UNIVERSIDADE TÉCNICA DE LISBOA

INSTITUTO SUPERIOR TÉCNICO

Mapping Runtime-Detected Loops from Microprocessors to Reconfigurable Processing Units

João Carlos Viegas Martins Bispo

Supervisor: Doctor João Manuel Paiva Cardoso

Co-Supervisor: Doctor José Carlos Monteiro

Thesis approved in public session to obtain the PhD Degree in

Information Systems and Computer Engineering

Jury Final Classification

Pass with Merit

Jury

Chairperson: Chairman of the IST Scientific Board

Members of the Committee:

Doctor JOÃO MIGUEL LOBO FERNANDES

Doctor JOÃO MANUEL PAIVA CARDOSO

Doctor JOSÉ CARLOS ALVES PEREIRA MONTEIRO

Doctor CHRISTIAN PLESSL

Doctor NUNO FILIPE VALENTIM ROMA

2012

UNIVERSIDADE TÉCNICA DE LISBOA

INSTITUTO SUPERIOR TÉCNICO

Mapping Runtime-Detected Loops from Microprocessors to Reconfigurable Processing Units

João Carlos Viegas Martins Bispo

Supervisor: Doctor João Manuel Paiva Cardoso

Co-Supervisor: Doctor José Carlos Monteiro

Thesis approved in public session to obtain the PhD Degree in

Information Systems and Computer Engineering

Jury Final Classification

Pass with Merit

Jury

Chairperson: Chairman of the IST Scientific Board

Members of the Committee:

Doctor JOÃO MIGUEL LOBO FERNANDES, Professor Catedrático da Escola de Engenharia, da Universidade do Minho;

Doctor JOÃO MANUEL PAIVA CARDOSO, Professor Associado da Faculdade de Engenharia da Universidade do Porto;

Doctor JOSÉ CARLOS ALVES PEREIRA MONTEIRO, Professor Associado do Instituto Superior Técnico, da Universidade Técnica de Lisboa;

Doctor CHRISTIAN PLESSL, Professor Auxiliar da Universidade de Paderborn, Alemanha;

Doctor NUNO FILIPE VALENTIM ROMA, Professor Auxiliar do Instituto Superior Técnico, da Universidade Técnica de Lisboa.

Funding Institutions

FCT – Fundação para a Ciência e a Tecnologia

2012

Resumo

Os sistemas computacionais baseados em processadores (GPPs) podem ser estendidos com co-processadores, unidades de processamento reconfiguráveis – RPUs, de modo a melhorar características relacionadas com o desempenho (ex.: tempo de execução, consumo de energia).

Técnicas tradicionais de particionamento hardware/software permitem-nos atingir esse objectivo. No entanto, é comum o processo ser moroso, necessitar de conhecimentos não-triviais sobre projecto de hardware digital, e o resultado final ficar muito dependente de aspectos específicos da arquitectura alvo, dificultando a portabilidade da solução para outros sistemas, mesmo que façam parte da mesma família de dispositivos.

Pretende-se com esta tese propor técnicas inovadoras que permitam o particionamento dinâmico de aplicações, ao nível da representação binária. O método aborda a migração automática de código em tempo de execução, do processador para o co-processador. A migração é baseada no *Megablock*, um novo tipo de *loop*, criado tendo em mente as características do particionamento dinâmico.

Neste trabalho são apresentadas técnicas e algoritmos que permitem a detecção, identificação, implementação e melhoramento de *Megablocks*, assim como um estudo aprofundado do uso do *Megablock* como unidade de detecção, sobre um conjunto abrangente de aplicações de referência.

As técnicas propostas para o melhoramento do desempenho incluem o desenrolamento de *loops* internos e o *pipelining* de Megablocks. Experiências que consideram estas técnicas e realizadas sobre um conjunto de 61 casos de estudo de referência revelam uma *aceleração* média de 5,6× (de 0,2× até 32×). Estes valores de aceleração consideram a execução completa dos casos de estudo e incluem os custos de comunicação entre o GPP e o RPU.

Abstract

Typical embedded computing systems based on general purpose processors (GPPs) can be extended with coprocessors, such as Reconfigurable Processing Units – RPUs, to improve performance characteristics such as execution time and/or energy consumption. A common step needed for mapping computations to these systems is the use of traditional hardware/software co-design. However, this step is usually time-consuming, non-trivial knowledge about digital system design is required, and the resultant partitioning is typically tied to the system architecture being considered. This prevents the portability of hardware/solutions, as well as performance portability between different embedded computing devices.

This thesis proposes novel techniques for dynamically partitioning applications at the binary level. The approach addresses the automatic migration of computations during runtime, from a GPP to an RPU acting as its coprocessor. The proposed techniques focus on the identification and mapping of a novel kind of loop, named Megablock, to an RPU. The Megablock was designed to be identified during runtime and to be a bridge between the sequential code of the GPP and the configuration of an RPU. The work presented shows methods and algorithms for the detection, identification, implementation, and optimization of Megablocks, as well as an extensive study of the impact of using the Megablock as a detection unit over a comprehensive set of benchmarks.

The proposed techniques for optimization of Megablocks include unrolling of inner loops and pipelining of Megablocks. Experiments considering a coarse-grained reconfigurable array as RPU, coupled to a soft-core microprocessor and using the techniques proposed in this thesis, reveal average overall execution speedups, including all communication overheads, of 5.6× (from 0.2× to 32×) over the software execution, when considering a set of 61 integer benchmarks.

Keywords: Dynamic Partitioning, Reconfigurable Computing, Loop Pipelining, Heterogeneous Architectures, Runtime Reconfiguration, Binary Translation, FPGA, Instruction Trace, Megablock, Embedded Systems

Acknowledgments

First and foremost, I would like to thank my advisor, João Manuel Paiva Cardoso, for taking me as his student, for all the time he spent around my work and for allowing me to do most of the work far away from his lab. I would also like to thank my co-advisor, Professor José Carlos Monteiro, for having accepted to co-supervise my work.

Also very important to this thesis was Nuno Paulino, for implementing in a system prototype some of the ideas of this thesis. The many emails where we exchanged ideas were invaluable; João Canas Ferreira, Nuno's advisor, for believing in the work we were doing; Jani Negrier for sharing the hardships of the PhD; Ricardo Jeremias, Tânia Lopes and Ana Pereira, for the continuous support during the thesis, and in particular the madness that is the final sprint; my parents, Dulce Bispo e Carlos Viegas, for the amazing support they have always given, and to my sister, Sofia Bispo, who has always been there.

However, this work would not have been possible without the support of the many people I have met during the several years that took this PhD, and they also deserve a mention. I want to thank Adriano Sanches, which took the PhD at the same time as me and under the same advisor, for the events we have shared; my colleagues from Oporto lab, Ali Azarian, Ricardo Nobre, Tiago Carvalho; André C. Santos from Lisbon; Carlo Galuzzi from Delft, for providing feedback on the mathematical formalisms of the Megablock; Aldric Negrier, for lending me "his" lab in Algarve; Ana Moreira, for helping me getting on track on one of those difficult PhD moments; Ricardo Avó, for keeping me updated.

I want to thank the many interesting people I have met in conferences, which made the experience much more interesting and worthwhile, such as (but not limited to) Bryan Olivier, Christian Plessl, Christian de Schryver, Diana Goehringer, Dirk Koch, Eduardo Marques, Eduardo de la Torre, Glen Gibb, Jing Yan, Juan C. Peña, Kazuei Hironaka, Lee Ping, Michael Hübner, Nabela Koob, Nelson Blanco, Ricardo Menotti, Ricardo dos Santos Ferreira, Rui Policarpo, Viktor Prasanna, Yale Patt, and many others.

I thank the support given by Fundação para a Ciência e Tecnologia (FCT), which provided a doctorate scholarship (SFRH /BD/36735/2007) for the duration of the PhD.

Table of Contents

Resumo iii

Abstract v

Acknowledgments vii

Table of Contents ix

Index of Figures xiii

Index of Tables xvii

Glossary of Terms xix

1 Introduction 1

1.1 Hardware/Software Co-Design 1

1.2 Dynamic Partitioning 2

1.3 Thesis Statement and Main Contributions 4

1.4 Organization 4

2 Background 7

2.1 General Purpose Processors and Execution Flow 7

2.2 Data Hazards 8

2.3 Coprocessors 8

2.4 Coprocessor Tradeoffs 9

2.5 Reconfigurable Processing Units 10

2.5.1 FPGAs 12

2.5.2 CGRAs 13

2.6 Dynamic Compilation 14

2.7 Summary 17

3 Related Work 19

3.1 Binary Translation 19
3.2 RPU Architectures 20
3.3 Dynamic Partitioning Approaches 21
3.3.1 WARP 21
3.3.2 CCA 27
3.3.3 DIM 32
3.3.4 Overview 35
3.4 Summary 38
4 The Megablock 39
4.1 Motivation 39
4.2 Megablock Definition 42
4.3 Megablock Detection 44
4.4 Megablock Intermediate Representation 47
4.5 Adapting Source Code to Megablock Detection 50
4.5.1 General Definition of the Transformations 50
4.5.2 C Transformations Targeting the MicroBlaze Processor 51
4.6 Summary 53
5 Transforming and Implementing Megablocks 55
5.1 Graph Transformations 55
5.1.1 Mapping MicroBlaze Assembly to Graph IR 55
5.1.2 Constant Folding and Propagation 57
5.1.3 Identity Simplifications 58
5.1.4 Multiplication to Multiplexer 58
5.2 Hardware Module for Megablock Detection 58
5.3 Megablock Translation using the Graph IR 61
5.4 Megablock Identification 65
5.5 Architectures for Implementing Megablocks 67

5.5.1 General 2D CGRA 69
5.5.2 Specialized Array (SAr) 71
5.5.3 Specialized Reconfigurable Array (SRA) 72
5.5.4 Folded CGRA (1D CGRA) 73
5.6 Megablock Pipelining 74
5.6.1 Inter-Iteration Dependencies 75
5.6.2 Architecture for Pipelined Megablocks 78
5.6.3 Megablock Pipelining Algorithm 83
5.6.4 Hardware Support for Megablock Pipelining 86
5.7 Summary 88
6 Experimental Results 89
6.1 Experimental Setup 89
6.2 Megablock Coverage 92
6.3 Megablock Mapping 97
6.3.1 Baseline Results 98
6.3.2 If-Conversion 107
6.3.3 Graph Transformations 113
6.4 Hardware Module for Megablock Detection 116
6.5 Megablock Pipelining 118
6.6 Application Examples 127
6.6.1 3D Path Planning Application 127
6.6.2 Dynamic Partitioning on an Embedded Processor – fir 128
6.7 Summary 132
7 Conclusions 135
7.1 Future Work 137
8 References 141
Appendix A – SRA Implementation 151

Appendix B – Additional Results 157
B-1 Baseline Geometric Means 157
B-2 If-Conversion Geometric Means 158
B-3 Pipelining (Sequential Schedule) Geometric Means 159
B-4 Pipelining (Overlapping Schedule) Geometric Means 160
Appendix C – Tools 163
About the Author 167
Index 169

Index of Figures

Figure 1.1. Block diagram of a typical target system which includes a RPU coprocessor acting as an accelerator of the GPP. 2

Figure 2.1. Example trade-offs when using a coprocessor. 10

Figure 2.2. Possible two-dimensional structure for a reconfigurable fabric (source: [36]). FU identifies Functional Units, MEM identifies local memories, and IOB identifies Input/Output blocks. 11

Figure 2.3. Types of RPU coupling with respect to the host system. 12

Figure 2.4. Dynamic Hardware-Software Partitioning problem formulation. 16

Figure 3.1. Block Diagram for the WARP Processor (source: [13]). 22

Figure 3.2. Block Diagram for the W-FPGA (source: [13]). 23

Figure 3.3. Binary to Hardware Translation Flow (source: [13]). 24

Figure 3.4. CCA-Enabled Processor Block Diagram (source:[53]). 28

Figure 3.5. Example of a CCA Implementation (source:[53]). 28

Figure 3.6. Mapping a subgraph into CCA (source: [53]): in the left are shown a sequence of instructions representing a subgraph code (top) and a CCA structure (bottom); in the right side of the *subgraph code* are shown the steps performed by the mapping algorithm. 31

Figure 3.7. DIM Block Architecture and Configuration Example (source: [14]). 33

Figure 3.8. Dynamic translation in the DIM Architecture (source: [14]). 34

Figure 4.1. a) Upper bound for overall application speedup as a function of the coverage, and b) ratio between $Speedup_{Overall}$ and $Speedup_{Hw}$ as a function of the coverage. 41

Figure 4.2. Example of the CFG of an inner loop. 42

Figure 4.3. a) C code for a *max* function and b) the MicroBlaze assembly code for a Megablock representing one of the possible execution paths. 44

Figure 4.4. Algorithm for detection of squares, up to a maximum size M. 45

Figure 4.5. Program execution partitioning according to basic blocks, fragments, and Megablocks. 47

Figure 4.6. Types of nodes and possible connections in a Megablock graph. 48

Figure 4.7. A data connection between two operation nodes. 49

Figure 4.8. Examples of the target code subject to transformation: a) single if statement; b) if-else statement; c) a chain of if-else statements with arbitrary size. 51

Figure 4.9. Equivalent code when applying *if-conversion* to a) single if statement; b) if-else statement; c) a chain of if-else statements with arbitrary size. 51

Figure 4.10. How to calculate the term *condition* in C using a) plain C and b) inline assembly, when targeting the MicroBlaze processor. 52

Figure 4.11. Applying *if-conversion* to a single if statement in C, when the *mux* operator is a) a multiplication and b) a logical *or*. 52

Figure 5.1. *Mul To Mux* transformation: a) graph before the transformation is applied; b) graph after the transformation. 58

Figure 5.2. Hardware solution for Megablock detection. 59

Figure 5.3. Diagram for the Megablock Detector. 60

Figure 5.4. Diagram for a hardware implementation of the Squares Detector. 61

Figure 5.5. Possible chain of steps in a *Translation* phase. 61

Figure 5.6. Algorithm for the function *rearrangeGraph.* 63

Figure 5.7. Algorithm for the function *rearrangeNode.* 64

Figure 5.8. Example of the function *rearrangeNode* 65

Figure 5.9. Routing algorithm in the Map step. 65

Figure 5.10. General system architectures for Megablock implementation. 68

Figure 5.11. General architecture for a 2D CGRA-based RPU which supports Megablocks. 70

Figure 5.12. Two possible SAr instances for two distinct Megablocks. 72

Figure 5.13. SRA instance for two hypothetical Megablocks. 73

Figure 5.14. General architecture for a Folded CGRA-based RPU which supports Megablocks. 74

Figure 5.15. C code for a *vecsum* function. 76

Figure 5.16. Assembly instructions of the repeating pattern of a Megablock found in the trace of *vecsum* running on a MicroBlaze processor, and their correspondent translation to operations to be mapped to a CGRA. 76

Figure 5.17. Graph representation of the repeating pattern of the Megablock found when executing *vecsum*. 77

Figure 5.18. General blocks for Megablock pipelined execution. 79

Figure 5.19. Execution of an LM with three stages. 79

Figure 5.20. Possible schedules for the modules of a pipelined RPU. 81

Figure 5.21. Execution using an overlapping schedule with an LM with 3 stages. 82

Figure 5.22. Algorithms for IM graph creation. 85

Figure 5.23. Input Module (IM) graph for a Megablock found in *vecsum*. 86

Figure 5.24. Loop Module (LM) schedule for a Megablock found in *vecsum*. 86

Figure 5.25. General architecture for a 2D CGRA-based RPU which supports Megablocks and Megablock pipelining. 87

Figure 6.1. Average coverage of the complete set of benchmarks when applying Megablock detection and varying several parameters. 93

Figure 6.2. Megablock detection ratio in the complete set of benchmarks. Indicates the ratio of benchmarks were valid Megablocks could be detected. 94

Figure 6.3. Individual coverage values in the main set of benchmarks, for Megablock detection using the default setup and Backward Branch Loop Detection. 96

Figure 6.4. Upper-bound speedups in the baseline case for three scenarios: execution time of the RPU is equal to Megablock CPL, execution time of the RPU is zero and execution time of the RPU and communication delays are zero. 103

Figure 6.5. Average a) speedup and b) IPC when varying the maximum number of load/store units per row. 104

Figure 6.6. Average a) speedup and b) IPC when varying the maximum number of arithmetic/logic units per row. 104

Figure 6.7. Average speedup when varying the ratio between the RPU and GPP clock frequencies. 106

Figure 6.8. Individual overall speedups for the baseline case, considering an RPU with a maximum of 8 parallel FUs and 2 load/store operations per cycle. 107

Figure 6.9. Upper bound speedups after *if-conversion* a) when considering inner loops and b) when unrolling inner loops. 112

Figure 6.10. Average a) speedup and b) IPC for adapted code when varying the maximum number of load/store units per row. 112

Figure 6.11. Average a) speedup and b) IPC for adapted code when varying the maximum number of arithmetic/logic units per row. 113

Figure 6.12. Average speedup for adapted code when varying the ratio between the RPU and GPP clock. 113

Figure 6.13. Average a) speedup and b) IPC after graph transformations, when varying the maximum number of load/store units per row 115

Figure 6.14. Average a) speedup and b) IPC after graph transformations, when varying the maximum number of arithmetic/logic units per row. 116

Figure 6.15. Average speedup after graph transformations, when varying the ratio between RPU and GPP clock frequencies. 116

Figure 6.16. LUTs, FFs and estimated maximum frequencies for Megablock Detector hardware designs. 117

Figure 6.17. FPGA resources increase when using pipelining with overlapping schedule over the non-pipelined implementation. 122

Figure 6.18. Average a) speedup and b) IPC after pipelining with overlapping schedule, when varying the maximum number of load/store units per row. 124

Figure 6.19. Average a) speedup and b) IPC after pipelining with overlapping schedule, when varying the maximum number of arithmetic/logic units per row. 124

Figure 6.20. Average speedup after pipelining with overlapping schedule, when varying the ratio between RPU and GPP clock frequencies. 124

Figure 6.21. Individual overall speedups for a pipelined architecture with overlapping schedule, considering a maximum of 8 parallel arithmetic/logic FUs and 2 load/store operations per clock cycle. 125

Figure 6.22. C code for a *fir* function. 129

Figure 6.23. Assembly code and corresponding graph operations for the *fir* Megablock. 130

Figure A.1. System Architecture (source: [27]). 152

Figure A.2. RPU Architecture (source: [27]). 152

Figure A.3. Array of FUs (source: [27]). 153

Figure A.4. PLB Injector Architecture (source: [27]). 153

Figure A.5. Speedups for DDR and BRAM scenario (source [27]). 155

Figure B.1. Average a) speedup and b) IPC in the baseline case when varying the maximum number of load/store units (geometric mean). 157

Figure B.2. Average a) speedup and b) IPC in the baseline when varying the maximum number of arithmetic/logic units (geometric mean). 157

Figure B.3. Average speedup in the baseline case when varying the ration between the RPU and GPP clock (geometric mean). 158

Figure B.4. Average a) speedup and b) IPC for adapted code when varying the maximum number of load/store units (geometric mean). 158

Figure B.5. Average a) speedup and b) IPC for adapted code when varying the maximum number of arithmetic/logic units (geometric mean). 158

Figure B.6. Average speedup for adapted code when varying the ration between the RPU and GPP clock (geometric mean). 159

Figure B.7. Average a) speedup and b) IPC for adapted code when varying the maximum number of load/store units (geometric mean). 159

Figure B.8. Average a) speedup and b) IPC for adapted code when varying the maximum number of arithmetic/logic units (geometric mean). 159

Figure B.9. Average speedup for adapted code when varying the ration between the RPU and GPP clock (geometric mean). 160

Figure B.10. Average a) speedup and b) IPC for adapted code when varying the maximum number of load/store units (geometric mean). 160

Figure B.11. Average a) speedup and b) IPC for adapted code when varying the maximum number of arithmetic/logic units (geometric mean). 160

Figure B.12. Average speedup for adapted code when varying the ration between the RPU and GPP clock (geometric mean). 161

Figure C.1. Options for program Megablock Extractor. 163

Figure C.2. Options for program Megablock Estimation. 164

Figure C.3. Options for program VHDL for Megablocks. 165

Figure C.4. Options for program VHDL for Megablock Detector. 165

Index of Tables

Table 3.1. Summary of characteristics for the three representative approaches: Warp, CCA, and DIM. 36

Table 5.1. Additional information acquired from the instructions in the Megablock sequence. 56

Table 5.2. Characteristics of the proposed Megablock identification methods: SAI and MSI.67

Table 5.3. Dependencies between the modules of a pipelined RPU. 80

Table 6.1. Characteristics of the benchmarks which form the set *no-ifs*. 90

Table 6.2. Characteristics of the benchmarks which form the set *ifs*. 91

Table 6.3. Megablock characteristics for the *no-ifs* set, only inner loops. 99

Table 6.4. Megablock characteristics for the *ifs* set, only inner loops. 100

Table 6.5. Megablock characteristics for the *no-ifs* set when applying unrolling. 101

Table 6.6. Megablock characteristics for the *ifs* set when applying unrolling. 101

Table 6.7. Cycle count and ratio of the *ifs* set, before and after *if-conversion*. 108

Table 6.8. Characteristics of the benchmarks which form the set *ifs (adapted)*. 109

Table 6.9. Megablock characteristics for the *ifs-adapted* set, only inner loops. 110

Table 6.10. Megablock characteristics for the *ifs-adapted* set when applying unrolling. 111

Table 6.11. Decrease in the number of Megablock operations, for the unrolled *no-ifs* set considering three transformations. 114

Table 6.12. Decrease in the number of Megablock operations, for the unrolled *ifs-adapted* set considering three transformations. 114

Table 6.13. IPC when the Megablock for each benchmark is executed in several platforms. 120

Table 6.14. Megablock mapping characteristics on the non-pipelined architecture. 120

Table 6.15. Comparing a non-pipelined and a pipelined architecture with sequential scheduling. 120

Table 6.16. Comparing a non-pipelined and a pipelined architecture with overlapping scheduling. 121

Table 6.17. CPL comparison between baseline and pipelined with overlapping schedule. .. 126

Table 6.18. Characteristics for the execution of the application 3dpp. 128

Table 6.19. Execution times for several implementations of the pattern detector for Megablocks. 131

Table 6.20. Average execution times in milliseconds of the *Translation* steps. 132

Table A.1. RPU FPGA Implementation.. 156

Glossary of Terms

ALU	Arithmetic Logic Unit. A digital circuit that performs arithmetic and logical operations
BRAM	Block-RAM. Configurable random access memory module present in most Xilinx FPGAs.
CGRA	Coarse-Grained Reconfigurable Array. A reconfigurable array usually including ALUs as processing elements and programmable at the word-level.
Contiguous Subsequence	Subsequence formed by consecutive elements of a sequence. The same as substring when the sequence of elements forms a string.
Coverage	Portion of GPP execution that will be replaced by execution in an RPU, over the total execution when the program runs only in the GPP.
CPL	Critical Path Length. In the context of RPUs, the number of clock cycles needed to complete the execution path with the highest number of cycles (i.e., critical path).
CPU	Central Processing Unit. The portion of a computer system that carries out the instructions of a computer program, to perform the basic arithmetical, logical, and input/output operations of the system.
Critical Loop	Loop region of a computer program where a high proportion of executed instructions occur, or where most time is spent during the program's execution.

DHSP	Dynamic Hardware-Software Partitioning. Technique where during runtime, sections of a software-only program are moved and executed in dedicated hardware components.
Dynamic Partitioning	Same as DHSP in the context of this thesis.
Executed Instructions Threshold	Megablock detection parameter. If the number of executed instructions in the processor corresponding to a detected Megablock falls below the *executed instructions threshold* the Megablock is ignored.
FF	Flip-Flop. Circuit with two stable states which can be used as a memory element to store state information. A key component of modern FPGAs.
FPGA	Field-Programmable Gate Array. Integrated circuit designed to be configured after manufacturing. Usually programmable at the bit-level.
Fragment	Sequence of executed basic blocks which do not jump backward.
FU	Functional Unit. A digital circuit which can perform operations and calculations. A more general form of the ALU.
GPP	General Purpose Processor. Specific term for a CPU which is programmable through instructions and has been designed to execute generic applications.
Hotspot	Same as Critical Loop in the context of this thesis.
ILP	Instruction-Level Parallelism. The parallelism associated with the instructions and/or primitive operations that can be performed simultaneously.

IM	Input Module. Section in the architecture of a pipelined Megablock responsible for generating the inputs for each iteration.
Induction Variable	Variable which value is increased or decreased by a fixed amount on every iteration of a loop, or is a linear function of another induction variable.
IPC	Instructions Per Cycle. Term used to describe one aspect of performance, the average number of instructions executed per clock cycle.
IR	Intermediate Representation. Data structure which represents a program or part of a program in an abstract way.
Kernel	Same as Critical Loop in the context of this thesis.
LM	Loop Module. Section in the architecture of a pipelined Megablock responsible for executing the iterations of the loop.
LOC	Lines of Code. Software metric used to measure the size of a software program by counting the number of lines in the text of the program's source code.
LUT	Look-Up Table. Hardware structure used to implement Boolean logic functions in hardware, such as AND, OR and XOR. A key component of modern FPGAs.
Maximum Pattern Size	Megablock detection parameter. Maximum number of pattern elements of a Megablock that can be detected.
Megablock	Loop structure which represents a repeatable sequence of instructions in the execution trace.

MSI	Megablock Signature Identification. A Megablock identification technique which relies on an individual signature for each Megablock.
PLB	Processor Local Bus. Bus structure provided by Xilinx to develop system architectures in Xilinx FPGAs.
RPU	Reconfigurable Processing Unit. A reconfigurable hardware unit for dedicated computations.
SAI	Single Address Identification. A Megablock identification technique which relies on the address of a single instruction.
SAr	Specialized Array. An RPU implementation which corresponds to a single Megablock.
SM	Store Module. Section in the architecture of a pipelined Megablock responsible for executing store operations according to their original order.
SRA	Specialized Reconfigurable Array. An RPU implementation which supports several Megablocks and which is runtime reconfigurable.
Subsequence	Part of a sequence of elements, where element order is maintained but consecutiveness is not enforced. E.g., *bd* is a subsequence of *abcde*.

Substring	Subsequence formed by consecutive elements of a sequence S of symbols (a string). E.g., *bcd* is a substring of *abcde*.
Type of Pattern Unit	Megablock detection parameter. The kind of pattern element used for detection (e.g., instruction, basic block).
Unrolling of Inner Loops	Megablock detection parameter. If enabled, gives priority to Megablocks with more pattern elements, forming Megablocks with unrolled inner loops. Otherwise, gives priority to Megablocks whose pattern has less elements.

1 Introduction

The challenging requirements of designing and implementing high-performance and flexible embedded systems at low cost have made the use of field programmable gate arrays (FPGAs) an attractive option [1]. These modern, high-capacity devices are being used as platforms for implementing complete systems-on-chip and include one or more general purpose processors (GPPs). Even as computation shifts to the multi-core paradigm, there is still the need for acceleration of specific computation tasks [2, 3], e.g., by connecting application-specific accelerators to the GPPs.

A flexible solution for the hardware accelerators is the use of Reconfigurable Processing Units (RPUs) [4, 5]. Figure 1.1 illustrates the organization of a typical architecture coupling an Reconfigurable Processing Unit (RPU) to the GPP. Many different possibilities can be used to couple the two main components of this architecture [6]. In the target organization illustrated, the RPU communicates with the GPP by direct connections and both have access to the system memory (i.e., the RPU acts as a traditional coprocessor). However, it is usually required a high design effort to implement those systems. The design-flow combines software development and hardware design, the latter usually starting from a specification in a hardware description language (HDL) such as Verilog [7], and thus requiring hardware design expertise.

1.1 Hardware/Software Co-Design

Hardware/software co-design [8, 9] is a methodology for designing embedded systems consisting of hardware and software components. An important part of hardware/software co-design is hardware/software partitioning. It can be used to select and map the parts of the application that will be executed in the GPP and in the RPU. It contains steps such as the detection of computation-intensive sections in the application (also known as hotspots or critical sections), mapping the computations to each of the components of the target architecture (i.e., the software and the hardware components), and adapting the software application (e.g., calls to special instructions are inserted in the application source code) to

use the hardware component. This requires the insertion of synchronization and data communication primitives.

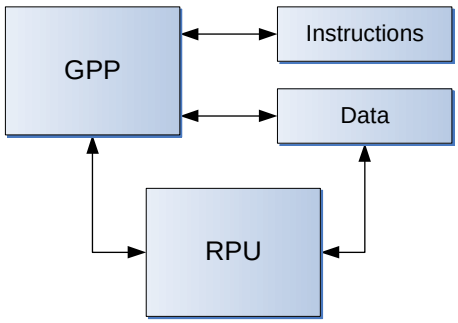


Figure 1.1. Block diagram of a typical target system which includes a RPU coprocessor acting as an accelerator of the GPP.

Depending on the tools, the hardware/software process can range from mostly manual to highly automated [10, 11]. For instance, and example of an automated approach is to use high-level synthesis tools, such as Catapult C (from Mentor Graphics), which translate C code to HDL [12]. This often requires rewriting the source code to fit the translator's requirements and limitations. Implementing the interface between the generated hardware and the software is also necessary, a task which might require additional, manually-developed hardware, and further source code modifications. In this scenario, the developer still needs non-trivial knowledge on digital systems design, and adapting applications to use custom hardware is done on an application-by-application basis.

There has been a continuous effort to automate the migration of computations from a GPP to custom hardware. In a promising approach the partitioning is done over the binaries of the application, while it executes on the processor [13, 14]. The computation is transparently moved from the GPP to the coprocessor. We refer to this approach as Dynamic Hardware-Software Partitioning (DHSP), or simply dynamic partitioning.

1.2 Dynamic Partitioning

Delaying partitioning until the application executes enables one to use information only available at runtime. With this information it is possible, for instance, to partition the application according to its current execution, enabling the implementation of more efficient designs. Another possible application of dynamic partitioning is to improve hardware

portability between different systems, by discovery of the specific RPU that is being used by the system at the time the application executes, and mapping the computation to that RPU.

Dynamic partitioning has its costs. As some of the partitioning steps are moved to runtime, there is additional overhead to be considered. In addition, as execution time becomes an important characteristic for the steps done online, it is necessary to adapt current algorithms or to propose new algorithms considering a runtime scenario.

Dynamic partitioning is reminiscent of dynamic compilation (also known as Just-In-Time – JIT – compilation). During JIT compilation, several compilation steps are delayed until the execution of the program (most notably, the generation of machine code). The Java platform [15] is probably the most popular example of dynamic compilation, and has been used with success to write applications which can execute in a variety of devices (e.g., smartphones) and across several operating systems (e.g., Windows, MacOs, Linux). The HotSpot [16] is an example of a Java Virtual Machine which uses dynamic compilation to bridge the gap in performance between a compiled and an interpreted language [17].

Performing compilation directly from the program binary is known as binary translation [18]. It has been successfully used to transparently execute programs in platforms not compatible with the ones they were originally compiled for. For instance, Pentium microprocessors use hardware binary translation to translate instructions of the old x86 ISA to the new ISA of the microprocessor [19]. The Rosetta [20] is a binary translation software used by Apple when it moved the Macintosh from PowerPC to Intel processors, to allow previous applications to run in the system without modification. Other example is the Crusoe [21] microprocessor, which performs binary translation dynamically in hardware. While in the first two cases binary translation was used to improve compatibility, possibly at the cost of performance, the Crusoe tries to achieve similar performance with a lower thermal envelope. It translates and executes binaries written in the Intel x86 ISA to a microprocessor which has a substantially different architecture, designed to be more power efficient.

In this thesis we propose novel techniques for DHSP in the context of embedded computing systems. In particular, we propose a novel kind of loop, the Megablock, designed for runtime detection and for adapting sequential code to parallel computation models. The proposed techniques focus on the Megablock, and the work presented here shows methods and algorithms for the detection, identification, implementation, and optimization of Megablocks (e.g., inner loop unrolling, pipelining of Megablocks). We also present an

extensive study of the impact of using the Megablock as a partitioning unit over a comprehensive set of benchmarks.

1.3 Thesis Statement and Main Contributions

Thesis Statement: *We can build a system which automatically moves loops, originally meant to be run on a general purpose processor, to a reconfigurable fabric, in order to improve the execution of the program according to some criteria (e.g., execution time, energy consumption). The loops are moved while the unchanged program binary executes in the processor, and by using adequate structures (i.e., the Megablock) and algorithms, it is possible to consider techniques (such as loop pipelining) not previously considered for dynamic mapping computations to reconfigurable fabrics.*

The focus of this thesis is on the use of DHSP in embedded systems. The main contributions of this thesis are:

- It proposes the Megablock, a repetitive pattern of instructions that represents a path in the execution flow.
- An algorithm for detection of Megablocks based on a pattern-matching technique which can be fully agnostic to the instruction format of the target GPP;
- A graph-based, architecture independent, intermediate representation for Megablocks;
- A scheme for applying an *if-conversion* technique to transform code such that it can expose more useful Megablocks when dealing with control-intensive applications;
- Proof-of-concept implementations for several of the proposed ideas and evaluation using an FPGA board;
- A technique which pipelines the iterations of Megablocks in hardware;
- An extensive study of the impact of the Megablock over a comprehensive set of integer benchmarks from embedded computing;

The results of this thesis have contributed to a number of publications [22-28].

1.4 Organization

The remainder of this thesis is organized as follows:

Chapter 2 introduces the concepts needed for the subsequent chapters of this thesis. The covered subjects include compilation in general (both static and dynamic), the processor/coprocessor paradigm, and reconfigurable hardware.

Chapter 3 introduces several research efforts in the fields related to our approach, such as approaches based on traces, runtime reconfiguration, and binary translation. Furthermore, we describe in detail three relevant approaches which focus on dynamic partitioning for reconfigurable architectures. All the three approaches transparently move instructions being executed in a General Purpose Processor (GPP) to reconfigurable hardware, bearing in mind embedded systems as a target.

Chapter 4 describes the Megablock, a repetitive pattern of executed instructions found in the trace of a program. In this chapter we propose the Megablock as a partitioning unit for moving sequential code to the parallel computing model provided by RPUs and compare with the partitioning units used in other works. The chapter also presents an algorithm for detection of Megablocks, proposes an Intermediate Representation (IR), and introduces source-to-source transformations to detect additional Megablocks.

Chapter 5 presents practical aspects related to the implementation of Megablocks. We explain how to build the Intermediate Representation (IR) introduced in Chapter 4, as well as introduce a set of transformations which can be applied over the IR. Although the detection of Megablocks can be done offline, during a profile phase, we still need a method to identify these previously detected Megablocks at runtime, when the application executes. In this chapter we propose two methods for runtime Megablock identification. Finally, we present several architecture models capable of implementing Megablocks, and explain how we can augment a Megablock-enabled architecture to support pipelining of Megablocks.

Chapter 6 presents extensive results using the techniques introduced in previous chapters, over a comprehensive set of benchmarks. We present results about Megablock coverage, consider several scenarios regarding Megablock mapping (i.e., baseline results, *if-conversion*, graph transformations), and show results for pipelined Megablocks.

Chapter 7 concludes the thesis and presents ideas on how to expand the current work.

Finally, we include three appendixes that present a proof-of-concept used to evaluate some of the techniques presented in this thesis, additional results, and a brief mention about the most important software tools developed for this thesis.

2 Background

The purpose of this chapter is to provide the core concepts needed to understand Dynamic Hardware-Software Partitioning (DHSP). The covered subjects include compilation in general (both static and dynamic), the processor/coprocessor paradigm, and reconfigurable hardware.

2.1 General Purpose Processors and Execution Flow

General Purpose Processors (GPPs) have been the central components of computing for the past decades [29]. High-level languages and compilers make GPPs relatively easy to program and many of today's applications run on GPPs.

Each GPP has an associated Instruction Set Architecture (ISA) [29], which defines the programming part of the GPP: data type support, allowed instructions, available registers, etc. The binary representation of the set of instructions directly supported by a GPP is called the machine language, and the human-readable version of the machine language is called the assembly language.

A program is formed by a sequence of instructions, laid out sequentially, and uniquely identified by an instruction address. When a program runs on a GPP, each executed instruction can be viewed as a step given by the program. By default, the GPP executes instructions in sequence. However, certain instructions can change the flow by instructing the GPP that the next instruction to be executed is several instructions ahead, or several instructions before in the sequence. The change of the execution flow is typically implemented by jump/branch instructions, which are commonly referred to as *control-flow instructions*.

There can be *conditional* or *unconditional* branches. Jump instructions (usually referred as unconditional branches) always change the execution flow. Conditional branches depend on a condition to change the execution flow (e.g., jump if the value of a certain register is zero). Conditional branch instructions define branching points in the code. A branch is a sequence of instructions that is executed if the condition of the branch instruction is met. Those branch instructions represent a point where the execution flow will take one of two paths (or more, if the destination address when the branch is taken is a variable). To make a distinction between

a jump/branch to an address after the jump/branch instruction and a jump/branch to an address before the jump/branch instruction, the former is called a forward branch, while the latter is called a backward branch.

The sequence of instructions executed by the GPP during an execution of a program is called a program *trace*. The trace represents all the paths and choices of a particular program execution. Note that different executions can generate different traces, depending, for instance, on the branches taken during the program execution.

Associated to jumps/branches is the basic block [30], a block of code with a single entry-point and a single exit-point. It usually corresponds to the sequence of instructions between the instruction executed after a branch, and the next jump/branch instruction.

2.2 Data Hazards

Most GPPs follow the Von Neumann model [31], which assumes that instructions are executed sequentially, typically following the order indicated by the program/compiler. However, one can improve performance by reordering some instructions. When reordering instructions, it is fundamental to maintain the original functionality of the program. An hazard happens whenever there is a data dependence between instructions, and a reordering of those instructions changes the correct behavior of the program [29]. Consider two instructions, A and B, where A executes before B. There are three possible data hazards:

RAW (read after write): When instruction B fetches a result which is written by instruction A, but instruction A has not completed yet, making B read a (possibly) wrong value.

WAW (write after write): When instruction A and instruction B write a result to the same place, and the last instruction writing the result is A, instead of B.

WAR (write after read): Instruction A reads a value which is latter overwritten by instruction B. Hazard happens when instruction B overwrites the value before instruction A could read it.

2.3 Coprocessors

When GPPs are not able to meet certain non-functional requirements (e.g., execution time, power dissipation, energy consumption), an effective way to improve the GPP system is to extend the GPP with a customized hardware unit, in the form of a coprocessor [8, 32].

Coprocessors can perform faster than a GPP for specific tasks because, among other things, they allow for more *parallelism*, by providing additional computational units not present in a general case; *alternative computation models* which can be more efficient for certain tasks (e.g., data-streaming); *spatial computing*, by replacing sequences of instructions (*temporal computing*) by direct connections between components, diminishing the instruction overhead related to fetch-decode stages [33].

However, developing and testing hardware is significantly harder than software. When using target architectures consisting of a GPP coupled to a hardware unit (e.g., acting as a coprocessor) one identifies which program sections are most frequently executed and then migrates those portions to the hardware unit. This is known as *hardware/software co-design* [8, 9]. This approach is usually viable as most applications follow the 90-10 rule of thumb: 90% of the execution time is spent on 10% of the program code [29], often found in small groups of instructions which are executed in loop for many iterations. Those code sections are known by various names, e.g., *critical loops*, *kernels*, *hotspots*. Most hardware/software co-design approaches start by identifying the loops of the programs.

2.4 Coprocessor Tradeoffs

Consider a computing system with a GPP running a program, and that at some point the system is at state A. After running the program for a while, the system arrives at state B, changing several elements of the system, e.g., the values in the registers of the GPP, the contents of the main memory. As a general case, we consider that the objective of a coprocessor is to help the processor going from state A to state B, while allowing for a trade-off between one or more parameters. It should be noted that it may not be necessary to arrive exactly at the same state B, some of the values can be temporary values which will not be used after. However, if the state is the same, we can guarantee that if the execution continues on the processor, it will be correct. State can have different meanings: for a simple embedded application it can refer to the actual state of the system: the values in the registers of the processor, of the memory at each address, etc. In a more complex system with virtual address space and concurrent processes, the state can, for instance, refer to the virtual state of a single process.

Figure 2.1 shows some trade-offs we can achieve when using a coprocessor. The figure represents two common parameters, execution time and consumed energy. In case a), the GPP

executes the program and takes the system from a state A to a state B, consuming a certain amount of energy and time. Case b) consumes about the same energy, but takes less time to execute, while case c) takes the same time to execute while reducing the energy consumption. Both arrive at the same state B.

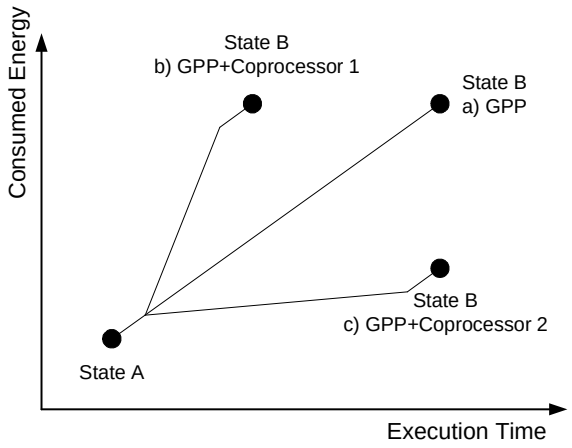


Figure 2.1. Example trade-offs when using a coprocessor.

When we go from a system state A to a system state B with the help of a coprocessor, tackling the problem at the same level of the considered states may provide a more fine-grained control of the problem. This is the case when considering hardware/software co-design: if, when working with an embedded application where the considered state is composed by the contents of the processor registers and of the memory, the partitioning was done at the level of the assembly, instead of going up to the source code; if, when working with a Java application where the considered state is the one given by the Java Virtual Machine (JVM) [15], the partitioning was done at the level of the bytecode representation, instead of going down to the processor implementation.

2.5 Reconfigurable Processing Units

One way to implement a coprocessor is to design an Application-Specific Integrated Circuit (ASIC). This is the solution which usually gives the best performance [34] with respect to area and power. However, ASICs are extremely expensive: as any Integrated Circuit (IC), they have initially high production costs and only become cost effective when mass-produced. In addition, ASICs can only do what they were designed for (the circuit is unchangeable after fabrication). Usually, ASICs are designed only when: the hardware unit will run a significant part of the computation; will be produced in high volumes; and when a

software-only solution would not give satisfactory results (e.g., hardware video encoders/decoders for established codecs, such as MPEG-2 and H.264).

These limitations associated to ASICs strongly motivate the use of reconfigurable hardware [35]. Reconfigurable hardware usually takes the form of an IC with several computational components and reconfigurable connections. Such as the ASIC, all components of a reconfigurable IC are already in place and cannot be modified in the field. What makes the hardware reconfigurable is the capability to change, through configuration, the functions of the components and their connections.

Figure 2.2 shows a possible reconfigurable fabric that can be used as an RPU in a system, with components represented as named boxes, connected by reconfigurable interconnect resources. The components can be as simple as Look-Up Tables (LUTs) and Flip-Flops (FFs) or as complex as Functional Units (FUs) with one or more Arithmetic-Logic Unit (ALU), memories or even entire General Purpose Processors (GPPs) [36].

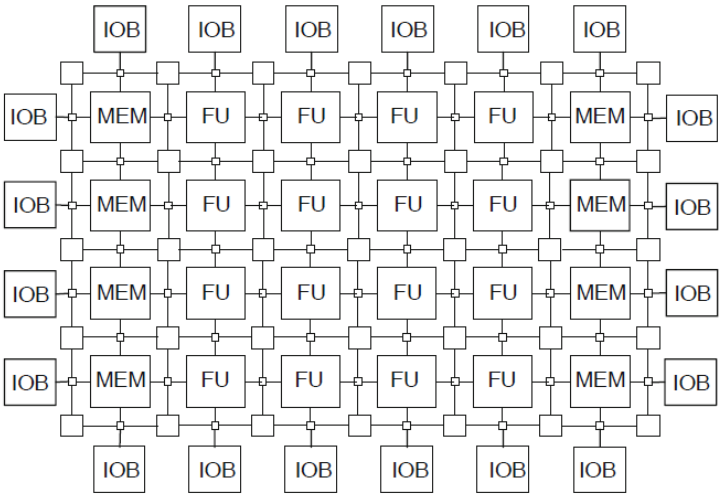


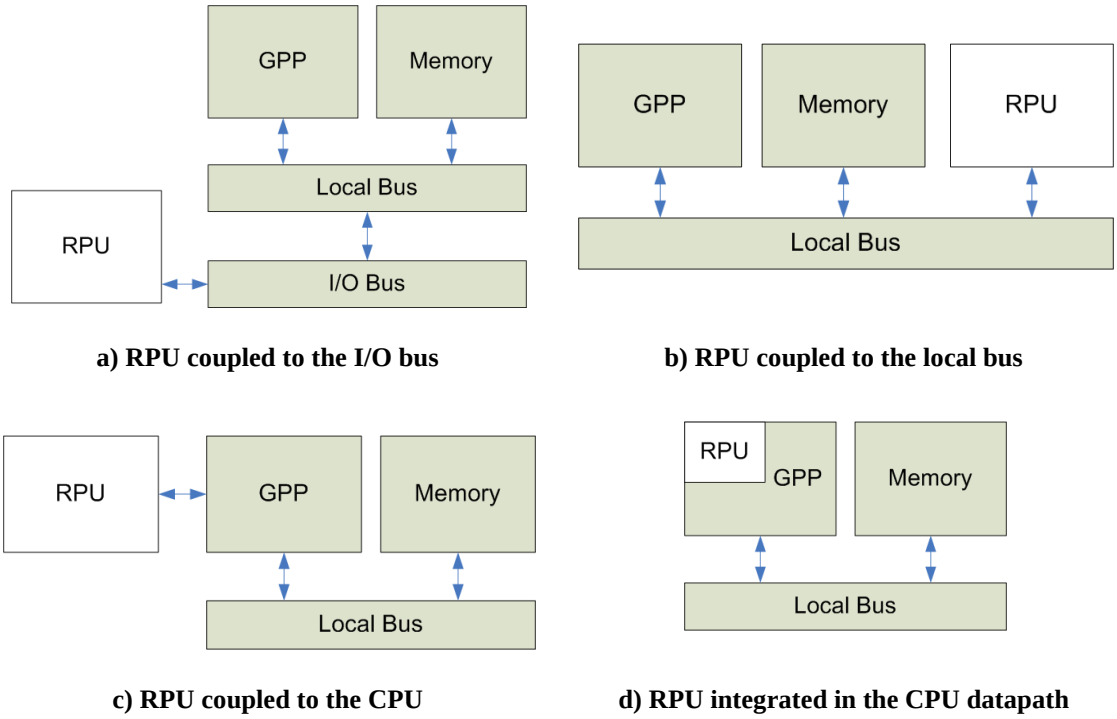
Figure 2.2. Possible two-dimensional structure for a reconfigurable fabric (source: [36]). FU identifies Functional Units, MEM identifies local memories, and IOB identifies Input/Output blocks.

The interconnect resources can be very simple, e.g., allowing connections between only some neighbor components, or more complex, e.g., allowing connections from each component with any other component in the fabric. Input/Output Blocks (IOBs) can be used for communication with components outside the RPU. Reconfigurable hardware can be classified into two groups according to the data-size of their components (*granularity*) [37]. *Fine-grained* reconfigurable hardware has components which work with data-sizes of a

couple of bits (e.g., LUTs, FFs). If the components have higher bit-widths (e.g., from 8 to 32 bits), the reconfigurable hardware is usually considered *coarse-grained*. Finer granularity means more flexibility on one hand, and higher overhead on the other.

The degree of flexibility provided by the components and connections of an RPU determines the effort needed to *map* computations to that RPU. An RPU with less flexibility in configurability is not as expressive, but requires lower mapping effort.

The *communication costs* between a GPP and an RPU depend on the kind of coupling the RPU has with the other components in the computing system. We can consider the four general cases of coupling [38] represented in Figure 2.3. They are ordered from the loosest coupling to the tightest coupling. Loosely coupled RPUs are easier to integrate in a system, but usually have higher communication delays. Tightly coupled RPUs are attractive due to lower latencies and communication delays, but integration with the host is more invasive and usually implies co-designing the RPU and the GPP.



a) RPU coupled to the I/O bus

b) RPU coupled to the local bus

c) RPU coupled to the CPU

d) RPU integrated in the CPU datapath

Figure 2.3. Types of RPU coupling with respect to the host system.

2.5.1 FPGAs

Field-Programmable Gate Arrays (FPGAs) [39] are an example of fine-grained reconfigurable hardware. Although some FPGAs can have coarse-grained components, such

as multipliers, their smaller and most common components are LUTs and FFs, addressable at the single bit level. FPGAs are mass-produced, making them relatively affordable, and their extreme flexibility allows for broad design space exploration. If the components of a reconfigurable fabric have a sufficiently fine granularity, as in the case of FPGAs, it is possible to implement virtually most digital hardware circuits, since their design components include typical basic blocks used when designing ASICs.

To design hardware circuits for FPGAs, we typically use Hardware Description Languages (HDLs), such as VHDL [40] or Verilog [41]. Specific suite of tools (e.g., Xilinx ISE [42]) can then synthesize the hardware described in the HDL into the configuration bits of a specific FPGA (known as the *bitstream*). Between the HDL description and the configuration bits there is a number of important steps, commonly handled by separate programs [43].

The first step is to convert the description into logic gates, by an RTL (Register-Transfer-Level) *Synthesis* tool [44]. At this point, the synthesis tool employs a number of optimizations, such as *logic minimization* [44]. The logic gates are fed to a *mapper*, which will find a correspondence between the abstract logic gates and the kind of components present in a specific reconfigurable device (*Mapping*). The next step, *Placement*, is responsible to assign each of the components in the description with a real component of the fabric. Then, in the *Routing* step, a *router* establishes the connections between the components. In the last step, the bitstream is generated.

2.5.2 CGRAs

Coarse-grained reconfigurable architectures (CGRAs) [37] use functional units with higher bit-widths (e.g., ALUs with 8, 16 or 32 bits) having native support to word level computing. CGRAs are an alternative to FPGAs for cases where flexibility at the bit-level is not necessary. By reducing flexibility, CGRAs are able to outperform FPGAs on certain characteristics.

For instance, due to the extreme flexibility and their ever increasing sizes, FPGAs have time-consuming design cycles. Furthermore, the back-end phases are very complex as the tools need to deal with a high volume of information[1]. For large designs, the total time of the mapping and placement and routing process can go from several minutes to several hours,

[1] For instance, a Virtex-5 XC5VLX110 needs a file of 29.1 Mbits to configure the entire device [45].

depending on the effort of the algorithms. By using larger and fewer building blocks, the size of the mapping problem reduces drastically, as is the case with CGRAs.[2]

CGRAs also have a more predictable clock frequency. Although there have been efforts on the development of asynchronous reconfigurable logic [48], most reconfigurable architectures are synchronous. In the case of FPGAs, the clock frequency is dictated by the maximum delay of the combinatory circuits between registers (i.e., critical path delay). This delay is not only dependent of the characteristics of the circuit design, but also dependent on the ability of the mapping tools to reduce this path. Coarse-grained arrays usually have fixed transfer rates between components, and a fixed-clock frequency.

2.6 Dynamic Compilation

Compilers [49] are programs which translate source code written in a programming language into another computer language (in most cases, machine language which can be executed by a GPP). Compilation is known as *static compilation* or *offline compilation* when it is performed prior to the execution of the program.

Although the first job of a compiler is to translate between languages (or representations of computations), most compilers also perform transformations and optimizations to the code [50]. It is crucial that a compiler produces target code that is functionally equivalent to the source code, but the quality of a compiler is usually measured by how well it tunes the program to specific requirements, such as execution time, or program size. As static compilation is done before the program is released, the compiler can use complex algorithms to transform the program, bearing in mind those requirements.

Dynamic compilation presents another approach for compilation. Steps of the compilation process are delayed until the execution of the program (*runtime*). Then, at runtime, those compilation steps are performed, possibly using additional information not available offline (e.g., specific information about the hardware which is running the program, information about the behavior of the program).

Java is a widely popular language [51] that relies on dynamic compilation for a number of compilation steps. The program is distributed in an intermediate representation (the Java

[2] There is a trend to use fine-grained reconfigurable fabrics (such as FPGAs) to implement CGRAs [46, 47], thus, creating an architectural layer easier to deal with.

bytecodes), which is written in a virtual ISA (i.e., the ISA of the Java Virtual Machine – JVM [15]), and compilation addressing the real processor is performed during program execution (known as JIT – Just In Time - compilation). Compiling during runtime allows Java compilers to take advantage of additional information available at runtime, as well as significantly improves the performance of interpreted code.

Delaying the last part of compilation to runtime is also used to enhance portability. By compiling to an intermediate representation (the Java bytecodes) theoretically every system with a mechanism implementing the JVM can run the program. This approach has already been used in embedded systems to distribute the same application (e.g., software games) across very different models of, e.g., smart phones. Another advantage of this approach is to allow developers to use the same toolchain and development environment to develop applications, instead of using a toolchain and special compilers for each target system.

In traditional hardware/software co-design [52], the decision of which parts of the program are executed in the GPP and which parts are executed to a coprocessor is performed at design time, and that information is encoded into the binary. An alternative approach is to delay this decision until the execution of the program. This way it is possible to benefit from dynamic compilation features, such as enhanced portability and runtime adaptation. Herein, we refer to this approach as Dynamic Hardware-Software Partitioning, DHSP, or simply dynamic partitioning (see the problem formulation in Figure 2.4) [13, 14, 53].

We consider at least four phases in dynamic partitioning: *Detection*, *Translation*, *Identification* and *Replacement*. These phases do not necessarily need to be executed by this order (e.g., *Translation* can be performed either after *Detection* or after *Identification*).

Detection determines which sections of the application are candidates to be moved to the coprocessor; *Translation* transforms detected sequences of instructions into an equivalent representation for the coprocessor; *Identification* finds, in the program execution, the sections which were detected as candidates to be moved; Finally, *Replacement* refers to the mechanisms by which the execution flow moves from the GPP to the coprocessor and vice-versa.

Each one of these phases can implement its own set of algorithms and have different levels of complexity. For instance, during *Detection*, an algorithm can, before deciding to accept a section as candidate, estimate if that particular section is worth moving to the coprocessor, to reduce the number of candidates. *Translation* algorithms can use intermediate representations, or perform transformations and optimizations over the sequence of

instructions using runtime information (e.g., Intermodular Inlining [54]). *Identification* can use several heuristics to locate the detected sequences in the execution trace. The *Replacement* can be done either by direct signals to the GPP, or by rewriting the instruction memory.

Problem formulation: Dynamic Hardware-Software Partitioning (DHSP)

Given a computational system composed by a GPP, a coprocessor, and an application which executes on the GPP, dynamic partitioning analyses the application execution, decides which sections of the application should be moved to a coprocessor, and executes the application according to the decision, so that the global execution of the application can be improved according to some established criteria (e.g., execution time, energy consumption).

Figure 2.4. Dynamic Hardware-Software Partitioning problem formulation.

Note that the problem formulation in Figure 2.4 states that deciding which computations to move and executing the decision take place during program execution. However, it does not imply that all the necessary steps to perform hardware-software partitioning must be done at runtime. For instance, if the coprocessor is an RPU, the dynamic partitioning system can have a repository of pre-built RPU configurations, and decides at runtime which configurations to be used. Alternatively, a full-runtime system performs all tasks (e.g., detection, translation, identification, replacement) during program execution. The problem formulation leaves these possibilities open for different implementations.

Being able to automatically take advantage of the coprocessors in a computing system without resorting to recompilation is particularly appealing to embedded systems. It is common for embedded systems to often rely on very specific hardware modules to meet their requirements. With this technique, it may become easier to take advantage of different accelerators or to try different hardware solutions. It can also enable seamless integration between applications and a family of RPUs which can vary in some particular features, such as local memory, and/or number of functional units.

2.7 Summary

This chapter briefly introduced a number of important concepts that are used throughout the thesis, from static to dynamic compilation, possibly considering hardware/software co-design, and the use of reconfigurable processing units (e.g., based on reconfigurable hardware) connected to a general purpose processor.

When compared to ASICs, reconfigurable hardware is much more flexible, and reconfigurable fabrics such as FPGAs can virtually implement the circuits of any ASIC. This flexibility comes at a price though: due to the reconfiguration overhead, reconfigurable hardware can be slower, needs more area and dissipate more power [39]. On the other hand, reconfigurable hardware offers the possibility of experimenting hardware designs, and of applying hardware acceleration to cases which otherwise would be cost-prohibitive. Another untapped potential of reconfigurable hardware is that it can adapt itself to each application, e.g., during runtime.

Although reconfigurable hardware promises the possibility of accelerating many types of applications, this promise remains partially unfulfilled, mainly to difficulties related to the available tools. By transparently migrating computation to coprocessors (e.g., RPUs) and using information available only at runtime, we see dynamic partitioning as a possible candidate to further bridge this gap.

We formulated the problem of dynamic partitioning, and identified four different phases in dynamic partitioning: *Detection*, *Translation*, *Identification* and *Replacement*.

3 Related Work

This chapter introduces relevant work related to our approach. We include approaches based on traces, runtime reconfiguration, and binary translation, and describe in detail three relevant works in dynamic partitioning addressing reconfigurable computing architectures. These three works transparently move instructions being executed in a General Purpose Processor (GPP) to reconfigurable hardware, bearing in mind embedded systems as target.

3.1 Binary Translation

Section 2.6 of Chapter 2 introduced dynamic compilation, using Java bytecodes as an example of an intermediate representation. There are cases where a computing system, instead of translating instructions of a virtual ISA (such as the Java Virtual Machine), uses the binary code for a real microprocessor [55-57]. This is called binary translation [18], and can be either static or dynamic.

Approaches such as Paek et al. [58] perform loop detection by doing static analysis of the executable binary. In their work they decompile the code and analyze loop structures. They focus on innermost loops, without branches and whose iteration count can be determined statically. They also consider loop unrolling when the iterations of both the inner and the outer loop can be determined statically (only the inner loop is unrolled). The target coprocessor is a data-flow oriented CGRA which supports context pipelining. After loops are detected, the binary is modified to include the CGRA mapping and communication routines.

One example of dynamic binary translation is the Crusoe microprocessor. The Crusoe translate and executes binaries written in the Intel x86 ISA on the fly, to a microprocessor which not only has different ISA, but a substantially different architecture [21][3]. The Crusoe uses a Very Long Instruction Word (VLIW) processor, an architecture designed to take advantage of Instruction Level Parallelism (ILP). The Crusoe is able to execute native x86

[3] Pentium microprocessors also uses binary translation, to translate instructions of the old x86 ISA to the new ISA of the microprocessor [19].

code at a performance level similar to a superscalar processor, while achieving a lower thermal envelope.

3.2 RPU Architectures

As previous work has shown, if we move critical loops to dedicated hardware units, we can have significant performance improvements [13]. There have been many proposals on accelerators using reconfigurable computing concepts, as well as a plethora of architecture designs. Most well-known examples include Matrix [59], RAW [60], Adres [61], REMARC [62], Morphosys [63], GARP [64], Chimaera [65], Piperench [66], XPP [67] and Rapid [68]. Each one of these architectures proposes unique features and tries to address faster execution and/or energy savings for a set of algorithms and/or domain-specific applications. Currently, there is a wide choice of hardware accelerators, and FPGA-based reconfigurable fabrics are an accessible technology to implement them. However, a significant hurdle for reconfigurable architectures is the significant cost of mapping the programs.

In a first phase, the portion of the program that executes on the reconfigurable hardware needs to be translated to the new architecture. In some cases, the reconfigurable architecture needs to be manually programmed, using an HDL like-language, while in other cases, the authors provide compilers specifically developed for the architecture, which either support an already established language, or a new high-level specification (e.g., as in MorphoSys [63]). However, writing a good compiler is not trivial, and it is to be expected that compilers for new and substantially different architectures are not as mature as compilers for well-established architectures, which already have many years of development and testing[4].

After the translation of program portions to the reconfigurable architecture, the program running on the GPP needs to call the custom hardware. These calls can be inserted in the executable code either manually by a programmer, or automatically by a compiler. There has been a substantial effort in the development of compilers which statically partition a program into software and hardware parts, and automatically generate the HDL description of the hardware parts [70, 71]. With an important role in that process are the C-to-gates compilers,

[4] Projects like LLVM [69] can partially solve this problem. LLVM is a compiler infrastructure which provides front-ends to well-known languages, and abstracts the target architecture from most phases of compilation, only introducing it when absolutely necessary.

which focus on synthesizing hardware modules, usually written in an HDL, from code written in a subset of C [10, 11].

3.3 Dynamic Partitioning Approaches

The execution trace, the sequence of instructions executed by a program, is the starting point of many dynamic approaches. It is possible to extract information from traces which is only available at runtime, such as the frequency of taken paths, and use that information to generate more efficient code.

Bala et al. [72] developed Dynamo, a system which transparently improves the code executed by a GPP. Dynamo monitors the execution of the native instructions of a GPP and uses runtime information to make native-to-native transformations. The working unit of Dynamo is the *fragment*, a dynamic version of the superblock [73]. A *fragment* is formed by a sequence of executed basic blocks which do not jump backward. By speculatively executing fragments, Bala et al. improved the execution time of code which was compiled with default compiler optimizations.

Gal et al. [74] used a trace-based compilation technique for dynamically-typed languages (e.g., JavaScript, Python). In such languages, the types of expressions are not statically defined and may vary during runtime. To cope with this, compilers produce code capable of resolving any kind of type combinations. The objective of their work is to reduce the expressions to the types being actually used by the application at runtime, producing more efficient code. They work over the granularity of the loop, based on the expectation that they represent a big portion of the program execution, and that inside loops, the types of the values are mostly invariant. Loops are detected and built over the execution trace by monitoring backward branches. They propose a structure called *trace tree*, which represents the hot-paths of a loop.

Below we examine in detail three relevant approaches which are closely related to the work in this thesis: Warp [13], CCA [75] and DIM [14].

3.3.1 WARP

Lysecky *et al.* propose the Warp Processor [13], a system which implements a full-online dynamic partitioning approach. The system includes a GPP, a fine-grained RPU (Reconfigurable Processing Unit), and a dynamic mapping module. The dynamic mapping module automatically detects critical loops on the GPP and maps the corresponding binary

code to the fine-grained reconfigurable, logic-based, RPU. Originally, the authors considered a system which used a common hardcore GPP. In a posterior work they used the same technique to improve the competitiveness of soft-core processors in embedded systems [76].

The Warp architecture is composed by a GPP, with separated buses for data and instructions (Harvard architecture), a profiler, an on-chip CAD (Computer-Aided-Design) module, and a custom-made FPGA acting as the RPU (see the block diagram in Figure 3.1).

The profiler is non-intrusive – i.e., the profiler does not use instrumentation, which changes the binary code and/or the processor execution to introduce instructions which gather information – and is attached to the instruction bus. The profiler is lightweight, and only monitors the addresses of the executed instructions.

The on-chip CAD Module is connected to the instruction bus and receives information from the profiler. It is responsible for translating the loops detected by the profiler to the FPGA. The CAD module is implemented as another GPP running the mapping tools developed by the authors of Warp.

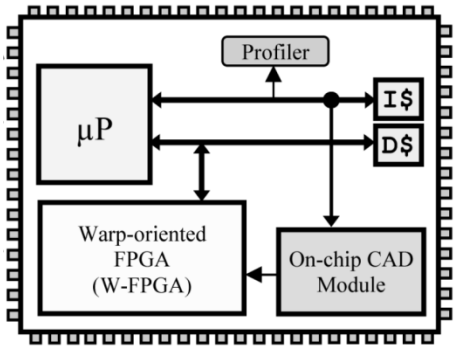


Figure 3.1. Block Diagram for the WARP Processor (source: [13]).

The custom FPGA, called Warp-Oriented FPGA (see Figure 3.2), besides the configurable logic, also includes a data-address generator (DADG) with loop control hardware (LCH), three input-output registers, and a 32-bit multiplier-accumulator (MAC). All memory accesses from the FPGA are handled by the address generator, and the LCH is used to reduce the loop overhead of critical kernels. A 32-bit MAC is included as it is an operation used frequently enough to justify dedicated hardware, instead of an implementation using the reconfigurable logic.

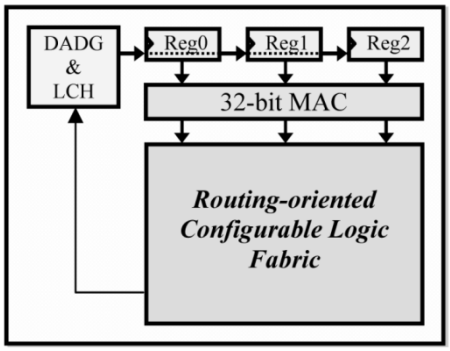


Figure 3.2. Block Diagram for the W-FPGA (source: [13]).

The reconfigurable logic of W-FPGA was designed to minimize the time spent during hardware synthesis, and has significant differences from the reconfigurable logic employed in commonly available FPGAs. Instead of optimizing the performance of Look-Up Tables (LUTs), e.g., by using LUTs with 5-6 inputs [77, 78], and of Configurable Logic Blocks (CLBs), e.g., by using clusters with 8 LUTs [79], they focused on a simpler design which allows faster mapping and placement. This resulted on a reconfigurable architecture which uses 3-input/2-output LUTs and CLBs with 2 LUTs each. Furthermore, the routing was also simplified, and each CLB is connected to a switch matrix which has 8 channels – 4 for the adjacent nodes, and 4 for routing between every other switch matrix. As a result, the mapping, placement and routing algorithms developed for W-FPGA are significantly simpler and faster than the ones used on common FPGAs.

The Warp system maps hotspots consisting of innermost short loops to W-FPGA. To detect those loops, they take advantage of the fact that there is a high correlation between short backward branches in a program and the beginning of a loop. Every time the profiler detects a backward branch, the address is stored in a small cache (16 entries of 8 bit values) which monitors branch frequencies. If the value of a branch frequency saturates, a shift is performed to all values, to maintain a list of relative frequencies. When an address reaches a certain threshold of saturations (the value of 10 is referred in [76]), the address is considered as the beginning of a critical loop.

After a loop is detected, the on-chip CAD Module reads the binary code with the instructions of the loop. It then transforms the loop instructions into hardware descriptions, and the hardware descriptions into a bitstream. The bitstream is then loaded into the custom FPGA (i.e., W-FPGA). There are, however, constraints in the implementation of each loop. A

loop may include accesses to the memory, but they must follow regular access patterns. In addition, the number of iterations of the loop must be known (however, the loop can terminate at any iteration).

The on-chip CAD Module does extensive transformations to loop instructions before they can be translated and run in the W-FPGA. The first step of the translation is *decompilation*. The CAD tool converts each binary instruction of the loop into an equivalent register transfer representation, which is independent of the instruction set. This representation is used to build a control flow graph (CFG) and a data-flow graph (DFG), then merged into a Control/Data Flow Graph (CDFG). The CDFG is used to apply standard compiler optimizations and to detect higher-level constructs such as loops and *if* statements [80].

The next step is *Partitioning* (as represented in Figure 3.3). The kernel identified by the profiler is analyzed, and by using a simple partitioning heuristic, which tries to maximize speedup and reduce energy, the partitioning algorithm decides if the kernel should be implemented in hardware.

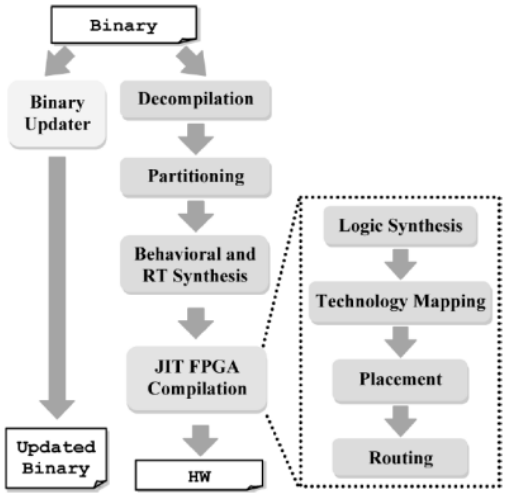


Figure 3.3. Binary to Hardware Translation Flow (source: [13]).

During *Behavioral and Register-Transfer Synthesis*, the CDFG is converted into a hardware circuit description, which is in turn converted to a *netlist* format. The *JIT (Just-In Time) FPGA Compilation* step is similar to the traditional synthesis, mapping, and placement and routing, albeit adapted to W-FPGA and using customized tools and algorithms, optimized for runtime utilization. In *Logic Synthesis*, the hardware circuit is optimized. The compiler creates a directed acyclic graph and applies a custom two-level logic minimization [81],

which traverses the logic network in a breath-first manner, applies logic minimization at each node and uses a single expansion phase.

During *Technology Mapping*, the compiler transforms, in a first-pass, the *netlist* representation to match the 3-input/2-output LUTs of W-FPGA, using a greedy hierarchical graph-clustering algorithm. During a second-pass, the compiler packs the LUTs into CLBs.

In the *Placement* step, the compiler uses a greedy dependency-based positional algorithm to place the CLB nodes onto the configurable logic. Initially, the algorithm determines the placement of the CLBs relatively to each other. After that, the result is superimposed and aligned.

Finally, the compiler uses a custom router [82, 83] to perform the *Routing*. The router uses the same algorithm used in the Versatile Place and Route's (VPR) tool [43, 84], with the routing model cost of the W-FPGA. The algorithm allows the overuse of routing resources and illegal routes, and eliminates illegal routes by repeating routing iterations. The algorithm is greedy and uses the adjusting cost to discourage selecting the same initial route during subsequent iterations. After determining a valid global routing, the compiler builds the routing conflict graph, having the W-FPGA technology into account. To resolve conflicts, it uses a simple and greedy vertex coloring algorithm [85].

If the translation succeeds, the program is updated by the Binary Updater (see Figure 3.3). The original program is modified by introducing a branch instruction which will jump to code responsible to initialize W-FPGA, instead of executing the instructions of the loop. The code for initializing the reconfigurable hardware includes an enable signal to W-FPGA, code to power-down the GPP into sleep mode and a jump to the instruction immediately after the end of the original software loop (skipping in this case the execution of the loop instructions by the GPP). When the W-FPGA finishes execution, it sends an interrupt which wakes up the processor and resumes its execution. The processor and W-FPGA execute in a mutually exclusive mode, i.e., only one of them can be executing at any given time. This simplifies access to data, avoiding data coherency and consistency issues. Furthermore, the authors refer that they have not found a significant advantage in parallel execution of both components for the tested cases. They also consider that only a single application (and single-threaded) is executing in the system.

For the experimental results with a hardcore CPU, they used two ARM7 processors, one as the GPP and one for the CAD module. The mapping algorithms needed, on average, 1.2 seconds to complete on a 40 MHz ARM7. They compared speedup and energy reduction of

critical regions for 15 selected benchmarks related to embedded systems. The applications considered are from NetBench [86], MediaBench [87], EEMBC [88], Powerstone [89] and their own on-chip logic minimization tool, ROCM [81].

When compared with a common FPGA (Xilinx Virtex-E), W-FPGA presents 1.5× faster clock frequencies and 25% less power. Overall, when compared to the execution of the benchmarks on an ARM7 at 100 MHz, the Warp Processor shows application speedups of 6.3× and energy reductions of 66%, on average. They identified memory accesses as the main bottleneck in the tested benchmarks.

For the experimental results with the soft-core GPP, they used two MicroBlaze processors [90], one as the GPP and one for the CAD module. They considered two FPGAs for implementation of the Warp Processor, a Xilinx Virtex-II Pro clocked at 100 MHz, and a Xilinx Spartan 3 clocked at 85 MHz. The mapping algorithms needed, on average, 11 seconds to complete the mapping of a single kernel. They compared speedups and energy reductions of critical regions for 6 selected benchmarks from EEMBC [88] and Powerstone [89]. When compared to the execution of the benchmarks in a single MicroBlaze at the same frequency as the corresponding Warp Processor, they present speedups of 5.1× and 5.9× on average, at 100 MHz and 85 MHz, respectively. They note that the higher speedup of the Spartan3 is due to the lower operating frequency of its base case. Of the six benchmarks used, one (*brev*) had a much higher speedup than the others. This happened because the critical kernel of the *brev* benchmark has intensive bit-manipulations which map very efficiently on an FPGA. Without considering this benchmark, the speedups are 3.3× and 3.6×, on average, at 100 MHz and 85 MHz, respectively.

The energy consumption depended on the dynamic partitioning scenario. In a scenario where a significant portion of the execution runs on the FPGA, and the changes between the GPP and the FPGA are infrequent, they present energy reductions of 65% and 55% for 100 MHz and 85 MHz, respectively. In a similar scenario, but where the changes are continuous, they present energy reductions of 55% and 24% for 100 MHz and 85 MHz, respectively. They justify the lower performance on energy of the Spartan3 to its lower static power dissipation. Since it is significantly lower than the static power dissipation of the Virtex II-Pro, the dynamic power dissipation of the MicroBlaze, which runs the CAD tools, represents a much higher overhead in the case of the Spartan3.

Finally, the authors compared the Warp Processor using the MicroBlaze with existing hardcore processors for embedded systems. For the Warp Processor, they only considered the

Spartan3 implementation, since they considered that the Virtex-II Pro dissipates too much power for embedded systems (e.g., it often exceeded 1W). For the hardcore processors, they considered a set of ARM processors (ARM7, ARM9, ARM10, and ARM11). For the same set of benchmarks, the Warp Processor with a MicroBlaze at 85 MHz had energy consumption comparable to an ARM10 at 325 MHz but executed 1.5× faster on average.

3.3.2 CCA

The Configurable Compute Accelerator (CCA) [91] is a special-purpose unit for executing complex instructions. It was designed to be integrated in the pipeline of a GPP (see the block diagram of Figure 3.4). However, instead of predefined special instructions, it executes arbitrary Data-Flow Graphs (DFGs). Also, instead of directly accessing the CCA through programming, the unit itself has hardware support for binary translation, which automatically moves code from the instruction pipeline to CCA.

Generally, a CCA consists of a 2-D array of simple functional units (FUs) interconnected in a feed-forward manner. The implementation of a CCA, such as other RPUs, has a fixed number of FUs and a fixed organization, but the operations and connections between the FUs are configurable. There can be many CCA implementations, depending on the target domain. Using results from previous work efforts [92, 93] and information from benchmark profiling, the authors propose the CCA shown in Figure 3.5 [91]. It is a triangular shaped matrix of FUs, where the FUs in any given row are homogeneous, and alternately, each row supports either arithmetic and logic operations, or only logic operations. Between each two adjacent rows of FUs, there is a crossbar for communication. This particular CCA presents some constraints: it is limited to 4 inputs and 2 outputs, does not support memory operations (e.g., load/stores), and the output of an FU can only be used as the input of an FU in the adjacent row.

The objective of CCA is to execute small clusters of simple instructions as one macro-instruction. To *detect* which portions of code should be moved to CCA, the mapping system performs *subgraph discovery*. To perform subgraph discovery, the instructions need to be transformed into a DFG representation first. Then, they run a subgraph discovery and selection algorithm in the resulting DFG, which substitutes clusters of the nodes using basic instructions (e.g., ADD, XOR ...) with macro-instructions which can be executed in CCA.

The authors studied the use of the CCA of Figure 3.5 (which has depth 4) as well as other CCAs [91]. They discovered that, for the selected benchmarks, 99.47% of the graphs could fit in a CCA with depth 7 or less. However, CCAs with lowers depths are attractive because they

have lower latencies. They have also considered several CCA implementations with depth 4, a case which can handle 82% of the graphs.

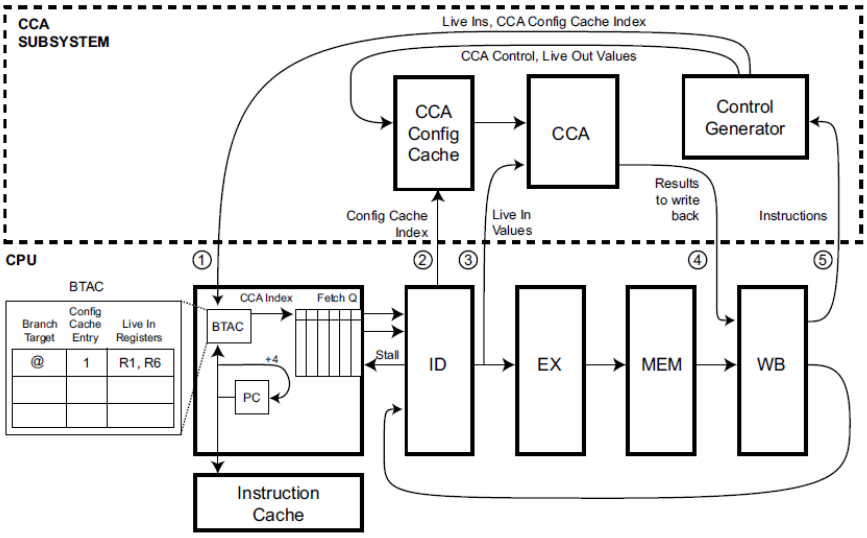


Figure 3.4. CCA-Enabled Processor Block Diagram (source:[53]).

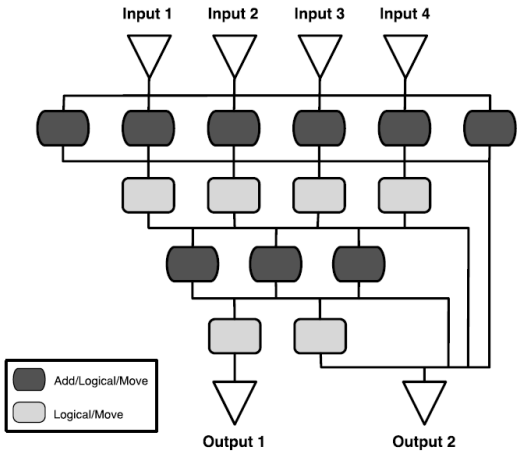


Figure 3.5. Example of a CCA Implementation (source:[53]).

The CCA approach proposes two methods for subgraph discovery: (a) using an optimal algorithm during compilation (static), and (b) using a heuristic during instruction retirement, in a trace cache (dynamic). The static method is employed offline, using code profiling and an optimal subgraph discovery algorithm developed by the authors and based on previous work [75, 94]. After detection of the graphs, the compiler modifies the binary so that the clusters of instructions which were chosen as good candidates for CCA graphs, i.e., clusters of

instructions that form a graph candidate to be mapped to CCA, can be easily *identified*. Initially, the authors used two new ISA instructions, CCA_START(*liveout, height*) and CCA_END to surround the cluster of instructions. In a subsequent work [53], they discarded those two instructions and instead encapsulated the cluster of instructions in a subroutine, and called it with a special instruction (BRL'). The authors refer that in case the binary needs to run in a processor which does not have a CCA, the special instructions CCA_START and CCA_END can be converted to NOPs, and the BRL' instruction can interpreted as a normal "branch and link".

For the dynamic *detection* approach, the authors propose a simpler algorithm for graph discovery [91]. Instead of doing an optimal search, which is too time-consuming for runtime, the algorithm uses a heuristic. Starting at a seed node, the graph grows upwards, towards the parent nodes. Each time a parent node is added, the new graph is considered as a mapping candidate. If adding a parent node violates CCA constraints, the node and its parents are discarded. When transforming the binary instructions to a graph, each operation is associated with a 'slack' value. A lower slack value of an operation represents a less critical operation to the dependence height of the DFG. The slack value is used to choose between multiple parents (lower slack values take priority). As the discovery of the graphs to map is done bottom-up, starting at a single node and growing up through its parents, the graphs will resemble the upside-down triangular structure of CCA.

The heuristic is applied in the instructions of a special trace cache, implemented using the rePlay framework [95]. This particular trace cache is called a *frame*[5] *cache*, which is similar to a trace cache, but is built upon predictions on the branches of several basic blocks. While the program runs, the *frame cache* builds the *frame*. After the *frame* is built, if at any point during execution any of the predictions happens to be wrong, the *frame* is discarded. This way, a *frame* can transparently cross basic block boundaries.

From the two approaches for *detection* that authors initially considered the, i.e., static and dynamic, they concluded that the *frame cache* requires a large amount of resources and power, which caused dynamic subgraph discovery using a *frame cache* prohibitive for embedded systems. In a subsequent work [53], focused on embedded processors, they

[5] A *frame* can be seen as a large basic block, with one entry-point and one exit-point. Branches inside the frame are converted to control flow assertions, and if one these assertions is triggered, the entire frame is discarded.

propose a general architecture framework for connecting any kind of CCA to a GPP, as well as a dynamic partitioning approach where the *detection* phase is done offline, during compilation.

Clark *et al*. [91] propose three possibilities for the execution stages where the *Replacement* and *Translation* can be done: during instruction decoding, inside the frame cache, and during instruction retirement. They concluded that the first case has low hardware overhead, but as it is done in the decode stage we are severely restricted by the stage latency. Moreover, it does not allow the crossing of basic block boundaries. The experimental results show that, of the three considered cases, the first case is the approach with the worse performance [91]. The second approach allows a better performance, but has a higher hardware overhead [91]. A subsequent work [53] focuses on a combination of the third approach with additional information from a *detection* phase done during static compilation. As this case is the approach they considered the best solution for embedded processors, we will describe herein the mapping algorithm behind the third case.

The mapping algorithm takes a sequence of instructions, detected as a complex instruction for CCA, and generates the configuration bits of the corresponding subgraph which can be obtained from this sequence of instructions. Figure 3.6 shows how the algorithm translates a sequence of instructions into a CCA configuration. The BRL' instruction in the *Subgraph Code* box signals a new subgraph. This subgraph was previously detected by the compiler and respects a number of characteristics: the number of inputs (or *live-ins*) and outputs (or *live-outs*) have predefined limits; all memory operations inside the graph are relative to temporary values (i.e., spill code); the subgraphs may cross basic block boundaries by using downward code motion during compilation.

The algorithm uses a *Current Producer* table, updated at each step, and which maps for each register and at that given time point, which FU produced the most recent definition of that register. The algorithm reads each instruction of the subgraph code in sequence, one at a time. For each instruction, it checks its input operands. If a given register in the input operand of the instruction is not in the *Current Producer* table, it is added to the list of inputs (Step 1 and 3 in Figure 3.6). The placement of an instruction is determined by which FUs produce a result used by that instruction. If an instruction needs a result from a previous FU, the placement of that instruction has to be, at least, immediately below the FU with greatest depth. For example, the instruction in Step 4 (see Figure 3.6) depends on the result of FU B,

which is in the first row of CCA. Thus, the instruction has to be placed on the second row of the CCA. The output operand of an instruction is marked in the *Current Producer* table.

CCA does not support memory operations (i.e., load/store operations), but has a special table to support spill code elimination. It is guaranteed by the compiler that any load/store in a subgraph refers to a temporary value that will not be used outside of the subgraph. Every time there is a store inside a subgraph (Step 2 in Figure 3.6), the table stores the memory offset of the store, as a way to identify the store, and the FU which has produced the value to store. When a load happens (Step 5 in Figure 3.6), the algorithm uses the information in the table to identify which FU has the needed value and correctly update the *Current Producer* table.

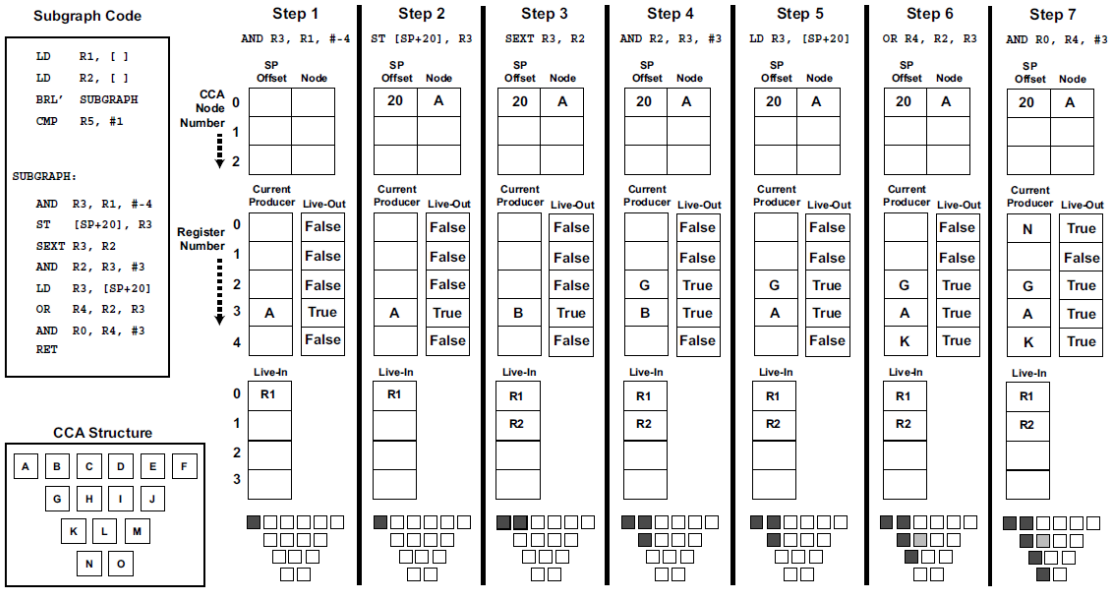


Figure 3.6. Mapping a subgraph into CCA (source: [53]): in the left are shown a sequence of instructions representing a subgraph code (top) and a CCA structure (bottom); in the right side of the *subgraph code* are shown the steps performed by the mapping algorithm.

As the CCA architecture is not pre-determined at compile time, the *detection* phase can extract subgraphs which will not map to a particular CCA. If a CCA does not have enough resources (e.g., FUs) to implement a particular subgraph, the mapping aborts and the sequence of instructions execute in the GPP.

After a subgraph is successfully mapped to CCA, *Replacement* is done by updating the entry of the corresponding branch on the Branch Target Address Cache (BTAC) represented in Figure 3.4. Branches to this location will trigger CCA execution.

For the experimental results, they present the speedups obtained using 29 selected benchmarks. The speedups were calculated as the ratio of execution cycles without and with

CCA. They use benchmarks taken from SPECint2000 [96] and MediaBench [87] repositories, and also include four encryption algorithms (3des, blowfish, rijndael, and rc4).

The results, achieved by simulation, revealed a maximum speedup of about 1.6×, and an average speedup of about 1.2×, with a CCA with 4 levels and using dynamic *detection* and *translation* [91]. When using the mixed static *detection*/dynamic *translation* of the most recent approach, they could improve the average performance to 1.60×, 1.91× and 2.79× for the SPECint2000, MediaBench, and encryption benchmarks, respectively, while using less hardware resources (in this case they do not use the *frame cache*) [53].

When comparing the first two approaches used for graph discovery, the authors concluded that the static *detection* was consistently better than dynamic *detection*. This was expected, since the static *detection* uses an optimal algorithm instead of a heuristic. However, the differences in the results obtained between the two approaches were minimal – both approaches achieve an average speedup between 1.2× and 1.3×. The authors explain the similar results by referring that they used an Instruction-Set Architecture (ISA) which has few registers (16). This increases the number of memory operations, and since CCA does not support them, it strongly limited the amount of computation which could be done in a single graph, as well as the exploration space of the static *detection* method. In the third approach, they exclusively used static *detection* and added support for spill code elimination, which contributed to the increase in the average speedup of the SPECint2000 and MediaBench benchmarks [53]. Furthermore, the results showed that CCAs with depths greater than 4 did not provide significant gains in performance [91].

3.3.3 DIM

Beck *et al.* [14] propose the Dynamic Instruction Merging (DIM) technique, a binary translation method to transparently move basic blocks from a general purpose MIPS processor to an RPU consisting of a coarse-grained reconfigurable array (CGRA). They tightly couple the CGRA to the processor: the CGRA works as an additional functional unit in the execution stage of the pipelining. They envision this architecture as a solution for accelerating embedded systems that need to execute many different kinds of computations.

The authors expect the DIM architecture to run at the same frequency as the processor. DIM (see Figure 3.7) has direct access to the contents of the register file of the processor through a set of buses and multiplexers. In the proposed CGRA, the processing elements are organized as a 2-D array (matrix) and only processing elements in adjacent rows

communicate directly. Each column of the matrix has only one type of element, existing three broad groups for the types of elements of the CGRA. The first group is composed of simple logic/arithmetic instructions which can be executed in less than a single clock-cycle. The second group includes memory load and store operations. Memory operations are assumed to have a delay equal to a cache-hit. If a cache-miss occurs, the FU stops until it is resolved. The third group is for complex elements which can take several clock-cycles to execute (e.g., multipliers). Although the authors refer in a previous work that the DIM supports loads and stores [14], in a posterior work [97] they show another architecture for DIM where the array has no external accesses and is composed by ordinary processing elements, such as ALUs, shifters and multipliers.

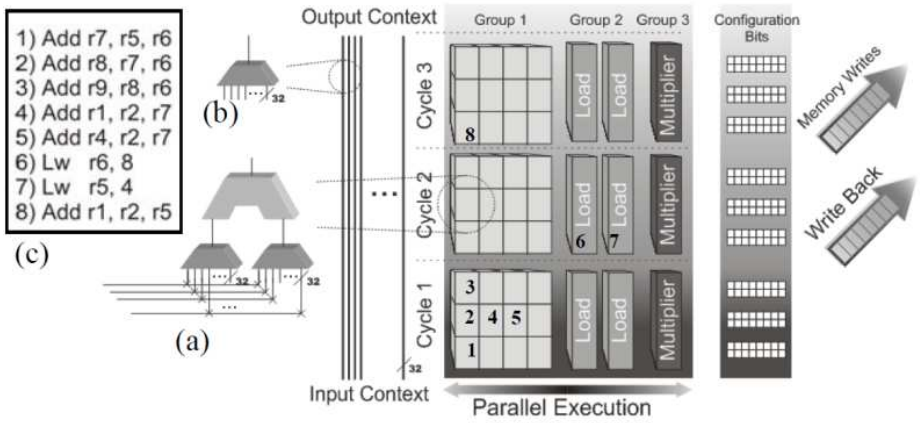


Figure 3.7. DIM Block Architecture and Configuration Example (source: [14]).

Figure 3.8 shows an overview of the *Replacement* system used in the DIM architecture. The instructions are read simultaneously by the processor and by the Binary Translation Unit (BT). They use a mixed *detection-identification-translation* phase: the first instruction after any branch is the beginning of a basic block, and is automatically considered for execution in the CGRA. Thus, after a branch, the binary translation hardware starts translating instructions to DIM. The translation is done instruction-by-instruction, similarly to the translation in CCA (see previous section). It continues until it finds an instruction not supported by DIM (e.g., a floating-point operation) or another branch. If the binary translator mapped a sequence of instructions with more than three instructions (a threshold chosen by the authors) the translation is stored in the Reconfiguration Cache and is indexed by the value of the Program Counter (PC) of the first instruction of the basic block.

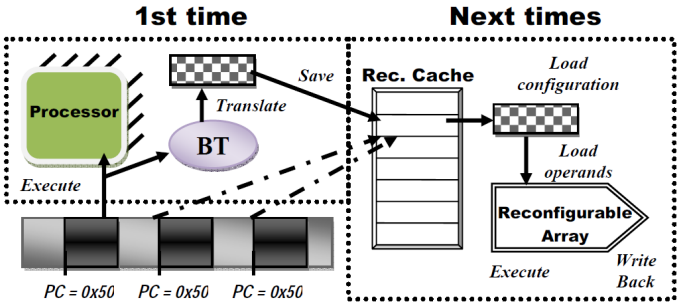


Figure 3.8. Dynamic translation in the DIM Architecture (source: [14]).

During normal execution of the processor, the address at the PC is read by the dynamic partitioning hardware while it is in the first stage of the processor pipelining. If the PC corresponds to an entry in the reconfiguration cache, DIM is reconfigured and executed in the fourth stage of the pipelining (Execution) instead of executing the instructions in the GPP. If DIM is not reconfigured at that time, the processor stalls until the reconfiguration is completed.

As previously referred, DIM uses an instruction-by-instruction translation algorithm similar to the one used in CCA. During *translation*, the algorithm keeps a number of tables where it stores information about the routing of the operands and the configurations of the processing elements.

For each incoming instruction, the first task is to check read-after-write (RAW) dependences using a *dependence table*. This table is built with the help of the array hierarchy of DIM. During translation, the last write to a register from an FU is known. With this information, the instruction is allocated and the *dependence table* updated. Finally, the routing is determined and configured.

The DIM authors claim that with this algorithm they can use larger windows for instructions, and consequently increase the ILP, when compared to the techniques used in superscalar processors [98]. It is also referred that the algorithm supports functional units with different delays and the handling of false dependencies.

The simplest *detection* approach used in DIM exploits exclusively the ILP inside basic blocks. As this ILP is limited, the DIM authors propose a speculative version of the DIM, which can cross the boundaries of up to three basic blocks. The speculative version uses a bimodal branch predictor [99] to decide if a given branch should be added to a certain DIM configuration. Each PC address is associated with only one DIM configuration. Since each

configuration only has one entry point, but can have several exit points (basic block branches), it can be considered as similar to the superblock [73]. When the branch predictor reaches a given maximum value, the instructions inside that branch are added to the current DIM configuration. If a given speculation misses a predefined number of times, the entire configuration is flushed out.

The DIM authors conclude that the performance of this approach, with or without speculation, is highly dependent of the number of instructions of each basic block. The more instructions a basic block has, the more instructions can be mapped to the CGRA and the higher the performance achieved can be.

For the experimental results, they use as GPP the Minimips [100], a processor based on a MIPS R3000, and estimated DIM power dissipation and area assuming a 0.18 μm CMOS process. They present the speedups for MiBench [101] benchmarks, a suite of benchmarks specific for embedded computing. They have used the benchmarks that the architecture supported (e.g., benchmarks without representative floating-point computations).

They claim an average speedup of 2.5× and an energy reduction of 1.7× [14] when using a CGRA with 48 rows and speculation enabled. Without speculation, they obtain an average speedup of about 2×. According to the DIM authors, the speedup comes from executing operations in parallel, and executing a small sequence (e.g., 2, 3) of simple operations (e.g., arithmetical and logical) in a single clock-cycle. They refer that, although the average power dissipation per clock cycle with and without DIM is similar, the DIM version needs fewer cycles to execute and consequently, consumes less energy.

3.3.4 Overview

All three approaches presented before, i.e., Warp [13], CCA [91], and DIM [14], show speedups for a number of benchmarks. Those approaches considering power dissipation also show energy reductions when using the RPU coupled to the GPP vs. the use of the GPP alone. The three efforts approach dynamic partitioning in different ways. Table 3.1 summarizes a number of characteristics of those approaches.

Warp is the only approach of the three which uses fine-grained reconfigurable hardware (W-FPGA) as the target RPU for dynamic partitioning. Comparing to a coarse-grained, it trades-off higher flexibility in the circuitry that can be implemented with higher overhead. It is also the approach which needs a more complex partitioning stage.

Characteristics	Warp [13, 76]	CCA [53, 91]	DIM [14, 97]
Partitioning Approach	Identify and decompile original loops, dynamically translate loops to the RPU	Identify segments of instructions which can be executed as macro-instructions on the CCA	Identify as many instructions as possible, inside one or more basic blocks, to be mapped to DIM
Coupling	RPU loosely coupled to the GPP, both share instruction and data memory	RPU tightly coupled to the GPP, RPU integrated in the GPP pipeline	RPU tightly coupled to the GPP, RPU integrated in the GPP pipeline
Granularity	Fine-grained RPU (LUTs, MAC)	Coarse-grained RPU (ALUs)	Coarse-grained RPU (ALUs)
Partitioning	Monitors addresses of executed instructions for short backward branches, representing inner loops	Detects subgraphs formed by clusters of instructions: 1) dynamically, inside a frame cache or 2) statically, at compilation time	Starting at any instruction after a branch and considering a limited number (3) of basic blocks
Size of the segment of code to be mapped in a configuration	Inner loops with few tens of lines of code	From a couple to a dozen of instructions across basic blocks	1) a couple to a dozen of instructions inside a basic block or 2) across up to three basic blocks with speculation
Benchmarks	NetBench, MediaBench, EEMBC, Powerstone, in-house tool ROCM	MediaBench, SPECint, encryption algorithms	MiBench suite
Target Domain	General Embedded systems	General Embedded and General Purpose Systems	General Embedded and General Purpose Systems
GPP	1) ARM7 at 100MHz 2) MicroBlaze at 85MHz	1) 4-issue superscalar ARM 2) in-order 5-stage pipelined ARM (ARM-926EJ)	Minimips soft-core based on the MIPS R3000
Size of the RPU	14.2 mm^2 with 180 nm library (~852,000 gates)	0.61 mm^2 with 130 nm library	> 1 million gates
Average Speedup	1) 6.3× 2) 5.9×	1) 1.2× 2) 2.3×	1) 2.0× 2) 2.5×
Average Energy Reduction	1) 66% 2) 24% - 55%	n.a.	2) 1.7×

Table 3.1. Summary of characteristics for the three representative approaches: Warp, CCA, and DIM.

Both CCA and DIM are integrated in the pipeline of the GPP, while W-FPGA works as a coprocessor. In the case of the Warp Processor, the mapped regions of code have to execute for a longer time to compensate for the overhead. In fact, the Warp Processor is, among the

three, the only approach which considers entire loops, while CCA and DIM present speedups by only exploiting ILP using a small number of basic blocks and without considering entire loops. The Warp Processor also presents the highest speedups, not only because it moves entire loops to hardware, but also because the fine-grained structure can dramatically accelerate applications with intensive bit manipulation.

However, Warp does not consider loop pipelining, a technique which has been extensively studied [48, 67, 102-104] and proven to be capable of substantial increases in performance. The technique has been studied in the context of CGRAs [61] and static binary compilation [58], although to the best of our knowledge, loop pipelining in the context of a dynamic partitioning system is still unaddressed.

Both CCA and DIM use reconfigurable hardware as an additional functional unit of the GPP. Being tightly coupled to the GPP gives access to the processor's registers and to the exploration of fine-grained instruction parallelism. However, this also means that the reconfigurable architecture has to work very closely with the processor architecture and the RPU and the GPP have to be designed together. This tightly coupling also places the RPU in the critical path of the processor, and limits the amount of work the RPU can do.

All the three approaches have embedded applications as one of their targets. They couple the RPU to GPPs commonly found in embedded devices, and they use embedded-related benchmarks. Unfortunately, the overlapping among the benchmarks used in the three approaches is very small. Among nine different suites of benchmarks, only one (MediaBench) was used by more than one approach (Warp and CCA).

Warp and CCA can easily map portions of code outside basic block boundaries, but DIM does this in a very limited fashion. Both Warp and CCA analyze the code before translating it – as it is executed, in the first case, or in a trace cache [95] / statically by a compiler in the second case. DIM directly translates the instructions to the hardware without previous analysis of the code. This approach is much lighter in comparison, but makes crossing basic blocks boundaries more difficult.

The detection of the critical kernels depends on the size of the RPU. Since the Warp Processor addresses entire loops, it detects backward branches, which usually identify inner loops. CCA looks for multiple partitioning units inside the graph constructed from frequently executed portions of code. DIM does not have a mechanism for detection of critical kernels: it tries to map the currently executing instructions every time a branch occurs.

3.4 Summary

This chapter briefly described the more relevant works concerning dynamic partitioning. We gave examples of binary translation, RPU architecture and dynamic partitioning, as well as a special attention to three approaches closely related to our work: Warp, CCA, and DIM. The results from the three approaches reveal trends which provide a useful guide to our own research work. They focus on embedded systems which use RISC processors as GPPs coupled to an RPU. RPUs based on coarse-grained reconfigurable logic showed a trade-off between potential for speedup and partitioning overhead. These works also show that there are many options in a continuum between fully static approaches and fully dynamic approaches, worth of being explored.

It became clear that going beyond the basic block has a significant impact. Previous work has shown that the size of program sections to be moved can become greatly constrained if we do not cross basic block boundaries [92]. Memory operations are another significant constraint, and we should consider memory operations as supported RPU operations, thus enabling the mapping of candidate sections with memory accesses.

In general, all approaches could achieve speedups around 2×, on average. The highest performance improvement is reported in the work of Lysecky et al. [13], where they achieved speedups around 6× with mapping of bit-level operations. However, the applications with intensive bit-manipulation operations are limited to very specific fields (e.g., encryption).

4 The Megablock

The ultimate goal of our approach is to move sequences of instructions from the General Purpose Processor (GPP) to a Reconfigurable Processing Unit (RPU) during runtime. The sequences of instructions (code) to be moved (e.g., small groups of instructions, individual basic blocks, entire loops) is fundamental for an efficient Dynamic Hardware-Software Partitioning (DHSP) method (herein referred as dynamic partitioning). The units of code to be moved from the GPP to an RPU influence the RPU architecture, the potential for improvement, and the algorithms that can be applied during the entire process.

We focus our attention to code units considering loops, as they are commonly the ones which contribute more to the overall execution time of the application. Specifically, this chapter presents the repetitive pattern of code proposed in this thesis, the Megablock, and explains why it is well-suited for moving sequential code to RPUs with a parallel computing model. We present an algorithm for detecting Megablocks, an Intermediate Representation (IR), and source-to-source transformations which enable the formation of better Megablocks.

4.1 Motivation

The impact of a coprocessor in the overall execution time of a program is related to the portion of program execution moved from the GPP to the coprocessor. Consider a metric for measuring program execution, such as the number of clock cycles (also known as latency) in a processor with fixed clock frequency. When using dynamic partitioning methods, one expects to move parts of the execution from the GPP to the coprocessor. We use the term *coverage* to refer to the portion of GPP execution that will be replaced by execution in an RPU, over the total execution when the program runs only in the GPP (see Equation (4.1)).

Let us consider a hardware accelerator which can improve the execution time of the sequential code moved from the GPP to the coprocessor by a factor represented as $Speedup_{Hw}$. The overall application speedup one can achieve, according to a particular coverage, is given by Equation (4.2). The speedup given by this equation is an upper bound which does not take into account any overhead (e.g., communication overhead).

Figure 4.1a) shows overall application speedups according to the percentage of execution that is moved to the coprocessor for several values of $Speedup_{Hw}$. Figure 4.1b) presents another view of the same data, by showing the ratio between $Speedup_{Overall}$ and $Speedup_{Hw}$, according to coverage. A zero percent coverage corresponds to a speedup of 1×, while 100% coverage corresponds to a speedup equal to $Speedup_{Hw}$. Both figures show that the overall application speedup is limited by the amount of execution we move to the coprocessor (Amdahl's law [105]). If we consider that $Speedup_{Hw}$ is a very high value (e.g., infinite), we obtain Equation (4.3). According to this equation, with 50% coverage we can never attain a speedup great than 2×, to attain an overall 3× speedup we need more than 66.6%, to attain an overall 4× speedup we need more than 75% of coverage, etc. Thus, when moving computation from a GPP, it is very important to move large portions of the program execution; otherwise the impact of the coprocessor in the overall speedup is limited.

$$\text{Coverage} = \frac{\text{Execution}_{\text{Moved}}}{\text{Execution}_{\text{Total}}} \times 100 \quad \textbf{(4.1)}$$

$$\text{Speedup}_{\text{Overall}} = \frac{1}{(1 - \text{Coverage}) + \frac{\text{Coverage}}{\text{Speedup}_{\text{HW}}}} \quad \textbf{(4.2)}$$

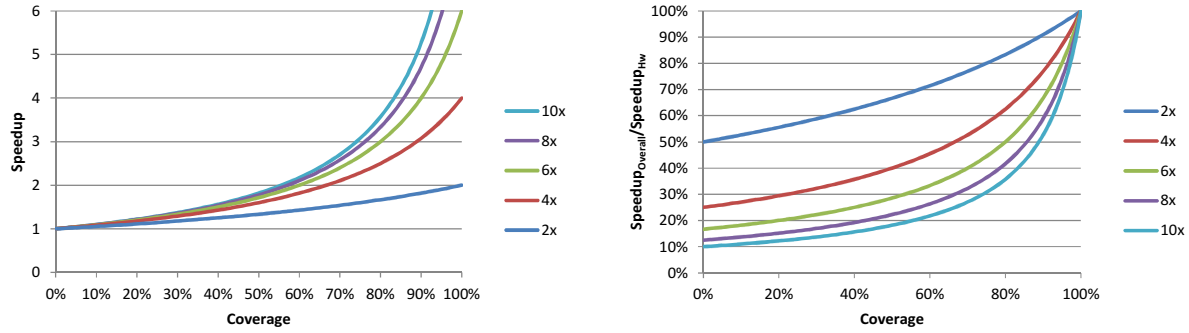
$$\text{Speedup}_{\text{Overall-Max}} = \frac{1}{(1 - \text{Coverage})} \quad \textbf{(4.3)}$$

High coverage is not the only condition for having a significant performance impact when moving computation from the GPP. For instance, the coverage can be distributed among many small trace segments, which may produce low improvements when communication overhead is also considered. However, high coverage is a necessary condition to achieve noticeable speedups (e.g., coverage of more than 50% is needed to achieve a speedup of 2×).

Approaches which move a set of instructions [53] or single basic blocks [14] can have high coverage. In this case, the impact is limited by what the coprocessor can do with that sequence of instructions. For instance, the speedup when moving a single basic block is mostly determined by its ILP, which is usually very limited [97].

An alternative to the case where a simple sequence of instructions is moved to the coprocessor, is to move entire loops [13]. Loops continuously repeat a similar sequence of instructions, increasing the potential for improvements when compared with a simple

sequence. A loop usually represents a considerable bigger portion of uninterrupted execution than, for instance, a single basic block.



a) b)

Figure 4.1. a) Upper bound for overall application speedup as a function of the coverage, and b) ratio between $Speedup_{Overall}$ and $Speedup_{Hw}$ as a function of the coverage.

Hardware implementing loops is usually coupled to the GPP according to the schemes in Figure 2.3a), b) and c). Moving computations away from the GPP enables coprocessors with more complex behavior, but increases communication overhead. However, as loops tend to execute for a longer time than a single sequence of instructions, they have a higher possibility to amortize the communication overhead. As loops generally have a higher potential for improvement than single sequences of instructions, we have chosen to explore a loop-based execution unit.

Figure 4.2 depicts the Control Flow Graph (CFG) for a possible inner loop. Each block can represent a unit of execution, such as a basic block. In this figure, after execution of block A, the execution can either continue to block B or block C. After block B or block C, the execution continues to block D. For each additional loop iteration the executions goes back to block A, otherwise the loop ends. The CFG represents the static behavior of the loop. However, without information about the dynamic behavior of the execution, we do not know with which frequency the two paths are taken, or if a particular path is taken at all.

The Warp Processor [13] implements loop-based dynamic partitioning. It detects small inner loops by analyzing short backward branches and retrieves the static description of the loop directly from the instruction memory. Structures such as the HyperBlock [106] also consider the static structure of a loop, but includes, for each path, the number of times it was taken until a certain execution point.

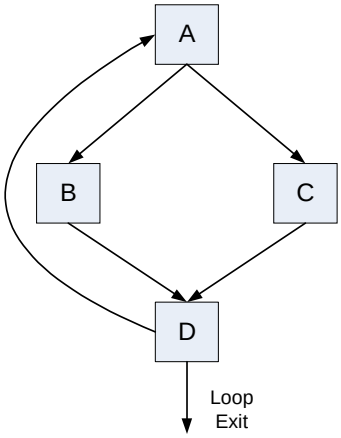


Figure 4.2. Example of the CFG of an inner loop.

The Dynamo system [72] takes another approach. Instead of starting from the static structure of the code, it builds a dynamic structure called a *fragment*, which is inspired by the superblock [73]. The runtime formation of a *fragment* starts when a basic block does not end with a backward branch (i.e., there is a forward branch, or the branch is not taken). A *fragment* ends when there is a backward branch at a branching point. Fragments represent a single execution path composed of several basic blocks. Previous work which considers the speculatively execution of *fragments* (i.e., assumes all the basic blocks in a *fragment* execute, and uses roll-back mechanisms when the assumed conditions are not met), showed improvements equivalent to the ones achieved when compiling with optimization flag –O4 of *gcc* [72].

The ideas behind the HyperBlock [106] , the superblock [73] and Dynamo [72] present good starting points for a dynamic loop structure which can be used to move code from a GPP to an RPU. We were very intrigued with the potential for improvement presented by the single path approach of the work done in the Dynamo system. Also, we consider that a repeatable execution path in sequential code can be interesting for a parallel computing model, and this has led us to focus on a loop structure, named as Megablock, with these characteristics.

4.2 Megablock Definition

The Megablock represents repetitive sequences of instructions in an execution trace. It typically represents a repetitive path formed during runtime. As with Dynamo's fragment, the Megablock also considers an execution path. However, the Megablock is strictly a loop

structure, which has a unit (the loop iteration) which repeats several times in sequence. The Megablock is also agnostic to the structure of the code and neither looks for jump instructions nor distinguishes between backward, forward or untaken jumps. Unlike the broader definition of a loop in Figure 4.2, the Megablock is a loop which continuously repeats the *same* sequence of instructions. A Megablock represents a contiguous repeating pattern in the execution trace, and a single execution of the pattern represents a single Megablock iteration. A definition of a Megablock is presented below.

Megablock Definition
Consider a statically defined program *P*, which is formed by the sequence of machine instructions $[i_1 i_2 \ \ldots \ i_m]$. Each execution of the program generates a sequence *T*, called a trace, formed with possibly repeated instructions from *P*. Consider S a sequence of instructions with size $m \geq 1$ present in *T* (being *m* the number of instructions). For instance, $[i_5 i_6 i_7]$ and $[i_8 i_2 i_3]$ are two specific three-instruction sequences. A Megablock is a contiguous subsequence of *T* formed by a repeatable sequence *S*, represented by *S{n}*, being $n \geq 1$ the number of times the sequence *S* repeats. E.g., if $S=[i_5 i_6 i_7]$ and *S{3}* is a Megablock found in *T* means that $[i_5 i_6 i_7 \ i_5 i_6 i_7 \ i_5 i_6 i_7]$ is a contiguous subsequence in *T*.

Megablocks have a simplified control-flow, where the same sequence of instructions executes in loop. In any point of the Megablock there can be guard instructions (referred herein as *exit points*), which test a condition to determine if the execution in the Megablock should continue. The conditions of all exit points are tested in all iterations of the loop. If any of the conditions fails, it signals the end of the execution of the Megablock. Consider Figure 4.3a), which contains the C code for the implementation of a *max* function. The function contains a *for* loop with an *if* structure, and forms a CFG similar to the one in Figure 4.2. A possible Megablock is formed during execution of the *max* function when the same path of the "*if*" is taken for several consecutive iterations (e.g., every iteration after finding the maximum value of the array).

Figure 4.3b) shows the sequence of MicroBlaze [90] assembly instructions that form the Megablock when the current value of the array is lower or equal than the maximum value up to that point (i.e., the expression "`v[i] > mx`" returns false). This Megablock contains two

exit points, represented by the branch instructions of the sequence (the fourth and the seventh instructions). The execution in the Megablock continues as long as the condition of the fourth instruction is met (corresponds to testing if the current value of the array is lower or equal than the maximum value up to that point), as well as the condition of the seventh instruction (which tests the induction variable of the loop).

```
unsigned int max(unsigned int* v,
int n) {                              1. 0x180 bslli r3, r4, 1026
  unsigned int mx=0, i;               2. 0x184 lw r3, r5, r3
  for (i=0; i<n; i++) {               3. 0x188 cmp r18, r3, r7
    if (v[i] > mx) {                  4. 0x18C bgeid r18, 12
      mx = v[i];                      5. 0x190 addik r4, r4, 1
    }                                 6. 0x198 rsubk r18, r4, r6
  }                                   7. 0x19C bnei r18, -28
  return mx;
}
```

Figure 4.3. a) C code for a *max* function and b) the MicroBlaze assembly code for a Megablock representing one of the possible execution paths.

4.3 Megablock Detection

The problem of detecting a Megablock is similar to an instance of the problem of detecting repeated substrings, e.g., *xx*, with *x* being a substring containing one or more elements. This is also known as *squares*, or tandem repeats [107]. In our case, substring *x* is equivalent to the previous sequence of instructions *S* (this can be achieved by representing each instruction by a symbol), and represents a single iteration of a loop. Although we want to find patterns with many repetitions (a square strictly represents only two repetitions), we observed that if a sequence of instructions forms a square, it is likely that more *x* elements will follow (e.g., *xxxx*…). The detection method considers that two repetitions are enough to declare the detection of a Megablock.

Our Megablock detection approach has been focused on schemes bearing in mind its suitability for runtime, either in the context of mapping or moving computations at runtime. There are algorithms which can find all tandem repeats in $O(n \log n + z)$, where n is the length of the string and z is the size of the output [108]. Another example is the use of linear time algorithms which uses suffix trees [109]. However, these algorithms are not suited to runtime

Megablock detection. Algorithms which use suffix trees need to preprocess the input string. Furthermore, as the stream of instructions is generated at a constant rate, the algorithm should have a constant processing time for each input, to be able to keep up with the GPP.

Figure 4.4 presents the algorithm developed to meet these requirements. The algorithm defines *a priori* the maximum size of the substring *x* (i.e., the number of pattern elements) in the squares.

```
M is maximum substring size
MatchingFifo has size M
CounterArray has size M, initialized to zero

processElement(PatternElement)
   for index 1 to M
      if PatternElement equals MatchingFifo[index]
         if CounterArray[index] < index
            CounterArray[index]++;
      else
         CounterArray[index] = 0;

   for index 1 to M
      if CounterArray[index] equals index
         signal match for square with substrings
         of size index

   insert PatternElement in MatchingFifo
```

Figure 4.4. Algorithm for detection of squares, up to a maximum size M.

It uses M counters, one for each substring size, from 1 to M, and a *FIFO* queue with read access to any index, which stores the previous M elements. When a new element arrives, it is compared with the M previous elements. The position in the *FIFO* of the previous element being compared determines the size of the substring being detected. If there is a match, the counter is incremented until the size of the substring. If there is a mismatch, the counter is reset to zero. For each counter, if there are as many consecutive matches as the size of the corresponding substring, a square with substrings of that size is detected. Finally, the element is inserted in the *FIFO*. When the *FIFO* is full, the oldest value is discarded.

According to the algorithm, when processing a single input, there can exist 1 to *M* matches for squares with different substring sizes. For instance, by feeding the pattern *aaaaaa* to the algorithm, after processing the last element it will detect 3 matches, for squares with substring sizes 1 (*a*), 2 (*aa*) and 3 (*aaa*), respectively.

We use an arbiter to select the most relevant match. For instance, to consider only inner loops, the priority is given to the match with the smallest substring size; to detect patterns with unrolled inner loops, but only when they appear inside outer loops (e.g., *aabaab*), the priority is given to the match with the highest substring size, but only if there is no match of a lower substring size simultaneously in the current and in the previous set of matches, to prevent unrolling in cases such as *aaaa*.

We consider four adjustable parameters when implementing a Megablock Detector: *maximum pattern size*, *type of pattern unit*, *unrolling of inner loops* and *executed instructions threshold*. In the algorithm we limit *a priori* the maximum number of pattern elements of the substrings that can be detected. The *maximum pattern size* refers to this size.

In the previous section, we indicated that the substring *x* is formed by one or more elements, and that *x* is equivalent to sequence *S* of the Megablock definition. Although we defined the elements of *S* (the contiguous repeated sequence) as single instructions, the elements of substring *x* can be coarser than instructions, to reduce the detection problem size. We refer to the kind of element used for detection (e.g., instruction) as the *type of pattern unit*.

Different kinds of units can be used for detection, as long as the pattern unit represents a contiguous subsequence of instructions in *T* (with *T* being the sequence of instructions that form the execution trace). For instance, in this thesis we consider instructions, basic blocks and fragments as possible detection units, as all of them represent a valid contiguous subsequence (or substrings) in the execution trace. Consider the example in pseudo-code in Figure 4.5. The Megablock ‘z’ can be formed by the instructions with addresses 10, 11, 30, 31 and 20; by the basic blocks starting with addresses 10, 30 and 20; or by the fragments B and C.

According to the rules of the arbiter that receives the matches that happen each clock cycle, it can give priority to smaller or larger patterns (Megablocks with less or more pattern elements). When priority is given to larger patterns, we consider that *unrolling of inner loops* is active. Otherwise, if priority is given to smaller patterns, *unrolling of inner loops* is not active.

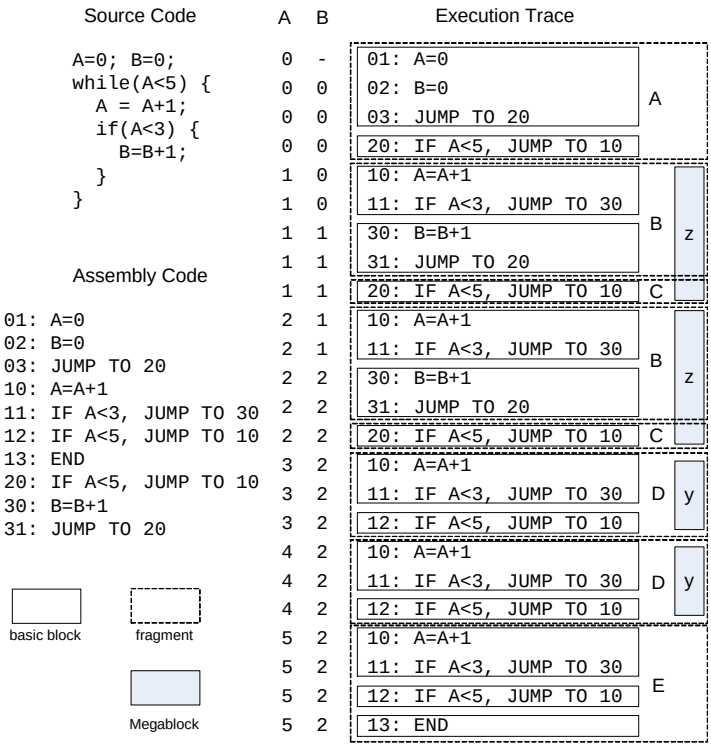


Figure 4.5. Program execution partitioning according to basic blocks, fragments, and Megablocks.

When a Megablock is detected, we are able to determine how many instructions the GPP executes until the Megablock exits. The *executed instructions threshold* refers to the minimum number of instructions that should be executed by the GPP (when a Megablock is detected) so that the Megablock is considered for implementation. If the number of executed instructions falls below the *executed instructions threshold* the Megablock is ignored.

4.4 Megablock Intermediate Representation

Intermediate Representations (IRs) are widely used in compilation as a way to express code in a more convenient way for transformations, mapping, and code generation [49]. The Megablocks are formed from instructions extracted from the execution trace of the processor used in the target system. Those instructions can be translated to a format similar to *three-address code* [49], an intermediate format commonly used when targeting GPPs. However, this format is ill-suited for computing models with intrinsic support to high parallelism degrees, as is usually the case with RPUs. Instead of translated to a *three-address code* format, the instructions of the Megablock are transformed into a graph representation, more akin to data-flow representations.

The proposed intermediate representation contains two data structures: a directed graph structure, which contains nodes and connections representing the relationship between data and operations, as well as additional information such as exit points; and a table which maps, for each output of the Megablock, which operation writes the last value, according to the original sequence of instructions.

The graph representation presented herein contains four kinds of nodes: *Operation*, *Livein*, *Constant* and *Exit*. It uses five types of connections: *data*, *control*, *liveout*, *feedback* and *exitAddress*. Figure 4.6 summarizes the available node types and the possible connections between nodes.

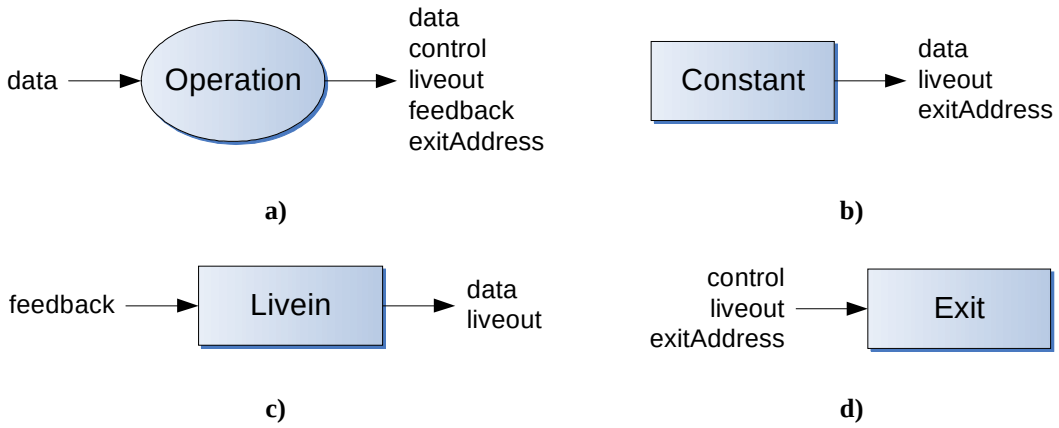


Figure 4.6. Types of nodes and possible connections in a Megablock graph.

The *Operation* node (see Figure 4.6a)), represents an operation of the graph (e.g., add, sub, mul). The *Constant* node (see Figure 4.6b)) represents an unchangeable, literal value (e.g., the integer value 100). The *LiveIn* node (see Figure 4.6c)) represents an external value which needs to be fetched before starting the Megablock execution. The *Exit* node (see Figure 4.6d)) represents an exit point of the Megablock.

There are five types of connections, described below. Note that certain types of connections include additional information represented herein with labels.

data: connections which represent the flow of data between outputs and inputs of operations. *Operation*, *Constant* and *Livein* nodes can be sources of data, but the *Operation* node is the only node which can be a sink of a *data* connection. *Constant* nodes are, by definition, unchangeable and cannot be sinks. *Livein* and *Exit* nodes can receive data from other nodes, but special types of connections are used to indicate what kind of data is being transmitted.

We use herein a label in the format "`OUT:IN`" for each *data* connection. `OUT` is the output index of the source node and `IN` is the input index of the destination node. When the source of the connection is a *Livein* or a *Constant* node, the `OUT` value is left blank (i.e., the label becomes "`:IN`"). For instance, in Figure 4.7 we have two *Operation* nodes connected by a *data* connection, which indicates that the output 0 from *Operation* 1 connects to the input 1 of *Operation* 2.

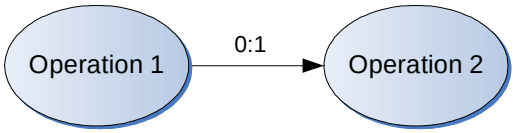


Figure 4.7. A data connection between two operation nodes.

control: boolean value (represented as 0 or 1) from an *Operation* node which indicates if an exit point is triggered or not. Only *Exit* nodes can be sinks of *control* connections. Each *control* connection includes a label in the format "`OUT`", where `OUT` is the output index of the source *Operation* node.

liveout: data connection which represents the value for one of the outputs of the Megablock, for a particular exit. Only *Exit* nodes can be sinks of *liveout* connections. Each *liveout* connection includes a label in the format "`OUT:SYSTEM_VAR`", where `OUT` is the output index of the source node and `SYSTEM_VAR` the name of the system variable to be updated. For instance, the name REG2 can represent the second general purpose register of the main processor. If the source node is a *Constant* or a *Livein* node, the value of `OUT` is left blank.

feedback: data connection which represents internal updates to the values which were initially fetched before Megablock execution started. Only *Livein* nodes can be sinks of *feedback* connections. Each *feedback* connection includes a label in the format "`OUT`", where `OUT` is the output index of the source node. If the source node is a *Constant* node, the value of `OUT` is left blank.

exitAddress: when processing Megablocks, in most cases it is possible to calculate, before Megablock execution, from which instruction address the processor needs to resume execution, after an exit point of the Megablock. However, it can be the case that the address can only be determined during Megablock execution. The *exitAddress* connection represents the instruction address from where the processor resumes execution, for a particular exit.

Only *Exit* nodes can be sinks of *exitAddress* connections. Each *exitAddress* connection includes a label in the format "`OUT`", where `OUT` is the output index of the source node.

4.5 Adapting Source Code to Megablock Detection

Megablocks can be detected in examples with control-flow in their loop bodies (causing the existence of branch instructions), if the same path consecutively repeats during execution. However, it can happen that no patterns ever form, due to the execution path being highly sensitive to values of input data. Even in the case where different paths of a loop are detected (corresponding to different Megablocks), if the paths themselves do not repeat enough times, Megablocks will execute a low number of iterations per call, which can lead to excessive overhead, possibly outweighing the benefit of the accelerator. This can prevent the use of Megablocks.

In this section we propose a set of rules for transforming source code with conditional statements into a straight-line code sequence, increasing the potential to detect better Megablocks. This transformation is commonly known as *if-conversion* in the compiler literature [29], and enables techniques such as *vector-mask control*, used to execute code with conditional execution in vector processors and GPUs. The benefit from the approach presented here is that it allows to perform *if-conversion* by doing source-to-source transformations, without additions to the language (e.g., pragmas) or modifications in the compilation tools.

4.5.1 General Definition of the Transformations

The main targets of the transformations are constructions of the type represented in Figure 4.8. We want to rewrite these sections so that the compiler writes straight-line code, as opposed to using branching instructions.

In their most general form, the structures in Figure 4.8 can be replaced by the corresponding structures in Figure 4.9. The equivalent code uses a ***mux*** binary operation, of the type *value* ***mux*** *condition*, where *value* represents any kind of data, and *condition* represents a boolean value. This operation returns *value*, if *condition* is true, or 0 if *condition* is false. In a later section, we will present concrete examples on how to implement the ***mux*** operation. The equivalent code in Figure 4.9 includes the boolean operators ***and***, ***or*** and negation (!).

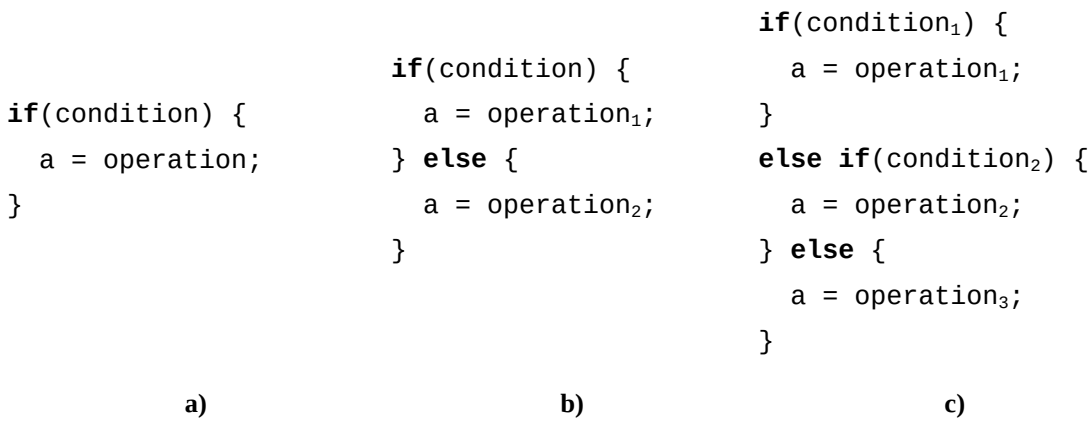


Figure 4.8. Examples of the target code subject to transformation: a) single if statement; b) if-else statement; c) a chain of if-else statements with arbitrary size.

a) `a = (operation mux condition) or (a mux !condition)`

b) `a = (operation1 mux condition) or (operation2 mux !condition)`

c) `a = (operation1 mux condition1) or (operation2 mux (condition2 and !condition1)) or (operation3 mux (!condition2 and !condition1))`

Figure 4.9. Equivalent code when applying *if-conversion* to a) single if statement; b) if-else statement; c) a chain of if-else statements with arbitrary size.

Each case can extend the examples of Figure 4.8 to have any statements as necessary, and the case in Figure 4.8c) can be extended to have as many conditions as necessary.

These transformations remove the branches because they force the loop, during each iteration, to execute the instructions of all paths (as opposed to execute only the instructions of a particular path). As such, when executing the transformed code in the GPP alone, the functionality is maintained, but generally the execution time will increase. However, when executing the transformed program in a system with support for dynamic partitioning, moving the new found Megablocks to an RPU can reduce the execution time, when compared to the original, unmodified program.

4.5.2 C Transformations Targeting the MicroBlaze Processor

Since we are neither modifying the source code language specification nor the compilation tools, the implementation of the technique is dependent on the target environment, and needs to be adapted to each particular case. In this section we provide transformation examples

when considering C as the source language, and targeting the MicroBlaze processor with the *mb-gcc* 4.1.2 compiler. Figure 4.10 shows how to calculate the term *condition* for the example `a cond b`, where *cond* is a comparison operator (i.e., >, <, >=, <=, == or !=). The transformation relies on the compiler resolving the condition to a boolean value without having to use branch instructions. This happened consistently in the tested cases when the comparison was done between a variable and zero. If the comparison with zero is done with an expression, instead of a variable, the compiler still uses branch instructions in some cases (e.g., when using expressions with more than one variable).

a)
```
temp = a-b;
condition = temp cond 0;
```

b)
```
asm("cmp %0,%1,%2": "=r" (temp): "r" (b), "r" (a));
condition = temp cond 0;
```

Figure 4.10. How to calculate the term *condition* in C using a) plain C and b) inline assembly, when targeting the MicroBlaze processor.

The example in Figure 4.10a) works when the values of `a` and `b` are signed values, and their values are such that during the subtraction an overflow/underflow never occurs. For a general case, we use the example in Figure 4.10b), which inserts the MicroBlaze assembly instruction `cmp` (`cmpu` when the comparison is between unsigned values).

Figure 4.11 shows the expression in Figure 4.9a) using two possible implementations of the ***mux*** operation written in C. The most straightforward implementation is to implement the operations as a multiplication (see Figure 4.11a)). However, multiplication may become too expensive to be used if there are many *mux* operations in the transformed code.

a)
```
a = (operation × condition) | (a × !condition);
```

b)
```
condition = ~condition + 1;
a = (operation & condition) | (a & ~condition);
```

Figure 4.11. Applying *if-conversion* to a single if statement in C, when the *mux* operator is a) a multiplication and b) a logical *or*.

Figure 4.11b) transforms the term *condition*, by inverting it and adding one. If the value of the term is zero, this transformation returns zero. However, if the value is one, the

transformation will set to one all bits of the term. After the transformation we can use the *bitwise and* operator (&) instead of a multiplication. Notice that now the *bitwise not* operation (~) is used, instead of the *logical not* operation (!).

The second approach seems to be more indicated for hardware implementation, as it uses simpler operations. However, it can result in a longer critical path when compared with the first approach, depending of the latency of the multiplication. Additionally, if after obtaining the intermediate representation graph we can detect that one of the operands of the multiplication is a boolean value (0 or 1), we can modify the graph and replace the multiplication by a mux operation (see the *mul. to mux.* transformation, described in Section 5.1.4).

4.6 Summary

In this chapter we described the Megablock, the loop structure we propose for moving instructions from a GPP to an RPU. We presented the characteristics we find desirable when selecting a portion of code to move to the RPU and suggested the Megablock as a candidate.

When comparing to the partitioning approaches presented in Chapter 3 (see Table 3.1), the Megablock differs from them as it represents repetitive patterns of code in the trace of the executing program, possibly representing a loop in the original code. For small loops, we expect the instructions covered by Megablocks to be on par to the instructions covered by the partitioning method of the Warp processor [13] (the other approach which considers loops). However, the Megablock has the potential to include nested loops, recursive calls, and loops formed with irregular constructions such as *gotos*. In addition, as the Megablock is built using segments of instructions forming an execution path, it allows for dynamic optimizations aware of information known during runtime, as opposed to approaches which rely only on the static structure of the code.

We presented an algorithm for Megablock detection, with suitable characteristics for runtime application, and suggested a graph intermediate representation for the Megablocks. Finally, we proposed a methodology which can be used to increase the quality of detected Megablocks without modifying the compilation tool flow, by applying source-to-source transformations.

5 Transforming and Implementing Megablocks

This chapter presents practical aspects related to the implementation of Megablocks. It explains how to build the Intermediate Representation (IR) introduced in the previous chapter, and proposes a set of transformations which can be applied over the IR.

The *Detection* of Megablocks can be done offline, during a profile phase. However, even in that case, for dynamic partitioning we need a method to identify these previously detected Megablocks at runtime, when the application executes. We propose two methods for runtime Megablock *Identification*, Single Address Identification (SAI) and Megablock Signature Identification (MSI).

We present several architecture models capable of implementing Megablocks, and explain how we can augment a Megablock-enabled architecture to support pipelining of Megablocks.

5.1 Graph Transformations

This section explains how to transform assembly instructions, such as the ones used by the MicroBlaze processor, into the IR presented in Section 4.4. Before mapping segments of executed instructions such as the Megablocks to a coprocessor, we can apply several transformations over the segments, e.g., to expose more ILP, and/or to reduce the number of instructions to map.

5.1.1 Mapping MicroBlaze Assembly to Graph IR

As in the experiments we use the MicroBlaze processor [90] as the target GPP, the examples in this section show how to convert a Megablock formed by MicroBlaze assembly instructions into the graph IR. Note, however, that a similar approach can be applied to other processors.

The first step is to extract information from each MicroBlaze instruction in the Megablock. For each instruction, we store information about the instruction address, the operation to be performed (i.e., *opcode*) and its operands. For each operand, it is

determined if it is an input or an output, the type (i.e., register or constant), and a label. If the operand is a constant, the label contains its literal value; if the operand represents a register, the label identifies the register.

Table 5.1 presents additional information extracted from the sequence of instructions, used during the construction of the graph.

Variable	Description
BranchTaken	If instruction represents a branch/jump, indicates if the branch/jump is taken or not during Megablock execution.
IsExit	Indicates if the instruction represents an exit point in the sequence of instructions (takes delay slots into account).
NextAddress	The address of the next instruction that will be executed in the sequence. If instruction is a branch/jump, indicates the address after delay slots.
NoJumpAddress	The address of the next instruction. If the current instruction is a branch/jump, indicates the address after delay slots, considering that the branch/jump is not taken.

Table 5.1. Additional information acquired from the instructions in the Megablock sequence.

After all information is collected, the graph can be built by considering the instructions according to the Megablock sequence. Each instruction can originate zero, one or more graph nodes. For instance, the MicroBlaze instructions do not allow to represent 32-bit constant values in a single instruction (the maximum is 16-bits). The instruction *imm* is used to indicate the 16 upper bits of a 32 bits constant [90] in the next instruction. The *imm* instructions are not translated to operations, but its information is used to directly create 32-bit values in the intermediate representation. The other case where instructions do not generate an operation is when an instruction is detected to be a *nop*, (“*or R0, R0, R0”* is a default *nop* instruction in the MicroBlaze processor).

Most instructions generate one equivalent operation. Load/store instructions and some jump/branch instructions are exceptions. Load/store instructions are unfolded into an addition, which sums a base address with the offset, and a load/store operation.

For most jump/branch instructions, the information from the sequence of instructions in the Megablock is sufficient to calculate the destination address of the jump/branch. However, in some cases (e.g., instruction *rtsd*) the destination address depends on the runtime value stored in a register and needs to be calculated during the execution of the Megablock. In this case, the branch/jump instruction is unfolded into a comparison operation, which will test the exit point, and an addition which calculates the destination address.

With the above information it is possible to build a graph with information about the data connections between operations, and add the exit points of the graph. After all instructions are processed, the *feedback* connections (Section 4.4) are added to the graph. Finally, we build a table with information about which operations write the output values of the Megablock.

5.1.2 Constant Folding and Propagation

One possible transformation is Constant Folding and Propagation (CFP). With CFP, operations with only constants as inputs, or using registers whose values are determined as constants, are replaced by the result of the operation. This transformation can be applied to every operation node, and each operation node defines its own rules on how it should behave in the presence of constant inputs. For instance, arithmetic and logic operations (e.g., *add*, *sub*, *and*, *xor*) use the arithmetic and logic rules that correspond to their operation.

CFP is extended to other types of operations, such as the comparisons which control the exit points. If an exit point has constant operands, it can be determined that the alternative path represented by the exit will never be taken (e.g., exits created from branches which represent calls to/returns from functions). In such cases, the operation node and the exit point can be removed. This is how CFP, applied to Megablocks, can remove operations related to function calls.

Connections of the type *feedback* originate from operation nodes, but can only connect to nodes of the type *LiveIn*, which represent input values to be read before Megablock execution. If an operation replaced by the CFP has a *feedback* connection as output, the input *Livein* at the end of the connection can also be replaced by a constant value, removing an input value from the Megablock. Since *LiveIn* nodes represent inputs of the Megablock, CFP can propagate the constant value transmitted by the feedback connection by performing another pass over the nodes of the graph, and this step can be repeated every time new *Livein* nodes are replaced by constant values at the end of the pass. We named this step as Multi-Pass CFP.

Multi-Pass CFP cannot be always applied. The graph transformed by this technique assumes that when the Megablock starts executing, the GPP has previously executed at least as many iterations of the Megablock as passes performed by the CFP (to guarantee

that the *LiveIn* nodes have the input values calculated by CFP). This can be enforced or not, depending on the identification method used (see Section 5.4).

5.1.3 Identity Simplifications

Another transformation applied is Identity Simplification. It takes advantage of the identity property of some operations. Opportunities to apply this transformation can appear in graphs created from assembly instructions since it is common for compilers to use the identity property to implement attributions in assembly. For instance, a high level instruction such as `a = 10` can be implemented with an add instruction such as `add r4, 10, 0`.

5.1.4 Multiplication to Multiplexer

The Multiplication to Multiplexer *(Mul to Mux)* transformation, with an example illustrated in Figure 5.1, is a form of strength reduction, where an expensive operation is replaced with an equivalent, less expensive operation. When we determine that one of the operands of a multiplication can only have the values 0 or 1, the multiplication can be safely replaced by a multiplexer, which chooses between the value 0 and the other operand. Opportunities for this transformation can appear when *if-conversion* is applied to the source code (Section 4.5).

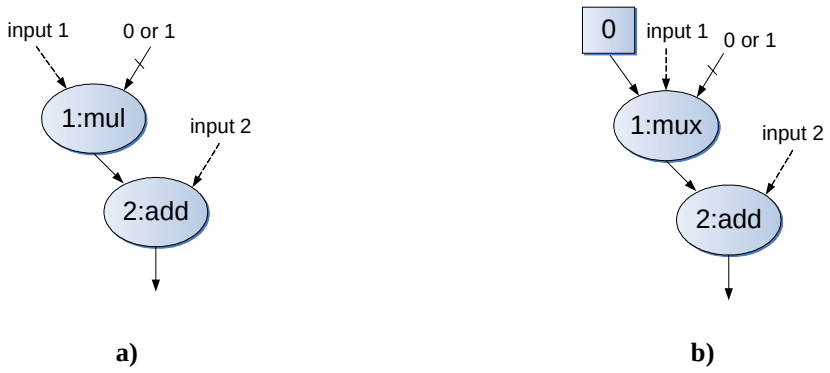


Figure 5.1. *Mul To Mux* transformation: a) graph before the transformation is applied; b) graph after the transformation.

5.2 Hardware Module for Megablock Detection

Figure 5.2 presents a hardware solution for Megablock detection, when using basic blocks as the detection unit. It has three main modules: the *Basic Block Detector* reads

the instructions executed by the processor, and detects which instructions correspond to the beginning of basic blocks. It outputs the instruction addresses corresponding to the beginning of basic blocks (signal *BB_address*), and a flag which indicates if the current instruction is the beginning of the basic block (signal *is_BB_address*).

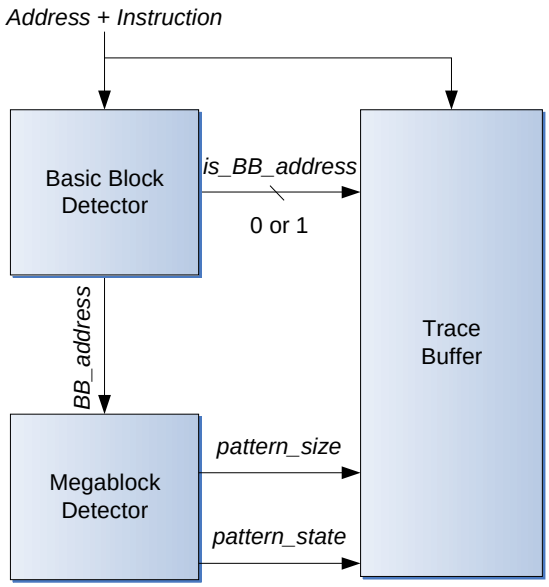


Figure 5.2. Hardware solution for Megablock detection.

The *Megablock Detector* receives pattern elements, which in this case is the first address of basic blocks. It outputs the size of the current pattern, or zero if no pattern is detected (signal *pattern_size*), and a control signal indicating the current state of the detector (signal *pattern_state*).

The module *Trace Buffer* is a memory that, when Megablock detection is active (i.e., the module is currently looking for Megablocks), stores the last instructions executed by the processor, their corresponding addresses, and a flag which indicates if the instruction corresponds to a pattern element of the Megablock (e.g., the start of a basic block). After a Megablock is detected, the *Trace Buffer* stops storing executed instructions and can be used to retrieve the detected Megablock.

Figure 5.3 presents the general diagram for the *Megablock Detector*. It contains three modules: the *Squares Detector* finds patterns of squares according to the algorithm presented in Section 4.3. It receives pattern elements and detects squares of size one up to a maximum, using as output a flag for each square size (*pattern_of_size_X*).

A pattern element can trigger one or more square sizes. The module *Pattern Size Arbiter & Encoder* receives the individual *pattern_of_size_X* flags, chooses which pattern size should be given priority and encodes the chosen size into a binary string. For instance, when detecting only inner loops, this module can be implemented as a priority encoder.

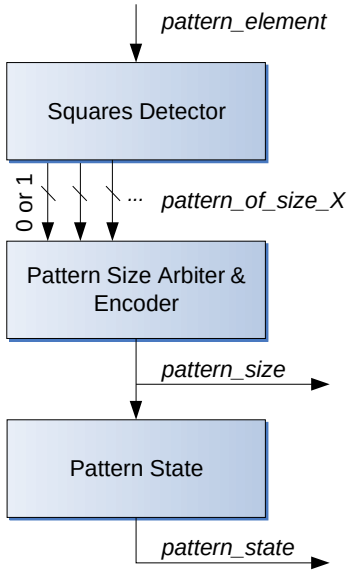


Figure 5.3. Diagram for the Megablock Detector.

The module *Pattern State* is a state machine which indicates the current state of the pattern, and can have one of five values: *Pattern_Started*, *Pattern_Stopped*, *Pattern_Changed_Sizes*, *Pattern_Unchanged* and *No_Pattern*.

Figure 5.4 presents the block diagram for a hardware implementation of the *Squares Detector*. The architecture in the figure can detect squares from size 1 up to 3, and can be easily extended to support larger square sizes. In the implementation presented here, the *pattern_element* signal corresponds to an instruction address.

Each detector for a specific square size (with exception of the detector for size one) uses a *FIFO*. When *FIFOs* have a reset signal they are usually implemented in hardware using Flip-Flops (FFs), becoming relatively expensive (a *FIFO* needs a number of FFs equal to the #bits × size of *FIFO*). However, if it is not necessary to access the intermediate values of *FIFOs*, they can be implemented with considerably less resources (e.g., if an FPGA has primitives for shift registers available). When using such *FIFOs*, the reduction factor in resources can be as high as 16 and 32 (e.g., when using the primitives SRL16 and SLR32 in Xilinx FPGAs, respectively) sizes [110, 111].

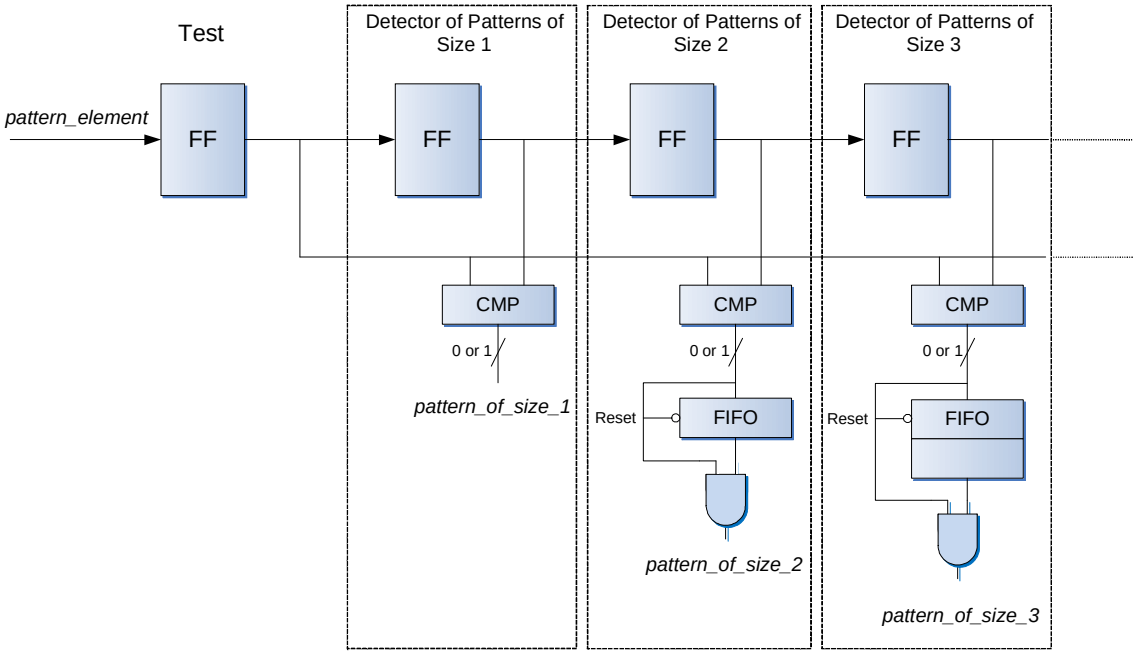


Figure 5.4. Diagram for a hardware implementation of the Squares Detector.

5.3 Megablock Translation using the Graph IR

Figure 5.5 presents a possible chain of steps for the *Translation* phase, where a set of assembly instructions representing a Megablock (e.g., the output of the Megablock Detector in Section 5.2) is transformed to an RPU configuration.

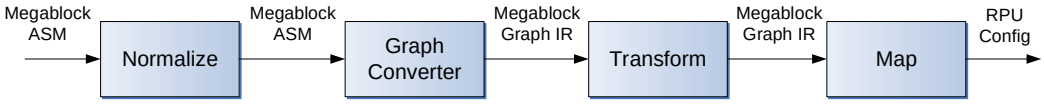


Figure 5.5. Possible chain of steps in a *Translation* phase.

Depending on the implementation, the chain of steps can be done during program execution (i.e., *online*), before program execution (i.e., *offline*) or in a mixed environment (i.e. a number of initial steps of the chain are done *offline*, and the remaining *online*). Likewise, each step in the chain can be implemented as a dedicated hardware module, or as a software program. For instance, in the implementation of a dynamic partitioning system presented in Appendix A, all the steps of the *Translation* chain are done *offline* and implemented in software.

Megablock implementations usually depend on the Megablock execution starting at a particular instruction, the *start instruction*. The step *Normalize* decides which

instruction of the Megablock is the start instruction. This step receives the assembly representation of the Megablock as input, and decides which instruction is considered as the start instruction.

Due to its repetitive nature, virtually any instruction of the Megablock can be used as the start instruction. In this work, we have used the following algorithm to calculate the start instruction: considering only instructions that correspond to pattern elements (e.g., first instruction of basic blocks), chose the one with lowest address, which is unique in the Megablock. If we cannot find a start instruction for a particular Megablock (e.g., all addresses appear more than one time, when considering the previous algorithm), the Megablock is not considered for mapping.

The step *Graph Converter* transforms the assembly representation of the Megablock into the graph intermediate representation (e.g., Section 5.1.1). The output of this step is a Megablock represented as a graph (i.e. the Megablock Graph). The step *Transform* applies transformations over the graph representation (e.g., Section 5.1.2 to Section 5.1.4). The output of this step is the graph representation of the Megablock, after applying transformations.

Finally, the step *Map* converts the graph representation into a configuration for the target coprocessor. The implementation of this step is highly dependent of the target architecture. In this section, we present a map algorithm which can be used for the architectures in Section 5.5.1 and Section 5.5.4. The mapping algorithm is divided in two parts, placement and routing.

Our placement algorithm has three steps, *buildDistanceGraph*, *addDependencies* and *rearrangeGraph*. Placement uses another graph representation of the Megablock, the *Distance Graph*, which can be built from a Megablock graph. The Distance Graph differs from the Megablock Graph in the following aspects: the placement takes into account timing constraints, so each node includes the latency of the operation it represents; since the graph is to be used to calculate the placement in architectures which can time-multiplex a design through several configurations, each node also includes information about the current configuration (i.e., *configuration level*) assigned to it (in row-based architectures, each configuration represents a row). Each connection between a source/sink node in a Distance Graph includes an additional parameter, *minimum distance*, which represents the minimum number of levels (i.e., rows) between two nodes (this value is usually dictated by the latency of the source node).

The first step of placement (*buildDistanceGraph*) is to build the Distance Graph from the Megablock Graph. Based on the latencies of each operation, to each node is assigned an initial configuration level which respect those latencies. At this point, the Distance Graph only has data connections. The second step (*addDependencies*) adds connections which represent dependencies between nodes (for instance, to serialize writes to memory, one can add connections between each store operation, representing the dependency). After including the dependency connections, the configuration level assigned to some operations may no longer be valid. We then apply the third step, *rearrangeGraph*, whose algorithm is represented in Figure 5.6. The function changes the configuration level values of each node so that they respect the existing connections and the given architecture constraints (e.g., the maximum number of memory operations per level/row).

```
"Constraints" contains architecture constraints

rearrangeGraph(DistanceGraph)
   CurrentLevel = 0
   while CurrentLevel <= getMaximumLevel(DistanceGraph)
      LevelNodes = getNodesFromLevel(DistanceGraph, CurrentLevel)
      NodesToMove = getNodesToMoveDown(LevelNodes, Constraints)
      for each Node in NodesToMove
         setLevel(Node, CurrentLevel+1)
         rearrangeNode(Node)

      CurrentLevel++;
```

Figure 5.6. Algorithm for the function *rearrangeGraph*.

The algorithm starts at the topmost level, and iterates over each level until there are no more levels. In each level, the nodes of that level are identified (*getNodesFromLevel*) and tested (*getNodesToMoveDown*). During the test, the nodes that do not respect the minimum distance indicated in the connections to their parents go immediately to the *NodeToMove* list. The remaining nodes are tested for architecture constraints. If there are not enough resources in the level/row for the remaining nodes, the outstanding nodes are added to the *NodeToMove* list. An algorithm can decide which nodes should

go to the list, and which nodes should stay in the level/row. The current implementation uses a first-come, first-served approach.

After deciding which nodes should stay on the current level/row, the other nodes are moved to the level/row below and each node is rearranged individually. Figure 5.7 presents the algorithm for the function *rearrangeNode*. When a given node does not respect the minimum distance between itself and a parent node, the node is pushed down until it reaches a valid position. After achieving a level which is valid to all parent nodes, the algorithm is recursively applied to the children nodes.

```
rearrangeNode(Node)
   for each ParentNode in Node
      NodeLevel = getLevel(Node)
      MinimumDistance = getMinDistance(ParentNode, Node)
      if  NodeLevel < MinimumDistance
         NewLevel = getLevel(ParentNode) + MinimumDistance
         setLevel(Node, NewLevel)
```

Figure 5.7. Algorithm for the function *rearrangeNode*.

For instance, consider the example in Figure 5.8. It represents a DistanceGraph created from a Megablock graph, after the first step in the placement algorithm. To each node was given an initial placement. The connections in the figure include the respective MinimumDistance. After adding node dependencies, the connection from the node in level 1 to the node *op* makes the placement of the latter invalid (underscored distance). Applying the *rearrangeNode* function to the node *op*, it is moved to level 3 (parent level (1) + minimum distance (2)). Since all connections from parents to *op* are valid, *rearrangeNode* is applied to all children of *op*.

After obtaining a valid placement for all nodes, the *route* algorithm presented in Figure 5.9 calculates the connections between nodes, and uses pass-through registers to communicate values between distant levels. It has one parameter, *MaxCommDistance*, which represents the maximum communication distance between levels/rows. A value of zero indicates an architecture which can only communicate between adjacent levels/rows.

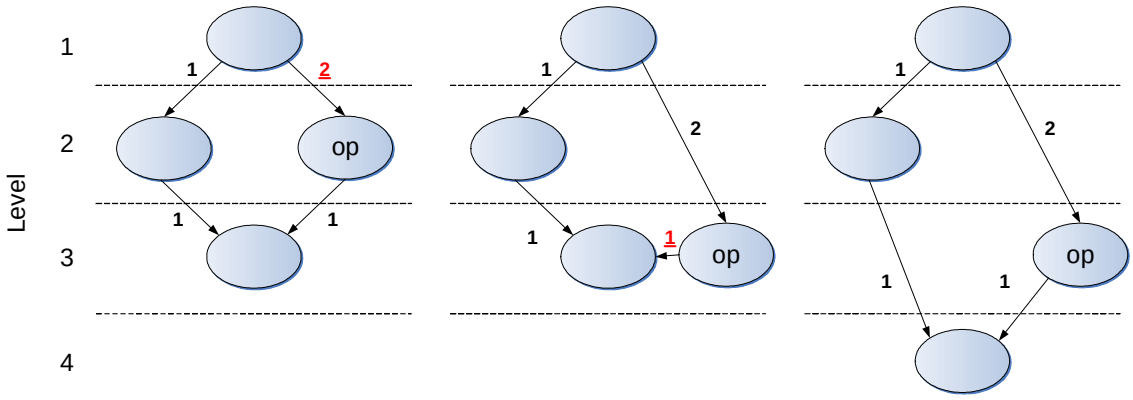


Figure 5.8. Example of the function *rearrangeNode*

```
route(DistanceGraph)

   for each Node in DistanceGraph
      for each ChildNode with data connection in Node
         NodeLatency = getLatency(Node)
         Distance = getLevel(ChildNode) – NodelLevel - NodeLatency
         if(Distance<=MaxCommDistace)
            addDirectConnection(Node, ChildNode)
         else
            usePassthrough(Node, Distance)

   MaxLevel = getMaximumLevel(DistanceGraph)
   for each OutputRegister in DistanceGraph
      OutputNode = getOutputNode(OutputRegister)
      NodeLatency = getLatency(OutputNode)
      Distance = MaxLevel – getLevel(OutputNode) - NodeLatency + 1
      usePassthrough(OutputNode, Distance)
```

Figure 5.9. Routing algorithm in the Map step.

5.4 Megablock Identification

After a Megablock has been detected for the first time, one can identify future calls to the same Megablock in the instruction trace. We propose two techniques for *Identification* of previously detected Megablocks: Single Address Identification (SAI) and Megablock Signature Identification (MSI).

SAI uses the address of the start instruction of the Megablock as the identifier. Megablocks are identified by examining the execution trace of the GPP, looking for the instruction address that matches the address of the start instruction assigned to the Megablock. As the Megablock identification with SAI is done just by examining a single address of the execution trace, there is no guarantee that the Megablock is executing when the address is detected.

MSI relies on using the Megablock instructions and their corresponding addresses to build a signature which uniquely identifies a Megablock. After identifying a signature, the method needs a synchronization period where it waits until the GPP executes the instruction which corresponds to the start address of the identified Megablock.

To build the signature, we can use any function which can generate a unique identification from the instructions and/or the addresses of the Megablock. For instance, we can use a hash function over the start address of each unit that forms the Megablock although, depending on the function used, the signature can be dependent on the start address of the Megablock (i.e., the result of the function is different depending on the start element of the sequence). To avoid this, we need to use a function which generates a signature from a list of inputs but whose result does not depend on which input is used as start. For instance, a sum of all individual addresses of the Megablock units respects this requirement. However, since the unit addresses are values which can be close to each other, it is common for this function to result in a high number of collisions. An alternative solution is to pass each address through a hash function [112], to introduce variation in the inputs, and sum all results.

Table 5.2 resumes most important characteristics of both methods. As in SAI the identification corresponds to the address of the start instruction of the Megablock, we cannot identify different Megablocks with the same start address as we would not be able to distinguish between them. Identification in MSI is decoupled from the address of the start instruction, and several Megablocks can have the same address. We can work around this limitation in SAI if the heuristic that assigns the start addresses takes into account which addresses have been used for previously detected Megablocks.

However, MSI needs to detect if the Megablock is executing before identifying it (for instance, with the help of the hardware module for Megablock detection introduced in Section 5.2). This introduces latency, as we need the Megablock to execute at least two iterations before it can be identified. And after identification, we need to

synchronize the execution, which can take up to a single iteration. SAI can identify a candidate at the moment the GPP asks its start instruction, although it is more prone to false positives.

Constant propagation with multiple passes assumes that the Megablock currently executing has run for at least as many iteration as the number of passes applied. As with SAI we do not have that guarantee, we can only use single pass constant propagation.

Characteristic	SAI	MSI
Identification	Start address of the Megablock	Signature made from several addresses
Address of Start Instruction	Same as identification. Only one Megablock for each address	Independent of identification. Multiple Megablocks can use the same address
Latency between Megablock identification and execution	No latency	At least 2 iterations
Constant Propagation	Single pass only	Up to multiple passes

Table 5.2. Characteristics of the proposed Megablock identification methods: SAI and MSI.

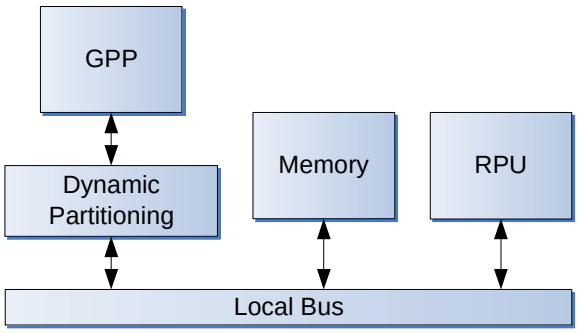
A possible implementation can use either method for identification of Megablocks, or include both methods. For instance, an implementation can use SAI as the default identification method, and use MSI when SAI is not able to identify a Megablock (e.g., when two Megablocks have the same start instruction).

5.5 Architectures for Implementing Megablocks

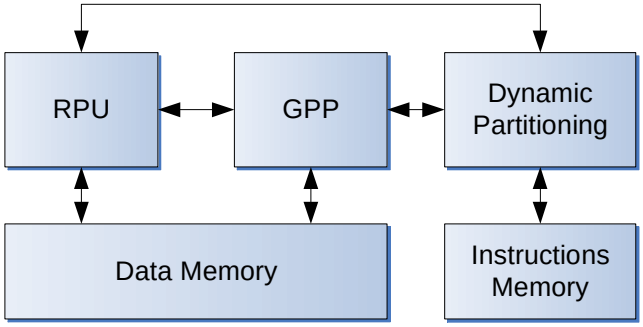
According to the coupling taxonomy presented in Figure 2.3 of Section 2.5, we present two general system architectures for implementing Megablocks. Figure 5.10a) shows an architecture with an RPU connected to the local bus, were all modules communicate through the same local bus. Figure 5.10b) shows an RPU coupled to the GPP. In this case all communication is done through dedicated channels. We do not consider the coupling in Figure 2.3a), an RPU coupled to the I/O bus, since we think it is very similar in implementation to the case in Figure 2.3b) but with potentially higher latency. We also do not consider the coupling in Figure 2.3d). We think the needed degree of integration of the RPU with the GPP is unsuitable for implementing Megablocks.

In both architectures we have a GPP, which will run the program, and an RPU which will execute the Megablocks. In Figure 5.10a), we consider that the GPP fetches

instructions through the local bus and that those instructions are intercepted by the Dynamic Partitioning module.



a) RPU coupled to a local bus



b) RPU coupled to the CPU

Figure 5.10. General system architectures for Megablock implementation.

The job of the Dynamic Partitioning module is to identify Megablocks in the instruction stream and handle the communication routines which exchange data between the GPP and the RPU. The module is also responsible for reconfiguring and starting the execution of the RPU and stall the GPP. The architecture of Figure 5.10b) is equivalent, but uses dedicated connections instead of a bus.

Equation (5.1) presents the general equation for estimating the overall speedup achieved by the architectures when using the RPU. CPU_{Cy} represents the clock cycles executed by the program when using only the GPP. The denominator of the equation considers the execution with the GPP and the RPU and divides the execution clock cycles into two parts: the cycles that belong to all calls to Megablocks ($MbCall_{Cy}$) and the cycles which are executed by the GPP ($CPU\text{-}Seq_{Cy}$). Equation (5.2) represents the clock cycles taken by a single call to a Megablock. The terms of these equations are

defined according to the specific implementations of the architectures of the system and RPU.

$$\text{Speedup} = \frac{\text{CPU}_{\text{Cy}}}{\sum \text{MbCall}_{\text{Cy}} + \text{CPU-Seq}_{\text{Cy}}} \quad (5.1)$$

$$\text{MbCall}_{\text{Cy}} = \text{RPU}_{\text{Cy}} + \text{Overhead}_{\text{Cy}} \quad (5.2)$$

5.5.1 General 2D CGRA

There have been several work efforts [14, 27] which transparently move computations from a GPP to a CGRA coprocessor with a 2D topology, as the one presented in Figure 5.11. Row-based CGRAs with forward communication have been used as targets for GPP computation [14, 113]. They use a simple communication scheme that significantly simplifies the routing phase.

Figure 5.11 shows the general architecture for a 2D RPU which can be used to implement Megablocks. It consists of a reconfigurable array with K rows of FUs (Functional Units) and forward communication between rows. The architecture contains an Iteration Control module, which will stop the RPU execution if an exit condition is activated. The FUs which can communicate with the Iteration Control module can be used to implement the operations that signal exits. The last row of the RPU is a row of output registers, which are updated with the iteration results if no exit signal is active. These registers are connected to the first row of FUs, which can use the results of the completed iteration in the next iteration.

This architecture executes the iterations of the Megablock *atomically*. If an iteration completes (i.e., there are no active exit signals after the execution of the last row) the results are committed to the Output Registers. Otherwise, the results of that iteration are discarded and execution in the RPU stops. Atomic iterations imply that when an exit point is activated during an iteration, the iteration is discarded and execution continues in the GPP at the beginning of the discarded iteration.

Before transferring execution to the GPP, the state of the system needs to return to the beginning of the last iteration. For instance, if all changes during RPU execution are restricted to internal communication inside the RPU, the state of the system is contained in the values of the Output Registers, which can be updated only if an iteration completes successfully. When an iteration fails, the values of Output Registers, which

currently have the results of the previous iteration, are not updated by the last iteration. These are the results communicated to the GPP. However, if the RPU changes the state of a memory, any change that occurred in the last iteration has to be reverted, which can imply a memory rollback mechanism.

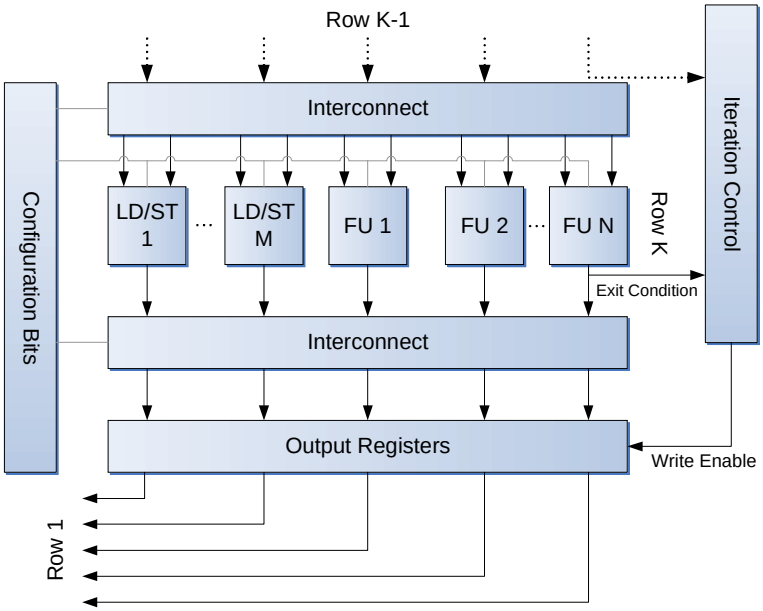


Figure 5.11. General architecture for a 2D CGRA-based RPU which supports Megablocks.

On the other hand, when iterations are atomic the architecture only needs to keep track of a set of output values, instead of a set per possible exit point. Furthermore, there are fewer restrictions when mapping the Megablock, the tools only need to ensure the results at the end of the iteration without the need to guarantee intermediate results; and when the execution returns to the processor, the instruction address where execution resumes is always the same, the address of the instruction that corresponds to the beginning of the Megablock iteration.

Equations (5.3) and (5.4) define the terms of the speedup equation (5.2) for the case were we model the latency in clock cycles. When an RPU based on the general 2D CGRA architecture is coupled to the system architectures presented in Figure 5.10. They represent the clock cycles needed for a single call to the RPU.

In equations (5.3) and (5.4), N_{It} is the number of iterations completed by the RPU. As iterations are executed atomically, the iteration where the exit point is activated will always be executed and discarded, which adds to the number of completed iterations. It_{Cy} is the number of clock cycles the RPU needs to complete an iteration. For instance,

if each row of the RPU takes one cycle to execute, the term corresponds without pipelining to the number of mapped rows. $\text{Communication}_{\text{Cy}}$ represents the clock cycles needed to communicate data to and from the RPU. It includes the communication of the values between the GPP and the RPU, as well as the RPU configuration bits. $\text{Partitioner}_{\text{Cy}}$ corresponds to the additional clock cycles needed by the dynamic partitioning system besides communication. Finally, since the parameter $\text{CPU-Seq}_{\text{Cy}}$ in equation (5.1) does not consider any part of the loop execution and we are considering an atomic execution of the iterations, the analytical model needs the parameter $\text{LastIterationInGpp}_{\text{Cy}}$, which represents the GPP clock cycles needed to execute the incomplete iteration discarded by the RPU.

$$\text{RPU}_{\text{Cy}} = (\text{N}_{\text{It}} + 1) \times \text{It}_{\text{Cy}} \tag{5.3}$$

$$\text{Overhead}_{\text{Cy}} = \text{Communication}_{\text{Cy}} + \text{Partitioner}_{\text{Cy}} + \text{LastIterationInGpp}_{\text{Cy}} \tag{5.4}$$

5.5.2 Specialized Array (SAr)

Megablocks can be translated to HDL descriptions and then synthesized to a reconfigurable fabric. Similar techniques have been previously used, for both offline and online scenarios. Kuzmanov *et al.* [114] extract kernels from an executable during a profiling phase. Those kernels are then processed and transformed offline into hardware descriptions and implemented using tools for FPGA-based hardware synthesis. The hardware implementations are then available during the execution of the program. Approaches such as Warp [13] propose an online hardware generation scheme which uses custom synthesis tools and custom reconfigurable fabrics.

Based on the general architecture for a 2D CGRA supporting Megablocks, depicted in Figure 5.11, we propose the Specialized Array (SAr), a specialization of the architecture for a single Megablock. Figure 5.12 presents two instances of the SAr for two different hypothetical Megablocks. Since the architecture only executes one Megablock, the functionality is fixed and does not have configuration bits. In the examples in Figure 5.12, the FUs are replaced by implementations of the Megablock operations, and the configurable interconnection resources are replaced by direct connections. Note that, depending on the implementation, the direct connections can

either be simple wires, or have *FIFOs* for synchronization of results. The execution is similar to what was described for the 2D CGRA general architecture in Figure 5.11. The iterations are executed atomically, and when the output of each operation is registered, the execution cycles of the SAr and the overhead are given by equations (5.3) and (5.4).

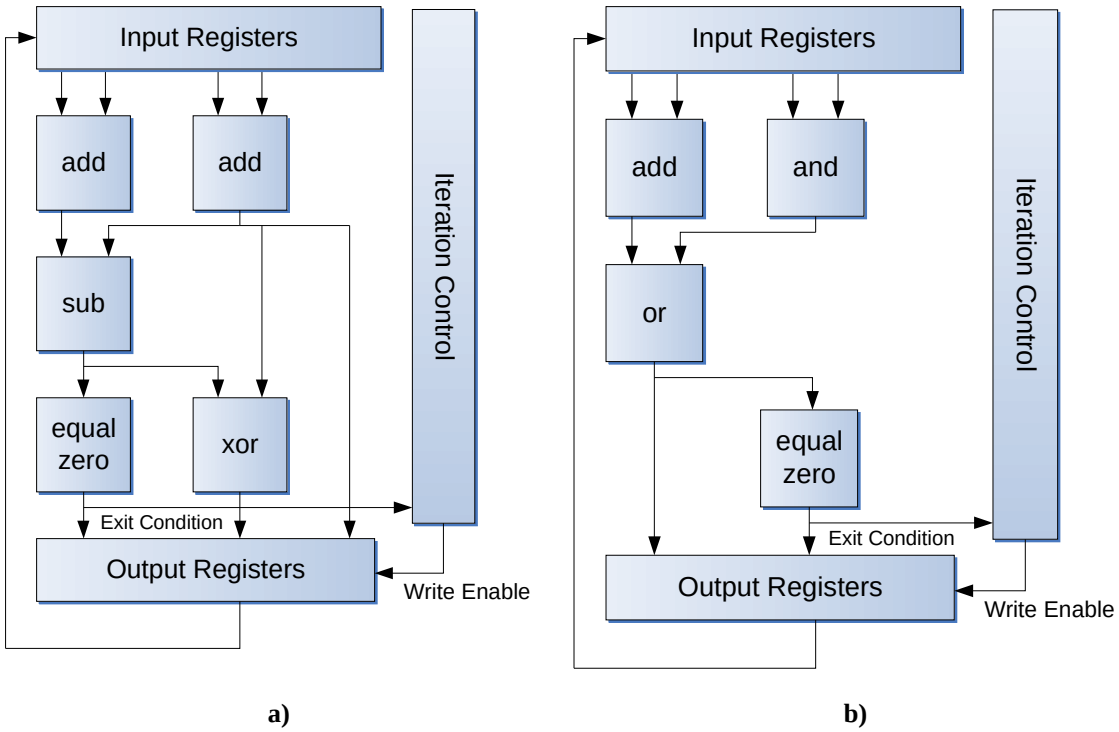


Figure 5.12. Two possible SAr instances for two distinct Megablocks.

5.5.3 Specialized Reconfigurable Array (SRA)

In the previous approach, a specialized module is created for each Megablock. However, as Megablocks are specific to a single program, and for any given program just one Megablock is executing at a time, only one of the hardware modules will be active at any given time. The Specialized Reconfigurable Array (SRA) merges individual Megablock implementations into a single runtime reconfigurable array. At any given time, the SRA can only execute one Megablock, but it can be reconfigured at runtime to execute any of the Megablocks it implements. The objective is to reduce resource usage and reconfiguration time (when compared with the general 2D CGRA) while providing an RPU with runtime reconfigurability (a validation of this approach is presented in Appendix A).

Figure 5.13 presents an instance of the SRA implementing the Megablocks depicted in Figure 5.12. The connections can be configured, according to the active Megablock. An implementation of this architecture can use direct connections for communication, as the SAr architecture example (presented in Figure 5.12), allowing several input wires to be multiplexed in the input ports of shared FUs. An alternative implementation can forward communication between adjacent rows (see Figure 5.13), using FUs to bypass values across rows (bypass FUs). The FUs marked with a "+" are reused between the Megablock configurations considered in this case. The execution clock cycles of the SRA and the overhead can be estimated with equations (5.3) and (5.4).

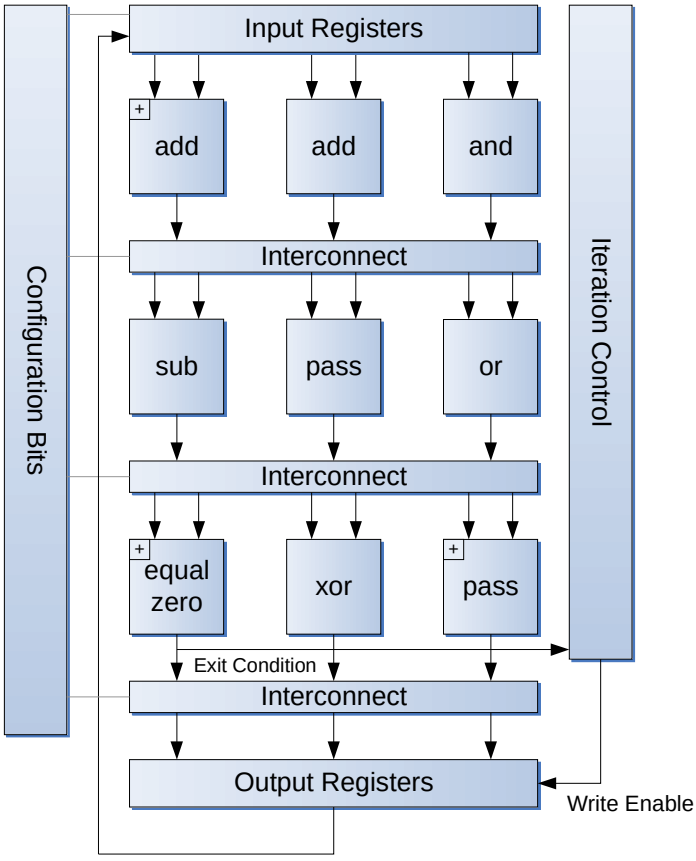


Figure 5.13. SRA instance for two hypothetical Megablocks.

5.5.4 Folded CGRA (1D CGRA)

For Megablocks with many operations (e.g., several hundreds), it can be impractical to implement all operations at the same time in hardware. The Folded CGRA (Figure 5.14) is composed of a single row of reconfigurable FUs and multiplexes the execution of each row over time. If the Folded CGRA is capable of changing its configuration

every clock cycle, and one does not consider pipelining, its execution becomes equivalent to a general 2D CGRA (see Section 5.5.1) and can use the same equations for modeling.

A Folded CGRA can be useful for a resource-constrained environment, when compared with the previous architectures, and it is an adequate option for implementing large Megablocks.

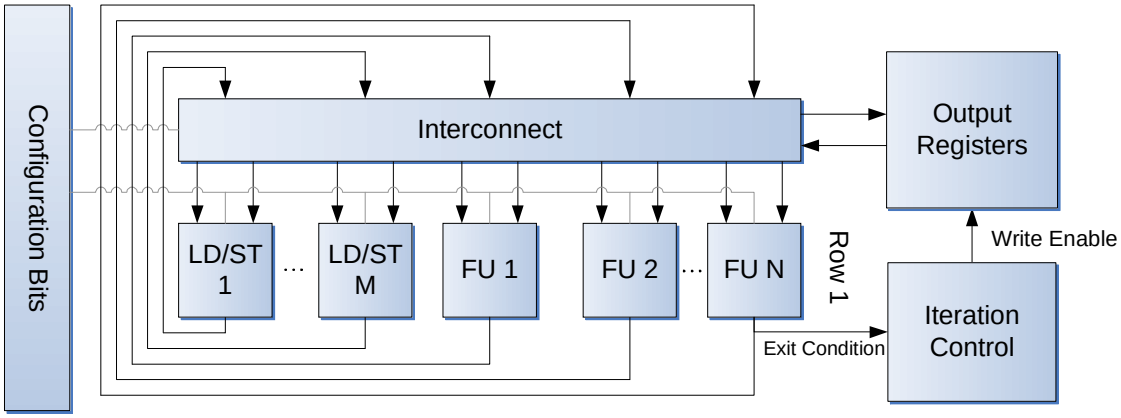


Figure 5.14. General architecture for a Folded CGRA-based RPU which supports Megablocks.

5.6 Megablock Pipelining

When mapping loops to 2D CGRAs, one can significantly improve performance (throughput and latency) by pipelining the iterations of the loop [48, 103, 104]. The main idea is to overlap consecutive loop iterations while preserving data-dependences and resource constraints. There are several ways to pipeline loops. The compiler community, which traditionally addressed GPPs, uses software pipelining techniques [49], being modulo scheduling [103] one of the possible schemes. In the context of hardware synthesis (e.g., high-level synthesis) loop pipelining is also known as loop folding [115] and it has been addressed by several authors (see, [48, 104], just to name a few). To the best of our knowledge, most approaches use the iterative modulo scheduling algorithm proposed by Rau [103]. As with typical loops, Megablocks can also be accelerated by pipelining their iterations.

In this section we present a technique to pipeline the iterations of Megablocks. The technique moves inter-iteration dependencies from the Megablock body to a separate module (i.e., the Input Module). The Megablock kernel becomes a data-flow graph which can be fully pipelined. The input module is not pipelined and is responsible for

feeding the Megablock kernel. We present a study of expected performance gains after applying this pipelining technique through estimation models, and suggest hardware extensions which enable Megablock pipelining in row-based 2D CGRAs (see Figure 5.11), as well as specialized architectures, such as the SAr (see Section 5.5.2).

5.6.1 Inter-Iteration Dependencies

The data dependencies [29] in Megablocks can be grouped into two types: direct and indirect dependencies. Direct dependencies are data dependencies between operations which are explicitly represented in the Megablock. They are exposed in the Megablock graph representation by data connections. Feedback connections are data connections between values of different iterations, and represent direct inter-iteration dependencies. Indirect dependencies are not explicitly represented, and usually correspond to operations which manipulate data in a medium external to the processor (e.g., memory accesses).

To pipeline Megablocks, we propose a technique that is capable of handling direct inter-iteration dependencies, by moving them to outside of the Megablock body, and that can be applied to Megablock without indirect inter-iteration dependencies.

Consider the C code for the function *vecsum* in Figure 5.15, which sums the elements of an array. Figure 5.16 shows the repeating pattern of a Megablock found in the execution trace in a MicroBlaze processor [90] of a program which uses the function *vecsum*, and Figure 5.17 represents the same Megablock as a graph, according to the representation introduced in Chapter 4, Section 4.4. The *addk* MicroBlaze instruction with address 18C adds the contents of register 3 to the contents of register 4, and stores the results back to register 3. The next instruction, a *sw* instruction with address 190, is a store operation. It sums the contents of register 7 with the contents of register 9, and the result is the memory address where the content of register 3 will be stored. As the previous instruction alters the content of register 3, which is needed by this *sw* instruction, there is a direct dependency between these two instructions, on the content of register 3. This dependency is represented in the Megablock graph representation as a data connection between the node *5:add* and *7:store*.

The first instruction, the *lw* instruction with address 180, reads the contents of register 9. As this register was lastly written by the *addik* instruction with address 19C in the previous iteration, there is a direct inter-iteration dependency between these two

instructions. This dependency is represented in the graph as a feedback connection between the node *10:add* and the input node *r9 (input).*

```
void vecsum(int* A, int* B, int* C, int n)
{
  int i;
  for(i = 0; i < n; i++) {
    C[i] = A[i] + B[i];
  }
}
```

Figure 5.15. C code for a *vecsum* function.

```
0x00000180 lw r3, r5, r9      → 0:add
                                 1:load
0x00000184 lw r4, r6, r9      → 2:add
                                 3:load
0x00000188 addik r10, r10, 1  → 4:add
0x0000018C addk r3, r3, r4    → 5:add
0x00000190 sw r3, r7, r9      → 6:add
                                 7:store
0x00000194 rsubk r18, r10, r8 → 8:rsub_carry
0x00000198 bneid r18, -24     → 9:equalZero
0x0000019C addik r9, r9, 4    → 10:add
```

Figure 5.16. Assembly instructions of the repeating pattern of a Megablock found in the trace of *vecsum* running on a MicroBlaze processor, and their correspondent translation to operations to be mapped to a CGRA.

The *lw* instructions with address 180 and 184 read values from the memory addresses given by the sum of the content of register 9 and the content of register 5 and 6, respectively. The *sw* instruction with address 190 writes the content of register 3 to the memory address given by the sum of the content of register 9 and the content of register 7. Depending on the values of register 5, 6 and 7, these instructions can be reading and writing to the same memory position in the same or in different iterations. If a Megablock contains instructions which write to memory, there might be indirect dependencies. Because registers can have any value, the dependency is not tied to the registers we use, but on the addresses accessed. Memory instructions will be dependent

if at any point in the Megablock, a write operation has the same target address of a previous or a subsequent read operation.

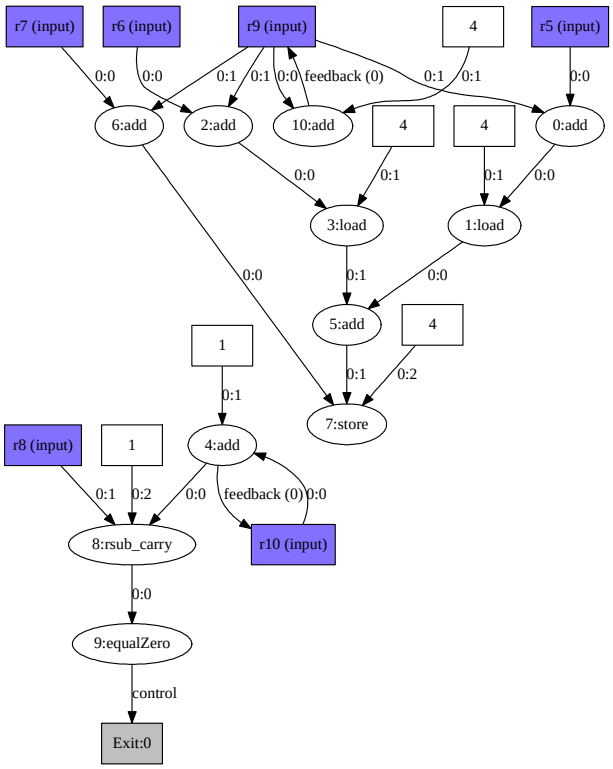


Figure 5.17. Graph representation of the repeating pattern of the Megablock found when executing *vecsum*.

We focus on the pipelining of Megablocks with specific characteristics. We will show later on (see Chapter 6, Section 6.4) that those characteristics/constraints will not prevent us to pipeline most of the Megablocks extracted from the set of benchmarks used in this work. Specifically, we consider Megablocks which an analysis can determine to have no indirect inter-iteration dependencies. This information implies memory disambiguation techniques, and can be provided either by a compiler, or extracted from the Megablock.

A Megablock does not have indirect inter-iteration dependencies if we can guarantee that: 1) *store* operations are executed according to their original order; 2) the contents of the addresses accessed by *load* operations are not changed during the Megablock. Guarantee 1) implies a mechanism for serializing the memory writes, and can be enforced when mapping the operations to the hardware. This guarantee avoids *output dependencies* between memory writes. Guarantee 2) is dependent on the program and

compiler options. If guarantee 2) holds, the values accessed by load operations are immutable, avoiding *true dependencies* and *anti-dependencies* between memory operations. This guarantee can be achieved when programs use separate memory regions (e.g., occurring with non-overlapped arrays) for reading and writing values. This information can be given to the compiler in C by using the `restrict` keyword of the C99 standard when declaring pointers [116], or can be determined in some cases by alias analysis techniques.

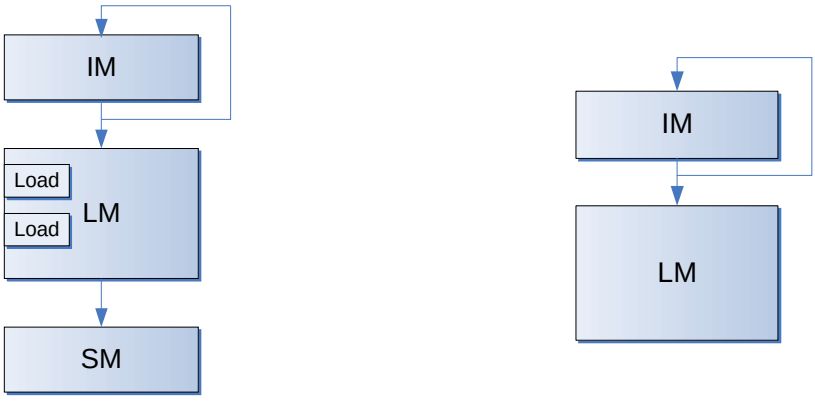
As guarantee 1) can be enforced by the mapping phase, we only have to ensure that Megablocks respect guarantee 2). As an example, the source code in Figure 5.15, which originated the assembly instructions of Figure 5.16 and the Megablock graph representation of Figure 5.17, uses different arrays for reading and writing, thus respecting guarantee 2). We assume that this information is given by the compiler as additional information. It can also be discovered by analysis of the Megablock[6].

5.6.2 Architecture for Pipelined Megablocks

Figure 5.18 shows two general RPU architectures for pipelining Megablocks. The architecture in Figure 5.18b) is a specialization of the architecture in Figure 5.18a), when considering Megablocks without memory accesses. Both architectures have an Input Module (IM) and a Loop Module (LM). The architecture with support for memory operations (see Figure 5.18b)) includes a Store Module (SM) and load units inside the LM. Both architectures execute iterations atomically, i.e. iterations are either fully executed or discarded. An iteration is discarded when it activates an exit point. When an exit point is activated, the Megablock execution ends.

The LM is a pipelined dataflow implementation of the Megablock repeating pattern (can be thought as the kernel), where the Megablock is split into several stages (see Figure 5.19). Each stage executes a different iteration of the Megablock, and when the LM advances a step (which can take from one to several clock cycles, depending on the Megablock and its implementation), all stages execute simultaneously.

[6] Note that this is not focused on this thesis. However, as an example: in this case, the value of register 9, which is used by the three memory operations, does not change between the loads and the store operations. If we know the values of r5, r6 and r7, we can calculate the minimum distance between the load and the store operations. If the minimum distance is D, this means that we can overlap up to D iterations, which will determine the maximum number of the pipeline stages we can have without incurring in indirect dependencies.



a) RPU with memory operations b) **RPU without memory operations**

Figure 5.18. General blocks for Megablock pipelined execution.

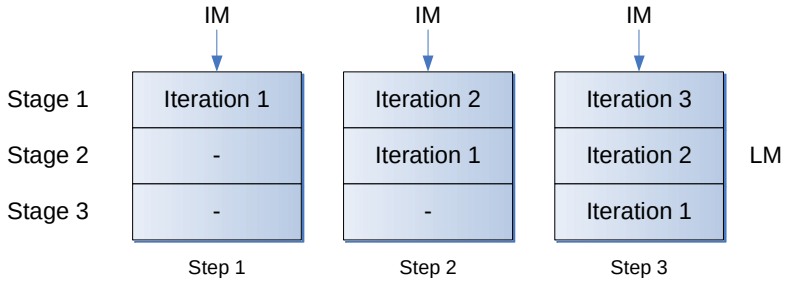


Figure 5.19. Execution of an LM with three stages.

An iteration completes when it finishes execution in the last stage of the LM without activating exit points. All exit points are delayed so that when they are checked, the corresponding iteration is in the last stage. After filling the pipeline (step 3 in Figure 5.19), the LM completes an iteration per step. To advance a step, the LM needs the values generated by the IM. The IM is responsible for generating the set of inputs for each iteration, and only depends on the values generated in the previous step of the IM.

This approach includes a module for *store* operations (i.e., SM) to implement guarantee 1) for indirect dependencies (i.e. *store* operations have to be executed by their original order). Since the LM executes operations of different iterations simultaneously, the s*tore* operations are moved outside the LM, i.e., to the SM. This way, all *store* operations are delayed to just after the last stage. The SM only execute if no exits are activated for that iteration, avoiding speculative writes to memory. The SM depends on the results of the LM.

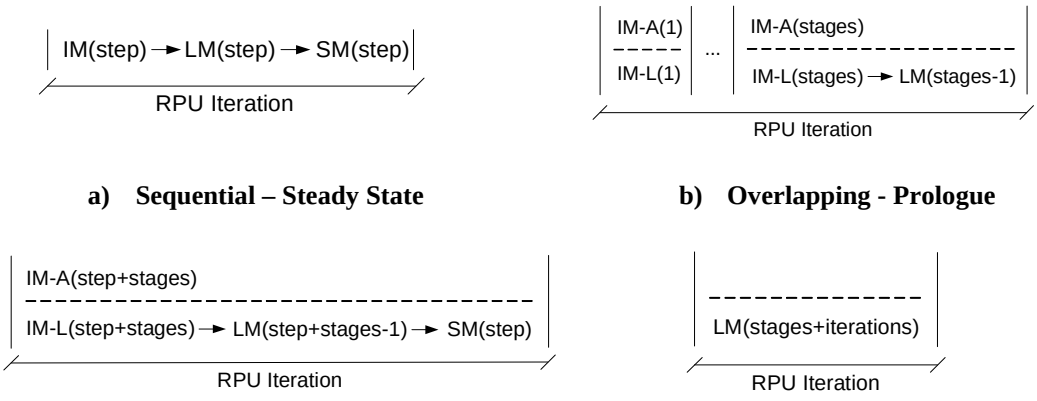
According to guarantee 2) for indirect dependencies, *load* operations are done from immutable locations. This means that *load* operations can be done in any order, and remain inside the LM. However, in this case the step of the LM only finishes after all *load* operations complete.

Table 5.3 summarizes the execution dependencies between the modules of the pipelined RPU. Figure 5.20 shows two schedules for the execution of the pipelined RPU with memory accesses. Figure 5.20 a) presents the steady state of the simplest execution schedule for the modules, which is to execute the modules sequentially. However, according to the dependencies, the IM only depends on its previous values, and as soon as it finishes execution, it can start computing the values of the next iteration. If we overlap the execution of the IM with the remaining modules, we obtain the schedule presented in Figure 5.20b) to d).

Module	Depends On Results From
IM (Input Module)	IM of previous iteration
LM (Loop Module)	IM of current iteration
SM (Store Module)	LM of current iteration

Table 5.3. Dependencies between the modules of a pipelined RPU.

The IM execution is split in two parts executed concurrently, IM-A and IM-L. IM-A refers to the execution of arithmetic and logic operations (e.g., addition, subtraction). IM-L corresponds to the execution of load operations. In this model store operations are not allowed in the IM. The IM is split in these two components as in real-life systems the number of concurrent memory accesses is usually very limited, and when the IM execution overlaps with the execution of the remaining modules, they will compete for the same limited resources. We consider that the execution of the IM associated to the load operations (IM-L) does not overlap with the remaining modules (LM and SM), which can also have memory operations. The LM can have a similar decomposition, LM-A and LM-L, where the arithmetic and logic components execute concurrently with both the IM-A component and the memory related components, in a third overlapping level. As the LM is pipelined, the arithmetic-logic part usually executes within one clock cycle, and the load operations represent the longest execution part of the LM. For simplicity's sake, this decomposition was not considered.



a) Sequential – Steady State **b) Overlapping - Prologue**

c) Overlapping – Steady State **d) Overlapping – Exit Iteration**

Figure 5.20. Possible schedules for the modules of a pipelined RPU.

The sequential schedule also has a prologue stage and an exit iteration, identical to the ones for the overlapping schedule, but without the overlapping of the arithmetic and logic operations. The RPU without memory accesses uses similar schedules, which do not include the SM.

Software pipelining algorithms usually consider a prologue, a steady state, and an epilogue. The purpose of the epilogue is to orderly terminate the execution of iterations which cannot execute in the steady state because there are no more new iterations to feed the pipeline. Our approach does not have an epilogue. Since we commit iterations atomically, we can simply ignore the iterations which have already started but have not yet terminated by the time an exit is activated.

Figure 5.21 shows the execution of the RPU modules when using an overlapping schedule, and considering that the LM has three stages and executes for two iterations. In the first step, the IM is the only module executing. In the second step, the results from the first step of the IM are ready and both the first step of the LM and the next step of the IM can start concurrently. The SM does not execute yet because it uses data from the last stage of the LM. At this point, the first iteration is in the first stage. As we are considering an LM with 3 stages, there is no data available in the third stage yet. When the first iteration executes in the last stage of the LM, the pipeline becomes full, and after execution, the SM can perform the stores of the first iteration and complete it. Each following step of the RPU completes an iteration. In the last step, the Megablock exits. As the stores of that iteration are neither performed nor the inputs of the next

iteration are needed, the execution stops after computing the results of the exit signals of the LM.

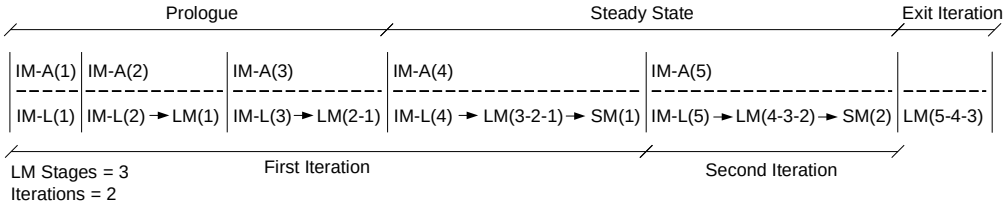


Figure 5.21. Execution using an overlapping schedule with an LM with 3 stages.

Equations (5.5) to (5.9) define the term RPU_{Cy} of equation (5.2) for the sequential and overlapping schedules, respectively, of the pipelined RPU with support for memory accesses. The term LM_{Stg} represents the number of stages in the LM, while N_{It} represents the number of completed iterations in the Megablock.

The sequential schedule equations (5.5) and (5.6) consider an RPU with and without memory accesses, respectively. The terms $IM\text{-}Avg_{Cy}$, $LM\text{-}Avg_{Cy}$ and $SM\text{-}Avg_{Cy}$ represent the average clock cycles needed to execute a single step of the IM, LM, and SM, respectively. $LM\text{-}Last_{Cy}$ represents the clock cycles needed to execute the last step of the LM in the exit iteration.

In the overlapping schedule equations (from (5.7) to (5.9)), the terms $IM\text{-}A(i)_{Cy}$, $IM\text{-}L(i)_{Cy}$, $LM(i)_{Cy}$ and $SM(i)_{Cy}$. represent the clock cycles needed to complete the step *i* of the corresponding module. Equations (5.8) and (5.9) consider that each module always executes in a fixed number of clock cycles represented by the terms $IM\text{-}A_{Cy}$, $IM\text{-}L_{Cy}$, LM_{Cy} and SM_{Cy}.

Usually, the latency of the LM is determined by the latency of the load operations. As the LM is pipelined, an LM without load operations will have the shortest step between all modules (usually one clock cycle). In this case, the IM latency becomes the dominant term. Considering the overlapping schedule without memory accesses, this means that the *Max* operation in equation (5.9) can be in most cases simplified to IM_{Cy}.

Equations (5.10) and (5.11) estimate the number of clock cycles needed by the RPU, for sequential and overlapping schedules, when Megablocks have a large number of iterations, well above the number of stages of the LM. These equations are useful for comparing the performance of both schedules, and for calculating maximum theoretical speedup limits when comparing with non-pipelined Megablock implementations.

5.6.3 Megablock Pipelining Algorithm

Consider the Megablock graph representation in Figure 5.17. As referred before in Chapter 4, Section 4.4, feedback connections in the graph representation can only point to nodes of the type *Livein*, and indicate the value that the input will have in the next iteration. They represent the *inter-iteration direct dependencies*.

Using the feedback connections we can extract the expressions which control the value of the inputs in the subsequent iterations. Starting at a *Livein* node with a feedback connection, traversing the graph in the opposite direction of the connection will reach the node that generates the input value for the next iteration. The algorithm *buildExpressionGraph* in Figure 5.22b) creates, given a node, a directed graph which represents the expression that calculates the values of that node. For instance, following the feedback connection in node *r9 (input)*, the values of the *Livein* are given by the output of the node *10:add.* Applying the algorithm *buildExpressionGraph* to this node will initially build a new graph. As it is the first time the algorithm sees the node *10:add*, this node is added to the graph. This node is an operation, so the algorithm is called recursively over each of the parent nodes of the node *10:add*. All the inputs of this node are either of type *Livein* or *Constant*, so after they are added to the graph, the algorithm stops. The algorithm returns a graph which represents the update expression for *r10 (input)*, which in this case is *r9 = r9 + 4*. The algorithm process to the next *Livein* node with a feedback connection (i.e., *r10*) and repeats the process. As this is the last *Livein* node with a feedback connection, there are no more expressions to extract.

In our pipelining technique, the algorithm *buildInputModuleGraph* (see Figure 5.22a)) is applied over the original Megablock graph to extract the *inputModuleGraph*, which represents the IM (see Figure 5.23). This graph is built by generating a graph for each *input* node which has a feedback connection, and merging the resulting graphs in a single graph. This graph represents the hardware structure responsible for generating the inputs for each Megablock iteration.

As the *feedback* connections from the original Megablock graph (see Figure 5.17) are being handled by the IM, when implementing the LM those connections can be ignored. Additionally, as our technique moves the store operations to outside of the LM, those operations are also removed from the graph. The scheduling of the resulting graph represents the LM. In Figure 5.24 we present a schedule of an LM graph using an “As-

Soon-As-Possible" (ASAP) based scheduler [117]. It results in a 3-stage pipeline. The SM is composed by the single store operation of the Megablock.

When implementing the pipelined Megablock, the outputs of the IM are connected to the inputs of the LM, and the values needed by the store operations are passed from the LM to the SM. *Input* nodes which do not have an incoming feedback connection do not change their value during loop execution and do not need to be updated.

$$\mathrm{RpuMem_{Cy}} = \left(\mathrm{IM\text{-}Avg_{Cy}} + \mathrm{LM\text{-}Avg_{Cy}}\right) \times \left(\mathrm{LM_{Stg}} + \mathrm{N_{It}} - 1\right) + \mathrm{SM\text{-}Avg_{Cy}} \times \mathrm{N_{It}} + \mathrm{LM\text{-}Last_{Cy}} \quad \textbf{(5.5)}$$

$$\mathrm{RpuNoMem_{Cy}} = \left(\mathrm{IM\text{-}Avg_{Cy}} + \mathrm{LM\text{-}Avg_{Cy}}\right) \times \left(\mathrm{LM_{Stg}} + \mathrm{N_{It}} - 1\right) + \mathrm{LM\text{-}Last_{Cy}} \quad \textbf{(5.6)}$$

$$\begin{aligned}\mathrm{RpuMemVar_{Cy}} = {} & Max\left(\mathrm{IM\text{-}A(i)_{Cy}}, \mathrm{IM\text{-}L(i)_{Cy}}\right) \\ & + \sum_{i=2}^{\mathrm{LM_{Stg}}} Max\left(\mathrm{IM\text{-}A(i)_{Cy}}, \mathrm{IM\text{-}L(i)_{Cy}} + \mathrm{LM(i-1)_{Cy}}\right) \\ & + \sum_{i=1}^{\mathrm{N_{It}}} Max\left(\mathrm{IM\text{-}A\left(i + LM_{Stg}\right)_{Cy}}, \mathrm{IM\text{-}L\left(i + LM_{Stg}\right)_{Cy}}\right. \\ & \left. + \mathrm{LM\left(i + LM_{Stg} - 1\right)_{Cy}} + \mathrm{SM(i)_{Cy}}\right) + \mathrm{LM\left(N_{It} + LM_{Stg}\right)_{Cy}}\end{aligned} \quad \textbf{(5.7)}$$

$$\begin{aligned}\mathrm{RpuMemFixed_{Cy}} = {} & Max\left(\mathrm{IM\text{-}A_{Cy}}, \mathrm{IM\text{-}L_{Cy}}\right) + Max\left(\mathrm{IM\text{-}A_{Cy}}, \mathrm{IM\text{-}L_{Cy}} + \mathrm{LM_{Cy}}\right) \\ & \times \left(\mathrm{LM_{Stg}} - 1\right) + Max\left(\mathrm{IM\text{-}A_{Cy}}, \mathrm{IM\text{-}L_{Cy}} + \mathrm{LM_{Cy}} + \mathrm{SM_{Cy}}\right) \\ & \times \mathrm{N_{It}} + \mathrm{LM_{Cy}}\end{aligned} \quad \textbf{(5.8)}$$

$$\begin{aligned}\mathrm{RpuNoMem\text{-}Fixed_{Cy}} = {} & \mathrm{IM\text{-}A_{Cy}} + Max\left(\mathrm{IM\text{-}A_{Cy}}, \mathrm{LM_{Cy}}\right) \times \left(\mathrm{LM_{Stg}} + \mathrm{N_{It}} - 1\right) \\ & + \mathrm{LM_{Cy}}\end{aligned} \quad \textbf{(5.9)}$$

$$\mathrm{RpuSequential_{Cy}} = \left(\mathrm{IM\text{-}Avg_{Cy}} + \mathrm{LM\text{-}Avg_{Cy}} + \mathrm{SM\text{-}Avg_{Cy}}\right) \times \mathrm{N_{It}} \quad \textbf{(5.10)}$$

$$\mathrm{RpuOverlapping_{Cy}} = Max\left(\mathrm{IM\text{-}A_{Cy}}, \mathrm{IM\text{-}L_{Cy}} + \mathrm{LM_{Cy}} + \mathrm{SM_{Cy}}\right) \times \mathrm{N_{It}} \quad \textbf{(5.11)}$$

Let us consider the IM in Figure 5.23. In this case we have an IM without loads (IM-L_{Cy} is 0) which can be executed in one clock cycle (IM-A_{Cy} is 1). The LM (see Figure 5.24) has three stages (LM_{Stg} is 3), and if we only consider the arithmetic and logic

operations, the maximum number of clock cycles a stage needs is one. If we admit two simultaneous loads per clock cycle, and with loads with one clock cycle latency, the maximum number of clock cycles a stage needs, taking into account memory accesses, is one (LM_{Cy} is one). If we admit one clock cycle for the store latency, the store module needs one cycle per step (SM_{Cy} is one).

```
buildInputModuleGraph(megablock)
   megablockDfg = createDfg(megablock)
   for each input of megablockDfg
      if input has feedback connection
         sourceNode = feedback parent
         inputDfg = buildExpressionGraph(sourceNode)
         add inputDfg to inputDfgList
   end for
   inputModuleGraph = mergeDfgs(inputDfgList)
```

a) Algorithm buildInputModuleGraph

```
buildExpressionGraph(sourceNode)
   if(sourceNode already added)
      return
   else
      add sourceNode

   if(sourceNode type is constant)
      return
   if(sourceNode type is livein)
      return

   for each parent of sourceNode
      buildExpressionGraph(parent)
   end for
```

b) Algorithm buildExpressionGraph

Figure 5.22. Algorithms for IM graph creation.

Let us consider a sequential scheduling. As we have memory accesses, equation (5.5) is used. In this case, the step in all modules has a fixed number of clock cycles, so the average number of clock cycles is the same as the number of clock cycles to execute a module. If the loop executes for 100 iterations, the number of clock cycles of the RPU execution is $(1 + 1) \times (3 + 100 - 1) + 1 \times 100 + 1 = 305$ clock cycles.

If we use an overlapping scheduling instead, and as the modules have a fixed number of clock cycles, equation (5.8) is used. If the loop executes for 100 iterations, the number of clock cycles of the RPU execution is $max(1, 0) + max(1, 0 + 1) \times (3 - 1) + max(1, 0 + 1 + 1) \times 100 + 1 = 204$ clock cycles. This results in a speedup of 1.49× when comparing the latency of the overlapped schedule over the sequential schedule, in this case.

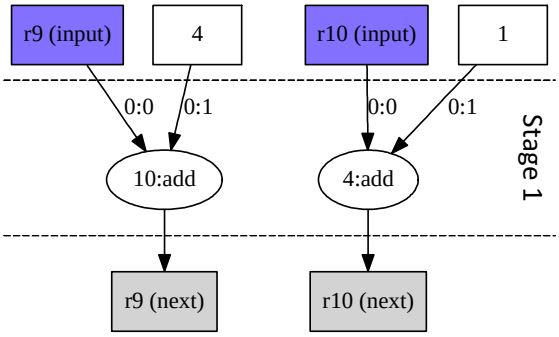


Figure 5.23. Input Module (IM) graph for a Megablock found in *vecsum*.

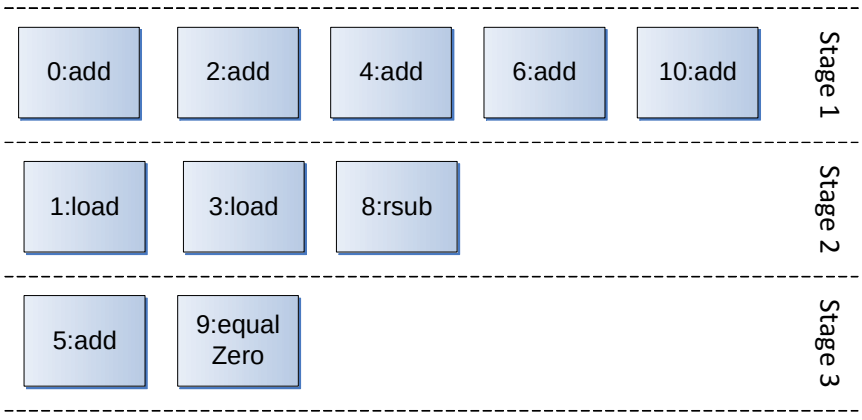


Figure 5.24. Loop Module (LM) schedule for a Megablock found in *vecsum*.

5.6.4 Hardware Support for Megablock Pipelining

Consider the general architecture, for a 2D CGRA with Megablock support, depicted in Figure 5.11. To enable our Megablock pipelining approach in such CGRAs, we propose three hardware extensions presented in Figure 5.25: (a) feedback lines to the top row, for the IM; (b) clock-enable control signals for each module; and (c) delays for the exit signals. The extensions enable the implementation of the IM, the LM, and the SM at the hardware level. The same extensions can be applied to specialized

architectures, such as the SAr (see Section 5.5.2) and the SRA (see Section 5.5.3). For simplicity, the CGRA in Figure 5.25 only has three rows.

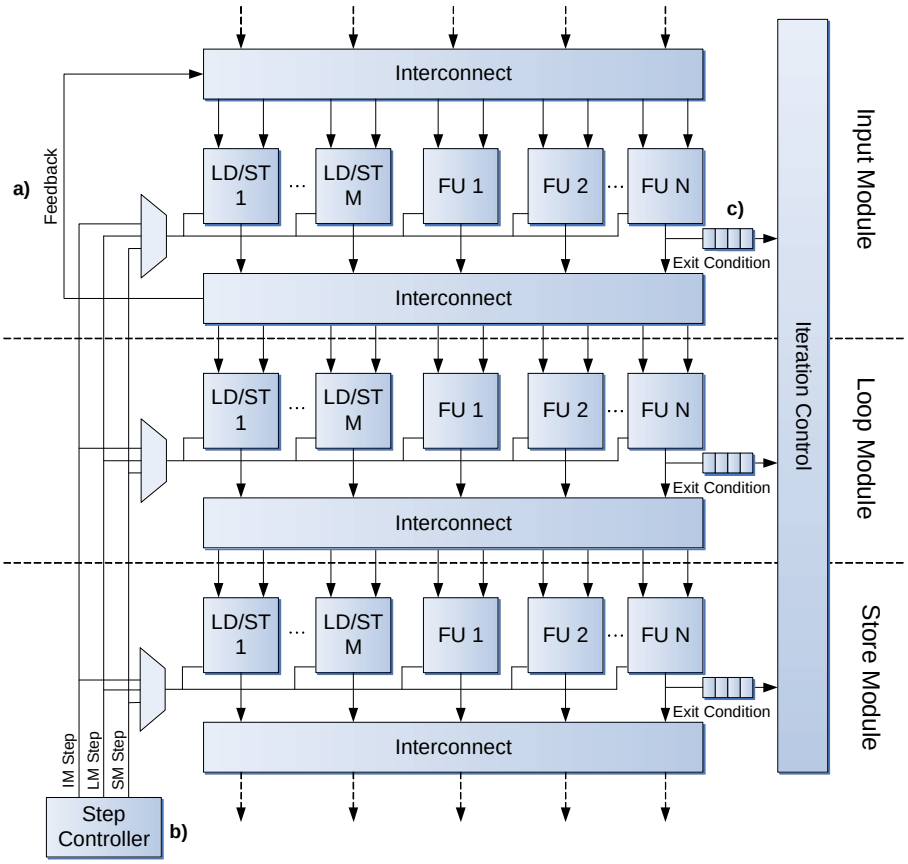


Figure 5.25. General architecture for a 2D CGRA-based RPU which supports Megablocks and Megablock pipelining.

The feedback lines (a) are needed for the IM re-alimentation. This kind of interconnection can be expensive, but as only Input Modules with a low number of stages are attractive for implementation, these lines can be present in only a restricted number of top rows. As the modules have producer-consumer relationships between them, we use a Step Controller (b) to indicate when there are values available for each module, and when they can execute. The exit delays (c) synchronize the exit signals so that when they activate, they always correspond to the iteration in the last stage. They can be implemented with simple 1-bit FIFOs.

5.7 Summary

This chapter focused on the practical aspects of using Megablocks. We explained how to transform assembly instructions into the Megablock graph representation. We proposed and compared two techniques for identifying previously detected Megablocks in a trace: Single Address Identification (SAI) and Megablock Signature Identification (MSI). We described RPU architectures able to implement Megablocks

Finally, we explored the possibility of pipelining Megablocks in hardware, by suggesting techniques to handle the inter-iteration dependencies, as well as architecture augmentations to support Megablock pipelining. This technique is appropriate for loops where the operations related to the update of values used across iterations represent a small part of the loop and can be executed with lower latency than the complete loop.

6 Experimental Results

This chapter presents extensive results about the techniques introduced in the previous chapters, such as characterization of Megablock coverage, and experiments considering several scenarios regarding Megablock mapping (i.e., baseline results, *if-conversion*, graph transformations) and pipelining of Megablocks.

6.1 Experimental Setup

We consider the general architecture described in Section 5.5, with an Reconfigurable Processing Unit (RPU) coupled to the General Purpose Processor (GPP) as depicted in Figure 5.10b). We use a MicroBlaze soft-core [90] as the target GPP, optimized for speed. The GPP communicates directly with the RPU through FSL connections [118]. We use a Xilinx Spartan-6 LX45 FPGA as the target FPGA platform for the implementation of hardware designs.

To evaluate our approach, we use a set of 66 benchmarks using integer data types from embedded computing (the benchmarks are available online [119]). We use *mb-gcc* 4.1.2 [120], the GCC compiler targeting MicroBlaze. By default, the optimization level flag is set to –O2[7]. The 66 benchmarks were separated in two sets, named *ifs* and *no-ifs*, according to the existence or non-existence of control-flow related constructions (e.g., *if* statements in C code) in the kernels, respectively. The *ifs* set contains 29 benchmarks and the *no-ifs* set contains 37 benchmarks. Table 6.1 and Table 6.2 present a characterization of the benchmarks that form the *no-ifs* and *ifs* sets, respectively. Column “Kernel LOC” indicates the number of lines of code that compromise the benchmark kernel, excluding comment and empty lines. The kernels contain a wide range of code sizes, from simple to complex examples. The lines of code vary between 6 and 241 in the *no-ifs* set, and from 9 to 226 in the *ifs* set.

[7] It has been observed that unoptimized code is much easier to schedule than optimized code [15]. Although it is not guaranteed that all the binaries running on the system have been compiled with optimizations, we opted to use compiler-optimized programs when evaluating scheduling algorithms by default.

Benchmark	Kernel LOC	#loops	Max. Loop nesting level	#Input/output arrays
Autcor	15	2	1	1/1
Bilinear	190	1	0	2/1
bob_hash	16	1	0	1/0
Checkbits	31	1	0	1/1
Checksum	103	1	0	1/1
compress1	15	1	0	0/0
compress2	20	1	0	0/0
corr_gen	17	2	1	2/1
Count	8	1	0	0/0
Dotprod	6	1	0	2/1
even_ones	9	1	0	0/0
Expand	14	1	0	0/0
fdct_8x8	241	4	2	1/1
fft	39	3	2	2/1
fibonacci	26	1	0	0/0
fir	16	2	1	2/1
gcd2	16	1	0	0/0
gouraud	12	1	0	0/1
hamming_dist	11	1	0	0/0
lookup2	55	1	0	1/0
maxstr1	6	1	0	0/0
maxstr2	25	1	0	0/0
md5	173	1	0	1/1
mulinv	18	1	0	0/0
perlins	96	1	0	1/1
pix_expand	12	1	0	1/1
popcmpr	20	1	0	0/0
popcnt	12	2	1	1/1
quantize	42	2	1	2/1
reverse	10	1	0	0/0
smooth	23	4	3	1/1
vecsum	10	1	0	2/1
wave_horz	31	4	2	3/1
wave_vert	40	4	2	3/2
ycbcr422p_rgb	148	1	0	3/1
yc_demux_be16	22	1	0	1/3
yc_demux_le16	22	1	0	1/3

Table 6.1. Characteristics of the benchmarks which form the set *no-ifs*.

Benchmark	Kernel LOC	#Ifs inside loop	Ifs Max Nesting	#Invoked Functions	#loops	Max. Loop nesting level	#Input/output arrays
adpcm_coder	67	9	1	0	1	0	1/1
adpcm_decoder	52	6	2	0	1	0	1/1
boundary	18	1	0	0	2	1	1/2
bubble_sort	14	1	0	0	2	1	1/0
change_brightness	24	1	1	0	1	0	1/1
compositing	12	2	0	0	1	0	2/1
conv_3x3	81	2	0	0	2	1	2/1
crc32	15	1	1	0	1	0	0/0
divlu	16	1	0	0	1	0	0/0
gcd1	15	1	1	0	1	0	0/0
idct_8x8_12q4	226	16	1	0	4	1	1/1
isqrt1	21	4	0	0	1	0	0/0
isqrt2	16	1	0	0	1	0	0/0
isqrt3	17	1	0	0	1	0	0/0
isqrt4	18	1	1	0	1	0	0/0
mad_16x16	36	1	0	1	4	3	2/1
mad_8x8	35	1	0	1	4	3	2/1
max	9	1	0	0	1	0	1/0
median_3x3	82	13	0	0	1	0	1/1
modexp	11	1	0	0	1	0	0/0
motion_estimation	22	0	0	1	4	3	2/1
perimeter	35	1	1	0	1	0	1/1
pix_sat	24	1	2	0	1	0	1/0
rgb_to_hsv_int	57	9	2	0	1	0	3/1
rng	177	7	1	0	1	0	1/1
sad_16x16	17	0	0	1	2	1	1/1
sad_8x8	17	0	0	1	2	1	1/1
sobel	51	1	0	2	1	0	1/1
viterbi_gsm	37	1	1	0	4	2	3/2

Table 6.2. Characteristics of the benchmarks which form the set *ifs*.

Columns *#loops* and *Max. loop nesting level* indicate the number of loop constructions in the code (e.g., *for* and *while* statements), and the maximum nesting level of the loops. All examples contain at least one loop construct, and most benchmarks do not have nested loops (27% and 34% of the benchmarks in the *no-ifs* and *ifs* sets have nested loops, respectively). Column *#Input/output arrays* indicate the number of arrays which are used as input/ output of the kernel. In most cases, the benchmarks either use one array for input values and another for output values, or do not use arrays at all.

Table 6.2 includes three additional columns. The column *#Ifs inside loop* indicates the number of control-flow constructions (e.g., *if* statements) found in the source code (only *ifs* inside loops are accounted for), while the column *Ifs max nesting* indicates the maximum size of an *if* statement chain. The column *Invoked Functions* indicates the number of times the benchmarks call an external function. This behavior was found only in a reduced number of benchmarks of the *ifs* set, being *abs* the only function called in those benchmarks.

All benchmarks use initialized input data. The input arrays are declared as global variables with static initializers to minimize the impact of the initialization when running the benchmark. The arrays are initialized with random values, with well-defined seeds to ensure the repeatability of the experiments. The total execution clock cycles of the benchmarks, when executing in the MicroBlaze considered for experiments, vary between 10,000 and 1,000,000 clock cycles.

For the speedup estimations concerning the Megablocks, we used the instruction latencies of a MicroBlaze processor optimized for speed (as defined in the MicroBlaze Reference manual [90]), for the equivalent operations of the intermediate representation. We consider that the program data fits in the FPGA Block RAMs (BRAMs), thus enabling loads and stores to memory to be done in one clock cycle [121], and we consider, as default, that up to 2 simultaneous memory accesses can be done in one clock cycle (this setup fits well with embedded devices, e.g., the dual-port BRAMs found in FPGAs [122], and memory architectures of DSPs [123]).

We used the tool Megablock Extractor (see Figure C.1a) and Figure C.1b) in Appendix C) to extract the Megablocks from the executable binaries, and the tool Megablock Estimation (see Figure C.2a) and Figure C.2b) in Appendix C) to simulate an architecture which supports the extracted Megablocks.

For the hardware implementations of the Megablocks, we used the tool VHDL for Megablocks (see Figure C.3 in Appendix C) to generate the hardware modules, and Xilinx ISE 12.2 to obtain synthesis and placement and routing results.

6.2 Megablock Coverage

The coverage of a detection method over the execution of a program is an important measure, as it indicates an upper bound of the impact an RPU can have (Section 4.1). To measure the coverage of the Megablock, we considered three adjustable parameters of

Megablock detection: *maximum pattern size*, *type of pattern unit* (the considered units are instruction, basic block and fragment) and *unrolling of inner loops*. To maximize the number of detected Megablocks, the parameter *executed instructions threshold* is set to 1 (see Section 4.3). The proposed identification methods (see Section 5.4), Single Address Identification (SAI) and Megablock Signature Identification (MSI) , can have an impact in the coverage and we also take it into consideration. We also indicate the detection ratio, i.e., in how many benchmarks we can detect at least one Megablock (coverage greater than 0%).

Figure 6.1 shows the average coverage obtained when using the Megablock detection considering several values for the parameters described above. The coverage results represent an average over the coverage of all benchmarks, including benchmarks without detected Megablocks (coverage equal to 0%). For instance, when using the SAI method and no unrolling of innermost loops, basic block as the detection unit, and 8 as the *maximum pattern size*, the average coverage achieved by the Megablock detection in the complete set of 66 benchmarks is 70%.

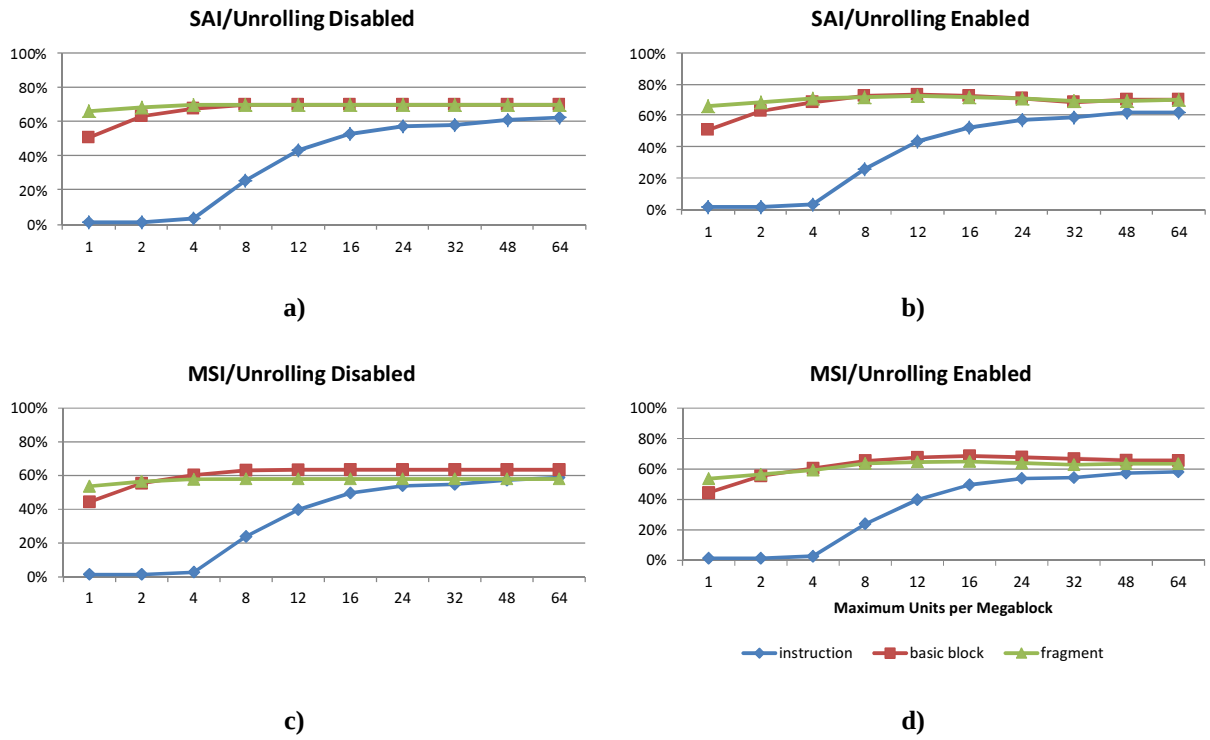


Figure 6.1. Average coverage of the complete set of benchmarks when applying Megablock detection and varying several parameters.

Figure 6.2 shows the ratio of benchmarks where Megablocks were detected (coverage greater than 0%). In this set of benchmarks, unrolling increases the ratio of benchmarks with

detected Megablocks to close to 100% (see Figure 6.2), and diminishes the differences when using basic blocks and superblocks as units when *maximum pattern size* is 24 or greater. However, unrolling has a modest impact in the average coverage (see Figure 6.1). The ratio of benchmarks with detected Megablocks was already high before unrolling (around 90%), and the coverage values of the new benchmarks are close to the average. Unrolling increases the average coverage by 3% in the best case.

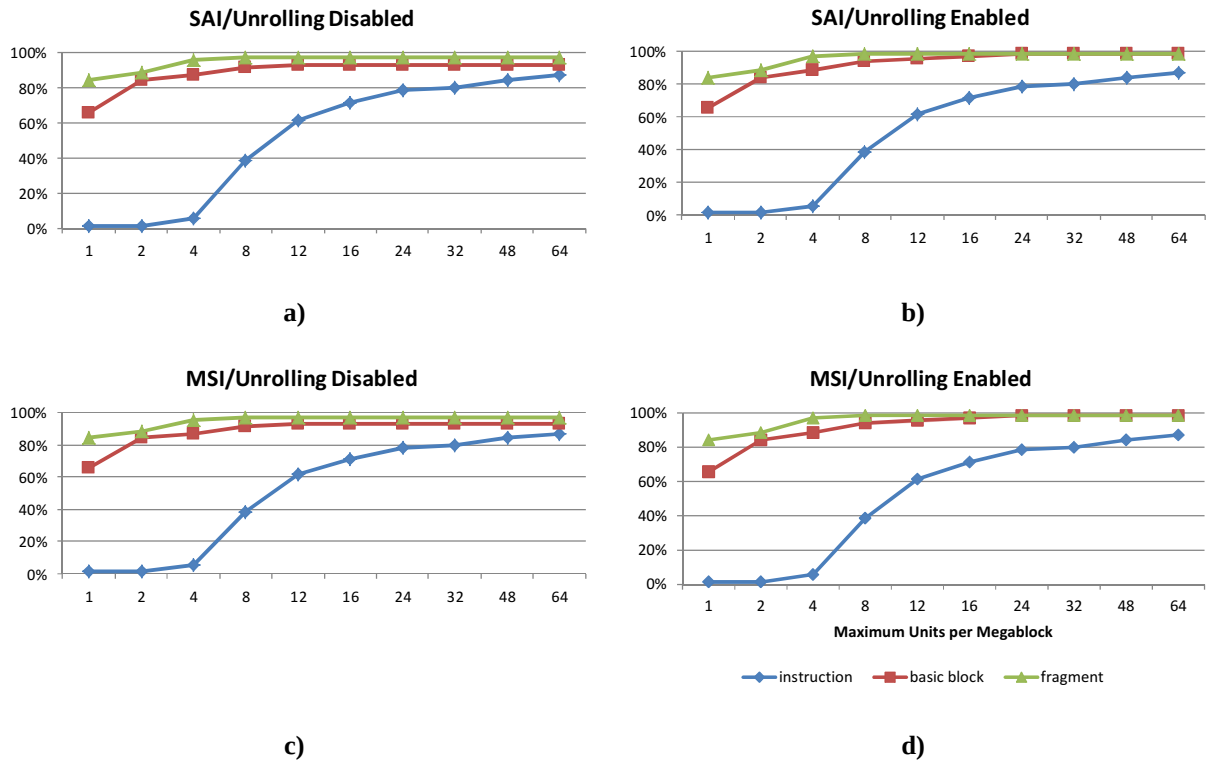


Figure 6.2. Megablock detection ratio in the complete set of benchmarks. Indicates the ratio of benchmarks were valid Megablocks could be detected.

When compared with the SAI method, the MSI method lowered the coverage in all cases. The main reason comes from the additional overhead in the MSI method when compared with the SAI method (it has at least an overhead of two iterations per call, which are needed to detect the Megablock).

In the other hand, in cases where there are many conflicts due to Megablocks having the same start address, MSI can execute both, potentially increasing the coverage. However, for the tested benchmarks, on average the additional overhead outweighed this benefit.

With the SAI method, the average coverage when using basic block and fragment units converged rapidly, at a value of *maximum pattern size* of 4. However, to obtain Megablock

detection rates close to 100% with basic block units, the *maximum pattern size* needs to be increased to 24 and unrolling must be enabled.

When using unrolling, in some cases the average coverage lowers with an increase in the *maximum pattern size*. This happens because when using unrolling, outer loops with unrolled inner loops are given a higher priority than isolated inner loops. If the path of the outer loop is not regular, the Megablock terminates sooner and completes a lower number of total inner loop iterations than if the inner loop had been implemented instead of the outer loop. Summarizing, unrolling can be counterproductive in benchmarks which do not form regular execution patterns, and aggravates when the number of iterations of the outer loop is low (e.g., less than 10).

According to the obtained values, we decided to use a default setup for Megablock detection, with a *maximum pattern size* of 24[8], and basic block as detection unit. *Unrolling of inner loops* is considered as an optional parameter.

We have chosen the basic block as the default detection unit as it is simpler to implement than fragments, and a *maximum pattern of size* of 24 provides similar Megablock detection coverage when using units either based on basic blocks or fragments.

Figure 6.3 shows individual coverage values for each benchmark, when considering the default Megablock detection setup and an implementation of the Backward Branch Loop Detection (BBLD) used in the Warp Processor [124, 125]. For the Megablock detection we disabled unrolling of inner loops to provide a fair comparison, as BBLD only supports inner loop detection. The results in Figure 6.3 show, on average, higher coverage when using the BBLD. This was expected, as the current Megablock detection trades-off the coverage obtained when statically considering all paths of a loop, with having a loop which represents an execution path. The advantage of having an execution path loop is that it forms a dataflow representation suited for non-sequential computation models. Furthermore, we can apply transformations which cannot be used, or that are more complex, when considering loops with branching code.

With respect to coverage, the results are highly dependent on the benchmark. In some examples the difference between coverage values is high (e.g., for *isqrt*, *maxstr*, *pix_sat*, *viterbi_gsm*), and in some cases Megablocks are not identified (e.g., *adpcm_coder*,

[8] A runtime adaptation of the *maximum patter size* according to the characteristics of the application running on the system is not considered in this work.

adpcm_decoder, *conv_3x3*, *smooth*). This happens with kernels of the benchmarks containing branches, and not forming repeating patterns during execution.

Figure 6.3. Individual coverage values in the main set of benchmarks, for Megablock detection using the default setup and Backward Branch Loop Detection.

Other examples allow high Megablock coverage values, above 80%, even when the kernels have branches (e.g., *change_brightness*, *gcd2*, *max*, *lookup2*). In these cases, there were one or more frequent paths representing most of the execution of the kernel. Megablock coverage close to 100% usually represents benchmarks which do not have branches inside the kernel (e.g., *autcor*, *bob_hash*, *compositing*, *dotprod*, *gouraud*, *vecsum*). There are also examples where the Megablock detection has high coverage values, but the BBLD has very low or 0% coverage (e.g., *bilinear*, *checkbits*, *checksum*, *md5*, *perlins*). This corresponds to kernels with loops above a certain size (i.e., several hundreds of instructions).

The objective of using Megablocks is to have a runtime structure better suited for implementation in an RPU than the static representation of the loop. However, the Megablock is only useful if a substantial portion of the program execution is spent inside Megablocks. In this section we explored the Megablock coverage over a number of detection configurations and arrived at a default setup. The average coverage achieved by Megablock detection in the main set of benchmarks when using the default setup is 70%, while about half of the benchmarks achieved coverage over 90%, which corresponds to an average overall speedup upper bound of 3.3× and 10× respectively. We consider that these results justify the use of Megablocks in a dynamic partitioning approach.

6.3 Megablock Mapping

We have developed tools which enabled us to study the impact of using Megablocks in a dynamic partitioning system. This section presents extensive estimation results over several configurations. We also have tested this approach by implementing a proof-of-concept system [126], whose presentation and results are available in Appendix A.

We considered the default Megablock detection setup (basic block as the *type of pattern unit*, *maximum pattern size* of 24), using the SAI method. When using SAI, there can be conflicts if two Megablocks share the same start address. If there are address conflicts between two or more Megablocks, the Megablock with higher coverage is chosen, according to an approximate coverage estimation performed during the detection phase. The mapping of memory operations respects the original order of the operations.

As target architecture we use the SAr architecture considering FUs with registered outputs (see Section 5.5.2). The reason for choosing this architecture is twofold. On a practical point of view, among the presented architectures, this was the easiest to test and implement. On the

other hand, it is still possible to obtain results meaningful for the other architectures. When the results of each FU are stored in registers in the SAr architecture, the latency of Megablock execution (i.e., clock cycles) is equivalent to the latency of the other architectures presented in Section 5.5, including the General 2D CGRA (see Section 5.5.1), the SRA (see Section 5.5.3) and the Folded CGRA (see Section 5.5.4).

We considered three parameters which define the target SAr architecture: maximum number of concurrent memory operations, maximum number of concurrent arithmetic-logic operations, and the ratio between the clock frequency of the RPU and the processor.

For each set of benchmarks, we considered the cases were unrolling of inner loops is disabled (*innerloops*) or enabled (*unrolled*). We consider *innerloops* as the default parameter, and present results with unrolling enabled for the cases where there is any change. The values are obtained with the tool Megablock Estimation (see Figure C.2a) and Figure C.2b) in Appendix C), which has support for an estimator based on a SAr architecture (see Section 5.5.2) with registered results. In the speedup calculations we consider all communication overheads.

6.3.1 Baseline Results

Table 6.3 and Table 6.4 present the characteristics of the detected Megablocks with *innerloops*, for the *no-ifs* and *ifs* sets, respectively. Table 6.5 and Table 6.6 present the same characteristics found in the *unrolled* case. For the baseline results, all graph transformations proposed in Section 5.1 are disabled.

The Critical Path Length (CPL) and Instruction Level Parallelism (ILP) results were calculated assuming there are no restrictions in the target architecture (e.g., unlimited arithmetic, logic and memory operations per clock cycle) and that memory operations are independent. These results indicate an upper bound of the values that can be obtained when implementing the Megablocks in practical architectures. We used a weighted average which has into account the number of times each Megablock was executed.

Benchmark	Megablocks Det./Exec.	Avg. It. per call	Avg. Op. per It.	Avg. ILP (Min/Max)	Avg. CPL (Min/Max)
autcor	1/1	159.0	13.0	2.6	5.0
bilinear	1/1	99.0	161.0	7.0	23.0
bob_hash	1/1	3999.0	11.0	1.4	8.0
checkbits	1/1	166.0	69.0	4.3	16.0
checksum	2/2	65.5	81.5	2.5 (2.0/3.0)	27.5 (3/52)
compress1	1/1	29.0	8.0	2.0	4.0
compress2	1/1	4.0	24.0	1.7	14.0
corr_gen	1/1	7.0	14.0	2.3	6.0
count	1/1	31.0	6.0	2.0	3.0
dotprod	1/1	2047.0	9.0	2.3	4.0
even_ones	1/1	31.0	6.0	2.0	3.0
expand	1/1	29.0	8.0	2.0	4.0
fdct_8x8	2/2	7.0	117.5	7.6 (7.4/7.8)	15.5 (15/16)
fft	3/2	11.0	34.3	4.9 (4.9/5.4)	7.0 (7/10)
fibonacci	1/1	2378.0	6.0	2.0	3.0
fir	1/1	3.0	11.0	2.2	5.0
gcd2	1/1	65.6	8.0	1.3	6.0
gouraud	1/1	1999.0	15.0	2.5	6.0
hamming_dist	1/1	31.0	6.0	2.0	3.0
lookup2	1/1	499.0	49.0	2.2	22.0
maxstr1	2/2	1.9	7.3	1.9 (1.3/4.8)	3.3 (3/5)
maxstr2	7/2	2.5	5.5	1.3 (1.3/6.0)	4.1 (4/11)
md5	1/1	99.0	837.0	1.9	451.0
mulinv	1/1	17.1	12.0	0.3	36.0
perlins	1/1	1023.0	124.0	4.3	29.0
pix_expand	1/1	4999.0	8.0	2.7	3.0
popcmpr	1/1	8.4	6.0	1.5	4.0
popcnt	1/1	31.0	8.0	2.7	3.0
quantize	1/1	199.0	13.0	1.9	7.0
reverse	1/1	31.0	7.0	2.3	3.0
smooth	0/0	N.A.	N.A.	N/A	N/A
vecsum	1/1	2047.0	11.0	2.8	4.0
wave_horz	2/2	7.0	16.5	0.4	40.5 (40/41)
wave_vert	2/2	7.0	12.5	2.5 (2.4/2.6)	5.0
ycbcr422p_rgb	3/1	9.8	90.0	6.4	14.0
yc_demux_be16	1/1	999.0	22.0	7.3	3.0
yc_demux_le16	1/1	999.0	22.0	7.3	3.0

Table 6.3. Megablock characteristics for the *no-ifs* set, only inner loops.

Benchmark	Megablocks Det./Exec.	Avg. It. per call	Avg. Op. per It.	Avg. ILP (Min/Max)	Avg. CPL (Min/Max)
adpcm_coder	0/0	N.A.	N.A.	N/A	N/A
adpcm_decoder	0/0	N.A.	N.A.	N/A	N/A
boundary	1/1	73.6	12.0	5.0	3.0
bubble_sort	2/1	6.8	9.0	2.5	4.0
change_brightness	3/2	10.2	11.0	2.6 (1.7/2.6)	5.1 (5/7)
compositing	1/1	199.0	18.0	2.0	11.0
conv_3x3	0/0	N.A.	N.A.	N/A	N/A
crc32	0/0	N.A.	N.A.	N/A	N/A
divlu	2/1	2.9	13.0	2.6	5.0
gcd1	2/2	16.4	5.0	2.2 (1.7/2.5)	2.4 (2/3)
idct_8x8_12q4	1/1	7.0	111.0	7.5	15.0
isqrt1	3/1	1.1	39.0	0.8	77.0
isqrt2	2/1	1.8	10.0	3.3	3.0
isqrt3	2/1	1.9	13.0	2.6	5.0
isqrt4	2/1	1.8	21.0	3.8	6.0
mad_16x16	1/1	15.0	13.0	1.8	8.0
mad_8x8	1/1	7.0	13.0	1.8	8.0
max	1/1	185.2	8.0	1.6	5.0
median_3x3	0/0	N.A.	N.A.	N/A	N/A
modexp	1/1	2.0	12.0	0.2	70.0
motion_estimation	1/1	15.0	13.0	2.0	8.0
perimeter	1/1	78.7	19.0	5.3	4.0
pix_sat	2/2	2.0	12.0	1.9	7.0
rgb_to_hsv_int	5/1	1.2	57.0	1.7	38.0
rng	18/3	1.1	53.3	4.9 (4.8/4.9)	13.0
sad_16x16	1/1	15.0	14.0	1.8	8.0
sad_8x8	1/1	7.0	14.0	1.8	8.0
sobel	2/1	3.7	44.0	3.8	13.0
viterbi_gsm	1/1	1.6	49.0	8.1	7.0

Table 6.4. Megablock characteristics for the *ifs* set, only inner loops.

Benchmark	Megablocks Det./Exec.	Avg. It. per call	Avg. Op. per It.	Avg. ILP (Min/Max)	Avg. CPL (Min/Max)
compress2	2/1	999.0	141.0	2.1	66.0
corr_gen	2/1	141.0	121.0	5.8	21.0
fdct_8x8	4/2	49.0	946.0	40.3 (39.6/41.0)	23.5 (23/24)
fft	6/2	11.0	34.3	4.9 (4.9/5.4)	7.0 (7/10)
fir	2/1	252.0	55.0	5.0	11.0
gcd2	2/2	46.6	10.1	1.3 (1.3/1.7)	7.2 (6/103)
maxstr1	9/2	1.6	22.3	2.8 (1.3/3.6)	6.9 (3/9)
maxstr2	27/2	2.5	6.3	1.3 (1.3/5.1)	4.2 (4/14)
mulinv	10/2	2.2	77.8	0.3 (0.3/0.4)	211.1 (36/632)
popcmpr	9/2	1.5	41.2	2.5 (1.5/3.5)	13.1 (4/22)
smooth	2/1	29.0	155.0	7.4	21.0
wave_horz	4/2	49.0	142.5	2.8	50.5 (50/51)
wave_vert	4/2	39.0	110.5	6.9 (6.6/7.2)	16.0
ycbcr422p_rgb	4/1	9.7	90.0	6.4	14.0

Table 6.5. Megablock characteristics for the *no-ifs* set when applying unrolling.

Benchmark	Megablocks Det./Exec.	Avg. It. per call	Avg. Op. per It.	Avg. ILP (Min/Max)	Avg. CPL (Min/Max)
adpcm_coder	4/1	1.0	80.0	6.4	16.0
adpcm_decoder	9/1	1.2	64.0	6.7	12.0
conv_3x3	4/2	2.0	88.1	8.8 (7.0/19.8)	13.1 (13/14)
crc32	6/2	1.6	6.1	2.0 (2.0/4.3)	3.0 (3/14)
idct_8x8_12q4	2/1	49.0	868.0	36.5	25.0
isqrt1	9/2	1.3	22.9	0.3 (0.2/0.5)	82.2 (37/148)
isqrt3	8/1	1.9	15.0	3.0	5.0
isqrt4	4/2	1.6	22.9	3.9 (3.8/5.0)	7.1 (6/30)
mad_16x16	2/1	15.0	163.0	9.6	24.0
mad_8x8	2/1	7.0	83.0	7.4	16.0
modexp	7/3	2.0	12.7	0.2 (0.2/0.3)	70.5 (70/105)
motion_estimation	2/1	15.0	165.0	11.0	24.0
pix_sat	8/2	2.0	12.0	1.9	7.0
rgb_to_hsv_int	11/3	1.2	60.9	1.8 (1.6/3.2)	38.3 (38/41)
rng	15/3	1.1	53.3	4.9 (4.8/4.9)	13.0
sad_16x16	2/1	15.0	178.0	9.5	24.0
sad_8x8	2/1	7.0	90.0	7.3	16.0
sobel	8/1	3.7	44.0	3.8	13.0
viterbi_gsm	7/3	4.8	33.4	6.6 (4.3/14.3)	4.7 (3/8)

Table 6.6. Megablock characteristics for the *ifs* set when applying unrolling.

The column *Megablocks Det./Exec.* shows how many Megablocks were found in the benchmarks, and how many of them could be used after identification. When considering the

no-ifs set, in most cases only one Megablock is detected. Unrolling increases the number of detected Megablocks in the affected benchmarks as besides the previously detected loops, it will also detect outer loops with the inner loops unrolled. Unrolling can detect loops in cases where no loops are found when looking for only inner loops (e.g., as in *smooth*). However, this strategy is less effective when the number of iterations of inner loops is variable.

Columns *Avg. It. per call and Avg. Op. per It.* represent the average number of iterations per Megablock call, and the average number of executed operations per Megablock iteration, respectively. The higher the number of iterations and the number of operations per iteration, the longer the Megablock executes uninterruptedly in the RPU, diminishing the impact of communication overhead.

Columns *Avg. ILP* and *Avg. CPL* are a weighted average of the ILP and CPL of the executed Megablocks, respectively. If the minimum and/or the maximum value are different from the average, they are presented between parentheses.

Considering the *innerloops* case, the ILP of the Megablocks ranges between 0.3 and 7.8 (average of 2.9) for the *no-ifs* set and between 0.2 and 8.1 (average of 3.0) for the *ifs* set. For the same sets, the CPL ranges between 3 and 451 (average of 22.2) and between 2 and 77 (average of 13.9). After unrolling, the average ILP increases to 4.3 and 5.7 and the average CPL increases to 30 and 16.3, for the *no-ifs* and *ifs* set, respectively. Unrolling inner loops creates larger Megablocks with larger CPL, which can translate to Megablocks which execute uninterruptedly on the RPU for longer periods. The larger ILP increases the parallelism potential.

Some benchmarks have ILP below 1 (e.g., *mulinv*, *wave_horz*, *isqrt1*, *modexp*). All cases correspond to Megablocks which have high-latency instructions, such as integer division, which take several cycles to finish execution (e.g., an integer division operation has a latency of 32 clock cycles in the considered architecture).

Figure 6.4 presents upper bound speedups considering three scenarios: in the Megablock (CPL based) scenario, the speedup is estimated considering the CPL of the baseline graph of the Megablock. This is equivalent to perform mapping with as many resources as needed. In the Megablock (Zero Cycles) scenario, the execution time of the RPU is considered to be zero, but considering communication overhead. This represents the maximum theoretical speedup possible with detected Megablocks. The final scenario considers that both the RPU execution time and communication time are zero.

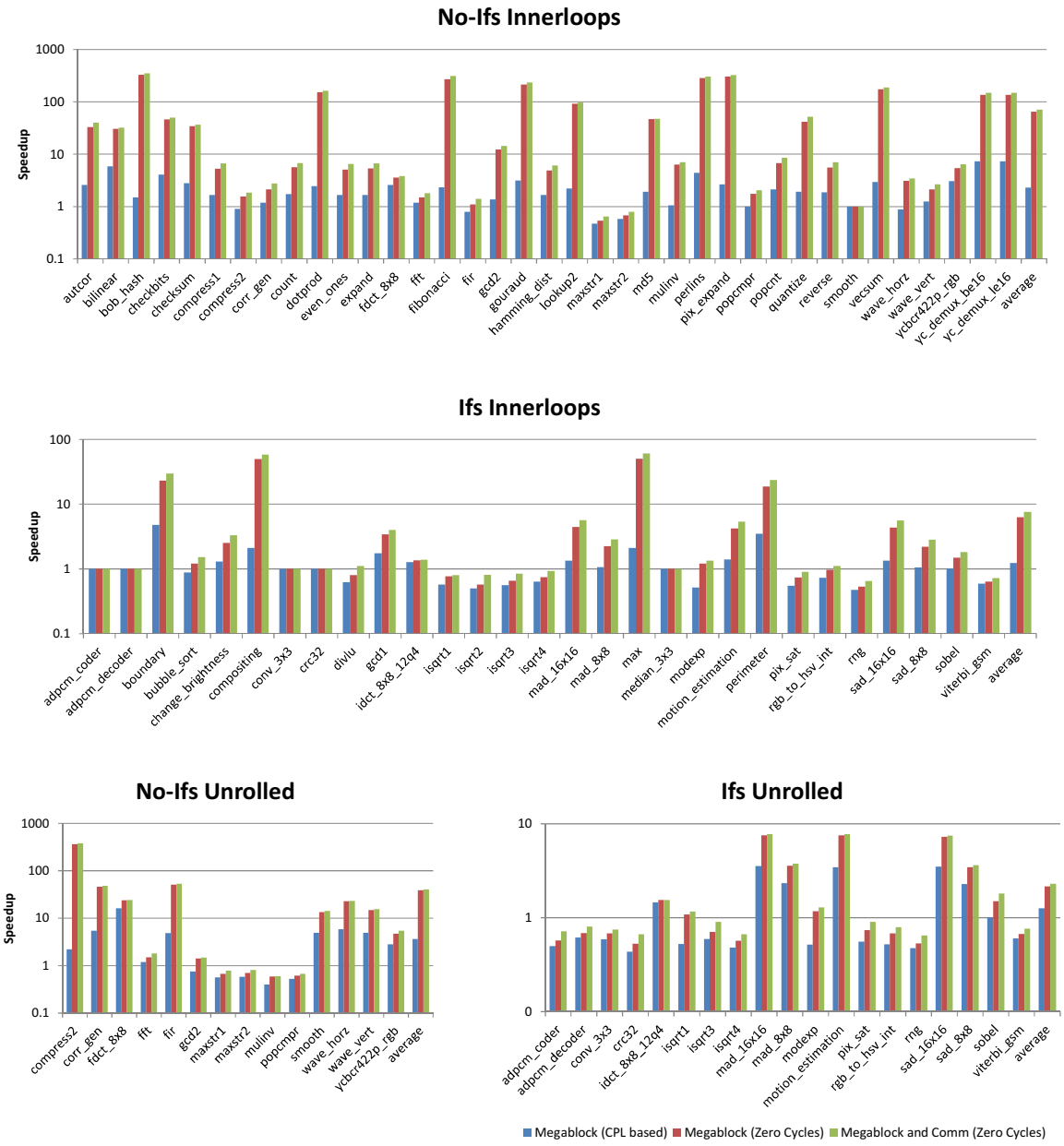
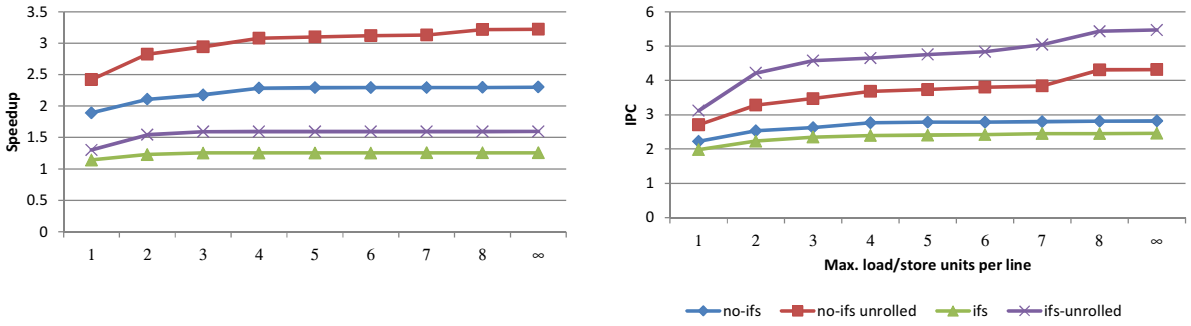


Figure 6.4. Upper-bound speedups in the baseline case for three scenarios: execution time of the RPU is equal to Megablock CPL, execution time of the RPU is zero and execution time of the RPU and communication delays are zero.

Generally, benchmarks without branches in the loop show a much higher potential for speedup. The average for the *no-ifs* set is one order of magnitude above the average for the *ifs* set. Unrolling can have a positive effect in some of the affected benchmarks. There is a big gap between the potential speedup when considering a straightforward implementation of the

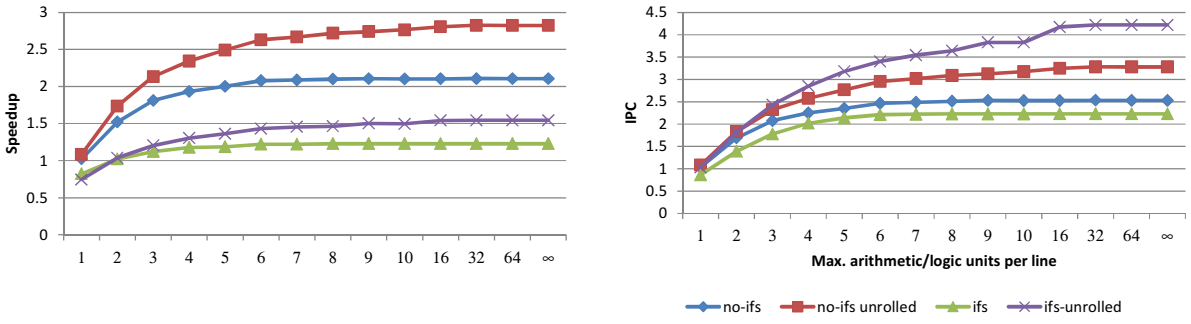
Megablock (CPL based) and the upper bound scenario (Zero Cycles). This gap suggests there is ample head room for performance improving techniques (e.g., loop pipelining).

Figure 6.5 and Figure 6.6 present estimation speedups and Instructions Per Cycle (IPC), when varying the number of available load/store units and the number of arithmetic-logical units, respectively. In Figure 6.5, the mapping was done using as many arithmetic-logic units as needed; in Figure 6.6 we limited the number of concurrent memory units to two. The lines represent arithmetic average values over the speedup and overall IPC of each benchmark of the set (in the sets where the *unrolled* option was used, we are considering all the benchmarks of the set, and not just the benchmarks where unrolling had impact).



a) **b)**

Figure 6.5. Average a) speedup and b) IPC when varying the maximum number of load/store units per row.



a) **b)**

Figure 6.6. Average a) speedup and b) IPC when varying the maximum number of arithmetic/logic units per row.

In Appendix B, Section B-1, we present the results using the geometric mean instead. In this case, the curves maintain their relative positions, although the absolute value is lower, and the gap between the *innerloops* case and the *unrolled* case shortens.

Regarding the number of load/store units, the greatest increase in speedup is when passing from 1 unit to 2 units (improvement of 11% for the *no-ifs* set with *innerloops*, and 17% when using unrolling). There are further improvements when adding concurrent memory accesses. However, the additional complexity of a larger number of concurrent memory operations can outweigh the benefit in speedup.

The benefit of adding parallel FUs becomes less effective at an earlier point in the *innerloops* case than the *unrolled* case. Unrolling exposes more parallelism, which can take advantage of a higher number of parallel FUs. For the baseline case in the *no-ifs* set, with a relatively small number of maximum FUs per row (e.g., 8 FUs) we can achieve 99% (*innerloops*) and 94% (*unrolled*) of the speedup when using unlimited resources.

Considering a default setup with 2 concurrent memory units and 8 parallel FUs, with the baseline system we achieve an average overall application speedup of 2.1× and 1.2× when considering the *no-ifs* and the *ifs* set, for the case *innerloops*, respectively. We consider this architecture setup can represent a typical implementation, and is referred herein as *8 FUs-2Mem*.

Unrolling increments the average speedup in both cases, to 2.7× and 1.4×, due to new Megablocks being detected in benchmarks were no Megablocks were detected before (e.g., *smooth*), or by detecting Megablocks which increase the coverage relative to inner loop detection (e.g., *compress2, fdct, fir*). In some cases, unrolling *decreases* the speedup (e.g., *gdc2, popcmpr*), but the increase in speedup in the other benchmarks compensates for these cases. This behavior is related to loops which have a variable number of iterations. When considering only inner loops, they can be successfully detected as Megablocks. However, when unrolling, loops with a different number of iterations will be detected as different Megablocks. If these Megablocks have SAI conflicts, the identification will not be as effective as when using only inner loops. Thus, unrolling does not always mean improvement.

The previous results assume that the processor and the RPU work at the same clock frequency. Figure 6.7 presents how the speedup varies when considering different ratios between the clock of the processor and of the RPU, for the case where the mapping can assign as many arithmetical-logical units as needed, and restricted to two memory operations per

cycle. For instance, a ratio of 1.5 means that if the processor is clocked at 100 MHz, the RPU is clocked at 150 MHz.

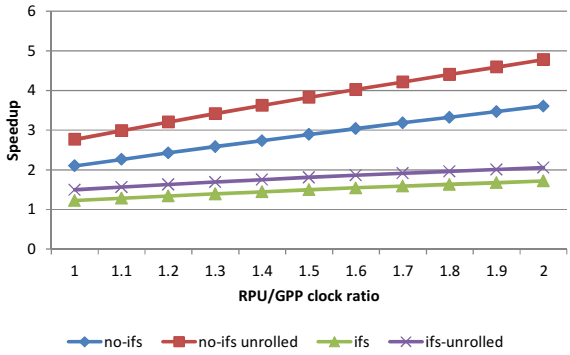


Figure 6.7. Average speedup when varying the ratio between the RPU and GPP clock frequencies.

Doubling the frequency of the RPU with respect to the processor increases the speedup by 1.7× and 1.4×, for the *no-ifs* and the *ifs* set, respectively. The increase in the *no-ifs* set is greater because its benchmarks spend a longer portion of the execution time in the RPU than the benchmarks in the *no-ifs* set.

Figure 6.8 presents individual speedups for the baseline case, when using the setup 8 FUs-2Mem. Overall, considering the complete set of 66 benchmarks, for the *innerloops* case we achieve speedups from 0.5× to 4.8×, with an average speedup of 1.7× (or 1.4×, when using the geometric mean). When activating unrolling of inner loops, we achieve speedups from 0.4× to 6.4×, with an average speedup of 2.2× (or 1.6×, when using the geometric mean). After applying *if-conversion* and graph transformation techniques, the average speedups increase slightly to 1.8× and 2.4× when using the arithmetic mean, and 1.6× and 2.1× when using the geometric mean, for the *innerloops* and *unrolled* cases respectively.

When considering only the benchmarks which provide speedup, for the *innerloops* case we achieve an average speedup of 2.4× (from 1.0× to 4.8×) over 31 benchmarks for the *no-ifs* set, and an average speedup of 1.8× (from 1.0× to 4.8×) over 13 benchmarks for the *ifs* set. When considering unrolling of inner loops, in the *no-ifs* set the average speedup increases to 3.1× (from 1.2× to 6.4×) over a set of 32 benchmarks, and in the *ifs (adapted)* set the average speedup increases to 2.5× (from 1.0× to 4.8×) over a set of 22 benchmarks.

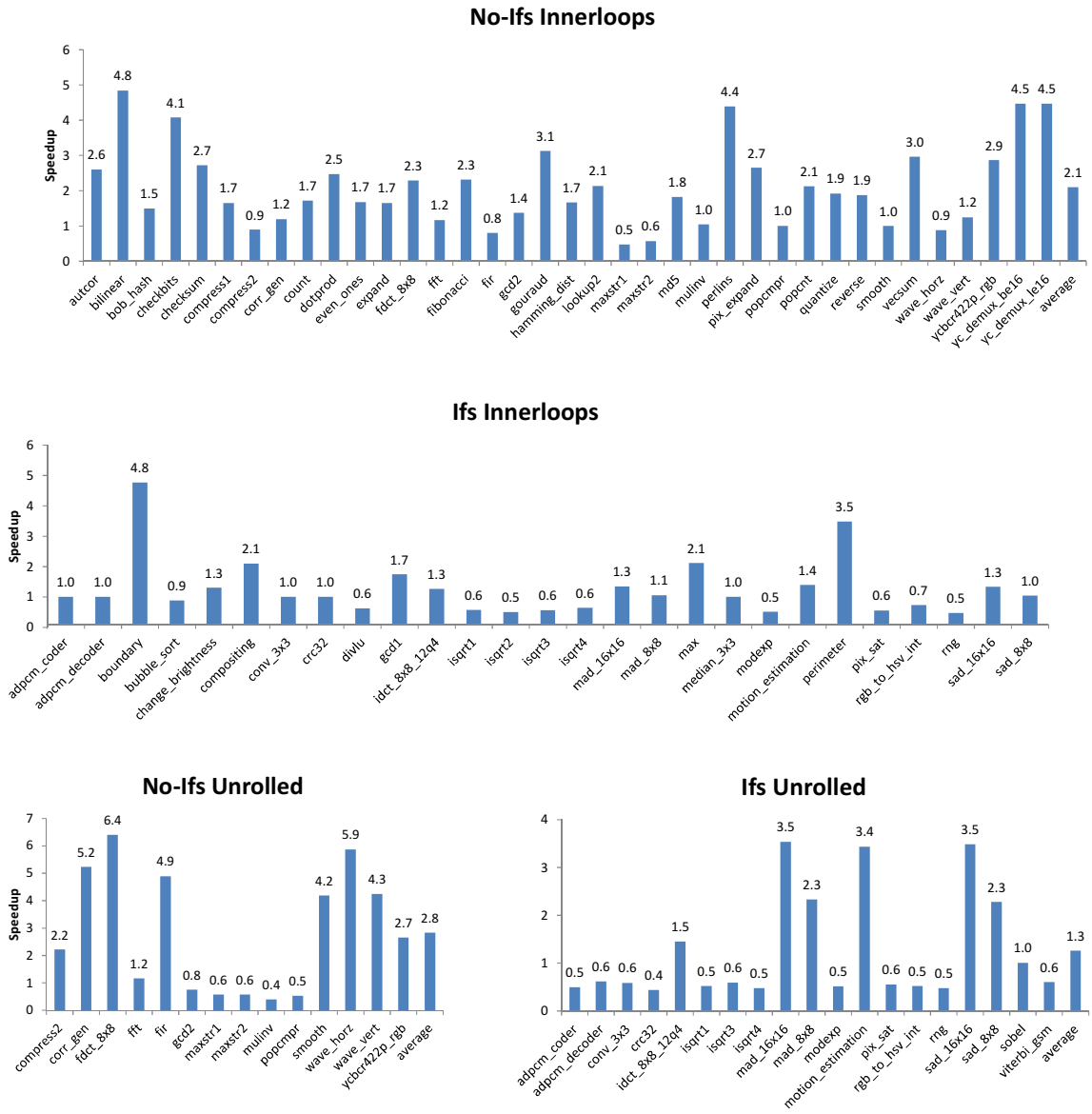


Figure 6.8. Individual overall speedups for the baseline case, considering an RPU with a maximum of 8 parallel FUs and 2 load/store operations per cycle.

6.3.2 If-Conversion

We applied *if-conversion* techniques (see Section 4.5) to the source code of the *ifs* set. We named the resulting set *ifs (adapted)*. Table 6.7 shows the clock cycles needed to execute the benchmarks in the MicroBlaze processor, after and before the *if-conversion*, and the ratio between them. A value greater than one means an increase in the number of clock cycles for the adapted benchmark. For all benchmarks in the *ifs* set, there was an increase in the execution time, which ranged from a negligible increase (1.04× in *conv_3x3*) to more than

twice the execution time (2.29× in *median_3x3*). The examples with higher increases correspond to cases whose kernels also have a higher number of *if* statements (see Table 6.2).

Benchmark	Original (#Clock Cycles)	Adapted (#Clock Cycles)	Ratio
adpcm_coder	61,841	92,308	1.49
adpcm_decoder	46,516	83,079	1.79
Boundary	180,730	291,024	1.61
bubble_sort	52,095	97,325	1.87
change_brightness	15,610	23,553	1.51
Compositing	48,283	56,293	1.17
conv_3x3	14,933	15,587	1.04
crc32	79,562	141,093	1.77
Divlu	456,708	533,093	1.17
gcd1	678,177	1,099,497	1.62
idct_8x8_12q4	150,679	308,851	2.05
isqrt1	183,449	190,403	1.04
isqrt2	185,919	243,093	1.31
isqrt3	29,527	31,993	1.08
isqrt4	15,932	21,493	1.35
mad_16x16	1,010,712	1,209,363	1.20
mad_8x8	261,145	312,339	1.20
Max	22,626	30,820	1.36
median_3x3	75,225	172,159	2.29
Modexp	1,875,680	2,485,980	1.33
motion_estimation	1,088,464	1,285,122	1.18
Perimeter	10,224	15,007	1.47
pix_sat	31,116	44,103	1.42
rgb_to_hsv_int	65,175	115,113	1.77
Rng	41,447	43,150	1.04
sad_16x16	39,515	47,195	1.19
sad_8x8	20,535	24,375	1.19
Sobel	49,953	59,534	1.19
viterbi_gsm	117,157	133,600	1.14
Average	-	-	**1.41**

Table 6.7. Cycle count and ratio of the *ifs* set, before and after *if-conversion*.

In this thesis, all speedups related to benchmarks which were modified by *if-conversion* techniques, such as the *ifs (adapted)* set, are relative to the execution time of the original unmodified program, and can be directly compared with all the other speedups (e.g., the speedups of the *ifs* set). The IPC values reflect the changes in the adapted code.

Table 6.8 presents some of the characteristics of the benchmarks in the *ifs (adapted)* set after adapting the source code of the *ifs* set.

Benchmark	Kernel LOC	#loops	Max. Loop nesting level	#Input/Output Arrays
adpcm_coder	66	1	0	1/1
adpcm_decoder	50	1	0	1/1
Boundary	13	2	1	1/2
bubble_sort	17	2	1	1/0
change_brightness	19	1	0	1/1
Compositing	15	1	0	2/1
conv_3x3	40	2	1	2/1
crc32	11	1	0	0/0
Divlu	15	1	0	0/0
gcd1	16	1	0	0/0
idct_8x8_12q4	177	4	1	1/1
isqrt1	33	1	0	0/0
isqrt2	15	1	0	0/0
isqrt3	17	1	0	0/0
isqrt4	15	1	0	0/0
mad_16x16	26	4	3	2/1
mad_8x8	25	4	3	2/1
Max	10	1	0	1/0
median_3x3	73	1	0	1/1
Modexp	11	1	0	0/0
motion_estimation	18	4	3	2/1
Perimeter	24	1	0	1/1
pix_sat	12	1	0	1/0
rgb_to_hsv_int	24	1	0	3/1
Rng	24	1	0	1/1
sad_16x16	11	2	1	1/1
sad_8x8	11	2	1	1/1
sobel	28	1	0	1/1
viterbi_gsm	29	4	2	3/2

Table 6.8. Characteristics of the benchmarks which form the set *ifs (adapted)*.

Table 6.9 and Table 6.10 present detected Megablock characteristics of the *ifs (adapted)* set, for the *innerloops* and *unrolled* cases, respectively. Applying *if-conversion* had two effects in Megablock detection and execution. In one hand, it enabled the detection of Megablocks on benchmarks were previously there were no Megablocks detected (e.g., *adpcm_coder*, *adpcm_decoder*, *crc32*, *median_3x3*). In the other hand, it decreased the

number of total detected Megablocks in other benchmarks, making the ratio between detected Megablocks and executed Megablock equal to one (i.e., all detected Megablocks are executed) in many cases (e.g., *change_brightness*, *rgb_to_hsv_int*, *rng*, *sobel*).

Benchmark	Detected/ Executed Megablocks	Avg. It. per call	Avg. Op. per It.	Avg. ILP (Min/Max)	Avg. CPL (Min/Max)
adpcm_coder	1/1	1023.0	88.0	2.1	41.0
adpcm_decoder	1/1	1023.0	79.0	2.2	36.0
boundary	1/1	99.0	27.0	3.9	7.0
bubble_sort	1/1	62.0	23.0	2.3	10.0
change_brightness	1/1	99.0	22.0	1.7	13.0
compositing	1/1	199.0	27.0	1.7	16.0
conv_3x3	0/0	N.A.	N.A.	N/A	N/A
crc32	1/1	7.0	12.0	2.0	6.0
divlu	1/1	31.0	15.0	1.9	8.0
gcd1	1/1	166.2	11.0	1.4	8.0
idct_8x8_12q4	2/2	7.0	355.0	15.3 (9.5/21.0)	21.0 (15/27)
isqrt1	2/1	1.1	70.0	0.7	99.0
isqrt2	1/1	15.0	13.0	2.2	6.0
isqrt3	1/1	15.0	17.0	2.1	8.0
isqrt4	1/1	5.0	32.0	2.7	12.0
mad_16x16	1/1	15.0	17.0	1.9	9.0
mad_8x8	1/1	7.0	17.0	1.9	9.0
max	1/1	2047.0	14.0	1.4	10.0
median_3x3	1/1	998.0	172.0	2.5	69.0
modexp	1/1	29.8	17.0	0.5	37.0
motion_estimation	1/1	15.0	19.0	2.1	9.0
perimeter	1/1	479.0	30.0	3.0	10.0
pix_sat	1/1	1999.0	21.0	1.5	14.0
rgb_to_hsv_int	1/1	499.0	162.0	3.4	47.0
rng	1/1	498.0	70.0	4.4	16.0
sad_16x16	1/1	15.0	17.0	1.9	9.0
sad_8x8	1/1	7.0	17.0	1.9	9.0
sobel	1/1	957.0	61.0	3.1	20.0
viterbi_gsm	2/2	7.0	41.0	4.8 (4.3/5.3)	8.0 (3/13)

Table 6.9. Megablock characteristics for the *ifs-adapted* set, only inner loops.

When comparing the *ifs (adapted)* set with the *ifs* set, the number of iterations generally increases. As the *if-conversion* technique includes several paths in the same Megablock, it can execute uninterruptedly in the RPU for a higher number of iterations.

Benchmark	Detected/ Executed Megablocks	Avg. It. per call	Avg. Op. per It.	Avg. ILP (Min/Max)	Avg. CPL (Min/Max)
conv_3x3	1/1	149.0	99.0	4.5	22.0
crc32	2/1	999.0	109.0	2.2	50.0
idct_8x8_12q4	4/2	49.0	2855.5	82.2 (44.5/119.9)	32.0 (26/38)
isqrt1	4/1	1.7	82.0	0.5	169.0
isqrt2	2/1	999.0	225.0	2.3	99.0
isqrt3	2/1	99.0	287.0	2.5	114.0
isqrt4	2/1	99.0	208.0	2.8	75.0
mad_16x16	2/1	15.0	278.0	11.1	25.0
mad_8x8	2/1	7.0	142.0	8.4	17.0
motion_estimation	2/1	15.0	312.0	12.5	25.0
sad_16x16	2/1	15.0	277.0	11.1	25.0
sad_8x8	2/1	7.0	141.0	8.3	17.0

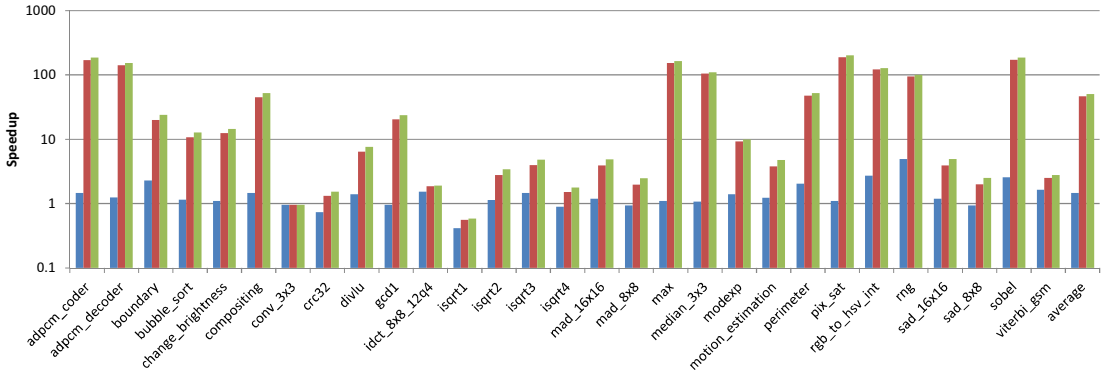
Table 6.10. Megablock characteristics for the *ifs-adapted* set when applying unrolling.

For the *innerloops* case, the average ILP decreases slightly (from 3.0 to 2.7), while in the *unrolled* case, it increases (from 5.7 to 6.5). The change in ILP between the *ifs* and *ifs (adapted)* sets is not significant (less than one operation), when considering the impact of *if-conversion* on the value of ILP of the whole set. However, the individual ILP does not have a general behavior and its increase or decrease depends on the benchmark.

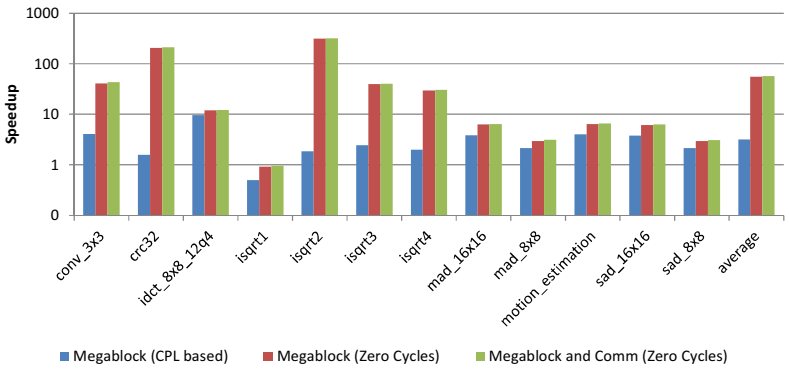
The average CPL significantly increases, from 13.9 to 20.3 and from 16.3 to 35.9 for the *innerloops* and *unrolled* cases, respectively, reaching similar values to the corresponding average CPL of the *no-ifs* set (22.2 and 30.0).

Figure 6.9a) and Figure 6.9b) presents upper bound speedups for the *ifs (adapted)* set, when considering only inner loops and inner loop unrolling, respectively. The average speedup potential of the *ifs* set is one order of magnitude below the potential of the *no-ifs* set (see Figure 6.4). After *if-conversion* the gap almost disappears.

If-conversion increases both the average speedup and the average IPC (see Figure 6.10 and Figure 6.11). For the *ifs (adapted)* set, the effect of enabling unrolling has a more pronounced effect in the values of speedup. The higher steepness of the slop in the clock ratio lines of the *ifs (adapted)* set (see Figure 6.12) is related to a higher portion of RPU execution. Appendix B, Section B-2, presents the results using the geometric mean instead. In this case, the performance of the adapted sets continues to be consistently above the performance of the unmodified sets.

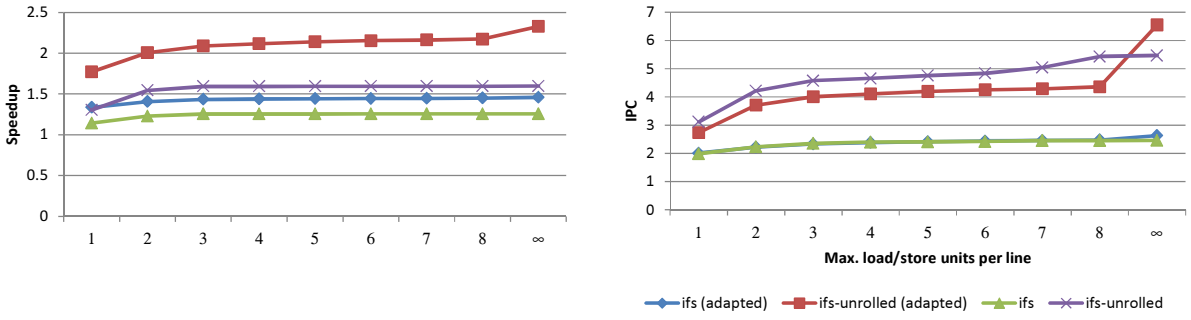


a)



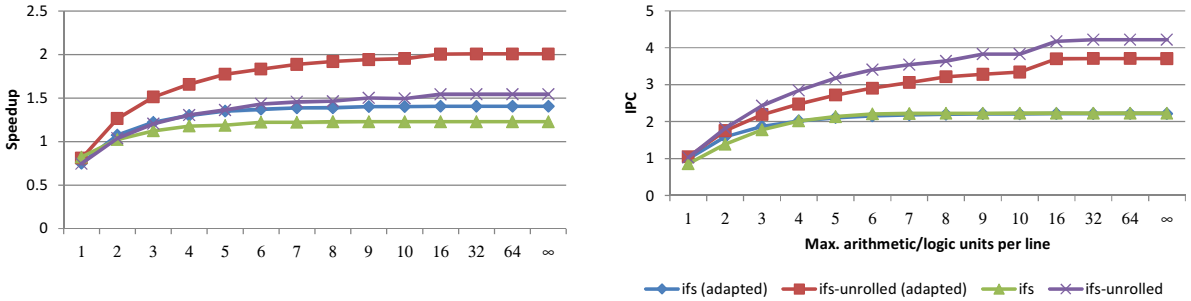
b)

Figure 6.9. Upper bound speedups after *if-conversion* a) when considering inner loops and b) when unrolling inner loops.



a) **b)**

Figure 6.10. Average a) speedup and b) IPC for adapted code when varying the maximum number of load/store units per row.



a) **b)**

Figure 6.11. Average a) speedup and b) IPC for adapted code when varying the maximum number of arithmetic/logic units per row.

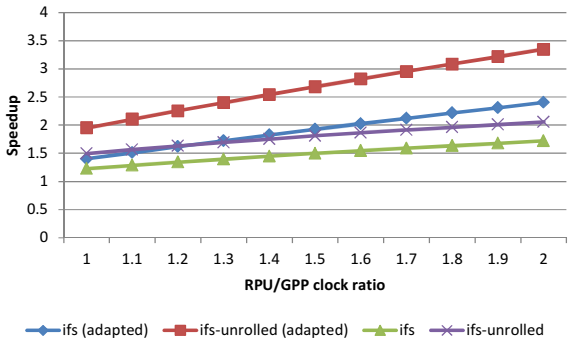


Figure 6.12. Average speedup for adapted code when varying the ratio between the RPU and GPP clock.

6.3.3 Graph Transformations

We can apply several transformations over the Megablock graph representation (see Section 5.1). For instance, we can apply Constant Folding and Propagation (CFP) to reduce the number of operations of the graph. We consider here three transformations in sequence: assembly instructions to intermediate representation (IR), Constant Folding and Propagation (IR + CFP), and Identity Simplifications (IR + CFP + IS).

When applying them to the *innerloops* case we did not observe significant changes. However, the *unrolled* case offered more opportunities for the considered transformations. Table 6.11 and Table 6.12 show the difference between the original number of instructions in the Megablock and the number of operations after the transformations when enabling unrolling, for the *no-ifs* and *ifs (adapted)* sets, respectively. A negative value represents a decrease in the number of operations when compared with the original assembly instructions.

Benchmark	IR	IR + CFP	IR + CFP + IS
compress2	3%	-14%	-16%
corr_gen	26%	8%	-9%
fdct_8x8	-9%	-11%	-13%
fft	30%	30%	15%
fir	20%	-2%	-17%
gcd2	0%	0%	0%
maxstr1	10%	-13%	-19%
maxstr2	2%	-2%	-20%
mulinv	1%	-9%	-41%
popcmpr	5%	4%	3%
smooth	14%	-21%	-40%
wave_horz	13%	-1%	-6%
wave_vert	18%	-10%	-24%
ycbcr422p_rgb	8%	1%	-5%
average	10%	-3%	-14%

Table 6.11. Decrease in the number of Megablock operations, for the unrolled *no-ifs* set considering three transformations.

Benchmark	IR	IR + CFP	IR + CFP + IS
conv_3x3	22%	7%	-5%
crc32	-16%	-36%	-38%
idct_8x8_12q4	25%	24%	23%
isqrt1	1%	-7%	-16%
isqrt2	1%	-30%	-31%
isqrt3	-4%	-23%	-24%
isqrt4	1%	-15%	-16%
mad_16x16	13%	-7%	-14%
mad_8x8	13%	-7%	-15%
motion_estimation	18%	0%	-19%
sad_16x16	13%	-7%	-8%
sad_8x8	13%	-7%	-9%
average	7%	-9%	-14%

Table 6.12. Decrease in the number of Megablock operations, for the unrolled *ifs-adapted* set considering three transformations.

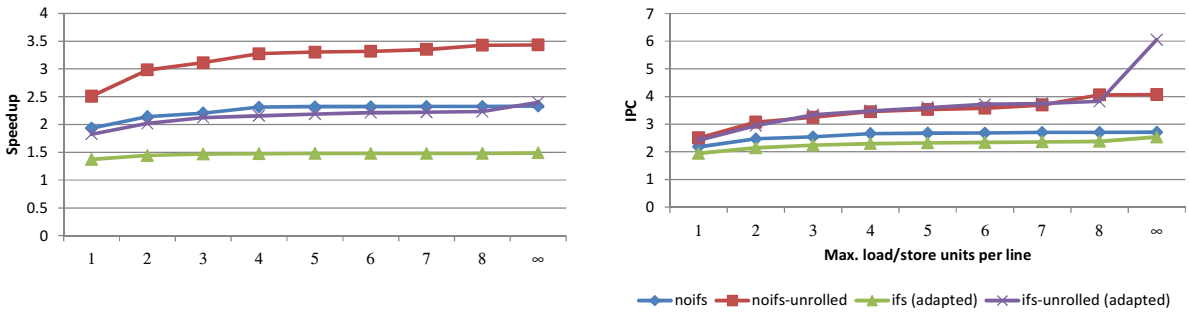
The first column represents the ratio when the assembly instructions are converted into the IR. The conversion can either increase or decrease the number of operations, depending on the instructions being converted. Converting memory instructions increases the number of operations, due to the unfolding of the instruction into the operations to calculate the address and the memory operation. Other instructions, such as *nops* or auxiliary instructions such as

imm, contribute to a reduced number of operations. On average, we have an increase of 10% and 7%, for the *no-ifs* and *ifs (adapted)* sets, after transforming the instructions to the IR.

Applying CFP generally decreases the number of operations, to -3% and -9% (*no-ifs* and *ifs (adapted)*) of the original assembly instructions on average. In most cases, the reduction provided by CFP is greater than the increase that results from transforming the instructions into the IR. Applying the IS transformation further decreases the number of operations. The effect is more pronounced in the *no-ifs* set than in the *ifs (adapted)* set. Considering this sequence of transformations (i.e., IR+CFP+IS), the reduction is on average the same in both sets (-14% for the *no-ifs* and *ifs (adapted)* sets)

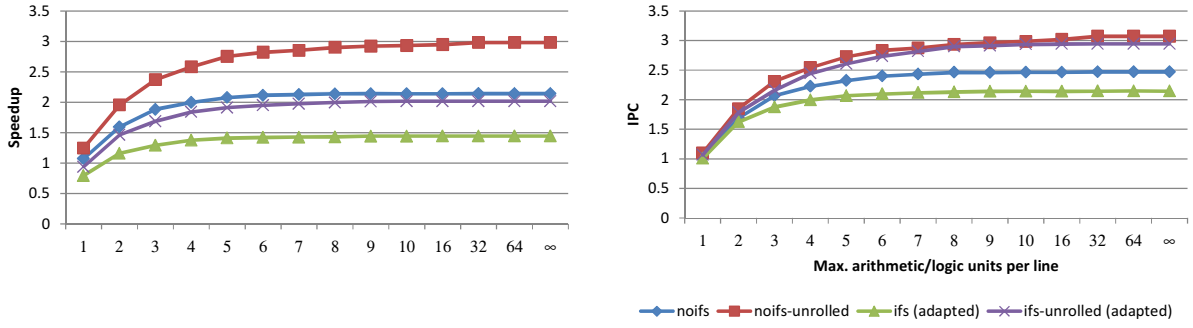
When considering the *unrolled* case, we observed that generally the transformations resulted in minor increases in the speedup and minor decreases in the IPC (see Figure 6.13, Figure 6.14, and Figure 6.15).

For the 8 FUs-2Mem configuration there was an increase of 1.07× and 1.03×, and a decrease in the IPC to 95% and 92% of the original value, for the *no-ifs* set and the *ifs (adapted)* set, respectively. The increase in speedup can be attributed to a decrease in the CPL of the Megablock in a few benchmarks. The most noticeable speedup was observed for the benchmark *fir* (1.5×). The decrease in IPC was expected, since on average, the transformations reduced the number of operations by 14%, for the affected Megablocks. While these transformations did not affect performance significantly, they are useful to lower the mapping effort, by reducing the size of the Megablocks to implement.



a) **b)**

Figure 6.13. Average a) speedup and b) IPC after graph transformations, when varying the maximum number of load/store units per row



a) **b)**

Figure 6.14. Average a) speedup and b) IPC after graph transformations, when varying the maximum number of arithmetic/logic units per row.

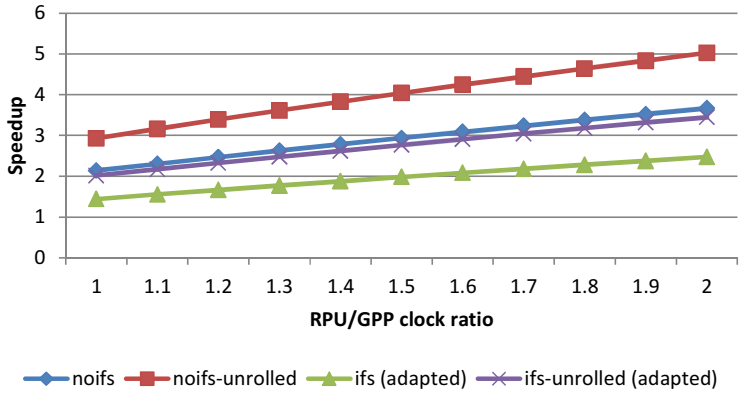


Figure 6.15. Average speedup after graph transformations, when varying the ratio between RPU and GPP clock frequencies.

6.4 Hardware Module for Megablock Detection

We developed a proof-of-concept HDL generator which outputs VHDL for Megablock Detectors, as depicted in Figure 5.3, according to several parameters (see Appendix C, Figure C.4). Figure 6.16 presents the resources needed to implement the Megablock Detector hardware module, when varying the *maximum pattern size* and the bit-width of the pattern element.

For the explored parameter ranges, the number of LUTs and FFs resources increases linearly with the increase of the *maximum pattern size*. Higher bit widths generally represent a higher number of used resources, although the increase is more significant for FFs than for LUTs. The behavior of the LUT resources is more irregular than the behavior of the FFs. We

attribute this behavior to the way the synthesis tool maps certain FPGA primitives (e.g., SRLs), used in the VHDL code.

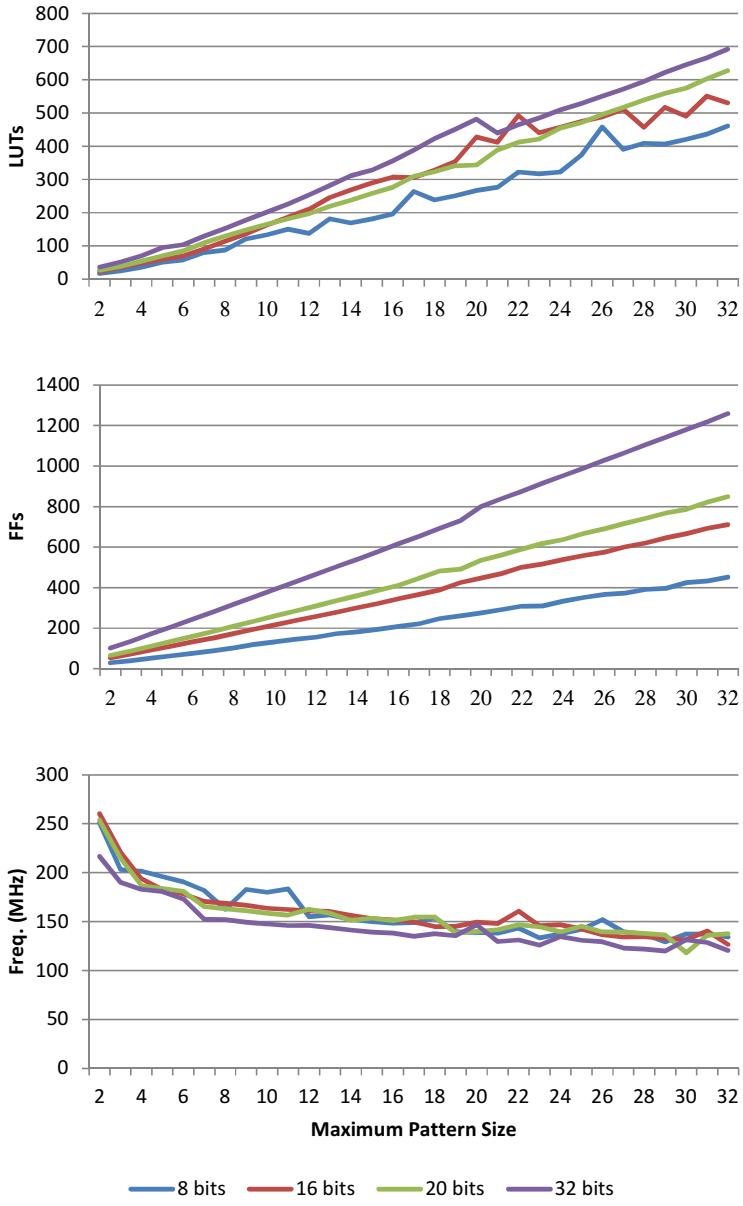


Figure 6.16. LUTs, FFs and estimated maximum frequencies for Megablock Detector hardware designs.

For the base case with a *maximum pattern size* of 24 elements, and considering an address space for instructions of 20 bits, the module needs 455 LUTs and 636 FFs, which represent around 1% of the targeted FPGA (a Xilinx Spartan-6 LX45). These values include the encoder and the state machine for determining the current state of the detector. The decrease of the maximum clock frequency with the increase of the *maximum pattern size* was expected,

as higher values for the *maximum pattern size* implies more complex logic paths in some parts of the Megablock Detection module (e.g., the comparison between the current pattern element and all the positions in the FIFO). There are engineering solutions which can mitigate the decrease in frequency, at the cost of some latency (e.g., by using fan-out trees).

However, the current implementation works at sufficient speed for the considered scenarios. For instance, considering the base case of a *maximum pattern size* of 24 elements, the maximum estimated clock frequency is between 134 MHz and 147 MHz (depending on the bit-width of the elements), which is enough to meet the clock frequency of the MicroBlaze soft processor.

Higher bit-widths generally produced designs with lower clock frequencies, although the impact is relatively small. The maximum impact of the bit-width on the clock frequency is on average 14% for the cases studied.

6.5 Megablock Pipelining

To demonstrate the proposed pipelining technique (presented in Section 5.6), we developed a proof-of-concept VHDL generator which converts a Megablock graph into a specialized hardware module (see Figure B.9). For a given Megablock, the tool can generate two types of modules: an implementation of the SAr architecture described in Section 5.5.2, and a pipelined version of the same architecture using the overlapping scheduled. The generation of the pipelined architecture is performed after applying *if-conversion* (when applicable) and the Megablock graph transformations described in Section 5.1. The version of the generator used herein does not support Megablocks with memory operations.

For the first part of this section, we consider a set of simple benchmarks without memory operations, named *memoryless* (i.e., *compress1*, *count*, *even_ones*, *expand*, *fibonacci*, *hamming_dist*, *popcmpr*, *reverse*, and *gcd1*). We implemented Megablocks representing a kernel of each benchmark, and estimated the clock cycles needed by the hardware module for executing the pipelined and the non-pipelined versions of the considered Megablocks using equation (5.1). We also estimated overall application speedups, taking into account all communication overheads. Considering the set *memoryless*, we synthesized two versions of the hardware module, with and without pipelining, to measure resource usage, confirm the execution cycles and validate the approach. In the end of the this section, we present overall

speedup estimations after pipelining considering the *no-ifs* and the *ifs (adapted)* sets of the previous section.

We consider that communication between the GPP and the hardware module is done through FSLs (Fast Simplex Link) [90] using *get* and *put* instructions, one for each Megablock input or output, respectively. Each one of these instructions takes one clock cycle to execute [90]. Based on an implementation (presented in Appendix A) of the architecture of Figure 5.10b), we estimate that the value for the term $\text{Partitioner}_{\text{Cy}}$ in equation (5.4) has a constant overhead of 8 clock cycles per Megablock call.

The SAr architecture can have as many FUs in a row and as many exits per row as needed. We defined the execution cycles of the operations as identical to the clock cycles needed by the MicroBlaze for equivalent instructions, when the processor is optimized for speed [90]. Similarly to other approaches [14], we assume each memory operation can be done in a single clock cycle. We also consider that the RPU is connected to local memories which support up to two simultaneous memory operations per cycle (e.g., dual-port BRAMs).

Table 6.13 presents the overall application IPC (Instructions per Cycle) achieved considering the Megablock for each benchmark in different kinds of RPUs. As expected, the IPC considering Megablocks and the RPU is higher than the IPC achieved by the MicroBlaze, which is below 1. Since all the RPUs used have several FUs executing in parallel, the IPC usually increases, proportionally to the ILP of the Megablock.

With pipelining, more than one row of the RPU is executing per clock cycle (in the steady state, all FUs execute in parallel in each iteration), and the IPC relative to the non-pipelined RPU increases. As expected, the IPC of overlapped schedule is consistently higher than the IPC of the sequential schedule.

Table 6.14 summarizes the characteristics of the Megablock considered for each benchmark of the *memoryless* set, when mapped to the non-pipelined version of the SAr. The number of operations in the Megablocks ranges between 4 and 11. When mapped to the non-pipelined SAr the number of rows ranged between 3 and 8, with the largest row having 3 FUs. The number of iterations per call ranges from around 8 (*popcmpr*) to a few thousands (*fibonacci*), having most benchmarks a number of iterations around 30.

Benchmark	MicroBlaze	RPU Non-Pipelined	RPU Pipelined Sequential	RPU Pipelined Overlapping
compress1	0.88	2.00	3.03	3.72
count	0.85	2.00	2.53	3.30
even_ones	0.85	2.00	3.36	4.93
expand	0.88	2.00	3.03	3.72
fibonacci	0.86	1.33	3.01	5.90
hamming_dist	0.85	2.00	3.36	4.93
popcmpr	0.88	1.50	3.41	4.80
reverse	0.87	2.33	3.03	3.96
gcd1	0.70	1.38	2.58	3.00

Table 6.13. IPC when the Megablock for each benchmark is executed in several platforms.

Benchmark	FUs	Rows	Max. FUs p/ row	Avg. Iterations p/ call
compress1	8	4	3	29.0
count	6	3	2	31.0
even_ones	6	3	3	31.0
expand	8	4	3	29.0
fibonacci	4	3	2	2378.0
hamming_dist	6	3	3	31.0
popcmpr	6	4	2	8.4
reverse	7	3	3	31.0
gcd1	11	8	3	166.2

Table 6.14. Megablock mapping characteristics on the non-pipelined architecture.

Table 6.15 and Table 6.16 present a comparison between the overall speedup when using non-pipelined and pipelined RPUs with a sequential and an overlapping schedule, respectively. The first two columns, “Non-Pipelined Speedup” and “Pipelined Speedup”, indicate the overall speedup achieved when considering an RPU without and with pipelining.

Benchmark	Non-Pipelined Speedup	Pipelined Speedup	Speedup Improvement	Non-Pipelined CPL (#clock cycles)	Pipelined CPL (Steady State) (#clock cycles)
compress1	1.65	1.34	0.81	4	5
count	1.72	1.36	0.79	3	4
even_ones	1.68	1.63	0.97	3	3
expand	1.66	1.34	0.81	4	5
fibonacci	2.32	3.46	1.49	3	2
hamming_dist	1.66	1.62	0.97	3	3
popcmpr	1.00	1.03	1.03	4	3
Reverse	1.88	1.49	0.80	3	4
gcd1	0.97	1.06	1.10	8	7
Average	1.62	1.59	0.98	3.89	4

Table 6.15. Comparing a non-pipelined and a pipelined architecture with sequential scheduling.

Benchmark	Non-Pipelined Speedup	Pipelined Speedup	Speedup Improvement	Non-Pipelined CPL (#clock cycles)	Pipelined CPL (Steady State) (#clock cycles)
compress1	1.65	1.54	0.93	4	4
count	1.72	1.64	0.95	3	3
even_ones	1.68	2.07	1.23	3	2
expand	1.66	1.54	0.93	4	4
fibonacci	2.32	6.83	2.95	3	1
hamming_dist	1.66	2.04	1.23	3	2
popcmpr	1.00	1.16	1.16	4	2
reverse	1.88	1.79	0.95	3	3
gcd1	0.97	1.22	1.26	8	6
average	1.62	2.20	1.36	3.89	3

Table 6.16. Comparing a non-pipelined and a pipelined architecture with overlapping scheduling.

Being each benchmark already accelerated by the RPU, the objective of Megablock pipelining is to increase the speedup provided originally by the RPU. The column "Speedup Improvement" represents the ratio between the non-pipelined speedup and the corresponding pipelined speedup. A value of 1 means that there is no difference in speedup between the non-pipelined and the pipelined version; a value greater than one represents an improvement in the speedup; a value lower than one represents a slowdown.

For instance, we estimate an overall speedup of 1.66× for the benchmark *hamming_dist*, before pipelining. With the pipelining overlapping schedule, the performance of the RPU can be improved by 1.23×. This translates into an overall speedup of 2.04×, after pipelining.

In this set of simple benchmarks, when using sequential scheduling, performance degradation happens in most of the cases after applying pipelining. Pipelining with overlapping schedule leads consistently to better performance than the sequential schedule. It is sometimes able to achieve speedups of benchmarks showing slowdowns with the sequential schedule (*evenones* and *hamming_dist*).

However, even with overlapping scheduling, the improvements are not very significant. With the exception of the *fibonacci* benchmark, which achieves a speedup increase from 2.3× to 6.8× after pipelining (an improvement of around 3.0×), the other benchmarks achieve a speedup of at most 2× (with improvements between 1.16× and 1.26×). In four benchmarks there were still slowdowns. We attribute these results to the fact of considering Megablocks without memory accesses. In those Megablocks, it is highly likely that the computations for the update of the inputs represent a significant part of the critical path of the Megablock, increasing the CPL of the Input Module (Section 5.6.2) and, consequently, the number of cycles needed to complete an iteration. In Megablocks with memory accesses, it is more

likely for the update of inputs to be related to the update of the addresses for the memory accesses. This can be confirmed with the CPL columns of Table 6.15 and Table 6.16. To achieve improvements when pipelining, the number of cycles of the steady state of the pipelined version must be lower than the number of cycles for a single original non-pipelined iteration.

Figure 6.17 compares the resource increase between RPUs without and with pipelining using an overlapping schedule.

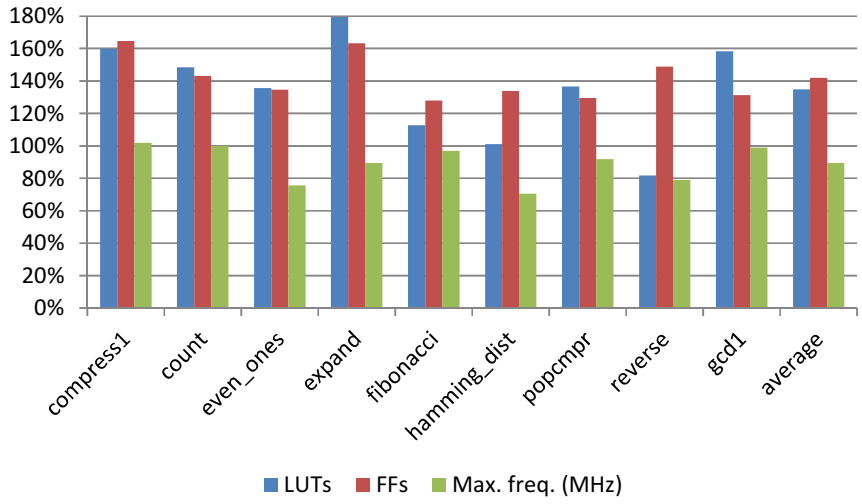


Figure 6.17. FPGA resources increase when using pipelining with overlapping schedule over the non-pipelined implementation.

The values show increases relative to the FPGA resources of the implementation of the non-pipelined modules. In all cases, the pipelined implementation uses more FF (flip-flop) resources (between 1.3× and 1.7× more resources), and generally, the LUTs (look-up table) resources increase too (between 1.01× and 1.8× more resources). This is to be expected, as the pipelined version includes additional modules (e.g., IM), which can have several stages. The maximum clock frequencies in the pipelined modules have moderate decreases for most of the benchmarks (between 0.71× and 1.02×).

Much of the increase in resources can be explained by the characteristics of the benchmarks. Having the CPL of the pipelined module very close to the CPL of the non-pipelined module indicates that the IM replicates most of the critical path of the LM. As these are small benchmarks, the critical path represents a big portion of the Megablock body. We

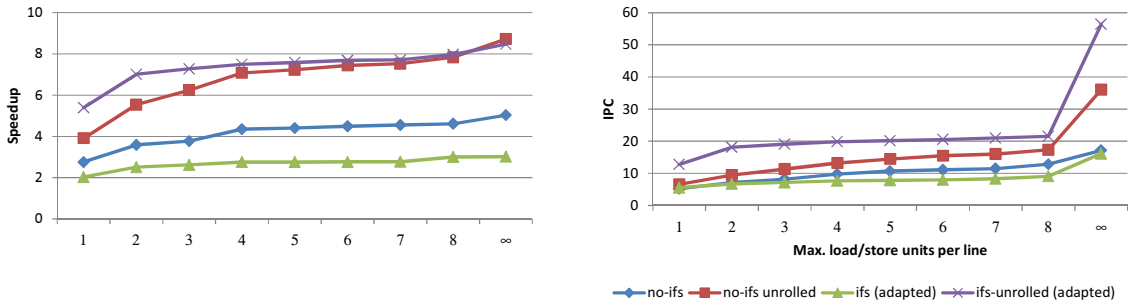
expect that in examples with memory accesses, the IM represents a small portion of the Megablock body.

In one benchmark, *reverse*, the number of LUTs decreases. When the sub-modules of the VHDL description of the pipelined version of this benchmark are implemented separately, the number of LUTs is always the same or greater for the non-pipelined version. We suggest that the number of LUTs decreases in the main module due to global hardware synthesis optimizations. The increased number of FFs enables the synthesis tool to use the FF logic (e.g., set and clear inputs) to implement some of the logic which was previously mapped to LUTs.

When considering the set of 66 benchmarks used in Section 6.3, as the *no-ifs* and the *ifs (adapted)* sets contain benchmarks with memory accesses, to apply the pipelining technique we need to ensure the guarantees presented in Section 5.6.1 to avoid inter-iteration dependencies. After examining the source code of the benchmarks, we discovered 5 benchmarks, out of 66 (7.6%), which did not respect the guarantees. A number of them compute some state during an iteration which is needed in the next iteration, e.g., *rng*, *viterbi*, *md5*. Others, e.g., *bubble_sort*, *fft*, modify parts of the input array and generate loop-carried dependencies which cannot be removed without changing the algorithm. These 5 benchmarks were not considered in the following results.

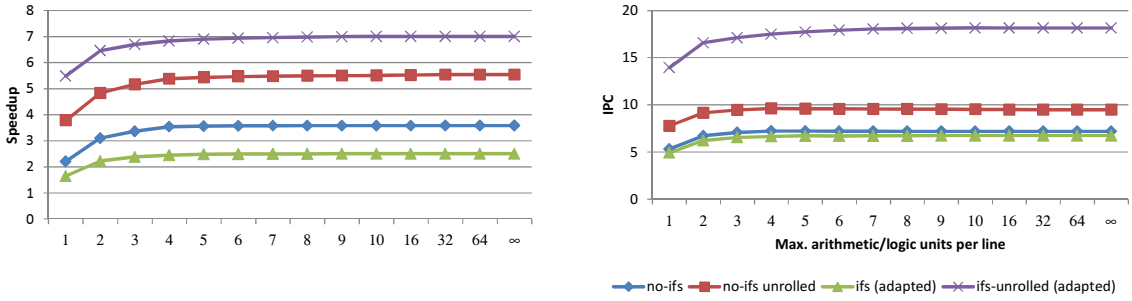
We observe significant increases in both speedup and IPC, when using pipelining with overlapping schedule (see Figure 6.18, Figure 6.19, and Figure 6.20). The *ifs (adapted)* set in particular shows great speedup potential when loop unrolling is enabled, which is similar or greater than the speedup of the *no-ifs* set under the same conditions. Appendix B, Section B-3 and Section B-4, present the results using the geometric mean instead, for sequential scheduling and overlapping scheduling, respectively. When considering the geometric mean, the *ifs (adapted)* set shows lower speedups than the *no-ifs* set for the *unrolled* case. For the *innerloops* case, the gap between the sets is larger.

In Figure 6.18, when going from a maximum of 8 memory operations to an unrestricted number of memory operations there is a significant spike in IPC that is not followed by the speedup. We attribute this behavior to a single IPC value which is distant from the rest of the data, from the benchmark *idct_8x8*. The lines of IPC and speedup have a similar behavior in Figure B.10 in Appendix B-4, which uses the geometric mean (which is less affected by extreme values) instead of the arithmetic mean.



a) **b)**

Figure 6.18. Average a) speedup and b) IPC after pipelining with overlapping schedule, when varying the maximum number of load/store units per row.



a) **b)**

Figure 6.19. Average a) speedup and b) IPC after pipelining with overlapping schedule, when varying the maximum number of arithmetic/logic units per row.

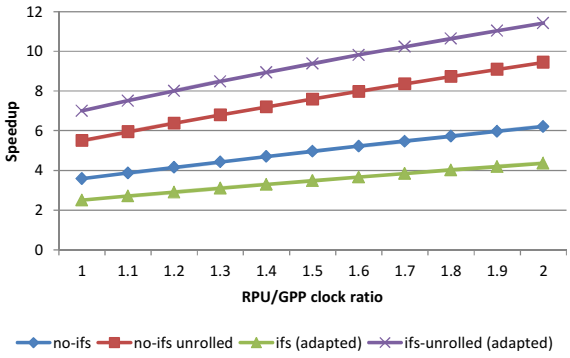


Figure 6.20. Average speedup after pipelining with overlapping schedule, when varying the ratio between RPU and GPP clock frequencies.

Figure 6.21 presents overall application speedups when considering pipelining with overlapping schedule and the 8 FUs-2Mem configuration.

We observe that in several cases, the pipelining contribution amplifies the improvement of the RPU by a factor of 2 or more. For instance, we estimate a speedup of 3× for the benchmark *vecsum*, before pipelining. With an overlapping schedule, the performance of the RPU can be improved by 1.96×. This translates into an overall speedup of 5.8×, after pipelining.

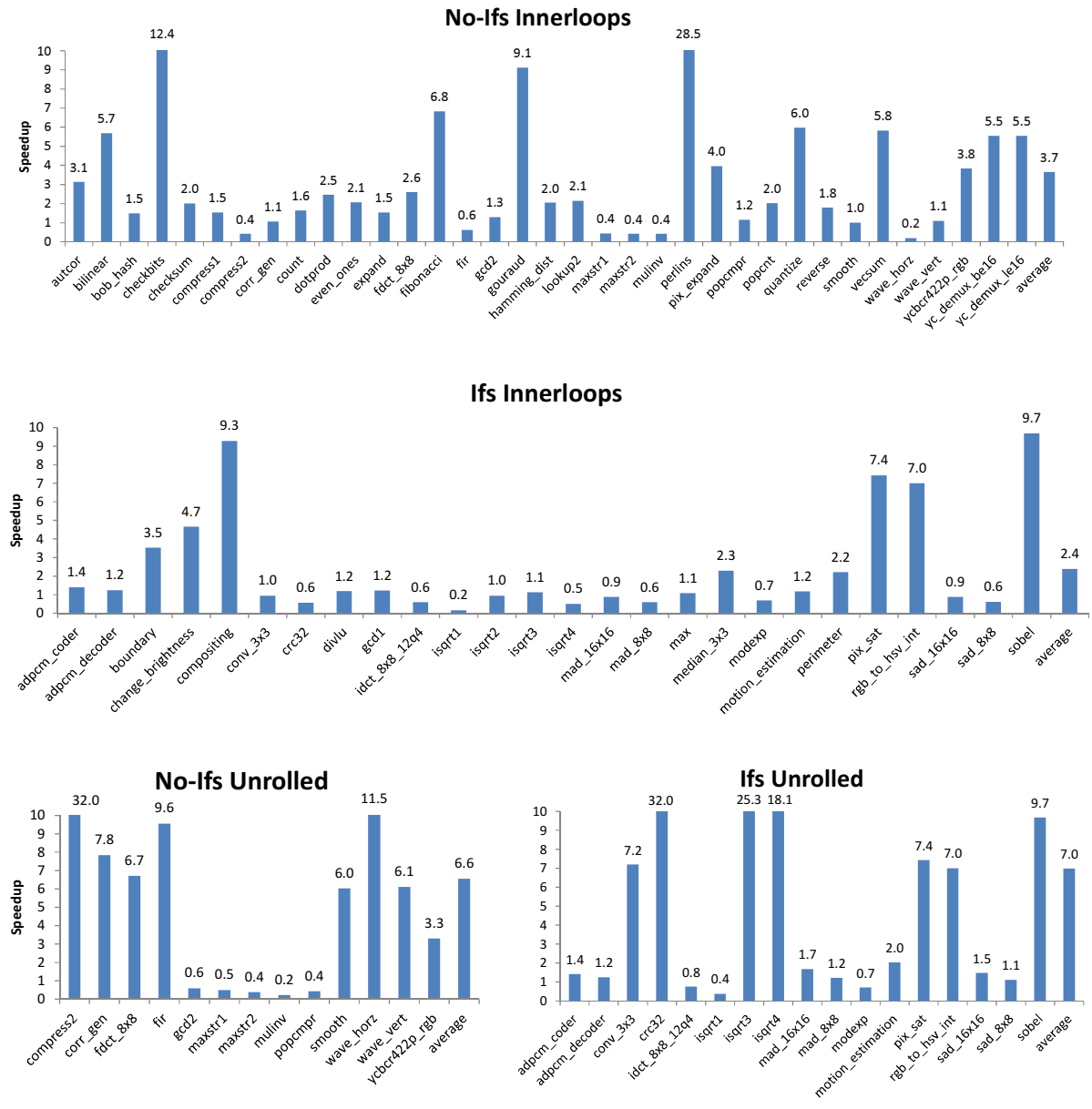


Figure 6.21. Individual overall speedups for a pipelined architecture with overlapping schedule, considering a maximum of 8 parallel arithmetic/logic FUs and 2 load/store operations per clock cycle.

There are noticeable speedups after pipelining, for several benchmarks. For the *innerloops* case we have *change_brightness*, from 1.6× to 9.3×; *checkbits*, from 4.1× to 12.4×;

compositing, from 1.6× to 9.3×; *fibonacci*, from 2.3× to 6.8×; *gouraud*, from 3.1× to 9.1×; *quantize*, from 2.2× to 6×; and *rgb_to_hsv_int*, from 2.3× to 7.0×. For the *unrolled* case we have *compress2*, from 2.5× to 32×; *crc32*, from 1.6× to 32.0×; *isqrt3*, from 2.5× to 25.3×; *isqrt4*, from 2.0× to 18.1×; and *pix_sat*, from 1.1× to 7.4×.

These speedups can be explained by two factors presented in Table 6.17. The first factor is the ratio between the average CPL of the executed Megablocks in the baseline scenario (*CPL (Baseline)* column), and the average number of cycles of the steady state when the Megablocks execute using the overlapping schedule (*Steady State Latency* column). The ratio between these two values (*Ratio* column) is an upper-bound for the possible increase in speedup when applying pipelining. For instance, *crc32* went from a speedup of 1.6× to a speedup of 32× after pipelining, which represents an improvement of 20× for a corresponding ratio of 24.5×. The second factor is the number of average iterations per Megablock call (last column). Note that in all cases, the number of average iterations is high (above 99). When using pipelining, the improvement comes from execution in the steady state. The higher the portion of execution is spent in the steady state (instead of the prologue), the closer the improvement is to the upper bound speedup given by the ratio between the baseline CPL and the steady state latency.

Benchmark	CPL (Baseline) (#clock cycles)	Steady State Latency (#clock cycles)	Ratio	Speedup Improvement	Avg. It. p/ call
change_brightness	12	2	6.0	5.81	99
checkbits	16	4	4.0	3.02	166
compositing	15	2	7.5	5.81	199
fibonacci	3	1	3.0	2.96	2,378
gouraud	6	2	3.0	2.94	1,999
quantize	6	2	3.0	2.69	199
rgb_to_hsv_int	55	16	3.4	3.04	499
compress2	65	4	16.3	12.80	999
crc32	49	2	24.5	20.00	109
isqrt3	112	2	56.0	10.12	99
isqrt4	73	2	36.5	9.05	99
pix_sat	14	2	7.0	6.73	2,000

Table 6.17. CPL comparison between baseline and pipelined with overlapping schedule.

Overall, we could apply the loop pipelining technique to 61 of the 66 benchmarks originally considered in this chapter. Considering this subset of 61 benchmarks and loop pipelining with overlapping schedule, for the *innerloops* case we achieve speedups from 0.2× to 28.5×, with an average speedup of 3.1× (or 1.8×, when using the geometric mean). When activating unrolling of inner loops, we achieve speedups from 0.2× to 32×, with an average speedup of 5.6× (or 3×, when using the geometric mean).

When considering only the benchmarks which provide speedup, for the *innerloops* case we achieve an average speedup of 4.4× (from 1.5× to 28.5×) over a set of 28 benchmarks for the *no-ifs* set, and an average speedup of 3.6× (from 1.1× to 9.7×) over a set of 15 benchmarks for the *ifs (adapted)* set. When considering unrolling of inner loops, in the *no-ifs* set the average speedup increases to 6.2× (from 1.5× to 32×) over a set of 32 benchmarks, and in the *ifs (adapted)* set the average speedup increases to 6.5× (from 1.1× to 32×) over a set of 22 benchmarks.

Combining the techniques previously presented (inner loop unrolling, *if-conversion*, graph transformations, loop pipelining) we were able to achieve results on par to those found in literature. For instance, Warp [13], the work we believe closest to our approach (e.g., uses loops as detection unit), reports an average speedup of 6.3× over a set of 15 benchmarks. Paek et al. [58], which also implements loop pipelining in CGRAs, but on a static context, report an average speedup of 9.4× when using examples of the DSPstone benchmark suite [127].

6.6 Application Examples

6.6.1 3D Path Planning Application

Using the same approach of Section 6.3, we applied the dynamic partitioning technique to an airborne collision avoidance application, known as 3D Path Planning (herein referred as *3dpp*), provided by Honeywell [128]. It consists of 841 lines of C code, distributed over 10 files and 48 functions. A step of the application requires 50,601,067 MicroBlaze clock cycles.

Most of the application time (~80%) is spent in a single function, *gridIterate*, which has 61 lines of C code and a nested loop with 3 levels. The function was modified with the *if-conversion* technique described in Section 4.5. When using this technique, usually the software execution time of the program increases, due to the execution of all paths of the loop in each iteration. In this case, it reduced slightly, representing 99.5% of the previous

execution time. We consider that two factors contribute to this effect: first, the loop contains one frequent path which is computationally intensive, while the other paths are rarely taken and are very lightweight (e.g., attribution of a constant to a variable). This contributes to an execution time of the function that is at least similar to the original. The reduction comes from the loop not having branches, eliminating the branch misprediction penalties which happen in the original function.

Table 6.18 presents the characteristics extracted from the *3dpp* application, when considering the default setup for Megablock detection of a maximum pattern size of 24, and the basic block as detection unit. In the current implementation, 10 Megablocks are responsible for about 64% and 87% of the software execution time, when considering inner loops or loop unrolling, respectively. We obtained a speedup of 1.1× when mapping only innermost loops, and a speedup of 2× when unrolling innermost loops. Most of the speedup improvement of the unrolled case comes from the higher coverage and higher number of iterations per call, and higher number of operations executed per iteration (reducing overhead). The average ILP for both cases is above 4, a positive contributor to the overall speedup.

Unrolled Loops	Speed up	Coverage	Megablocks Det./Exec.	Avg. It. p/ call	Avg. Op. p/ It.	Avg. ILP (Min/Max)	Avg. CPL (Min/Max)
No	1.1	64%	11/10	9.4	36.0	4.1 (2.5/7.0)	9.3 (2/33)
Yes	2.0	87%	22/10	17.1	153.2	4.4 (2.9/8.0)	39.6 (2/165)

Table 6.18. Characteristics for the execution of the application 3dpp.

6.6.2 Dynamic Partitioning on an Embedded Processor – fir

We have developed Java tools (see Appendix C) which implement and simulate some of the phases of dynamic partitioning (e.g., detection, translation). Additionally, we have ported the tools to the Android platform [129] and built a software version of the Megablock Detector in C language. This way, it was possible to measure the execution time of the several steps of these phases, when executing on embedded processors.

In this section we focus on an application and present examples of the several steps for that application. We selected the *fir* benchmark since it was complex enough to be an interesting example, and small enough to illustrate the process.

Figure 6.22 presents the C code for the kernel of the *fir* function. After compiling the code according to the setup described in Section 6.1, we simulated it so we could detect

Megablocks. We used the default setup for Megablock detection (24 as maximum pattern size, and basic block as detection unit) and enabled loop unrolling. Two Megablocks where detected, one representing 0.1% of the software execution time, and the other representing around 98% of the software execution time. The first Megablock was discarded, only the latter was considered in this example.

```
void fir_original(int x[], int c[], int M1, int N1, int *y) {
   int j, i;
   y[0]=c[0]*x[0];
   y[1]= c[0]*x[1]+c[1]*x[0];
   y[2]= c[0]*x[2]+c[1]*x[1]+c[2]*x[0];

   for(j=3; j<M1; j++) {
     int output=0;
     for(i=0; i<N1; i++) {
       output+=c[i]*x[j-i];
     }
     y[j] = output;
   }
}
```

Figure 6.22. C code for a *fir* function.

Figure 6.23 shows the considered Megablock, and contains three columns of information. The first column shows the addresses of the instructions of the Megablock body; the second column shows the assembly instructions executed by the MicroBlaze processor that form the Megablock; and the third column shows the corresponding graph operations when transforming the code to the graph representation. When an assembly instruction is represented by two or more graph operations (e.g., *lw*, *lwi*, *sw*), the additional graph operations appear separated by commas.

The instruction addresses of the first column that are in bold (six in total) represent the addresses needed to detect the Megablock. They correspond to the first address of the six basic blocks that represent the Megablock. As explained in Section 4.3, two repetitions of the same sequence of addresses are enough to detect a Megablock, which means that 12 addresses where needed to detect this Megablock.

```
  Address     Instruction           Graph Op.
0x00000208 bleid r9, 52        → 0:lessOrEqualZero
0x0000020C addk r10, r0, r0    → 1:add
0x00000210 addk r8, r6, r0     → 2:add
0x00000214 addk r7, r10, r0    → 3:add
0x00000218 bslli r3, r7, 1026  → 4:sll
0x0000021C lwi r5, r8, 0       → 5:add, 6:load
0x00000220 lw r4, r11, r3      → 7:add, 8:load
0x00000224 addik r7, r7, 1     → 9:add
0x00000228 addik r8, r8, -4    → 10:add
0x0000022C mul r4, r4, r5      → 11:mul
0x00000230 rsubk r18, r7, r9   → 12:rsub_carry
0x00000234 bneid r18, -28      → 13:equalZero
0x00000238 addk r10, r10, r4   → 14:add
0x00000218 bslli r3, r7, 1026  → 15:sll
0x0000021C lwi r5, r8, 0       → 16:add, 17:load
0x00000220 lw r4, r11, r3      → 18:add, 19:load
0x00000224 addik r7, r7, 1     → 20:add
0x00000228 addik r8, r8, -4    → 21:add
0x0000022C mul r4, r4, r5      → 22:mul
0x00000230 rsubk r18, r7, r9   → 23:rsub_carry
0x00000234 bneid r18, -28      → 24:equalZero
0x00000238 addk r10, r10, r4   → 25:add
0x00000218 bslli r3, r7, 1026  → 26:sll
0x0000021C lwi r5, r8, 0       → 27:add, 28:load
0x00000220 lw r4, r11, r3      → 29:add, 30:load
0x00000224 addik r7, r7, 1     → 31:add
0x00000228 addik r8, r8, -4    → 32:add
0x0000022C mul r4, r4, r5      → 33:mul
0x00000230 rsubk r18, r7, r9   → 34:rsub_carry
0x00000234 bneid r18, -28      → 35:equalZero
0x00000238 addk r10, r10, r4   → 36:add
0x00000218 bslli r3, r7, 1026  → 37:sll
0x0000021C lwi r5, r8, 0       → 38:add, 39:load
0x00000220 lw r4, r11, r3      → 40:add, 41:load
0x00000224 addik r7, r7, 1     → 42:add
0x00000228 addik r8, r8, -4    → 43:add
0x0000022C mul r4, r4, r5      → 44:mul
0x00000230 rsubk r18, r7, r9   → 45:rsub_carry
0x00000234 bneid r18, -28      → 46:notEqualZero
0x00000238 addk r10, r10, r4   → 47:add
0x0000023C bslli r3, r12, 1026 → 48:sll
0x00000240 addik r12, r12, 1   → 49:add
0x00000244 sw r10, r19, r3     → 50:add, 51:store
0x00000248 rsubk r18, r12, r22 → 52:rsub_carry
0x0000024C bneid r18, -68      → 53:equalZero
0x00000250 addik r6, r6, 4     → 54:add
```

Figure 6.23. Assembly code and corresponding graph operations for the *fir* Megablock.

Table 6.19 contains execution times, in milliseconds, for several implementations of the pattern detector used to detect Megablocks, executing on different targets. The execution times represent the time each implementation needed to process the given number of addresses (column *#Addresses*). The given addresses are repetitions of the 6 address sequence of the *fir* Megablock. The values in the column *Hardware Module at 50MHz* correspond to an implementation of the architecture described in Section 5.2, clocked at 50 MHz. It can process one address every clock cycle. The column *MicroBlaze at 50MHz* represents a C implementation of the algorithm in Figure 4.4, running directly on a MicroBlaze processor clocked at 50 MHz. Column *Cortex-A8 at 1GHz* corresponds to an implementation of the same algorithm in Java, running on a Cortex-A8 clocked at 1GHz, over the Android 2.2 platform.

#Addresses	Time using Hardware Module at 50MHz (ms)	Time using a MicroBlaze at 50MHz – C (ms)	Time using a Cortex-A8 at 1GHz – Java (ms)	Speedup (HW vs. MicroBlaze / HW vs. A8)
12	0.0002	2.7	0.6	11,251/2,500
24	0.0005	5.7	1.3	11,963/2,708
48	0.0010	14.0	2.8	14,594/2,917
96	0.0019	30.8	5.9	16,036/3,073
192	0.0038	64.3	12.5	16,757/3,255
384	0.0077	131.5	24.8	17,118/3,229
768	0.0154	265.7	78.7	17,298/5,124

Table 6.19. Execution times for several implementations of the pattern detector for Megablocks.

Generally, the execution times grow linearly with the input (doubling the size of the input doubles the execution time). There is an exception in the Cortex case, where going from 384 addresses to 768 addresses tripled the execution time, instead of doubling. We think this is due to calls from the system to the garbage collector, during execution of the detector.

When comparing execution speeds, the hardware module is much faster than the software implementations: around 3,000× faster than the Cortex case and around 16,000× faster than the MicroBlaze case. This difference can be explained by the highly parallel design of the hardware module, and by the software version not being fully optimized for the target platforms. For the tested cases, excluding the last row, the execution time of the Cortex processor is around 5× faster than the execution time of the MicroBlaze processor.

Table 6.20 shows average execution times, in milliseconds, when running a Java implementation of the *Translation* steps described in Section 5.3, on a Cortex-A8 clocked at 1 GHz over an Android 2.2 platform. The *Translation* phase took, on average, about 79 ms to

transform the assembly code of the Megablock in Figure 6.23 into a mapping configuration for architectures of the kind described in Sections 5.5.1 and 5.5.4. The most expensive operation is the conversion from assembly code to the graph intermediate representation, representing 58% of the execution time. Next we have Placement and the Transform, each one taking 20% and 12% of the time, respectively. The most light-weight steps are the Routing and the Normalization, each one with 6% and 4% of the total execution time.

Using the values of Table 6.19 to extrapolate an execution time for the case where we use an implementation in C, executing in a MicroBlaze at 50 MHz, we obtain a total time for the *Translation* phase of about 400 ms.

Normalize	Graph Converter	Transform	Mapping		Total
			Placement	Routing	
3.03	46.00	9.71	15.45	4.89	79.09

Table 6.20. Average execution times in milliseconds of the *Translation* steps.

6.7 Summary

In this chapter we analyzed and evaluated the use of the Megablock as a loop for Dynamic Hardware/Software Partitioning. We used an extensive set of benchmarks from embedded system domains, and compared our loop detection method with the method used by Warp [125]. We concluded that the Megablock achieves coverage values close to the Warp method, while providing loops with straight-forward and clearly defined control-flow, which can be easily converted to data-flow representations.

We estimated the overall application speedup achievable with the Megablock, considering 66 benchmarks and several scenarios. Considering default mapping parameters, in the baseline scenario we estimate a speedup of 1.7× and 2.2× for the *innerloops* and the *unrolled* cases, respectively. Applying graph transformations and *if-conversion* increases the overall speedup to 1.8× and 2.4× for the *innerloops* and the *unrolled* cases, respectively. Graph transformations did not change the performance significantly, but helped in reducing the number of operations of the Megablock, which can reduce the mapping effort and configuration sizes.

Applying the Megablock pipelining technique can significantly improve the overall application speedup. Considering this subset of 61 benchmarks and loop pipelining with overlapping schedule, we estimate an average speedup of 3.1× and 5.6× for the *innerloops*

and the *unrolled* cases, respectively. From the proposed optimization techniques, Megablock pipelining was the one with the highest impact on performance.

We observed that, for the architecture parameters, the access to memory was a more limiting factor for the speedup than the number of available arithmetical/logical FUs. The speedup values stabilized very quickly for low number of arithmetical/logical FUs (e.g., between 4 and 8 FUs), while there was still noticeable increases in speedup when considering the scenario with unbounded memory accesses. However, we consider that the differences were not high enough as to justify the increased complexity of using more than 2 concurrent memory accesses per cycle. The biggest improvement in speedup, when considering the number of concurrent memory accesses, was consistently when going from 1 memory access to 2 concurrent memory accesses.

7 Conclusions

The main objective of this thesis was to research Dynamic Hardware/Software Partitioning (DHSP) techniques, as a way to take advantage of a reconfigurable processing unit (RPU) acting as a coprocessor in a general purpose processor (GPP) based embedded computing system. The research efforts and experiments were focused on the development of algorithms and techniques in the context of dynamic partitioning, and on the speedups resultant with the migration of computations from the GPP to the RPU.

We proposed novel techniques for dynamically partitioning applications at the binary level, as well as addressing the automatic migration of computations during runtime from a GPP to the RPU. As to maximize the impact of dynamic partitioning one must consider large portions of program execution, an important aspect of this work was to propose a novel kind of loop structure, attractive for architectures with native support to high-degrees of parallelism. This led us to the Megablock, a loop formed by repetitive sequences of instructions present during program execution. We found that the Megablock can represent significant portions of the program execution in most benchmarks, justifying its use as a detection unit for dynamic partitioning. Furthermore, being the Megablock a loop, it is inherently akin to hardware reuse and loop pipelining.

The presented work proposes techniques for the detection, identification, implementation, and transformation of Megablocks, as well as a study of the impact of using the Megablock as a detection unit over an extensive set of benchmarks consisting of 66 functions/kernels. Using an automated approach, we were able to evaluate and explore the techniques proposed in this thesis over this set of benchmarks in a variety of situations.

One of the objectives of this thesis was to test a general approach for dynamic partitioning. To evaluate the impact on automatically moving streams of instructions executed by a GPP to an RPU, we use a set of benchmarks which covers many situations and code characteristics, over several execution scenarios. Rather than being tied to a specific RPU, this thesis explored a number of architecture models, from specific implementations suitable for the today's FPGA technology, to models possible only on future reconfigurable fabrics. We believe that the work presented tackle these issues by the following reasons:

1) As a general principle, we avoided limiting the scope of the proposed techniques when possible. For instance, the presented methodology can be applied to either online or offline scenarios.
2) The Megablocks are detected through a pattern-matching technique which is fully agnostic to the instruction format of the GPP and can be applied to traces of instructions of other processors.
3) Before optimizations and mapping, Megablocks are first converted to an RPU independent Intermediate Representation (IR). To evaluate our techniques in a different GPP, we only need a translator from the specific GPP instructions to the IR.
4) Although we provide a concrete example for the *if-conversion* technique (the language-processor pair C-MicroBlaze), we propose general transformation rules which can be applied to other language-processor combinations.
5) Finally, in this thesis we consider two distinct Megablock implementations: a first one using custom designs obtained by a VHDL representation of a Megablock with the option to support pipelining (see Chapter 6, Section 6.4); and a second one considering a CGRA coprocessor, suitable for executing different Megablocks (see Appendix A).

In order to improve performance, we present a technique for pipelining Megablocks. The technique simplifies the creation of a pipelined version of a loop by taking advantage of the characteristics of the Megablock (e.g., the loop contains only one path). It also presents new ideas, such as avoiding the implementation of an epilog by using atomic loop iterations, or delay the stores to the end of the iteration to avoid output dependencies and simplify the implementation of atomic iterations.

The analysis of the related work shows that dynamic partitioning can be useful in embedded systems. Although it is unlikely that an approach for automatic optimization of general computations will have better results than a handcrafted solution, the improvements achieved by dynamic partitioning can be enough to allow applications to generally take advantage of the existence of reconfigurable hardware in embedded computing systems, as mapping critical sections by hand for each application and device is too costly to be widely used.

Warp [13], the work we believe closest to our approach (e.g., uses loops as detection unit), reports an average speedup of 6.3× over a set of 15 benchmarks. CCA [91] and DIM [14] report a speedup of 2.3× and 2.5×, respectively. A comparison with similar benchmarks

indicate that we were able to achieve results on par with those found in literature, thanks to techniques such as inner loop unrolling and loop pipelining.

Our evaluations consider a RPU coupled to a soft-core microprocessor and the techniques proposed in this thesis (e.g., graph transformations, loop pipelining). When using the complete set of 66 benchmarks and the baseline case with the default RPU architecture (see Chapter 6, Section 6.3.1), for the *innerloops* case we achieve speedups from 0.5× to 4.8×, with an average speedup of 1.7× (or 1.4×, when using the geometric mean). When activating unrolling of inner loops, we achieve speedups from 0.4× to 6.4×, with an average speedup of 2.2× (or 1.6×, when using the geometric mean). After applying *if-conversion* and graph transformation techniques, the average speedups increase slightly to 1.8× and 2.4× when using the arithmetic mean, and 1.6× and 2.1× when using the geometric mean, for the *innerloops* and *unrolled* cases respectively.

We could apply the loop pipelining technique to 61 of the 66 benchmarks in the set. Considering this subset of 61 benchmarks and loop pipelining with overlapping schedule, for the *innerloops* case we achieve speedups from 0.2× to 28.5×, with an average speedup of 3.1× (or 1.8×, when using the geometric mean). When activating unrolling of inner loops, we achieve speedups from 0.2× to 32×, with an average speedup of 5.6× (or 3×, when using the geometric mean).

Furthermore, we have implemented a prototype system for dynamic partitioning, based on an FPGA board (see Appendix A). The prototype is fully functional and runtime reconfigurable, and is capable of transparently moving computations from a GPP to a RPU without changing the executable binary. Although the prototype results were contaminated by high memory access latencies, we estimate reasonable speedups when the CPU directly accesses data stored in local memories, and with the current type of coupling, the system is easily adaptable to other CPUs. These results clearly show a strong evidence of the importance of the techniques proposed in this thesis.

7.1 Future Work

In this thesis we deeply explored the Megablock. However, there is room to justify further research, either with the proposed version of the Megablock, or with a new, extended version. For instance, the address conflicts which can appear when using the Single Address Identification (SAI) method (Chapter 5, Section 5.4) usually indicate different paths of the

same loop. This can be an opportunity for a new loop structure, which supports several paths found during runtime; or we can do Megablock merging and automatically create a single Megablock out of Megablocks with the same start address, by using *if-conversion* techniques.

The detection method, together with the representation, can be extended to detect which sections of a Megablock correspond to an inner loop. With this information, it can be possible to reroll inner loops which are too big to fit the target hardware; more importantly, it would be a step forward for supporting inner loops with a variable number of iterations. Currently, if loop unrolling is enabled, each different iteration count of unrolled inner loops is detected as a distinct Megablock.

The transformations that can be used over the Intermediate Representation can be further explored. We can use transformations that, in addition to the reduction of the number of operations or the critical path length, also focus other aspects, such as increasing ILP [26]. Transformations can be also used to tailor the Megablock to specific units in the target architecture; to evaluate whether some inputs of a Megablock are constant through the entire loop, and specialize the Megablock according to those constants. Merging Megablocks can enable further transformations. For instance, the *gcc* compiler for the Xilinx MicroBlaze soft-core processor includes branch instructions in the code to convert 32-bit to 64-bit integer values. This could be detected and simplified, removing one of the paths of a merged Megablock.

As expected, memory accesses were a bottleneck in many programs and future work should consider memory analysis techniques, such as alias analysis. Alias analysis can determine if two memory operations refer to the same location (address). This enables transformations such as elimination of *redundant loads* [130] or *scalar replacement*, which are very effective in the presence of code inserted by the compiler for *register spilling* (i.e., when due to register pressure, temporary variables are stored in memory). Alternatively, we may be able to analyze and deal less conservatively with data-dependences in order to increase the parallelism degree. Alias analysis is a well know optimization, but its application is limited when used in static compilers [131] due to memory address ambiguity. Since some of the memory addresses that are ambiguous at compile time can be resolved at runtime, applying this optimization dynamically can open new opportunities.

The use of source-to-source transformations to implement *if-conversion* presents a way to do hardware/software co-design where instead of using custom software compilation tools, new programming languages or language extensions, it is possible to write plain code in the

target language in a suitable way for the hardware, without having to spend time in low-level system design details such as the communication between the processor and the coprocessor. Programs can be rewritten in the same target source code to better fit Megablock detection and/or the mapping to the coprocessor. One possible research avenue is the use of multiple binaries for the same function and to opt dynamically to the more suitable binary.

Runtime identification of certain computation patterns may allow further improvements and/or the resolution to apply a certain optimization technique. One example would be the identification that the computations being executed are related to loads from a memory region followed by computations and then storing to a distinct, not overlapping, memory region. This identification may allow more aggressive loop pipelining techniques.

The mapping process can benefit from the use of additional information provided by the compiler that generated the binaries, or by the analysis of the binaries prior to their execution. This additional information would be beneficial for most Megablock optimizations. As an example, the identification of the array variable associated with each load/store in the execution trace would help loop pipelining and the use of memory banks.

A more advanced approach would use data speculation to obtain optimized Megablocks. During runtime, the system may track data ranges and specific values for the registers and based on the probabilities may optimize Megablocks considering a certain value in a register. This may be a viable option for Megablocks without side-effects as in this case roll-back is simplified.

8 References

[1] E. Monmasson, L. Idkhajine, M. N. Cirstea, I. Bahri, A. Tisan, and M. W. Naouar, "FPGAs in Industrial Control Applications," in *IEEE Transactions on Industrial Informatics*, vol. 7, pp. 224-243, May 2011.

[2] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," in *Computer*, vol. 41, pp. 33-38, 2008.

[3] Y. Patt, "Future Microprocessors: Multi-core, Mega-nonsense, and What We Must Do Differently Moving Forward," in *Parallel@Illinois Distinguished Lecture Series*, 2010.

[4] Z. Guo, W. Najjar, F. Vahid, and K. Vissers, "A quantitative analysis of the speedup factors of FPGAs over processors," in *FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, Monterey, California, USA, 2004, pp. 162-170.

[5] J. Henkel, "A low power hardware/software partitioning approach for core-based embedded systems," in *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, 1999, pp. 122-127.

[6] S. Hauck and A. DeHon, *Reconfigurable computing: the theory and practice of FPGA-based computation*: Morgan Kaufmann Pub, 2008.

[7] L. Józwiak, N. Nedjah, and M. Figueroa, "Modern development methods and tools for embedded reconfigurable systems: A survey," in *Integration, the VLSI Journal*, vol. 43, pp. 1-33, January 2010.

[8] T. Wiangtong, P. Y. K. Cheung, and W. Luk, "Hardware/Software Codesign: a Systematic Approach Targeting Data-intensive Applications," in *IEEE Signal Processing Magazine*, vol. 22, pp. 14-22, 2005.

[9] G. Stitt, F. Vahid, and S. Nematbakhsh, "Energy savings and speedups from partitioning critical software loops to hardware in embedded systems," in *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 3, pp. 218-232, 2004.

[10] M. Graphics, "Catapult C synthesis," in *http://www.mentor.com*, 2008.

[11] Y. Ben-Asher, N. Rotem, and E. Shochat, "Finding the best compromise in compiling compound loops to Verilog," in *Journal of Systems Architecture*, vol. 56, pp. 474-486, 2010.

[12] S. Ben Othman, A. Ben Salem, and S. Ben Saoud, "Hw acceleration for FPGA-based drive controllers," in *Industrial Electronics (ISIE), 2010 IEEE International Symposium on*, pp. 196-201, 2010.

[13] R. Lysecky, G. Stitt, and F. Vahid, "Warp Processors," in *ACM Trans. Des. Autom. Electron. Syst.*, vol. 11, pp. 659-681, 2006.

[14] A. C. S. Beck, M. B. Rutzig, G. Gaydadjiev, and L. Carro, "Transparent reconfigurable acceleration for heterogeneous embedded applications," in *Proc. Conf. Design, Automation and Test in Europe (DATE'08)*, Munich, Germany, 2008, pp. 1208-1213.

[15] T. Lindholm and F. Yellin, *The Java virtual machine specification*: Prentice Hall PTR, 1999.

[16] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox, "Design of the Java HotSpot™ client compiler for Java 6," in *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 5, p. 7, 2008.

[17] J. M. Bull, L. A. Smith, L. Pottage, and R. Freeman, "Benchmarking Java against C and Fortran for scientific applications," in *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, 2001, pp. 97-105.

[18] R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, and S. G. Robinson, "Binary translation," in *Communications of the ACM*, vol. 36, pp. 69-81, February 1993.

[19] B. Case, "'Intel Reveals Pentium Implementation Details," in *Microprocessor Report*, vol. 5, pp. 9–17, 1993.

[20] Apple, "Universal Binary Programming Guidelines, Second Edition," 2009.

[21] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson, "The Transmeta Code Morphing™ Software: using speculation, recovery, and adaptive retranslation to address real-life challenges," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, ed: IEEE Computer Society Washington, DC, USA, 2003, pp. 15-24.

[22] J. M. P. Cardoso, J. Bispo, and A. K. Sanches, "The Role of Programming Models on Reconfigurable Computing Fabrics," in *Chapter XII in the book: Behavioral Modeling for Embedded Systems and Technologies: Applications for Design and Implementation*, L. Gomes and J. M. Fernandes, Eds., ed: IGI Global, 2009.

[23] J. Bispo and J. M. P. Cardoso, "On Identifying Segments of Traces for Dynamic Compilation," in *Proc. Intl. Conf. on Field Programmable Logic and Appl. (FPL'10)*, Milano, Italy, 2010, pp. 263-266.

[24] J. Bispo and J. M. P. Cardoso, "Using the Megablock to Partition Programs for Embedded Systems at Runtime," in *INForum 2010 - II Simpósio de Informática*, Univ. do Minho, Braga, Portugal, 2010, pp. 699-710.

[25] J. Bispo and J. M. P. Cardoso, "On Identifying and Optimizing Instruction Sequences for Dynamic Compilation," in *Proc. Intl. Conf. on Field-Programmable Tech.*, Beijing, China, 2010, pp. 437-440.

[26] J. Bispo and J. M. P. Cardoso, "Techniques for Dynamically Mapping Computations to Coprocessors," in *Intl. Conf. on ReConFigurable Comp. and FPGAs (ReConFig'2011)*, Cancun, Mexico, 2011, pp. 505-508.

[27] J. Bispo, N. Paulino, J. M. P. Cardoso, and J. C. Ferreira, "From Instruction Traces to Specialized Reconfigurable Arrays," in *Intl. Conf. on ReConFigurable Comp. and FPGAs (ReConFig'2011)*, Cancun, Mexico, 2011, pp. 386-391.

[28] J. Bispo, N. Paulino, J. C. Ferreira, and J. M. P. Cardoso, "Transparent Trace-Based Binary Acceleration for Reconfigurable HW/SW Systems," in *IEEE Transactions on Industrial Informatics*, 2012 (under review).

[29] J. Hennessy, D. Patterson, D. Goldberg, and K. Asanovic, *Computer architecture: a quantitative approach*: Morgan Kaufmann, 2003.

[30] F. E. Allen, "Control flow analysis," in *SIGPLAN Not.*, vol. 5, pp. 1-19, 1970.

[31] J. Von Neumann, "First Draft of a Report on the EDVAC," in *Annals of the History of Computing, IEEE*, vol. 15, pp. 27-75, 1993.

[32] M. Gokhale and P. S. Graham, *Reconfigurable computing: Accelerating computation with field-programmable gate arrays*: Springer Verlag, 2005.

[33] J. Becker, R. Hartenstein, M. Herz, and U. Nageldinger, "Parallelization in co-compilation for configurable accelerators-a host/accelerator partitioning compilation method," in *Proceedings of the Asia and South Pacific-Design Automation Conference (ASP-DAC'98)*, 1998, pp. 23-33.

[34] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26(2), pp. 203-215, 2007.

[35] R. Tessier and W. Burleson, "Reconfigurable computing for digital signal processing: A survey," in *The Journal of VLSI Signal Processing*, vol. 28, pp. 7-27, 2001.

[36] J. M. P. Cardoso and P. C. Diniz, *Compilation Techniques for Reconfigurable Architectures*: Springer Verlag, 2008.

[37] R. Hartenstein, "A decade of reconfigurable computing: a visionary retrospective," in *Proceedings of the conference on Design, automation and test in Europe*, ed: IEEE Press Piscataway, NJ, USA, 2001, pp. 642-649.

[38] J. M. P. Cardoso and M. P. Vestístias, "Architectures and compilers to support reconfigurable computing," in *Crossroads*, vol. 5, pp. 15-22, 1999.

[39] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," in *Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*, ed: ACM New York, NY, USA, 2006, pp. 21-30.

[40] D. L. Perry, *VHDL*: McGraw-Hill, 1993.

[41] D. Thomas and P. Moorby, *The Verilog hardware description language*: Springer Verlag, 2008.

[42] I. Xilinx, "Xilinx ISE Design Suite 12.2," ed, 1995-2010.

[43] V. Betz, J. Rose, and A. Marquardt, *Architecture and CAD for Deep-Submicron FPGAs*. Hingham, Mass: Kluwer Academic, 1999.

[44] G. De Micheli, *Synthesis and optimization of digital circuits*: McGraw-Hill Higher Education, 1994.

[45] Enclustra. (last update in 2008). *Wiki FPGA - Virtex-5 LX*. Available: http://www.wikifpga.com/index.php?title=Virtex-5_LX

[46] C. Brunelli, F. Garzia, D. Rossi, and J. Nurmi, "A coarse-grain reconfigurable architecture for multimedia applications supporting subword and floating-point calculations," in *Journal of Systems Architecture*, vol. 56, pp. 38-47.

[47] S. Shukla, N. W. Bergmann, and J. Becker, "QUKU: A Two-Level Reconfigurable Architecture," in *Proceedings of the IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures*, 2006, pp. 109-116.

[48] J. Teifel and R. Manohar, "Highly pipelined asynchronous FPGAs," in *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, ed: ACM New York, NY, USA, 2004, pp. 133-142.

[49] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools*: Pearson, 2006.

[50] S. S. Muchnick, *Advanced compiler design and implementation*: Morgan Kaufmann, 1997.

[51] T. Software. *TIOBE Programming Community Index*. Available: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[52] G. De Micheli, R. Ernst, and W. H. Wolf, *Readings in hardware/software co-design*: Morgan Kaufmann, 2002.

[53] N. Clark, J. Blome, M. Chu, S. Mahlke, S. Biles, and K. Flautner, "An Architecture Framework for Transparent Instruction Set Customization in Embedded Processors," in *Proc. 32nd Ann. Intl. Symp. Computer Architecture (ISCA'05)*, 2005, pp. 272-283.

[54] T. Kistler, "Dynamic Runtime Optimization," in *Proceedings of the Joint Modular Languages Conference on Modular Programming Languages*, ed: Springer-Verlag London, UK, 1997, pp. 53-66.

[55] L. Baraz, T. Devor, O. Etzion, S. Goldenberg, A. Skaletsky, Y. Wang, and Y. Zemach, "IA-32 execution layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium/spl reg/-based systems," in *36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-36)*, 2003, pp. 191-201.

[56] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. B. Yadavalli, and J. Yates, "FX! 32: A profile-directed binary translator," in *IEEE Micro*, vol. 18, pp. 56-64, 1998.

[57] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, 2005, pp. 41-46.

[58] J. K. Paek, K. Choi, and J. Lee, "Binary acceleration using coarse-grained reconfigurable architecture," in *ACM SIGARCH Computer Architecture News*, vol. 38, pp. 33-39, 2011.

[59] E. Mirsky and A. DeHon, "MATRIX: a reconfigurable computing architecture with configurable instruction distribution and deployable resources," in *IEEE Symposium on FPGAs for Custom Computing Machines* 1996, pp. 157-166.

[60] E. Waingold, M. Taylor, V. Sarkar, W. Lee, Victor Lee, J. Kim, M. Frank, P. Finch, S. Devabhaktuni, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal, "Baring it all to software: Raw machines," in *Computer*, vol. 30, pp. 86-93, 1997.

[61] 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 *Field-Programmable Logic and Applications*, ed, 2003, pp. 61-70.

[62] T. Miyamori and K. Olukotun, "A Quantitative Analysis of Reconfigurable Coprocessors for Multimedia Applications," in *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 2-11, 1998.

[63] H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. C. Filho, "MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications," in *IEEE Transactions on Computers*, vol. 49, pp. 465-481, 2000.

[64] J. R. Hauser and J. Wawrzynek, "Garp: a MIPS processor with a reconfigurable coprocessor," in *The 5th Annual IEEE Symposium on FPGAs for Custom Computing Machines*, 1997, pp. 12-21.

[65] Z. A. Ye, A. Moshovos, S. Hauck', and P. Banerjee, "CHIMAERA: a high-performance architecture with a tightly-coupled reconfigurable functional unit," in *Proceedings of the 27th International Symposium on Computer Architecture*, 2000, pp. 225-235.

[66] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. R. Taylor, "PipeRench: a reconfigurable architecture and compiler," in *Computer*, vol. 33, pp. 70-77, 2000.

[67] V. Baumgarte, G. Ehlers, F. May, A. Nückel, M. Vorbach, and M. Weinhardt, "PACT XPP—A Self-Reconfigurable Data Processing Architecture," in *J. Supercomput.*, vol. 26, pp. 167-184, 2003.

[68] C. Ebeling, D. C. Cronquist, P. Franklin, J. Secosky, and S. G. Berg, "Mapping applications to the RaPiD configurable architecture," in *The 5th Annual IEEE Symposium on FPGAs for Custom Computing Machines*, 1997, pp. 106-115.

[69] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, Palo Alto, California, 2004, pp. 75-88.

[70] K. Bondalapati, P. Diniz, P. Duncan, J. Granacki, M. Hall, R. Jain, and H. Ziegler, "DEFACTO: A design environment for adaptive computing technology," in *Parallel and Distributed Processing*, pp. 570-578, 1999.

[71] Y. Yankova, G. Kuzmanov, K. Bertels, G. Gaydadjiev, Y. Lu, and S. Vassiliadis, "DWARV: DelftWorkbench Automated Reconfigurable VHDL Generator," in *17th International Conference on Field Programmable Logic and Applications (FPL)*, 2007, pp. 697-701.

[72] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: a transparent dynamic optimization system," in *Proceedings of the ACM SIGPLAN 2000 conference on Programming Language Design and Implementation*, Vancouver, British Columbia, Canada, 2000, pp. 1-12.

[73] W. M. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, and G. E. Haab, "The superblock: an effective technique for VLIW and superscalar compilation," in *The Journal of Supercomputing*, vol. 7, pp. 229-248, 1993.

[74] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, and J. Orendorff, "Trace-based just-in-time type specialization for dynamic languages," 2009, pp. 465-478.

[75] N. Clark, H. Zhong, and S. Mahlke, "Processor acceleration through automated instruction set customization," in *36th International Symposium on Microarchitecture*, 2003, pp. 129-140.

[76] R. Lysecky and F. Vahid, "Design and implementation of a MicroBlaze-based warp processor," in *ACM Transactions on Embedded Computing Systems*, vol. 8, pp. 1-22, 2009.

[77] S. Singh, J. Rose, P. Chow, and D. Lewis, "The effect of logic block architecture on FPGA performance," in *IEEE Journal of Solid-State Circuits*, vol. 27, pp. 281-287, March 1992.

[78] P. Chow, S. O. Seo, J. Rose, K. Chung, G. Páez-Monzón, and I. Rahardja, "The design of an SRAM-based field-programmable gate array, part I: Architecture," in *IEEE*

Transactions on Very Large Scale Integration (VLSI) Systems (TVLSI), vol. 7, pp. 191-197, 1999.

[79] A. Marquardt, V. Betz, and J. Rose, "Speed and area tradeoffs in cluster-based FPGA architectures," in *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 8, pp. 84-93, 2000.

[80] G. Stitt, R. Lysecky, and F. Vahid, "Dynamic hardware/software partitioning: a first approach," in *Proceedings of the 40th conference on Design automation*, ed: ACM New York, NY, USA, 2003, pp. 250-255.

[81] R. Lysecky and F. Vahid, "On-Chip logic minimization," in *Proceedings of the Design Automation Conference (DAC)*, 2003, pp. 334-337.

[82] R. Lysecky, F. Vahid, and S. Tan, "Dynamic FPGA routing for just-in-time FPGA compilation," in *Proceedings of the Design Automation Conference (DAC)*, 2004, pp. 954-959.

[83] R. Lysecky, F. Vahid, and S. Tan, "A study of the scalability of on-chip routing for just-in-time FPGA compilation," in *Proceeding of the Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2005, pp. 57-62.

[84] V. Betz and J. Rose, "VPR: A new packaging, placement, and routing for FPGA research," in *Proceeding of the Internation Workshop on Field Programmable Logic and Applications (FPLA)*, 1997, pp. 213-222.

[85] D. Brelaz, "New methods to color the vertices of a graph," in *Communications of ACM*, vol. 22, pp. 251-256, 1979.

[86] G. Memik, W. H. Mangione-Smith, and W. Hu, "Netbench: A benchmarking suite for network processors," in *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, ed: IEEE Press Piscataway, NJ, USA, 2001, pp. 39-42.

[87] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: a tool for evaluating and synthesizing multimedia and communicatons systems," in *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, ed: IEEE Computer Society, 1997, pp. 330-335.

[88] EEMBC. (2005). *The Embedded Microprocessor Benchmark Consortium*. Available: http://www.eembc.org

[89] A. Malik, B. Moyer, and D. Cermak, "A low power unified cache architecture providing power and performance flexibility (poster session)," in *Proceedings of the 2000 international symposium on Low power electronics and design*, ed: ACM New York, NY, USA, 2000, pp. 241-243.

[90] I. Xilinx, "Microblaze Processor Reference Guide v13.4," in *reference manual*, 2011.

[91] N. Clark, M. Kudlur, H. Park, S. Mahlke, and K. Flautner, "Application-Specific Processing on a General-Purpose Core via Transparent Instruction Set Customization," in *Proc. 37th Ann. IEEE/ACM Intl. Symp. Microarch.*, Portland, USA, 2004, pp. 30-40.

[92] P. Yu and T. Mitra, "Characterizing Embedded Applications for Instruction-Set Extensible Processors," in *DAC '04: Proceedings of the 41st annual conference on Design automation*, 2004, pp. 723-728.

[93] F. Spadini, M. Fertig, and S. J. Patel, "Characterization of repeating dynamic code fragments," Technical Report CHRC-02-09, University of Illinois at Urbana-Champaign2002.

[94] K. Atasu, L. Pozzi, and P. Ienne, "Automatic application-specific instruction-set extensions under microarchitectural constraints," in *Proceedings of the 40th Design Automation Conference*, 2003, pp. 256-261.

[95] S. J. Patel and S. S. Lumetta, "rePLay: A hardware framework for dynamic optimization," in *IEEE Transactions on Computers*, vol. 50, pp. 590-608, June 2001.

[96] J. L. Henning, "SPEC CPU2000: Measuring CPU performance in the new millennium," in *Computer*, vol. 33, pp. 28-35, 2000.

[97] A. C. Beck, M. B. Rutzig, G. Gaydadjiev, and L. Carro, "Run-Time Adaptable Architectures for Heterogeneous Behavior Embedded Systems," in *Proc. 4th Intl. Works. Reconf. Comput.: Architectures, Tools and Applications*, 2008, pp. 111-124.

[98] J. Burns and J. L. Gaudiot, "SMT layout overhead and scalability," in *IEEE Transactions on Parallel and Distributed Systems*, pp. 142-155, 2002.

[99] J. E. Smith, "A study of branch prediction strategies," in *8th International Symposium on Computer Architecture* 1981, pp. 135-148.

[100] E. Z. Bem and L. Petelczyc, "MiniMIPS: a simulation project for the computer architecture laboratory," in *SIGCSE '03*, NY, USA, 2003, pp. 64-68.

[101] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *IEEE International Workshop on Workload Characterization WWC-4*, ed, 2001, pp. 3-14.

[102] V. Allan, R. Jones, R. Lee, and S. Allan, "Software pipelining," in *ACM Computing Surveys (CSUR)*, vol. 27, pp. 367-432, 1995.

[103] B. R. Rau, "Iterative modulo scheduling: An algorithm for software pipelining loops," 1994, pp. 63-74.

[104] Y. Ben-Asher and N. Rotem, "Synthesis for Variable Pipelined Function Units," in *International Symposium on System-on-Chip (SOC)*, Tampere, Finland, 2008, pp. 1-4.

[105] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Spring Joint Computer Conference*, 1967, pp. 483-485.

[106] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the hyperblock," in *Proc. 25th Ann. Intl. Symp. on Microarch.*, ed: IEEE Computer Society Press, 1992, pp. 45-54.

[107] J. V. Leeuwen, *Handbook of Theoretical Computer Science: Algorithms and Complexity*: MIT Press 1990.

[108] M. G. Main and R. J. Lorentz, "An O(n log n) algorithm for finding all repetitions in a string," in *Journal of Algorithms*, vol. 5, pp. 422-432, 1984.

[109] D. Gusfield and J. Stoye, "Linear time algorithms for finding and representing all the tandem repeats in a string," in *Journal of Computer and System Sciences*, vol. 69, pp. 525-546, 2004.

[110] J. Bispo, Y. Sourdis, J. M. P. Cardoso, and S. Vassiliadis, "Regular Expression Matching for Reconfigurable Packet Inspection," in *IEEE International Conference on Field Programmable Technology (FPT'06)*, Bangkok, Thailand, 2006, pp. 119-126.

[111] J. Bispo and J. M. P. Cardoso, "Synthesis of Regular Expressions for FPGAs," in *International Journal of Electronics (IJE)*, vol. 95, pp. 685-704, Taylor & Francis, January 2008.

[112] P. Hsieh. (2008). *Hash functions*. Available: http://www.azillionmonkeys.com/qed/hash.html

[113] W. Sheng, W. He, J. Jiang, and Z. Mao, "Full Automatic Task Compilation Flow from C to REmus Coarse Grain Reconfigurable Media Processor," in *JCIT: Journal of Convergence Information Technology*, vol. 6, pp. 193-202, 2011.

[114] G. Kuzmanov, G. Gaydadjiev, and S. Vassiliadis, "The MOLEN processor prototype," in *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2004, pp. 296-299.

[115] D. D. Gajski, *Principles of Digital Design*: Prentice Hall, 1996.

[116] ISO, "ISO/IEC 9899:TC2 Committee Draft," ed, 2005.

[117] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys, "Chapter 9 Sequencing and scheduling: Algorithms and complexity," in *Handbooks in Operations Research and Management Science*. vol. Volume 4, A. H. G. R. K. S.C Graves and P. H. Zipkin, Eds., ed: Elsevier, 1993, pp. 445-522.

[118] H.-P. Rosinger, "Connecting Customized IP to the MicroBlaze Soft Processor Using the Fast Simplex Link (FSL) Channel," XAPP529 (v1.3), Xilinx2004.

[119] J. Bispo. (2011). *Megablock Tool Suite*. Available: http://suikasoft.com/specs/

[120] I. Xilinx, "Microblaze Software Reference Guide v2.2," in *reference manual*, 2002.

[121] B. H. Fletcher, "FPGA Embedded Processors - Revealing True System Performance " Memec, Embedded Training Program - Embedded Systems Conference (San Francisco), 2005.

[122] P. Alfke, "Xilinx Spartan-6 FPGA User Guide Lite," ed. EE Times - Design: UBM Electronics, 2009.

[123] E. A. Lee, "Programmable DSP Architectures: Part I," in *ASSP Magazine, IEEE,* vol. 5, pp. 4-19, 1988.

[124] F. Vahid, G. Stitt, and R. Lysecky, "Warp processing: Dynamic translation of binaries to FPGA circuits," in *Computer,* vol. 41, pp. 40-46, 2008.

[125] A. Nair and R. Lysecky, "Non-intrusive dynamic application profiler for detailed loop execution characterization," in *International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, 2008, pp. 23-30.

[126] N. Paulino, "Generation of Reconfigurable Circuits from Machine Code," Master Thesis, Engineering Faculty, FEUP, Porto, Portugal, 2011.

[127] V. Zivojnovic, J. M. Velarde, C. Schlager, and H. Meyr, "DSPstone: A DSP-oriented benchmarking methodology," in *Proc. of the Intern. Conf. on Signal Processing and Technology*, 1994, pp. 715-720.

[128] Honeywell. (2012). Available: http://honeywell.com/Pages/Home.aspx

[129] E. Burnette, *Hello, Android: introducing Google's mobile development platform*: Pragmatic Bookshelf, 2009.

[130] A. Diwan, K. S. McKinley, and J. E. B. Moss, "Type-based alias analysis," in *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation* Montreal, Quebec, Canada, 1998, pp. 106-117.

[131] D. W. Wall, "Limits of instruction-level parallelism," in *Fourth International Conference on Architectural Support for Programming Languages and Operatings Systems*, 1991, pp. 176-188.

[132] Digilent, "Atlys Board Reference Manual," 2011.

Appendix A – SRA Implementation

In the context of this thesis and of an MSc thesis concluded at FEUP (Faculty of Engineering of the University of Porto), it was developed a prototype system for dynamic partitioning based on an FPGA board [27, 126]. The system implements some of the ideas presented in this thesis and can automatically move, at runtime, loops from a MicroBlaze executable binary to a Reconfigurable Processing Unit (RPU). We use Megablocks as the partitioning unit. The Megablock detection is done offline, through cycle-accurate simulation of applications during a profile phase. The detected Megablocks are transformed into the IR, which is used to create an RPU tailored to the detected Megablocks. The RPU is an implementation of the SRA architecture (see Section 5.5.3), is runtime reconfigurable and can use several configurations during a single program execution. The implementation uses Single Address Identification (SAI – see Section 5.4) to identify the Megablocks detected during the profiling phase. In our current implementation, *Detection* and *Translation* (i.e., generation of the RPU) is done offline, while *Identification* and *Replacement* is done online, without changes in the executable binary.

Figure A.1 shows the general architecture of the embedded system prototype, which consists of a GPP (a Xilinx MicroBlaze soft-core in this case) and a loosely coupled RPU, both connected to the system bus (in this case, a Processor Local Bus – PLB). To avoid modifications to the GPP, we use an Injector module which monitors the instructions executed by the GPP and communicates with the Reconfiguration Module (RM) to trigger the use of the RPU. The RM is responsible for the RPU reconfiguration. The program code executed by the GPP is in external memory (DDR2). The prototype was designed for an FPGA environment: instead of proposing a single all-purpose RPU, we developed a tool chain which generates the HDL description of an RPU tailored for the application to be run on the system. This step is done automatically.

The target architecture was implemented on a Xilinx Spartan-6 LX45 FPGA [122] and a Digilent Atlys board [132] was used to run the examples.

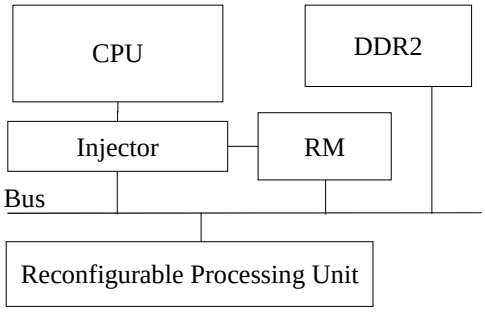


Figure A.1. System Architecture (source: [27]).

Figure A.2 presents the main components of the RPU, and Figure A.3 illustrates a possible array of FUs of an RPU. The RPU uses a peripheral bus interface unit to feed operands and retrieve results through memory mapped registers. The array of FUs contains all the blocks necessary to execute the previously detected Megablocks. The array of FUs is organized in rows with variable number of single-operation FUs. If an operation has a constant input, the RPU generation process tailors the FU to that input (e.g., *bra* FU in Figure A.3). The implementation supports arithmetic and logic operations with integers, including carry operations. Crossbar connections are used between adjacent rows, and are runtime reconfigurable, allowing the use of multiple Megablocks during the execution of a program. Connections spanning more than one row are established by pass-through FUs (*pass* FUs in Figure A.3). RPU configuration is performed by writing to configuration registers. These registers control the routing of the operands through the RPU and indicate which exit conditions should be active.

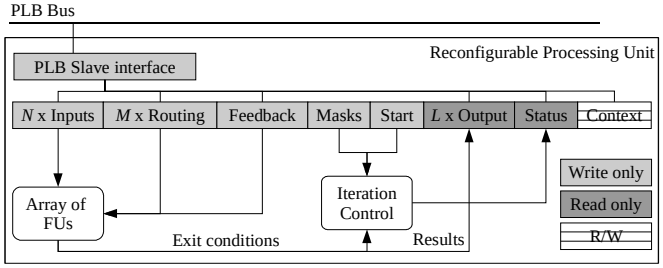


Figure A.2. RPU Architecture (source: [27]).

The RPU was specifically designed to run loops with one path and multiple-exits, such as Megablocks. The number of iterations of the loop does not need to be known before execution: the RPU keeps track of the exits points (e.g., *bne* FU in Figure A.3) of the Megablock and signals when an exit occurs (via a status register). When this happens the

current iteration is discarded, and execution resumes in the GPP at the beginning of the iteration. In the current version of the RPU, all operations complete within one clock cycle and each iteration takes as many clock cycles as the number of rows (depth) of the RPU.

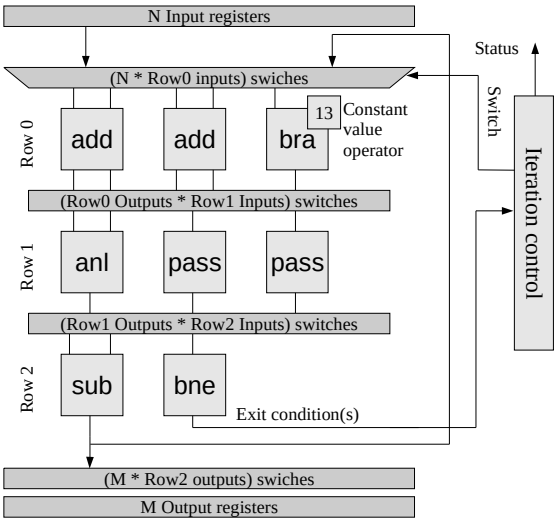


Figure A.3. Array of FUs (source: [27]).

Figure A.4 shows the architecture of the PLB Injector, responsible for interfacing the GPP with the rest of the system, as well as for starting the reconfiguration process. Each RPU configuration is associated to a single instruction address.

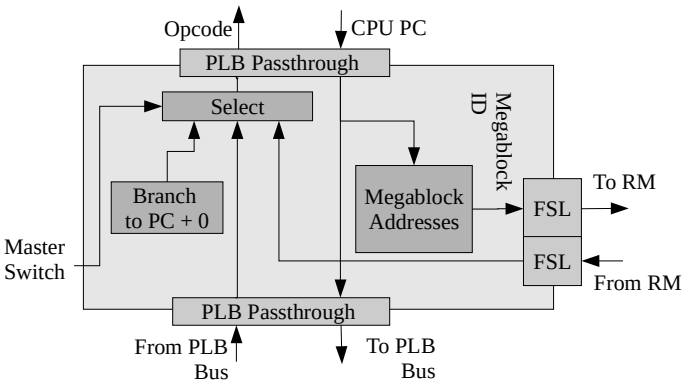


Figure A.4. PLB Injector Architecture (source: [27]).

The Injector monitors the instruction addresses placed on the bus by the GPP until it sees the start of a Megablock. The Injector then stalls the execution of the GPP while reconfiguration is occurring and communicates the Megablock ID to the RM. After reconfiguring the RPU, the RM sends instructions to the Injector with in turn are fed to the GPP. The instructions will cause the GPP to branch to a memory position containing a

previously prepared Communication Routine (CR). By executing it, the GPP copies the contents from its register file to the appropriate input registers of the RPU. When the RPU execution ends, the GPP completes the CR by retrieving the values from the output registers of the RPU and resumes execution of the program code. This way, we can change the execution flow of the GPP without overwriting the original instructions of the program, nor interfering with the original software tool chain.

We developed a tool suite to extract the Megablocks, map them to the RPU, and generate the configuration bits. The input of the tool is the executable file (i.e., the ELF file). The tool suite uses a cycle-accurate simulator of the MicroBlaze to monitor execution traces. The detected Megablocks are then processed by two tools: one generates Verilog descriptions for the RPU and the Injector, and the other generates the CRs for the GPP. The Verilog generation tool parses Megablock information, determines FU sharing across graphs, assigns FUs to rows, adds pass-through units, and generates files containing the placement of FUs. FUs are shared between different Megablocks, since at any given time there is only one Megablock executing in the RPU. The tool also generates routing information to be used at runtime (configuration of the inter-row switches), as well as the data required for Megablock *Identification*. The generated RPU is tailored to a specific set of Megablocks; switching between members of this set is accomplished by configuring the inter-row switches. Input values in the GPP's register file, needed at runtime for Megablock execution, are transmitted to the RPU by executing the CRs on the GPP.

Figure A.5 presents speedups for two scenarios. In the first one, referred as DDR case, results are obtained from execution on the FPGA and running the kernels from DDR memory. The execution times were measured using timers. The second set of results (BRAM case) was obtained by estimation, considering that the programs are stored in internal memory (BRAMs). Results include all communication overheads. We present values for a set of 6 benchmarks, a weighted average of the set and a synthetic benchmark which combines the 6 benchmarks (*merge-all*).

In the DDR scenario, the MicroBlaze has a 23 cycle penalty for each instruction it executes. Most of the achieved speedup comes from avoiding execution of instructions in the GPP and executing operations on the RPU instead. However, for each call to the RPU, the GPP executes a CR which passes the values to the RPU through the bus. Since the CRs are in DDR, they also incur that penalty. The DDR access latency is the main contributor to the very

high overhead of this scenario (approximately 92% of the total execution time, on most cases).

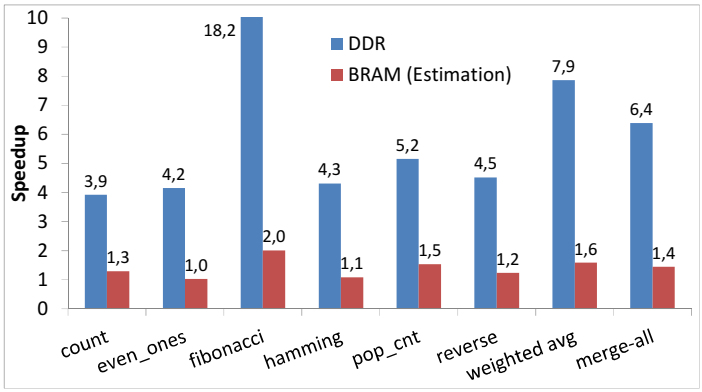


Figure A.5. Speedups for DDR and BRAM scenario (source [27]).

The situation is aggravated by the relatively low number of instructions moved by Megablock call (around 200 instructions executed per loop, in most cases). The overhead includes the identification of the Megablock, configuration of the RPU and execution of the CR. Since the RM fetches instructions from local memories, a large part of the overhead comes from executing the CRs in the GPP afterwards. The speedups measured for the DDR scenario include all overheads and range from 3.9× to 18.2×.

The BRAM scenario is the best possible case for the MicroBlaze processor regarding performance. When considering the BRAM scenario, the speedups and the overheads are significantly reduced, as there is no longer a high penalty for fetching instructions from memory (and as a consequence, the MicroBlaze executes the program faster). We used a cycle-accurate MicroBlaze simulator to calculate the execution time on the GPP. We considered the same overheads of the DDR scenario. We estimated the execution time for CRs considering an average of 1.18 cycles per executed instruction, and added a PLB latency of 9 cycles to write/read operands/results to/from the RPU. RPU execution cycles were calculated by multiplying the RPU's depth and the number of iterations. We estimate speedups between 1.03× and 2.01× (including all overheads).

In both scenarios the speedup of the synthetic benchmark *merge-all* is lower than the speedup of the weighted average. This is mostly due to the overhead of RPU reconfiguration, which only happens in the *merge-all* case.

Table A.1 characterizes the FPGA implementation of the RPUs. The maximum clock frequencies of the RPUs for individual benchmarks ranged from 85 to 139MHz, which is

above the clock frequency of the MicroBlaze. Individual RPUs do not use more than 9% (2369) of the LUTs and 2% (1170) of the FFs. The *merge-all* RPU uses about 55% of the LUTs and 27% of the FFs that would be needed if the RPU was generated with no sharing of FUs.

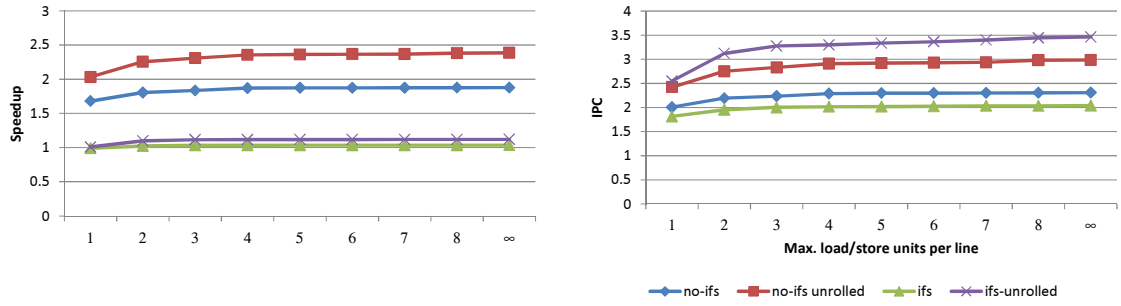
Kernels	**FPGA Implementation**		
	LUTs	***FFs***	***Max. Freq.(MHz)***
count	1433	926	99.30
even_ones	2331	1153	132.83
fibonacci	2369	1170	121.56
hamming	1739	1086	138.08
pop_cnt	1758	1058	137.97
reverse	1780	1072	139.06
merge-all	6325	1719	85.19

Table A.1. RPU FPGA Implementation

Appendix B – Additional Results

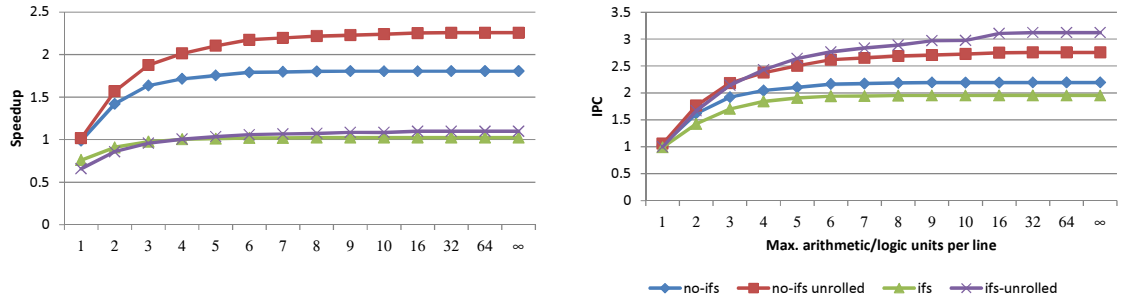
In Chapter 6 we presented several figures with results which were calculated using the arithmetic mean. In this Appendix we present another version of the same figures, which use the geometric mean to calculate the average values.

B-1 Baseline Geometric Means



a) **b)**

Figure B.1. Average a) speedup and b) IPC in the baseline case when varying the maximum number of load/store units (geometric mean).



a) **b)**

Figure B.2. Average a) speedup and b) IPC in the baseline when varying the maximum number of arithmetic/logic units (geometric mean).

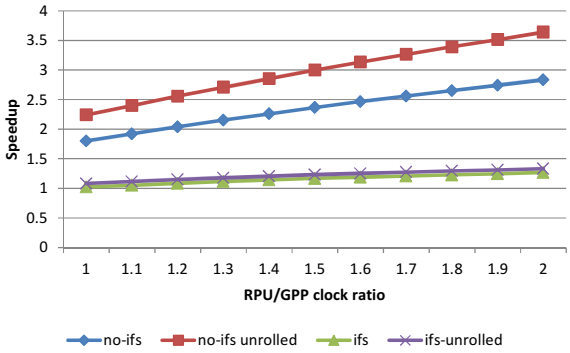
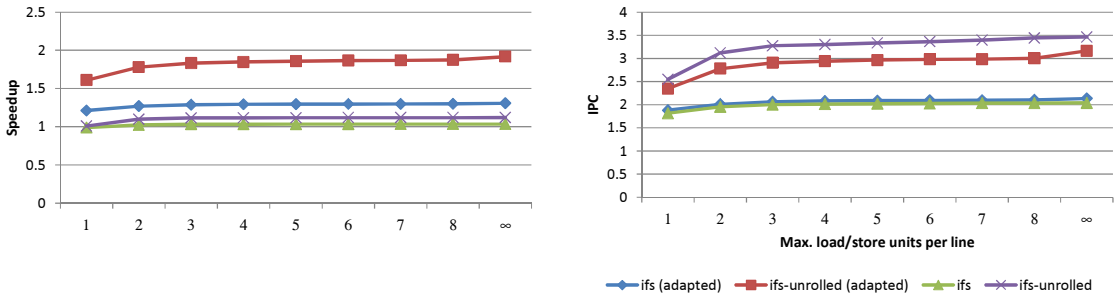


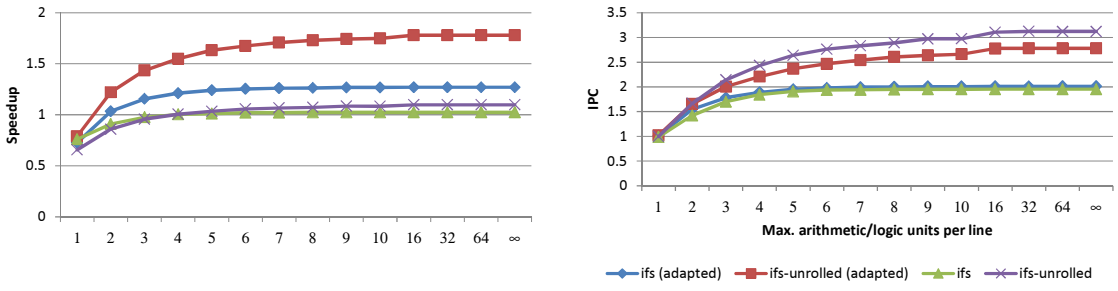
Figure B.3. Average speedup in the baseline case when varying the ration between the RPU and GPP clock (geometric mean).

B-2 If-Conversion Geometric Means



a) **b)**

Figure B.4. Average a) speedup and b) IPC for adapted code when varying the maximum number of load/store units (geometric mean).



a) **b)**

Figure B.5. Average a) speedup and b) IPC for adapted code when varying the maximum number of arithmetic/logic units (geometric mean).

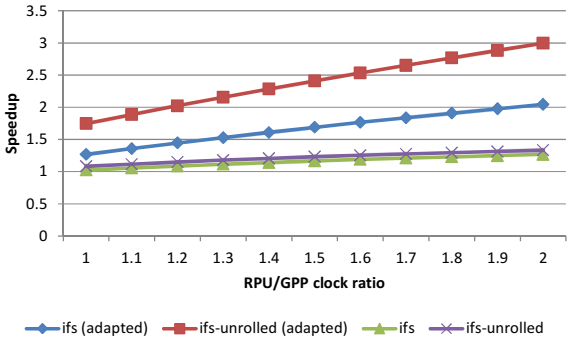
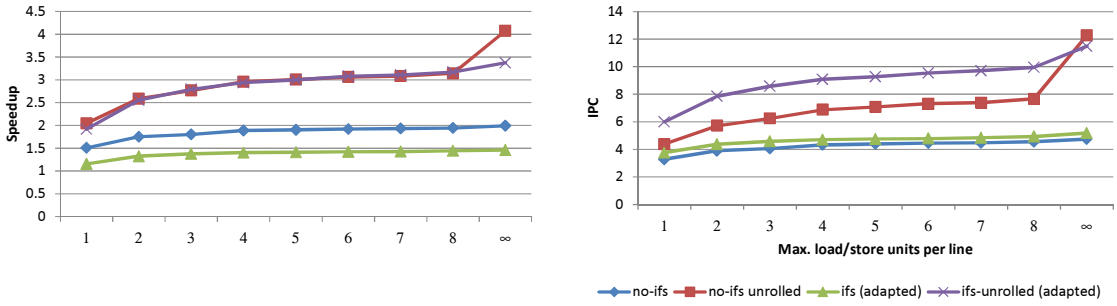


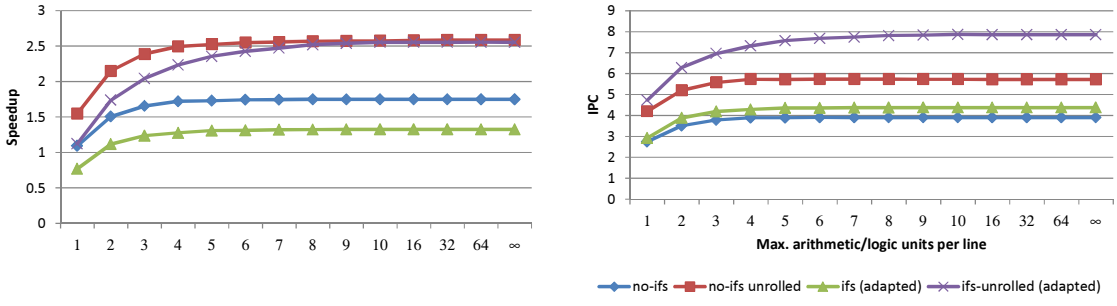
Figure B.6. Average speedup for adapted code when varying the ration between the RPU and GPP clock (geometric mean).

B-3 Pipelining (Sequential Schedule) Geometric Means



a) **b)**

Figure B.7. Average a) speedup and b) IPC for adapted code when varying the maximum number of load/store units (geometric mean).



a) **b)**

Figure B.8. Average a) speedup and b) IPC for adapted code when varying the maximum number of arithmetic/logic units (geometric mean).

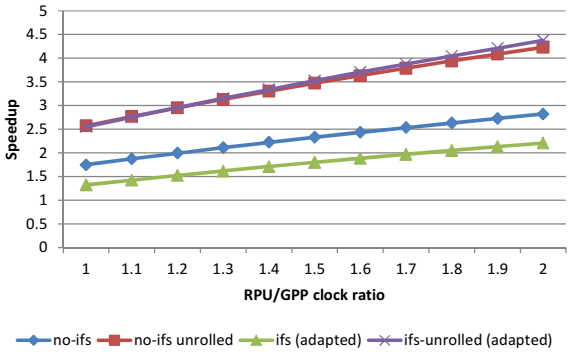
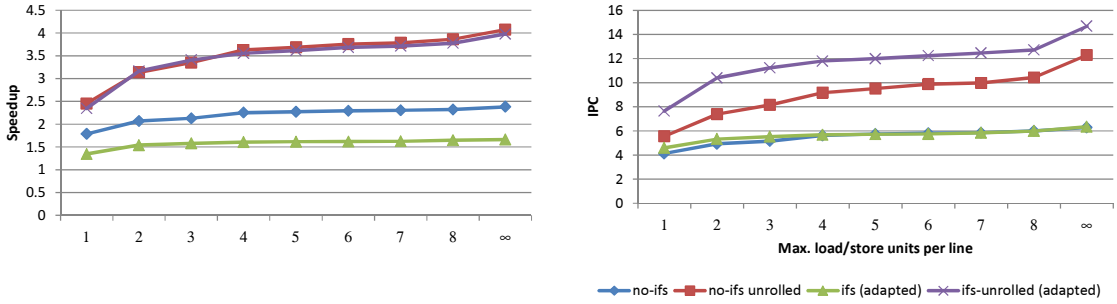


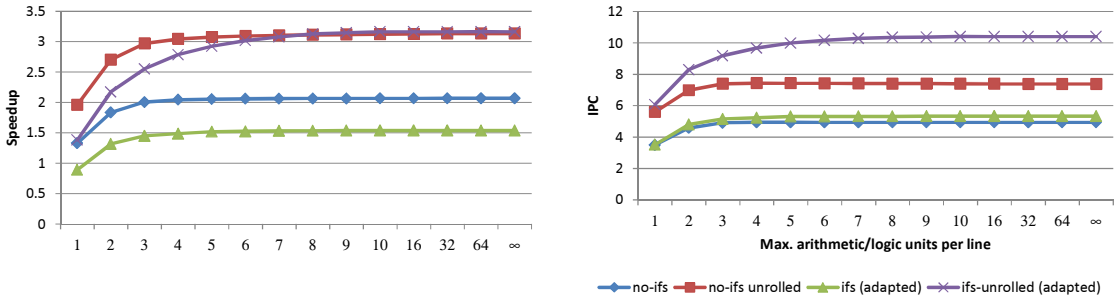
Figure B.9. Average speedup for adapted code when varying the ration between the RPU and GPP clock (geometric mean).

B-4 Pipelining (Overlapping Schedule) Geometric Means



a) **b)**

Figure B.10. Average a) speedup and b) IPC for adapted code when varying the maximum number of load/store units (geometric mean).



a) **b)**

Figure B.11. Average a) speedup and b) IPC for adapted code when varying the maximum number of arithmetic/logic units (geometric mean).

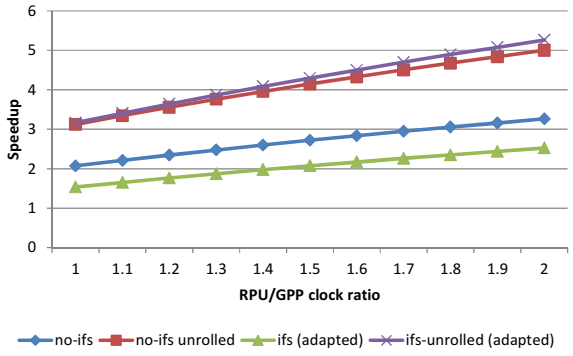
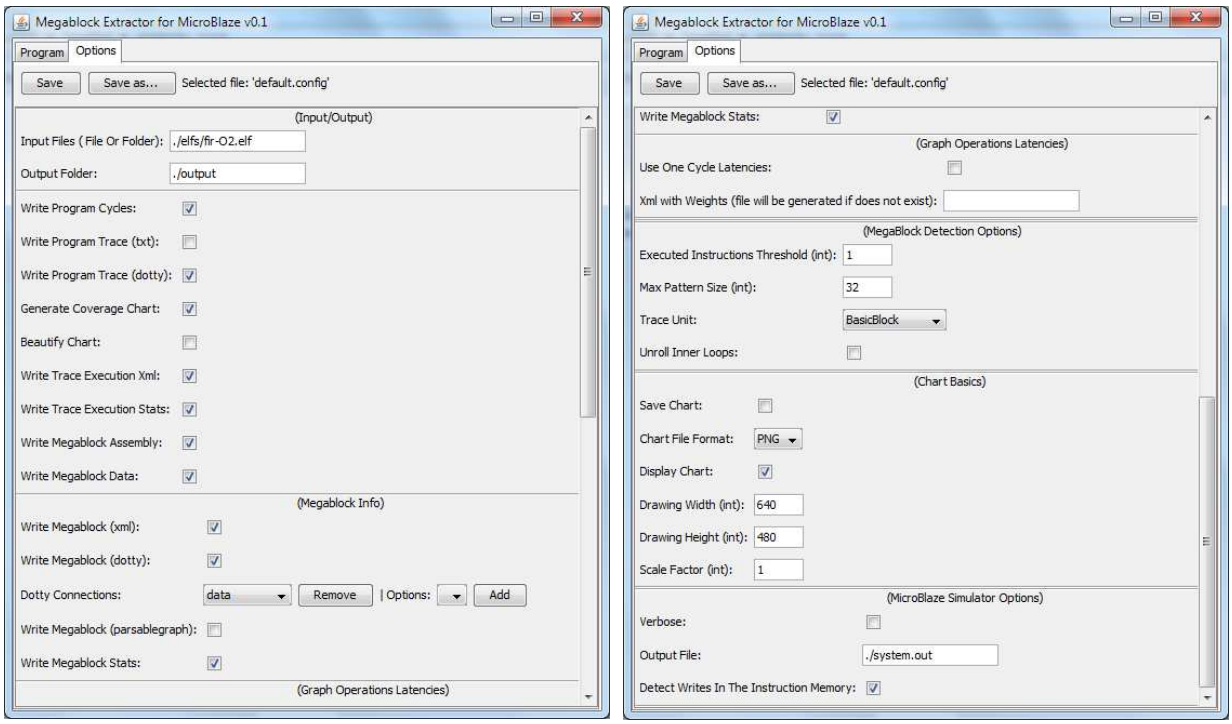


Figure B.12. Average speedup for adapted code when varying the ration between the RPU and GPP clock (geometric mean).

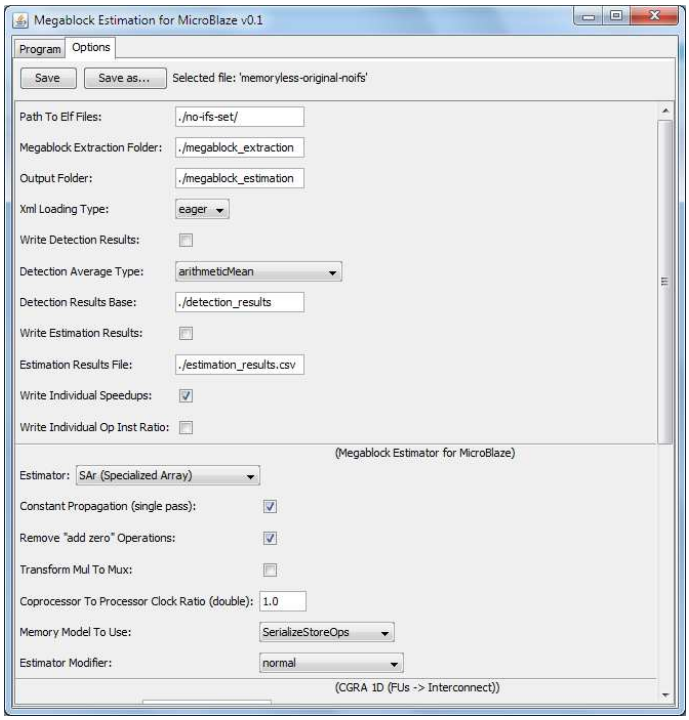
Appendix C – Tools

We developed a number of software tools to evaluate and validate the techniques proposed in this thesis. The tools are available online [119]. We include below screenshots of the most relevant software tools developed in the context of this thesis: Megablock Extraction (see Figure B.9), Megablock Estimation (see Figure B.9), VHDL for Megablocks (see Figure B.9) and VHDL for Megablock Detector (see Figure C.4).

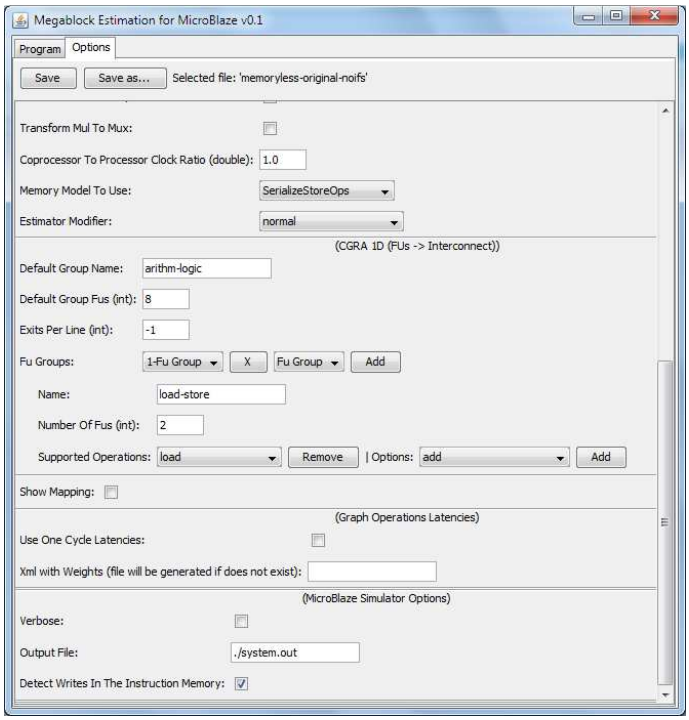


a) **b)**

Figure C.1. Options for program Megablock Extractor.



a)



b)

Figure C.2. Options for program Megablock Estimation.

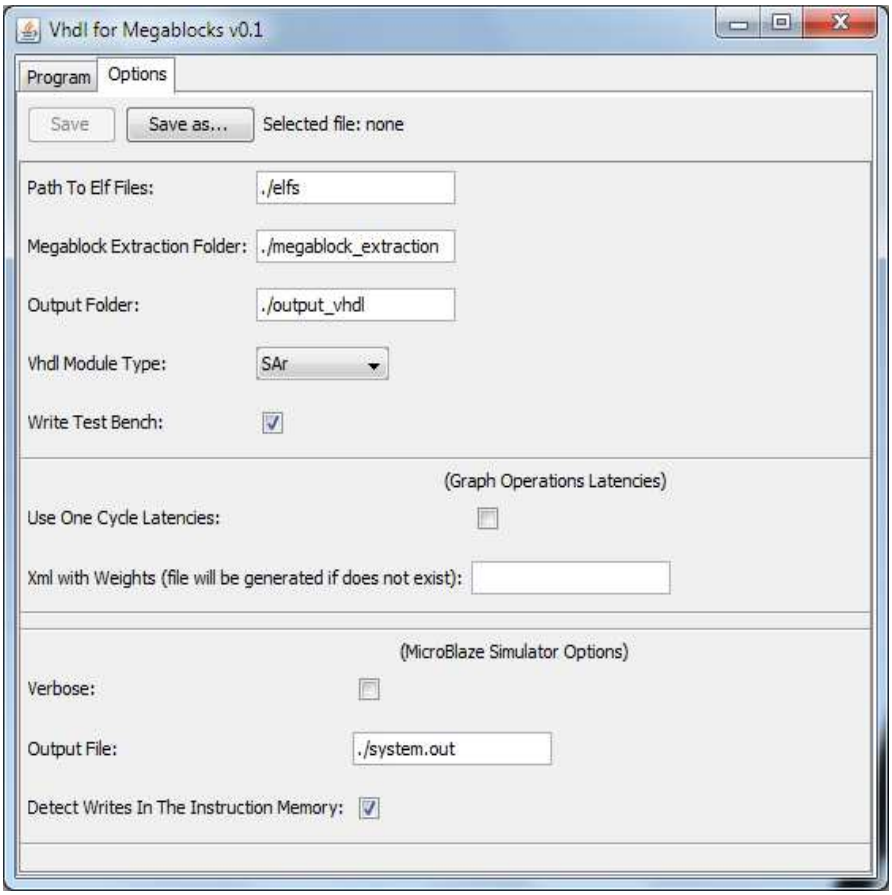


Figure C.3. Options for program VHDL for Megablocks.

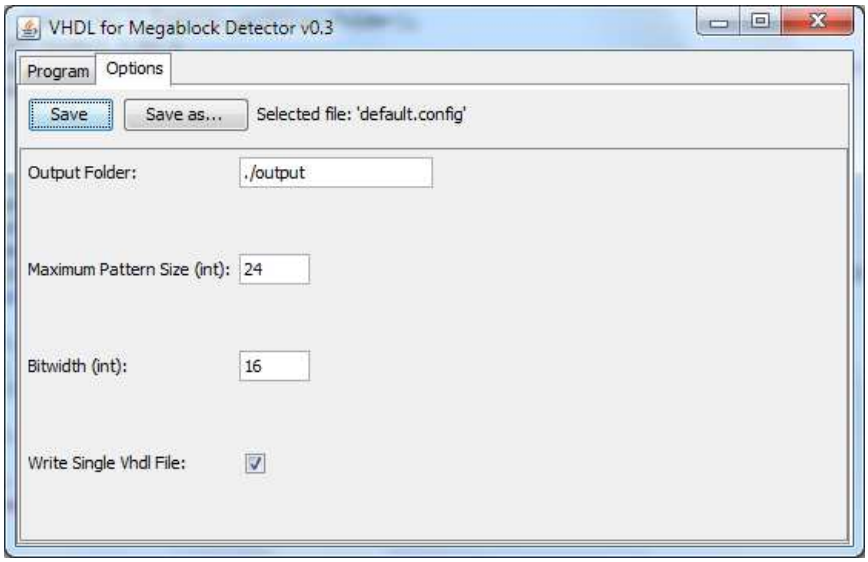


Figure C.4. Options for program VHDL for Megablock Detector.

About the Author

João Bispo received his 5-year Engineer degree in Computer Systems and Informatics from University of Algarve in July 2006.

Prior to graduation, he spent 5 months at the Computer Engineering division of Delft University of Technology as Guest Investigator. Before applying to a doctorate scholarship, he was involved in a bilateral cooperation project and in the context of that project visited the University of Karlsruhe, and spent a year as a researcher at INESC-ID, in Lisbon.

He is a member of the SPeCS research group at FEUP, Porto, and during 2011 was a regular visitor of the lab.

His research interests include Reconfigurable Computing, Architecture Design Exploration, Automatic Generation of Hardware for Specific Applications and Java Platform Programming.

List of Publications Related to this Thesis:

- J. Bispo and J. M. P. Cardoso, "Hardware Pipelining of Runtime-Detected Loops," in 25th Symposium on Integrated Circuits and Systems Design, Brasília, Brazil, 2012.
- J. Bispo, N. Paulino, J. C. Ferreira, and J. M. P. Cardoso, "Transparent Runtime Migration of Loop-Based Traces of Processor Instructions to Reconfigurable Processing Units," *Selected Papers from the 2011 International Conference on Reconfigurable Computing and FPGAs (ReconFig 2011)*, 2012 (under review).
- J. Bispo, N. Paulino, J. C. Ferreira, and J. M. P. Cardoso, "Transparent Trace-Based Binary Acceleration for Reconfigurable HW/SW Systems," *IEEE Transactions on Industrial Informatics*, 2012 (under review).
- J. Bispo, N. Paulino, J. M. P. Cardoso, and J. C. Ferreira, "From Instruction Traces to Specialized Reconfigurable Arrays," in Intl. Conf. on ReConFigurable Comp. and FPGAs (ReConFig'2011), Cancun, Mexico, 2011, pp. 386-391.
- J. Bispo and J. M. P. Cardoso, "Techniques for Dynamically Mapping Computations to Coprocessors," in Intl. Conf. on ReConFigurable Comp. and FPGAs (ReConFig'2011), Cancun, Mexico, 2011, pp. 505-508.
- João Bispo and João M. P. Cardoso, "On Identifying and Optimizing Instruction Sequences for Dynamic Compilation," in *Int'l Conference on Field-Programmable Technology (FPT'10)*, Tsinghua University, Beijing, China: 2010, pp. 437-440.

- João Bispo and João M. P. Cardoso, "Using the Megablock to Partition Programs for Embedded Systems at Runtime," in *INForum - Simpósio de Informática*, Univ. do Minho, Braga, Portugal, 2010.
- João Bispo and João M. P. Cardoso, "On Identifying Segments of Traces for Dynamic Compilation," in *Int'l Conference on Field Programmable Logic and Applications (FPL)*, Milano, Italy, 2010, pp. 263-266.
- João M. P. Cardoso, João Bispo, and Adriano K. Sanches, "The Role of Programming Models on Reconfigurable Computing Fabrics," in *Chapter XII in the book: Behavioral Modeling for Embedded Systems and Technologies: Applications for Design and Implementation*, L. Gomes and J. M. Fernandes, Eds.: IGI Global, 2009.

Publications Unrelated to this Thesis:

- João Bispo and Ana Paiva, "A model for emotional contagion based on the emotional contagion scale," in *3rd Int'l Conference on Affective Computing and Intelligent Interaction (ACII)*, Amsterdam, Netherlands, 2009, pp. 1-6.
- Yiannis Sourdis, João Bispo, João M. P. Cardoso, and Stamatis Vassiliadis, "Regular Expression Matching in Reconfigurable Hardware," *The Journal of VLSI Signal Processing Systems*, vol. 51, pp. 99-121, Springer, April 2008.
- Carlos Morra, João M. P. Cardoso, João Bispo, and Juergen Becker, "Retargeting, Evaluating, and Generating Reconfigurable Array-Based Architectures," in *6th IEEE Symposium on Application Specific Processors (SASP 2008)*, Anaheim CA, USA, 2008, pp. 34–41.
- Carlos Morra, João Bispo, João M. P. Cardoso, and Juergen Becker, "Combining Rewriting-Logic, Architecture Generation, and Simulation to Exploit Coarse-Grained Reconfigurable Architectures," in *The Sixteenth Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'08)*, Stanford, Palo Alto, CA, USA, 2008, pp. 320-321.
- João Bispo and João M. P. Cardoso, "Synthesis of Regular Expressions for FPGAs," *International Journal of Electronics (IJE)*, vol. 95, pp. 685-704, Taylor & Francis, January 2008.
- João Bispo and João M. P. Cardoso, "A Preliminary Idea for Adapting Programs to Parallel Environments," in *Proceedings of ACACES 2008 Poster Abstracts: Advanced Computer Architecture and Compilation for Embedded Systems*, L'Áquila, Italy, 2008, pp. 231-234.
- João Bispo, Yiannis Sourdis, João M. P. Cardoso, and Stamatis Vassiliadis, "Synthesis of Regular Expressions Targeting FPGAs: Current Status and Open Issues," in Int'l Workshop on Applied Reconfigurable Computing (ARC'07), Mangaratiba, Rio de Janeiro, Brazil, 2007, pp. 179-190.
- João Bispo, Yiannis Sourdis, João M. P. Cardoso, and Stamatis Vassiliadis, "Regular Expression Matching for Reconfigurable Packet Inspection," in *IEEE Int'l Conference on Field Programmable Technology (FPT'06)*, Bangkok, Thailand, 2006, pp. 119-126.

Index

2D CGRA, xiii, xvi, xvii, 72, 73, 74, 75, 76, 89, 90, 103
Apple, 3, 148
basic block, xxvi, 8, 30, 31, 34, 36, 37, 38, 39, 43, 44, 61, 97, 99, 102, 134, 135
binary translation, 3, 5, 19, 20, 27, 33, 34, 39
BRAM, xviii, xxiii, 96, 158, 159
C code, xv, xvi, xviii, 2, 45, 46, 78, 79, 93, 133, 135
Catapult C, 2, 147
CCA, 27
CGRA, 13, 71, 76
compilation tools, 53, 54, 145
control-flow, 7, 45, 52, 93, 96, 138
coprocessor, vii, xv, 1, 2, 5, 7, 9, 10, 15, 16, 19, 37, 41, 42, 43, 57, 64, 71, 141, 142, 145, 150
Critical Path Length, xxiii, 103, 107
critical sections, 1, 143
Crusoe, 3, 19
custom hardware, 2, 21
DHSP, xxiv, 2, 4, 7, 16, 41, 141, *See* dynamic partitioning
DIM, 33
dynamic compilation, 3, 15, 17, 19
dynamic partitioning, 5, 15, 19, 21, 70, 73, 102, 141
embedded systems, 1, 4, 5, 15, 17, 19, 22, 26, 27, 30, 33, 39, 142, 147
execution trace, 16, 21, 45, 48, 50, 68, 78, 145
FPGA, 13, 20, 22, 73
fragment, 21, 44, 45, 48, 97, 99
GPP, 7, 14
hardware accelerator, 41
Hardware/software co-design, 1
high-level synthesis, 2, 77
hotspots, 1, 9, 24
if-conversion, xvi, xvii, xxi, 4, 6, 52, 53, 55, 60, 93, 111, 112, 113, 114, 116, 117, 124, 133, 138, 142, 143, 144, 145
ILP, 20, 35, 107, 116, 144
IM, 81
inner loop unrolling, 4, 116, 133, 143
Instruction Set Architecture, 7
Intel, 3, 19, 148
intermediate representation, 4, 15, 19, 50, 55, 56, 58, 64, 96, 119, 138
Intermediate Representation, xii, 5, 41, 50, 57, 142, 144
IPC, 109, 114, 120
Java, 3, 10, 15, 19, 134, 137, 148, 171
JIT, 3, 15, 25
Linux, 3
LM, 81
Macintosh, 3
MacOs, 3
Megablock, 4, 45, 50
Megablock coverage, 6, 93, 101, 102
Megablock *Identification*, xiii, 57, 68, 158
Mentor Graphics, 2
MicroBlaze, xii, xv, xvi, 26, 27, 37, 46, 54, 55, 57, 58, 79, 93, 96, 112, 124, 125, 126, 133, 135, 137, 138, 142, 144, 151, 153, 155, 158, 159, 160
Microprocessor, 148, 152
MSI, 68, 97
Multi-core, 147
overhead, 3, 9, 12, 17, 23, 27, 31, 36, 37, 39, 42, 43, 52, 74, 75, 98, 99, 107, 125, 134, 152, 159
parallel computing, 5, 41, 44
partitioning unit, 4, 5, 155
pattern elements, xxv, xxvi, 47, 48, 49, 61, 62, 64
pattern unit, 48, 97, 102
pattern-matching, 4, 142
Pentium, 3, 20, 148
pipelining, v, vii, xvii, xviii, 4, 6, 19, 33, 35, 38, 57, 73, 76, 77, 80, 81, 84, 86, 89, 90, 91, 93, 109, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 139, 141, 142, 143, 145, 153
PLB, xviii, xxv, 155, 157, 159
PowerPC, 3
profile phase, 5, 57, 155
proof-of-concept, 6, 102, 122, 124

reconfigurable architecture, 20, 21, 24, 38, 149, 150
reconfigurable computing, 19, 20, 149, 150
Rosetta, 3
RPU, 11, 16, 70, 81
Runtime Reconfiguration, vii
SAI, xxi, xxvi, 57, 68, 69, 91, 97, 98, 99, 102, 110, 144, 155
SAr, 73, 103, 125
sequential code, vii, 4, 5, 41, 44
SM, 81
Source Code, xii, 52
source-to-source, 5, 41, 52, 56, 145
SRA, xiii, xiv, xvi, xxvi, 75, 76, 90, 103, 155
synthesis, 13, 23, 74, 77, 96, 129
systems-on-chip, 1
Transmeta, 148
Verilog, 1, 13, 147, 149, 158
VHDL, xix, 13, 96, 122, 124, 129, 142, 149, 151, 167, 169
Warp, 22, 74, 99, 133
Windows, 3
x86, 3, 19, 20