REC` column) before the second task begins.

accelerator; and *ii)* the assignments of those mappings and core architectures to the available set of reconfigurable slots, together with any required reconfiguration commands. This way, an execution plan can be represented as a list of tuples `(TASK,ARCH,SLOT,REC,TIME)`, where `TASK` is a task identification, `ARCH` is the mapped architecture identification, `SLOT` is the reconfigurable slot for that architecture, `REC` indicates whether reconfiguration is required prior to the task execution and `TIME` is the assigned starting time.

To take advantage of these execution plans, the Hypervisor can operate in one of two modes:

- On the *Scheduler* mode, the Hypervisor strictly follows the order in the execution queue, corresponding to the current execution model, and issues the tasks to their mapping component in its corresponding reconfigurable slot, as well as the required reconfiguration commands. In case there is a change in the execution environment, such as a low-power scenario, the Hypervisor switches the considered execution plan and continues the execution from where it left;
- On the *Resource Management* mode, instead of strictly following an execution plan, the Hypervisor only extracts the mappings between tasks and cluster architectures. Then, it monitors the core throughput variations and switches between execution plans in real-time in order to balance the available resources, ensuring a system adaptation to applications with dynamic workload variations. To ensure the compliance with the application performance, power, and energy constraints, it dynamically selects the most appropriate execution plan from a list of feasible candidates.

5.2.2.A Design Space Exploration Specification Model

The conceived DSE algorithm requires the use of a formal representation of both the underlying application and of the target processing structure. In particular, it was considered the general system specification model proposed in [83] for the description of the data flow between the several application tasks and the hardware components to define a new representation that also supports reconfigurable systems.

According, the herein defined model relies on a data-flow graph to represent the application's

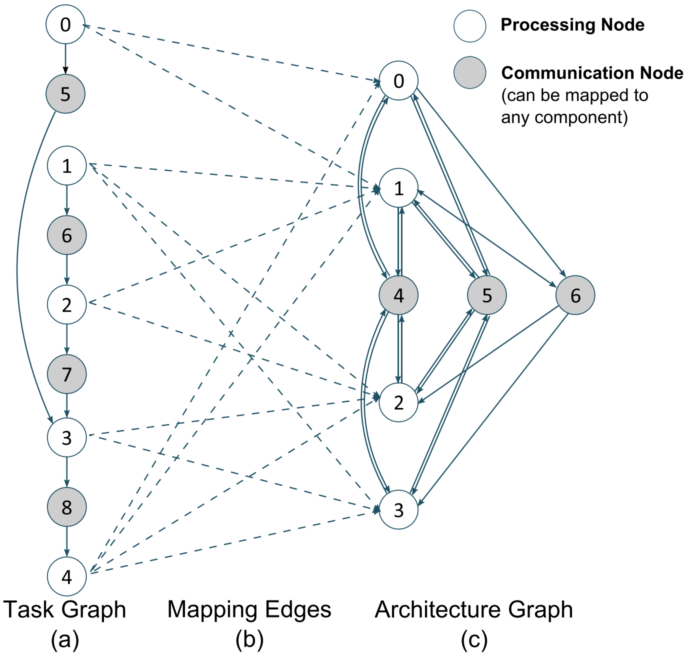


Figure 5.6: Example of a system specification comprehending the Task (a) and the Architecture (c) Graphs, together with the set of Mapping Edges (b) between them. The values inside each node of the graphs correspond to unique identifications for each task (a) and component (c) of the system.

tasks and their interactions [128, 129]. Although it is manually defined for this particular study, future implementations can consider pre-processing modules that automatically extract the applications characteristics and elaborate such a graph.

In a first step, the algorithm must be provided with the set of different types of supported operations (including communication). Such operations can range from fine-grained, such as integer or floating-point instructions, to coarse-grained operations, such as entire functions or tasks. This way, each task, t_i, is represented by:

- n_k - number of operations (or tasks) of type-k;
- d_s - size of its assigned data chunk.

Based on this assumption, the application is represented by a data-flow Task Graph, $G(V_T, E_T)$, composed of a set of tasks V_T. This set comprises both *processing tasks* ($V_T^p \subset V_T$) and *communication tasks* ($V_T^c \subset V_T$). The Task Graph also comprises the set of edges, E_T and the data flow between them. Each edge is given by the tuple $e = (t_i, t_j)$, with $t_i, t_j \in V_T$. An example Task Graph is shown in Fig. 5.6(a).

A similar formalization is applied to the definition of the architecture model. This way, an architecture is represented by a data-flow Architecture Graph, $G(V_C, E_C)$, composed of a set of components V_C (comprising both *processing components*, $V_C^p \subset V_C$, and *communication channels*, $V_C^c \subset V_C$), and the set of edges, E_C, that represent the data flow between them. Each edge is given by the tuple $e = (c_i, c_j)$, with $c_i, c_j \in V_C$. Every component, c_i, represents a reconfigurable

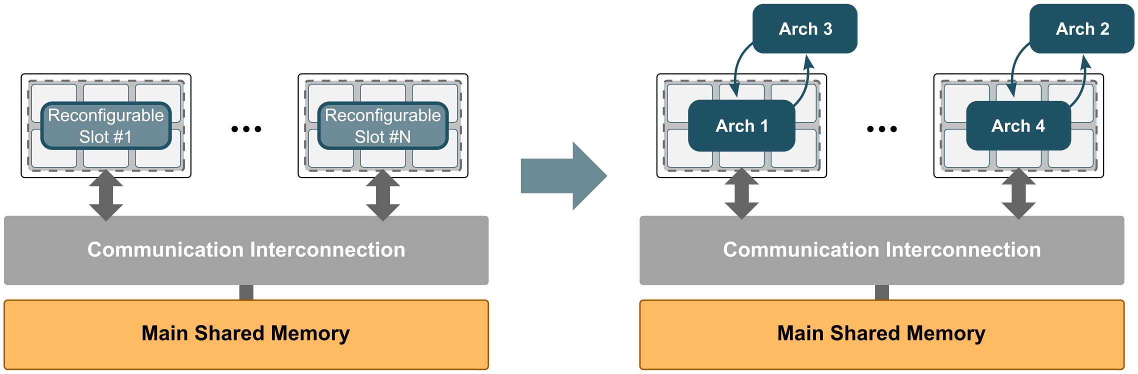


Figure 5.7: Example of an architecture model, represented by both the location and organization of the reconfigurable slots and by the possible component assignments to each of those slots. Each reconfigurable slot can accommodate only one architecture at the time. Each component is given an unique numeric identification, later used for the Architecture data-flow Graph (see Fig 5.6(c)).

module and is defined by:

- ca_i - component architecture identification;
- s_i - reconfigurable slot used by the component;
- l_{i_k} - execution latency for operations (or kernels) of type k in the component (-1 if the type of operation cannot be executed);
- ls_i - communication setup overhead (only for communication channels),
- lr_i - component reconfiguration/adaptation time;
- p_i - power consumption of the component;
- pr_i - reconfiguration/adaptation power consumption.

The latency of each communication channel, i, is given by adding its setup overhead (ls_i) to the delay of transferring a single data block (l_{ki}) times the amount of transferred data.

One of the main challenges posed by the representation of a reconfigurable accelerator is concerned with the definition of a formal model that adequately represents the adaptability of the system. To do so, since a vast number of different cluster architectures can be instantiated in each of the accelerator's reconfigurable slots, each combination of architecture-to-slot instantiation is assumed as a different component. Hence, if two consecutive tasks are mapped to two different components that correspond to the same slot, a reconfiguration command must be triggered in between, as it will be later described. Fig. 5.7 depicts an example Architecture Model, corresponding to the same system specification previously presented in Fig. 5.6(c).

Having defined both the Application and the Architecture Models, a set of Mapping Edges that represent each possible mapping of each task to all the components of the system is created, by simultaneously parsing both graphs. For each pair of processing task t_i and processing component c_j, a mapping edge m_{ij} is created if that component can execute all types of operations of that task. In particular, if two consecutive processing tasks are mapped to the same component,

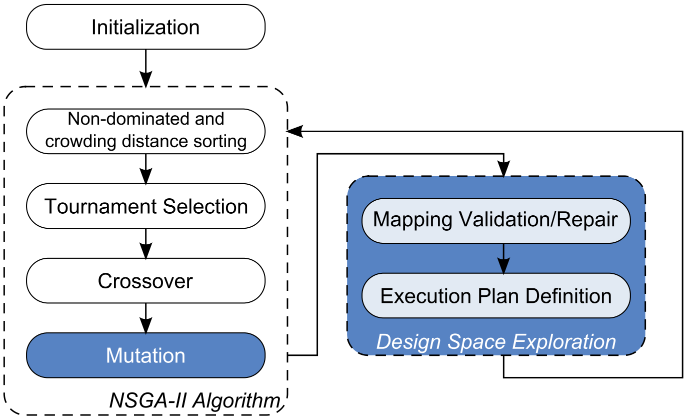

Initialization
Non-dominated and crowding distance sorting
Tournament Selection
Crossover
Mutation
NSGA-II Algorithm
Mapping Validation/Repair
Execution Plan Definition
Design Space Exploration

Figure 5.8: NSGA-II extension (highlighted blocks) for DSE.

the communication task between them can also be mapped to that same component with zero latency (detailed below).

5.2.2.B Design Space Exploration with Multi-Objective Optimization

The conceived DSE methodology is based on the Non-dominated Sorting Genetic Algorithm II (NSGA-II) (previously described in Chapter 2). Accordingly, to generate a set of execution plans for the Hypervisor, a solution space is defined according to three objective functions: *latency*, *peak power dissipation* and *total energy consumption*. Naturally, the resulting set of optimized execution plans represent the best possible trade-offs for these three goals.

To attain such a goal, the original MOO algorithm was adapted to provide a new individual encoding, based on the information gathered in the system specification models defined in Section 5.2.2.A. Consequently, appropriate crossover and mutation operators have been devised according to the considered encoding, as well as new *mapping validation/repair* and *execution plan definition* operators (see the corresponding flowchart in Fig. 5.8). In particular, the *mapping validation/repair* operator is used to decode, validate and repair the set of solutions generated by NSGA-II. Conversely, the *execution plan definition* operator is used to produce the time-ordered task and reconfiguration command execution queues (see Fig. 5.5), corresponding to the generated solutions, but also to calculate the values of each objective function. The resulting function values are then sent back to the selection operator of NSGA-II for evaluation, thus closing the optimization loop.

Problem Definition

For each individual (p) from a given population (P_t) in its t-th generation, the DSE methodology is defined as the Multi-Constraint Multi-Objective problem:

$$\min_{p \in P_t} f_p = (f_{L_p}, f_{P_p}, f_{E_p}), \text{ subject to (5.6)-(5.9)} \qquad (5.1)$$

Each objective function is formally defined according to the system specification models and variables, as described in Section 5.2.2.A. The first objective function (f_L), targeting the minimization of the execution latency (in clock cycles), is given by the finishing time of all the application's tasks:

$$f_L = \max_{t_i \in V_T} (s_{t_i} + l_{t_i}), \tag{5.2}$$

where the finishing time of a given task ($t_i \in V_T$) is obtained by adding its starting time (s_{t_i}) and its latency (l_{t_i}). Moreover, each task latency is computed by considering its assigned data chunk size (d_s), its respective number of operations (n_k) and the latency (l_{j_k}) of each operation on its mapping component (m_{ij} - binary value indicating a valid mapping):

$$l_{t_i} = d_s \times \big(\sum_k l_{j_k} \times n_k\big) \times m_{ij}, \text{ with } t_i \in V_T,\, m_{ij} \in \{0, 1\} \tag{5.3}$$

For the other two objective functions, a time frame (t) is defined to represent a period (with duration l_t), where no changes occur in the architecture and in the execution state of the accelerator's cores. The set of time frames corresponding to the total execution time of a given solution is given by T. Hence, the second objective function (f_P) minimizes the peak power dissipation (in Watts) at any given time frame:

$$f_P = \max_{t \in T} \sum_{t_i \in V_C} (p_i + pr_i \times r_i) \times a_i \ . \tag{5.4}$$

It is calculated based on the current allocation of each component (a_i), its power dissipation (p_i) and its reconfiguration/adaptation power consumption (pr_i), in case such reconfiguration occurs (r_i). The a_i and r_i variables take a binary value (0 or 1) indicating whether a component is allocated and whether reconfiguration is required, respectively.

Finally, the third objective function (f_E) minimizes the total energy consumption (in Joules):

$$f_E = \sum_{t \in T} \big(\sum_{t_i \in V_C} (p_i + pr_i \times r_i) \times a_i \big) \times l_t, \tag{5.5}$$

The consumed energy in each time frame (t) is calculated by adding the total instantaneous power dissipation of the system at that time and multiplying it by the duration of the time frame (l_t). Finally, the energy consumption corresponding to all time frames is added up.

Hence, to ensure that the algorithm generates a set of feasible solutions representing possible mappings of tasks to valid architectures, it must be defined as a constrained problem. Such constraints are herein enumerated:

1. Each processing task must be mapped to a single processing component:

$$\sum_{a \in V_C^p \subset V_C} m_{ai} = 1, \text{ for each } t_i \in V_T^p \subset V_T \tag{5.6}$$

2. Each communication task must be mapped either to a processing component or to a communication channel:

$$\sum_{a \in V_C^p \subset V_C} m_{ai} + \sum_{b \in V_C^c \subset V_C} m_{bi} = 1, \quad \text{for each } t_i \in V_T^c \subset V_T \tag{5.7}$$

3. When two adjacent processing tasks are mapped to the same component, the communication task between them must be mapped to that same component:

$$\begin{aligned} &\text{If } m_{ai} = 1 \text{ and } m_{aj} = 1 \text{ and } \exists\ t_k \in V_T \\ &\text{such that } \exists\ e_x = (t_i, t_k), e_y = (t_k, t_j) \in E_T, \\ &\text{then } m_{ak} = 1, \\ &\text{where } t_i, t_j \in V_T \text{ and } a \in V_C \end{aligned} \tag{5.8}$$

4. When two adjacent processing tasks are mapped to separate components, the communication task between them must be mapped to a communication channel that connects both components:

$$\begin{aligned} &\text{Let } t_k \in V_T : \exists\ e_x = (t_i, t_k), e_y = (t_k, t_j) \in E_T, \\ &\text{if } m_{ai} = 1 \text{ and } m_{bj} = 1, \text{ then } m_{ck} = 1, \\ &\textit{iff } \exists\ c_c \in V_C : \exists\ e_w = (c_a, c_c), e_z = (c_c, c_b) \in E_C, \\ &\text{where } c_a, c_b \in V_C \text{ and } t_i, t_j \in V_T \end{aligned} \tag{5.9}$$

The last constraint refers to the actual values of the objective functions. These values are only required to be non-negative, since the system specification model maximizes the value of each function, given the tasks' latency, component area, and power characteristics.

Individual Encoding

Having defined the DSE problem, the next step is to encode the solutions of the problem as individuals of a given population. As it is standard practice in genetic algorithms [92], the encoding of the solutions is referred to as a *chromosome*, unique to each, randomly generated and modified by the algorithm. Each chromosome is posteriorly decoded and evaluated, according to a set of operators related to the problem being solved.

In the defined DSE problem, the solutions represent possible mappings of tasks to processing cores and communication channels, according to the set of constraints defined above. This way, each solution is represented as an array, where each index accounts for a different task, and the corresponding value depicts the mapping of that task to a given component. However, such a simple representation leads to a high number of infeasible mappings (e.g., mapping of a task to a component that cannot execute all its operations). Hence, instead of a single value, each cell of this array comprehends a list that contains all the feasible mappings of its corresponding task, being the head of that list the current mapping (see Fig. 5.9). Accordingly, the head of the list is denoted as *gene*, while the rest of the list is named *gene repair list*. Hence, although very rarely and greatly depending on the system architecture, the only situation where an infeasible solution can occur is whenever two communicating tasks are mapped to two components that have

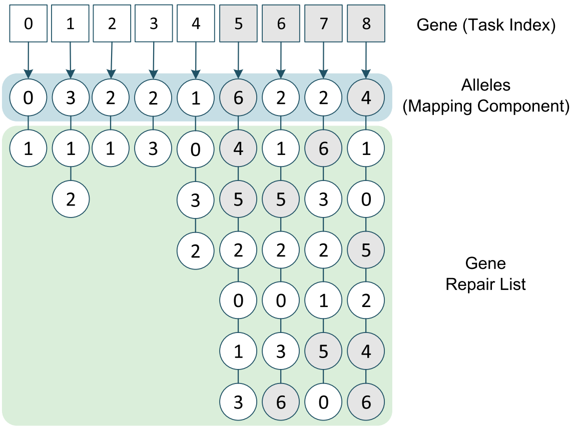


Figure 5.9: Individual encoding, representing a possible solution for the specification defined in Fig. 5.6.

no communication channels between them. While such a situation is not a typical architectural design, it may occur, especially in hierarchical architectures.

Genetic Operators

Since a custom encoding was specifically defined for the DSE algorithm, the genetic operators, i.e. *crossover* and *mutation* must also be defined accordingly.

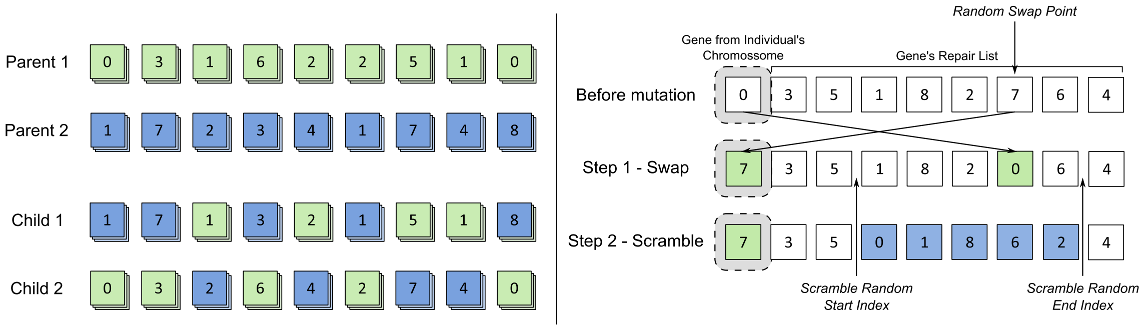
The *crossover* operator is based on the multi-point crossover method, characterized by providing the best diversity among the offspring population [130]. This operator randomly selects which genes of the chromosome array of two parent individuals are cross-copied to two newly created child individuals, as shown in Fig. 5.10.A. However, since each element of the devised chromosome is also composed of a repair list, instead of only copying the gene, both are copied to the new individual. This ensures compliance with the above-defined constraints.

Most existing encodings [131] that rely on repair lists to maintain feasible results either require several mutation and repair steps [83] or do not entirely exploit the diversity imposed by the operator [88]. To overcome both issues, a two-step *mutation* operator was devised (see Fig. 5.10.B). This operator, denoted as *Swap-Scramble* mutation, starts by first swapping the gene value with a random value from the repair list. Next, a subset of the repair list is randomly selected, and the values of that subset are randomly re-arranged or scrambled. This way, no additional procedures are required, and more diversity is achieved in the population.

DSE Operators

To complete the DSE optimization procedure, the *mapping validation/repair* and the *execution plan definition* operators are defined to evaluate and repair the set of solutions obtained from the MOO algorithm and to generate the resulting execution plans.

Given the devised solution encoding, constraints (5.6) and (5.7) are implicitly verified. In ac-



A. Multi-point Crossover Operator

B. Two-Step Swap-Scramble Mutation Operator

Figure 5.10: New DSE operators for the optimization algorithm: (A) Multi-point crossover operator and (B) two-step *Swap-Scramble* mutation operator.

Algorithm 4 Mapping Validation/Repair Operator

```
Input: $mappings[]$ ⇒ chromosome, $SSpec$ ⇒ system specification
Variables: $tc$ ⇒ communication task, $tsrc$ ⇒ source task, $tsk$ ⇒ sink task, $m(t)$ ⇒ mapping of task $t$
Output: $feasibleMapping$ (true/false)

$feasibleMapping \leftarrow$ true
for all communication tasks do
    if $m(tsrc) = m(tsk) = m(tc)$ then
        continue
    else if $m(tsrc) = m(tsk) \neq m(tc)$ then
        $m(tc) \leftarrow m(tsrc)$
    else if $m(tsrc) \neq m(tsk)$ then
        search $component$ that connects $m(tsrc)$ and $m(tsk)$ and is present in $tc$ repair list
        if $component$ found then
            $m(tc) \leftarrow component$
        else
            $feasibleMapping \leftarrow$ false
            return
        end if
    end if
end for
return
```

cordance, only communication-related infeasible mappings may occur, i.e. if constraints (5.8) or (5.9) are not satisfied. Instead of immediately discarding such solutions, a repair algorithm (Algorithm 4) that takes advantage of the considered gene repair list was also devised. Whenever constraint (5.8) is violated, the algorithm immediately maps the communication task to the component to which its source and sink tasks are mapped. On the other hand, when constraint (5.9) is violated, the algorithm tries to find (and map) an alternative component that will connect the components. If such a component is found, the algorithm immediately maps the communication task to it; otherwise, the solution is deemed unfeasible and subsequently discarded.

Accordingly, the final step of the algorithm corresponds to the operator that defines the execution plan. This operator (see Algorithm 5) serves two purposes: *i)* generation of the execution queue for the current solution; and *ii)* calculation of the values corresponding to the three objective functions. The execution plan definition algorithm is based on a common iterative scheduling procedure. By following the devised task graph, an ordered priority list is kept, comprehending all candidate tasks that are waiting to be initiated. Accordingly, a given task can be started if the

Algorithm 5 Execution Plan Definition Operator

Input: $mappings[] \Rightarrow$ chromosome, $SSpec \Rightarrow$ system specification
Variables: $alloc[] \Rightarrow$ slot configuration, $state[] \Rightarrow$ slot state, $candidates[] \Rightarrow$ candidate priority list, $inProgress[] \Rightarrow$ running tasks, $tk.map \Rightarrow$ mapping of a task
Output: $queue[], peak\_power, latency, t\_energy$

```
initialize $queue[]$ based on $mappings[]$ and $SSpec$
$candidates[] \leftarrow$ input tasks
$peak\_power, t\_energy \leftarrow 0$
$time \leftarrow 1$
repeat
  for each task $tk$ in $candidates[]$ do
    $ready \leftarrow 1$
    for each $source$ in $tk$ do
      if $source.start \neq -1$ and $source.start + source.latency < time$ then
        continue
      else
        $ready \leftarrow 0$; break
      end if
    end for
    if $ready = 1$ and $state[tk.map.slot] = 0$ then
      if $alloc[tk.map.slot] \neq tk.map.arch$ then
        $tk.latency \leftarrow tk.latency + tk.map.rec\_time$
        $tk.power \leftarrow tk.power + tk.map.rec\_power$
      end if
      $alloc[tk.map.slot] \leftarrow tk.map.arch$
      $state[tk.map.slot] \leftarrow 1$
      $tk.start \leftarrow time$
      $inProgress[] \leftarrow tk$ (in order of finish time)
      $queue[] \leftarrow tk$

      add all $tk.edges[]$ to $candidates[]$
    end if
  end for
  add up $power$ for all $tk$ in $inProgress[]$
  $peak\_power \leftarrow max(peak\_power, power)$
  $tk \leftarrow pop(inProgress[])$
  $ntime \leftarrow tk.finish$
  $state[tk.map.slot] \leftarrow 0$
  $t\_energy \leftarrow t\_energy + power \times (ntime - time)$
  $time \leftarrow ntime$
until all tasks enqueued
$latency \leftarrow time$
return $queue[]$
```

reconfigurable slot of its mapping component is not in use. As soon as such slot becomes free, the architecture of the mapped component is checked against the former component that was previously allocated to that slot. In case the architectures are different, a reconfiguration command must be triggered before the task can be started.

Furthermore, every time there is a change in the configuration of the system (i.e., a reconfiguration or a component's execution starts or finishes), its total power dissipation is calculated, as well as the period while it stays constant. This way, the peak power dissipation can be obtained by comparing those measures and by choosing the minimum. The total energy consumption is calculated by adding the products of the power measures by their corresponding durations. Finally, the computed latency value corresponds to the time when the last task completes its execution.

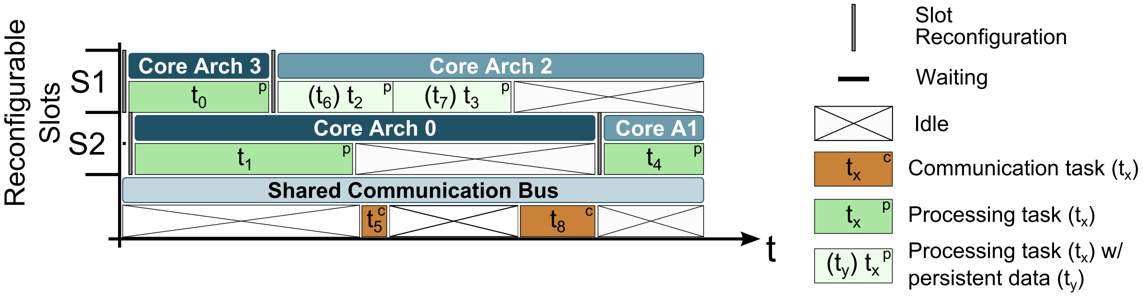


Figure 5.11: Example of an execution queue, obtained with Algorithm 5 for the encoding in Fig. 5.9, corresponding to the mapping of the application described with the Task Graph in Fig. 5.6(a) to the architecture in Fig. 5.7. Note that when two adjacent tasks are mapped to the same component, the communication task between them is mapped to that same component with zero latency (shown between parenthesis in the second task's execution).

5.2.2.C Optimized Execution Plans

Each solution that is obtained from the DSE algorithm is represented by an execution plan. This plan contains the time-ordered execution queue (as shown in Fig. 5.11), the mappings and the required reconfiguration operations for the generated solution. Then, a post-processing routine selects a representative subset of the obtained optimal solutions (based on the system's processing requirements) and provides them to the Hypervisor in the form of execution plans. In turn, the Hypervisor uses these execution plans in one of the two possible modes, i.e., in *Scheduler* or *Resource Management* mode.

5.3 Experimental Validation

This section presents a set of experimental case studies that aim at demonstrating and validating the adaptability of the designed reconfigurable many-core accelerator and optimization mechanisms. The presented experiments do not intend on providing an in-depth evaluation, but instead to assess the viability of exploiting the application's characteristics (before and during execution) to specialize and optimize the underlying processing system, in turn offering new computing efficiency capabilities.

5.3.1 Methodology

For this evaluation, three case studies were considered that demonstrate the accelerator's adaptability according to the proposed Hypervisor operation modes, namely:

- **A -** The first case study evaluates the Hypervisor's automatic application characterization and reconfiguration optimization, according to the implemented reconfiguration policies;
- **B -** The second case study demonstrates the gains obtained by running the proposed DSE algorithm to generate optimized execution plans for the Hypervisor;

- **C -** The last case study assesses the capabilities of the DSE algorithm to optimize the execution of applications with dynamic workloads.

To perform the aimed experimentations, the first two case studies were constructed around an arithmetic benchmark designed to demonstrate the adaptability offered by the reconfigurable accelerator and assess its potential for improving performance and energy efficiency. The benchmark is composed of 4 phases, each one corresponding to a linear algebra data-parallel computation kernel with distinctive processing requirements, namely: *i)* Kernel 1 performs the sum of two integer vectors (Eq. 5.10); *ii)* Kernel 2 computes the inner product of two integer vectors (Eq. 5.11); *iii)* Kernel 3 performs the sum of two single-precision floating-point vectors (Eq. 5.12); and *iv)* Kernel 4 computes the inner product of two single-precision floating-point vectors (Eq. 5.13).

$$\textbf{Kernel 1}: \quad \boldsymbol{v}_i^I = \boldsymbol{a}_i^I + \boldsymbol{b}_i^I \Big|_{i=1,\ldots,N} \tag{5.10}$$

$$\textbf{Kernel 2}: \quad \alpha^I = \textstyle\sum_{i=[1,N]} \boldsymbol{a}_i^I \times \boldsymbol{b}_i^I \tag{5.11}$$

$$\textbf{Kernel 3}: \quad \boldsymbol{v}_i^F = \boldsymbol{a}_i^F + \boldsymbol{b}_i^F \Big|_{i=1,\ldots,N} \tag{5.12}$$

$$\textbf{Kernel 4}: \quad \alpha^F = \textstyle\sum_{i=[1,N]} \boldsymbol{a}_i^F \times \boldsymbol{b}_i^F \tag{5.13}$$

Each input dataset ($\boldsymbol{a}_i^I$, $\boldsymbol{b}_i^I$, $\boldsymbol{a}_i^F$ and $\boldsymbol{b}_i^F$) is an independent and randomly generated array with 300 million integer/floating-point cells. These datasets are then partitioned and dynamically assigned to the processing clusters by the Hypervisor mechanism at runtime. Since the main focus of this study is to evaluate the accelerator's reconfiguration capabilities, this experimental evaluation assumes that data is locally stored in the processing cluster. As such, all data is initially stored in each core local memory, to alleviate the contention in the memory hierarchy.

Accordingly, the first two case studies adopt the reconfigurable accelerator implementation detailed in Section 5.1.6. To summarize, the accelerator is composed of 7 reconfigurable slots interconnected by a shared communication bus. Each slot is able to instantiate one of the three possible cluster types (A, B and C), each composed of a set of homogeneous core architectures, or turned off to manage power supply. Although all core architectures support the necessary operations to execute the complete arithmetic benchmark, the simpler architectures rely on software libraries (instead of hardwired instructions) to execute the most complex arithmetic operations (e.g., multiplication and FP operations). In particular, the most complex architecture (Type C) provides hardware specific units for every type of operations (integer and FP addition and multiplication). On the other hand, the intermediate architecture (Type B) only includes hardware units for integer addition and multiplication (no FP operations) and the simpler architecture (Type A) only includes arithmetic units for simple integer operations (no multiplication or FP operations). These differences result in different processing latencies for each kernel and in distinct area requirements for the corresponding cores, which in turn result in more or less cores instantiated in each cluster.

The third case study relies on a proof-of-concept benchmark based on a biological sequence alignment application model [132, 133] (further detailed in the case study). The adopted accelerator model considers the existence of seven reconfigurable slots (cluster architectures are also detailed in the case study) interconnected by a streaming communication bus similar to the bidirectional ring described in Section 4.3.2.

5.3.2 Case Study A: Learning-based Automatic Reconfiguration

The first considered case study demonstrates the Hypervisor's capability for runtime application monitoring and characterization. This is performed by first considering two different scenarios without any previous knowledge of the application being executed, resulting in the definition of an optimized execution model. This model is then used to demonstrate the designed reconfiguration policies. For all the conducted experiments, the accelerator was initially configured with an equally balanced distribution of computing cluster architectures (3× *Type A* + 2× *Type B* + 2× *Type C*). The observed results are shown regarding the attained performance gains and energy savings.

5.3.2.A Runtime Monitoring and Learning

To demonstrate the capabilities of the Hypervisor to provide the best-fitted architecture for each kernel, two different execution scenarios were considered. Both these scenarios assume an untrained Hypervisor, with no *a priori* knowledge of the kernels' characteristics. The considered scenarios assess the learning capabilities of the Hypervisor by running the benchmark's computing kernels in different orders.

The dynamic adaptation that is conducted by the Hypervisor is depicted in Figs. 5.12 and 5.13. Initially, the Hypervisor allows each cluster to execute its assigned chunk of a kernel with its currently assigned configuration. Then, upon completion of the first kernel chunk, the Hypervisor sends a reconfiguration command to that same cluster, to adapt its architecture to the currently executing kernel, according to the set of received values of the performance counters. The longer execution time observed in Fig. 5.13 is due to the first chunks of kernels 3 and 4 being initially executed in clusters of *Type A* and *Type B*. This means that the FP operations present in those kernels are initially executed with software libraries, resulting in increased latency. As such, the Hypervisor only knows that FP operations are needed after the execution of the first chunk, and only then it issues the appropriate reconfiguration command.

5.3.2.B Reconfiguration Policies

The application models obtained after the execution of the application in the previously described untrained scenarios can be used to demonstrate the implemented reconfiguration policies. Accordingly, as soon as the Hypervisor has access to the running application's characteristics, it

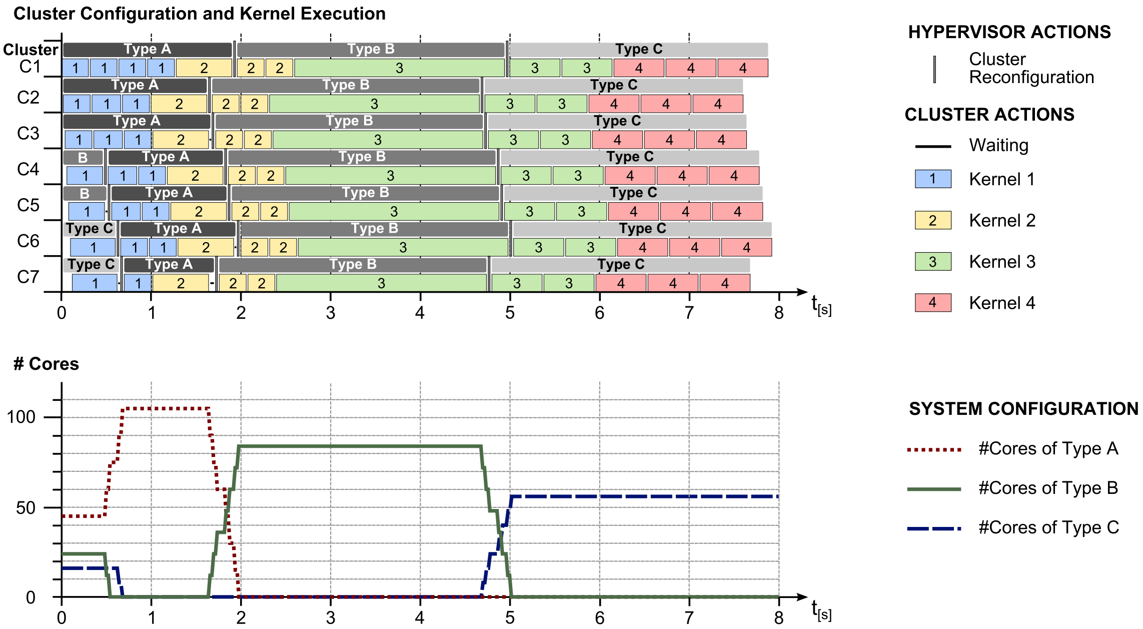


Figure 5.12: Real-time adaptation of the processing architecture, without any *a priori* knowledge of the computing kernels (Kernel order: 1, 2, 3, 4).

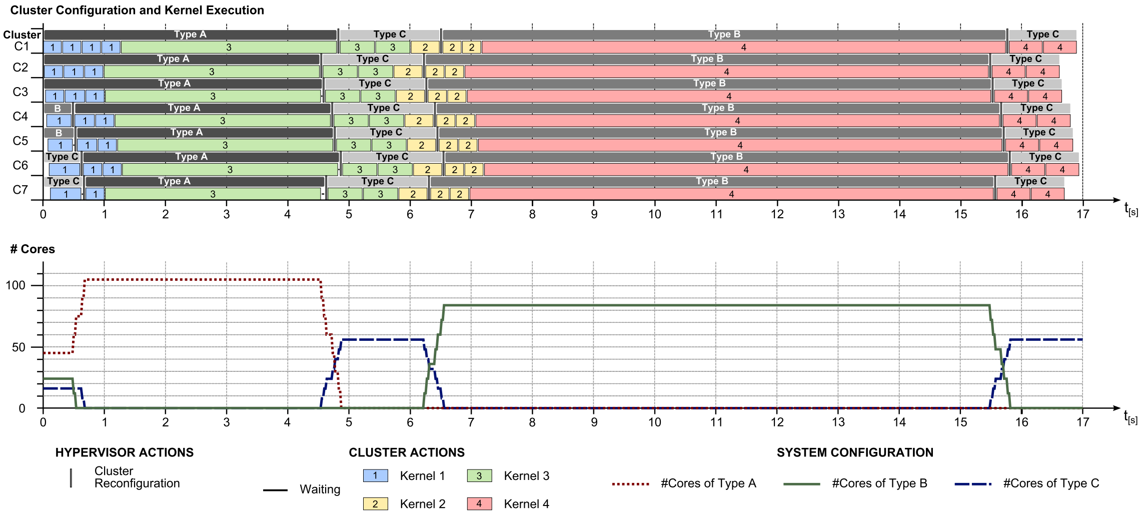


Figure 5.13: Real-time adaptation of the processing architecture, without any *a priori* knowledge of the computing kernels (Kernel order: 1, 3, 2, 4).

can immediately trigger the reconfiguration process to adapt a cluster to the best-fitted architecture for a given kernel, according to the deployed reconfiguration policy.

This is particularly clear when testing the execution time policy described in Algorithm 1, which allows the system to dynamically select the set of clusters that provide the best performance for each kernel under execution. The experimental results for this policy are presented in Fig. 5.14 and allow concluding that the system is able to adapt to the best possible configuration, while also achieving a well-balanced data chunk distribution to the several processing clusters.

The second deployed reconfiguration policy considers the maximization of the system per-

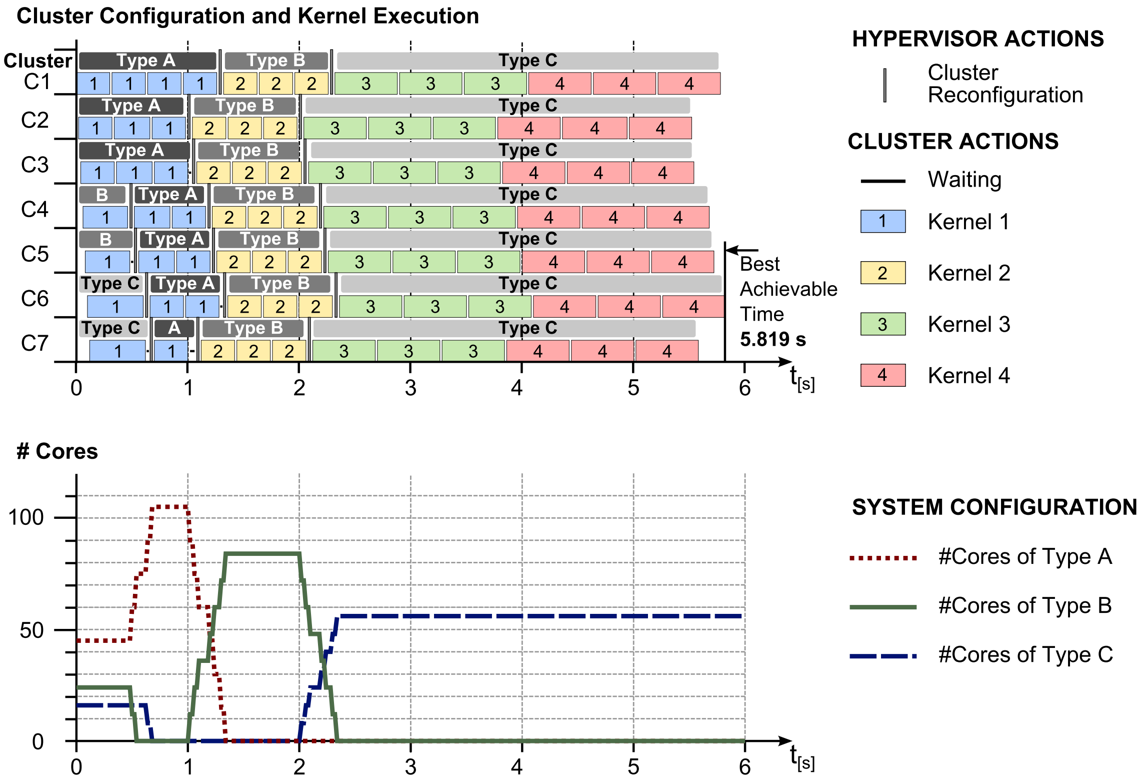


Figure 5.14: System real-time adaptation, according to the minimum execution-time optimization policy.

formance while establishing a given power ceiling (see Algorithm 2). To ensure a more realistic test, this power-ceiling was also varied at runtime. In Fig. 5.15 it is possible to observe some clusters being replaced by empty configurations when the power ceiling decreases, to meet this constraint. On the other hand, as soon as the allowed power consumption level increases, the system reactivates these turned-off clusters, to maximize the accelerator throughput.

The last optimization policy, previously described in Algorithm 3, considers the minimization of the power consumption while assuring a minimum performance level. To show the adaptivity of the proposed system, it is further assumed that the application under execution establishes a different minimum throughput for each kernel, as shown in Fig. 5.16. As it can be observed, the system can adapt the clusters in real-time, not only to ensure the required performance level but also to minimize the power consumption, by disabling inactive clusters.

5.3.2.C Execution Time Speedup and Energy Savings

To evaluate the performance gains and energy savings that are obtained with the reconfigurable accelerator, the dynamic execution policy presented in Fig. 5.14 was compared with four different static configurations (i.e., without reconfiguration), each one composed by 7 independent clusters: *i*) a system with 7 *Type A* clusters; *ii*) a system with 7 *Type B* clusters; *iii*) a system with 7 *Type C* clusters; and *iv*) a heterogeneous mix composed of 2 *Type A* clusters, 2 *Type B* clusters

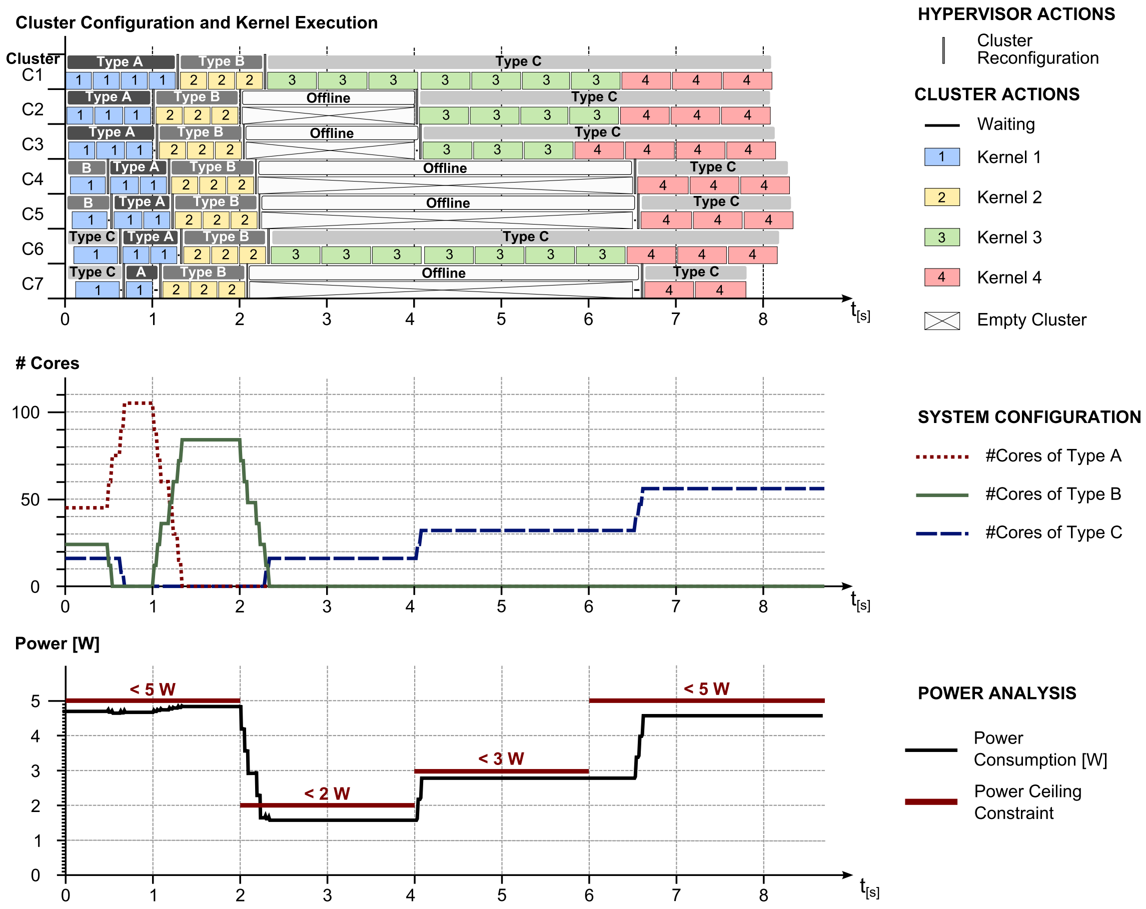


Figure 5.15: System real-time adaptation, according to the power-ceiling constraint policy.

and 3 *Type C* clusters.

Table 5.4 presents the obtained results regarding the execution time and energy consumption for the considered setups. Despite containing 105 cores, it can be observed that the system with only *Type A* clusters represents the worst case, both regarding performance and energy. This is explained by the fact that *Type A* clusters must perform the multiplication operations of Kernel 2 through a combination of logic shifts and additions, and the floating-point operations of Kernels 3 and 4 through calls to software libraries. Naturally, this represents a large energy overhead, which results in a total consumption of 240 Joules. The best homogeneous static configuration is obtained by using only *Type C* clusters. Even though only 56 cores can be implemented in this case, it performs about 4× faster than the worst-case configuration. The best static configuration was achieved by using the considered heterogeneous configuration (2× *Type A* + 2× *Type B* + 3× *Type C*), which provides a trade-off between complexity and execution time/energy consumption.

Finally, it can be observed that the offered adaptability allows the dynamic system to combine all the advantages of the above-described configurations. By adapting, at runtime, to the requirements of the different kernels, it is assured that the system always provides the best-optimized configuration for each application phase, by trading core complexity with the total number of cores.

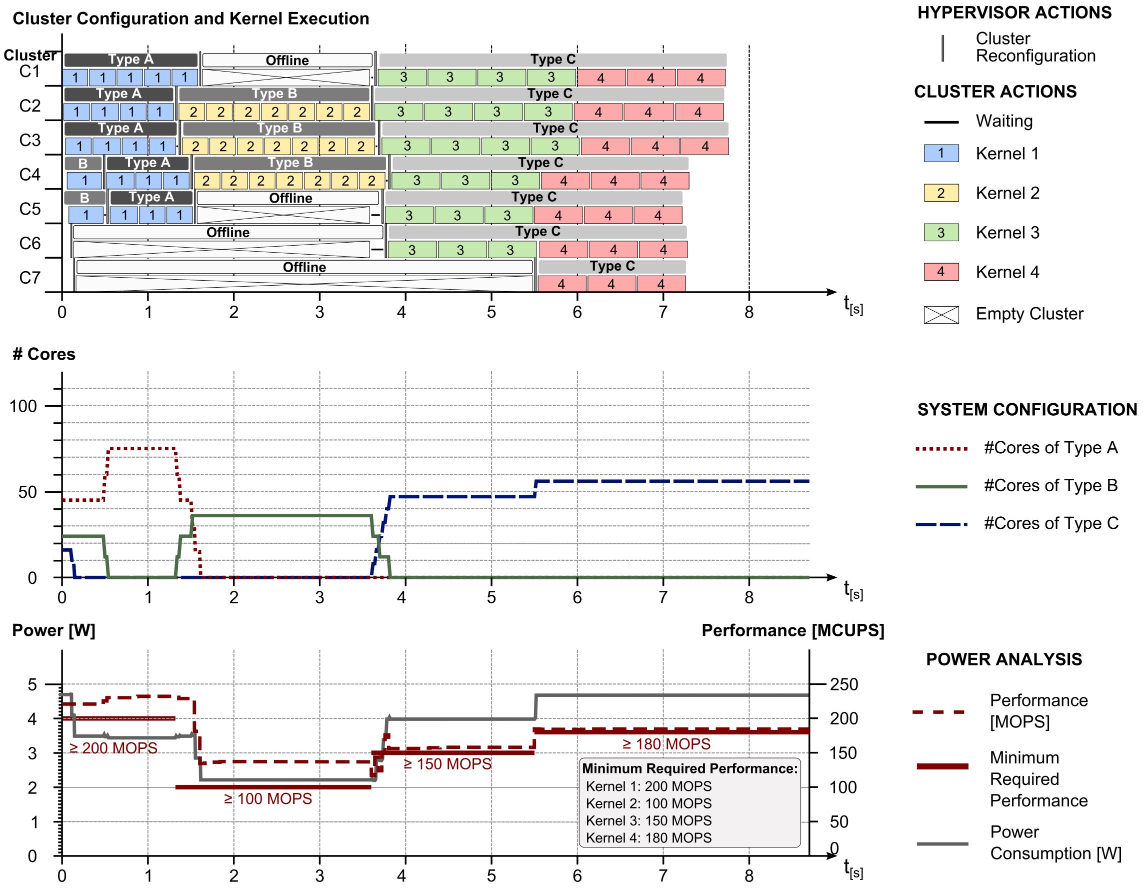


Figure 5.16: System real-time adaptation, according to the minimum assured performance policy.

Table 5.4: Execution time and energy results.

	Execution Time [s]	Energy Consumption [J]	Dynamic System Speedup	Dynamic System Energy Gain
Dynamic System	5.819	23.28	-	-
Static 7× *Type A* Clusters	55.715	239.93	9.575	10.31
Static 7× *Type B* Clusters	29.784	132.72	5.118	5.70
Static 7× *Type C* Clusters	11.997	50.44	2.062	2.17
Static Heterogeneous Mix	18.378	79.13	3.158	3.40

Thus, it is possible to achieve execution speedups ranging from 2.1×, when compared with the best static case, to 9.5×, when compared to the worst static case, while consuming from 2.2× to 10.3× less energy, respectively.

5.3.3 Case Study B: Compiler-Assisted Reconfiguration

The second case study demonstrates the Hypervisor's operation in *Scheduler* mode. In this mode, it relies on the set of optimized execution plans generated by the DSE algorithm (before execution) for the considered arithmetic benchmark.

To generate the set of optimized execution plans, the adopted NSGA-II algorithm was configured with a population of 200 individuals in 2000 generations and crossover and mutation prob-

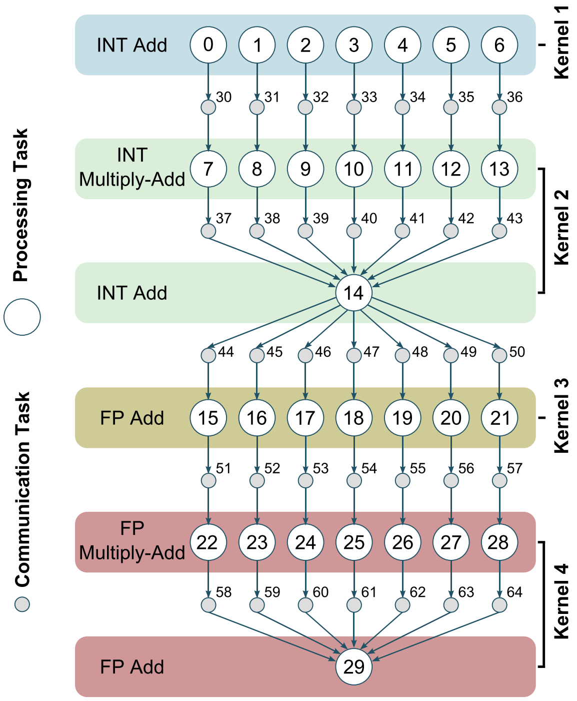


Figure 5.17: Arithmetic benchmark application model.

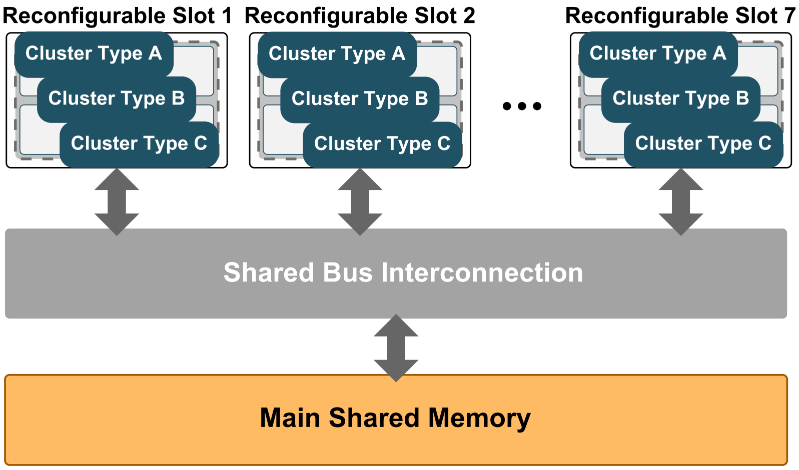


Figure 5.18: Arithmetic benchmark architecture configuration.

abilities of 0.95 and 0.2, respectively. This configuration was initially based on the ranges of parameter values suggested in [93] and further fine-tuned by observing the algorithm's evolution, regarding convergence characteristics and spread of the resulting solutions in the Pareto Front.

For this experiment, the DSE algorithm was configured with a data-flow model (Task Graph) of the devised arithmetic benchmark (as shown in Fig. 5.17) and an architecture model (Architecture Graph) of the implemented reconfigurable accelerator (depicted in Fig. 5.18). The characteristics of each cluster configuration are shown in Table 5.5, together with the latency of each vector operation on every core architecture (measured in Clock Cycles per Operation (CCPO)) and their

Table 5.5: Arithmetic kernel benchmark characterization (operation latency is shown in average Clock Cycles per Operation (CCPO)).

		INT Latency [CCPO]		**FP Latency [CCPO]**		**Communication**	**Reconfiguration**	
Components (# Cores)	Power [mW]	Add	Multiply	Add	Multiply	**Latency [CCPO]**	Latency [ms]	Power [mW]
Cluster Type A (15)	615.20	2.15	4.38	25.09	51.39	0	10	44
Cluster Type B (12)	636.58	2.44	2.19	16.78	15.83	0	10	44
Cluster Type C (8)	600.65	3.66	3.28	3.79	3.79	0	10	44
Shared Bus (n.a.)	168.10	n.a.	n.a.	n.a.	n.a.	10 (avg.)	n.a.	n.a.

Support Power	367.9 mW
Operating Frequency	100 MHz

corresponding power dissipation. It also includes the reconfiguration latency and power dissipation, as well as the power dissipation of the accelerator's supporting infrastructure. This way, a comprehensive model of the entire reconfigurable accelerator could be established and applied to the DSE algorithm.

The obtained Pareto Front is depicted in the plots from Fig. 5.19, representing the interactions between each objective function in the three-dimensional search space. It represents the interactions between peak power dissipation and latency; between energy consumption and latency; and between peak power dissipation and energy consumption, plotted in Figs. 5.19.A, 5.19.B and 5.19.C, respectively. Moreover, the observed spread of the solutions (also unaffected) results in several levels of optimization, representing different trade-offs between the defined objective functions. This provides the Hypervisor with greater flexibility for adapting the execution to system runtime requirements.

The execution plans depicted in Figs. 5.20 and 5.21 represent two possible solutions taken from the Pareto Front (see Fig 5.19). Fig. 5.20 depicts a solution that represents the best execution-time optimized plan, mostly trading off latency for peak power consumption. By analyzing the execution queue of this solution, it can be concluded that even though it is best optimized for latency, it also presents some reduction of the peak power dissipation when running Kernels 1 and 2, by turning off some unnecessary clusters. The solution depicted in Fig. 5.21 illustrates the execution plan with the lowest peak power dissipation, although resulting in higher execution time. In this case, the system uses two slots in parallel, for a short period, to run Kernels 1 and 2 (with lower latencies), while for Kernels 3 and 4, corresponding to the most time-consuming kernels, only one slot is used.

The solution depicted in Fig. 5.21 represents, respectively, 54% and 45% lower peak power dissipation and energy consumption values than those obtained with the solution depicted in Fig. 5.20. It clearly shows that the Hypervisor (in *Scheduler* mode) can fully take advantage of the execution plans generated by the DSE algorithm when adapting the execution to system runtime requirements. These results are sustained by the observed Pareto Front, comprehending solutions that represent different trade-offs between the considered optimization targets, thus providing different levels of performance, peak power dissipation, and energy consumption. In

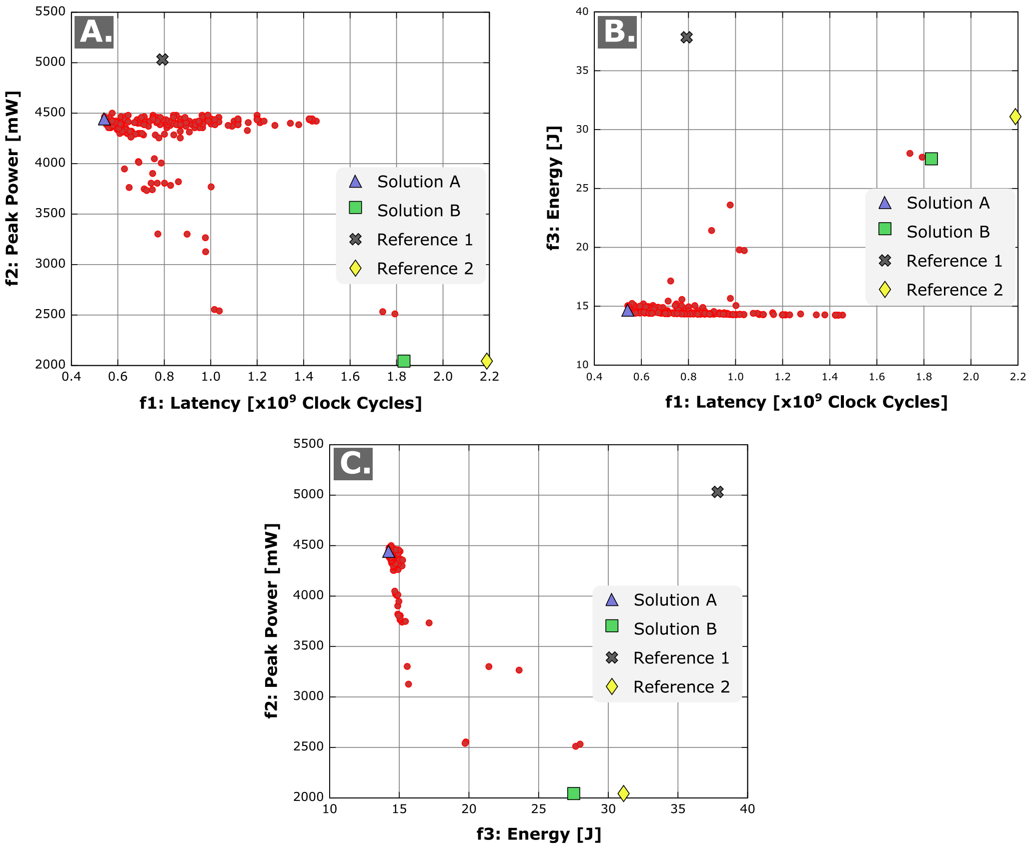


Figure 5.19: Interactions between the three objective functions in the Pareto front. (A) interaction between peak power dissipation (Eq. 5.4) and latency (Eq. 5.2); (B) interaction between energy consumption (Eq. 5.5) and latency (Eq. 5.2); and (C) interaction between peak power dissipation (Eq. 5.4) and energy consumption (Eq. 5.5). The marked solutions represent the execution plans shown in Figs. 5.20 and 5.21 (*Solution A* and *Solution B*) and the experiments performed in Case Study A for the minimum execution time policy (*Reference 1*) and for the power-ceiling policy (with a fixed 2 W limit - *Reference 2*).

fact, when compared with the Hypervisor's reconfiguration policies, the DSE algorithm allows the Hypervisor to adapt the system to real-time requirements (by switching the execution plans) without any runtime penalization to re-evaluate the best-suited cluster configuration. Moreover, it also allows greater flexibility to cope with such requirements, given the number of generated solutions.

5.3.4 Case Study C: Resource Management for Dynamic Workloads

The execution complexity of a running application may be directly related to the characteristics of the input data. Moreover, user-defined parameters often provide additional levels of complexity. As a result, such applications may be difficult to model and optimize with a compile-time algorithm, since the models must take into account those parameters and the variable characteristics of the input data itself. As such, this third case study depicts how the execution of such applications can be optimized by taking advantage of the *Resource Management* mode of the

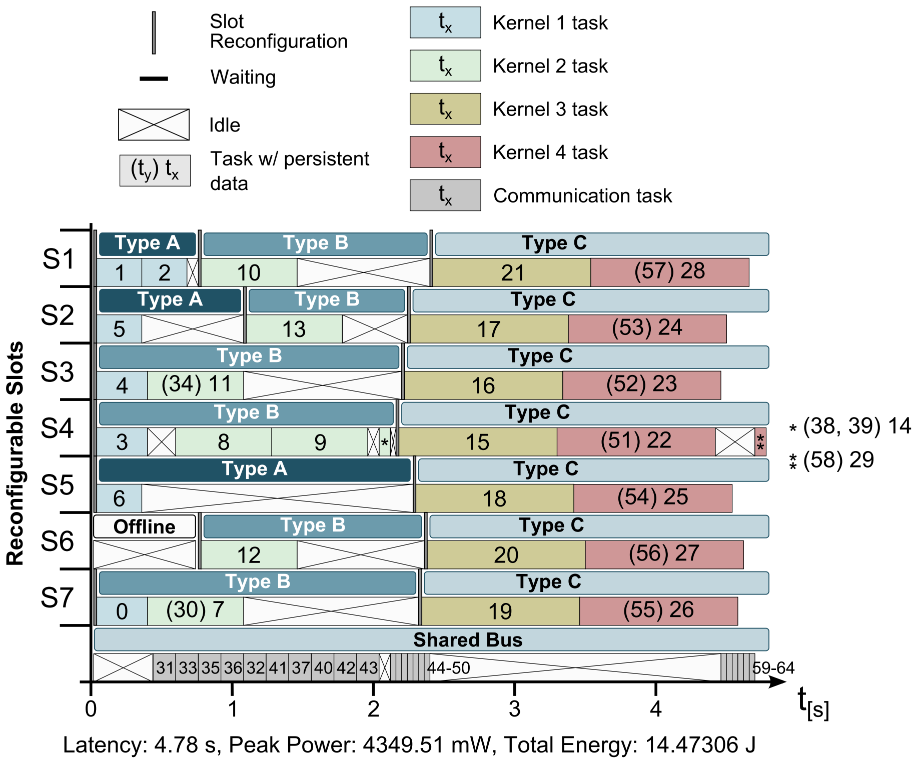


Figure 5.20: Execution plan referring to the application and architecture models from Figs. 5.17 and 5.18, representing the best execution-time optimized solution obtained from the resulting Pareto front. Task identifications shown between parenthesis represent communication tasks between two adjacent tasks mapped to the same component. In this case, the data corresponding to the communication tasks persists in the component, while switching the processing tasks.

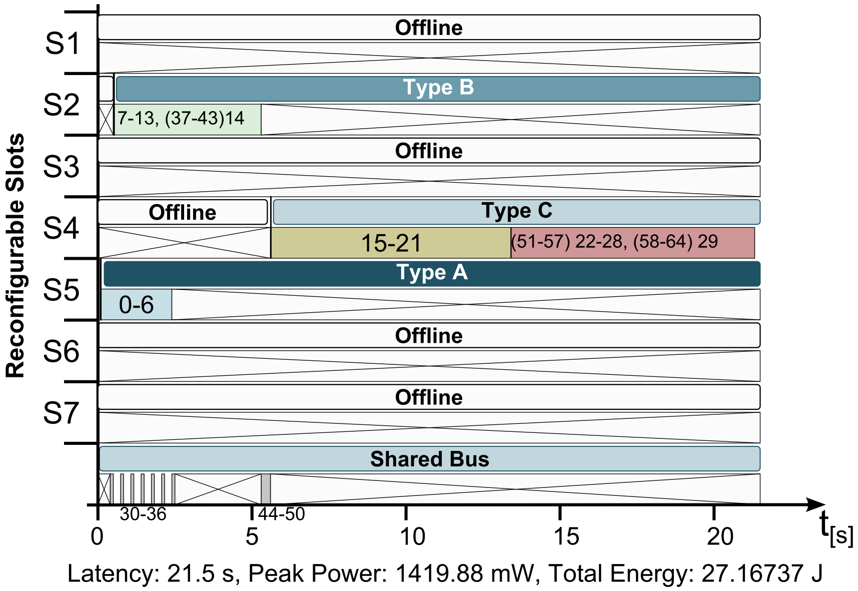


Figure 5.21: Execution plan referring to the application and architecture models from Figs. 5.17 and 5.18, representing the lowest peak power dissipation solution obtained from the resulting Pareto front.

Hypervisor. It is demonstrated that by varying specific parameters in the application model, different search granularities and intensities are attained, resulting in different levels of output data size and latency in each phase. Such a scenario can be modeled by running the DSE algorithm with different parameter levels, thus generating different execution plans for different application workload intensities. The execution plans can then be used by the Hypervisor to adapt the system architecture according to the application's real-time requirements.

Accordingly, this case study relies on a proof-of-concept benchmark based on a biological sequence alignment application model [132, 133], composed of multiple processing phases and comprehending several existing dedicated processing architectures for each of those phases. According to existing implementations [134], a heuristic *Indexed Search* within the whole data set is initially performed, to identify regions of interest in a reference biological sequence for a set of query sequences to be analyzed. Then, a *Filtering* phase is followed, that further refines the search. The execution of both phases is performed according to specific user-defined parameters. The third and final phase of the application comprehends a *Local Alignment* algorithm, which takes the resulting interest regions, carries out the local alignment and returns the corresponding score results.

5.3.4.A Architecture and Application Models

The modeled accelerator structure considers the same accelerator topology as the previous cases studies, with 7 reconfigurable slots. However, for this case study, they interconnected through a streaming communication bus (deploying point-to-point communication between all slots). The adopted core architectures are based on existing processing architectures [28, 120, 134], specially optimized for each phase. Accordingly, each available cluster configuration was also specifically dimensioned for each application phase (see Table 5.6). In particular, the *Indexed Search* phase is executed on the Application Specific Instruction-set Processor (ASIP) architecture proposed in [134], referred here as `Index ASIP`; the *Filtering* phase is executed on the MB-LITE soft-core architecture [120], denoted as `MB-LITE M`; and the *Local Alignment* phase is performed on the ASIP architecture proposed in [28], named `Align ASIP`.

Although a total number of seven reconfigurable slots are considered, slots 1, 2 and 3 are reserved for each of the processing architectures, as shown in Fig. 5.22.C, to ensure that at least one single cluster of each of the three architectures is always present in the system, interconnected in a pipelined manner (according to the application model in Fig.5.22.A). Further details about each component are presented in Table 5.6.

5.3.4.B Dynamic Workload Modeling

To demonstrate how the adopted application can make use of the DSE algorithm, illustrative models of the complexity and output data size of each phase were derived, by taking into account

Table 5.6: Biological sequence alignment case study: component specifications and application model equations.

Execution Phase	Components (# Cores)		Power [mW]	Latency [CCPO]	Output Size [# cells]	Reconfiguration Latency [ms]	Reconfiguration Power [mW]
Index Search	Index ASIP	(14)	701.30	$[K + log_2(M - K)] \times \frac{N}{K}$	$\text{HITS} = \frac{N \times (M-K)}{K^2} \times \beta$	10	44
Filtering	MB-LITE M	(12)	636.58	$\text{HITS} \times log_2(\text{HITS}) + \text{HITS}$	$\text{P. SITES} = \text{HITS} \times R/\gamma^2$	10	44
Local Alignment	Align ASIP	(5)	610.50	$\text{P. SITES} \times 2N^2/\alpha$	n.a.	10	44
Communication	Stream Interconnect	(1)	101.78	1 (per node hop)	n.a.	n.a.	n.a.

Support Power	367.9 mW
Operating Frequency	100 MHz

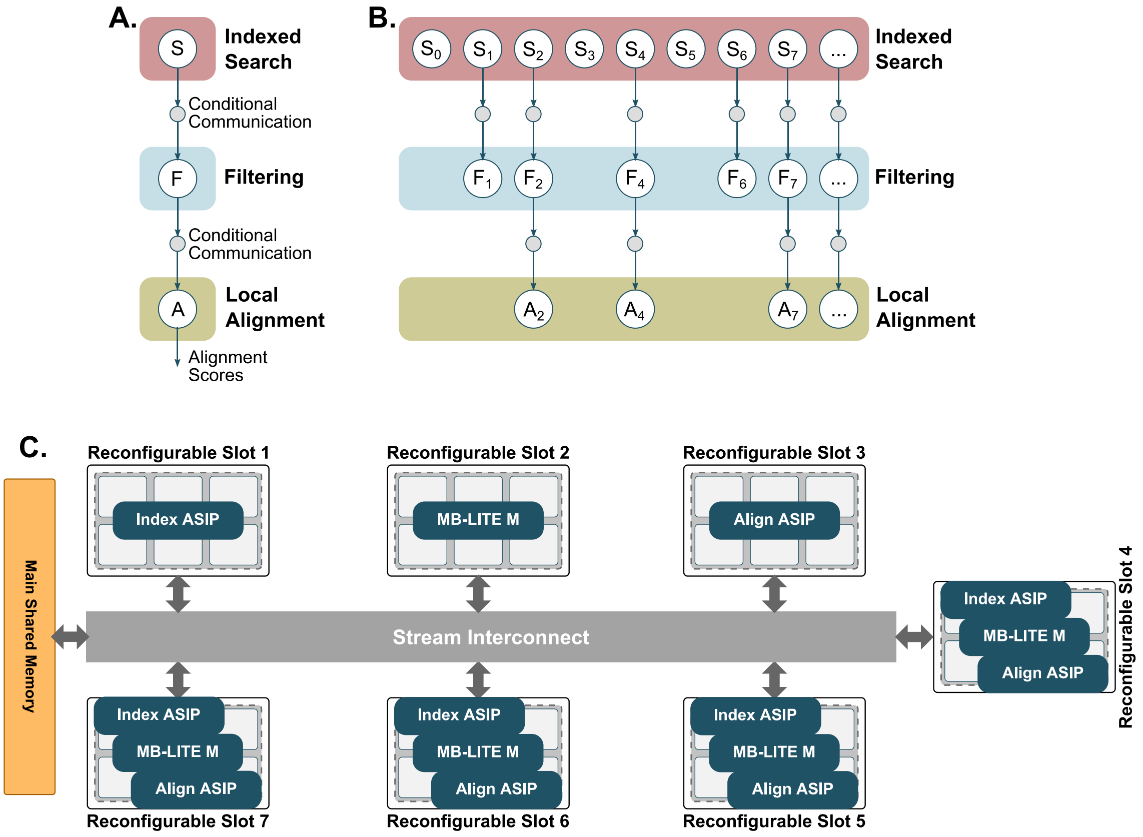


Figure 5.22: Biological sequence alignment application model (A); example execution with variable application phase workloads (B); and architecture configuration (C). Although the application model accounts for every possible outcome for the communication between phases, the size of the output of each phase (if any) is only known during runtime.

the characteristics of those applications [123, 132, 133]. The *Latency* and *Output Size* columns in Table 5.6 represent approximate models that are used to estimate the latency and output data size of each execution phase, depending on user-defined parameters and the size of the reference and query sequences. For the *Indexed Search* phase latency, the K value represents a user-defined parameter that defines the size of the search seed, while M and N represent the sizes of the reference and query sequences, respectively. The output size formula defines the amount of resulting pointers (addresses) corresponding to the obtained regions of interest, i.e., the number of HITS. In this equation, β represents a data-dependent value that reflects the characteristics of the reference sequence. The *Filtering* phase refines the indexed search results, by further reducing the number of regions of interest (HITS), leading to a smaller subset of potential sites (P. SITES) for local alignment. The latency and data output size of this phase depend on the amount of HITS resulting from the Indexed Search phase, where R is a user-defined parameter to adjust the filtering intensity, while γ is a variable characterizing the similarity between the query and the reference sequences. The last phase implements a *Local Sequence Alignment* based on a dynamic programming approach. Its latency depends on the number of P. SITES resulting from the filtering phase and on the size of the query sequences. The α parameter is a speedup factor that depends on the used alignment algorithm [123].

By varying the user-defined parameters and the intrinsic characteristics of the input data set, several different scenarios can be exploited by the DSE algorithm, resulting in distinct execution plans that will be used by the Hypervisor to adapt the execution in real-time. Although the latency and output sizes have a direct relation with the characteristics of the input data set, in the considered case the β and γ values were set to 0.7 and 0.5, respectively, representing a fair differentiation across the query and reference sequences, respectively. The α value was set to 6, corresponding to the SIMD striped search algorithm [123]. Furthermore, two levels of user-defined parameters were used, representing different scenarios for a wide relaxed search ($K = 100$, $R = 0.1$) and a fine-tuned intensive search ($K = 15$, $R = 0.01$).

5.3.4.C Dynamic Resource Management Optimization

By extracting the mappings of the execution plans resulting from the DSE algorithm, the architecture configurations shown in Figs. 5.23 and 5.24 were derived to be provided to the Hypervisor. This way, the Hypervisor can adapt the architecture (in runtime) in two different abstraction levels. On a higher management level, the Hypervisor monitors the result of the application and switches the execution plan between the available scenarios, depending on user requirements or on the input data characteristics. For instance, between a relaxed search scenario (depicted in Fig. 5.23) and a more intensive search (shown in Fig 5.24). On a lower management level, the Hypervisor can adapt the architecture in each reconfigurable slot depending on the data flow intensity existing between phases of the application, according to the execution plan being considered at that instant.

Hence, the mapped processing cluster types and reconfigurable slot allocations, shown in Figs. 5.23 and 5.24, represent execution-time optimized solutions for the considered scenarios. It is possible to verify the different number of cores of each architecture allocated according to the intensity of each phase of the application, resulting from the chosen input parameters. As it should be expected, for a wide and relaxed search, the *Indexed Search* phase is characterized by a lower complexity and results in few regions of interest, leading to a relaxed *Filtering* phase and fewer potential alignment sites for the *Local Alignment* phase. This results in an even allo-

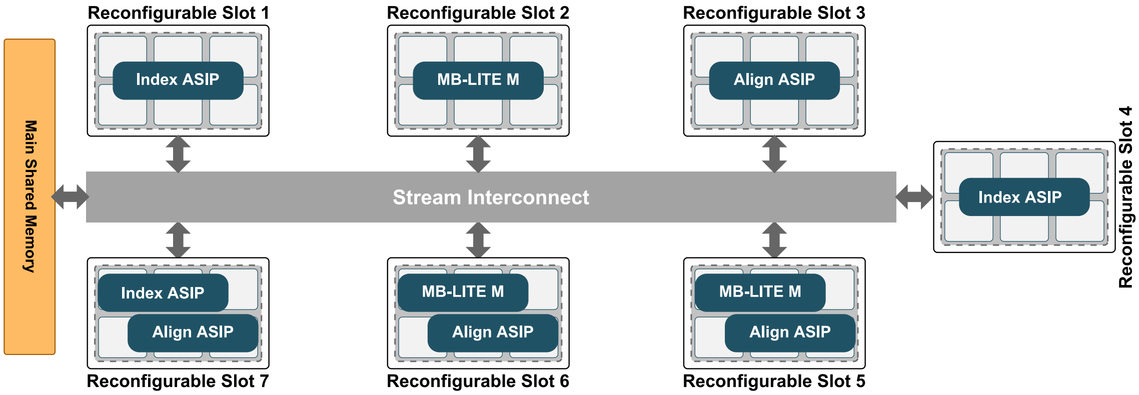


Figure 5.23: Resulting relaxed optimization execution plan for the biological sequence alignment application. Even though the plan is optimized for a specific execution scenario, the Hypervisor can still fine tune the configuration of the system according to the amount of data outputted by each phase.

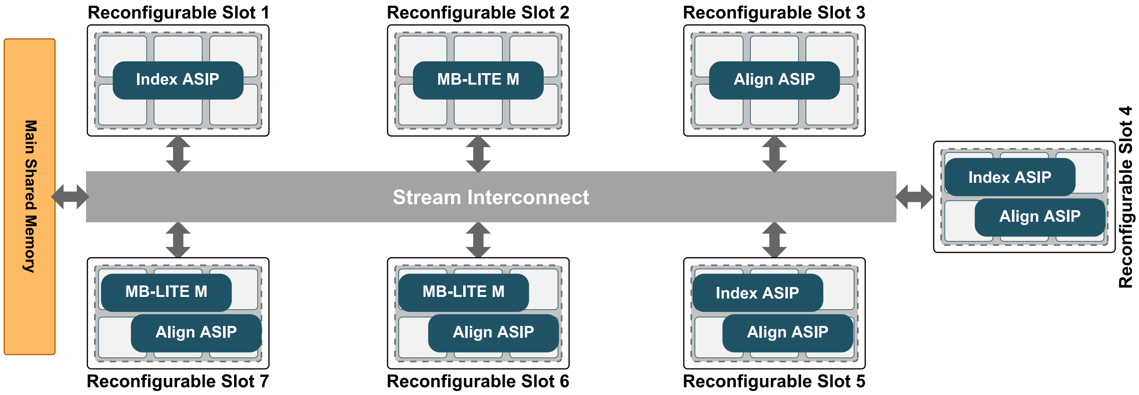


Figure 5.24: Resulting intensive optimization execution plan for the biological sequence alignment application.

cation of the three types of processing clusters, as shown in Fig. 5.23. On the other hand, for a fine-tuned intensive search, the complexity of the *Indexed Search* phase is higher and finds significantly more regions of interest, which even after an intensive *Filtering* phase, results in a lot of potential alignment sites for the *Local Alignment* phase. The solution for this scenario results in larger numbers of allocated `Index ASIPs` and `Align ASIPs` (see Fig. 5.24), allowing a more flexible balancing of the architectures by the Hypervisor, depending on the real-time throughput requirements.

5.4 Summary

This chapter presented a study on the implementation of reconfigurable processing infrastructures. It was particularly devised to validate the viability of partial reconfiguration to efficiently adapt an accelerator's processing architecture to a running application's requirements and system runtime constraints. To do so, a reconfigurable many-core accelerator was designed and implemented on an FPGA device, also providing a fully functional platform for future research development and evaluation.

The accelerator is managed by a Hypervisor mechanism that is responsible for monitoring the execution of an application to learn its characteristics and trigger the reconfiguration procedure according to a set of reconfiguration policies. These policies allow the Hypervisor to balance the accelerator's performance and energy consumption (in different execution contexts).

The Hypervisor was complemented by a compile-time DSE algorithm that leverages application and architecture models, to generate the mapping and scheduling of tasks to the accelerator's processing resources and plan their reconfiguration before execution.

The conducted experiments sustained and demonstrated the advantages of using partial reconfiguration as a viable alternative to established energy saving technologies (such as power supply management and performance throttling mechanisms). Additionally, the performed study provided valuable insights into the viability of exploiting the application's characteristics to accelerate it by reconfiguring, specializing and balancing the system's processing resources, ultimately leading to improved computing efficiency.

The observed viability of the studied functionalities opens up several possibilities to deploy market-viable reconfigurable acceleration systems. In particular, the proposed compile-time application analysis tools can be further extended to support an automatic modeling of the application before the execution phase. This can be done by exploiting static analysis mechanisms (similar to those used for the memory access pattern analysis proposed in Chapter 3) to generate the application's data-flow model. On the other hand, renewed attention can be given to the topology and architecture of the accelerator's reconfigurable resources, with a particular emphasis to the adopted granularity (e.g., full-processor PEs vs. functional unit arrays) and the corresponding supporting infrastructure. Naturally, to achieve a viable computing platform, support for standard parallel programming languages must be provided while maintaining the system's adaptive capabilities as abstracted and transparent as possible.

6

Conclusions and Future Work

Contents

6.1 Conclusions 146

6.2 Main Contributions 147

6.3 Future Work 148

This chapter concludes this dissertation, by summarizing the presented work and discussing the relevant conclusions and contributions. The chapter is also complemented by a brief discussion about future work and possible research directions.

6.1 Conclusions

This dissertation proposed new adaptive computing structures with the goal of providing mechanisms to improve the performance and energy efficiency of future generations of computing systems. It is argued that conventional general-purpose processing systems and memory organizations are limited in what concerns their ability to cope with application characteristics that are inherently performance degrading (such as poor memory access spatial and temporal locality or high contention due to mass parallelization). As an alternative, it is proposed that an application can be analyzed and modeled by compile-time tools to enable the deployment of adaptive general-purpose processing and data communication architectures. Such structures allow the computing system to adapt itself to the execution context and improve its performance and energy efficiency.

Accordingly, a new compile-time memory access pattern analysis tool was proposed. It combines a data-pattern descriptor specification with source code transformations to enable data stream communication mechanisms in conventional cache-based infrastructures.

At a first stage, this allowed the deployment of new stream prefetching mechanisms embedded in conventional Graphics Processing Units (GPUs) and General Purpose Processor (GPP) systems, as an alternative to the utilization of conventional predictive prefetching schemes. However, despite the significant performance and efficiency gains that were attained, the implementation of the proposed mechanisms in off-the-shelf devices was bound by their consolidated memory access infrastructures.

Hence, by taking a step further, it was proposed a new In-Cache Stream (ICS) communication paradigm deployed on a dedicated many-core accelerator infrastructure. It is supported by a dedicated Stream Management Controller (SMC) (placed close to the main memory) that aims at improving memory access efficiency by deploying a set of memory bandwidth optimization and data reutilization and reorganization techniques, through in-time stream manipulation. Such mechanisms are supported by a dedicated communication infrastructure designed to deploy a full data streaming communication scheme in a conventional cache-coherent memory hierarchy. This allows the infrastructure to adapt itself to the application communication scheme by simultaneously and cooperatively exploiting memory-addressed and stream-based data accesses.

The same compile-time application analysis principle was also explored for efficient processing acceleration through the development of a reconfigurable many-core accelerator. It leverages application information gathered both at compile-time and at runtime to perform the partial reconfiguration of its processing cores, in turn adapting its processing infrastructure to runtime application requirements and the system execution context.

Finally, all the proposed computing techniques and mechanisms were successfully validated as alternatives to conventional processing systems and memory organizations. They showed to be capable of providing the means to cope with the throughput demands of current applications and to tackle the challenges recognized by the High-Performance Computing (HPC) community.

6.2 Main Contributions

To better highlight the techniques and mechanisms presented in this dissertation, their main features and contributions are summarized in the following paragraphs.

Compile-Time Memory Access Encoding A new compile-time memory access analysis tool was proposed to identify, describe and encode an application's memory access pattern. It relies on a new memory access pattern description specification, capable of efficiently representing data access patterns resulting from deterministic address sequences and indirect memory accesses. The proposed tool is also paired with a code transformation pass that replaces the array subscript indexation of each encoded data access with a stream reference. In the considered set of benchmarks, the combination of both approaches allowed a conversion of up to 90% of data loads to streams, resulting in up to 23% code size reductions.

Efficient Data Stream Generation A dedicated Data Stream Controller (DSC) architecture was designed to index memory access patterns encoded by the proposed descriptor specification. It implements an efficient descriptor decoding architecture that can deploy a single-cycle per address generation throughput, with minimal hardware resource requirements. Its architecture was devised to be deployed both as a stream prefetcher or as a dedicated stream controller.

Stream Prefetching in GPGPUs The designed DSC was implemented as a stream prefetcher, integrated into the memory subsystem of a real GPU device (via simulation). Despite being integrated with the L1 data cache as a typical prefetcher (potentially interfering with its operation), performance gains of up to 9x were achieved, resulting in a range of 7% up to 90% energy consumption reductions. These results showed the viability of adopting an explicit data access encoding to perform the acquisition of data before it is requested by the system's processing cores. Hence, without relying on (potentially) inaccurate prefetching prediction heuristics and monitoring delays, it was possible to mitigate the high contention that characterizes the massively-parallel memory access structure of a GPU device and improve its throughput.

Data Streaming in Modern CPUs The conceived DSC was also deployed in the memory subsystem of a GPP as a fully functional data streaming mechanism detached from the system's cache hierarchy. The devised mechanism exploits the combination of the DSC and the code transformations of the proposed compilation tool. The obtained results for this implementation showed that the proposed mechanism is able to outperform by up to 40% two of the most prominent state-of-the-art prefetchers, as a result of the provided two-fold performance improvement that directly impacts the whole system's performance.

Adaptive In-Cache Stream Communication The promising results of the DSC drove the development of a new ICS communication model integrating the efficient benefits of stream-based communication in a conventional cache-based structure. In the devised infrastructure, the generation of data streams is performed by a new memory-aware SMC structure. It relies on the DSC and is paired with a burst controller to optimize the main memory bandwidth, along with a reorder buffer to exploit data reorganization and reutilization techniques through on-the-fly stream manipulation. The combination of such features allowed the SMC to reduce the memory access latency in up to 20x. Such an efficient memory access is reflected in the overall performance increase, averaging gains of 127x and 54x, when compared to a reference setup and a state-of-the-art stride prefetcher, respectively. The implemented infrastructure also allows for significant energy savings (averaging 91%), resulting in overall energy efficiency improvements as high as 245x.

Reconfigurable Computing Acceleration A reconfigurable many-core accelerator was designed and implemented on an Field-Programmable Gate Array (FPGA) device, to exploit its partial reconfiguration technology. It is managed by a specially devised Hypervisor module that leverages application information to perform the reconfiguration of the accelerator's cores according to the application requirements and the system's energetic context. Such information is gathered both at runtime (through core execution monitoring) and at compile-time (through the deployment of a Design Space Exploration (DSE) algorithm that leverages application and system models to generate optimized execution plans). The performed experimentations concluded that the devised accelerator allows a significant reduction of both the execution time and energy consumption when compared with static homogeneous or non-homogeneous implementations with fixed numbers of cores. This is supported by the observing performance gains, and energy consumption reductions, of up to 9.5× and 10×, respectively. When combined with the compile-time DSE algorithm, further reductions of up to 54% and 45% in peak power dissipation and energy consumption are obtained, respectively, when compared to non-optimized solutions.

6.3 Future Work

Within the wide scope that characterizes the work presented in this dissertation, several potential future research directions can be considered:

- Extension of the proposed compile-time memory access pattern analysis tools to support the detection and description of memory access patterns with further levels of irregularity. In particular, it can be considered the deployment of memory access analysis at the LLVM Intermediate Representation (IR) level. Such an approach not only leads to a full abstraction of the application source code (allowing the detection of memory access sequences that are not explicitly exposed, such as pointer-chasing structures), but it also allows the integration of the proposed tools in the compiler's optimization passes (such as the vectorization or parallelization phases).
- Development of speculative stream generation mechanisms to support the prefetching of highly irregular memory access patterns (e.g., graph analysis or pointer-chasing). To do so, new speculative stream descriptors could be designed that preemptively follow multiple memory access generation conditional paths (mirroring the coding style that generates irregular data accesses) and later synchronize with the processing core's execution to discard unwanted data and select the correct stream sequence.
- Integration of the proposed ICS model with the developed reconfigurable many-core accelerator. To do so, it can be considered a new processor architecture that extends the stream-based communication paradigm to the processor itself. In particular, it can be studied the possibility of including a new stream register file, transparent to the processor's Instruction Set Architecture (ISA), effectively bringing the ICS paradigm to the register level (i.e., from L1 to L0 memory). The addition of such a transparent mechanism can allow the processor to seamlessly access data streams provided by an ICS controller, while still retaining the conventional load/store operation. With the gathered knowledge in partial reconfiguration, the processor architecture can be composed of scalable reconfigurable functional unit arrays, allowing a transparent communication of streams to reconfigurable computing units.
- Such an integration can be complemented with the development of further compile-time application analysis tools (possibly integrated with the LLVM IR optimization passes) to identify specific code regions that can be parallelized and executed in specialized reconfigurable hardware, in turn allowing a full integration of all the techniques and mechanisms proposed in this dissertation.

Bibliography

[1] H. Esmaeilzadeh, E. Blem, R. St Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in 38th International Symposium on Computer Architecture (ISCA'11). IEEE/ACM, 2011, pp. 365–376.

[2] T. Scogland, B. Subramaniam, and W.-c. Feng, "The green500 list: escapades to exascale," Computer Science-Research and Development, vol. 28, no. 2-3, pp. 109–117, 2013.

[3] O. Villa, D. R. Johnson, M. O'Connor, E. Bolotin, D. Nellans, J. Luitjens, N. Sakharnykh, P. Wang, P. Micikevicius, A. Scudiero et al., "Scaling the power wall: a path to exascale," in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE Press, 2014, pp. 830–841.

[4] R. Kumar, T. G. Mattson, G. Pokam, and R. Van Der Wijngaart, "The case for message passing on many-core chips," in Multiprocessor System-on-Chip. Springer, 2011, pp. 115–123.

[5] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Größlinger, and L.-N. Pouchet, "Polly-polyhedral optimization in llvm," in Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT), vol. 2011, 2011.

[6] S. Pop, A. Cohen, C. Bastoul, S. Girbal, G.-A. Silber, and N. Vasilache, "Graphite: Polyhedral analyses and optimizations for gcc," in Proceedings of the 2006 GCC Developers Summit. Citeseer, 2006, p. 2006.

[7] Z. Majo and T. R. Gross, "Matching memory access patterns and data placement for numa systems," in Proceedings of the Tenth International Symposium on Code Generation and Optimization. ACM, 2012, pp. 230–241.

[8] K.-A. Tran, T. E. Carlson, K. Koukos, M. Själander, V. Spiliopoulos, S. Kaxiras, and A. Jimborean, "Clairvoyance: Look-ahead compile-time scheduling," in Code Generation and Optimization (CGO), 2017 IEEE/ACM International Symposium on. IEEE, 2017, pp. 171–184.

[9] A. Venkat, M. Hall, and M. Strout, "Loop and data transformations for sparse matrix code," in ACM SIGPLAN Notices, vol. 50, no. 6. ACM, 2015, pp. 521–532.

[10] V. Kiriansky, Y. Zhang, and S. Amarasinghe, “Optimizing indirect memory references with milk,” in Parallel Architecture and Compilation Techniques (PACT), 2016 International Conference on. IEEE, 2016, pp. 299–312.

[11] M. Srivastava, A. Chandrakasan, and R. Brodersen, “Predictive system shutdown and other architectural techniques for energy efficient programmable computation,” IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 4, no. 1, pp. 42–55, March 1996.

[12] Y. Zhu and V. Reddi, “High-performance and energy-efficient mobile web browsing on big/little systems,” in 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA), Feb 2013, pp. 13–24.

[13] R. Kumar, A. Martinez, and A. Gonzalez, “Dynamic selective devectorization for efficient power gating of simd units in a hw/sw co-designed environment,” in 25th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), Oct 2013, pp. 81–88.

[14] G. Semeraro, G. Magklis, R. Balasubramonian, D. Albonesi, S. Dwarkadas, and M. Scott, “Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling,” in Eighth International Symposium on High-Performance Computer Architecture (ISCA), Feb 2002, pp. 29–40.

[15] T. S. Muthukaruppan, M. Pricopi, V. Venkataramani, T. Mitra, and S. Vishin, “Hierarchical power management for asymmetric multi-core in dark silicon era,” in Proceedings of the 50th Annual Design Automation Conference, 2013, pp. 174:1–174:9.

[16] J. Lee, N. B. Lakshminarayana, H. Kim, and R. Vuduc, “Many-thread aware prefetching mechanisms for GPGPU applications,” in 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, 2010, pp. 213–224.

[17] D. Zhang, X. Ma, and D. Chiou, “Worklist-directed prefetching,” IEEE Computer Architecture Letters, vol. PP, no. 99, pp. 1–1, 2016.

[18] Y. Guo, P. Narayanan, M. A. Bennaser, S. Chheda, and C. A. Moritz, “Energy-efficient hardware data prefetching,” IEEE Transactions on Very Large Scale Integration Systems, vol. 19, no. 2, pp. 250–263, 2011.

[19] A. Jain and C. Lin, “Linearizing irregular memory accesses for improved correlated prefetching,” in IEEE/ACM International Symposium on Microarchitecture (MICRO-46). ACM, 2013, pp. 247–259.

[20] Y. Ishii, M. Inaba, and K. Hiraki, “Access map pattern matching for high performance data cache prefetch,” Journal of Instruction-Level Parallelism, vol. 13, pp. 1–24, 2011.

[21] M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad, “An efficient temporal data prefetcher for l1 caches,” IEEE Computer Architecture Letters, vol. PP, no. 99, pp. 1–1, 2017.

[22] P. Michaud, “Best-offset hardware prefetching,” in 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE, 2016, pp. 469–480.

[23] S. H. Pugsley, Z. Chishti, C. Wilkerson, P.-f. Chuang, R. L. Scott, A. Jaleel, S.-L. Lu, K. Chow, and R. Balasubramonian, “Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers,” in High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on. IEEE, 2014, pp. 626–637.

[24] S. Kondguli and M. Huang, “Division of labor: A more effective approach to prefetching,” in 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA), June 2018, pp. 83–95.

[25] G. Venkatesh, J. Sampson, N. Goulding-Hotta, S. K. Venkata, M. B. Taylor, and S. Swanson, “Qscores: Trading dark silicon for scalable energy efficiency with quasi-specific cores,” in 44th International Symposium on Microarchitecture (ISCA’11). IEEE/ACM, 2011, pp. 163–174.

[26] S. Paiágua, F. Pratas, P. Tomás, N. Roma, and R. Chaves, “Hotstream: Efficient data streaming of complex patterns to multiple accelerating kernels,” in 2013 25th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD). IEEE, 2013, pp. 17–24.

[27] Y. Wang, X. Zhou, L. Wang, J. Yan, W. Luk, C. Peng, and J. Tong, “Spread: A streaming-based partially reconfigurable architecture and programming model,” IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 21, no. 12, pp. 2179–2192, Dec 2013.

[28] N. Neves, N. Sebastião, D. Matos, P. Tomás, P. Flores, and N. Roma, “Multicore SIMD ASIP for next-generation sequencing and alignment biochip platforms,” IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 23, no. 7, pp. 1287–1300, 2015.

[29] GCC, the GNU Compiler Collection, GNU Project, October 2013, http://gcc.gnu.org/.

[30] C. Lattner and V. Adve, “Llvm: A compilation framework for lifelong program analysis & transformation,” in Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization. IEEE Computer Society, 2004, p. 75.

[31] T. F. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos, “Practical off-chip meta-data for temporal memory streaming,” in 2009 IEEE 15th International Symposium on High Performance Computer Architecture. IEEE, 2009, pp. 79–90.

[32] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti, "Efficiently prefetching complex address patterns," in Proceedings of the 48th International Symposium on Microarchitecture. ACM, 2015, pp. 141–152.

[33] K. J. Nesbit and J. E. Smith, "Data cache prefetching using a global history buffer," in Software, IEE Proceedings-, Feb 2004, pp. 96–96.

[34] N. Neves, P. Tomás, and N. Roma, "Efficient data-stream management for shared-memory many-core systems," in 25th International Conference on Field Programmable Logic and Applications (FPL). IEEE, 2015, pp. 508–515.

[35] N. Neves, P. Tomás, and N. Roma, "Adaptive in-cache streaming for efficient data management," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 25, no. 7, pp. 2130–2143, March 2017.

[36] N. Neves, P. Tomás, and N. Roma, "Stream data prefetcher for the gpu memory interface," The Journal of Supercomputing, vol. 74, no. 76, pp. 2314–2328, June 2018.

[37] N. Neves, A. Mussio, F. Gonçalves, P. Tomás, and N. Roma, "In-cache streaming: Morphable infrastructure for many-core processing systems," in Euro-Par 2016: Parallel Processing Workshops. Springer International Publishing, 2017, pp. 775–787.

[38] N. Neves, H. Mendes, R. J. Chaves, P. Tomás, and N. Roma, "Morphable hundred-core heterogeneous architecture for energy-aware computation," IET Computers & Digital Techniques, vol. 9, no. 1, pp. 49–62, 2015.

[39] N. Neves, P. Tomás, and N. Roma, "Host to accelerator interfacing framework for high-throughput co-processing systems," in XI Jornadas sobre Sistemas Reconfiguráveis (REC), 2015, pp. 31–38.

[40] N. Neves, R. Neves, N. Horta, P. Tomás, and N. Roma, "Multi-objective kernel mapping and scheduling for morphable many-core architectures," Expert Systems with Applications, vol. 45, pp. 385–399, 2016.

[41] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim et al., "A cloud-scale acceleration architecture," in The 49th Annual IEEE/ACM International Symposium on Microarchitecture. IEEE Press, 2016, p. 7.

[42] J. R. G. Ordaz and D. Koch, "A soft dual-processor system with a partially run-time reconfigurable shared 128-bit simd engine," in 2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP), July 2018, pp. 1–8.

[43] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, "Plasticine: A reconfigurable accelerator for parallel patterns," IEEE Micro, vol. 38, no. 3, pp. 20–31, May 2018.

[44] W. Hussain, R. Airoldi, H. Hoffmann, T. Ahonen, and J. Nurmi, "Harp2: an x-scale reconfigurable accelerator-rich platform for massively-parallel signal processing algorithms," Journal of Signal Processing Systems, vol. 85, no. 3, pp. 341–353, 2016.

[45] T. Chau, X. Niu, A. Eele, W. Luk, P. Cheung, and J. Maciejowski, "Heterogeneous reconfigurable system for adaptive particle filters in real-time applications," in Reconfigurable Computing: Architectures, Tools and Applications, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, vol. 7806, pp. 1–12.

[46] P. Petrica, A. M. Izraelevitz, D. H. Albonesi, and C. A. Shoemaker, "Flicker: a dynamically adaptive architecture for power limited multicore systems," in ACM SIGARCH Computer Architecture News, vol. 41, no. 3. ACM, 2013, pp. 13–23.

[47] M. Modarressi, A. Tavakkol, and H. Sarbazi-Azad, "Application-aware topology reconfiguration for on-chip networks," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 19, no. 11, pp. 2010–2022, 2011.

[48] R. Pal, K. Paul, and S. Prasad, "Rekonf: A reconfigurable adaptive manycore architecture," in IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA), 2012, pp. 182–191.

[49] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman et al., "Serving dnns in real time at datacenter scale with project brainwave," IEEE Micro, vol. 38, no. 2, pp. 8–20, 2018.

[50] K. T. Sundararajan, T. M. Jones, and N. P. Topham, "The smart cache: An energy-efficient cache architecture through dynamic adaptation," International Journal of Parallel Programming, vol. 41, no. 2, pp. 305–330, 2013.

[51] Y.-T. Chen, J. Cong, H. Huang, B. Liu, C. Liu, M. Potkonjak, and G. Reinman, "Dynamically reconfigurable hybrid cache: An energy-efficient last-level cache design," in Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012. IEEE, 2012, pp. 45–50.

[52] G. Kalokerinos, V. Papaefstathiou, G. Nikiforos, S. Kavadias, M. Katevenis, D. Pnevmatikatos, and X. Yang, "Fpga implementation of a configurable cache/scratchpad memory with virtualized user-level rdma capability," in International Symposium on Systems, Architectures, Modeling, and Simulation, 2009 (SAMOS'09). IEEE, 2009, pp. 149–156.

[53] E. Ebrahimi, O. Mutlu, and Y. N. Patt, “Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems,” in 2009 IEEE 15th International Symposium on High Performance Computer Architecture. IEEE, 2009, pp. 7–17.

[54] T. Hussain, O. Palomar, O. Unsal, A. Cristal, E. Ayguadé, and M. Valero, “Advanced Pattern based Memory Controller for FPGA based HPC applications,” in 2014 International Conference on High Performance Computing & Simulation (HPCS). IEEE, 2014, pp. 287–294.

[55] L. Peled, S. Mannor, U. Weiser, and Y. Etsion, “Semantic locality and context-based prefetching using reinforcement learning,” in Proceedings of the 42Nd Annual International Symposium on Computer Architecture, ser. ISCA ’15. New York, NY, USA: ACM, 2015, pp. 285–297.

[56] J. Park and P. Diniz, “Synthesis of pipelined memory access controllers for streamed data applications on fpga-based computing engines,” in Proceedings of the 14th international symposium on Systems synthesis. ACM, 2001, pp. 221–226.

[57] J. Park and P. C. Diniz, “Data reorganization and prefetching of pointer-based data structures,” IEEE Design and Test of Computers, vol. 28, no. 4, pp. 38–47, 2011.

[58] D. Koufaty, D. Reddy, and S. Hahn, “Bias scheduling in heterogeneous multi-core architectures,” in Proceedings of the 5th European Conference on Computer Systems, ser. EuroSys ’10, 2010, pp. 125–138.

[59] J. Cong and B. Yuan, “Energy-efficient scheduling on heterogeneous multi-core architectures,” in Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design, 2012, pp. 345–350.

[60] N. Chitlur, G. Srinivasa, S. Hahn, P. Gupta, D. Reddy, D. Koufaty, P. Brett, A. Prabhakaran, L. Zhao, N. Ijih, S. Subhaschandra, S. Grover, X. Jiang, and R. Iyer, “Quickia: Exploring heterogeneous architectures on real prototypes,” in IEEE 18th International Symposium on High Performance Computer Architecture (HPCA), Feb 2012, pp. 1–8.

[61] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, “Scheduling Heterogeneous Multi-cores Through Performance Impact Estimation (PIE),” in Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA’2012), 2012, pp. 213–224.

[62] S. Akram, A. Papakonstantinou, R. Kumar, and D. Chen, “A workload-adaptive and reconfigurable bus architecture for multicore processors,” International Journal of Reconfigurable Computing, vol. 2010, p. 2, 2010.

[63] W. Wang, P. Mishra, and S. Ranka, “Dynamic cache reconfiguration and partitioning for energy optimization in real-time multi-core systems,” in 48th ACM/EDAC/IEEE Design Automation Conference (DAC), June 2011, pp. 948–953.

[64] R. Rodrigues, A. Annamalai, I. Koren, S. Kundu, and O. Khan, “Performance per watt benefits of dynamic core morphing in asymmetric multicores,” in 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT), Oct 2011, pp. 121–130.

[65] R. Rodrigues, A. Annamalai, I. Koren, and S. Kundu, “Improving performance per watt of asymmetric multi-core processors via online program phase classification and adaptive core morphing,” ACM Transactions on Design Automation of Electronic Systems, vol. 18, no. 1, pp. 5:1–5:23, Jan. 2013.

[66] R. Chaves, G. Kuzmanov, and L. Sousa, “On-the-fly attestation of reconfigurable hardware,” in International Conference on Field Programmable Logic and Applications (FPL). IEEE, 2008, pp. 71–76.

[67] H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. Chaves Filho, “Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications,” IEEE transactions on computers, vol. 49, no. 5, pp. 465–481, 2000.

[68] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, “Adres: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix,” in International Conference on Field Programmable Logic and Applications. Springer, 2003, pp. 61–70.

[69] V. Baumgarte, G. Ehlers, F. May, A. Nückel, M. Vorbach, and M. Weinhardt, “Pact xpp—a self-reconfigurable data processing architecture,” The Journal of Supercomputing, vol. 26, no. 2, pp. 167–184, 2003.

[70] P. Garcia and K. Compton, “Kernel sharing on reconfigurable multiprocessor systems,” in International Conference on ICECE Technology (FPT). IEEE, 2008, pp. 225–232.

[71] E. Caspi, M. Chu, R. Huang, J. Yeh, J. Wawrzynek, and A. DeHon, “Stream computations organized for reconfigurable execution (SCORE),” in Field-Programmable Logic and Applications: The Roadmap to Reconfigurable Computing, 2000, pp. 605–614.

[72] Z. Chen, R. N. Pittman, and A. Forin, “Combining multicore and reconfigurable instruction set extensions,” in Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA), 2010, pp. 33–36.

[73] M. A. Watkins and D. H. Albonesi, “Remap: A reconfigurable heterogeneous multicore architecture,” in 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, 2010, pp. 497–508.

[74] M. G. Lorenz, L. Mengibar, M. G. Valderas, and L. Entrena, “Power consumption reduction through dynamic reconfiguration,” in Field Programmable Logic and Application. Springer, 2004, pp. 751–760.

[75] C. Kim, M. Chung, Y. Cho, M. Konijnenburg, S. Ryu, and J. Kim, “Ulp-srp: Ultra low power samsung reconfigurable processor for biomedical applications,” in 2012 International Conference on Field-Programmable Technology, Dec 2012, pp. 329–334.

[76] L. H. Crockett, R. A. Elliot, M. A. Enderwitz, and R. W. Stewart, The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc. Strathclyde Academic Media, 2014.

[77] Xilinx, “Vivado Design Suite User Guide,” Xilinx, Tech. Rep. UG973, 2018.

[78] Altera, “Implementing FPGA Design with the OpenCL Standard,” Altera, Tech. Rep. WP-01173-3.0, 2013.

[79] M. Technologies, “MaxCompiler - White Paper,” Maxeler Technologies, Tech. Rep., 2011.

[80] M. I. Daoud and N. Kharma, “A hybrid heuristic–genetic algorithm for task scheduling in heterogeneous processor networks,” Journal of Parallel and Distributed Computing, vol. 71, no. 11, pp. 1518–1531, 2011.

[81] M. Camelo, Y. Donoso, and H. Castro, “MAGS–An approach using multi-objective evolutionary algorithms for grid task scheduling,” International Journal of Applied Mathematics and Informatics, vol. 5, no. 2, 2011.

[82] H. F. Sheikh and I. Ahmad, “Dynamic task graph scheduling on multicore processors for performance, energy, and temperature optimization,” in Green Computing Conference (IGCC), 2013 International. IEEE, 2013, pp. 1–6.

[83] T. Blickle, J. Teich, and L. Thiele, “System-level synthesis using evolutionary algorithms,” Design Automation for Embedded Systems, vol. 3, no. 1, pp. 23–58, 1998.

[84] V. Krishnan and S. Katkoori, “A genetic algorithm for the design space exploration of datapaths during high-level synthesis,” IEEE Transactions on Evolutionary Computation, vol. 10, no. 3, pp. 213–229, 2006.

[85] M. Holzer, B. Knerr, and M. Rupp, “Design space exploration with evolutionary multi-objective optimisation,” in International Symposium on Industrial Embedded Systems, 2007. SIES’07. IEEE, 2007, pp. 126–133.

[86] G. Palermo, C. Silvano, and V. Zaccaria, “ReSPIR: a response surface-based Pareto iterative refinement for application-specific design space exploration,” IEEE Transactions on

Computer-Aided Design of Integrated Circuits and Systems, vol. 28, no. 12, pp. 1816–1829, 2009.

[87] G. Mariani, A. Brankovic, G. Palermo, J. Jovic, V. Zaccaria, and C. Silvano, “A correlation-based design space exploration methodology for multi-processor Systems-on-Chip,” in Proceedings of the 47th Design Automation Conference. ACM, 2010, pp. 120–125.

[88] C. Erbas, S. C. Erbas, and A. D. Pimentel, “A multiobjective optimization model for exploring multiprocessor mappings of process networks,” in Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis. ACM, 2003, pp. 182–187.

[89] M. Glaß, M. Lukasiewycz, R. Wanka, C. Haubelt, and J. Teich, “Multi-objective routing and topology optimization in networked embedded systems,” in International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, 2008. SAMOS 2008. IEEE, 2008, pp. 74–81.

[90] B. Miramond and J.-M. Delosme, “Design space exploration for dynamically reconfigurable architectures,” in Proceedings of the conference on Design, Automation and Test in Europe-Volume 1. IEEE Computer Society, 2005, pp. 366–371.

[91] R. Czarnecki and S. Deniziak, “Co-synthesis of dynamically reconfigurable SOPCs specified by conditional task graphs.” Open Cybernetics & Systemics Journal, vol. 2, 2008.

[92] K. Deb, Multi-Objective optimization using evolutionary algorithms, ser. Wiley paperback series. Wiley, 2008.

[93] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, “A fast and elitist multiobjective genetic algorithm: NSGA-II,” IEEE Transactions on Evolutionary Computation, vol. 6, no. 2, pp. 182–197, Apr 2002.

[94] L. Shang and N. K. Jha, “Hardware-software co-synthesis of low power real-time distributed embedded systems with dynamically reconfigurable FPGAs,” in Proceedings of the 2002 Asia and South Pacific Design Automation Conference. IEEE Computer Society, 2002, p. 345.

[95] S. Wildermann, F. Reimann, D. Ziener, and J. Teich, “Symbolic design space exploration for multi-mode reconfigurable systems,” in Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis. ACM, 2011, pp. 129–138.

[96] N. B. Grigore and D. Koch, “Placing partially reconfigurable stream processing applications on fpgas,” in 2015 25th International Conference on Field Programmable Logic and Applications (FPL), Sept 2015, pp. 1–4.

[97] A. A. El-Moursy and F. N. Sibai, "V-set cache: An efficient adaptive shared cache for multi-core processors," Journal of Circuits, Systems, and Computers, vol. 23, no. 07, p. 1450095, 2014.

[98] S. Ghosh, M. Martonosi et al., "Cache miss equations: An analytical representation of cache misses," in ACM International Conference on Supercomputing. ACM Press, 1997, pp. 317–324.

[99] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to GPU codes," in Innovative Parallel Computing (InPar), 2012. IEEE, 2012, pp. 1–10.

[100] J. Dongarra, M. A. Heroux, and P. Luszczek, "Hpcg benchmark: a new metric for ranking high performance computing systems," Knoxville, Tennessee, 2015.

[101] "Clang. LibTooling," http://http://clang.llvm.org/docs/LibTooling.html, accessed: 30-July-2018.

[102] "LogiCORE IP AXI DMA v6.03a," Xilinx, Tech. Rep. PG021, 2012.

[103] G. Wallace, "The JPEG still picture compression standard," Consumer Electronics, IEEE Transactions on, vol. 38, no. 1, pp. xviii–xxxiv, 1992.

[104] S. Zhu and K.-K. Ma, "A new diamond search algorithm for fast block-matching motion estimation," IEEE Transactions on Image Processing, vol. 9, no. 2, pp. 287–290, 2000.

[105] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in IEEE International Symposium on Workload Characterization, 2009. IISWC 2009. IEEE, 2009, pp. 44–54.

[106] I. Corporation, "Intel 64 and ia-32 architectures optimization reference manual," Intel Corporation, Tech. Rep. 248966-040, April 2018.

[107] NVIDIA, "NVIDIA's Next Generation CUDA^{TM} Compute Architecture: Fermi^{TM}," NVIDIA, Santa Clara, Calif, USA, 2009.

[108] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in IEEE International Symposium on Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE, 2009, pp. 163–174.

[109] NVIDIA, "NVIDIA GP100 Pascal Architecture," White paper. [Online]. Available: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf, 2016.

[110] S. Lee, K. Kim, G. Koo, H. Jeon, W. W. Ro, and M. Annavaram, "Warped-compression: enabling power efficient GPUs through register compression," in 42nd Intl Symposium on Computer Architecture. ACM, 2015, pp. 502–514.

[111] M. Stephenson, S. K. S. Hari, Y. Lee, E. Ebrahimi, D. R. Johnson, D. Nellans, M. O'Connor, and S. W. Keckler, "Flexible software profiling of GPU architectures," in 42nd Intl Symp on Computer Architecture. ACM, 2015, pp. 185–197.

[112] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Orchestrated scheduling and prefetching for GPGPUs," ACM SIGARCH Computer Architecture News, vol. 41, no. 3, pp. 332–343, 2013.

[113] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, "Boosting mobile GPU performance with a decoupled access/execute fragment processor," in ACM SIGARCH Computer Architecture News, vol. 40, no. 3. IEEE Computer Society, 2012, pp. 84–93.

[114] A. Sethia, G. Dasika, M. Samadi, and S. Mahlke, "APOGEE: Adaptive prefetching on GPUs for energy efficiency," in Proceedings of the 22nd international conference on Parallel architectures and compilation techniques. IEEE Press, 2013, pp. 73–82.

[115] Y. Torres, A. Gonzalez-Escribano, and D. R. Llanos, "Understanding the impact of CUDA tuning techniques for Fermi," in 2011 International Conference on High Performance Computing and Simulation (HPCS). IEEE, 2011, pp. 631–639.

[116] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "GPUWattch: enabling energy optimizations in GPGPUs," ACM SIGARCH Computer Architecture News, vol. 41, no. 3, pp. 487–498, 2013.

[117] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," SIGARCH Comput. Archit. News, vol. 39, no. 2, pp. 1–7, 2011.

[118] AMBA® 4 AXI4-Stream Protocol, v1.0, ARM, Ltd., March 2010, http://infocenter.arm.com.

[119] "TN-41-01: Calculating Memory System Power for DDR3," Micron Technology, Inc., Tech. Rep., 2007.

[120] T. Kranenburg and R. van Leuken, "MB-LITE: A robust, light-weight soft-core implementation of the MicroBlaze architecture," Design, Automation and Test in Europe Conference and Exhibition (DATE), pp. 997–1000, March 2010.

[121] AMBA® AXI and ACE Protocol Specification, Issue E, ARM, Ltd., February 2013, http://infocenter.arm.com.

[122] M. F. Cloutier, C. Paradis, and V. M. Weaver, "Design and analysis of a 32-bit embedded high-performance cluster optimized for energy and performance," in Proceedings of the 1st International Workshop on Hardware-Software Co-Design for High Performance Computing. IEEE Press, 2014, pp. 1–8.

[123] M. Farrar, "Striped Smith-Waterman speeds database searches six times over other SIMD implementations," Bioinformatics, vol. 23, no. 2, p. 156, 2007.

[124] V. Bandishti, I. Pananilath, and U. Bondhugula, "Tiling stencil computations to maximize parallelism," in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. IEEE Computer Society Press, 2012, p. 40.

[125] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," Proceedings of the IEEE, vol. 86, no. 11, pp. 2278–2324, 1998.

[126] MicroBlaze Processor Reference Guide, v14.3, Xilinx Inc., October 2012.

[127] LogiCORE IP Floating-Point Operator v5.0, Xilinx Inc., March 2011, www.xilinx.com/support/documentation/ip_documentation/.

[128] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprette, "System design using Khan process networks: the Compaan/Laura approach," in Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, 2004, vol. 1. IEEE, 2004, pp. 340–345.

[129] L. Huang, G. Sethuraman, and B. Chapman, "Parallel data flow analysis for OpenMP programs," in A Practical Programming Model for the Multi-Core Era. Springer, 2008, pp. 138–142.

[130] K. De Jong and W. Spears, "A formal analysis of the role of multi-point crossover in genetic algorithms," Annals of Mathematics and Artificial Intelligence, vol. 5, no. 1, pp. 1–26, 1992.

[131] C. Pilato, D. Loiacono, F. Ferrandi, P. L. Lanzi, and D. Sciuto, "High-level synthesis with multi-objective genetic algorithm: A comparative encoding analysis," in IEEE Congress on Evolutionary Computation, 2008. CEC 2008 (IEEE World Congress on Computational Intelligence). IEEE, 2008, pp. 3334–3341.

[132] W. R. Pearson and D. J. Lipman, "Improved tools for biological sequence comparison," Proceedings of the National Academy of Sciences, vol. 85, no. 8, p. 2444, 1988.

[133] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman, "Basic Local Alignment Search Tool (BLAST)," Journal of Molecular Biology, vol. 215, pp. 403–410, 1990.

[134] N. Sebastião, T. Dias, N. Roma, and P. Flores, "Optimized ASIP architecture for compressed BWT-Indexed Search in bioinformatics applications," in High Performance Computing and Simulation (HPCS), 2014 International Conference on. IEEE, 2014.

Publications

Publications in Scientific Journals

- N. Neves, P. Tomás, and N. Roma, “Stream data prefetcher for the gpu memory interface,” The Journal of Supercomputing, vol. 74, no. 76, pp. 2314–2328, June 2018
- N. Neves, P. Tomás, and N. Roma, “Adaptive in-cache streaming for efficient data management,” IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 25, no. 7, pp. 2130–2143, March 2017
- N. Neves, R. Neves, N. Horta, P. Tomás, and N. Roma, “Multi-objective kernel mapping and scheduling for morphable many-core architectures,” Expert Systems with Applications, vol. 45, pp. 385–399, 2016
- N. Neves, H. Mendes, R. J. Chaves, P. Tomás, and N. Roma, “Morphable hundred-core heterogeneous architecture for energy-aware computation,” IET Computers & Digital Techniques, vol. 9, no. 1, pp. 49–62, 2015

Publications in International Conferences and Workshops

- N. Neves, A. Mussio, F. Gonçalves, P. Tomás, and N. Roma, “In-cache streaming: Morphable infrastructure for many-core processing systems,” in Euro-Par 2016: Parallel Processing Workshops. Springer International Publishing, 2017, pp. 775–787
- N. Neves, P. Tomás, and N. Roma, “Efficient data-stream management for shared-memory many-core systems,” in 25th International Conference on Field Programmable Logic and Applications (FPL). IEEE, 2015, pp. 508–515

Publications in National Conferences

- N. Neves, P. Tomás, and N. Roma, “Host to accelerator interfacing framework for high-throughput co-processing systems,” in XI Jornadas sobre Sistemas Reconfiguráveis (REC), 2015, pp. 31–38