Compilation of Stream Programs onto Embedded Multicore Architectures

by

Weijia Che

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved July 2012 by the
Graduate Supervisory Committee:

Karamvir Chatha, Chair
Sarma Vrudhula
Chaitali Chakrabarti
Aviral Shrivastava

ARIZONA STATE UNIVERSITY

August 2012

ABSTRACT

In recent years, we have observed the prevalence of stream applications in many embedded domains. Stream programs distinguish themselves from traditional sequential programming languages through well defined independent actors, explicit data communication, and stable code/data access patterns. In order to achieve high performance and low power, scratch pad memory (SPM) has been introduced in today's embedded multicore processors. Current design frameworks for developing stream applications on SPM enhanced embedded architectures typically do not include a compiler that can perform automatic partitioning, mapping and scheduling under limited on-chip SPM capacities and memory access delays. Consequently, many designs are implemented manually, which leads to lengthy tasks and inferior designs. In this work, optimization techniques that automatically compile stream programs onto embedded multi-core architectures are proposed. As an initial case study, we implemented an automatic target recognition (ATR) algorithm on the IBM Cell Broadband Engine (BE) [17]. Then integer linear programming (ILP) [19] and heuristic [18] approaches were proposed to schedule stream programs on a single core embedded processor that has an SPM with code overlay. Later, ILP and heuristic approaches for Compiling Stream programs on SPM enhanced Multicore Processors (CSMP) [20] were studied. The proposed CSMP ILP and heuristic approaches do not optimize for cycles in stream applications. Further, the number of software pipeline stages in the implementation is dependent on actor to processing engine (PE) mapping and is uncontrollable. We next presented a Retiming technique for Throughput optimization on Embedded Multi-core processors (RTEM) [14].

RTEM approach inherently handles cycles and can accept an upper bound on the number of software pipeline stages to be generated. We further enhanced RTEM by incorporating unrolling (URSTEM) [16] that preserves all the beneficial properties of RTEM heuristic and also scales with the number of PEs through unrolling.

ACKNOWLEDGEMENTS

First I would like to give my special thanks to my graduate advisor Dr. Karam Chatha for directing me throughout my Ph.D study. It is his guidance that brought me into the exciting and challenging world of embedded systems. The vision he shared with me has always been the beacon that leads me through darkness and confusion. His kind help contributes to almost all the research that I have done and all the papers that I have published. I can never forget the efforts that Dr. Chatha devoted to push my first paper through. It is his selfless dedication that motivates and encourages me to conquer whatever challenges come in front of me. All the days and nights we worked together are going to be precious memories that I will cherish forever. I am honored to have the opportunity to work with Dr. Chatha during the five years of my Ph.D study. His wisdom will continue to guide me and help me success in my career after graduation.

I also would like to thank my committee members for agreeing to join my committee. I sincerely appreciate their time and professional guidance. Without their kind support I will not be able to achieve what I have achieved today. I would like to thank Dr. Vrudhula for his dedication to Embedded Systems Consortium that benefits all the members including me. I would like to thank Dr. Shrivastava for his active enthusiasm and contributions for the collaborations of the compiler micro-architecture lab and the computing systems research lab. I would like to thank Dr. Chakrabarti for her professional knowledge and guidance. I consider myself very lucky to have such a kind and helpful committee for my Ph.D study.

All the lab members who have spent time with me, discussed research problems with me, conducted projects with me, and wrote papers with me, I would like to take this opportunity to thank them too. It is their support and accompaniment that took me through the toughest times. I would like to thank Sushu Zhang for introducing me to the computing systems research lab and getting me started when I was a fresh Ph.D student. She is a great mentor and provided me endless of support during my entire Ph.D study. I would like to thank Micheal Baker for all the exciting discussions we had and the projects we worked together. I would like to thank Glenn Leary for always being a great help whenever a challenge is encountered. I also like to thank Amrit Panda for his professional knowledge in architectures that has enlightened me in so many different ways, HaeSeung Lee for the paper we worked together, and Jyothi Swaroop for the support and discussions we shared. There are many other group members who have been of great help. Pravin Dalale, Nikhil Ghadge, Derek Woodman, and many others. They all helped me in one way or another and I would like to thank them for their time and support.

I am luckily to have both technical and financial supports from outside of Arizona State University and I would like to show my appreciation here. In particular, the collaboration with Raytheon company provided me with the experience of programming on the IBM Cell Broadband Engine and led me to success of the compiler work that follows. The Semiconductor Research Corporation has been a great help in both leading me technically and supporting my study financially. The task of system-level design of streaming applications on domain specific multi-sore processors has gradually evolved as part of my dissertation. The internship that was offered by The MathWorks provided me the first opportunity to step into industry and trained me from

an entirely different perspective from academia. The collaborations with the industry partners not only helped me successful in my academic study but also lays the foundation for a bright future for me after my graduation.

Last but not least, I would like to thank the people who have known me for the longest time and have been there for me throughout my life, my parents and my brother. I cannot put it into words to show my appreciation and love. Except that they are always in my heart even when I am thousands of miles away from home.

TABLE OF CONTENTS

Page

LIST OF TABLES xvi

LIST OF FIGURES xvii

1 INTRODUCTION 1

1.1 StreamIt Language 3

1.2 IBM Cell BE 6

1.3 Contributions 8

CHAPTER

2 DESIGN OF AN ATR ALGORITHM ON IBM CELL BROADBAND ENGINE 11

2.1 Automatic Target Recognition 12

2.2 Previous Work 13

2.3 Reference Implementation on Pentium4 PC 15

2.4 Design of ATR on the IBM Cell BE 16

2.4.1 Porting ATR to PPE 17

2.4.2 Parallelized implementation of ATR 18

2.4.3 Application parallelization on 6 SPEs 18

2.4.4 Double buffering 19

2.4.5 2-D FFT 20

2.4.6 Matrix transpose 20

CHAPTER Page

2.4.7 Small function relocation 21
2.4.8 SIMD & loop unrolling 22
2.4.9 Pretouch memory 22
2.4.10 Mailbox communication 23
2.5 Experimental Results 24
2.6 Summary 25
3 SCHEDULING OF STREAM PROGRAMS ON ONE EMBEDDED CORE WITH CODE OVERLAY 33
3.1 Preliminaries 34
3.1.1 Code overlay 35
3.1.2 Basic pre-fetching and deep pre-fetching 36
3.1.3 Data overlay 39
3.2 Design Trade-offs 40
3.2.1 PASS generation 40
3.2.2 Actor to region assignments 43
3.2.3 Segmentation 44
3.2.4 Data overlay 45
3.3 Problem Formulation 46
3.4 Related Work 47

3.5 3-stage ILP Without Pre-fetching 49
3.5.1 Stage 1 ILP: Scheduling 50
3.5.1.1 Decision variables 50
3.5.1.2 Derived variables 50
3.5.1.3 Constraints 51
3.5.1.4 Objective function 53
3.5.2 Stage 2 ILP: Region Assignment 54
3.5.2.1 Variables derived from Stage 1 54
3.5.2.2 Decision variables 54
3.5.2.3 Derived variables 54
3.5.2.4 Constraints 55
3.5.2.5 Objective function 56
3.5.3 Stage 3 ILP: Segmentation 56
3.5.3.1 Results derived from Stage 1 and Stage 2 . . 56
3.5.3.2 Decision variables 56
3.5.3.3 Derived variables 57
3.5.3.4 Constraints 59
3.5.3.5 Objective function 59
3.6 3-stage ILP With Pre-fetching 60

3.6.1 Stage 1 ILP: Scheduling 60

3.6.1.1 Additional derived variables 60

3.6.1.2 Modified constraints 61

3.6.1.3 Updated objective function 61

3.6.2 Stage 2 ILP: Region Assignment 61

3.6.2.1 Additional variables derived from Stage 1 . . 62

3.6.2.2 Additional derived variables 62

3.6.2.3 Updated constraints 62

3.6.2.4 Updated objective function 63

3.6.3 Stage 3 ILP: Segmentation 63

3.6.3.1 Additional derived variables 63

3.6.3.2 Updated derived variables 63

3.6.3.3 Updated constraint 64

3.7 SDF Scheduling Heuristic 64

3.7.1 Code overlay overhead calculation 64

3.7.2 Overall description 66

3.7.3 Region assignments 68

3.7.4 Segmentation 69

3.7.5 Overall algorithm complexity 70

3.8 Extensions to SDF Scheduling Heuristic 71

3.8.1 Basic pre-fetching optimization 71

3.8.2 Deep pre-fetching optimization 73

3.8.3 Data overlay optimization 75

3.9 Experimental Results 81

3.9.1 Experimental setup 81

3.9.2 Comparison of 3-stage ILP and heuristic with minimum buffer scheduling 82

3.9.3 Impact of each optimization 84

3.9.4 Impact of SPM size 87

3.9.5 Code overlay evolution 88

3.9.6 Impact of scaling DMA cost 89

3.9.7 Impact of scaling code size and run time 91

3.10 Summary 92

4 SCHEDULING OF STREAM PROGRAMS ON SPM BASED MULTICORE PROCESSORS THROUGH FUSION AND FISSION . . 94

4.1 Motivation 94

4.2 Previous Work 96

4.2.1 Problem description 97

CHAPTER Page

4.3 Integer Linear Programming Approach 100

4.3.1 Constraints 100

4.3.2 Objective function 104

4.3.2.1 Computation cost 104

4.3.2.2 Communication cost 105

4.3.2.3 Execution cost 105

4.3.2.4 Effective execution cost 105

4.3.2.5 Overall cost function 105

4.4 Heuristic Approach 106

4.4.1 Overlay Scheme 108

4.4.2 Cost functions 108

4.4.2.1 Buffer usage 109

4.4.2.2 Computation cost 109

4.4.2.3 Communication cost 109

4.4.2.4 Overall cost of solution 110

4.5 Experimental Results 111

4.5.1 Comparisons with 256KB SPE memory 112

4.5.2 Comparisons with 16KB SPE memory 115

4.6 Summary 117

CHAPTER Page

5 SCHEDULING OF STREAM PROGRAMS ON SPM BASED MULTICORE PROCESSORS THROUGH RETIMING 119

5.1 Retiming 120

5.2 Motivation 121

5.3 Problem Description 124

5.4 Related Work 125

5.5 Resolving Cycles 127

5.6 Pre-processing 131

5.7 Integer Linear Programming Approach 132

5.7.1 Decision variables 134

5.7.2 Derived variables 134

5.7.3 Constraints 136

5.7.4 Cost functions 139

5.7.5 Objective function 140

5.8 RTEM Heuristic Approach 140

5.8.1 AlgorithmII 141

5.8.1.1 Calculation of $\tau_c(v)$ 142

5.8.1.2 Calculation of $\tau_o(v)$ 143

5.8.2 AlgorithmFEAS 144

5.8.3 AlgorithmRTEM 145

5.8.3.1 Implementation of smart double buffering scheme 146

5.8.4 Complexity 148

5.9 Experimental Results 148

5.9.1 Experimental setup 148

5.9.2 Overall performance comparison 148

5.9.3 Comparison with different SPM sizes 151

5.9.4 Comparison with different double buffering schemes . . 153

5.9.5 Comparison with different number of pipeline stages and PEs 154

5.10 Summary 156

6 UNROLLING AND RETIMING OF STREAM PROGRAMS ON SPM BASED MULTICORE PROCESSORS 158

6.1 Problem Description 161

6.2 Related Work 163

6.3 URSEM Heuristic Approach 165

6.3.1 Pre-processing 165

6.3.2 URSEM heuristic algorithm 165

6.3.3 AlgorithmRDL 167

CHAPTER Page

6.3.4 AlgorithmDeltaCD 168

6.3.4.1 Construction of RG to PE mapping 169

6.3.4.2 Calculation of code, data memory usage . . . 170

6.3.4.3 Calculation of processor workload 171

6.3.4.4 Calculation of $\tau_c(v)$ 171

6.3.4.5 Calculation of $\tau_o(v)$ 173

6.3.5 Algorithm Complexity 173

6.4 Experimental Results 174

6.4.0.1 Overall performance comparison 174

6.4.0.2 Impact of optimizations 175

6.4.0.3 Performance scaling with PEs 176

6.4.1 Performance scaling with Delays 178

6.4.1.1 Performance scaling with SPMs 178

6.5 Summary 179

7 CONCLUSION AND FUTURE WORK 181

7.1 Future Work on OpenCL 183

7.2 Future Work on TI Multicore 184

REFERENCES 186

CHAPTER Page

APPENDIX 196

APPENDIX A 196

APPENDIX B 205

LIST OF TABLES

Table Page

1.1 Benchmark Specifications 5

2.1 Stages in the ATR allgorithm. 14

2.2 Profile Analysis of ATR on host PC and PPE. 17

2.3 Profile Analysis of ATR on PPE and SPEs. 27

2.4 Computation/Communication Latency Comparison. 28

2.5 Problem State Area Mapped Mailbox Performance. 30

3.1 Design trade-offs with PASS generation. Buffer Usage and Available Code Memory in the table are represented in bytes. 42

3.2 Design trade-offs with actor to region assignments. Available Code Memory and Region Sizes in the table are represented in bytes. . 44

3.3 Architecture and SDF Description 47

3.4 Benchmark Specifications 81

4.1 Architecture and SDF Description 98

4.2 Base and Derived Variables 102

4.3 Maximum SPE Buffer usage of SGMS, CSPM_{ilp} & CSPM_{heu} (BYTES). 112

4.4 Run time of SGMS, CSPM_{ilp} & CSPM_{heu} (SECONDS). 113

4.5 Maximum communication buffer usage of SGMS, CSMP_{ilp} & CSMP_{heu}. 118

5.1 SDF and architecture specification 123

LIST OF FIGURES

Figure Page

1.1 StreamIt program example. 4

1.2 IBM Cell BE architecture overview. 6

1.3 DMA performance with 6 SPEs. 7

2.1 ATR algorithm overview. 13

2.2 2-Dimensional real FFT. 26

2.3 Data partition for the morphological filter 27

2.4 fft_1d_r2 profile on one SPE. 28

2.5 Unoptimized and Optimized FDF. 29

2.6 Data partition in matrix transpose module. 29

2.7 Task schedule. The modules annotated with SPE execute in parallel on 6 SPEs. 30

2.8 Frequency Domain Filtering. 30

2.9 Latency comparison for integrating each optimization. 31

2.10 Computation Comparison for Each Process in ATR. 32

3.1 Segment-region code overlay overview. 35

3.2 Code pre-fetching and deep pre-fetching. 37

3.3 DMA engine status with basic pre-fetching. 37

3.4 DMA engine status with deep pre-fetching. 39

3.5 Data overlay overview. 40

3.6 Stream program with one producer and one consumer. The code, token sizes are in bytes. 41

3.7 Stream program with four actors. The code sizes are in bytes. . . 43

3.8 Design trade-offs with data overlay. 46

Figure Page

3.9 Our 3-stage ILP and heuristic approaches compared with minimum buffer scheduling. 83

3.10 Impact of each optimization in our heuristic approach. 84

3.11 Memory usage comparison. 85

3.12 SPM size variation (1st set). 88

3.13 SPM size variation (2nd set). 89

3.14 Overlay cost evolution with SDF scheduling. 90

3.15 Impacts of scaling DMA overhead. 91

3.16 Impacts of scaling code size and run time. 92

4.1 (A) filter-batch, and batch-processor mapping; (B) steady-state . 101

4.2 Computation and communication costs for SGMS 110

4.3 Computation and communication costs for CSMP_{ilp} and CSMP_{heu} 111

4.4 SGMS, CSMP_{ilp} and CSMP_{heu} with 256KB SPM. 114

4.5 SGMS, CSMP_{ilp} and CSMP_{heu} with 16KB SPM. 115

4.6 Performance of CSMP_{ilp} and CSMP_{heu} with 256KB and 16KB SPM 116

4.7 DCT with 256KB and 16KB SPM 117

5.1 Retiming example. 120

5.2 Single appearance SDF construction from a multiple appearance stream program without cycles. 128

5.3 Single appearance SDF construction from a multiple appearance stream program with a tight cycle dependence. 130

5.4 Single appearance SDF construction from a multiple appearance stream program with a loose cycle dependence. 131

Figure Page

5.5 A simple example of retiming a stream program with 3 PEs and 4 software pipeline stages. 134

5.6 RTEM ILP and heuristic against CSMP ILP and heuristic approaches. 149

5.7 Performance comparison with different size of SPMs. 152

5.8 Smart double buffering against blind double buffering and no double buffering. 153

5.9 Performance comparison with different pipeline stages. 155

5.10 Performance comparison with different number of PEs. 156

6.1 Code overlay and data overlay. 161

6.2 Processor memory state transitions. 173

6.3 Overall performance comparison. 175

6.4 Impact of optimizations. 176

6.5 Performance scaling with PEs. 177

6.6 Performance scaling with Delays. 179

6.7 Performance scaling with SPMs. 180

Chapter 1

INTRODUCTION

Recent years have witnessed the recognition of stream computing as an important model of computation in many embedded system domains, such as signal processing, multimedia, and network processing. Stream applications share common characteristics such as well defined independent actors, explicit exposed data communication, and stable code/data access patterns. Due to these characteristics, several languages have been developed in the past few years to model stream applications (formally referenced as stream languages). Example stream languages include StreamIt [71], CAL [29], CUDA [60], Brook [13]. Many of these languages model the compute intensive units of a program as actors and expose the data communication among distinct actors as FIFOs. Many stream languages in fact implement the synchronous data flow (SDF) model of computation.

Processor designers have responded to the high performance requirements of stream applications by developing domain specific multi-core processors. Examples of commercial processors that are aimed at streaming applications include IBM Cell Broadband Engine (BE) [22] [64], Tilera64 [72], Intel Larrabee [68], Nvidia GeForce series [43], Ageia's PhysX [76], TI TMS320C6472 [73] and many DSPs. In many of these embedded architectures, SPM has replaced traditional caches for faster access time, smaller chip area, and lower power/energy consumption. In an SPM enhanced design, the workload of dynamic management of the limited on-chip SPM is shifted from

the hardware side to a programmer or compiler. Data and code transfers among various memory elements are realized through direct memory access (DMA) engines and are completely software managed.

Current design frameworks for developing stream applications on SPM enhanced embedded architectures typically do not include a compiler that can automatically address the limited on-chip SPM and memory access delays and efficiently perform partitioning, mapping and scheduling under various design trade-offs. Consequently, many designs are implemented manually. In a manual design, the programmer has to manage the code and data transfers among various memory elements during the entire program life time. Due to the limited on-chip SPM capacity, code overlay and data overlay schemes have to be implemented for sharing the same physical memory with different code/data segments. To amortize memory access delays, double buffering (DB) for overlapping data communication with computation has to be evaluated. The introduction of double buffering scheme requires storing an extra copy of data, which could result in additional code and data overlay overhead. Given the challenges and various design trade-offs discussed above, manual development of stream programs on SPM enhanced architectures often leads to lengthy design time and inferior quality designs.

In this dissertation, we propose optimization techniques that automatically compile stream programs onto embedded multicore architectures. As an initial case study, we implement an automatic target recognition (ATR) algorithm on the IBM Cell BE [17]. Then integer linear programming (ILP) and heuristic approaches are proposed to schedule stream programs on a single core embedded processor that has an SPM with code overlay [19] [18]. Later, ILP

and heuristic approaches for Compiling Stream programs on SPM equipped Multicore Processors are studied (named as CSMP ILP and CSMP heuristic respectively) [20]. The CSMP ILP and heuristic approaches cannot optimize feedback cycles in stream programs and also could result in very deep software pipeline stages. We next present a Retiming algorithm for Throughput optimization on Embedded Multicore processors (named as RTEM) [14]. RTEM heuristic inherently optimizes feedback cycles and allows a user to specify the number of software pipeline stages to be generated. RTEM heuristic relies on the existing parallelism in an application therefore may not scale with the number of PEs of an embedded multicore processor. Finally, we provide Unrolling and Retiming of Stream programs on Embedded Multicore processors (URSEM) that preserves all the beneficial properties of RTEM and also scales with the number of PEs.

Since StreamIt language and IBM Cell BE are used throughout our experiments as the software and hardware specifications, we begin with a discussion that introduces both of them.

1.1 StreamIt Language

We adopted StreamIt language from MIT [71] as the input specification to our experiments. StreamIt programs implement the synchronous data flow (SDF) model of computation [49]. Four basic structures, namely filter, pipeline, split-join, and feedback-loop are provided by StreamIt to construct a stream program. The actors/filters[1] in an SDF represent small compute intensive units in a stream application. The edges in an SDF stand for data communica-

[1]Filter is the formal name for an actor in a StreamIt program. We use actor and filter interchangeably in this chapter.

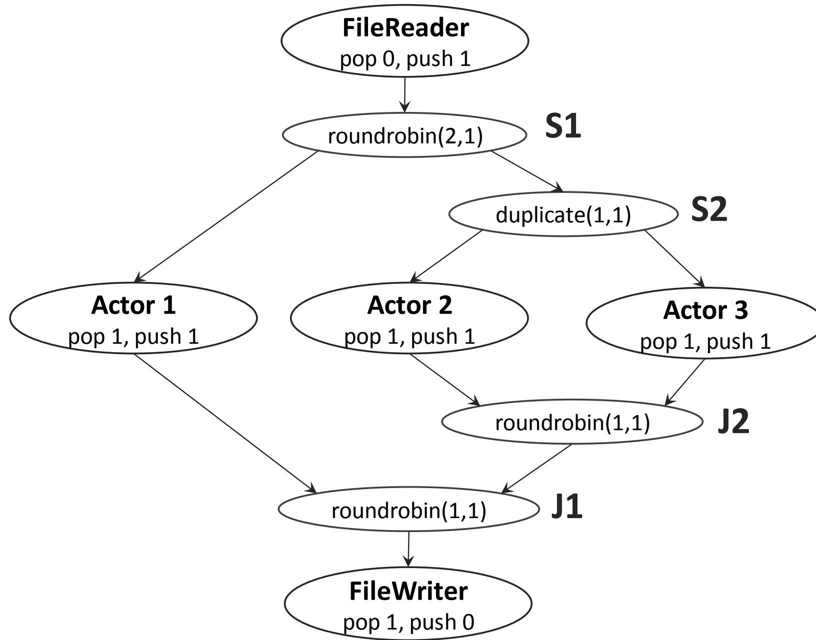


Figure 1.1: StreamIt program example.

tion/FIFOs among actors. At each iteration, an actor consumes a constant amount of data from its incoming edges and produces a constant amount of data to its outgoing edges. We require the SDF to be consistent [49] in our technique. In fact, all legal StreamIt programs are consistent by default. In other words, there exists a steady-state execution state for a valid stream program. In Figure 1.1 we provide an example of a simple stream program. In the figure, FileReader and FileWriter handle the I/O operations of the program. FileReader pushes one token (denoted by push 1) to its outgoing edge in each execution and thus serves as a token source. FileWriter consumes one token from its incoming edge in each execution and serves as a token sink. There are two split-join structures in the program, one consists of two roundrobin filters (S1 and J1) and the other is constructed with a pair of duplicate and roundrobin (S2 and J2). roundrobin and duplicate filters are built in data flow

Table 1.1: Benchmark Specifications

Benchmark Names	Number of Actors	Number of Edges
Beamformer	56	58
Bitonicsort	40	46
Channelvocoder	55	70
DCT	40	69
DES	53	60
FFT	17	16
Filterbank	85	99
Fmradio	43	53
MPEG2-Subset	23	26
Serpentfull	120	128
TDE	29	28
Vocoder	116	150
Average	56	67

*The size of Bitonicsort is 8 points. The size of DCT is 8 by 8, and the size of FFT is 256 single precision complex points.

filters in the StreamIt language. The weight array attached to each roundrobin or duplicate denotes the data tokens it pops (J1, J2) from its incoming edges or pushes (S1, S2) to its outgoing edges in each execution. The difference between a roundrobin filter and a duplicate filter is that a roundrobin only collects or splits data tokens according to its weight array while a duplicate first replicates each token according to its weight array and then splits them to its outgoing edges.

Twelve benchmarks that are delivered with StreamIt compiler version 2.1.1 will be used extensively to in our experiments. Table 1.1 details the characteristics of each benchmark. The first column provide us with the benchmark names. The second and third columns provides us with the number of actors and edges in each benchmark. The last row calculates the average for each column.

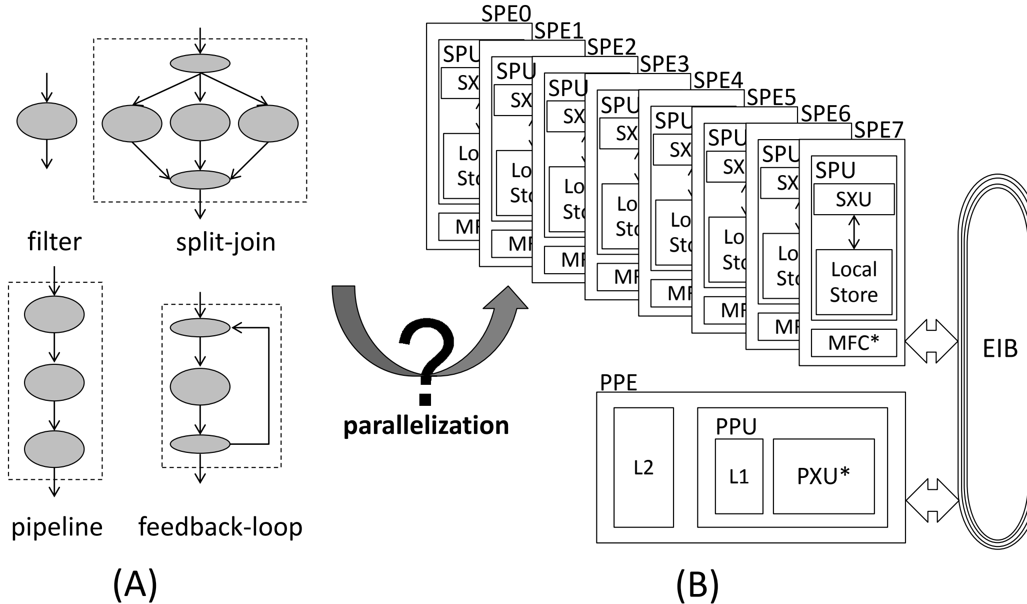


Figure 1.2: IBM Cell BE architecture overview.

1.2 IBM Cell BE

IBM Cell BE was used as the target architecture to evaluate the efficiency of our techniques. IBM Cell BE is a heterogeneous multicore processor collaboratively developed by IBM, Sony and Toshiba [64]. Figure 1.2 provides an architecture overview of this architecture. There are nine processing elements with one PowerPC Engine (PPE) and eight Synergistic Processing Engines (SPEs) [33]. PPE in the Cell BE is a 64-bit dual issue, dual threaded, in-order processor. It works as a control plane that launches tasks on SPEs. Eight SPEs run as high performance data processing planes. Each SPE has a 128×128 bit register file and supports single instruction multiple data (SIMD) operations. Each SPE also hosts an SPM of 256 KB that is formally referred as the SPE local store. A four-ring structured element interconnect bus (EIB) [44] connects the PPE, eight SPEs, and the memory controllers, providing a cumulative bandwidth of over 204.8 GBps. Direct memory access or DMA

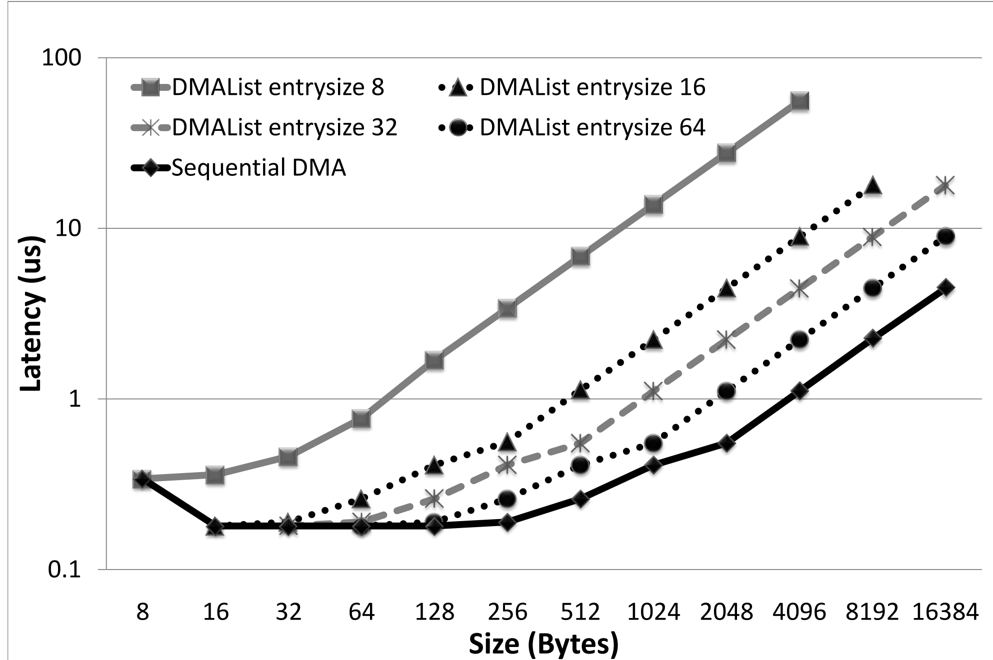


Figure 1.3: DMA performance with 6 SPEs.

(which can be launched by either the PPE or an SPE) is the primary mechanism for communicating between the local store of two SPEs or between an SPE local store and the off-chip PPE main memory[2]. The non-blocking nature of a DMA engine permits the amortization of communication overhead by concurrent computation. In the IBM Cell BE architecture, up to sixteen independent DMAs can be launched simultaneously by each initiating core.

As the software controlled memory management is one of the key features of Cell BE, we characterize the performance of the DMA engine in Figure

[2]The CELL BE also supports signals and mail boxes for inter-SPE and PPE-SPE communication. However, these are primarily useful for synchronization for very small data items. As we are more concerned about large data items we focus on DMA.

1.3. The plot consists of two types of curves. The DMA list curves (4 in number) and the sequential DMA curve. DMA list as the name suggests denotes that a list of DMA requests was transferred with one command to the DMA engine. The DMA list curve of 8 bytes denotes that each DMA in the list was for 8 bytes. Similarly, the DMA list curve of 16, 32 and 64 denotes the size of each entry in the DMA list. The x-axis represents the total size of the DMA list request. The number of entries in a DMA list curve can be determined by dividing the x-axis index by the DMA list size of the curve. Finally, the sequential DMA curve denotes that a single DMA was initiated. Notice that both the x-axis and y-axis are on logarithmic scale with bases of 2 and 10, respectively.

1.3 Contributions

The contributions of this dissertation are summarized below:

- We design and implement an ATR algorithm [17] on the IBM Cell BE (Chapter 2). Eight optimizations that exploit both the specific algorithm constructs of the ATR algorithm and the architectural features of the Cell processor were implemented. The latency of the final Cell BE implementation is more than 25 times faster than the fully optimized PPE implementation and almost 20 times faster than our best efforts on a Pentium4 CPU. This initial manual design validates the computing power of the Cell BE. It also reveals the overheads and bottle-necks that are involved in developing stream applications for embedded architectures with SPMs.

- We propose ILP [19] and heuristic [18] approaches that schedule stream programs on a single core embedded processor that has an SPM with code overlay (Chapter 3). The three-stage ILP approach extensively explores the design alternatives with different schedules, code/data partitions, and actor to region/segment assignments. Experimental results demonstrate that our ILP approach is able to efficiently explore various design trade-offs and generate high quality solutions. Although the ILP approach generates high quality solutions, it could take a very long time to run due to the problem size. A fast heuristic algorithm that efficiently balances between a minimum buffer schedule and a minimum actor switching schedule, and solves the same problem with comparable results in a matter of seconds is also discussed.

- We present ILP and heuristic approaches for automatic compilation of stream programs onto embedded multicore processors that incorporate SPMs [20] (Chapter 4). In the ILP approach (CSMP ILP), fusion and fission operators are implemented by assigning actors to batches and then batches to PEs. The ILP formulation models both the code overlay and communication overheads under the constraint of limited on-chip SPM capacities and memory access delays. Experimental results show that CSMP ILP approach is able to effectively balance the computation and communication overheads when mapping stream programs onto multicore architectures. To overcome the long algorithm run time of CSMP ILP approach, we also provide a fast heuristic approach (CSMP heuristic) that solves the same problem with comparable results in a matter of seconds.

- CSMP ILP and heuristic approaches assume absence of feedback cycles in the program. The number of software pipelines stages being generated in the final schedule is uncontrollable. In Chapter 5, we propose a fast heuristic (RTEM) that schedules stream programs onto SPM based multicore processors through retiming [14]. Trade-offs between double buffering and code overlay are explored intensively in this approach. More importantly, the retiming approach inherently handles feedback cycles and it can accept a user specified upper bound on the number of software pipeline stages.

- When the number of PEs is very large and the existing parallelism in a stream program is comparably limited, RTEM heuristic fails to generate high quality solutions. In Chapter 6, we present unrolling and retiming of stream formats onto embedded multicore processors (URSEM) as our last optimization. URSEM preserves all the beneficial properties of RTEM heuristic and scales with the number of PEs. Apart from code overlays for addressing limited on-chip SPM capacities, code pre-fetching and data overlays are also introduced to address the increased code and data requirement caused due to unrolling.

Chapter 2

DESIGN OF AN ATR ALGORITHM ON IBM CELL BROADBAND ENGINE

This chapter presents the design and optimization of an ATR algorithm on the IBM Cell BE. The ATR algorithm and the Cell BE are good representatives of stream applications and domain specific multicore processors. ATR belongs to the important class of signal processing algorithms that are widely utilized in Radar and electronic surveillance systems. The Cell BE is aimed at streaming applications that exhibit limited run time or dynamic variation during execution. Stream applications permit aggressive static or design time optimizations for maximizing their performance. The Cell BE designers recognized this fact and incorporated a 256KB local store or scratch pad for each SPE instead of caches. The local store is shared for both code and data, that are fetched under software control through DMAs. Thus the well know problems of functional partitioning, load balancing, communication versus computation trade-offs that are encountered during parallelization of an application must now be addressed in the context of a software controlled memory hierarchy. It is this additional complexity that makes designing applications on the Cell BE a daunting task. This chapter presents eight optimizations that are also applicable to other applications and processors that demonstrate similar characteristics. The contributions of the chapter include:

- Design of an ATR algorithm on the Cell BE.
- A detailed discussion of four basic categories of optimizations and their effects.

- An optimized scheme for Frequency Domain Filtering (FDF) with symmetric kernels.
- A generic design flow for porting streaming applications onto domain specific multicore architectures.

We begin with an introduction to the ATR algorithm in 2.1. We discuss previous work on optimizing applications on the Cell BE in Section 2.2. In Section 2.3 we discuss the design and optimization of a reference implementation of the ATR algorithm on an Intel Pentium4 based PC platform along with detailed profile analysis. The design flow of porting the ATR algorithm to the Cell BE along with various optimizations are presented in Section 2.4. We analyze the experimental results in Section 2.5 and summarizes in Section 2.6.

2.1 Automatic Target Recognition

Automatic target recognition (ATR) algorithms belong to the class of high performance computation intensive image processing algorithms that are widely used in applications such as target detection, radar processing, and pattern recognition. In this chapter, we present an optimized implementation of the detection algorithm proposed by David Casasent and Anqi Ye [17]. Their ATR algorithm is a fusion of several detection algorithms. A high-level flow of the algorithm is given in Figure 2.1 and the short descriptions for each stage are provided in Table 2.1. In Figure 2.1, the morphological filter detects both the objects and the edges from an input image. The wavelet transform filter that can execute in parallel with the morphological filter extracts the edges. A weighted subtraction therefore gives us only the detected objects. The

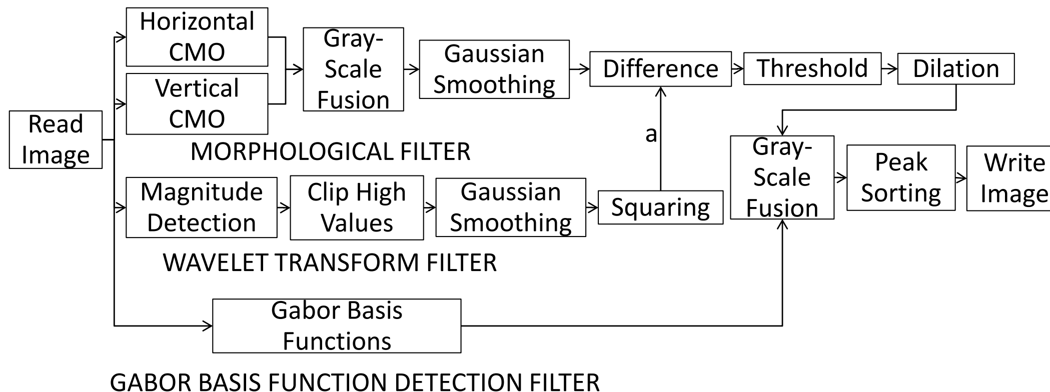


Figure 2.1: ATR algorithm overview.

following threshold operator removes the background noises and the dilation operator recovers the shape of the objects. A third branch in the algorithm, namely the Gabor basis functions based detection filter utilizes a pre-trained kernel to do a frequency domain filtering (FDF) and outputs the detected objects. A grayscale fusion of the two branches reduces the false alarm rate P_{FM} while trying to maintain the object detection rate P_D. In the final stage, the peak sorting operator further erases potential false detections.

2.2 Previous Work

ATR [9][10] are high performance signal processing applications that perform automatic target acquisition, identification, and tracking. Due to their high performance requirements, ATR algorithms have been traditionally implemented on reconfigurable logic based systems [67][23]. Researchers have also explored specialized support vector machine implementations for ATR [78]. The recent advent of commercially available specialized multi-core processor architectures offer exciting new platforms for implementation of ATR. However

Table 2.1: Stages in the ATR allgorithm.

Dilation Erosion	$D_i = MAX_{i-K \leq j \leq i+K} A_j$ $E_i = MIN_{i-K \leq j \leq i+K} A_j$
Closing Opening	Dilation - Erosion Erosion - Dilation
HCMO VCMO	Horizontal Closing - Opening Vertical Closing - Opening
Gray Scale Fusion	$G_i = MIN(A_i, B_i)$
Gaussian Smoothing	$GS(A) = IFFT(B)$ $B = FFT(A) * FFT(K)$ K is a Gaussian Kernel
Difference	$D_i = A_i - B_i$
Threshold	$T_i = 0, \quad if\ A_i < T$ $A_i,\ if\ A_i > T$
Gabor Wavelet	$G_i = \sum_{j=-S}^{S} A_{i+j} * K_{i+j}$ K is a Gabor Kernel
Magnitude	$M_i = A_i, \quad if\ \|A_i\| > T$ $0,\ otherwise$
Clip High Values	$C_i = 0, \quad if\ A_i > T$ $A_i,\ if\ A_i < T$
Squaring	$S_i = A_i^2$
2-D Dilation	Horizontal dilation followed by vertical dilation
Gabor Basis Functions	$GB(A) = IFFT(B)$ $B = FFT(A) * X_K$ $X_K = \sum \alpha_i * FFT(K_i)$ K_i is a Gabor Basis Kernel (GBK) α_i is weight of GBK
Peak Sorting	Iteratively do: 1 detect object 2 delete object 3 delete its neighborhood

to the best of our knowledge, we are not aware of any existing implementation of ATR on such processor architectures.

In recent past, researchers have designed several optimized implementations of stream applications on the Cell BE. In the following we discuss a few representative implementations. Kato et al. [42] implemented a real time dig-

ital media application on the Cell BE. Petrini et al. [63] ported the Sweep3D application on the Cell BE. More recently, Baker et al. [4] designed a scalable implementation of the H.264 decoder on the Cell BE. The design experience and optimizations utilized by existing research are relevant for developing any application on the Cell BE. This chapter proposes optimizations that are particularly aimed at exploiting the unique algorithmic characteristics of the ATR application.

Researchers have begun to recognize the daunting challenge of developing applications on the specialized multi-core processor architectures such as the Cell BE. Eichenberger et al. [27] [28] proposed compiler techniques such as memory alignment, branch prediction, SIMDization, thread level parallelism, and data management for alleviating the task of programming the Cell BE. Maeda et al. [53] proposed a multi-layered programming model and a real time resource scheduler. Fatahalian et al. [30] presented a programming language, Sequoia, to facilitate the management of memory hierarchy when developing program with multi-level parallelism. Despite all the efforts that are involved, automated techniques for obtaining optimized parallel implementations on the Cell BE are still in their infancy and not close to commercial deployment.

2.3 Reference Implementation on Pentium4 PC

We first implemented the ATR algorithm on our host PC with a $3.2GHz$ Intel Pentium4 CPU. The original implementation demonstrated a latency of 3.26 s for a 512×512 pixel image with 6 distinct objects (varying from 2×4 to 10×4 pixel size) that were detected. Column 2 of Table 2.2 gives a break down of the run time for various stages of the algorithm. As we can see

from the table, the three frequency domain filtering (FDF) modules namely CMO Gaussian, Wavelet Gaussian and Gabor Basis Functions dominate the run time as each of them takes approximately 900 ms to complete. FDF modules contain one 2-dimensional Fast Fourier Transform (FFT), one point-wise matrix multiplication, one 2-dimensional inverse FFT (IFFT), and some miscellaneous functions for type casting and image normalization. Hence, we optimized the code for FFT and IFFT first. In particular, we applied two optimizations. The original FFT implementation accepted complex inputs. However, the image data only has real coefficients with imaginary parts as zero. Further, the FFT calculation has a symmetric kernel for real and imaginary parts of the data. Thus, we can pack two data points into one complex data, and reduce the number of calculations by a factor of 2. Additional data re-organization are required to recover the correct result. The improvements due to this optimization are depicted in Column 3 with gray shaded cells. We applied similar optimizations for IFFT. As our second optimization, we pre-computed the FFT and IFFT twiddle factors and accessed them by table look-up. The improvement due to this optimization over the previous step is depicted in Column 4 with gray cells. Figure 2.2 depicts the overall approach. After the application of these two optimizations the run time of the ATR algorithm reduced to 1.33 s.

2.4 Design of ATR on the IBM Cell BE

We utilized the Sony Playstation3 (PS3) with Fedora 7 as the target platform. The Cell BE has 8 SPEs. However, in PS3 only 6 of them are available to the programmer. In this section we discuss the various optimizations that

Table 2.2: Profile Analysis of ATR on host PC and PPE.

Functions	Run Time (ms)						
	PC			PPE			
	Original	FDF Real	FDF Coeff.	Original	VMX	FDF Real	FDF Coeff.
Read Image	1.38	1.38	1.38	1.06	0.86	0.86	0.86
HCMO	207.52	207.52	207.52	106.71	106.71	106.71	106.71
VCMO	144.27	144.27	144.27	132.65	132.65	132.65	132.65
CMO Fusion	3.97	3.97	3.97	3.82	1.90	1.90	1.90
CMO Gau.	907.9	517.24	267.9	3964.30	1992.68	1072.45	412.54
Gabor Wavelet	105.05	105.5	105.5	185.66	161.51	161.51	161.51
Detect& Clip	1.13	1.13	1.13	1.80	1.80	1.80	1.80
Wavelet Gau.	904.47	514.16	264.47	3962.79	2000.84	1070.81	410.67
Squaring	2.58	2.58	2.58	10.56	5.78	5.78	5.78
Difference	16.8	16.8	16.8	16.59	6.45	6.45	6.45
Threshold	1.04	1.04	1.04	3.91	3.91	3.91	3.91
Dilation	41.37	41.37	41.37	27.69	27.69	27.69	27.69
GBF	906.80	506.65	256.87	4052.68	1997.55	1087.74	407.23
Fusion	2.17	2.17	2.17	1.90	1.90	1.90	1.90
Peak Sorting	13.01	13.01	13.01	6.25	6.25	6.25	6.25
Write Image	0.55	0.55	0.55	3.68	2.41	2.41	2.41
Total Run Time	3260.01	2079.70	1330.01	12482.05	6450.89	3690.82	1690.26

*The cell with gray background indicates that the function run time is reduced by the current optimization.

were applied to achieve a high performance implementation of the ATR on the Cell BE. There are a total of eight optimizations that are categorized into 4 basic classes: data parallelism (application mapping), computation acceleration (FFT, SIMD), communication acceleration (matrix transpose, pre-touch memory, mailbox), and communication overhead amortization (double buffering, small function relocation). The various optimizations exploit both the algorithmic characteristics of the ATR application and specific architectural features of the Cell BE. As a first step toward porting ATR onto the Cell BE, we obtained an implementation solely on the PPE (denoted by PPE-Ori).

2.4.1 Porting ATR to PPE

Column 5 of Table 2.2 gives the performance of the original implementation on the PPE which was 12.48 s. This latency is much higher than the original

PC implementation as the PPE has a relatively simpler architecture that does not support out of order execution. We next compiled the program such that the Vector Multimedia Extensions (VMX) on the PPE were utilized (Column 6 of Table 2.2). Finally, FDF specific optimizations were applied (Columns 7 and 8 of Table 2.2). The final run time of the implementation was 1.69 s.

2.4.2 Parallelized implementation of ATR

In the following subsections we discuss the 8 optimizations that were applied for obtaining a parallelized implementation of ATR on the Cell BE. In the following the various optimizations are discussed in the order in which they were applied in the case study. Table 2.3 depicts the performance improvements that were obtained by application of each optimization.

2.4.3 Application parallelization on 6 SPEs

We exploited the inherent data parallelism in the ATR algorithm for mapping it to 6 SPEs. The 512×512 input image was divided into 128 block each of size 4×512 for the morphological operations. Figure 2.3 illustrates this data partition scheme. Similarly for the FDF, the input image was divided into 512 blocks with the size of 1×512 for each block. Note that in both the partition schemes, either 4×512 (type of unsigned char) or 1×512 (type of float), the size of each data piece is 2KB. This data size was selected in favor of the DMA/DMAList performance as discussed in Section 1.2. The latency of the first parallel implementation was 0.797 s (Column 2 of Table 2.3). Column 2 of Table 2.4 gives break-up of the communication and computation overheads for a few critical functions.

2.4.4 Double buffering

Double buffering as the name suggests doubles the amount of memory utilized for communication between PPE and SPE. Essentially, while the SPE may be operating upon data in buffer 0, a DMA operation may be loading new data in buffer 1 from PPE. Similarly while the SPE may be writing to buffer 0, a DMA may be transferring data from buffer 1 to PPE. Double buffering enables amortization of communication overheads by overlapping it with computation. Double buffering is a well known mechanism for data memory management in software controlled memory systems. Double buffering can also be considered as pipelining of data reading, computation and data writing stages. Similar to pipelining, it also has a "warm up" and "drain out" time which should be ideally as small as possible. However, we also need to take the DMA/DMAList performance into account (Section 1.2). Consequently, a granularity of 2KB was selected to implement our double buffering scheme.

Column 3 in Table 2.3 gives the improvement achieved due to double buffering. As the results indicate double buffering only gave us incremental improvement to 0.769 s. Column 3 of Table 2.4 shows the same trend. This poor improvement was later diagnosed to high translation look aside buffer (TLB) and page table entry (PTE) misses overhead. Once these misses were eliminated the overall performance showed a significant improvement. Further details that addresses pre-touching of memory are discussed in Section 2.4.9.

2.4.5 2-D FFT

We utilized the 1-D FFT kernel fft_1d_r2 from the IBM SDK 3.0 to compute the 2-D FFT on the SPE. Figure 2.4 plots the performance of the fft_1d_r2 kernel. As it is written in assembly, it demonstrates a performance of 5.62 μs for a 1024 point FFT. However, it also posed several difficulties when it was utilized to implement a 2-D FFT. First, To compute 2-D FFT on the whole image, two sequential executions of the fft_1d_r2 kernel and 2 matrix transposes are required. Figure 2.5, Algorithm 1 line 1 $\sim$ line 4 illustrates this basic scheme. Second, fft_1d_r2 only takes complex input with each element represented by a pair of $< real, imag >$, indicating that additional data re-formulation is required. In our implementation, we utilized the spu_shuffle intrinsic to efficiently re-format the data. We also managed to remove two matrix transposes from the FDF by modifying the flow as illustrated in Figure 2.5, Algorithm 2. The optimizations reduced the run time of each FDF module by almost half. The profile analysis is shown in Column 4 of Table 2.3. Column 3 of Table 2.4 gives the corresponding computation time and communication time. Since the FDF modules only work on half the data compared to the previous step, the data communication time also showed a large reduction. Consequently, we achieved an overall 34% performance improvement at this step.

2.4.6 Matrix transpose

There are two matrix-transpose operations in each FDF module. We first utilized DMAList to fetch 128 bit data for each column, that is 512x4 $floats$, and then transpose them back as 2 rows. Figure 2.6 (A) gives an overview

of this scheme. The shape change is due to the representation of a complex data point. However, this simple scheme has a very big problem: for each single DMA inside the DMAList, the data size is only 128 bit, and 7 out of 8 of them are not 128 Bytes aligned. Both these aspects greatly impacted the performance [45]. The resulting matrix-transposes were so expensive that they took more than 40% of the total run time. Observing that the bottleneck comes from the DMAList operations, our second implementation gathers 32x32 floats from the PPE main memory, transposes them, and puts them back as 16x64 floats, as illustrated in Figure 2.6 (B). In the new approach, the data communication for the FDF module dropped down by more than half as shown in Column 5 of Table 2.4. The overall performance was improved by 35% as reported on Column 5 of Table 2.3.

2.4.7 Small function relocation

In this step we analyzed the computation granularity of non-performance intensive functions. Specifically, functions whose execution on the PPE could be hidden by other functions executing in parallel on SPE were assigned to PPE. Further, if the data transfer time for some functions executing on SPE was greater than their run time on the PPE, they were also assigned to the PPE. Figure 2.7 shows the final parallel schedule for execution of the application on the Cell BE. A module that is labeled as SPE in the figure executes in parallel across the 6 SPEs. A module that is not labeled executes on the PPE. Further, the "syn" module denotes barrier synchronization. Figure 2.8 gives the schedule for execution of high level modules (namely CMO Gaussian smoothing, Wavelet Gaussian smoothing, Gabor basis function detection). Column 6 of Table 2.3 illustrates the impact of this optimization. The cells marked with '-'

denote that after small function relocation, these functions were either hidden by the execution of other functions, or they were combined to the end of the FDF modules.

2.4.8 SIMD & loop unrolling

There are two approaches that can be applied for code vectorization on the Cell BE, namely, manual vectorization or the IBM xlc compiler. The xlc compiler reduces the effort required for vectorization. However, in our experience the compiler was not found to be very effective as manual optimizations out performed the compiler generated code. Consequently, functions that were computationally intensive were vertorized/unrolled manually. On the other hand, less computationally intensive functions were optimized by the xlc compiler in our implementation. Column 7 of Table 2.3 and Column 6 of Table 2.4 detail the effects of this optimization.

2.4.9 Pretouch memory

There are two basic categories of memory hierarchy misses occurring in our code, namely TLB (Translation Lookaside Buffer) misses, and the PTE (Page Table Entry) misses. The TLB misses always take place when the SPEs first get started. We reference this delay by TLB warm-up time. As soon as the TLB table gets filled up, this delay disappears. The TLB warm-up time is predictable, and more or less stable. This delay in our implementation is hidden by the computation of "HCMO" and "Gabor Wavelet" through double buffering.

Besides TLB warm-up time, normal TLB misses also occur in the program. Normal TLB misses are unpredictable and thus hard to eliminate. Pre-

touching the memory can substantially reduce the overhead. Pre-touching implies that the memory is accessed but not operated upon. Pre-touching by the PPE ensures that the required page is in the main memory. Pre-touching by SPE ensure that the TLB entry is upto date. Pre-touching is scheduled to occur earlier than the execution of the DMA/DMAList transfers that actually access the page.

PTE misses are handed over to the operating system as an interrupt. Therefore, the processing time for PTE misses are much higher than TLB misses. This overhead could be thousands of micro-seconds compared to several micro-seconds for TLB misses. We discovered that PTE misses happen when a certain memory location is referenced by the SPEs for the first time before any PPE functions touch the same memory. Therefore, we managed to eliminate PTE misses by pre-touching the memory with PPE. Those operations were able to bring the run time down to 0.087 seconds. Examining the computation and communication time of each function in Table 2.4, we obverse that after this step, the communication time became smaller than the computation time for most of the functions mapped to the SPEs. The full impact of double buffering was realized after this optimization.

2.4.10 Mailbox communication

Mailbox communication is utilized to implement all synchronizations in our design. Experiments on the Cell processor showed that the classic mailbox implementation could take thousands of micro-seconds in the worst case. This unpredictable behavior resulted in degraded performance of our code. We examined several alternative implementation strategies including signaling, problem state area mapped mailbox, and DMA. The problem state area mapped

mailbox implementation turned out to be the best solution with more than 5×10^5 rounds of synchronization per second between the PPE and 6 SPEs. A detailed profile of problem state area mapped mailbox performance is shown in Table 2.5. After the application of this final optimization, the run time of our program dropped down to less than 70 milliseconds as detailed in Column 9 of Table 2.3.

2.5 Experimental Results

In our final implementation, the code and data adds up to 131.625 KB for each SPE, which indicates that we have more than 120 KB of local store memory left for the stack and heap. Figure 2.9 summarizes the performance impacts of various optimizations on Pentium4 based PC, on PPE only, and on Cell processor across one PPE and 6 SPEs. The final run time coupled with the computation/communication time for each process is shown in Figure 2.10. As we can see from the figure, HCMO, VCMO and 2-D Dilation take up to 40% of the total run time in the final implementation. There are two reasons. First, the Dilation and Erosion operators that serve as the basic modules for the CMO and 2-D Dilation processes are sequential operations. Consequently, SIMDization has limited impact. Second, the Dilation and Erosion consist of conditional branches, which cannot be handled efficiently on SPEs that are primarily aimed at stream applications.

2.6 Summary

We presented a detailed case study of designing an ATR algorithm on the Cell BE in this chapter. Various optimizations that exploited both the unique features of the application as well as the target architecture were presented. The optimizations are applicable to other algorithms and architectures that have similar features as ATR and the Cell BE. The final implementation shows a latency of 0.07 s on a PS3 platform with 6 SPEs. The optimized performance was almost 20 times faster than our best efforts on a Pentium4 CPU. The achieved performance validates both the impact of our optimizations and the processing capabilities of the Cell BE.

Algorithm: 2 Dimensional Real FFT

Row FFT
/*ROW_{ij} denotes the i_{th} row, j_{th} element in the image, R and I are one dimensional arrays that store the real and imag inputs of IDFFT*/
1 for i from 0 to 256
2 for j from 0 to 511
3 $R_j = ROW_{2i,j}$, $I_j = ROW_{2i+1,j}$
4 endfor
/*R and I are input arrays of 1DFFT, tempR and tempI are output arrays*/
5 1DFFT$(R, I, 512, tmpR, tmpI)$
6 $R_{512} = R_0$, $I_{512} = I_0$
/*R' and I', 2 dimensional arrays that store the results of 1DFFTs*/
7 for j from 0 to 511
8 $R'_{2i,j} = tmpR_j + tmpR_{512-j}$, $R'_{2i+1,j} = tmpI_j + tmpI_{512-j}$
9 $I'_{2i,j} = tmpI_j - tmpI_{512-j}$, $I'_{2i+1,j} = tmpR_{512-j} - tmpR_j$
10 endfor
11 endfor
12 Matrix Transpose
Col FFT
13 for i from 0 to 256
14 for j from 0 to 511
15 $R_j = R'_{i,j}$, $I_j = I'_{i,j}$
16 endfor
17 1DFFT$(R, I, 512, tempR, tempI)$
18 for j from 0 to 511
19 $R'_{i,j} = tempR_j$, $I'_{i,j} = -tempI_j$
20 if $i \neq 0$ and $i \neq 256$ /*recover the other half of the image*/
21 $R'_{512-i,512-(j+1)} = tempR_j$
22 $I'_{512-i,512-(j+1)} = -tempI_j$
23 endif
24 endfor
25 endfor

Figure 2.2: 2-Dimensional real FFT.

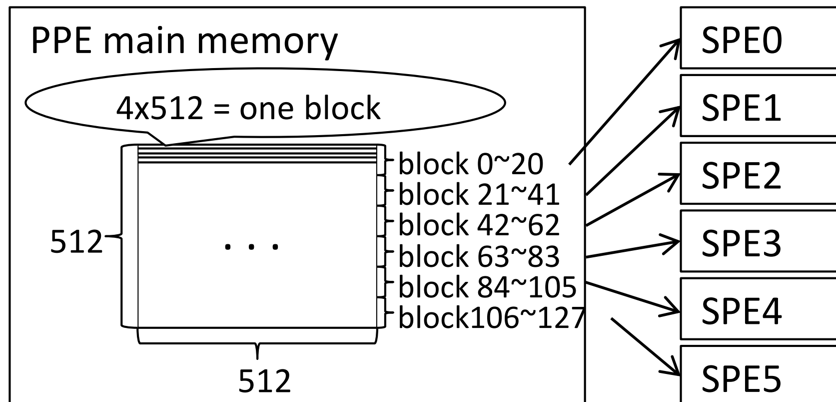


Figure 2.3: Data partition for the morphological filter

Table 2.3: Profile Analysis of ATR on PPE and SPEs.

Functions	Run Time (ms)							
	SPE Orig.	SPE Double Buffering	SPE Real & Coef.	SPE Matrix Trans.	SPE Small Func.	SPE Simd	S/PPE PreTouch Memory	S/PPE Mailbox
Read Image	0.86	0.86	0.86	0.86	0.86	0.86	0.86	0.86
HCMO	44.72	36.65	36.65	36.65	32.26	32.26	17.75	13.72
VCMO	29.49	22.37	22.37	22.37	21.52	21.52	14.18	13.20
CMO Fusion	1.90	1.90	1.90	1.90	1.90	1.90	0.61	0.61
CMO Gau.	219.65	214.23	133.25	68.35	65.72	62.87	7.28	4.23
Gabor Wavelet	8.29	8.29	8.29	8.29	4.15	4.15	4.15	3.75
Detect & Clip	1.80	1.80	1.80	1.80	-	-	-	-
Wavelet Gau.	227.21	221.10	124.73	63.21	57.38	57.24	7.26	3.99
Squaring	5.78	5.78	5.78	5.78	-	-	-	-
Difference	6.45	6.45	6.45	6.45	-	-	-	-
Threshold	3.91	3.91	3.91	3.91	-	-	-	-
Dilation	45.32	31.43	31.43	31.43	31.43	31.43	18.49	17.03
GBF	190.82	204.22	121.65	72.16	49.84	46.99	7.91	3.78
Fusion	1.90	1.90	1.90	1.90	-	-	-	-
Peak Sorting	6.25	6.25	6.25	6.25	6.25	6.25	6.25	6.25
Write Image	2.41	2.41	2.41	2.41	2.41	2.41	2.41	2.41
Total Time	796.76	768.55	509.63	333.72	279.80	267.88	87.15	69.83

*The cells colored with gray indicate that the function is affected by the current optimization. The cells marked with '-' indicates the function execution is hidden by the execution of other functions, or padded to the end of the FDF modules.

Table 2.4: Computation/Communication Latency Comparison.

Functions	Computation/Communication Time (ms)					
	SPE Original	SPE Double Buffering	SPE Real & Coef.	SPE Matrix Small Func	SPE SIMD	S/PPE Pretouch Mailbox
HCMO	12.77/20.14	12.77/17.23	12.77/17.23	12.77/17.23	12.77/17.23	12.77/0.24
VCMO	11.84/8.90	11.84/6.60	11.84/6.60	11.84/6.60	11.84/6.60	11.84/0.16
CMO Gau.	18.31/165.05	18.31/161.05	5.12/117.50	5.74/54.72	3.12/54.72	3.12/1.75
Wavelet Gau.	18.20/161.75	18.20/176.3	4.46/101.3	4.79/49.57	3.04/49.57	3.04/1.69
Dilation	16.52/18.30	16.52/12.13	16.52/12.13	16.52/12.13	16.52/12.13	16.52/0.33
GBF	18.46/149.54	18.46/154.49	5.25/97.70	5.41/36.54	3.18/36.54	3.18/1.62

*The cells colored with gray indicates that either the run time or the communication time of the function were effected by the current optimization.

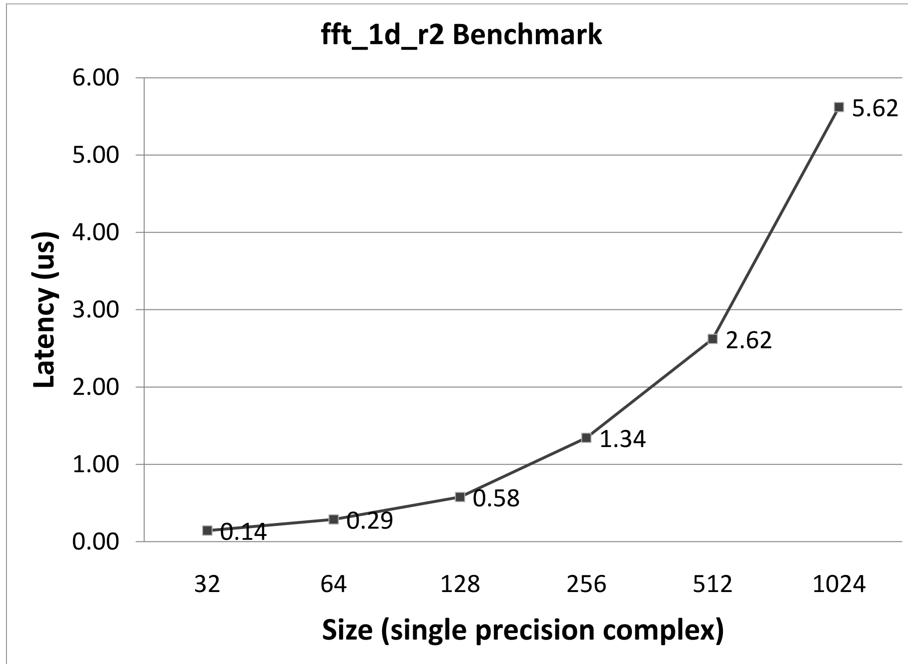


Figure 2.4: fft_1d_r2 profile on one SPE.

FDF Algorithm 1	FDF Algorithm 2
1 Row 1DFFTs	1 Row 1DFFTs
2 Image Transpose	2 Image Transpose
3 Column 1DFFTs	3 Column 1DFFTs
4 Image Transpose	4 Multiply Transposed Kernel
5 Multiply Kernel	5 Column 1DIFFTs
6 Row 1DIFFTs	6 Image Transpose
7 Image Transpose	7 Row 1D IFFTs
8 Column 1DIFFTs	
9 Image Transpose	

Figure 2.5: Unoptimized and Optimized FDF.

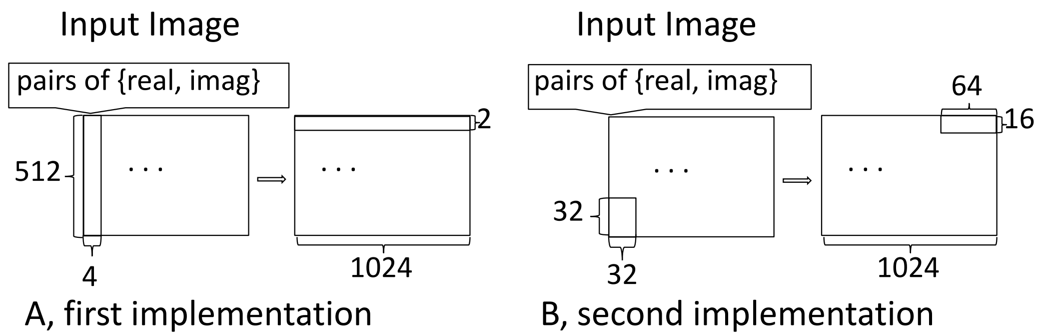


Figure 2.6: Data partition in matrix transpose module.

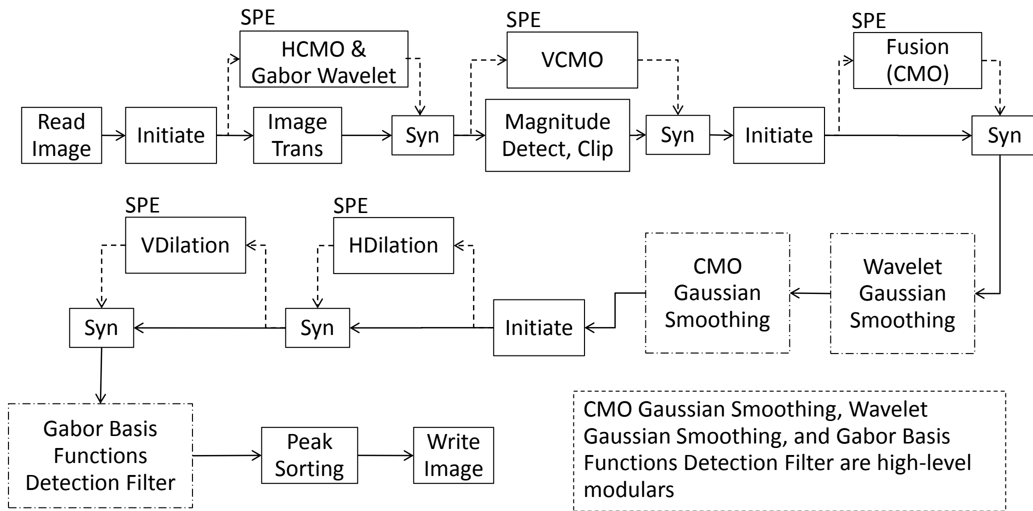
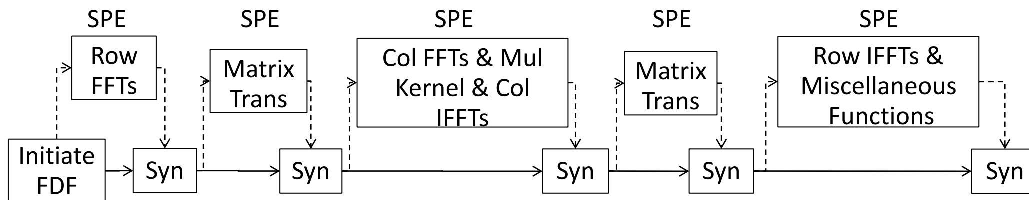


Figure 2.7: Task schedule. The modules annotated with SPE execute in parallel on 6 SPEs.



*Miscellaneous functions include: Difference, Threshold (CMO); Squaring (Wavelet); and Fusion (Gabor Basis).

Figure 2.8: Frequency Domain Filtering.

Table 2.5: Problem State Area Mapped Mailbox Performance.

Num. Of SPEs	Avg. Roundtrip Time (us)	Roundtrips Per Second
1	0.54	$18.52\text{x}10^5$
2	0.77	$12.99\text{x}10^5$
3	1.04	$9.62\text{x}10^5$
4	1.46	$6.85\text{x}10^5$
5	1.74	$5.75\text{x}10^5$
6	1.87	$5.35\text{x}10^5$

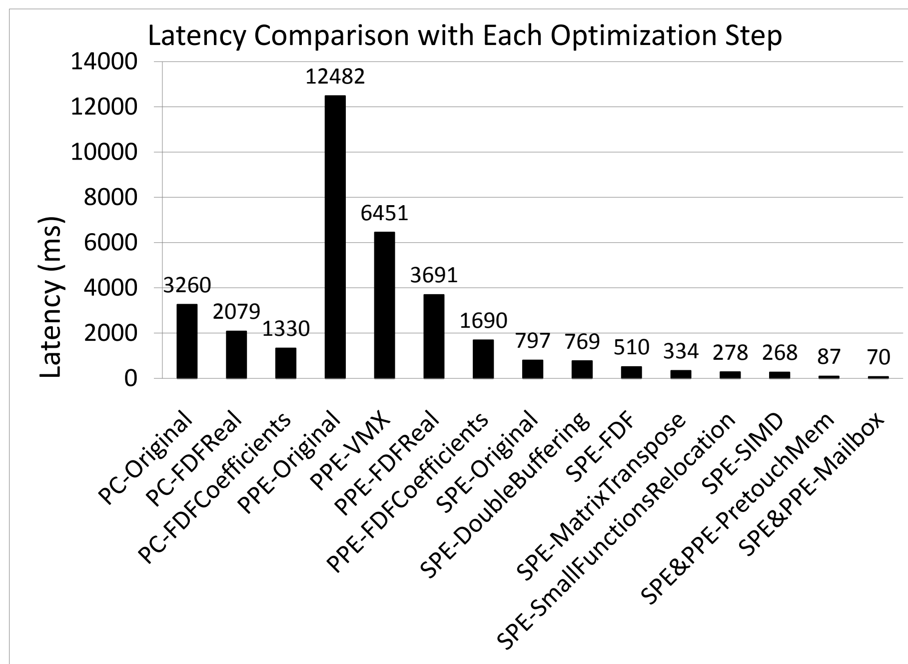


Figure 2.9: Latency comparison for integrating each optimization.

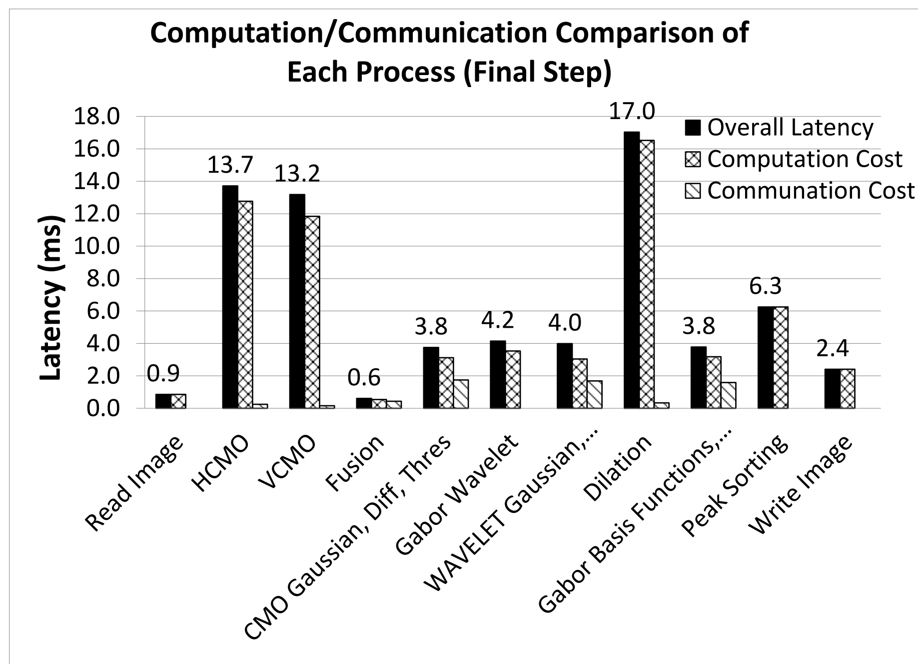


Figure 2.10: Computation Comparison for Each Process in ATR.

Chapter 3

SCHEDULING OF STREAM PROGRAMS ON ONE EMBEDDED CORE WITH CODE OVERLAY

In a typical SPM enhanced processor, an on-chip SPM is usually very small (in a matter of KBytes) and is used to host both actor code and data buffer. Scheduling of stream programs on such an architecture involves division of the limited on-chip SPM between actor code and data buffer, and execution of actors in such a manner that the physical SPM is time shared among different actor codes (formally referenced as code overlay). A traditional minimum buffer schedule could result in a very high code overlay overhead and therefore may not be optimal. To derive an efficient schedule with low code overlay overhead, it's necessary to partition the available code memory into regions such that actors mapped to the same region will also share the same physical memory. Further, code and data transfers between an on-chip SPM and its remote main memory is realized through DMA engine. To amortize the DMA communication overhead, actor codes need to be grouped into segments to reduce the actual number of DMA transfers. In this chapter we propose a three-stage ILP formulation and a fast heuristic for scheduling stream programs on SPM enhanced processors with the objective of overall latency minimization. We incorporated code pre-fetching into our ILP and heuristic approaches to improve on performance. Further, deep pre-fetching and data overlay optimizations were also investigated in our heuristic approach. The efficiency of our approaches was evaluated by compiling ten stream applications onto one SPE of an IBM Cell BE. Comparison between our approaches and a minimum buffer scheduling approach is discussed in the experimental results section.

The contributions of this chapter include:

- A 3-stage ILP formulation that extensively explores design alternatives with different schedules, code and data partitions, actor to region and segment assignments with the objective of latency minimization.
- Extension to our 3-stage ILP that incorporates a basic pre-fetching optimization to further reduce the code overlay overhead.
- An efficient heuristic approach for the same problem that is able to achieve comparable results as the 3-stage ILP approach with much faster algorithm run time.
- Extension to our heuristic approach that incorporates basic pre-fetching, deep pre-fetching, and data overlay.

In the next section we motivate our problem by discussing various design trade-offs. Section 3.3 of this chapter formulates the problem. Section 3.4 investigates related work. Section 3.5 presents our 3-stage ILP approach. An extension to the 3-stage ILP approach is presented in Section 3.6. Section 3.7 discusses the implementation of our heuristic approach. Extensions with basic pre-fetching, deep pre-fetching, and data overlay are provided in Section 3.8. Finally, Section 3.9 presents our experimental results and Section 3.10 summarizes this chapter.

3.1 Preliminaries

We first introduce some basic concepts that will be used throughout this chapter, including code overlay, basic pre-fetching, deep pre-fetching, and data

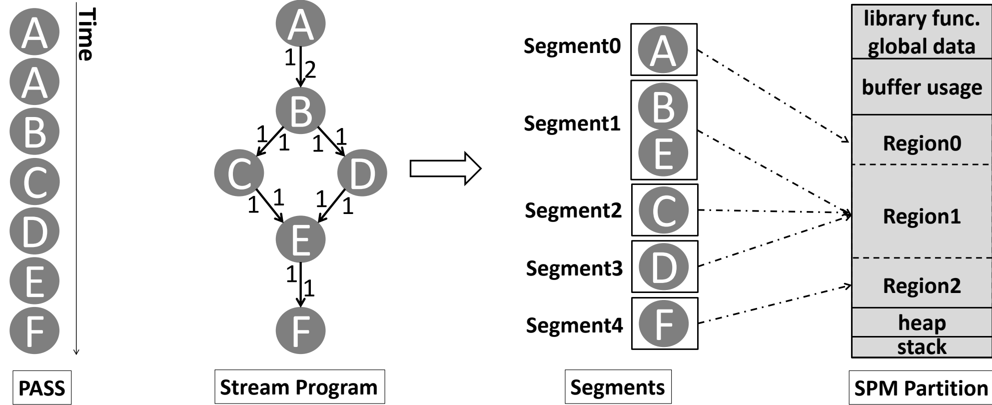


Figure 3.1: Segment-region code overlay overview.

overlay.

3.1.1 Code overlay

We utilize the region-segment code overlay scheme that is supported by spu-gcc version 4.1.1 for executing StreamIt code on an SPE. In the region-segment scheme, as illustrated in Figure 3.1, The following three stages were performed to schedule a stream program/an SDF[1] onto an SPM,

- PASS generation - A Periodical Admissible Sequential Schedule (PASS) is defined on an SDF graph as a finite sequence of actor firings that brings buffers back to their initial state. In our problem instances, a PASS is also a valid steady-state schedule of the stream program. In the PASS generation stage, a PASS for the given stream program is generated and then its buffer usage is calculated. The available code memory thus is given by the difference of the SPM size and the buffer usage[2].

[1] A stream program can be described by an SDF as discussed in Section 1.1.

[2] We assume that the memory of library functions, global data, stack and heap has already been subtracted from the SPM.

- Actor to region assignments - In this stage, we assign actors to regions such that each actor is mapped to one and only one region and the sum of all region sizes is no more than the available code memory. The size of each region is given by the largest actor size assigned to it. During the program execution, actors assigned to the same region are overlayed with each other in the same physical location.

- Segmentation - A segment is a group of actors that are moved to the SPM altogether. In the segmentation stage, we selectively group actors in the same region into segments to amortize DMA base cost. Each segment size is given by the sum of all actor code sizes mapped to it. After segmentation, a segment will be the smallest granularity for any code transfer. To respect the memory constraint, each segment size must be no more than its region size.

In region-segment overlay scheme, an instance of the actor is assigned to exactly one segment and each segment is assigned to one region[3]. At any time period, there can be only one segment present in any region. The regions essentially represent the memory partition for code. A code overlay overhead is encountered when an actor to be executed is not present in the on-chip SPM. Under such a scenario, the overlay manager will have to fetch the segment from the main memory and a code overlay overhead is introduced.

3.1.2 Basic pre-fetching and deep pre-fetching

Since the DMA engine works independently from the execution unit of an embedded core, we overlap DMA transfers with actor executions. Figure 3.2

[3] An actor can have multiple instances in a PASS.

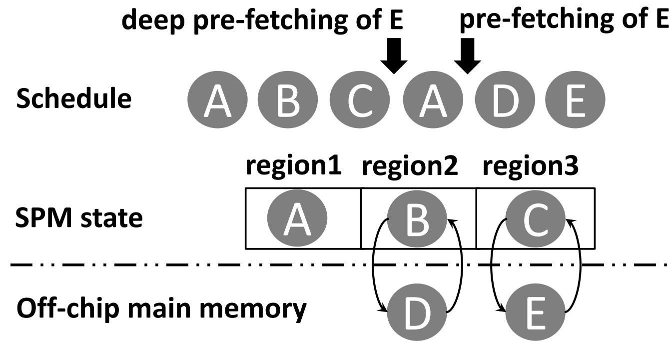


Figure 3.2: Code pre-fetching and deep pre-fetching.

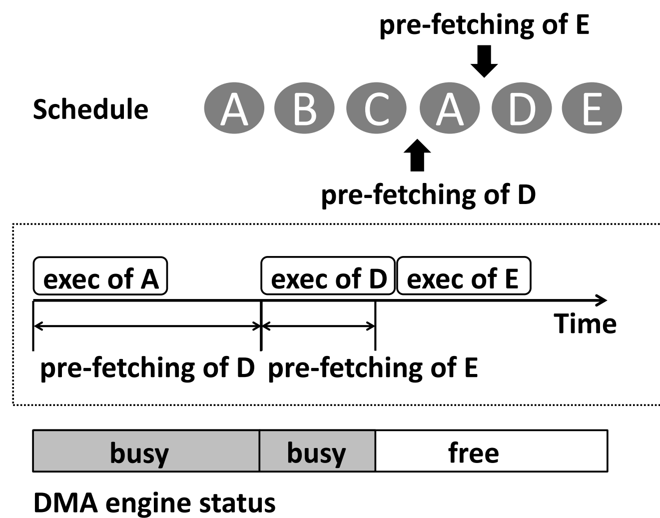


Figure 3.3: DMA engine status with basic pre-fetching.

introduces the behavior of a basic pre-fetching scheme and a deep pre-fetching scheme. In the example, we have an actor execution sequence of A, B, C, A, D, E. Actor A is mapped to region 1, actor B and D are mapped to region 2, and actor C and E are mapped to region 3. Suppose actor C just finishes execution, the current memory state of the SPM is A in region 1, B in region 2, and C in region 3 as described in Figure 3.2. Without code pre-fetching, we will execute A, overlay D, execute D, overlay E and execute E.

Let us focus on the code overlay of actor E. Without code pre-fetching, actor executions of A and D and code overlay of E are totally sequential. The code overlay overhead equals the DMA cost of actor E. In the basic pre-fetching scheme, if the current actor execution introduces additional code overlay overhead, we try to overlap it with the previous actor's execution. In the basic pre-fetching scheme, if the previous actor resides in a different region from the current actor, we issue pre-fetching. Otherwise, a basic pre-fetching cannot be issued because of memory conflict. In Figure 3.2, we will initiate the DMA transfer of E before execution of D with a basic pre-fetching.

The deep pre-fetching scheme extends the basic pre-fetching scheme by searching backward along the PASS and issuing a pre-fetching as early as possible. That is, immediately after the last execution of an actor from the same region. In Fig 3.2, we start from D, continue with A, and we stop at C since both C and E resides in region 3. Therefore with deep pre-fetching, we can start the pre-fetching of actor E right after actor C's execution.

The extension from the basic pre-fetching scheme to deep pre-fetching scheme seems reasonable and straight forward. However, there is another constraint that we need to consider. That is, we only have one DMA engine for each processing engine. In the basic pre-fetching scheme, this is not a problem since the DMA engine is guaranteed to be idle when we initiate the DMA pre-fetching of the next actor. This is because when we start the execution of the current actor, the previous pre-fetching has completed (one of the prerequisite that an actor can be executed) as introduced in Figure 3.3. However, this is not necessary true for deep pre-fetching. For example in Figure 3.4, we can initiate the code pre-fetching of E immediately after actor C' execution.

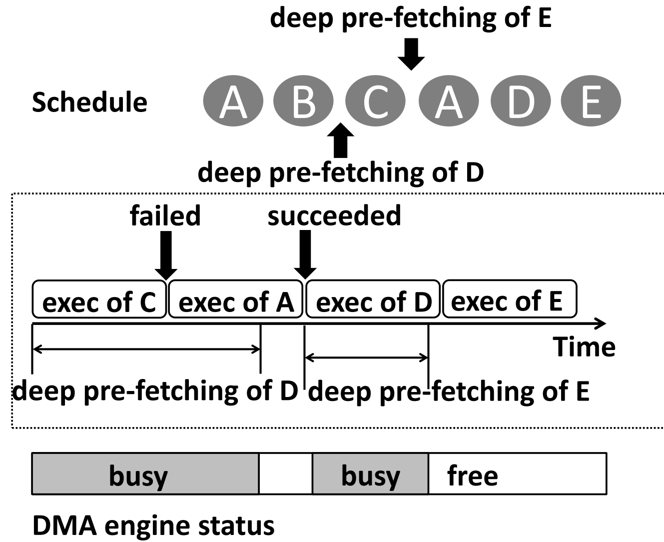


Figure 3.4: DMA engine status with deep pre-fetching.

However, at this time the DMA engine is still busy with pre-fetching of actor D's code. In this case, we utilize the largest DMA engine idle period available. In the example, actor D's execution period is utilized to overlap DMA transfer of actor E's code.

3.1.3 Data overlay

In order to reduce the buffer usage of a given schedule, we can also introduce data overlay. The basic idea is that we do not have to keep a data token for its entire life time in the local SPM. A data token can be transferred to the off-chip main memory as soon as it is produced and we get it back before it is being consumed. Figure 3.5 introduces this data overlay scheme. Blindly incorporating data overlay to every data token will introduce additional data overlay overhead as we have to circle every data token through the off-chip main memory. Further, data overlay also uses the same DMA engine that is used by code pre-fetching. Unnecessarily introducing data overlay will also

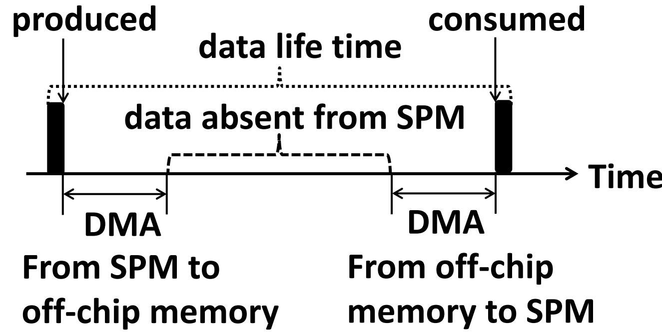


Figure 3.5: Data overlay overview.

impact code pre-fetching optimization. Finally, as the time to execute each actor is fairly constant and all internal data are stored locally during the entire program execution in our problem instances, the problem of minimizing the latency of executing a stream program on an SPM based architecture reduces to minimizing the code overlay and data overlay overheads introduced by the on-chip SPM's limited capacity.

3.2 Design Trade-offs

In this section we discuss the design trade-offs in each stage of scheduling a stream application on an SPM enhanced processor, including PASS generation, actor to region assignments, segmentation, and data overlay.

3.2.1 PASS generation

Two properties of a PASS are particular important in terms of scheduling SDF models onto an limited size SPM,

- Buffer usage - buffer usage of a PASS is the total memory required for storing the internal data buffer during its entire execution. The

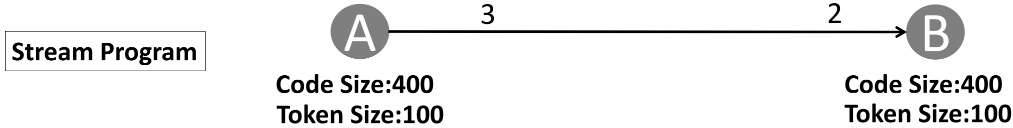


Figure 3.6: Stream program with one producer and one consumer. The code, token sizes are in bytes.

same buffer can be re-used at different time intervals as long as the correct program behaviour is maintained. In our problem, a given SPM is partitioned into buffer usage and code memory. The smaller the buffer usage, the more memory we can devote to program code.

- Actor switches - actor switches of a PASS is captured by the number of actor executions diverging from one actor to another in the PASS. A larger number of actor switches typically indicates that an actor is likely to be evicted out of the on-chip SPM after its execution and thus a higher code overlay overhead.

Assume we have a program that consists of only one producer and one consumer, as described in Figure 3.6. Per execution, A pushes three tokens to edge A→B and B pops two tokens from the edge. In a PASS or a steady-state execution, A will have two executions and B will have three executions. A Minimum Buffer Schedule (MBS) that achieves the smallest buffer usage is given by PASS={A, B, A, B, B}. The buffer usage of this MBS is four tokens (or 400 bytes) and the number of actor switches is four, as illustrated in Table 3.1. Note that one of the actor switches is introduced between the last execution of B and the first execution of A. A Minimum Switch Schedule (MSS) is given by PASS={A, A, B, B, B}. The buffer usage of the MSS is six tokens (or 600 bytes) and the number of actor switches is two.

Table 3.1: Design trade-offs with PASS generation. Buffer Usage and Available Code Memory in the table are represented in bytes.

Minimum Buffer Schedule		Minimum Switch Schedule	
A, B, A, B, B		A, A, B, B, B	
Buffer Usage	400	Buffer Usage	600
actor Switches	4	actor Switches	2
SPM Size = 800			
Avail. Code memory	400	Avail. Code memory	200
Overlay Cost	ABAB	Overlay Cost	$+\infty$
SPM Size = 1000			
Avail. Code Memory	600	Avail. Code Memory	400
Overlay Cost	ABAB	Overlay Cost	AB

In Table 3.1 row 5-7, we examine the code overlay cost of MBS and MSS under an SPM size of 800 bytes. In this configuration, the memory available for code in MBS is 400 bytes. Since the code size of A and B are also 400 bytes as given in Figure 3.6, the code memory is able to accommodate one and only one actor at any time interval. The code overlay overhead for MBS is equal to the cost of transferring actors A and B each twice from the remote memory to the on-chip SPM. Under the same configuration, the memory available for code in MSS is 200 bytes. In this case, the total code memory is less than the largest actor code size (400 bytes), therefore the program cannot execute and we set the overlay cost to be $+\infty$.

In another configuration, we set the SPM size to be 1000 bytes as illustrated in Table 3.1 row 8-10. The memory available for code in MBS is 600 bytes and in MSS, 400 bytes. In both cases, the total code memory can only accommodate one and only one actor. From Table 3.1, MBS will have four actor switches (two A $\Rightarrow$ B and two B $\Rightarrow$ A) and MSS will have two actor switches (one A $\Rightarrow$ B and one B $\Rightarrow$ A). The code overlay overhead of MSS is half of the overlay overhead of MBS in this case.

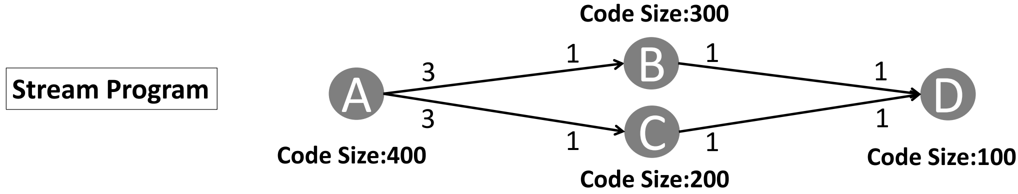


Figure 3.7: Stream program with four actors. The code sizes are in bytes.

From the above discussion, we can see that neither MBS nor MSS is always optimal for our latency minimization problem. A schedule that achieves the minimum latency should balance the buffer usage and the number of actor switches rather than concentrate on optimizing one of them.

3.2.2 Actor to region assignments

At this stage, we assume that we already have a PASS and the available code memory is calculated correspondingly. Under the scenario that not all code can fit into the available code memory, a programmer or compiler has to partition the code memory into regions and assign actors to regions. For a stream program described in Figure 3.7, the total code size is 1000 bytes. Table 3.2 provides two feasible actor to region assignments under a code memory of 700 bytes. One is given by {A, B} {C} {D} and the other given by {A} {B, C, D}. Actors within one pair of braces are assigned to the same region. In both solutions, the sum of all region sizes is 700 bytes, indicating that the code memory constraint is satisfied. In the first solution, actors A and B are overlayed with each other and in the second solution, actors B, C, and D are overlayed with each other. Depending on the DMA behavior and the PASS from the previous stage, either of the two solutions in Table 3.2 could be superior than the other.

Table 3.2: Design trade-offs with actor to region assignments. Available Code Memory and Region Sizes in the table are represented in bytes.

Avail. Code Memory=700				
1st Region Assignments			2nd Region Assignments	
{A, B}	{C}	{D}	{A}	{B, C, D}
R_{AB}=400	R_C=200	R_D=100	R_A=400	R_{BCD}=300
Overlayed Actors: A,B			Overlayed Actors: B,C,D	

3.2.3 Segmentation

The communication cost of transferring code or data between the on-chip SPM and the remote memory can be modeled as

$$T_c(x) = T_{base} + T_{slope} * x \tag{3.1}$$

In Equation (3.1), T_{base} captures the base cost that is encountered for every DMA transfer. $T_{slope} * x$ calculates the addition overhead for every byte of code or data being transferred. In the segmentation stage, we exploit the opportunities of grouping actors in the same region into segments to amortize the DMA base cost. The resulted segment size is equal to the sum of all actor code sizes assigned to it. The largest segment of each region determines the region size. For the same example given in Figure 3.7, assume the code memory available is 700 bytes and we adopt the second solution of actor to region assignments as presented in Table 3.2. For region {B, C, D}, we can group actors C and D together without violating the region size. However, grouping actor C and D into one segment doesn't always promise us a performance improvement. Consider two feasible PASS of the stream program described in Figure 3.7. The first PASS is given by A, B, B, B, C, D, C, D, C, D. In this case, actors C and D are always executed together. Grouping C and D into one segment will definitely result in code overlay overhead reduction. A

second PASS for the same stream program could be A, B, C, B, C, B, C, D, D, D. Now if we group actors C and D into one segment, it's likely that the code overlay overhead will increase, considering that for the first two executions of actor C, we bring in C and D together (C, D are grouped into the same segment). However, only actor C is executed before the execution of actor B evicts both C and D out of the on-chip SPM.

3.2.4 Data overlay

Data overlay optimization reduces the data buffer usage by circling data tokens around the off-chip main memory and reduces the total time that they are present in the local SPM. Extra DMA transfers that are used to implement data overlay could result in extra latency overhead. For the example given in Figure 3.8 with PASS={A, B, C, D}. Without data overlay, when we execute actor B, there are 9 tokens alive, 1 on edge A→B, 4 on edge B→C, and 4 on edge A→D. This is also the time interval with the largest buffer usage, which is 900 Bytes. With data overlay, we can transfer the 4 tokens on edge A→D to the off-chip main memory after A finishes execution and then retrieve the data tokens before the execution of D. In this case the buffer usage is 500 bytes. The minimum memory requirement without data overlay thus is 900 (buffer usage)+100 (overlay region) = 1000 bytes. With data overlay, the minimum memory requirement is 500 (buffer usage) + 100 (overlay region) = 600 bytes. Data overlay optimization reduces the buffer usage at the expense of potential increase of data overlay overhead. Another side effect of data overlay optimization is the occupancy of DMA engine, which could potentially block the code pre-fetching discussed in the previous paragraph.

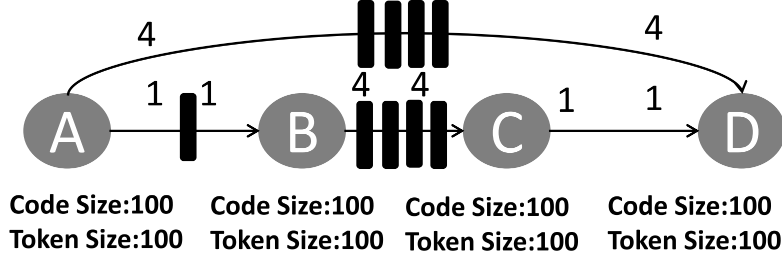


Figure 3.8: Design trade-offs with data overlay.

3.3 Problem Formulation

In this section, we formulate the problem that was discussed in the previous sections. The input to our problem is given by an SDF specification and an architecture description. More specifically, the SDF specification is given by graph $G < V, E >$ where each node $j \in V$ represents an actor and each edge $e \in E$ represents a data transfer between two actors. A node j is again given by the following parameters $< C_j, \tau_j, N_j >$ and an edge e is given by $< P_{ej}, C_{ej} >$ as described in Table 3.3. The architecture description is specified by its on-chip SPM size and DMA behavior as illustrated in Table 4.1, $P < C_p, \tau_{base}, \tau_{slope} >$.

Further, in our problem instances, we assume that the memory for storing library functions, global data, stack and heap[4] has already been reserved. Consequently the on-chip SPM memory C_p is only partitioned for program code and internal buffer. Given an SDF specification and architecture description, the objective of our techniques is to derive a PASS with actor to region assignments $< V, R >$ and actor to segment assignments $< V, S >$ such that the code overlay overhead is minimized. The calculation of the code over-

[4]Since in a stream application we typically have no recursive calls and dynamic allocations of memories, we can assume that the stack and heap size is bounded by a constant.

Table 3.3: Architecture and SDF Description

Constant			Description
SDF (G)	V	C_j	Code size of actor j
		τ_j	Run time of actor j
		N_j	executions of actor j in a PASS
	E	P_{ej}	Tokens produced to edge e by actor j.
		C_{ej}	Tokens consumed from edge e by actor j.
Arch. (P)	SPM	C_p	Scratch pad memory size
	DMA	τ_{base}	Base latency for any DMA transfer
		τ_{slope}	Additional latency increasing rate with data size

lay cost is fairly complicated and will be discussed in our 3-stage ILP approach as the objective function and in our heuristic approach as a subroutine.

3.4 Related Work

Many previous work have been conducted over the years to statically assign program code and data to SPM based architectures. Steinke et al. [69] presented an algorithm that selectively chooses program code and data to place in SPM. Angiolini et al. [1] [2] developed a post compiler technique that maps certain segments of external memory to physically partitioned banks of an on-chip SPM. Avissar et al. [3] presented a compiler strategy that partitions global and stack data among different memory units. Nguyen et al. [59] presented a memory allocation scheme for embedded systems where the SPM size is unknown at compile time. In contrast to these approaches, our work focuses on dynamic management of an SPM where code segments are overlayed with each other during run time.

There have been several previous work that address the problem of dynamic management of SPM with code overlay. Verma et al. [75] [74] discussed

the dynamic management of SPM as an extension to the Global Register Allocation problem and proposed an allocation technique that copies program code and data into SPM at runtime. Egger et al. [26] provided an integer linear programming (ILP) approach that loads required program code into the SPM on demand at runtime. Janapsatya et al. [37] developed an optimization that utilizes concomitance metrics to determine appropriate code segments to be loaded into an SPM. Pabalkar et al. [62] presented an ILP and a heuristic that overlay code based on static analysis of the global call control flow graph (GCCFG) of a program. While the above work focus on minimization of power or energy consumption, our work studies the dynamic management of SPM with the objective of overlay minimization.

Most recently Baker et al. [5] addressed instruction mapping on an SPM by partitioning it into regions and loading functions to regions. Functions assigned to the same region are overlayed with each other during program execution. Jung et al. [41] also utilized the same function to region assignment scheme and presented two heuristics for generating function to region mappings. Our work distinguishes from the above two approaches in that we focus on stream applications rather than traditional C++ programs. In our problem, a steady-state schedule of the stream program has to be generated together with code, data memory partition. Apart from actor to region assignments, we also introduce segmentation to amortize DMA base cost. Further, we also extend our heuristic approach with code pre-fetching and data overlay techniques to overlap DMA transfers with actor executions.

In the literature of scheduling SDF models on SPM enhanced embedded processors, Bandyopadhyay et al. [6] [7] present an SPM allocation scheme

that makes optimal use of SPM by analyzing the structure and semantics of a heterogeneous data flow model. However, their work is focused on static allocation of code and data, whereas in our approach we optimize for dynamic code and data overlay. This chapter presents a 3-stage ILP and fast heuristic approaches that are able to perform SDF scheduling, region assignment and segmentation. Further, in addition to the basic code pre-fetching approach, the proposed heuristic also incorporates deep code pre-fetching and data overlay optimizations. The deep pre-fetching optimization tries to issue a pre-fetching of code at a much earlier time than a basic pre-fetching and data overlay optimization tries to reduce the data buffer usage of a schedule by transferring data to the off-chip memory after it is produced, and retrieve it before it is consumed.

3.5 3-stage ILP Without Pre-fetching

Given the problem described above, we propose a 3-stage ILP to calculate a steady-state schedule as well as actor to region, and actor to segment assignments that minimize the overlay overhead under tight SPM constraints. Our first stage ILP calculates the schedule of actor executions with the objective of granting sufficient memory for code with moderate actor switches. In our second stage ILP, we partition the code memory into regions and investigate actor to region assignments to further reduce the code overlay overhead. In the last stage, we utilize segmentation to group small actors into segments to amortize the base DMA cost. The output of our approach is a steady-state schedule coupled with code and data memory partition as well as actor to segment and actor to region assignments. Since the variables that model

the stream characteristics and architecture features are the same for all three stages, we provide them globally in Table 3.3.

3.5.1 Stage 1 ILP: Scheduling

In this stage, we calculate the sequence of actor executions in the steady-state schedule based on the assumption that each actor occupies a separate segment and there is only one region for code. The code overlay overhead is estimated by the number of actor switches in the steady-state execution and the memory available for code.

3.5.1.1 Decision variables

- a_{imjn}, $\{0,1\}$, indicates whether the m_{th} copy of actor i executes before the n_{th} copy of actor j in the steady-state schedule.
- n_j, *integer*, indicates actor j has executed n_j times before entering the steady-state execution.
- C_{data}, *integer*, indicates the memory allocated for data.
- C_{code}, *integer*, indicates the memory allocated for code.

3.5.1.2 Derived variables

- b_{jn}, *integer*, indicates the number of actor executions except for actor j itself before its n_{th} copy.

$$\forall j \in V, \forall n \in N_j : b_{jn} := \sum_{i \in V \setminus j} \sum_{m \in N_i} \begin{cases} a_{imjn}, i < j \\ 1 - a_{jnim}, i > j \end{cases}$$

- s_{jn}, $\{0,1\}$, indicates whether the n_{th} copy of actor j is adjacent to its $(n+1)_{th}$ copy in the steady-state schedule.

$$\forall j \in V, \forall n \in [1, |N_j|-1] : (1 - s_{jn}) * M_j \geq b_{jn+1} - b_{jn}$$

$$\forall j \in V : (1 - s_{j|N_j|}) * M_j \geq b_{j|N_j|} - b_{j1}$$

$$\text{where } M_j := \sum_{i \in V \setminus j} |N_i|.$$

3.5.1.3 Constraints

1. Execution Order:

   i) In a legal sequence, the m_{th} copy of actor i executes either before or after the n_{th} copy of actor j.

$$\forall i \in [1, |V|-1], \forall m \in N_i, \forall j \in [i+1, |V|], \forall n \in N_j :$$

$$a_{imjn} + a_{jnim} = 1$$

   ii) If the execution sequence is the m_{th} copy of i followed by the n_{th} copy of j followed by the l_{th} copy of k, then the m_{th} copy of i must execute before the l_{th} copy of k.

$$\forall i \in V, \forall m \in N_i, \forall j \in [i, |V|], \forall n \in N_j, \forall k \in [j, |V|], \forall l \in$$

$$N_k : a_{imkl} \geq a_{imjn} + a_{jnkl} - 1, a_{imkl} \leq a_{imjn} + a_{jnkl}$$

   iii) The m_{th} copy of actor i always executes before its $(m+1)_{th}$ copy.

$$\forall i \in V, m \in [1, |N_i|-1], n \in [m+1, |N_i|] : a_{imin} = 1$$

2. Initial Buffer Condition: The initial buffer distribution on each edge, denoted by C_e^0, must be non-negative.

$$\forall e \in E : C_e^0 := \sum_{j \in V} n_j * (P_{ej} - C_{ej}), C_e^0 \geq 0$$

3. Data Dependency: After each actor execution, there is no edge with negative tokens. The following constraint, C^e_{jn} indicates the accumulated buffer usage immediately after executing the n_{th} copy of actor j. C^0_e provides the initial buffer condition, $n * (P_{ej} - C_{ej})$ computes the data stored to each edge by n executions of actor j. The last summation computes the tokens produced/consumed by actors $(i \in V \setminus j)$ executed before the n_{th} copy of j.

$$\forall j \in V, \forall n \in N_j, \forall e \in E : C_{jne} := C^0_e + n * (P_{ej} - C_{ej})$$
$$+ \sum_{i \in V \setminus j} \sum_{m \in N_i} (P_{ei} - C_{ei}) * \begin{cases} a_{imjn}, i < j \\ 1 - a_{jnim}, i > j \end{cases}, \; C_{jne} \geq 0$$

4. Data Memory: For each execution of actor j, we calculate the sum of memory usage at all edges. The memory allocated for data should be greater or equal to the maximum of them.

$$\forall j \in V, n \in N_j : C_{data} \geq \sum_{e \in E} C_{jne}$$

5. Code Memory: The memory allocated for code must be at least as large as any actor.

$$\forall j \in V : C_{code} \geq C_j$$

6. Processor Memory: The memory allocated for code and data should be no more than the size of the scratchpad memory.

$$C_p \geq C_{data} + C_{code}$$

7. Overlay Code Size Lower Bound: The lower bound of code size is given by the portion of the code size that exceeds the code memory.

$$C_{low} \geq \sum_{j \in V} C_j - C_{code},\ C_{low} \geq 0$$

3.5.1.4 Objective function

- We utilize the number of actor switches in the steady-state schedule and the memory allocated for code to estimate the overlay overhead,

$$Minimize \quad \tau_{overlay} := \lambda * \sum_{j \in V} \sum_{n \in N_j} (1 - s_{jn}) + 2\mu * C_{low}$$

In the objective function, $\sum_{j \in V} \sum_{n \in N_j} (1 - s_{jn})$ calculates the number of actor switches in the steady-state schedule and C_{low} implies the lower bound of code size to be overlayed. λ and μ indicate the weights assigned to actor switches and lower bound of code size to be overlayed, respectively. If $\lambda = 0$ then a minimum buffer schedule is derived; if $\mu = 0$ then a minimum actor switching schedule is derived. In our objective function, we set $\lambda = \sum_{j \in V} \frac{1}{|N_j|}$ and $u = \sum_{j \in V} \frac{1}{C_j}$. The weight assignment can be interpreted as saving a memory equal to average actor code size is equivalent to reducing n actor switches, where n is the average number of actor copies in the steady-state execution. Since C_{low} is a lower bound, and in the steady-state schedule not only can an actor have multiple copies, but also different actors can be mapped to the same memory location, we add a factor of 2 to increase the weight of memory.

3.5.2 Stage 2 ILP: Region Assignment

In this stage we already have the actor execution sequence from the previous stage and we calculate the number of regions, the size of each region, and actor to region assignments. The objective is to minimize code overlay estimated by the sum of actor switches of all the regions.

3.5.2.1 Variables derived from Stage 1

- a_{jn}, *integer*, indicates the n_{th} copy of actor j is assigned to slot a_{jn} in the steady-state schedule.
- C_{code},*integer*, indicates the memory allocated for code.

3.5.2.2 Decision variables

- b_{jr}, $\{0, 1\}$, indicates actor j is assigned to region r.

3.5.2.3 Derived variables

- c_{ir}, $\{0, 1\}$, indicates the i_{th} slot is assigned to region r.

$$\forall j \in V, n \in N_j, r \in R : c_{ir} = b_{jr},\ \text{where } i = a_{jn}$$

- s_{jn},$\{0, 1\}$, if $|V_j| \geq 2$ indicates in region r, whether the $(n+1)_{th}$ copy of actor j executes immediately after the n_{th} copy of j, or the first copy of actor j in the next steady-state execution executes immediately after the last copy of j in the current execution; else, s_{j1} indicates for a single appearance actor j, whether there are other actors assigned to the same region. If yes, $s_{j1} = 0$ indicates there is an actor switch for j; else $s_{j1} = 1$

implies that actor j is always present in the memory.

$$\forall j \in V, \forall r \in R : \begin{cases} \text{if } |N_j| \geq 2, \\ \qquad \forall n \in [1, |N_j| - 1], \forall i \in [a_{jn} + 1, a_{jn+1} - 1] : \\ \qquad s_{jn} \leq 2 - b_{jr} - c_{ir}, \\ \qquad \forall i \in [1, a_{j1} - 1] \cup [a_{j|N_j|} + 1, N] : \\ \qquad s_{j|N_j|} \leq 2 - b_{jr} - c_{ir}, \\ \text{else} \\ \qquad \forall i \in V \setminus j : s_{j1} \leq 2 - b_{jr} - b_{ir} \end{cases}$$

where $N = \sum_{j \in V} |N_j|$.

3.5.2.4 Constraints

1. Actor to Region Assignment: Each actor is assigned to one and only one region.

$$\forall j \in V : \sum_{r \in R} b_{jr} = 1$$

2. Region Size: The size of each region is no less than the biggest actor being assigned to it.

$$\forall j \in V, \forall r \in R : C_r \geq b_{jr} * C_j$$

3. Code Memory: The sum of the sizes of all the regions should be no more than the memory allocated for code.

$$C_{code} \geq \sum_{r \in R} C_r$$

3.5.2.5 Objective function

- The objective at this stage is to minimize the code overlay overhead, which is estimated by the number of actor switches in all the regions.

$$Minimize \quad N_{switch} := \sum_{j \in V} \sum_{n \in N_j} (1 - s_{jn})$$

3.5.3 Stage 3 ILP: Segmentation

In this stage we have the sequence of actor executions as well as the actor to region assignment from the previous stages and we calculate actor to segment assignment to amortize the the base cost of DMA operation. The objective is to further minimize the code overlay with actor segmentation.

3.5.3.1 Results derived from Stage 1 and Stage 2

- a_{jn}, $integer$, indicates the slot the n_{th} copy of actor j is assigned to.
- C_{code}, $integer$, memory available for code overlay.
- b_{jr}, $\{0, 1\}$, indicates actor j is assigned to region r.
- g_{ir}, $integer$, indicates slot i in region r is corresponding to slot g_{ir} in the steady-state schedule.

3.5.3.2 Decision variables

- d_{js}, $\{0, 1\}$, indicates actor j is assigned to segment s.

3.5.3.3 Derived variables

- C_s, *integer*, indicates the size of segment s, which is given by the sum of code sizes of all the actors assigned to segment s.

$$\forall s \in S : C_s := \sum_{j \in V} d_{js} * C_j$$

- e_{is}, $\{0, 1\}$, indicates whether the actor resides in the i_{th} slot in the steady-state schedule is assigned to segment s.

$$\forall j \in V, \forall n \in N_j, \forall s \in S : e_{xs} := d_{js}, \text{ where } x = a_{jn}$$

- x_{jk}, $\{0, 1\}$, indicates whether actor j, k are assigned to the same segment.

$$\begin{aligned} &\forall j \in [1, |V| - 1], \forall k \in [j + 1, |V|], \forall s \in S : \\ &x_{jk} \geq d_{js} + d_{ks} - 1 \end{aligned}$$

- y_{jk}, $\{0, 1\}$, indicates whether actor j, k are assigned to the same region.

$$\forall j \in [1, |V| - 1], \forall k \in [j + 1, |V|] : y_{jk} := \sum_{r \in R} b_{jr} * b_{kr}$$

- N_r, *integer*, indicates the number of slots in region r.

$$\forall r \in R : N_r := \sum_{j \in V} b_{jr} * |N_j|$$

- s_{ir}, $\{0, 1\}$, indicates whether the consecutive slots i and $i + 1$ in region r are assigned to different segments.

$$\forall r \in R, \forall i \in [1, N_r], \begin{cases} \text{if } N_r \geq 2 \text{ and} \\ \qquad \text{if } i \neq N_r, \forall s \in S : s_{ir} \geq e_{xs} - e_{ys}, \\ \qquad \text{else}, \forall s \in S : s_{ir} \geq e_{xs} - e_{zs}, \\ \text{else}, \ s_{1r} = 0 \end{cases}$$

where $x = g_{ir}, y = g_{i+1r}, z = g_{1r}$.

- τ_{ris}, $real$, indicates the overhead of fetching segment s if there is a code overlay between the i_{th} slot and $(i+1)_{th}$ slot in region r. The overhead is corresponding to fetching the segment which the $(i+1)_{th}$ slot in region r is assigned to. There is an overlay overhead between the i_{th} slot and $(i+1)_{th}$ slot in region r if they are being assigned to different segments. The last slot and the fist slot in the same region are also treated as consecutive slots and handled accordingly.

$$\forall r \in R, \begin{cases} \text{if } (N_r \geq 2) \\ \quad \forall i \in [1, N_r - 1], s \in S : \\ \quad \tau_{ris} \geq T_{base} + C_s * T_{slope} + (s_{ir} + e_{xs} - 2) * M \\ \quad \tau_{ris} \geq 0 \\ \quad \forall r \in R, s \in S : \\ \quad \tau_{rN_rs} \geq T_{base} + C_s * T_{slope} + (s_{N_rr} + e_{ys} - 2) * M \\ \quad \tau_{rN_rs} \geq 0 \end{cases}$$

where $x = g_{i+1r}, y = g_{1r}, M = T_{base} + \sum_{j \in F} C_j * T_{slope}$.

3.5.3.4 Constraints

1. Actor to Segment Assignment: Each actor j is assigned to one and only one segment.

$$\forall j \in V : \sum_{s \in S} d_{js} = 1$$

2. Segment to Region Assignment: If actor j, k are assigned to the same segment, then they must be assigned to the same region.

$$\forall j \in [1, |V| - 1], \forall k \in [j + 1, |V|] : x_{jk} \leq y_{jk}$$

3. Region Size: The region size is greater or equal to the largest segment assigned to it.

$$\forall r \in R, \forall s \in S : C_r \geq \sum_{j \in V} b_{jr} * d_{js} * C_j$$

4. Code Memory: The code memory must be able to accommodate all the regions.

$$C_{code} \geq \sum_{r \in R} C_r$$

3.5.3.5 Objective function

- The objective is to minimize overlay overhead for the given schedule and actor to region assignments from the previous steps,

$$Minimize \quad \tau_{overlay} := \sum_{r \in R} \sum_{i \in [1, N_r]} \sum_{s \in S} \tau_{ris}$$

3.6 3-stage ILP With Pre-fetching

The DMA Engine in the SPE works independently from the execution unit, which enables us to overlap code overlays with actor executions. In this section we reformulate our 3-stage ILP to incorporate the code pre-fetching scheme, in which we initiate pre-fetching of the next actor into the local memory while executing the current actor. Compared to the 3-stage ILP without pre-fetching, the modifications are only made to the memory constraint and the cost functions. Therefore, we only discuss the constraints that are modified and the objective functions which are updated in the remainder of this section.

3.6.1 Stage 1 ILP: Scheduling

At this stage we make the same assumption of actor to segment and region assignments as discussed in the previous Stage 1 ILP. However, we utilize the average actor run time to estimate the actor executions that can be overlapped with code overlay. One more derived variable is added to the original formulation as described below.

3.6.1.1 Additional derived variables

- τ_{jn}, $real$, indicates the code overlay overheads for the n_{th} copy of actor j with code pre-fetching incorporated.

$$\tau_{jn} \geq ((\tau_{base} + \tau_{slope} * C_j) - \tau_{avg}) * (1 - s_{jn})$$

$$\tau_{jn} \geq 0$$

$$\text{where } \tau_{avg} := \sum_{j \in V} \tau_j * |N_j| / (\sum_{j \in V} |N_j|).$$

3.6.1.2 Modified constraints

- Code Memory: The memory allocated for code with pre-fetching must be able to accommodate the largest actor plus a buffer for storing the pre-fetched code. The size of the buffer for the pre-fetched code should be at least as large as any actor. Therefore the modified code memory constraint is given by,

$$\forall j \in V : C_{code} \geq 2 * C_j$$

3.6.1.3 Updated objective function

- The objective is to minimize code overlay overhead with code pre-fetching. In the following equation the numerator in the summation calculates the overlay overheads based on the assumption that each actor occupies a separate segment and all actors are being assigned to the same region. We divided it by the base DMA cost to estimate the corresponding actor switches. Therefore, we can leave the weight factors λ and μ and the second element in the original objective function untouched.

$$Minimize \quad \tau_{overlay} := (\lambda/\tau_{base}) * (\sum_{j \in V} \sum_{n \in N_j} \tau_{jn}) + 2\mu * C_{low}$$

3.6.2 Stage 2 ILP: Region Assignment

In Stage 2, based on the sequence derived from the previous stage, we calculate the overlay overhead of each actor execution and the available execution time that can be utilized to overlap with the code overlay DMA. We define the available execution time for overlap (ETA) due to a particular actor in a slot of the schedule as the sum of the execution time of the actor in that slot plus any executions of the same actor in immediately previous slots. We denote

ETA for n^{th} copy of actor j assigned in slot $i = a_{jn}$ of the schedule as γ_i. For example in schedule $AABAB$, $\gamma_1 = \tau_A, \gamma_2 = 2\tau_A, \gamma_3 = \tau_B, \gamma_4 = \tau_A$ and $\gamma_5 = \tau_B$. We treat the last actor execution in the schedule and the first execution as consecutive actor executions too.

3.6.2.1 Additional variables derived from Stage 1

- γ_i, $real$, indicates ETA of slot i in the steady-state schedule.

3.6.2.2 Additional derived variables

- τ_{jn}, $real$, indicates the overlay overhead for the $(n+1)_{th}$ copy of actor j after pre-fetching. Notice that the last slot in the previous execution is also treated as immediately before the current first slot. Therefore,

$$\forall j \in V, n \in N_j : \tau_{jn} \geq ((\tau_{base} + C_j * \tau_{slope}) - \gamma_x) * s'_{jn}$$

where $s'_{jn} = 1 - s_{jn}$, x is the slot immediately before the $(n+1)_{th}$ copy of j.

$$x = \begin{cases} \sum_{j \in V} |N_j|, \text{ if } n = |N_j| \text{ and } a_{j1} = 1, \\ a_{j1-1}, \text{ if } n = |N_j| \text{ and } a_{j1} \neq 1, \\ a_{j(n+1)-1}, \text{ else.} \end{cases}$$

3.6.2.3 Updated constraints

- Code Memory: The size of the code memory must be greater than or equal to the sum of all the region sizes plus the buffer for pre-fetched code.

$$C_{code} \geq \sum_{r \in R} C_r + C_{fmax}, \text{where } C_{fmax} \text{ is the largest code size.}$$

3.6.2.4 Updated objective function

- The objective at this stage is to minimize the overlay overhead with pre-fetching optimization.

$$Minimize \quad \tau_{overlay} := \sum_{j \in V} \sum_{n \in N_j} \tau_{jn}$$

3.6.3 Stage 3 ILP: Segmentation

In this stage we group small actors into segments and pre-fetch a segment instead of an actor with the objective of minimizing the code overlay overhead.

3.6.3.1 Additional derived variables

- C_{smax}, $integer$, indicates the maximum segment size.

$$\forall s \in S : C_{smax} \geq C_s$$

3.6.3.2 Updated derived variables

- τ_{ris}, $real$, indicates the overhead of fetching segment s for the i_{th} slot in region r if any.

$$\forall r \in R, \begin{cases} \text{if } N_r \geq 2 \\ \quad \forall i \in [1, N_r - 1], s \in S : \\ \quad \tau_{ris} \geq T_{base} + C_s * T_{slope} - \gamma_z + (s_{ir} + e_{ys} - 2) * M \\ \quad \tau_{ris} \geq 0 \\ \quad \forall r \in R, s \in S : \\ \quad \tau_{rN_rs} \geq T_{base} + C_s * T_{slope} - \gamma_z + (s_{N_rr} + e_{xs} - 2) * M \\ \quad \tau_{rN_rs} \geq 0 \end{cases}$$

where $x = g_{1r}, y = g_{i+1r}, M = T_{base} + \sum_{j \in F} C_j * T_{slope}$, and $z = \begin{cases} g_{ir}, & \text{if } g_{ir} \geq 2, \\ N, & \text{else.} \end{cases}$

3.6.3.3 Updated constraint

- Code Memory: The size of the code memory must be greater than or equal to the sum of all the region sizes plus the buffer for pre-fetched segment.

$$C_{code} \geq \sum_{r \in R} C_r + C_{smax}$$

The objective function at this stage remains unchanged since we incorporate the effect of pre-fetching of segments into the calculation of τ_{ris}.

3.7 SDF Scheduling Heuristic

Although our 3-stage ILP approach generates close to optimal solutions, it suffers from very long algorithm run time. In this section, we present a fast heuristic approach that is able to achieve comparable performance results in a matter of seconds. We first provide a base approach where a PASS for the given SDF is generated simultaneously with actor to region and actor to segment assignments. The objective is to minimize code overlay overhead.

3.7.1 Code overlay overhead calculation

Prior to discussion of our SDF scheduling heuristic, we provide the calculation of code overlay overhead as a subroutine in Algorithm 1. In the algorithm we first initialize *code_overlay* to be 0, Line 1. *mem_state* is an array that

```
1:  code_overlay ← 0
2:  for r ∈ R do
3:      /* s_last is the last segment that is loaded to region r following PASS */
4:      Initialize mem_state[r] ← s_last
5:  end for
6:  for i ∈ [0, |PASS| − 1] do
7:      s_cur ← getSegment(< V, S >, i)
8:      r_cur ← getRegion(< V, R >, i)
9:      if s_cur ≠ mem_state[r_cur] then
10:         code_overlay ← code_overlay + T_c(C_{s_cur})
11:         mem_state[r_cur] ← s_cur
12:     end if
13: end for
14: return code_overlay
```

Algorithm 1: $calCodeOverlay(G, PASS, < V, R >, < V, S >)$

keeps track of the segments that are present in each region. We assume the SDF is being executed in an iterative manner, therefore the segment in each region before the current execution is given by the last segment that is loaded to each region in the previous execution, Line 2-4. For each actor execution in the given PASS, the subroutine checks whether segment s_{cur} that contains the actor is already loaded to its corresponding region r_{cur}. If segment s_{cur} is absent from r_{cur}, we increase *code_overlay* by $T_c(C_{s_{cur}})$, Line 10. $T_c(C_{s_{cur}})$ calculates the DMA cost for transferring segment s_{cur} from the off-chip memory to the local SPM. T_c is the DMA cost function that is discussed in Section 3.2.3. $C_{s_{cur}}$ is given by $C_{s_{cur}} = \sum_{v \in s_{cur}} C_v$, where C_v is the code size of actor v[5]. Then $mem_state[r_{cur}]$ is updated with s_{cur}, indicating segment s_{cur} is loaded into region r_{cur}, Line 11. After iterating through the entire PASS, *code_overlay* is returned.

[5] C_v is equivalent to C_j described in Table 3.3

3.7.2 Overall description

A high level description of our heuristic approach is given in Algorithm 2. In the algorithm we first initialize the overlay overhead $min_overlay$ to be infinitely large. The initial PASS is set to be a minimum buffer schedule of the given SDF [38]. We deliberately evolve the PASS from a minimum buffer schedule to a minimum actor switch schedule in this algorithm. The buffer usage buf_mem and SPM memory available for code, $code_mem$, is calculated based on the given PASS. Starting from Line 5, we enter an iterative procedure where at each iteration, we perform actor to region assignments ($RegionAssignment$) and actor to segment assignments ($Segmentation$). The implementation details of the $RegionAssignment$ and $Segmentation$ are given by Algorithm 3 and Algorithm 4 with the discussions provided in Sections 3.7.3 and 3.7.4, respectively. The total region size after $RegionAssignment$ is calculated by $\sum_{r \in R} C_r$, where C_r denotes the size of region r and is given by $max_{v \in r} C_r$. Line 8 in Algorithm 2 checks whether $RegionAssignment$ is successful. If $RegionAssignment$ succeeds, further actor to segment mapping is generated and $cur_overlay$ is updated accordingly, Line 12. If $cur_overlay$ is less than $min_overlay$, we update $min_overlay$ and store the current PASS, actor to region/segment assignments to solution.

After current evaluation, we generate the next PASS to be evaluated by collapsing two non-consecutive executions of the same actor in the PASS, Line 15. We implement this procedure by creating a copy of the current $PASS$ for every actor execution. In the temporary $PASS$ we check whether there is another execution v_{next} of the same actor in the $PASS$. If v_{next}'s previous

```
1: Initialize min_overlay ← +∞
2: Initialize PASS ← MinBufferScheduling(G)
3: Calculate current buffer usage, buf_mem, of PASS
4: code_mem ← C_p − buf_mem
5: repeat
6:    /*Actor to region assignment*/
7:    <V,R> ← RegionAssignment(G, PASS, code_mem)
8:    if ∑_{r∈R} C_r ≤ code_mem then
9:       /*Actor to segment assignment*/
10:      <V,S> ← Segmentation(G, PASS, <V,R>)
11:      /*Overlay overhead*/
12:      cur_overlay ← calCodeOverlay(G, PASS, <V,R>, <V,S>)
13:      if cur_overlay < min_overlay then
14:         min_overlay ← overlay
15:         solution ← clone(G, PASS, <V,R>, <V,S>)
16:      end if
17:   end if
18: /*Evolve from Min. Buf to Min. Switch*/
19: until collapseTwoExecs(PASS) = false
20: return solution
```

Algorithm 2: $MinOverlayScheduling(G, P)$

actor execution is not the current actor, then v_{next} is found. We remove it from the temporary $PASS$ and inserted it right after v_{cur}. We validate the new $PASS$ by checking the data tokens on each edge at every time interval. If there is no negative token on any edge at any time interval, the new $PASS$ is legal. An illegal $PASS$ is discarded. Among all legal $PASS$s, we select the one that has the least buffer usage increment to be the next $PASS$ to be evaluated. The procedure terminates when no two non-consecutive executions of the same actor can be found. Upon termination, a solution that consists of a PASS, actor to region, and actor to segment assignments is returned.

3.7.3 Region assignments

For a given SDF, Algorithm 3 assigns actors to regions such that each actor is mapped to one and only one region and all regions fit into the available code memory. In this stage we assume that each actor occupies a separate segment. Since the actors being assigned to the same region are overlayed with each other over time, we would like them to interfere with each other as little as possible. In the algorithm, we use the number of times that two regions interfere with each other during the program execution to define their interaction factor (IF). In Algorithm 3, we first initialize actor to region assignments $< V, R >$ such that each actor is assigned to a different region. We construct IF table for every pair of regions $< r_i, r_j >$ by iterating through the given PASS. If there is an execution switching from region r_i to region r_j or vice versa, we increase IF of $< r_i, r_j >$ by one. The current total region size $region_mem$ is calculated based on the actor to region mapping. We keep on collapsing two regions with the smallest IF while the total region size is larger than $code_mem$ and the number of regions is more than one, Line 4. At each iteration, we collapse a region pair with minimum IF by moving all actors from r_j to r_i. If there are several region pairs that have the same minimum IF, we collapse the region pair that decreases $region_mem$ the most. After each collapsing of a region pair, we update actor to region assignments $< V, R >$, IF table and recalculate $region_mem$. The algorithm terminates when the first actor to region mapping $< V, R >$ that fits into $code_mem$ is found or $|R| = 1$. The complexity of IF table construction and the iterative procedure are both $O(n^3)$, where n is the number of actor executions in the given $PASS$. Therefore, the complexity of RegionAssignment algorithm is $O(n^3)$.

1: Initialize actor to region assignments $< V, R >$, as each actor occupies a separate region
2: Construct IF table entry for each region pair $< (r_i, r_j), Integer >$, where $r_i, r_j \in R, i < j$
3: $region_mem \leftarrow \sum_{r \in R} C_r$
4: while $region_mem > code_mem$ and $|R| > 1$ do
5: Collapse a region pair with minimum IF
6: Update $< V, R >$ and IF table
7: $region_mem \leftarrow \sum_{r \in R} C_r$
8: end while
9: return $< V, R >$

Algorithm 3: $RegionAssignment(G, PASS, code_mem)$

3.7.4 Segmentation

The actor sizes assigned to each region could be very diverse and the DMA base cost may be overwhelming if we have too many DMA transfers. In the segmentation phase, we explore opportunities of combining actors into segments to amortize the DMA base cost. In Algorithm 4, we initialize the actor to segment mapping $< V, S >$, as each actor occupies a different segment. The minimum overlay for the given PASS, $min_overlay$, is calculated based on the current PASS, $< V, R >$, and $< V, S >$ mappings. Then we start an iterative procedure where for each region, we examine combining every pair of segments s_i and s_j for opportunities of code overlay reduction. Note that even if two segments in the same region can be grouped together, it does not necessarily promise a performance gain (refer to the discussion provided in Section 3.2.3). In Algorithm 4, if collapsing the current segment pair does not violate the region size constraint, then we try to update the actor to segment mapping by moving all actors from s_j to s_i. If the overlay overhead after collapsing s_i and s_j is less than $min_overlay$, we update $min_overlay$. In the next iteration s_i is re-evaluated with every other segment. Otherwise, we

1: Initialize actor to segment assignments $< V, S >$, as each actor occupies a separate segment
2: $min_overlay \leftarrow calOverlay(G, PASS, < V, R >, < V, S >)$
3: for each region $r \in R$ do
4: for each segment pair (s_i, s_j), where $s_i, s_j \in R, i < j$ do
5: if collapsing s_i and s_j does not violate region size then
6: Update $< V, S >$
7: $overlay \leftarrow calOverlay(G, PASS, < V, R >, < V, S >)$
8: if $overlay < min_overlay$ then
9: $min_overlay \leftarrow overlay$
10: else
11: Restore $< V, S >$ to the previous state where s_i and s_j were not collapsed
12: end if
13: end if
14: end for
15: end for
16: return $< V, S >$

Algorithm 4: $Segmentation(G, PASS, < V, R >)$

restore the actor to segment mapping to the previous state where s_i and s_j are recognized as separated segments. The exhaustive search in Algorithm 4 is less expensive than enumerating every pair of segments in S and the complexity of *calCodeOverlay* is O(n). Therefore, the complexity of Algorithm 4 is $O(n^3)$.

3.7.5 Overall algorithm complexity

Given that the procedures of actor to region assignment and segmentation are both in $O(n^3)$ and they are nested in the loop of PASS generation, which is $O(n)$, the overall complexity of our heuristic is $O(n^4)$.

3.8 Extensions to SDF Scheduling Heuristic

In the section we extend our SDF scheduling heuristic with three optimizations, namely basic pre-fetching, deep pre-fetching, and data overlay. We discuss each of the optimization in the following.

3.8.1 Basic pre-fetching optimization

We first incorporate the basic pre-fetching optimization into our existing approach. Basic pre-fetching optimization tries to overlap DMA transfer of the current segment with the previous actor's execution. We denote the current actor being executed as v_{cur} and the previous actor executed as v_{pre}. The corresponding regions and segments for the current actor and the previous actor are r_{cur}, r_{pre}, s_{cur}, and s_{pre} respectively. If s_{cur} is absent from the local SPM, we try to overlap DMA transfer of s_{cur} with v_{pre}'s execution. That is, we try to issue a pre-fetching of s_{cur} right before the execution of v_{pre}. The overlay overhead with this basic pre-fetching optimization is calculated based on the following scenarios.

- If v_{cur} resides in the same region with v_{pre}, then code pre-fetching for s_{cur} cannot be issued because of memory conflict. In this case, DMA transfer for s_{cur} is given by $T_c(C_{s_{cur}})$ ($T_c(C_{s_{cur}})$ is provided in Section 3.7.1).
- Else, s_{cur} does not reside in the same region with v_{pre}. In this case, a code pre-fetching for segment s_{cur} can be issued before v_{pre} starts execution. The DMA transfer for s_{cur} is overlapped with v_{pre}'s execution and the resulting code overlay overhead is given by $T_{overlap}(s_{cur}) = max(0, T_c(C_{s_{cur}}) - \tau_{v_{pre}})$, where $\tau_{v_{pre}}$ indicates the run time of actor v_{pre}.

```
1: Initialize code_overlay ← 0
2: for r ∈ R do
3:     /* s_last is the last segment that is loaded to region r following PASS */
4:     Initialize mem_state[r] ← s_last
5: end for
6: for i ∈ [0, |PASS| − 1] do
7:     s_cur ← getSegment(< V, S >, i)
8:     r_cur ← getRegion(< V, R >, i)
9:     r_pre ← getRegion(< V, R >, (i − 1 + |PASS|)%|PASS|)
10:    if s_cur ≠ mem_state[r_cur] then
11:        if r_cur = r_pre then
12:            code_overlay ← code_overlay + T_c(s_cur)
13:        else
14:            code_overlay ← code_overlay + T_overlap(s_cur)
15:        end if
16:        mem_state[r_cur] ← s_cur
17:    end if
18: end for
19: return code_overlay
```

Algorithm 5: $calCodeOverlayBasicPre(G, PASS, < V, R >, < V, S >)$

If DMA of s_{cur} is less than τ_{pre}, then 0 is returned. Otherwise, the DMA cost that exceeds v_{pre}'s execution is returned.

We incorporate this basic pre-fetching in our original heuristic approach and update the code overlay overhead calculation as shown in Algorithm 5. In Algorithm 5, the initializations of $code_overlay$ and mem_state are identical to Algorithm 1. Then we calculate the code overlay overhead of each actor execution with basic pre-fetching, Line 6-18. The mod operation in $(i - 1 + |PASS|)\%|PASS|)$ wraps i to the end of a given $PASS$ when $i = 0$. The total code overlay overhead is returned after iterating through the entire PASS, Line 19.

3.8.2 Deep pre-fetching optimization

The basic pre-fetching optimization only considers overlapping the DMA fetch cost of s_{cur} with previous actor v_{pre}'s execution. However, the pre-fetching can be issued at a much earlier time to grant longer execution time to overlap with a DMA transfer. Essentially, we can trace back the given PASS and identify the last actor execution v^r_{pre} that resides in the same region with v_{cur}. A pre-fetching for s_{cur} can be issued as soon as v^r_{pre} finishes its execution. With this deep pre-fetching optimization, we may over use the DMA engine, meaning there could be several concurrent DMA transfers on the fly. To resolve this problem, we introduce a DMA engine status for each actor execution.

Algorithm 6 details the procedure for finding the maximum DMA engine idle period to issue a pre-fetch of s_{cur}. The inputs to Algorithm 6 are the current PASS, the DMA engine status and locations of v^r_{pre} and v_{cur}. In Algorithm 6, max_period, cur_period are first initialized to 0. max_period_start and cur_period_start are initialized to $v^r_{pre} + 1$. We iterate the current position cur_pos from $v^r_{pre} + 1$ to $v_{cur} - 1$. The mod operation in $(cur_pos + 1)\%|PASS|$ wraps the current position to the head of PASS when $(cur_pos + 1) \geq |PASS|$. The subroutine iteratively updates max_period_start and max_period when a DMA engine idle period that is larger than max_period is found. Upon termination, max_period_start and max_period is returned.

With deep pre-fetching a DMA transfer for s_{cur} is issued at time interval max_period_start and the corresponding code overlay overhead for transferring s_{cur} is given by Algorithm 7. The inputs to Algorithm 7 are

```
1:  max_period_start ← v^r_pre + 1, max_period ← 0
2:  cur_period_start ← v^r_pre + 1, cur_period ← 0
3:  for  cur_pos = v^r_pre + 1; cur_pos ≠ v_cur;
    cur_pos = (cur_pos + 1)%|PASS| do
4:     if STATUS[cur_pos] = idle then
5:        cur_period ← cur_period + τ_cur_pos
6:     else
7:        if cur_period > max_period then
8:           max_period ← cur_period
9:           max_period_start ← cur_period_start
10:       end if
11:       cur_period ← 0
12:       cur_period_start ← cur_pos + 1
13:    end if
14: end for
15: return < max_period_start, max_period >
```

Algorithm 6: $findMaxPeriod(PASS, STATUS, v^r_{pre}, v_{cur})$

the current PASS, the DMA engine status, the DMA cost of s_{cur}, the deep pre-fetching start location, and the current execution location. Starting from $cur_loc = max_period_start$, we keep on deducting actor's run time at cur_loc from dma_cost until dma_cost is less than or equal to 0, or we run out of the DMA idle period. Note that when the DMA fetch cost of s_{cur} is larger than max_period, part of the DMA fetch cost that is not overlapped with successive actor's executions is returned. We change the DMA engine status of cur_loc from idle to busy if it is utilized to implement deep pre-fetching.

To adopt the deep pre-fetching optimization into our heuristic, we issue a DMA transfer of s_{cur} at max_period_start for each segment that is overlayed and update the calculation for code overlay overhead as described in Algorithm 8. The difference from the basic pre-fetching algorithm is that when v_{cur} is absent from the local SPM, we first identify the range that a pre-

1: $cur_loc \leftarrow max_period_start$
2: while $dma_cost > 0$ and $STATUS[cur_loc] = idle$ and $cur_loc \neq v_{cur}$ do
3: $STATUS[cur_loc] \leftarrow busy$
4: $cost \leftarrow cost - \tau_{cur_loc}$
5: $cur_loc \leftarrow (cur_loc + 1)\%|PASS|$
6: end while
7: return $max(0, dma_cost)$

Algorithm 7: $setDMABusy(PASS, STATUS, dma_cost, max_period_start, v_{cur})$

fetching call for s_{cur} can be inserted, Line 10-13. Then we issue pre-fetching s_{cur} at the start of the longest DMA engine idle period, Line 14-16.

3.8.3 Data overlay optimization

To reduce the data buffer usage of a given schedule, we further introduce data overlay as discussed in Section 3.1.3. Now we have two ways of reducing memory, namely by collapsing two regions or by data overlay. We update Algorithm 3-*RegionAssignment* in our SDF scheduling approach with Algorithm 9 to either perform data overlay or collapsing two regions when the data buffer usage and the total region size exceed the local SPM. The first three lines in Algorithm 9 initialize $< V, R >$, IF, and $region_mem$, which are identical to Algorithm 3. Then life time of each data segment is initialized for implementing data overlay. A data segment is defined as the total number of tokens produced on each of an actor's outgoing edge. For example, given the SDF described in Figure 3.8, there are two data segments being produced at actor A' execution, one consists of 4 tokens (on edge $A \rightarrow D$) and one consists of 1 token (on edge $A \rightarrow B$). The life time of a data segment starts from the actor execution where it is produced and ends when all its tokens are consumed. $LIFE[i][j][k] = 1$ denotes that the data segment produced by

```
1: Initialize code_overlay ←0
2: for r ∈ R do
3:    /* s_last is the last segment that is loaded to region r following PASS */
4:    Initialize mem_state[r] ← s_last
5: end for
6: for j ∈ [0, |PASS| − 1] do
7:    s_cur ← getSegment(< V, S >, j)
8:    r_cur ← getRegion(< V, R >, j)
9:    if s_cur ≠ mem_state[r_cur] then
10:       i ← (j − 1 + |PASS|)%|PASS|
11:       while getRegion(< V, R >, i) ≠ r_cur do
12:          i ← (i − 1 + |PASS|)%|PASS|
13:       end while
14:       < max_start, max_period >←
          findMaxPeriod(PASS, STATUS, i, j)
15:       cost ← T_c(C_{s_cur})
16:       overhead ← setDMABusy(PASS, STATUS, cost, max_start, j)
17:       code_overlay ← code_overlay + overhead
18:       mem_state[r_cur] ← s_cur
19:    end if
20: end for
21: return code_overlay
```

Algorithm 8: $calCodeOverlayDeepPre(G, PASS, STATUS, < V, R >, < V, S >)$

the ith actor execution in the given $PASS$ on the actor's j^{th} outgoing edge is alive at time interval k. The current data buffer usage buf_mem is calculated by $calBuf(PASS, LIFE)$. Function $calBuf(PASS, LIFE)$ iterates through a given $PASS$ and calculates the buffer usage of each time interval k by summing up all data tokens that are alive at k. The maximum data buffer usage among all time intervals is returned as the data buffer usage of the given $PASS$.

Line 6 in Algorithm 9 initializes data overlay overhead to be 0. Starting from Line 7, we enter a while loop where we keep on collapsing two regions or applying data overlay until the total region size and data buffer fits into the

1: Initialize actor to region assignments $< V, R >$, as each actor occupies a separate region.
2: Construct IF table entry for each region pair $< (r_i, r_j), Integer >$, where $r_i, r_j \in R, i < j$
3: $region_mem \leftarrow \sum_{r \in R} C_r$
4: Initialize life time of all data segments $LIFE$
5: $buf_mem \leftarrow calBuf(PASS, LIFE)$
6: $data_overhead \leftarrow 0$
7: while $region_mem + buf_mem > C_p$ and $!(|R| = 1$ and $buf_mem = BUF_MIN)$ do
8:     if $buf_mem = BUF_MIN$ then
9:         $do_weight \leftarrow +\infty$, $co_weight \leftarrow 0$
10:     else if $|R| = 1$ then
11:         $co_weight \leftarrow +\infty$, $do_weight \leftarrow 0$
12:     else
13:         $do_weight \leftarrow \Delta_{t_do}/\Delta_{m_do}$, $co_weight \leftarrow \Delta_{t_co}/\Delta_{m_co}$
14:     end if
15:     if $do_weight < co_weight$ then
16:         $< buf_mem, overhead > \leftarrow DataOverlay(PASS, STATUS, LIFE)$
17:         $data_overhead \leftarrow data_overhead + overhead$
18:     else
19:         Collapse region pair $< r_i, r_j >$ with minimum IF. Update $< V, R >$, and IF table.
20:         $region_mem \leftarrow \sum_{r \in R} C_r$
21:     end if
22: end while
23: return $<< V, R >, data_overhead >$

Algorithm 9: $RegionAssignmentAndDataOverlay(G, PASS, STATUS)$

local SPM, or both operations fail ($|R| = 1$ and $buf_mem = BUF_MIN$). BUF_MIN is the minimum buffer usage required even with data overlay and it is given by the maximum buffer usage of each actor, including input buffers and output buffers. In each iteration of the while loop, we calculate the data overlay weight factor do_weight and the code overlay weight factor co_weight. The weight factors indicate the code or data overlay overhead for each unit of memory requirement reduction. If buf_mem reaches BUF_MIN, we set do_weight to be $+\infty$ and co_weight to be 0, Line 9.

Else if there is only one region left, we set co_weight to be $+\infty$ and do_weight to be 0, Line 11. Otherwise we calculate the data overlay overhead Δ_{t_do} and memory saving Δ_{m_do} for applying data overlay. Correspondingly we also calculate the code overlay overhead Δ_{t_co} and memory saving Δ_{m_co} for collapsing two regions. Δ_{t_do} is calculated by applying Algorithm 10 (discussed in the following paragraph). We can calculate Δ_{m_do} by simply taking the difference of data buffer usage before and after applying data overlay. Note that since the DMA engine $STATUS$ and data segment life time $LIFE$ are modified by Algorithm 10, only their copies are passed in at this step (the original copies of $STATUS$ and $LIFE$ are modified when we realize data overlay and code pre-fetching optimizations as discussed in Algorithm 9, Line 16 and Algorithm 11, Line 11 that is discussed later). For calculating Δ_{t_co} and Δ_{m_co}, we collapse the region pair $< r_i, r_j >$ with minimum IF and calculate the code overlay overhead before and after. Again, $< V, R >$, IF, and $STATUS$ are modified by this procedure. We pass in their copies instead of references. Based on the calculated do_weight and co_weight, we either perform data overlay (Algorithm 10) and update buf_mem, Line 16-17, or collapse two regions and update $region_mem$, Line 19-20. Upon termination of the while loop, We return the actor to region mapping $< V, R >$ and the total data overlay overhead $data_overhead$.

The algorithm for data overlay is provided in Algorithm 10. Given the current $PASS$, DMA engine status, and the life time of each data segment, Algorithm 10 iteratively introduces data overlay until a memory reduction is achieved or BUF_MIN is reached. In the algorithm, the buffer usage before data overlay is stored to old_buf. cur_buf is initialized to old_buf and $overhead$ is initialized to 0. At each iteration of the while loop, we identify

1: $old_buf \leftarrow calBuf(PASS, LIFE)$
2: $cur_buf \leftarrow old_buf$, $overhead \leftarrow 0$
3: while $old_buf \neq BUF_MIN$ and $cur_buf = old_buf$ do
4: Find time interval $k \in [0, |PASS| - 1]$ with the largest data buffer usage
5: $< i, j > \leftarrow findToken(PASS, STATUS, LIFE, k)$
6: $cost \leftarrow T_c(getDataSegment(PASS, i, j))$
7: $overhead \leftarrow overhead + setDMABusy(PASS, STATUS, cost, i, k)$
8: $overhead \leftarrow overhead + setDMABusy(PASS, STATUS, cost, k, j)$
9: Update $LIFE$ for $< i, j >$
10: $cur_buf \leftarrow calBuf(PASS, LIFE)$
11: end while
12: return $< cur_buf, overhead >$

Algorithm 10: $DataOverlay(PASS, STATUS, LIFE)$

time interval k with the largest data buffer usage, Line 4. Then for all data segments that are alive at k, we find the data segment $< i, j >$ (the data segment that is produced on the jth outgoing edge of the ith actor execution in the given $PASS$) that if overlayed will result in the smallest overhead, Line 5. The overhead of overlaying a data segment $< i, j >$ at time interval k is given by

$$overhead = max(0, cost - backward_period) + max(0, cost - forward_period)$$
$$\text{where } cost = T_c(getDataSegment(PASS, i, j)). \tag{3.2}$$

In the above equation, the $backward_period$ and $forward_period$ denote the consecutive DMA engine idle periods that can be used for pushing data segment backward to the off-chip memory and bringing it forward to the local SPM. They are calculated using Algorithm 6. $getDataSegment$ returns the data segment size that is produced on the jth edge of the ith actor execution in the given PASS. $T_c(getDataSegment\ (PASS, i, j))$ calculates the DMA cost for transferring data segment $< i, j >$ between the local SPM and the off-chip

1: Initialize $min_overlay \leftarrow +\infty$
2: Initialize $PASS \leftarrow MinBufferScheduling(G)$
3: repeat
4: Initialize $STATUS$ to be idle for every time interval
5: /* Perform actor to region assignment and data overlay */
6: $<< V, R >, do_overhead >\leftarrow$
$RegionAssignmentAndDataOverlay(G, PASS, STATUS)$
7: if $\sum_{r \in R} C_r \leq code_mem$ then
8: /* Perform actor to segment assignment */
9: $< V, S >\leftarrow Segmentation(G, PASS, STATUS, < V, R >)$
10: /* Calculate current code overlay overhead */
11: $cur_overlay \leftarrow calCodeOverlayDeepPre(G, PASS, STATUS, < V, R >, < V, S >)$
12: $cur_overlay \leftarrow cur_overlay + do_overlay$
13: if $cur_overlay < min_overlay$ then
14: $min_overlay \leftarrow cur_overlay$
15: $solution \leftarrow clone(G, PASS, < V, R >, < V, S >)$
16: end if
17: end if
18: until $collapseTwoExecs(PASS) = false$
19: return $solution$

Algorithm 11: $MinOverlaySchedulingOptimized(G, P)$

main memory. Then we overlay token $< i, j >$, Line 6-8, update its life time Line 9, and calculate the buffer usage after overlaying $< i, j >$, Line 10.

The overall minimum overlay scheduling algorithm after incorporating deep pre-fetching and data overlay is given in Algorithm 11. Compared to Algorithm 2, we added the initialization of DMA engine status in Line 4. The original $RegionAssignment$ is replaced with $RegionAssignmentAndDataOverlay$ and the $cur_overlay$ is the sum of code overlay overhead with deep pre-fetching[6] and data overlay overhead, Line 12. After incorporating data overlay operation, the complexity of $RegionAssignment\ AndDataOverlay$ becomes

[6] In Algorithm 11, every call to $calCodeOverlay$ is replaced with $calCodeOverlayDeepPre$, including the $Segmentation$ subroutine. It is why $STATUS$ is also passed in as a parameter to $Segmentation$ in Algorithm 11.

Table 3.4: Benchmark Specifications

Benchmarks	Total Code Size (Bytes)	Minimum Buffer (Bytes)
Beamformer	12356	2272
Bitonicsort	576	256
Channelvocoder	22996	6800
DCT	2673	1024
DES	2256	1024
FFT	2318	2048
Filterbank	41879	416
Fmradio	34285	204
Serpentfull	10056	3584
TDE	3226	6144
Average	13262	2377

$O(n^4)$. The complexity of *Segmentation* is $O(n^3)$. They are both nested inside the loop of SDF scheduling, which is $O(n)$. Therefore the overall algorithm complexity becomes $O(n^5)$.

3.9 Experimental Results

3.9.1 Experimental setup

We evaluated the efficiency of our heuristic by compiling ten benchmarks from the StreamIt compiler 2.1.1 [57] onto one SPE of the IBM Cell BE. The source code of each benchmark is delivered with the StreamIt compiler and a brief discussion can be found in [56]. Table 3.4 first column provides the benchmark names. The second and third columns provide the code size and minimum buffer usage of each benchmark. The last row of the table computes the average values for each column. We implemented our techniques as an optimization pass in the StreamIt compiler that operates on the intermediate representation (IR) of a stream application. Each benchmark is cross-compiled on our PC for executing on one SPE.

3.9.2 Comparison of 3-stage ILP and heuristic with minimum buffer scheduling

Figure 6.3 compares the overlay overhead of our 3-stage ILP and heuristic with a minimum buffer scheduling approach. The SPM size is set to be 8K in the experimental set up. The 8K SPM is selected such that for most of the benchmarks it is enough to hold the minimum buffer usage but still not so large that all code and data fit into it. The performance results of minimum buffer scheduling, 3-stage ILP with/without code pre-fetching, and heuristic with/without code pre-fetching, deep pre-fetching, and data overlay are provided in Figure 3.9. The x-axis gives us the benchmark names and the y-axis provides the overlay overhead for each benchmark normalized to the minimum buffer scheduling results. BP, DP, and DO are short forms for basic pre-fetching, deep pre-fetching, and data overlay respectively.

For benchmarks *Bitonicsort*, *DCT*, *DES* and *FFT*, there is no overlay overhead in all techniques. The reason is that for these four benchmarks, the buffer usage and total code size all fit into the 8K SPM with a minimum buffer schedule. For the remaining six benchmarks, our heuristic approach without code pre-fetching achieves an overlay overhead reduction of 50% compared with the minimum buffer scheduling. The performance is within 5% compared with 3-stage ILP approach which takes exponential time to run. With basic pre-fetching our heuristic generates no overlay overhead for most of the benchmarks. This is due to the fact that with basic pre-fetching, most of the DMA transfers are hidden by actor executions. Compared with minimum buffer scheduling, the average overlay overhead reduction is around 97% with

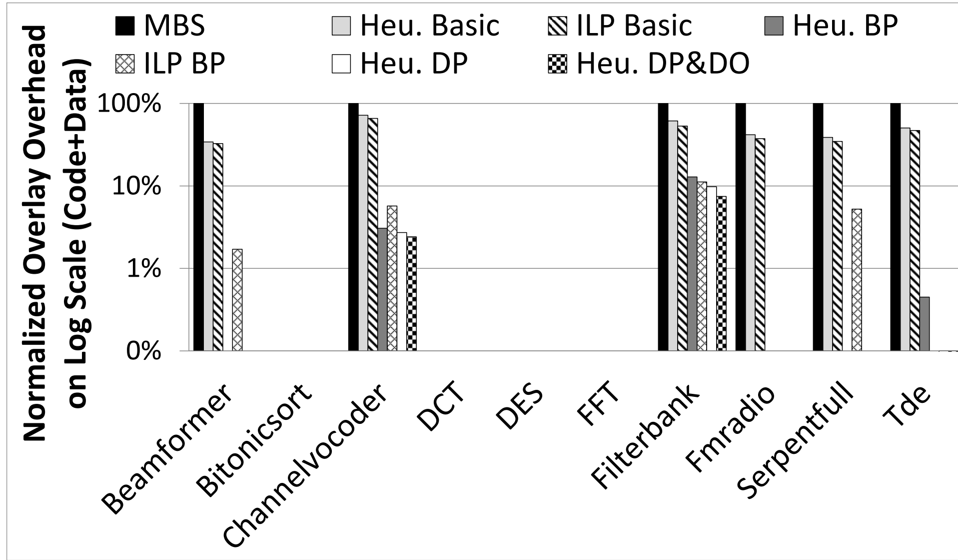


Figure 3.9: Our 3-stage ILP and heuristic approaches compared with minimum buffer scheduling.

basic pre-fetching. For these benchmarks that still impose overlay overhead after basic pre-fetching, deep pre-fetching optimization of our heuristic results in an average overlay overhead reduction of 23% compared with results of basic pre-fetching. Further introducing of data overlay gives us a performance improvement of 19% compared with results without data overlay. The average algorithm run time of our heuristic is 84 seconds and the average algorithm run time of the 3-stage ILP is 25926 seconds. In other words, compared with the previous 3-stage ILP approach, our heuristic runs more than 300 times faster. With optimizations of deep pre-fetching and data overlay, we also achieved better performance than the ILP approach. The 3-stage ILP approach serves as a baseline optimization that provides reference performance results. The improved performance is resulted from the fact that with prefetching, a PASS not only affects the code memory and number of actor switches, but also the

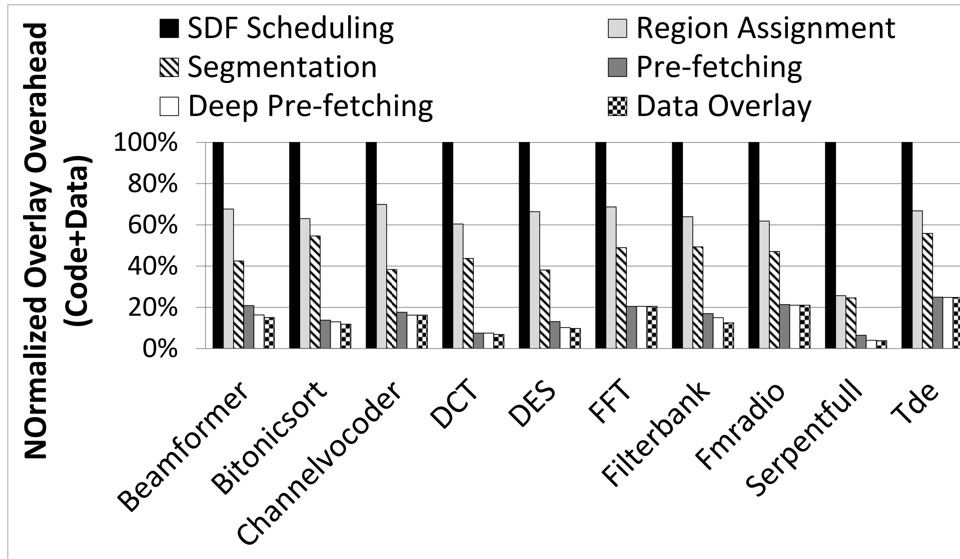


Figure 3.10: Impact of each optimization in our heuristic approach.

actor execution that can be utilized to overlap with DMA transfers. Our heuristic is able to efficiently perform this trade-off by evolving its PASS from a minimum buffer schedule to a minimum actor switching schedule.

3.9.3 Impact of each optimization

We evaluate the impact of each optimization in our heuristic in this section. Since our heuristic algorithm is much faster than the previous 3-stage ILP approach, we were able to run our experiments through a series of different SPM sizes. The SPM sizes are determined in the following way. We first calculated $memMIN$ and $memMAX$ for each benchmark such that there is no feasible solution for any SPM smaller than $memMIN$ (without data overlay) and there is no overlay overhead for any SPM larger than $memMAX$. We iterate the SPM size from $memMIN$ to $memMAX$ with a step size of $(memMAX - memMIN)/STEPS$ ($STEPS$ is set to 10 in our experiments).

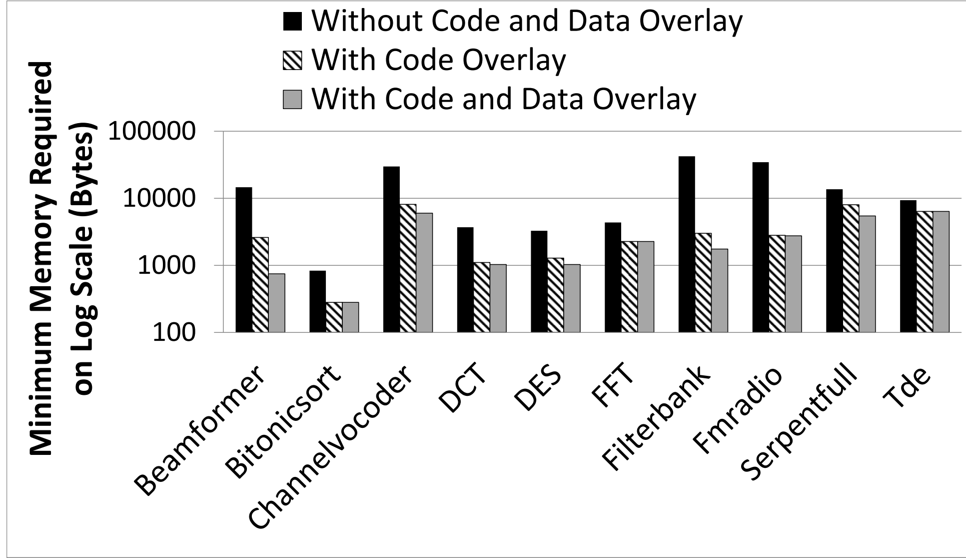


Figure 3.11: Memory usage comparison.

The calculation of $memMIN$ and $memMAX$ is given by,

$$memMIN = getMem(MBS) + max_{v \in V} C_v$$

$$memMAX = getMem(MBS) + \sum_{v \in V} C_v$$

In the above equation, $getMem(MBS)$ calculates the buffer usage of a minimum buffer schedule and $max_{v \in V} C_v$ calculates the largest code size among all actors. Figure 3.10 shows the average overlay overhead after each optimization. The overlay overhead is normalized to the result at the SDF scheduling stage. Observed from Figure 3.10, the region assignment delivers the most significant performance improvement at an average of 48%. This is due to the fact that without region assignment, all actors are assigned to the same region. The local SPM is not fully utilized and all actors are overlayed with each other. Pre-fetching gives us the next most significant performance improvement at around 28%. With pre-fetching, most of the

DMA transfers are hidden by actor executions. In our heuristic, segmentation explores opportunities to combine actors in the same region to reduce the actual number of DMA transfers. It delivers an average performance gain of 17%. Finally, deep pre-fetching and data overlay each give us an additional overlay reduction of 1%.

The performance improvements of deep pre-fetching and data overlay are not as significant as other optimizations because the following reasons. Code overlay overhead has been almost optimized away with basic pre-fetching. Also when the SPM size is very small compared with the total code and data size, we only have a limited number of regions and the opportunities for implementing deep pre-fetching is highly restricted. For example if there are only two regions, pre-fetching and deep pre-fetching in fact behave identical to each other. Data overlay optimization has a potential to reduce the data buffer usage of a schedule. However, it occupies DMA engine for transferring data, thus could potentially impact the code pre-fetching optimization. In our heuristic, data overlay optimization is more significant in terms of improving the feasibility. In Figure 3.11, we provide the minimum memory requirement for a program to be executable on an SPM. The x-axis provides us with the benchmark names and the y-axis shows the memory required for a feasible solution to exist. We experimented under three configurations, heuristic without code overlay, heuristic with code overlay, and heuristic with code and data overlay. The average memory required for the twelve benchmarks without code and data overlay is 15640 bytes. With code overlay, the average memory required drops down to 3588 bytes. Adding data overlay further reduces the average memory requirement to 2774 bytes, a more than 20% improvement compared with solutions without data overlay.

3.9.4 Impact of SPM size

In this section, we show the code overlay overhead of each benchmark under various SPM sizes. Ten different SPM configurations were taken for each benchmark as discussed in Section 3.9.3. Figure 3.12 and Figure 3.13 present the performance results from the first five and second five benchmarks, respectively. The x-axis in each figure provides the SPM steps and the y-axis provides the normalized overlay overhead of our heuristic with deep pre-fetching and data overlay. The performance results at each SPM step is normalized to the overlay overhead at step 0. As we iterate from $memMIN$ to $memMAX$, the overlay overhead went down dramatically for all the benchmarks we experimented with. There are three benchmarks, BeamFormer, ChannelVocoder, and SerpentFull that impose no overlay overhead even before the SPM size reaches $memMAX$. This is due to data overlay optimization. Note that for benchmark DES step 1, 2, 4, 9 and benchmark Tde step 5, although the SPM size was increased, our heuristic generates almost the same overlay overhead as the previous step. This is due to the fact that we terminate the process of region assignment and data overlay in our heuristic as soon as a feasible solution is found. However, there could be scenarios where by further collapsing two regions and thus resulting in more actors in certain regions, segmentation could benefit so much that a better solution is generated. Our heuristic can be improved at this point at the cost of increasing the algorithm complexity from $O(n^5)$ to $O(n^8)$. For the interest of algorithm run time, we left segmentation out of region assignment and data overlay optimization in our SDF scheduling heuristic.

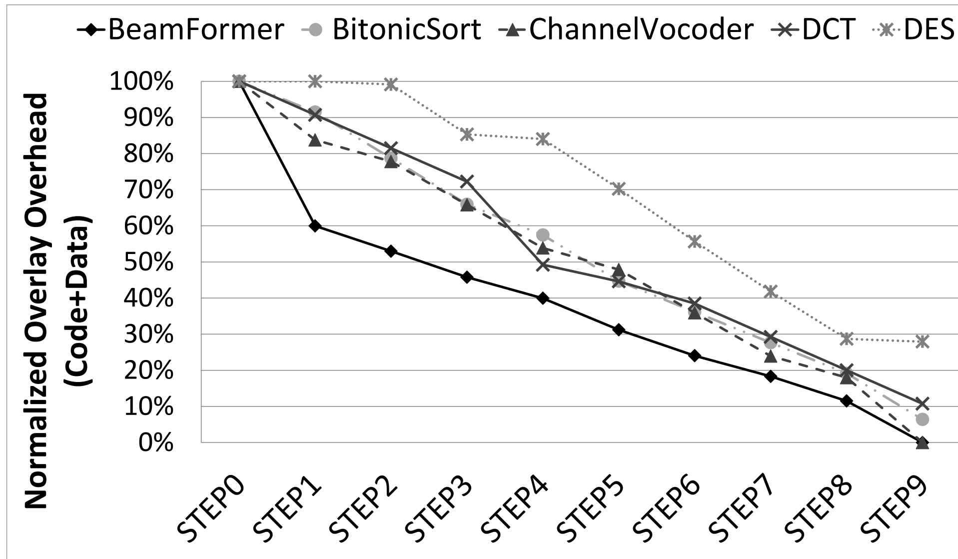


Figure 3.12: SPM size variation (1st set).

3.9.5 Code overlay evolution

In this section, we demonstrate the evolution of code overlay overhead with SDF scheduling. The SPM size is again set to be 8K and we plot the code overlay overhead as the PASS gradually evolves from a minimum buffer schedule to minimum actor switch schedule. Figure 3.14 shows the performance results normalized to the overlay overhead from the initial stage where a minimum buffer schedule was adopted. For benchmarks BitonicSort, DCT, DES, and FFT, there is no overlay overhead and our heuristic terminates immediately. Figure 3.14 plots the overlay overhead evolution for the remaining six benchmarks. We calculate the code overlay overhead $cur_overlay$ for each schedule being generated. If it is smaller than the recorded code overlay overhead, then we update $min_overlay$ with $cur_overlay$. $min_overlay$ maintains the

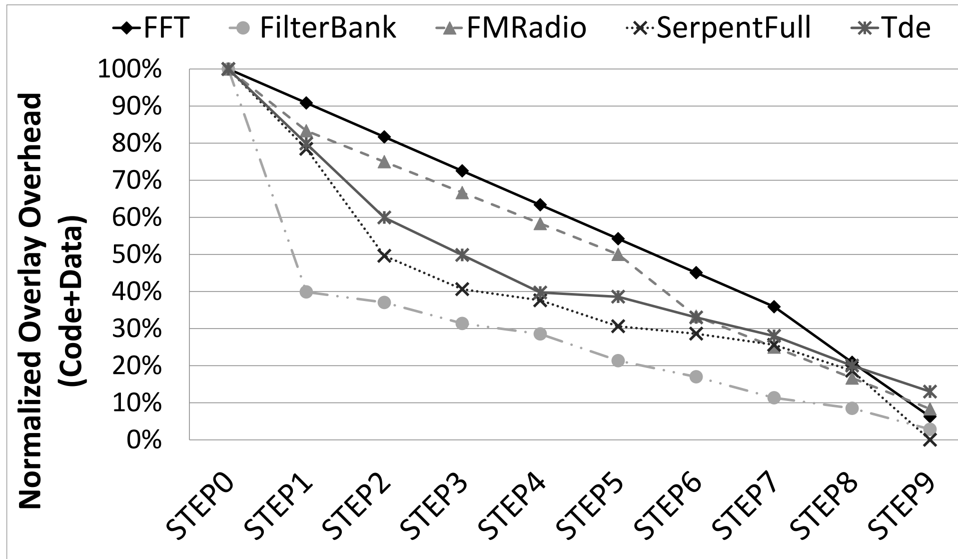


Figure 3.13: SPM size variation (2nd set).

best solution obtained so far. As observed from Figure 3.14, the code overlay overhead gradually reduces as the schedule evolves. It reaches a steady-point after a certain number of steps where a best solution is recorded. Benchmark SerpentFull achieved a very low overlay overhead with its minimum buffer schedule. Therefore its overlay overhead was not updated until an improved schedule is achieved much later.

3.9.6 Impact of scaling DMA cost

In this section, we examine the impact of scaling DMA transfer cost to simulate the scenarios where there are multiple stream applications running on SPM based multi-core architecture. With a fixed on-chip bandwidth multi-core architecture, as we add more and more cores the DMA transfer cost becomes larger and larger. In this experiment, we scaled the DMA cost by 2, 4, 8, and

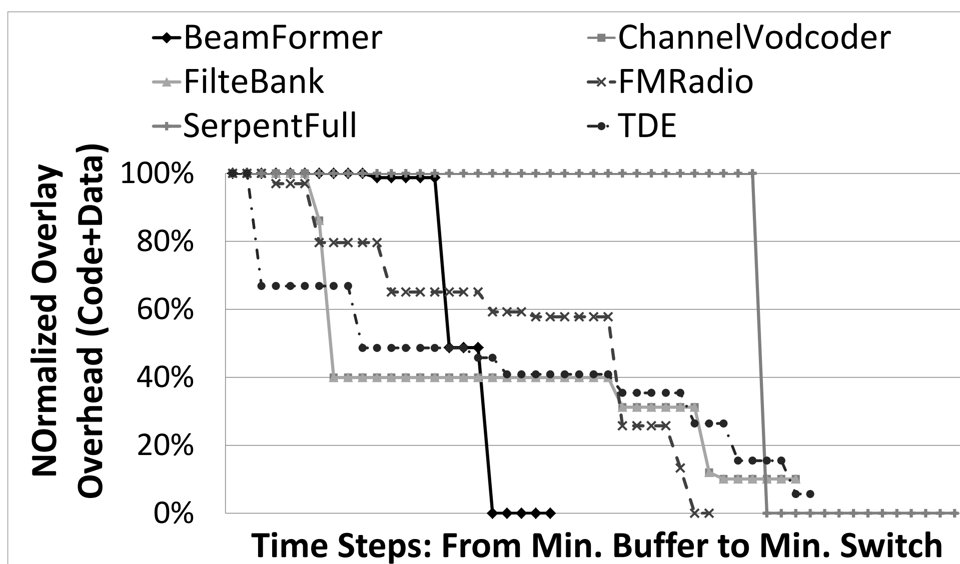


Figure 3.14: Overlay cost evolution with SDF scheduling.

16. The SPM is set to be 8K. In Figure 3.15, the x-axis shows the six out of ten benchmarks that generate overlay overhead. The y-axis shows the overlay overhead normalized to overlay overhead without scaling. For benchmarks BeamFormer, FMRadio, and SerpentFull, there is no overlay overhead at scale factor of 1 with deep pre-fetching and data overlay implemented. Therefore their overlay overhead is normalized to the performance at scale factor 2. As observed from Figure 3.15, the overlay overhead increases much faster than the DMA overhead. This because both deep pre-fetching and data overlay optimizations rely on using actor executions to overlap with DMA transfers. When the DMA cost scales up, not only the costs for transferring code and data scales up, it also greatly impacts the deep pre-fetching and data overlay optimization.

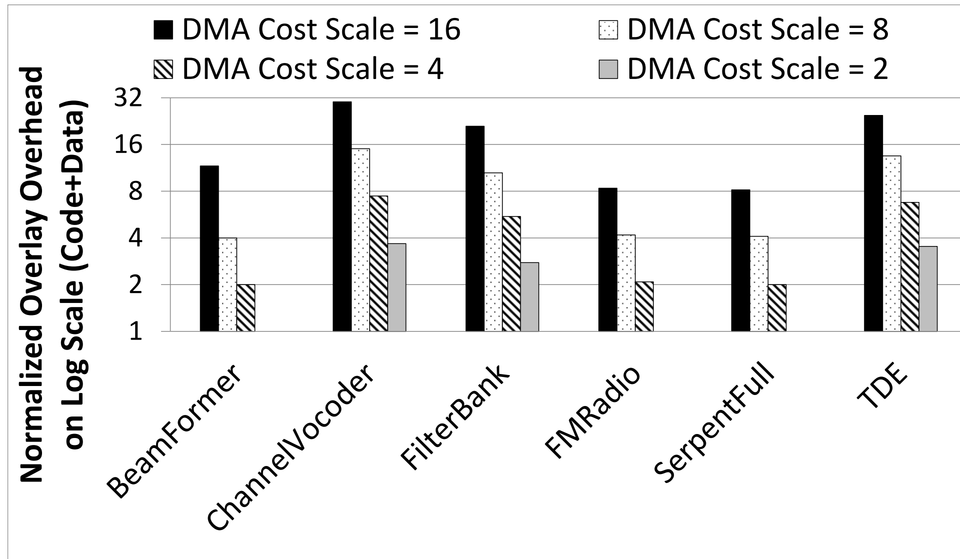


Figure 3.15: Impacts of scaling DMA overhead.

3.9.7 Impact of scaling code size and run time

In this section, we simulate scheduling larger applications by scaling the code size and run time of each actor in the original benchmark. The SPM size is set to be 8K. In Figure 3.16, the x-axis provides the benchmarks and the y-axis provides the normalized overlay overhead with deep pre-fetching and data overlay. The overlay overhead is normalized to the overlay overhead without scaling. When there is no feasible solution, the overlay overhead is infinite and we only show it up until 32. We examine scale factors of 2, 4, 8, and 16. Observed from Figure 3.16, several benchmarks become infeasible after a few scaling steps, for example ChannelVocoder and SerpentFull at scaling factor 2, FilterBank and FMRadio at scaling factor 4, and Tde at scaling factor 16. The behavior is resulting from the fact that for these benchmarks, the data buffer usage is close to the SPM size and their actor code sizes are

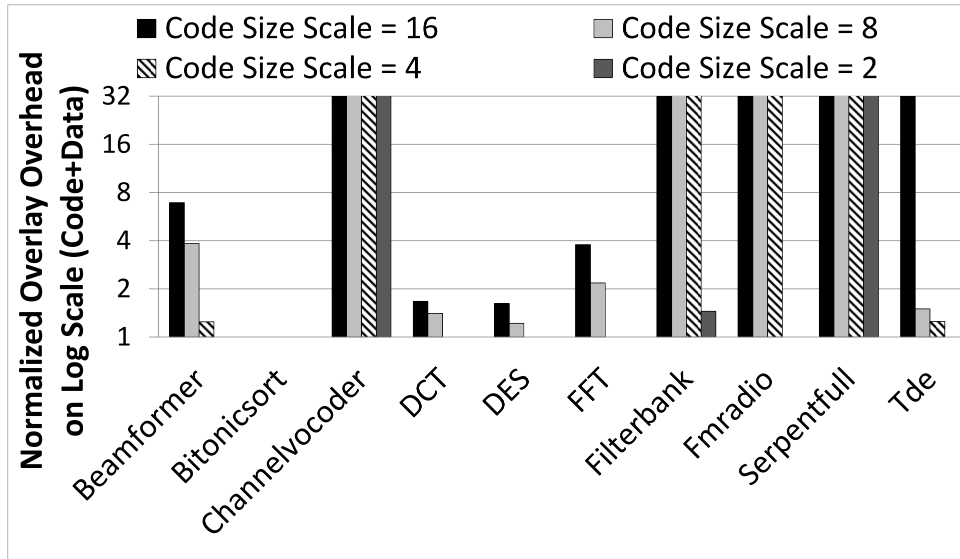


Figure 3.16: Impacts of scaling code size and run time.

comparably large. After scaling, the largest actor code size cannot fit into the available code memory. For BeamFormer, BitonicSort, DCT, DES, and Tde (the first several scaling steps), the overlay overhead increase is less than the scaling factor. This is because when the run time of an actor is scaled, we have a better chance to utilize actor executions to overlap with code and data overlay. The fact that the DMA base cost does not change after the code size is scaled also contributes to this behavior.

3.10 Summary

We presented both a 3-stage ILP formulation and a fast heuristic for scheduling SDF specifications onto SPM based architectures with the objective of latency minimization. We also presented extensions to our basic approaches with a code pre-fetching optimization. Deep pre-fetching and data overlay

were also investigated in our heuristic approach. The experimental results demonstrate that our ILP approaches are able to efficiently explore various design trade-offs and generate high quality solutions. Our ILP approaches suffers from long algorithm run time for large inputs. Our heuristic is able to achieve comparable results with our 3-stage ILP approach within a matter of seconds. We evaluated the efficiency of our heuristic approach with different SPM configurations, schedules and optimizations. The final results show that our heuristic approach is efficient and fast in all cases.

Chapter 4

SCHEDULING OF STREAM PROGRAMS ON SPM BASED MULTICORE PROCESSORS THROUGH FUSION AND FISSION

The stream processing characteristics of many embedded system applications in multimedia and networking domains have led to the advent of stream based programming formats. Several multicore processors aimed at embedded domains incorporate scratchpad memories (SPM) due to their superior power consumption characteristics. This chapter addresses the problem of compiling stream programs onto multi-core processors that incorporate SPM. Performance optimization on SPM based processors requires effective schemes for software based management of code and/or data overlay. In the context of our problem instance the code overlay scheme impacts both the stream element to core mapping and memory available for inter-processor communication. The chapter presents an optimal integer linear programming (ILP) formulation and heuristic approach that effectively exploit the SPM to maximize the throughput of stream programs when mapped to multicore processors. The experimental results demonstrate the effectiveness of the proposed techniques by compiling StreamIt based benchmark applications on the IBM Cell processor and comparing the performance with an existing approach.

4.1 Motivation

Increasing performance requirements of many embedded system applications has led to the advent of multicore processor architectures. The Intel IXP series [40], Sun Niagara [47], IBM Cell Broadband Engine (CE) [64], Nvidia

GEForce series [43], Intel Larrabee [68], Tilera Tile64 [72] are all instances of multicore processors aimed at embedded domains. In addition to multiple homogeneous cores, these processors also incorporate specialized architectural features to achieve high performance in a reasonable power envelop. In terms of the memory constructs scratchpad memory (SPM) has been incorporated in several processors due to its lower power consumption requirements in comparison to traditional caches.

The architectural innovations have brought forth the challenge of programming these novel embedded processors. At present a typical design flow does not include a compiler that can effectively parallelize the application and achieve maximal performance. In the absence of such a framework, the designer is required to manually split the application into multiple threads which she assigns to individual cores. Additionally, the designer is also required to include code segments for inter-core communication. The processor vendor does supply traditional compiler tool chains that can take the threads assigned to individual cores and generate assembly. Thus, performance optimization on multicore embedded processors is a lengthy manual task which leads to inferior implementations.

In recent years we have seen the emergence of stream programming languages that capture the inherent streaming characteristics of many embedded system applications in the multimedia, network processing and gaming domains. Brook [13], CUDA [60], Baker [21], Sh [54] and StreamIt [71] are some of the stream based programming languages that have been proposed. Stream based formats effectively capture the spatial and temporal parallelism in an application, and therefore are particularly suited for programming on

multicore processors. However, the challenges associated with compiling traditional multi-threaded programs on multicore processors (as described above) also hold true for stream languages.

This chapter addresses the problem of compiling and optimizing stream programs on embedded multicore processors that incorporate SPM. In particular we consider the compilation of StreamIt programs on the IBM Cell BE[1].

4.2 Previous Work

Previous research has addressed mapping of synchronous dataflow (SDF) specifications on heterogeneous multicore processors [65]. More recently Chen et al. [21] and Ostler et al. [61] proposed techniques for mapping stream program based specifications on network processors. Liao et al. [77] proposed parallelization schemes for the Brook language on general purpose multicore processors. In contrast to these approaches our research is focussed on embedded multicore processors that have SPMs. There has been recent research that have proposed a dynamic scheduler [12] and compiler optimizations for single threaded code [46] for Cell BE. In contrast our research is focussed on static code optimizations and compiling stream programs on Cell BE. Gordon et al. [32] proposed an approach that utilized fusion/fission operators for maximizing the performance of StreamIt programs when compiled for the RAW architecture. In addition to utilizing similar operators, we are also concerned with addressing the trade-off between computation time, code overlay and communication overheads imposed by the SPMs.

The work that comes closest to ours is by Kudlur et al. [48] that

[1]The discussion of StreamIt and IBM Cell BE are provided globally in Chapter 1

also considered the compilation of StreamIt programs on the Cell BE. They proposed an integer linear programming (ILP) formulation for the problem. However, their ILP formulation did not consider memory constraints of the SPM and consequently the code overlay costs associated with a mapping. Further, they also did not model inter-core communication (direct memory access or DMA) overheads in the ILP and assumed that all such overheads could be hidden by computation. However, in several practical instances the SPM code overlay overhead is significant. Further, in many instances the communication overheads cannot be hidden by computation time. Ignoring these overheads leads to inferior designs. Our work overcomes these limitations and makes the following contributions:

1. An optimal ILP formulation for Compiling Stream programs on SPM equipped Multicore Processors (named as CSMP_{ilp}) that models both the code overlay and communication overheads.

2. A fast polynomial time heuristic (named as CSMP_{heu}) for the same problem that is able to achieve comparable results as the ILP formulation in a matter of seconds.

We establish the effectiveness of the proposed techniques by compiling StreamIt benchmark programs for Cell BE, and comparing the performance with an existing approach [48].

4.2.1 Problem description

The inputs to the problem consist of the architectural description of the target multicore processor and SDF based intermediate format that captures the

Table 4.1: Architecture and SDF Description

	Constant	Description
Arch.	C_p	Local memory size of the processor
	L_p	1 (true) if overlay overhead exists for the local mem.
	T_{init}	Lowest DMA transfer overhead
	D_{init}	Largest DMA size that can be transferred with T_{init}
	T_{slope}	Rate of increase of DMA overhead beyond D_{init}
SDF	C_f	Size of code and local data for filter f
	S_f	1 (true) if f is non-fissable (stateful)
	τ_f	Running time of filter f
	f_{pe}	Producer of FIFO e
	f_{ce}	Consumer of FIFO e
	C_e	Data produced to/consumed from e

stream program. The cores in the architecture are described by a set P where each $p \in P$ is given by a tuple $p\langle C_p, L_p, T_{init}, D_{init}, T_{slope}\rangle$ as described in Table 4.1. During the characterization of the Cell BE it was found that the DMA overhead is constant at $2.1\mu s$ ($= T_{init}$) below a block size of $1KB$ ($= D_{init}$). Beyond $1KB$ the DMA transfer overhead was found to increase at $0.075\mu s/KB$ ($= T_{slope}$).

The stream program is described by a SDF $G\langle F, E\rangle$ where F is the set of filters, and E is the set of FIFOs between filters. Each $f \in F$ is given by a tuple $f\langle C_f, S_f, \tau_f\rangle$ as described in Table 4.1. Each FIFO $e \in E$ is given by $\langle f_{pe}, f_{ce}, C_e\rangle$ also described in Table 4.1. In the SDF description we assume that each filter executes only once. Consequently, each FIFO has only one data size (C_e) associated with it. It is quite straight forward to transform from the traditional SDF description [49] to our intermediate format.

The objective is to seek a mapping of the SDF on the multicore architecture such that the throughput of the design is maximized subject to

the memory constraints. The problem is quite complex as it involves several design trade-offs.

The throughput of a SDF on multicore processor can be optimized by utilizing fusion and fission operators [61][48]. For example, consider a linear SDF G' with three filters A, B, and C with execution times $50ms$, $50ms$ and $200ms$, respectively. If G' is to executed on a processor with 3 cores, then fusing (or merging) A and B and executing them sequentially on core 1, and a fission (or replication) of C on cores 2 and 3 maximizes the throughput. However, for large SDFs fusion cannot be applied indiscriminately as cores have a memory restriction beyond which an additional code overlay overhead adversely impacts performance. Thus, there is a trade-off between the benefit of fusion (as it frees up cores for slower filters) and the resulting overlay overhead. Further, fission cannot be applied in the case of a stateful filter.

In the discussion thus far we have ignored the communication overhead for a FIFO whose producer and consumer filters are mapped to different cores. As the DMAs can be launched in a non-blocking manner, it is possible to amortize the communication overhead with filter execution. However, this scheme (also known as double buffering) requires more memory space. The FIFO is assigned two locations on the producer and consumer core's memory, respectively. While the producer and consumer read data from one location in their respective FIFO buffers, a DMA operation transfers data between the other two FIFO buffers. As the FIFO is assigned on the same memory as the code, there is a trade-off between the memory usage of the two, and the resulting performance.

Finally, in the above discussion we assumed that the communication overhead can be effectively hidden by double buffering. However, it may not be the case if the communication overhead itself is too large. In addition to considering overlay overheads, the fusion/fission operators and mappings must also take into account the resulting communication overheads.

In the following sections we describe an ILP formulation and heuristic approach that are both able to effectively address the various design trade-offs and generate high quality solutions.

4.3 Integer Linear Programming Approach

We describe an ILP approach (called CSMP$_{i}lp$) for compiling stream programs on multicore processors incorporated with SPMs. In our ILP the fission and fusion operators are implemented by first assigning the filters to batches, and then the batches to processors. In the mapping, each filter must be assigned to exactly one batch and each processor must be assigned one batch to execute. Figure 4.1.A provides an example of the mapping with 6 filters and 3 processors and Figure 4.1.B sketches the steady-state execution. Ostler el al. [61] proved that such a batching strategy can generate optimum solutions for stream programs. The base and derived variables of the ILP are described in Table 4.2.

4.3.1 Constraints

The constraints of our ILP are described below. Some of the constraints are identical to the ILP by Ostler et al. [61], however for the integrity of this dissertation we provide each constraint in detail.

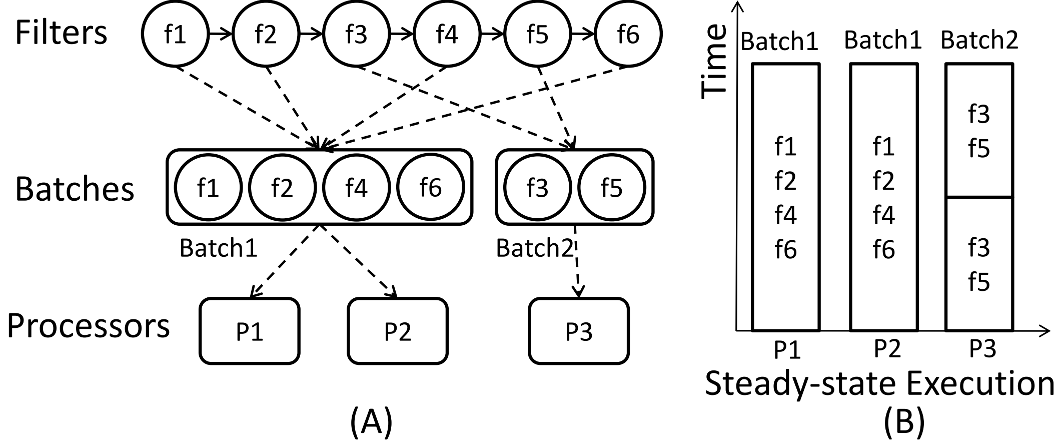


Figure 4.1: (A) filter-batch, and batch-processor mapping; (B) steady-state

- File read and write: The source and sink filters are assumed to be file read and write operations, respectively that can only execute on the PPE. Therefore, we map them to the first batch which is set to $nonfissable$, and map the batch to the PPE (proc 1).

$$a_{11} = 1,\ a_{|F|1} = 1, b_{11} = 1,\ S_1 = 1,\ n_{11} = 1$$

- Filter to batch assignment: Each filter is mapped to one and only one batch.

$$\forall f \in F : \sum_{b \in B} a_{fb} = 1$$

- Batch to processor assignment: If batch b has n copies, then exactly n processors must be assigned to execute b.

$$\forall b \in B : \sum_{p \in P} b_{bp} = \sum_{n=1}^{|P|} n_{bn} * n$$

- Processor utilization: Each processor must be assigned exactly one batch to execute.

$$\forall p \in P : \sum_{b \in B} b_{bp} = 1$$

Table 4.2: Base and Derived Variables

	Var	Type	Description
Base Var.	a_{fb}	0/1	filter f belongs batch b
	b_{bp}	0/1	batch b is assigned to processor p
	n_{bn}	0/1	number of replicated copies of batch b
	i_{fn}	0/1	iteration number of filter f
	p_f	0/1	1 if filter f always in SPM
Derived Var.	C'_f	real	amount by which C_f exceeds $1KB$
	C'_e	real	amount by which C_e exceeds $1KB$
	P_{eb}	0/1	f_{pe} of e belongs to b
	C_{eb}	0/1	f_{ce} of e belongs to b
	M_{eb}	0/1	P_{eb} AND C_{eb}
	P'_{eb}	0/1	$P_{eb} - M_{eb}$
	C'_{eb}	0/1	$C_{eb} - M_{eb}$
	K_e	0/1	f_{pe} and f_{ce} belong to different batches
	I_e	int.	iteration difference of e's f_{pe} and f_{ce}
	IM_{eb}	int.	I_e AND M_{eb}
	IC'_{eb}	int.	I_e AND C'_{eb}
	X_{fb}	0/1	a_{fb} AND p_f
	Y_{fb}	0/1	f mapped to b, and b has multiple copies
	$C_p(max)$	real	1. code size of largest filter f mapped to p 2. f not present in SPM of p

- Batch utilization: A batch needs to be mapped to some processor to execute only when there is at least one filter being assigned to it.

$$\forall b \in B : \sum_{p \in P} b_{bp} * MAX_VAL \geq \sum_{f \in F} a_{fb}$$

$$\sum_{p \in P} b_{bp} \leq \sum_{f \in F} a_{fb} * MAX_VAL$$

- Stateful filter: If a batch consists of a non-fissable filter then it has only one copy.

$$\forall b \in B : \sum_{n=1}^{|P|} n_{bn} \leq 1$$
$$S_b * MAX_VAL \geq \sum_{f \in F} a_{fb} * S_f$$
$$S_b \leq \sum_{f \in F} a_{fb} * S_f$$
$$n_{b1} \geq S_b \ /* nofissable\ batch\ has\ 1\ copy */$$

MAX_VAL is some large value and S_b indicates whether batch b is nonfissable.

- Iteration assignment: Each filter runs at some iteration number. If the producer and consumer of an edge e are being assigned to different batches, then the producer runs at least 2 iteration numbers higher than the consumer.

$$\forall f \in F : \sum_{n \in N} i_{fn} = 1, \forall e \in E : I_e \geq 2 * K_e$$

- Buffer Usage: The buffer usage of batch b is calculated by its incoming $(I_e * C_e)$, outgoing $(2 * C_e)$ and internal $(I_e + 1) * C_e$ edges.

$$C_b(buf) := \sum_{e \in E} IC'_{eb} * C_e + \sum_{e \in E} 2 * P'_{eb} * C_e + \sum_{e \in E} (IM_{eb} + M_{eb}) * C_e$$

- Code overlay: The code overlay overhead is given by the fetch time for all the filters that are not present in the SPM.

$$\forall b \in B : \tau_b(overlay) := \sum_{f \in F} (a_{fb} - X_{fb}) * (T_{init} + T_{slope} * C'_f)$$

- Processor Memory: The sum of the buffer usage and the region for code overlay must be less than the processor local memory size.

$$\forall b \in B, p \in P : C_{bp} \leq b_{bp} * MAX_VAL$$

$$C_{bp} \geq C_b(buf) + C_{bp}(code) + (b_{bp} - 1) * MAX_VAL$$

$$C_{bp} \geq 0$$

$$C_p \geq \sum_{b \in B} C_{bp} + \sum_{f \in F} C_f * (1 - L_p)$$

where C_{bp} (code) is given by

$$C_{bp}(code) := \sum_{f \in F} (C_f * X_{fb} + C_p(max)) * L_p$$

4.3.2 Objective function

The execution time of a batch is given by the maximum of its computation (which includes code overlay overheads) and communication time. The effective execution time of a batch equals the execution time of that batch divided by the number of its copies. If a batch has multiple copies, we also introduce a fission overhead for additional *split/join* nodes and potentially more *peek* operations (captured by ϵ).

4.3.2.1 Computation cost

The computation cost of batch b is given by the sum of the computation cost of all the filters being assigned to it plus its overlay overhead.

$$\forall b \in B : \tau'_b := \sum_{f \in F} a_{fb} * \tau_f + \tau_b(overlay)$$

4.3.2.2 Communication cost

The communication cost of batch b is given by the time to fetch all its input data to the local memory in the steady-state execution.

$$\forall b \in B : \tau''_b := \sum_{e \in E} C'_{eb} * (T_{init} + T_{slope} * C'_e)$$

4.3.2.3 Execution cost

The execution cost of a batch is the maximum of its computation and communication times.

$$\forall b \in B : \tau_b \geq \tau'_b,\ \tau_b \geq \tau''_b$$

4.3.2.4 Effective execution cost

The effective execution cost is given by

$$\forall b \in B : \Gamma_b := \sum_{n=1}^{|P|} \frac{1}{n} * \tau_{bn} + \sum_{f \in F} Y_{fb} * (\epsilon + split_join_work)$$

where, τ_{bn} is given by

$$\begin{aligned} \forall b \in B, n \in [1, |P|] : & \tau_{bn} \leq n_{bn} * MAX_VAL \\ & \tau_{bn} \geq \tau_b + (n_{bn} - 1) * MAX_VAL \\ & \tau_{bn} \geq 0 \end{aligned}$$

4.3.2.5 Overall cost function

The objective function is to minimize effective execution time over all batches and thus, maximize the throughput.

$$\forall b \in B : \Gamma \geq \Gamma_b, \quad Minimize(\Gamma)$$

```
1:  B = initialize(G), B = B_f ∪ B_nf
2:  initialize_iteration(B)
3:  K = C(B_nf, B, P)
4:  B = B_f ∪ fuse_number(B_nf, K)
5:  B = fuse_number(B, |P|)
6:  C = cost(B, G, P)
7:  for i = |B_f| − 1 down to 1 do
8:      B′ = B ∪ fuse_number(B_f, i)
9:      B″ = fission_number(B′, |P|), C′ = cost(B″, G, P)
10:     if C′ < C then
11:         C = C′, B_s = B″
12:     end if
13: end for
14: return  B_s
```

Algorithm 12: $CSMP_{heu}(P, G)$

4.4 Heuristic Approach

In this section, we present a heuristic approach (called CSMP$_{heu}$) that can be utilized for compiling stream programs on SPM enhanced multicore processors. The heuristic involves iterative application of fusion/fission operators, and estimation of the performance of the resulting SDF. The performance estimation considers both overlay costs and communication costs. The overlay costs take into account the trade-off between the buffer requirement for communication and code memory. Although the application of fusion/fission operators is similar to Ostler et al. [61], our approach can address the design complexity introduced by the SPM.

The main routine of our approach is shown in Algorithm 12. The function initialize() assigns each filter $f \in G$ to a distinct batch $b \in B$. The non-fissable batches that include a non-fissable filter are denoted by B_{nf} ($B_f = B/B_{nf}$). initialize_iteration() assigns the iteration number of every filter. As each batch is assigned to a different processor the difference between

the iteration numbers of producer and consumer filters is set as 2. As explained earlier this enables overlapping of DMA transfers with computation. We next calculate $K = C(B_{nf}, B, P)$ (given below) which denotes the number of batches that non-fissable filters should be fused into. K is given by the product of number of cores and the ratio of the summation of run time of the non-fissable batches over the summation of the run time of all batches.

$$K := \left\lceil \frac{\sum_{f \in B_{nf}} \tau_f}{\sum_{f \in B} \tau_f} \times |P| \right\rceil$$

The function fuse_number(B_{nf},K) fuses the input set of batches (B_{nf}) into K distinct batches. The fusion operation considers all pairs of batches and fuses the pair which has the lowest total cost after the merge. Next we fuse the batches using function fuse_number($B, |P|$). Thus, the total number of batches are now equal to the number of cores. Let C denote the effective execution time of this design.

We next iteratively (within the for loop) explore different design alternatives that can improve upon the initial solution. In each iteration we first generate a solution by fusing fissionable batches. For example in the first iteration we fuse and reduce the number of fissionable batches by one. We then apply the fission operator that iterative replicates the slowest batches until the total number of batches are the same as the number of cores. We save the solution (B_s) that gives the lowest cost. At the end of the for loop we return the best solution.

The computational complexity is determined by the for loop. The computational complexity of the fuse operation is $O(n^2)$ as it considers all pairs of batches. The complexity of fuse_number() is $O(n^3)$, and consequently the complexity of the overall routine is $O(n^4)$. In the following we elaborate upon

the overlay scheme and cost function calculations utilized by our approach.

4.4.1 Overlay Scheme

We utilized a greedy overlay scheme in the interest of efficiency. We first calculate the buffer usage of the batch b, and derive the memory available for code ($= C_{code}$). If C_{code} is able to accommodate all the filters in batch b then code overlay is not required. Otherwise, we need to determine an overlay scheme and estimate the resulting overhead. We first assign as many filters as possible into C_{code} in decreasing order of their code size. Then we remove the last filter that was assigned to C_{code} and utilize the available memory ($C_{overlay}$) to overlay the remaining filters. We sort the remaining filters of b in decreasing order of their iteration number and assign them to segments (S_i) as long as the segment size does not exceed the $C_{overlay}$. The overlay overhead is given by:

$$\tau_{overlay}(b) = \sum_{S_i \in b} \tau(S_i)$$

$$\tau(S_i) = \begin{cases} T_{init} \text{ if } |S_i| \leq D_{init} \\ T_{init} + (|S_i| - D_{init}) * T_{slope}, \text{ Otherwise} \end{cases}$$

4.4.2 Cost functions

In the following paragraphs we detail the calculation of buffer usage, the computation cost, and the communication cost.

4.4.2.1 Buffer usage

The buffer usage of a batch b is given by the memory required for storing all the incoming, outgoing and internal data of b. Therefore, it is given by:

$$buf(b) = \sum_{\forall e \in b} buf(e)$$

$$buf(e) = \begin{cases} (I_e + 1) \times C_e \text{ if } f_{pe}, f_{ce} \in b \\ 2 \times C_e \text{ if } f_{pe} \in b, f_{ce} \notin b \\ I_e \times C_e \text{ if } f_{pe} \notin b, f_{ce} \in b \end{cases}$$

4.4.2.2 Computation cost

The computation cost of batch b is given by the sum of the computation time of all the filters being assigned to it plus its overlay overhead.

$$\tau_{comp}(b) := \tau_{overlay}(b) + \sum_{f \in b} \tau_f$$

4.4.2.3 Communication cost

The communication cost of batch b is given by the time for b to fetch all its data in the steady-state execution.

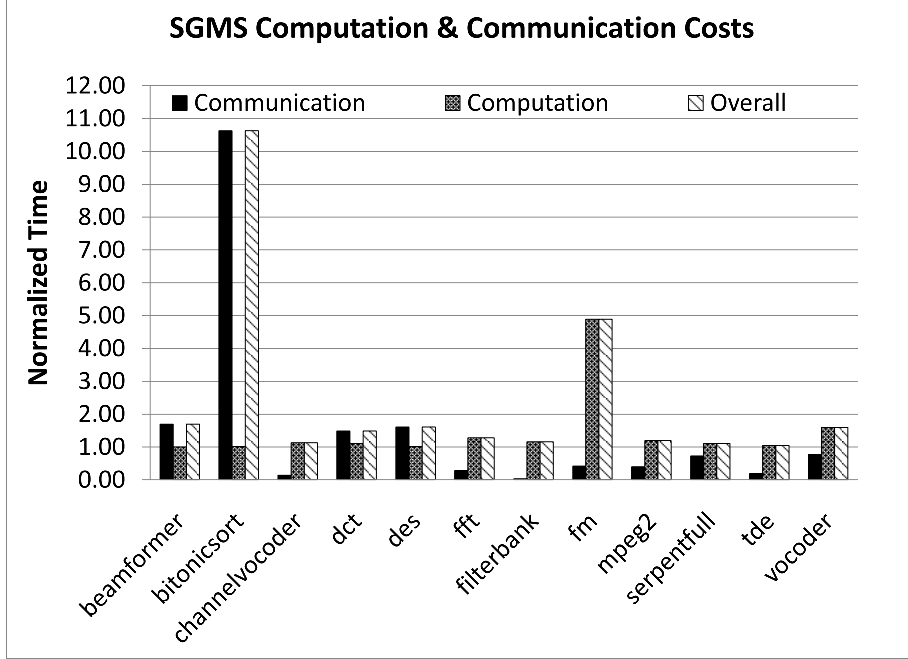


Figure 4.2: Computation and communication costs for SGMS

$$\tau_{comm}(b) = \sum_{\forall e: f_{pe} \notin b, f_{ce}} \tau_{comm}(e)$$

$$\tau_{comm}(e) = \begin{cases} T_{init} \text{ if } C_e \leq D_{init} \\ T_{init} + (C_e - D_{init}) \times T_{slope} \text{ otherwise} \end{cases}$$

4.4.2.4 Overall cost of solution

The cost of a batch is given by $\tau(b) = \max(\tau_{comm}, \tau_{comp})$. The effective cost of a batch is given by $\tau_{eff}(b) = N/\tau(b)$ where N is the number of copies of b. Finally, the overall cost of the application is given by the largest effective execution time over all batches.

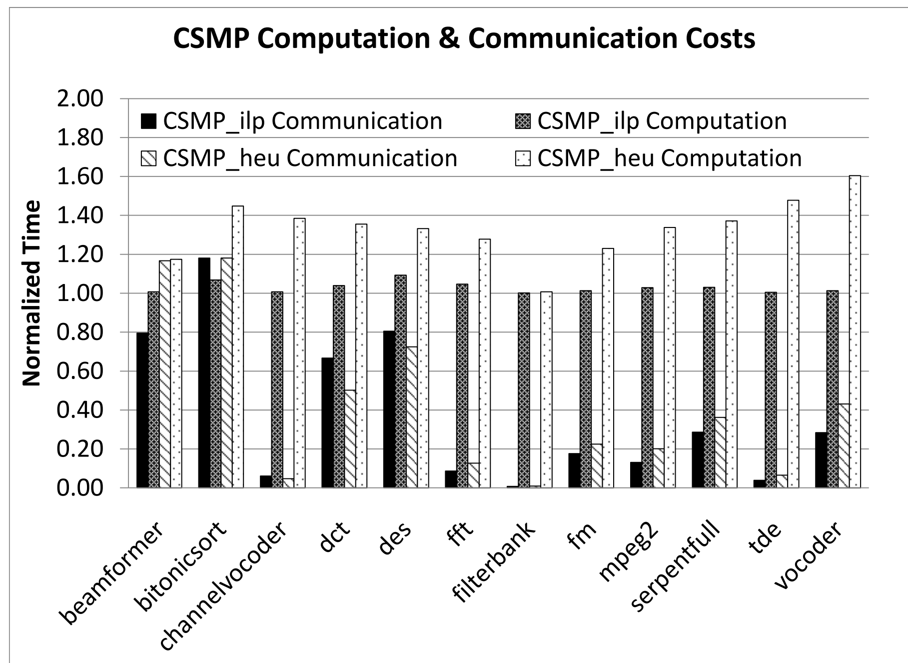


Figure 4.3: Computation and communication costs for CSMP_{ilp} and CSMP_{heu}

4.5 Experimental Results

In this section we present experimental results that evaluate CSMP_{ilp} and CSMP_{heu}, and compare them with the SGMS approach [48]. We utilized StreamIt benchmarks that are delivered with the 2.1.1 version of the compiler. We compiled the benchmark applications on Sony PS3 platform that includes the Cell BE. Due to hardware constraints of the platform only 6 SPEs and the PPE are available to the application developer. We utilized the StreamIt compiler to generate C implementations which were then passed to the respective gcc compilers to obtain implementations on the SPE and PPE.

Table 4.3: Maximum SPE Buffer usage of SGMS, $CSPM_{ilp}$ & $CSPM_{heu}$ (BYTES).

	SGMS	$CSPM_{ilp}$	$CSPM_{heu}$
beamformer	1004	1176	1036
bitonicsort	400	236	564
channelvocoder	10216	6492	12412
dct	196608	16384	172032
des	8320	10116	7116
fft	24576	34816	40960
filterbank	2588	2672	1924
fm	204	368	196
mpeg2	24742	31028	39856
serpentfull	10204	38504	93696
tde	122880	120514	145920
vocoder	2636	2304	4476

4.5.1 Comparisons with 256KB SPE memory

We first ran the three techniques with 256KB SPM which is the size of the SPE local store. Figure 4.2 presents the computation and communication costs of the SGMS solutions. The y-axis in the figure stands for the steady-state execution time of each benchmark normalized to its lower bound. The lower bound is calculated by the total execution time of all the filters in that benchmark over the number of processors. As is illustrated in Figure 4.2, for 4 benchmarks the SGMS solutions have their communication costs overwhelming the the computation costs. In the extreme case, for the bitonicsort benchmark, the communication cost is more than 10x over the computation cost. Figure 4.3 presents the computation and communication costs of our $CSMP_{ilp}$ and $CSMP_{heu}$ algorithms. In our ILP solutions, there is only one benchmark with its communication cost larger than the computation cost. The ratio, however,

Table 4.4: Run time of SGMS, $CSPM_{ilp}$ & $CSPM_{heu}$ (SECONDS).

	SGMS	$CSPM_{ilp}$	$CSPM_{heu}$
beamformer	12	4172	0.67
bitonicsort	80	4124	0.46
channelvocoder	59	26319	1.38
dct	1541	91764	0.51
des	165	33083	1.19
fft	29	3328	0.08
filterbank	236	42297	10.86
fm	59	38541	0.51
mpeg2	101	16364	0.11
serpentfull	56	23853	55.83
tde	116	34412	0.15
vocoder	60	22387	5.62

SGMS and $CSMP_{ilp}$ ran on server with Quad-Core Intel(R) Xeon(TM) CPU at 2.8GHz and $CSMP_{heu}$ ran on PC with Intel(R) Core(TM)2 Quad CPU at 2.4GHz.

is only 1.1x, a much smaller number comparing to 10x as in the SGMS solutions. In the solutions generated by $CSMP_{heu}$ the communication costs are hidden by the computation costs for all benchmarks. Figure 4.4, compares the overall performance of the SGMS, $CSMP_{ilp}$ and $CSMP_{heu}$. As is observed from the figure, $CSMP_{ilp}$ always performs better than the SGMS. $CSMP_{heu}$ also out performs the SGMS on average. The above results show that our techniques are able to effectively trade-off the computation and communication costs to balance the overall performance.

In Table 4.3, we present a comparison of the buffer usage of the SGMS, our ILP and heuristic. As we can see, the buffer usage of the three approaches are more or less comparable to each other. This is because the memory usage of all the benchmarks are fairly small comparing to the scratchpad memories. Therefore, the memroy constraint hasn't been apposed yet. In the next section,

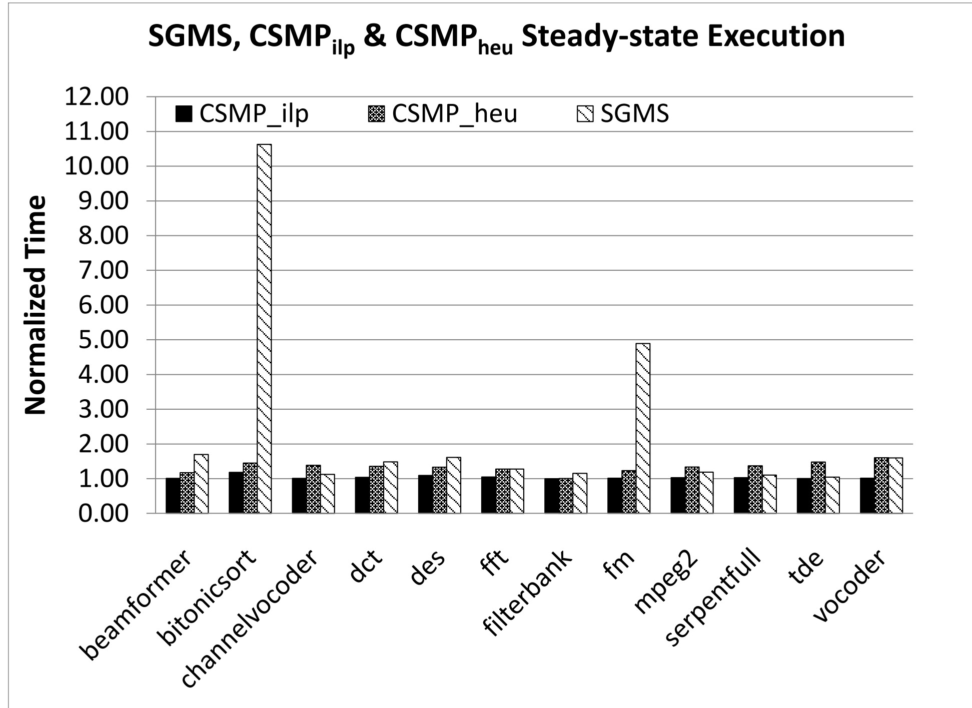


Figure 4.4: SGMS, CSMP_{ilp} and CSMP_{heu} with 256KB SPM.

we artificially shrink the scratchpad memories to 16KB and demonstrate how it affects the mapping of the SGMS, our ILP and the heuristic algorithm.

In Table 4.4 we compare the run time of the SGMS, CSMP_{ilp} and CSMP_{heu}. On average, the SGMS took 209.5 seconds to finish and CSMP_{ilp} took 28387 seconds. The numbers indicate the SGMS is over 100 times faster than CSMP_{ilp}. The primary reason is that, SGMS tries to do load balancing solely based on the steady-state execution time of each filter. CSMP_{ilp} takes the whole graph as input and does load balancing based on the filter execution time, the data communication, the current schedule, the buffer usage, and more importantly, the overlay scheme. On the other hand, CSMP_{heu} is

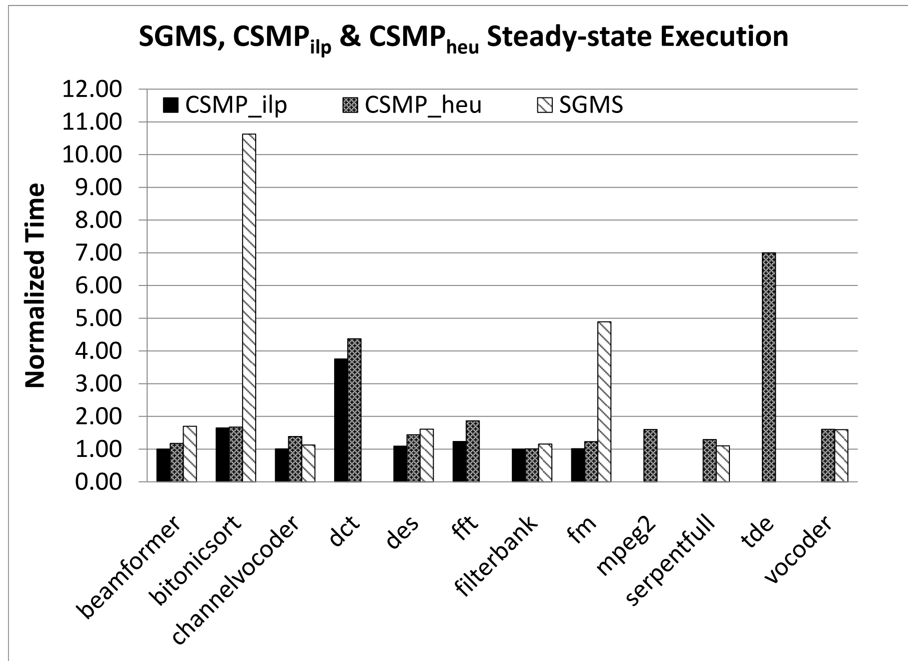


Figure 4.5: SGMS, CSMP_{ilp} and CSMP_{heu} with 16KB SPM.

very fast even though it considers all the design trade-offs. 7 out of the 12 benchmarks terminate within 1 second. The serpentfull benchmark requires the longest run time as it has 120 filters and 128 edges. In summary our heuristic approach is able to generate high quality solutions (27.8% slower than our ILP on average, 44.5% faster than SGMS on average) in very short times run times.

4.5.2 Comparisons with 16KB SPE memory

The solutions for the previous experiment did not require code overlays due to the large size of the SPM. We conducted a second set of experiments with SPM size constrained to 16KB. We timed out CSPM_{ilp} after 9 hours and utilized the solution if it was feasible. CSPM_{ilp} was unable to generate solutions for

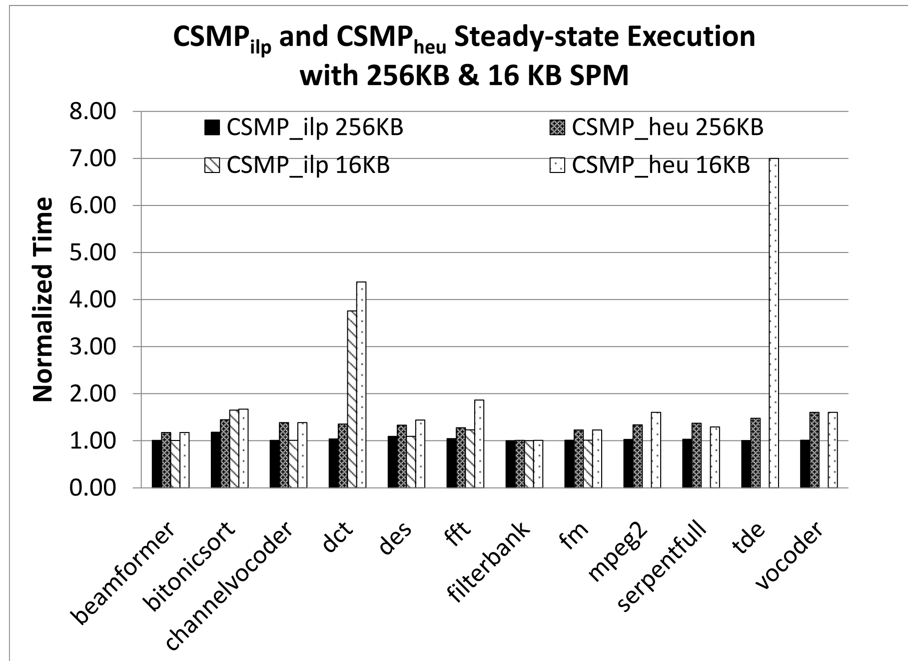


Figure 4.6: Performance of CSMP_{ilp} and CSMP_{heu} with 256KB and 16KB SPM

4 benchmarks (namely mpeg2, serpentfull, tde and vocoder). Table 4.5 lists the maximum communication buffer usage for SGMS, CSPM_{ilp} and CSPM_{heu}. SGMS was unable to find solutions in four instances (shown in bold) as the communication buffer requirement violated the SPM memory constraint. The overall performance of the SGMS, CSPM_{ilp} and CSPM_{heu} are presented in Figure 4.5. Some of the CSPM_{ilp} data points are missing as it timed out. In the case of SGMS the data points that are missing (dct, fft, mpeg2, tde) indicate infeasible solutions. CSPM_{heu} was able to generate feasible solutions for all benchmarks. Except for two benchmarks (channel vocoder and serpentful) our heuristic is able to out perform SGMS.

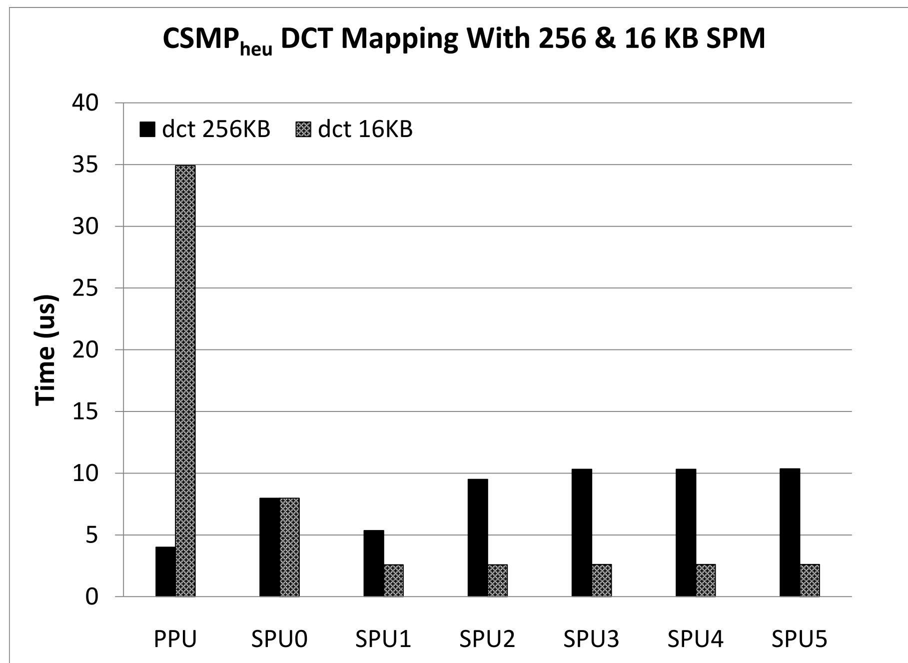


Figure 4.7: DCT with 256KB and 16KB SPM

Figure 4.6 presents the performance of CSPM$_{ilp}$ and CSPM$_{heu}$ solutions with 256KB and 16 KB SPM capacities, respectively. Reducing the SPM size leads to lower performance in all cases. Figure 4.7 shows the executions times on the PPE and SPE for dct with 256KB and 16 KB SPM. As can be seen from the figure reduction in the SPM size leads to more computation migrating to the PPE.

4.6 Summary

The chapter addressed the problem of compiling stream programs on embedded multicore processors that incorporate SPMs. We proposed an ILP formulation and heuristic approach that are able to effectively consider all

Table 4.5: Maximum communication buffer usage of SGMS, CSMP_{ilp} & CSMP_{heu}.

	SGMS (Bytes)	CSMP_{ilp} (Bytes)	CSMP_{heu} (Bytes)
beamformer	1004	1156	1036
bitonicsort	400	288	564
channelvocoder	10216	11212	12400
dct	196608	10240	8192
des	8320	15708	12604
fft	24576	14336	14436
filterbank	2588	3472	1920
fm	204	328	248
mpeg2	24742	NA	1126
serpentfull	10204	NA	5836
tde	122880	NA	0
vocoder	2636	NA	5756

* The bold faced values indicate that they exceeded SPM capacity.

the design trade-offs. We evaluated the approaches by compiling StreamIt programs on the Cell BE processor, and comparing with an existing technique, namely SGMS. The experimental results showed that our approaches are able to effectively balance the computation and communication overheads when mapping stream programs on multicore processors. Further, our heuristic approach is able to generate high quality solutions even under tighter SPM capacity constraints while SGMS produces infeasible solutions. Future work will address dynamic scheduling of stream programs.

Chapter 5

SCHEDULING OF STREAM PROGRAMS ON SPM BASED MULTICORE PROCESSORS THROUGH RETIMING

The prevalence of stream computing in signal processing, multi-media, and network processing domains has resulted in a new trend of programming and architecture design. Stream applications distinguish themselves from traditional sequential programing languages through well defined independent actors, explicit data communication, and stable code and data access patterns, all of which come together to enable a compiler to automatically schedule them on multicore processors. For fast and efficient execution of stream applications, scratch pad memory (SPM) has been introduced into today's embedded multicore processors. Performance optimization on SPM based multicore architectures requires a programmer, or compiler to efficiently manage the limited on-chip memories and bandwidth. In this chapter, we address the problem of automatic compilation of stream programs onto SPM based multicore architectures through a retiming technique. An integer linear programming (ILP) approach is first provided. Although the ILP grants us with high quality solutions, it suffers from very long algorithm run time. In the second part of this chapter, we introduce a heuristic technique that solves the same problem with comparable results and runs in a matter of seconds. Trade-offs between double buffering for hiding data communication with computation and code overlay for sharing the limited on-chip memory among different code segments are explored intensively in our techniques. The efficiency of our techniques was evaluated by compiling several stream applications for the IBM Cell Broadband Engine (BE) and compared their results with existing approaches.

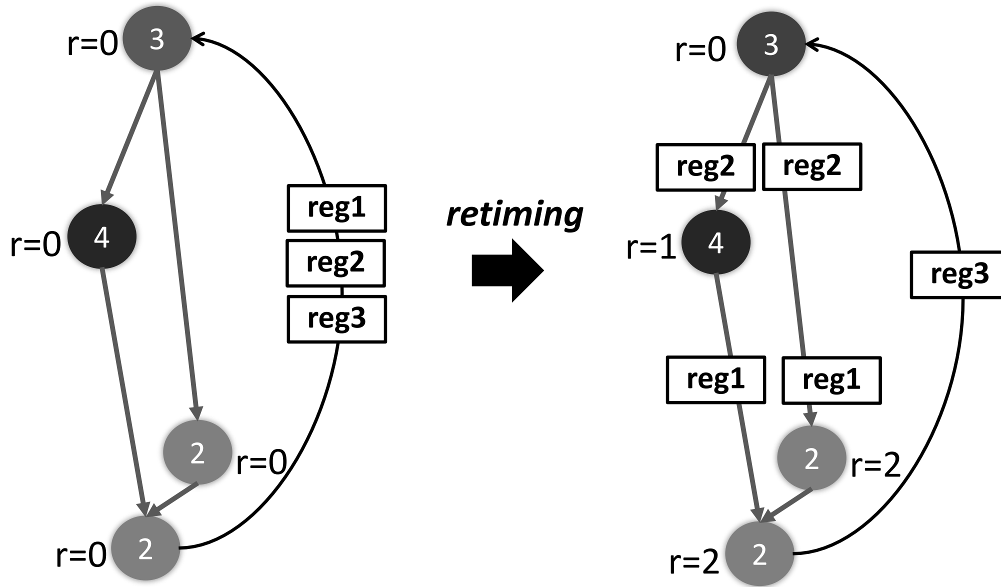


Figure 5.1: Retiming example.

5.1 Retiming

Classic retiming technique is primarily used in the domain of circuit design. The input to the retiming algorithm is a synchronous circuit given by $G < V, E, d(v), w(e) >$. V represents function units in the circuit and E represents data dependencies among distinct function units. $d(v)$ indicates the function delay of unit v and $w(e)$ indicates the initial register count on each edge e. A retiming of G is a vertex to integer mapping $r : V \rightarrow Z$ [50]. The graph after retiming is denoted by $G_r < V, E, d(v), w_r(e) >$, where for an edge $e : u \rightarrow v$, $w_r(e)$ in the retimed graph is given by $w_r(e) = w(e) + r(v) - r(u)$. The retiming technique can be viewed as a transformation that alters the clock period of a circuit by inserting and deleting registers while keeping the circuit's functionality unchanged. Traditionally, the retiming technique is adopted for clock period minimization of synchronous circuit [50].

In Figure 5.1 we provide a simple retiming example. The stream pro-

gram is composed with four actors. The number inside each actor denotes its run time, $d(v)$. Before retiming, the retiming delay r of each actor is set to 0 as show in the example. The initial register count on feedback loop edge is 3. For the rest of edges, the initial register count is 0. Then we perform retiming algorithm that is discussed in [50] and get the retimed graph. After retiming, the actor with rum time 3 has retiming delay $r = 0$, the actor with run time 4 has retiming delay $r = 1$, and the two actors with run time 2 has retiming delay $r = 2$. By applying $w_r(e) = w(e) + r(v) - r(u)$, we can calculate the number of registers on each edge after retiming and they are shown in the retimed graph. After removing edges with $w_e(e) > 0$, the longest critical path in all connected subgraphs determines the initiation interval, which in our example is $2 + 2 = 4$.

5.2 Motivation

The retiming technique is intriguing for compiling SDF specifications onto multicore architectures because of the following reasons. i), throughput maximization of scheduling an SDF can be achieved through minimizing its steady-state execution time; ii), the steady-state execution time of a single appearance SDF is equivalent to the clock period of the corresponding synchronous circuit. Further, the retiming technique also inherently handles cyclic dependencies in a data flow graph. By appropriately adopting the retiming technique, an upper bound on the resulting number of pipeline stages could be imposed.

Although retiming technique is intriguing in several ways to solve our problem, traditional clock minimization approaches [50] with retiming are not directly applicable to our problem instances. Traditional approaches assume

that there are infinite resources. The solution given by retiming was interpreted as each node occupies a separate specialized hardware, data communications among different function units are immediate (without any overhead), and there is no code to be stored for a function unit (hardware implemented). As a result, the critical path in the graph after retiming is defined as the clock period of the retimed graph. In our problem instances, we have a fixed number of PEs, data communications among different PEs take time, and each PE has a limited on-chip SPM for storing program code and data. Because of the limited number of PEs, actor to PE mapping needs to be generated with the retiming process. For each PE, if not all the program code and internal data buffer can fit into the SPM, then code overlay needs to be implemented. The steady-state execution time instead of clock period defines the performance metric and is calculated by the largest execution time among all pipeline stages. To amortize data communication overhead, double buffering (DB) scheme needs to be exploited. Double buffering scheme requires a dedicated pipeline stage and additional memory for storing an extra copy of data. A user specified allowable pipeline stages could affect whether double buffering scheme could be implemented. Arbitrarily introducing double buffering also increases buffer usage, therefore could lead to additional code overlay overhead. A smart double buffering scheme that trades off data communication overhead with code overlay needs to be implemented to this effect. Given the above design trade-offs, we propose an ILP and a fast heuristic algorithm that automatically schedule an SDF G onto an embedded multicore architecture P through retiming such that i) the throughput is maximized; ii) the memory constraint of each on-chip SPM is respected; and iii) the number of software pipeline stages is no more than a user specified value. The contributions of

Table 5.1: SDF and architecture specification

	Constant	Description
SDF	$d(v)$	Delay/runtime of actor v
	$C(v)$	Code size of actor v
	$w(e)$	Number of registers on edge e
	$C(e)$	Data going through edge e
Architecture	$\|P\|$	Number of PEs
	$C(p)$	Local SPM size of PE p
	T_{init}	Base cost for any DMA transfer
	T_{slope}	Rate of increase of DMA transfer
	D_{init}	Largest data/code size with T_{init}
Pipeline constraint	N_{user}	User-specified pipeline stages

this chapter include:

1. An ILP formulation that performs retiming and actor to PE mapping with double buffering and code overlay for throughput maximization - RTEM ILP.

2. A fast heuristic approach that is able to solve the same problem with comparable performance results in a matter of seconds - RTEM heuristic.

In the next section we formally define our problem. Section 5.4 discusses related work. Section 5.5 describes our strategies for properly handling of cycles that might be present in a stream program. Section 5.7 describes our ILP formulation. In Section 5.8, we discuss the limitations of our ILP approach and provides a retiming heuristic. Finally Section 5.9 presents our experimental results and Section 5.10 concludes this chapter.

5.3 Problem Description

The input to our problem consists of a software specification and a hardware architecture description. The software specification is represented by a synchronous data flow (SDF) graph $G < V, E >$ that is extracted from the intermediate representation of a stream application. We require the input SDF to our technique to be consistent. An SDF is defined as consistent if a finite input sequence that avoids both buffer under-flow and over-flow on each edge can be constructed [49]. Before entering our technique, we convert a classical SDF (typically a multiple appearance SDF where each filter in the steady-state execution has multiple executions) to a single appearance SDF[1]. In the single appearance SDF $G < V, E >$, V represents the actors and E represents the edges. Each actor $v \in V$ is again given by a tuple $< d(v), C(v) >$ as illustrated in Table 5.1. $d(v)$ and $C(v)$ implies the run time and code size of an actor that is estimated instruction by instruction[2]. Each $e \in E$ is given by $< w(e), C(e) >$. $w(e)$ indicates the user specified initial register distribution on each edge, which is in fact determined by the initial function of each actor. $C(e)$ indicates the data size being transferred through edge e in the single appearance SDF. The target embedded multicore architecture is represented by P, where each $p \in P$ is given by a tuple $< C(p), T_{init}, T_{slope}, D_{init} >$. $C(p)$ indicates the size of the on-chip processor memory. We model the Direct Memory Access (DMA) behavior in the IBM Cell BE by three parameters T_{init}, T_{slope} and D_{init}. T_{init} denotes the base cost for any DMA transfer operation. When the data or code size being transferred is smaller than D_{init}(=1KB),

[1] In the case when cycles exist in the SDF, the transformation from a multiple appearance SDF to a single appearance SDF is not trivial and the discussion is provided in Section 5.5.

[2] In the case when the number of loop iterations is data dependent, then a constant five is assumed for optimization purpose.

there is only a base cost of T_{init}(=0.21us). Otherwise, an increasing rate of T_{slope}(=0.075us/KB) is encountered for every extra byte of data or code. Finally there is a constant N_r that denotes the user specified allowable number of software pipeline stages in the resulting solution.

The output of our technique is an actor to PE mapping, memory partition of each on-chip SPM, a software pipeline schedule of the stream program across PEs, and data and code transfers among different memory elements with double buffering and code overlay that maximize the overall throughput.

5.4 Related Work

Several previous approaches have addressed the problem of implementing streaming workload on embedded multicore processors. A hierarchical framework for scheduling SDF onto multicore processors was discussed by Pino et al. [66]. The objective of this work is to reduce the number of actors in an SDF and at the same time still preserve enough parallelism. More recently Chen et al. [21] and Ostler et al. [61] proposed techniques for mapping stream based applications onto network processors. Chen et al. proposed a Shangri-La compiler framework that maps a C-like packet program onto a network processor. Ostler et al. investigated fusion and fission operations and provides an integer linear programming (ILP) and a heuristic approach to map stream like applications onto network processors. Liao et al. [52] investigated parallelizing Brook language onto general purpose multicore processors through data and computation transformations. Gordon et al. [31] [32] explored trade-offs between data and task level parallelisms and developed a heuristic algorithm to generate multi-threaded code for the RAW architecture. Stratton et al. [70]

developed a framework of MCUDA that executes CUDA language on a shared memory multicore processor. The framework contains both a set of source-level compiler transformations and a run time system for parallel execution and demonstrates that CUDA can be an effective data-parallel programming model for shared memory multicore architecture. In contrast to the above approaches, our technique focuses on embedded multicore processors that incorporate SPMs. In addition to actor to PE mapping for load balancing, double buffering for computation and communication overlap, we also face the challenge of efficiently managing the limited on-chip SPMs for program code and data buffers. An efficient code overlay scheme that shares the on-chip SPM over different code segments is critical when the SPM is smaller than the total size of the program code and data buffers.

There have been approaches that concentrate on automatic compilation of stream applications onto SPM based architectures. Hormati et al. [35] proposed a Sponge compiler framework for mapping stream languages onto GPUs. The primary focus of this work is the abstraction of hardware architectures and provide portability across different generations of GPUs. Kudlur et al. [48] came up with an ILP that unfolds and partitions a stream application onto SPM based multicore processors. In his ILP formulation, the communication overhead was assumed to be zero and the on-chip SPMs are assumed to be sufficient large to accommodate all program code and data. An improved version of the same work was later presented by Choi et al. [24]. In this work, again all the DMA transfers for data communication are assumed to be hidden and have zero cost. A memory constraint is added to the original ILP formulation to capture the limited size of the on-chip SPM. However, no code overlay is implemented. When the program code and data size is big-

ger than the on-chip SPMs, the technique fails to generate a valid solution. Our approach is distinguished from the above approaches in that we explore the trade-offs between double buffering and code overlay under a limited local SPM and efficiently address the SPM constraint through smart double buffering and code overlay. In our problem instances, the actor to PE mapping not only impacts the load balancing, computation communication overlap, but also the data, code partition of the on-chip SPMs, which essentially determines the code overlay overhead.

The previous work that comes closest to us are the CSMP ILP and heuristic approaches proposed by Che et al. [20]. Both approaches in this work utilize fusion and fission operations to schedule stream formats onto SPM based multicore processors. The existing techniques make no guarantee on the number of software pipeline stages being generated, thus may result in high latency. The CSMP and heuristic approaches also treat any loop structure as a single actor thus overlook the opportunities to optimize cycles. Our retiming ILP and heuristic approaches maximize the throughput with a user-specified number of pipeline stages and inherently handles cycles that may be present in the stream applications.

5.5 Resolving Cycles

Cycles in a stream program impose several barriers on software synthesis and code generation, therefore greatly influence the optimizations that can be applied. The presence of cycles in a program essentially indicates cyclic data dependencies that if not properly taken care of can result in deadlock in scheduling. In our ILP and heuristic approaches, we require a single appearance SDF.

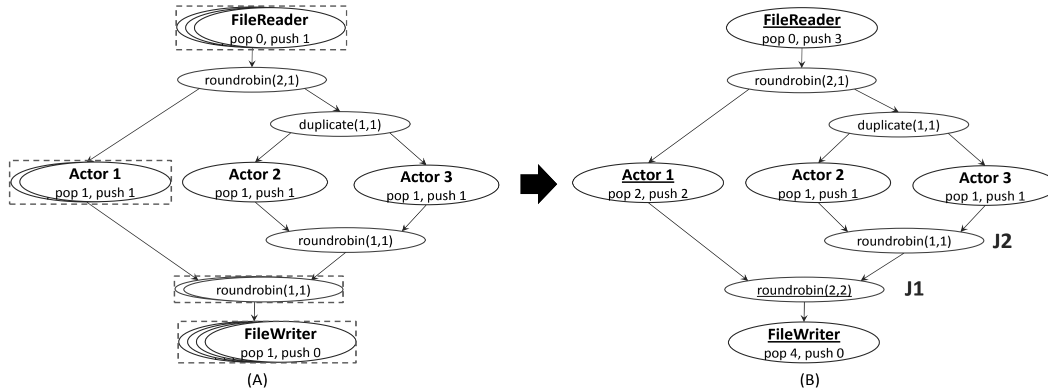


Figure 5.2: Single appearance SDF construction from a multiple appearance stream program without cycles.

Given a data flow model without cycles, we can simply treat all executions of each actor in the steady-state execution as one and thus derive a single appearance SDF. An example for transforming the stream program given by Figure 1.1 to a single appearance SDF is provided in Figure 5.2. The stream program on the left hand side is the original multiple appearance SDF and on the right hand side is the resulting single appearance representation. The right hand side actor with an underlined name typically indicates that one execution of the current actor corresponds several executions of the original actor on the left hand side. In the case when cycles are present in a stream program, the transformation is non-trivial. Bhattacharyyan et al. [8] [11] proved that if throughout a single schedule period (or a steady-state execution) of a data flow model, the actors can be partitioned into two subsets such that one subset is precedence-independent of the other subset, then a single appearance schedule exists[3]. Precedence-independent of two subsets is formally referenced as subindependence and defined as S_1 is subindependent of S_2 if

[3]The existence of a single appearance schedule of an SDF and the existence of a single appearance SDF is equivalent in the sense that given a single appearance SDF (consistent), we can easily construct its single appearance schedule by following a data flow order.

for each edge e directed from an actor of S_2 to an actor of S_1, the number of delays on e is at least equal to the number of tokens consumed from e in the steady-state execution. Bhattacharyyan et al. [11] further provides process for transforming a multiple appearance SDF to a single appearance SDF by removing loosely dependent edges and breaking cycles. In the case when there is no single appearance SDF for the stream program, a clustering technique is employed to transform cycles that can not be broken (a tight cycle) into a single actor. Then, a single appearance SDF can be constructed for the new graph.

In this chapter, instead of breaking cycles of an SDF by removing loosely dependent feedback edges, we derive a single appearance SDF by preserving all its edges and trying to process an actor as many times as the delays on its feedback edge permit. Figure 5.3 and Figure 5.4 illustrates how our technique handles a multiple appearance SDF with a tight dependent cycle and a loose dependent cycle respectively. The number N on the left hand side of each actor in the original stream program denotes the number of executions of that actor in the steady-state execution. The number on the feedback edge (5d in Figure 5.3 and 20d in Figure 5.4) indicates the delays that presents on that edge. A shaded circle denotes that the actor is stateful. In Figure 5.3 roundrobin actor has N=10 executions in the steady-state. In order to execute roundrobin actor 10 times consecutively, 10 delays (tokens) must be present in its feedback edge. As described in Figure 5.3 (A), there are only 5 delays (5d) present, indicating that a single appearance SDF is not possible for the original graph. In this case, we treat the entire cycle as a single actor and derive the single appearance SDF as shown in Figure 5.3 (B). After transformation, the feedback loop becomes a cycle inside the combined high level

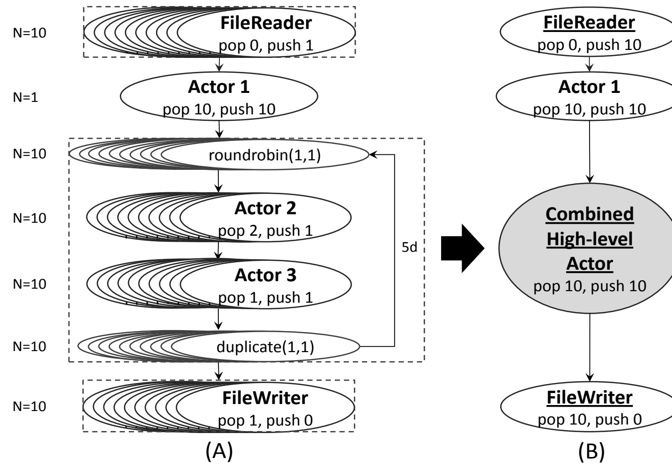


Figure 5.3: Single appearance SDF construction from a multiple appearance stream program with a tight cycle dependence.

actor. We shade it to denote that the combined high level actor is stateful. The value stored on the feedback edge becomes a state variable. In Figure 5.4 (A), for the same input stream program, if the delays on the feedback edge is 20 (20d), we can consecutively execute roundrobin actor 10 times in a group for two iterations. Therefore, a delay of 2 (2D) is put on the feedback edge of the resulting single appearance SDF as shown Figure 5.4 (B). With a loose dependent cycle, we can still collapse it into a single actor and Figure 5.4 (C) presents the corresponding single appearance SDF. Both solution (B) and solution (C) in Figure 5.4 have their own advantages and limitations. Solution (B) has a better granularity and does not introduce extra stateful actors while solution (C) doesn't have any cycle thus can adopt a broader range of optimizations. In our techniques, solution (B) will serve as the input since our optimizations already inherently handle cycles.

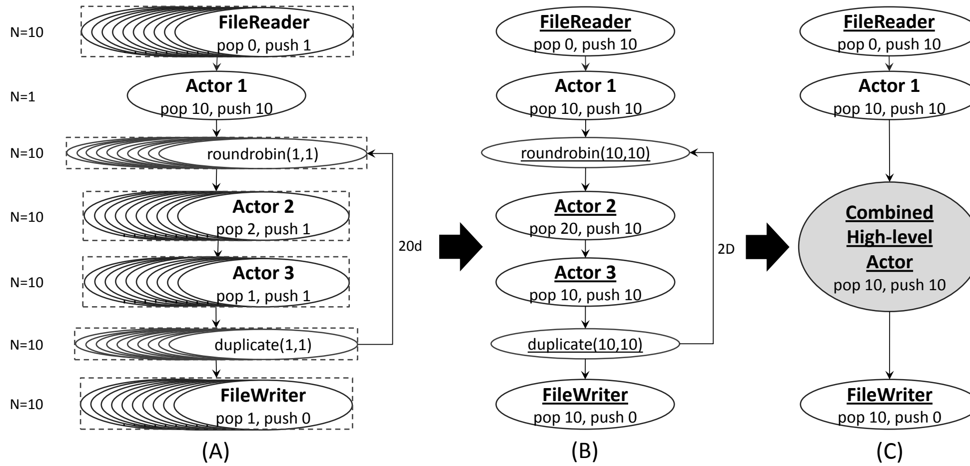


Figure 5.4: Single appearance SDF construction from a multiple appearance stream program with a loose cycle dependence.

5.6 Pre-processing

Prior to entering our RTEM ILP and heuristic approaches, we carry out the following pre-processing steps to legalize the inputs.

- If the FileReader is not present in the SDF, we add a dummy source actor s_{dummy} with zero code size and run time and introduce edges with $C_e = 0$ directing from s_{dummy} to every actor that has no incoming edge.
- If FileWriter is not present, we add a dummy sink actor t_{dummy} with zero code size and run time and introduce edges with $C_e = 0$ directing from every actor without any outgoing edge to t_{dummy}.
- Update SDF specification $G < V, E, d, w >$ with a feedback edge from sink to source.
- Traverse each $v \in V$ from source to sink following a breadth first search (BFS) manner. For each v being visited, push as many registers as possible from its incoming edges to its outgoing edges. Upon completion

of the above process, the number of registers on the feedback edge from sink to source defines the inherent pipeline stages that exist in the graph. We denote it by N_G.

- Add $N_{user} - N_G$[4] registers to the feedback edge from sink to source. N_{user} is a user specified number of pipeline stages. We require that N_{user} to be no less than N_G.

After the above pre-processing steps, we have an SDF specification of a stream program that has one feedback edge from its sink actor to its source actor. The delays on the feedback loop edge denotes the maximum pipeline stages after retiming.

5.7 Integer Linear Programming Approach

In this section, we formulate the problem described in Section 5.3 through an ILP approach. In the ILP approach, retiming delay to actor mapping and actor to PE mapping are performed. Simultaneously, smart double buffering is selectively introduced between a pair of producer and consumer that are assigned to different PEs. A code overlay scheme is also generated based on the current data buffer usage and code memory. Constraints such as valid retiming, valid mapping, unique scheduling order, and limited number of pipeline stages are imposed as discussed in the remainder of this section. Finally, the completion time of each actor due to processor work load, inter-pipeline and intra-pipeline data communications are calculated. The objective is to minimize the largest

[4] In the scenario where the program has to start with FileReader as the first actor to be executed, one extra pipeline stage might be introduced. This scenario can be simply handled by reducing the user specified number of software pipeline stages by one.

completion time among all actors, which is also the steady-state execution time of the entire stream program.

In Figure 5.5 we provide a simple example of mapping the stream program given by Figure 5.2 (B) to 3 PEs and 4 software pipeline stages. Figure 5.5 graph (A) provides the input to our ILP formulation and graph (B) provides the graph after retiming. In the retimed graph (B) of Figure 5.5, r denotes the retiming delay of each actor, p denotes the actor to PE mapping, and S denotes each connected subgraphs after retiming. In the retimed graph (B), actor $roundrobin(2,1)$ and $Actor1$ are separated by one pipeline stage and they are mapped to two different PEs. Therefore in a pipelined execution manner, $roundrobin(2,1)$ and $Actor1$ will execute simultaneously. However, an inter-pipeline communication cost is encountered in this case because double buffering cannot be implemented. Correspondingly, $Actor1$ and $roundrobin(2,2)$ are separated by 2 pipeline stages. In this case, a double buffering can be either introduced or not depending on the memory constraint and code overlay overhead. In another case, actor $roundrobin(2,2)$ and $FileWriter$ belong to the same subgraph after retiming [5]. $FileWriter$ cannot execute until $roundrobin(2,2)$ finishes its execution and transfers its data to $FileWriter$. From the above discussion, to determine the completion time of an actor, inter-pipeline and intra-pipeline data communication, processor work load, double buffering and code overlay, they all need to be captured in the ILP formulation. In the remainder of the section, the details of our ILP formulation that captures the program behavior as discussed above will be discussed thoroughly.

[5] Actors belong to the same subgraph also belong to the same software pipeline stage.

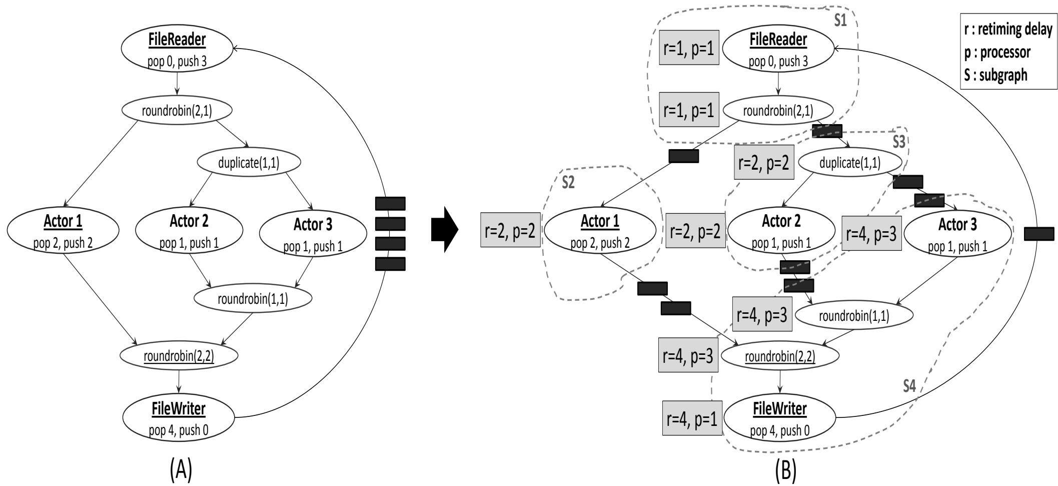


Figure 5.5: A simple example of retiming a stream program with 3 PEs and 4 software pipeline stages.

5.7.1 Decision variables

- a_{rv}, 0/1. If 1, indicates that a retiming delay r is assigned to actor v.
- b_{vp}, 0/1. If 1, indicates that actor v is assigned to PE p.
- d_{uv}, 0/1. If 1, indicates that double buffering is enabled between actor u and actor v.
- p_v, 0/1. If 1, indicates that actor v is assigned to the on-chip SPM. Otherwise, actor v is assigned to the off-chip main memory and is brought into the overlay region of an on-chip SPM at run time.

5.7.2 Derived variables

- w^r_{uv}, *integer*, number of delays on edge $e : u \rightarrow v$ after retiming. In the following equation, $\sum_{r \in R} a_{rv} * r$ and $\sum_{r \in R} a_{ru} * r$ calculate the integer retiming delay of actor v and actor u and w_{uv} indicates the initial delays

on edge $u \rightarrow v$ ($w_{uv} := w(e) | e : u \rightarrow v$).

$$\forall u, v \in V | \exists e : u \rightarrow v : w^r_{uv} := w_{uv} + \sum_{r \in R} a_{rv} * r - \sum_{r \in R} a_{ru} * r$$

- x_{uv}, 0/1. If 1, indicates that there is at least one delay on an edge $e : u \rightarrow v$ in the retimed graph.

$$\forall u, v \in V | \exists e : u \rightarrow v : x_{uv} * N_r \geq w^r_{uv}$$
$$x_{uv} \leq w^r_{uv}$$

- y_{uv}, 0/1, is a derived variable that is defined on y_{uvp}. In the following constraint, the first inequality ensures that if actor u and v are both assigned to some PE p then the corresponding y_{uvp} being set to 1. Otherwise, the second and third inequalities make sure that y_{uvp} equals 0. $y_{uv} = 1$ indicates that actor u and v are assigned to the same PE.

$$\forall u, v \in V | u \neq v, p \in P : y_{uvp} \geq b_{up} + b_{vp} - 1$$
$$y_{uvp} \leq b_{up}$$
$$y_{uvp} \leq b_{vp}$$
$$\forall u, v \in V | u \neq v : y_{uv} := \sum_{p \in P} y_{uvp}$$

- z_{vp}, 0/1. If 1, indicates that actor v is always present in the on-chip SPM of p. Otherwise, there is a potential code overlay for executing v on p. Similar to the way that we constructed y_{uv}, in the following constraint the first inequality ensures that if both the conditions of actor v is mapped to p and actor v is always present in the on-chip SPM, then the corresponding z_{vp} is set to 1, otherwise z_{vp} is set to 0.

$$\forall v \in v, p \in P : z_{vp} \geq b_{vp} + p_v - 1$$
$$z_{vp} \leq b_{vp}$$
$$z_{vp} \leq p_v$$

- s_{uv}, 0/1. If 1, indicates that actor u is scheduled before actor v for execution. In the following constraint, the first equation ensures that actor u is either scheduled before or after actor v. The next two inequalities make sure that if there is a schedule sequence of u, v, w, then s_{uw} is set to 1 and vice versa.

$$\forall u, v \in V | u \neq v : s_{uv} + s_{vu} = 1$$

$$\forall u, v, w \in V | u \neq v \neq w : s_{uw} \geq s_{uv} + s_{vw} - 1$$

$$s_{uw} \leq s_{uv} + s_{vw}$$

- γ_v, *integer*, the code overlay overhead of actor v. γ_v equals 0, if actor v is always present in the on-chip SPM. Otherwise, the code overlay overhead is given by the DMA latency of transferring actor code of v from the off-chip main memory.

$$\forall v \in V : \gamma_v := T_v * (1 - p_v)$$

where T_v is a pre-calculated value that is given by,

$$T_v = \begin{cases} T_{init}, \text{ if } C(v) \leq D_{init} \\ T_{init} + T_{slope} * (C(v) - D_{init}), \text{ otherwise} \end{cases} \tag{5.1}$$

5.7.3 Constraints

- Retiming Delay to Actor Assignments: Each actor is allocated one and only one retiming delay.

$$\forall v \in V : \sum_{r \in R} a_{rv} = 1$$

- Valid Retiming: After retiming, there is no edge with negative delays.

$$\forall u, v \in V | \exists e : u \rightarrow v : w^r_{uv} \geq 0$$

- Actor to PE Assignments: Each actor is assigned to one and only one PE to execute.

$$\forall v \in V : \sum_{p \in P} b_{vp} = 1$$

- Scheduling Order: For actors u, v that are mapped to the same PE ($y_{uv} = 1$) and there is an edge connecting actor u and v without any delay, actor u must be scheduled before actor v to respect the data dependencies.

$$\forall u, v \in V | \exists e : u \rightarrow v : s_{uv} \geq y_{uv} - w^r_{uv}$$

- Double Buffering: Double buffering can be introduced between a producer and a consumer only when they are separated by at least 2 pipeline stages [6].

$$\forall u, v \in V | \exists e : u \rightarrow v : 2 * d_{uv} \leq w^r_{uv}$$

- FileReader, FileWriter: This constraint is specific to the heterogeneous architecture of the IBM Cell BE. In the IBM Cell BE architecture, only the PPE hosts a file system and therefore the FileReader and FileWriter can only be processed on the PPE. For architectures that don't have such limitations, the constraint can be simply removed.

$$b_{11} = 1, \quad b_{|V|1} = 1, \quad s_{1|V|} = 1 \tag{5.2}$$

- Processor Memory: The processor memory should be able to hold all actor code, data buffers and a code overlay region. In the following constraint $C^{intra}_{uvp}, C^{in}_{uvp}, C^{out}_{uvp}$ captures the number of buffers allocated for each edge e on PE p when e is an intra-processor edge, an incoming edge,

[6] For actors mapped to the same PE, there is no difference between an execution with and without double buffering.

or an outgoing edge respectively. $\sum_{v \in V} z_{vp} * C(v)$ calculates the memory allocated for code that always presents in the on-chip SPM of PE p and $C_p^{ovleray}$ captures memory usage of the code overlay region.

$$\forall p \in P : C(p) \geq \sum_{u,v \in V | \exists e : u \rightarrow v} (C_{uvp}^{intra} + C_{uvp}^{in} + C_{uvp}^{out}) * C(e) +$$
$$\sum_{v \in V} z_{vp} * C(v) + C_p^{overlay}$$

In the following inequality, both b_{up} and b_{vp} equals 1 indicates that actor u and v are mapped to the same PE. In this case, C_{uvp}^{intra} is greater than or equal to $w_{uv}^r + 1$.

$$\forall u, v \in V | \exists e : u \rightarrow v, \forall p \in P : C_{uvp}^{intra} \geq w_{uv}^r + 1 + (b_{up} + b_{vp} - 2) * N_r$$
$$C_{uvp}^{intra} \geq 0$$

Correspondingly, b_{vp} equals 1 and b_{up} equals 0 indicates that only the consumer is mapped to PE p, In this case, C_{uvp}^{in} is greater than or equal to w_{uv}^r. In the case when $w_{uv}^r = 0$, the second inequality ensures that at least one buffer is allocated on the consumer side to implement DMA.

$$\forall u, v \in V | \exists e : u \rightarrow v, \forall p \in P : C_{uvp}^{in} \geq w_{uv}^r + (b_{vp} - b_{up} - 1) * N_r$$
$$C_{uvp}^{in} \geq b_{vp} - b_{up}$$

On the producer side, we allocate 2 buffers if double buffering is enabled. Otherwise only 1 buffer is allocated. In the following inequality b_{up} equals 1 and b_{vp} equals 0 indicates only the producer is mapped to PE p and d_{uv} indicates whether double buffering is enabled.

$$\forall u, v \in V | \exists e : u \rightarrow v, \forall p \in P : C_{uvp}^{out} \geq b_{up} - b_{vp} + d_{uv}$$
$$C_{uvp}^{out} \geq 0$$

Finally the code overlay region should be no less than any actor code size that is mapped to p and not always present in the on-chip SPM.

$$\forall v \in V : C_p^{overlay} \geq (b_{vp} - z_{vp}) * C(v)$$

5.7.4 Cost functions

- The completion time of scheduling actor v due to computation cost and code overlay overhead is given by

  $$\forall u, v \in V | u \neq v : \Delta_v \geq \Delta_u + d(v) + \gamma_v + (s_{uv} + y_{uv} - 2) * MAX_VAL$$
  $$\Delta_v \geq d(v) + \gamma_v$$

  where MAX_VAL is a large constant and is given by $\sum_{v \in V}(d(v) + T_v)$. In the above inequalities, s_{uv} and y_{uv} equal 1 captures each actor u that is assigned to the same PE as actor v and is scheduled before v. The earliest start time of actor v due to processor workload is Δ_u. Therefore the completion time of v is given by its earliest start time plus its run time and code overlay overhead (if any), which is given by $\Delta_u + d(v) + \gamma_v$.

- The completion time of scheduling actor v due to inter-pipeline communication costs is given by

  $$\forall u, v \in V | \exists e : u \rightarrow v : \Delta_v \geq d(v) + (x_{uv} - y_{uv} - d_{uv}) * T_e$$

  In the above inequality, x_{uv} equals 1 and y_{uv} equals 0 captures the scenario when actor u and v are assigned to two separate pipeline stages and two different PEs. In this case, if d_{uv} equals 0 (no double buffering), then a communication cost of T_e between actor u and actor v is encountered. T_e is given by

  $$T_e = \begin{cases} T_{init}, \text{ if } C(e) \leq D_{init} \\ T_{init} + T_{slope} * (C(e) - D_{init}), \text{ otherwise} \end{cases} \quad (5.3)$$

- The completion time of scheduling actor v due to intra-pipeline communication costs is given by

  $$\forall u, v \in V | \exists e : u \rightarrow v : \Delta_v \geq \Delta_u + T_e + d(v) - (w^r_{uv} + y_{uv}) * MAX_VAL$$

  In the above inequality, w^r_{uv} and y_{uv} both equal 0 captures the scenario when there is an edge e directing from actor u to actor v with no delay and actor u, v are assigned to different PEs. In this case, actor v cannot execute until actor v has completed its execution and transferred its data to u.

5.7.5 Objective function

Finally the objective function is given by $Minimize\ \Delta$, where Δ is given by

$$\forall v \in V : \Delta \geq \Delta_v$$

5.8 RTEM Heuristic Approach

Although the ILP approach provides us high quality solutions, it suffers from very long algorithm run time for large input sets. In this section, we introduce a retiming heuristic approach (RTEM heuristic) that is able to generate comparable results with RTEM ILP in a matter of seconds. The main routine of our heuristic approach has three components, namely AlgorithmII, AlgorithmFEAS, and AlgorithmRTEM. AlgorithmII schedules actors in a retimed graph G_r onto P with the objective of minimizing its initiation interval (II). II defines the smallest time distance that any two consecutive instances of an iterative program can be scheduled. In our problem scenarios, II is equivalent to the steady-state execution time. AlgorithmFEAS tests whether a given II

1: Initialize the workload of each PE $\Delta(p)$ to be zero.
2: Let G_0 be a subgraph of G_r with all actors in G_r and all these edges where $w_r(e) = 0$.
3: Perform a topological sort on all actors in G_0 and store the sorted result in V_s.
4: for all actor $v \in V_s$ do
5: for all $p \in P(v)$ do
6: Calculate $\Delta(p)$, assuming that actor v is scheduled on p.
7: end for
8: Identify p_{min} that results in the smallest $Max_{p \in P}\Delta(p)$. Schedule actor v on p. Update its data memory $C_{data}(p)$, code memory $C_{code}(p)$ and work load $\Delta(p)$. Set the completion time of v, $\Delta(v)$, to be the updated workload of p.
9: end for
10: return $Max_{p \in P}\Delta(p)$

Algorithm 13: AlgorithmII(G_r, P)

is achievable by iteratively calling AlgorithmII and adjusting the retiming at each iteration. AlgorithmRTEM resides at the highest level and calculates the smallest II achievable through a binary search. In the remainder of this section, we discuss our RTEM technique following a bottom-up manner.

5.8.1 AlgorithmII

The input to AlgorithmII is a retimed graph $G_r < V, E, d, w_r >$[7] and an architecture configuration P. AlgorithmII iteratively schedules each actor in G_r onto P with the objective of minimizing the maximum workload among all PEs in P (equivalent to minimizing the final II). In algorithm AlgorithmII, we first initialize the workload of each PE, $\Delta(p)$, to be zero. Then a subgraph G_0 of G_r is constructed by including all actors in G_r and exactly these edges with $w_r(e) = 0$ (line 1-2). After that a topological sort is performed to obtain

[7] $w_r(e)$ discussed in this section is equivalent to $w^r_{uv}|e : u \rightarrow v$ in the previous ILP formulation section. We change to the current denotation to achieve consistency with the existing literature on retiming.

a sequence of totally ordered actors in G_0. In the total order, if there is an edge from actor u to v, then u must precede v. If the sequence of u and v is immaterial, then we enforce that the actor with a larger workload precedes the actor with a smaller workload. The total order is obtained to respect the data dependencies (line 3). For each actor in the total order, we calculate the resulting workload of scheduling v on p for each $p \in P(v)$. $P(v)$ is the set of PEs that v could be scheduled on (line 5-7). The PE that results in the smallest workload among all PEs in P is selected to schedule v. The code memory, data memory and workload of that PE is then updated accordingly. Then the completion time of actor v is set to the current updated workload (line 8). The calculation of the updated workload for scheduling v on p is given by

$$\Delta(p)' \leftarrow d(v) + Max\{\tau_c(v), \Delta(p) + \tau_o(v)\}$$

In the above equation, $d(v)$ is the delay/run time of actor v. $\tau_c(v)$ models the earliest start time of v due to data dependencies. $\Delta(p)$ indicates the workload of p before scheduling v on it and $\tau_o(v)$ indicates the code overlay overhead of scheduling v on p. $\Delta(p) + \tau_o(v)$ models the earliest start time due to processor workload. The calculation $\tau_c(v)$ and $\tau_o(v)$ is given in the following,

5.8.1.1 Calculation of $\tau_c(v)$

The earliest start time of actor v due to data dependencies is given by $\tau_c(v) = Max_{\forall e:u \rightarrow v}\{\tau_{inter}(v), \Delta(u) + \tau_{intra}(v)\}$. u is any producer of v and $\Delta(u)$ is the completion time of u. By the time we try to schedule v on p, u has already been scheduled, therefore $\Delta(u)$ is known to us. For inter-pipeline data dependencies, the producer and consumer that are scheduled on two different PEs can execute simultaneously. For intra-pipeline data dependencies, since

there is no retiming delay between the producer and consumer, the consumer cannot start execution until its producer finishes. Therefore, in this case $\Delta(u)$ is added to the cost function. Equation (5.4) and (5.5) in the following details the calculation of $\tau_{inter}(v)$ and $\tau_{intra}(v)$

$$\tau_{inter}(v) = \max_{\forall e:u \to v, w_r(e) \geq 1, p(v) \neq p(u)} \{T_e\} \tag{5.4}$$

$$\tau_{intra}(v) = \max_{\forall e:u \to v, w_r(e) = 0, p(v) \neq p(u)} \{T_e\} \tag{5.5}$$

In the above equations, u is any producer of v, $w_r(e)$ indicates the retiming delay on edge e, p indicates the PE an actor is scheduled on and T_e indicates the cost of transferring a data size of C_e through DMA. T_e is given by Equation (5.3).

5.8.1.2 Calculation of $\tau_o(v)$

In our memory partition of a processor, we conservatively allocate an overlay region of C_{fmax}. C_{fmax} is the largest code size among all actors. As long as the SPM can accommodate the temporary data and the code overlay region, any actor can be scheduled on that PE. We update the code and data memory of p after scheduling v on it by,

$$C_{code}(p)' \leftarrow C_{code}(p) + C(v) \tag{5.6}$$

$$C_{data}(p)' \leftarrow C_{data}(p) + C_{intra}(v) + C_{inter}(v) \tag{5.7}$$

$C_{intra}(v)$ and $C_{inter}(v)$ in equation (5.7) is given by,

$$C_{intra}(v) = \sum_{e:u \to v, p(v) = p(u)} (w_r(e) + 1) * C(e) \tag{5.8}$$

$$C_{inter}(v) = \sum_{e:u \to v, p(v) \neq p(u)} (w_r(e) + \delta) * C(e) + \sum_{e:v \to w, p(v) \neq p(u)} C(e) \tag{5.9}$$

where δ is set to 1 if $w_r(e) = 0$, δ is set to 0 otherwise.

In Equation (5.8), $C_{intra}(v)$ captures the buffer allocated for intra-processor edges whose consumer is v. In this case, the number of buffers to be allocated equals $w_r(e) + 1$, where $w_r(e)$ denotes the number of delays on edge $e : u \rightarrow v$. $C_{inter}(v)$ in Equation (5.9) captures the buffer usage of inter-processor incoming and outgoing edges. For each incoming edge of v, we allocate $w_r(e) + \delta$ buffers of size $C(e)$. δ is introduced such that at least one buffer is allocated on the consumer side. For an outgoing edge we allocate a buffer of size C_e. We assume no double buffering at this step. Later when double buffering is considered, additional buffers will be added to the producer and consumer whenever necessary. If there are extra retiming delays we can utilize, the smart double buffering procedure in our algorithm will handle them properly in the second iteration (refer to Section 5.8.3.).

If after scheduling actor v on PE p, $C_{code}(p)' + C_{data}(p)' + C_{fmax} < C_p$, then all program code and internal data buffers can fit into the on-chip SPM. In this case, no code overlay overhead is introduced ($\tau_o(v) = 0$). Otherwise, if $C_{data}(p)' + C_{fmax} > C_p$, then the SPM cannot accommodate the data and the overlay region, v must be scheduled on some other PE rather than p ($\tau_o(v) = +\infty$). Alternatively, if $C_{code}(p)' + C_{data}(p)' + C_{fmax} > C_p$ and $C_{data}(p)' + C_{fmax} \leq C_p$ then a code overlay overhead of T_v is introduced for scheduling actor v on PE p, where T_v is given by Equation (5.1).

5.8.2 AlgorithmFEAS

AlgorithmFEAS in our retiming heuristic determines whether a given initiation interval II is achievable. In AlgorithmFEAS, the retiming r of each actor v is first being initialized to be zero (line 1). Then we run the procedure described

1: For each actor v, initialize its retiming $r(v) \leftarrow 0$.
2: Initialize $count \leftarrow 0$
3: while $count < |V| - 1$ do
4: Construct G_r based on the retiming r of each actor.
5: Apply $AlgorithmII(G_r, P)$ to calculate $\Delta(v)$.
6: Iterate over each $v \in V$ following BFS manner.
If $\Delta(v) > II$, set $r(v) \leftarrow r(v) + 1$.
7: $count++$
8: end while
9: $MinII \leftarrow AlgorithmII(G_r, P)$
10: If $MinII \leq II$ return $MinII$; else return -1.

Algorithm 14: AlgorithmFEAS(G, P, II)

in line 4-6 for $|V| - 1$ iterations. At each iteration, we construct the retimed graph G_r based on the current retiming by setting $w_r(e) = w(e) + r(v) - r(u)$ for each edge. Upon termination, we apply AlgorithmII to calculate the completion time $\Delta(v)$ of each actor v. For actor v such that $\Delta(v) > II$, we move it to the next pipeline stage by increasing its retiming r by one (line 6). After $|V| - 1$ iterations we record the smallest II found so far as $MinII$. We return $MinII$ if it is smaller than the given II. Otherwise, -1 is returned to denote that the operation failed.

5.8.3 AlgorithmRTEM

AlgorithmRTEM in our approach utilizes a binary search to find the smallest achievable II. The lower bound L and upper bound U of the binary search are given by

$$L = \max\{\sum_{v \in V} \frac{d(v)}{|P|}, \max_{v \in V} d(v)\}, U = \sum_{v \in V} d(v) \tag{5.10}$$

In Equation (5.10), $\sum_{v \in V} d(v)/|P|$ is the lower bound imposed by limited number of PEs and $Max_{v \in V} d(v)$ is the maximum delay of any actor. U is the workload of p when all actors are assigned to it. In the algorithm,

we first copy the original value of N_{user} to N_{user_ori} and update N_{user} with $\min\{N_{user_ori}, |P|\}$ such that we can assign one retiming group[8] to one PE after scheduling. Then a binary search is carried out to calculate the smallest II achievable under a constraint of no more than N_{user} retiming groups. We examine whether a given II is achievable by applying AlgorithmFEAS. At this step we assume that each retiming group is allocated with one PE. If $N_{user_ori} = |P|$, then the algorithm directly returns the $MinII$ being found (line 11). Alternatively, if $N_{user_ori} < |P|$, we first find the retiming group with delay k that has the maximum parallelism (maximum width in a BFS) from the results of the previous step. We allocate $|P| - N_{user} + 1$ PEs to retiming group k and one PE for each other retiming group. Then we apply binary search again to find the smallest achievable II under the updated allocation. Finally, if $N_{user_ori} > |P|$, we explore the trade-offs between inter-pipeline communication costs and code overlay by introducing smart double buffering scheme between different pipeline stages.

5.8.3.1 Implementation of smart double buffering scheme

We implement the smart double buffering scheme by first calculating the number of retiming delays that are available for double buffering, which is given by $ExtraReg = N_{user_ori} - |P|$ (in our approach we apply smart double buffering only when $N_{user_ori} > |P|$). We don't need more than $|P|$ extra retiming delays as they are sufficient for introducing double buffering for every pipeline stage[9]. We calculate the cost saving of each PE through double buffering by

[8]A retiming group is a group of actors that have the same retiming delay in the retimed graph.

[9]Double buffering is only considered for retiming groups that are disconnected from each other in the retimed graph. If double buffering is introduced to a retiming group, then all the inter-pipeline communication for its outgoing edges are double buffered.

1: Store $N_{user_ori} \leftarrow N_{user}$ and update $N_{user} = \min\{N_{user_ori}, |P|\}$.
2: /* For a given II, test whether it is feasible by AlgorithmFEAS(G, P, II) */
$MinII \leftarrow$ BinarySearch of II from L to U.
3: if $N_{user_ori} < |P|$ then
4: Find pipeline stage k with maximum parallelism
5: Allocate $(|P| - N_{user} + 1)$ PEs to k, one PE to each pipeline stage $k' \neq k$
6: $MinII \leftarrow$ BinarySearch of II from L to U.
7: end if
8: if $N_{user_ori} > |P|$ then
9: Apply smart double buffering scheme to further reduce inter-pipeline data communication overhead
10: end if
11: return $MinII$

Algorithm 15: AlgorithmRTEM(G, P, N_{user})

$\tau_{saving} = \Delta'(p) - \Delta(p)$, where $\Delta'(p)$ is the workload of p after double buffering and $\Delta(p)$ is the original workload. $\Delta'(p)$ is derived by calling AlgorithmII with the new settings where $\tau_{inter}(v)$ is set to zero and $C_{inter}(v)$ is adjusted with

$$C_{inter}(v)' \leftarrow C_{inter}(v) + \sum_{e:u \rightarrow v, w_r(e)=1} C_e + \sum_{e:v \rightarrow w} C_e \tag{5.11}$$

In Equation (5.11), $u \rightarrow v$ and $v \rightarrow w$ are the edges that are double buffered. For every incoming edge of v that has only one buffer allocated, we add one buffer for implementation of double buffering. For every outgoing edge of v we allocate one extra buffer. We sort the retiming groups by decreasing savings and introduce double buffering to each retiming group iteratively. We terminate this procedure when there is no extra register to be allocated, or τ_{saving} becomes negative.

5.8.4 Complexity

The complexity of our AlgorithmII is $O(|E|)$ for a sparse SDF. Therefore AlgorithmFEAS and AlgorithmRTEM run in $O(|V||E|)$ and $O(|V||E|log_2 U)$ respectively.

5.9 Experimental Results

5.9.1 Experimental setup

In this section we evaluate the efficiency of our ILP and RTEM heuristic approaches by implementing a compiler framework and compared the performance results against existing approaches. Specifically, we consider twelve applications from the StreamIt compiler version 2.1.1 [57]. The StreamIt compiler takes a stream application and compiles it into multi-threaded C++ code. We instrumented the compiler such that it outputs an SDF representation of the application with the information described in Table 5.1. The details of each benchmark is given in Table 1.1. We employed a PlayStation 3 system that hosts an IBM Cell BE as the target hardware platform. In the PlayStation 3 system there are 6 SPEs available to the programmers. Therefore we have 7 programmable PEs in total with one PPE and 6 SPEs.

5.9.2 Overall performance comparison

We first utilized all 7 PEs available in the IBM Cell BE and compared our RTEM ILP and heuristic performance against the results from our previous CSMP ILP and heuristic approaches presented in [20]. The user specified number of pipeline stages is set to 14 in this experimental setup. The CSMP ILP

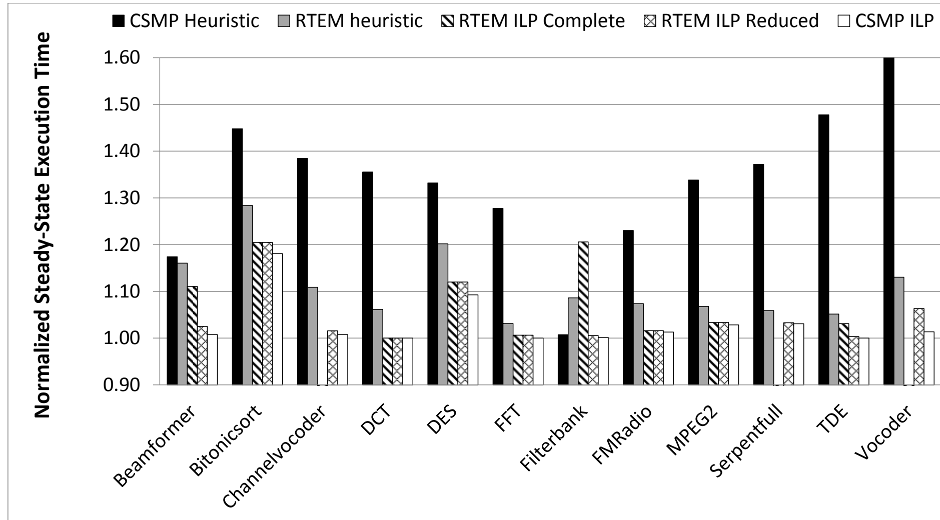


Figure 5.6: RTEM ILP and heuristic against CSMP ILP and heuristic approaches.

and heuristic employed fusion and fission operations for mapping and scheduling a stream program onto a multicore architecture. The resulting software pipeline stages for CSMP ILP and heuristic approaches can not be pre-specified and is determined by the solution. In the CSMP ILP and heuristic approaches, each actor is assigned to a separate batch in the initialization. Then an iterative fusion operation is performed to reduce the number of batches to $|P|$. They keep on merging two batches with the smallest workload[10] and replicates the stateless batch with the largest workload. Stateful filters were paid extra attention for their limitation on the optimizations that can be applied (no fission is allowed). Figure 5.6 shows the experimental results from CSMP ILP and heuristic, and RTEM ILP and heuristic approaches. The x-axis in the figure provides the twelve benchmarks we experimented with. The y-axis gives

[10]The workload here is the effective execution time considering computation costs, communication costs and code overlay overhead.

us the performance results for each technique normalized to its lower bound L. The calculation of lower bound L is given by Equation(5.10) in Section 5.8.3. In the figure, RTEM ILP has two sets of results, RTEM ILP complete and reduced. RTEM ILP complete models the complete ILP formulation we proposed in Section 5.7. Because this ILP is very complicated and takes a long time to run, we reduced it by removing the calculation of processor memory and code overlay. The reduction is reasonable under the current experimental set up since each SPE in the IBM Cell BE hosts a 256K SPM, which is sufficiently large compared to the benchmark code and data memory usage we experimented with. Later, we conducted a series of tighter SPM sizes to examine the impact of on-chip memory constraint and they are discussed in the next subsection.

As observed from Figure 5.6, our RTEM ILP reduced formulation achieves comparable results with the CSMP ILP approach and they both approach the lower bound. Our RTEM ILP reduced formulation has a little performance degradation compared to the CSMP ILP approach due to the constraint on the resulting number of software pipeline stages and the fact that no unrolling is introduced. The CSMP ILP approach on the other hand, implements implicit unrolling which results larger program and deeper software pipeline stages. In this case, the average number of pipeline stages for the CSMP ILP approach is 25, which is almost two times higher than our RTEM ILP. The CSMP ILP results are illegal under the constraint of no more than 14 pipeline stages. For our RTEM complete formulation, we terminated the ILP solver[11] after 48 hours and output the best integer solution generated so far. There are three benchmarks, Channelvocoder, Serpentfull and Vocoder,

[11] The ILP solver we adopted is the optimizer from FICO^{XM}Xpress Optimization Suite.

that did not generate any integer solution upon termination of the ILP solver and we leave them blank in the figure.

Our RTEM heuristic deviates from our RTEM ILP reduced formulation by less than 10% for all 12 benchmarks we experimented with. Further, our RTEM heuristic approach always outperforms the CSMP heuristic approach except for the Filterbank benchmark. The reason for our RTEM outperforming the CSMP heuristic is that our RTEM works at the granularity of a single actor while the CSMP heuristic tries to combine filters into a limited amount of batches. As the CSMP heuristic evolves, the granularity that the algorithm operates on becomes larger and larger, which makes it harder and harder to balance the workload. On average, our RTEM heuristic outperforms the CSMP heuristic by more than 15%. More importantly, our RTEM heuristic approach imposes that the generated schedule contains no more than a user specified number of pipeline stages. In this experimental setup, the schedules derived from our heuristic approach is guaranteed to have no more then 14 software pipeline stages. The CSMP heuristic approaches has no such guarantees. The average number of pipeline stages for the CSMP heuristic is 23, which is a violation of the user specified constraint. The average algorithm run time of our RTEM heuristic approach is 2 seconds, which is at the same level as CSMP heuristic approach (6 seconds), and much faster than the CSMP ILP approach (2837 seconds).

5.9.3 Comparison with different SPM sizes

The previous results were conducted under the SPM size of 256KB that didn't introduce any code overlay overhead. Next we constructed another set of experiments that evaluate our technique under tight SPM constraints. We

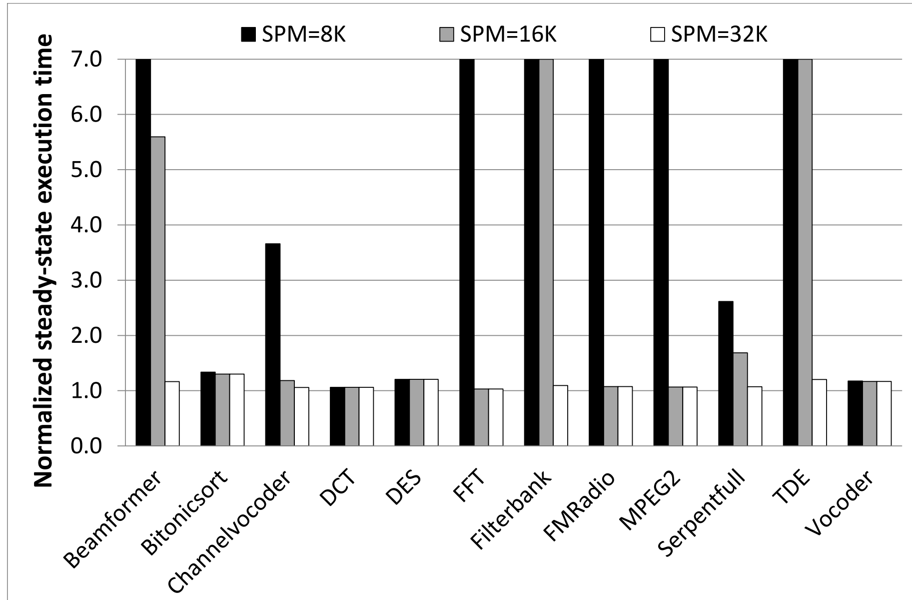


Figure 5.7: Performance comparison with different size of SPMs.

shrunk our SPM size to 32KB, 16KB and 8KB respectively and applied our RTEM heuristic technique under the new configurations. The resulting performances are presented in Figure 5.7. We only provide the results from our RTEM heuristic because the RTEM ILP complete formulation run time is very long and prohibits us from collecting sufficient data within a reasonable time. There are four benchmarks, Bitonicsort, DCT, DES, and Vocoder that introduce zero overlay overhead even under 8KB memory, which indicates that the total memory usage of their program code and internal data buffers is less than 8KB. The rest of the benchmarks encountered code overlay overhead at a certain SPM size and quickly converged to the solution that all the actors must be scheduled on PPE only. The results indicate that when the code overlay is encountered, the SPM becomes very precious. Under such situa-

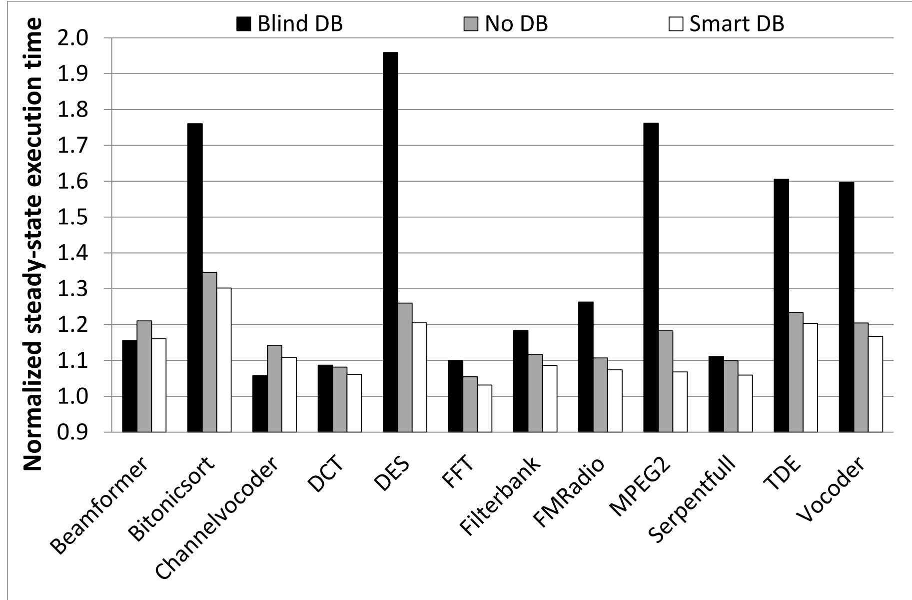


Figure 5.8: Smart double buffering against blind double buffering and no double buffering.

tion, arbitrarily introducing double buffering may in fact degrade the quality of the results.

5.9.4 Comparison with different double buffering schemes

In this section we provide the comparison of our RTEM heuristic performance results under different double buffering schemes. Figure 5.8 validates the efficiency of our smart double buffering scheme by comparing its results against no double buffering and blind double buffering schemes. In this experimental set up we specified N_{user} to be 9. In no double buffering scheme, we simply change the number of retiming groups from 9 to 7. In the blind double buffering scheme, we utilized $N_{blind} = \lfloor N_{user}/2 \rfloor$ to substitute the original number of admissible pipelines stages and enabled double buffering for all inter-pipeline

data communications. Figure 5.8 demonstrates the results from these three schemes. We observe that blind double buffering scheme generates the worst results most of the time. This is because we are sacrificing the number of admissible pipeline stages in order to amortize the inter-pipeline communication costs. When the number of pipeline stages is smaller than the number of PEs, we have to schedule a particular retiming group across several PEs. The performance is heavily dependent on the parallelism that inherently exists in the original program. There are two special benchmarks, Beamformer and Channelvocoder, that achieved their best performance under the blind double buffering scheme. The counter-intuitive results are due to the fact that there are large split-join and/or duplicate-join structures that exist in the two benchmarks and they provide sufficient parallelism. A more detailed discussion on this counter-intuitive behavior can be found in Section 5.9.5. Our RTEM outperforms no double buffering scheme in that no double buffering scheme ignores the extra retiming delays available, therefore overlooks the opportunities to further reduce the inter-pipeline communication costs. As a result, our smart double buffering scheme outperforms blind double buffering scheme by more than 20% and no double buffering scheme by 5%.

5.9.5 Comparison with different number of pipeline stages and PEs

In this section, we demonstrate how our RTEM heuristic performs under different number of admissible pipeline stages and PEs. We first utilized 7 PEs and varied the number of feedback registers from 3 to 11 and the results are shown in Figure 5.9. In Figure 5.9 the x-axis provides the benchmark that we experimented with. The y-axis presents the steady-state execution time normalized to the lower bound L. From Figure 5.9, we observe a trend of reduced

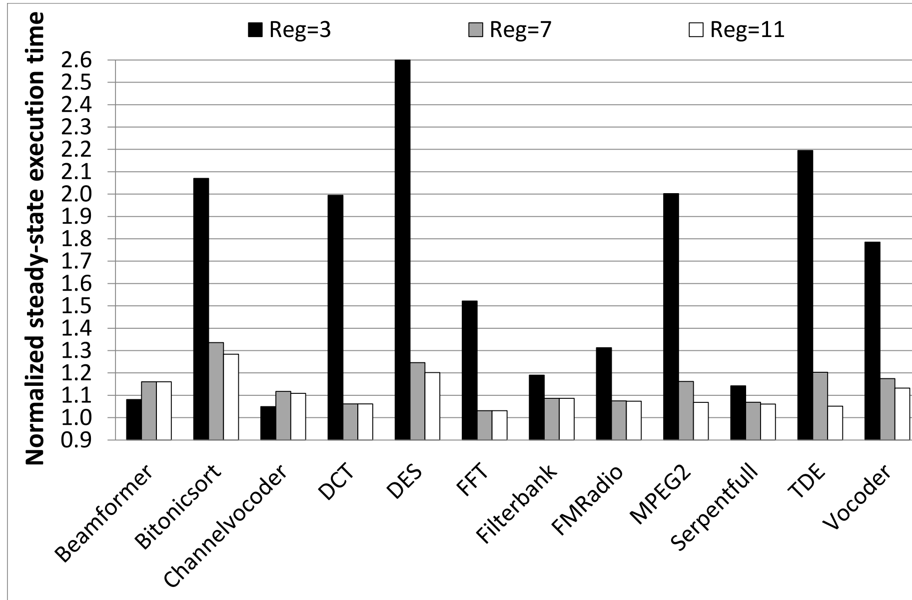


Figure 5.9: Performance comparison with different pipeline stages.

normalized execution time for most of the benchmarks when we increase the number of admissible pipeline stages. The average performance for 11 admissible pipeline stages is 7% better than 7 pipeline stages and is 34% better than 3 pipeline stages. There are two special benchmarks, Beamformer and Channelvocoder that achieved their best performance under 3 pipeline stages. This is because both these benchmarks contain large split-join structures that provide sufficient embarrassing parallelism. No intra-pipeline dependency is introduced for these benchmarks, even we were scheduling one retiming group across several PEs. In another experimental setup, we fixed the number of pipeline stages to 7 and varied the number of PEs and show the performance results in Figure 5.10. Note that the performance results for $PE = 3$ are normalized to the lower bound L, and for $PE = 7$ and $PE = 11$, they are

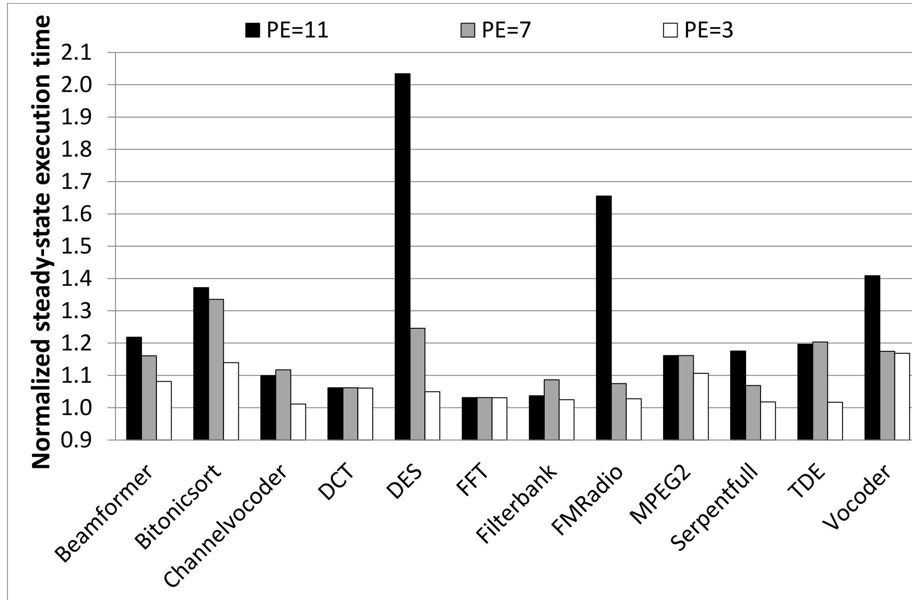


Figure 5.10: Performance comparison with different number of PEs.

normalized to $PE = 3$. This experimental setup examines how our RTEM heuristic scales with the number of PEs in a multicore architecture. We saw a trend of reduced normalized execution time when we increased the number of PEs from 3 to 11. To be more specific, we achieved an performance gain of more than 50% by increasing the number of PEs from 3 to 7. Further increasing the number of PEs to 11 gave us another performance improvement of around 30%. The results demonstrated that our heuristic scales with the number of PEs and the trend only slows down when the existing parallelism of the program becomes the limiting factor.

5.10 Summary

We proposed RTEM ILP and heuristic approaches for compilation of stream programs onto SPM based multicore processors in this chapter. The ILP per-

forms retiming and actor to PE mapping that schedule a stream program onto a multicore architecture. The ILP formulation effectively evaluates load balancing, computation communication overlap, smart double buffering and code overlay, thus provides us high quality solutions. Although the ILP approach is able achieve very high quality solutions, the algorithm run time could be very long for large input sets. In the second part of this chapter, we proposed a RTEM heuristic that solves the same problem in a matter of seconds and achieves comparable results. Our RTEM ILP and heuristic approaches inherently handles cycles in the stream applications and maximizes the throughput under a user specified number of pipeline stages. Experimental results show that our RTEM ILP approach achieves comparable results with the CSP ILP approach even with the constraints on the resulting number of software pipeline stages. Further, our RTEM heuristic approach outperforms the existing CSMP heuristic approach by 15% on average. Our future work will address unrolling stream programs with loop structures and data overlay between on-chip SPMs and the off-chip main memory.

Chapter 6

UNROLLING AND RETIMING OF STREAM PROGRAMS ON SPM BASED MULTICORE PROCESSORS

The previous discussed CSMP ILP/heuristic [20] and RETM heuristic [14] approaches solve the problem of compilation of stream programs onto SPM based embedded multicore processors with the objective throughput optimization. Both approaches exploit load balancing with computation/communication overlap, double buffering and code overlay. The CSMP approaches utilize fusion and fission operations to map a stream program onto embedded multicore processors. In CSMP approaches, a loop structure is treated as a high-level actor and the resulting software pipeline stages of the schedule is uncontrollable. Also, as the fusion operation proceeds, the granularity that the algorithm can operate on becomes larger and larger, which makes it harder and harder to achieve load balancing. RTEM heuristic adopts the retiming technique that is traditionally seen in circuit design. It can impose an upper bound on the resulting number of software pipeline stages. Further, RTEM heuristic always works on the granularity of a single actor, thus has a better potential to achieve load balancing. It also naturally handles loop structures in a stream program, which is a property inherited from traditional retiming techniques. However, RTEM heuristic does not scale with the number of PEs. When the number of PEs is very large and the existing parallelism in a stream program is comparably limited, RTEM fails to generate high quality solutions.

In this chapter, we propose optimization technique that retime and unroll stream programs onto SPM based embedded multicore architectures. The proposed approach inherits all the beneficial properties from RTEM heuristic,

namely it efficiently addresses the limited on-chip SPM capacities and memory access delays. When the code and data size is larger than the given SPM capacity, it balances double buffering and code overlay in such a way that the overall performance is optimized. Further, it also inherently handle cycles that may present in a stream program. Finally, It can also accept an upper bound on the number software pipeline stages to be generated.

Given the above discussed requirements, we propose to perform unrolling and retiming simultaneously for scheduling stream programs onto SPM based embedded multicore architectures. The proposed algorithm performs unrolling and retiming iteratively. It terminates when no further performance improvement can be achieved. At each iteration, the proposed approach will first unroll the stream program by the given factor. Currently, the unrolling algorithm discussed by Chao et al. [51] is considered. To ensure that two executions of a stateful actor never overlap, a feedback edge with one delay will be introduced to each statefull actor before unrolling. Then the minimum steady-state execution time achieved at each unroll factor is calculated through a binary search. The actual number of retiming groups generated by the retiming procedure determines whether a list scheduling or smart double buffering is to be implemented. List scheduling schedules one retiming group onto several processors to fully utilize the hardware resources. Since all actors belong to the same retiming group are in fact within the same software pipeline stage, an intra-pipeline communication overhead could be encountered for a pair of producer and consumer that is scheduled on different PEs. When double buffering is introduced, we need to allocate extra data buffers. The data memory increase could result in higher code overlay overhead and occasionally even infeasible solution.

In order to efficiently cope with the increased code and data size in an unrolled graph, we propose to extend our RTEM heuristic with code pre-fetching and data overlay. Code pre-fetching dramatically reduces the code overlay overhead by overlap the DMA transfers of code with actor computation. Data overlay reduces the data buffer usage when the SPM is extremely limited. We discuss the code pre-fetching scheme and data overlay scheme in the following. In our basic code overlay scheme, only one overlay buffer is allocated for all actors that are (i) scheduled on the current processor p and (ii) mapped to the off-chip main memory. Figure 6.1 (A) depicts the program behavior of this code overlay scheme. In the example, ActorM is scheduled to execute next and ActorN is currently present in the overlay buffer. At Time 1, the DMA engine brings ActorM from the off-chip memory and evicts ActorN. As soon as the DMA transfer completes, we invoke the execution of ActorM (at Time 2). Since the DMA engine and the execution unit in each PE operate independently, we can pre-fetch ActorM while executing ActorN, as shown in Figure 6.1 (B). Code pre-fetching improves performance at the expense of one extra buffer allocation. When the SPM capacity is extremely limited, data overlay will be introduced to stream data back to the off-chip main memory. In our data overlay scheme (if triggered), we only allocate one buffer for each edge that belongs to processor p and push the rest of buffers back to the off-chip main memory. Figure 6.1 (C) depicts the data memory without data overlay and Figure 6.1 (D), with data overlay. Data overlay reduces the data memory usage at the expense of circling through the off-chip memory for every data produced/consumed.

In this chapter, we propose a heuristic approach that automatically compiles a stream application onto embedded multicore processors with the

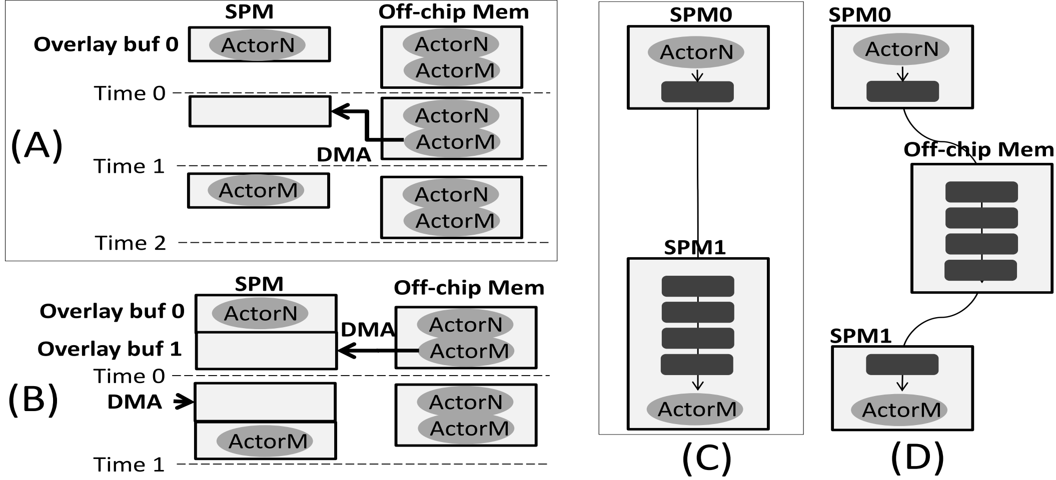


Figure 6.1: Code overlay and data overlay.

objective of throughput optimization. Our heuristic approach is able to:

- Unroll and Retime Stream formats onto Embedded Multicore processors (URSEM) with the objective of throughput maximization.
- Exploit trade-offs among code overlay, data overlay and double buffering, thus efficiently address the limited on-chip SPMs and DMA delay.
- Schedule stream applications with loop structures and also accept an upper bound on the resulting software pipeline stages.

6.1 Problem Description

The input to our problem is composed of an SDF representation of the stream application and a hardware description of the target architecture. The SDF specification is given by $G < V, E >$ where V in G represents actors/filters and E represents edges. Prior to invoking our optimization technique, we transform the given SDF into a single appearance SDF (discussed in Section

5.5). The resulting SDF and the target architecture is described in Table 5.1. DMA latency is approximated by $T_c(x) = T_{init}$, if $x \leq D_{init}$ and $T_c(x) = T_{init} + (x - D_{init}) * T_{slope}$, otherwise. In the equation, $T_c(x)$ denotes the latency of transferring x bytes of code/data. The output of our technique is an actor to PE mapping and a software pipelined schedule together with double buffering, and code/data overlays that maximize the throughput.

The classical retiming techniques [50] [25] alters delays among various function units of a circuit and retains its original logic. The retiming approaches are intriguing to our problem in that they handles loop structures inherently. By properly constraining the retiming delays, an upper bound on the resulting number of software pipeline stages can be imposed. Unfortunately, the existing approaches that employ retiming for throughput optimization are not directly applicable to our problem due to memory access delays, double buffering, limited on-chip SPMs, and code/data overlays. In our problem instances, code/data overlays reduce memory usage by sharing the same physical memory with different code segments and data sets over time. In our basic code overlay scheme, we allocate one overlay buffer for all actors that are (i) scheduled on processor p and (ii) mapped to the off-chip main memory. Figure 6.1 (A) depicts the program behavior of this code overlay scheme. In the example, ActorM is scheduled to execute next and ActorN is currently present in the overlay buffer. At Time 1, the DMA engine brings ActorM from the off-chip memory and evicts ActorN. As soon as the DMA transfer is completed, we invoke the execution of ActorM (at Time 2). Since the DMA engine and the execution unit in each PE operate independently, we can pre-fetch ActorM while executing ActorN, as shown in Figure 6.1 (B). Code pre-fetching improves performance at the expense of one extra buffer

allocation. In our data overlay scheme (if triggered), we allocate one buffer for each edge that belongs to processor p and push the rest of buffers back to the off-chip main memory. Figure 6.1 (C) depicts the data memory without data overlay and Figure 6.1 (D), with data overlay. Data overlay reduces the data memory usage at the expense of circling data through the off-chip memory.

In the rest of this chapter, Section 6.2 discusses related work. Section 6.3 presents our URSEM approach. Section 6.4 provides experimental results and Section 6.5 concludes the paper.

6.2 Related Work

Several previous approaches have addressed the problem of implementing stream workload on embedded multicore processors. A hierarchical framework for scheduling SDF onto multicore processors was discussed by Pino et al. [66]. More recently Ostler et al. [61] proposed techniques for mapping stream based applications onto network processing processors. Liao et al. [52] investigated parallelizing Brook language onto general purpose multicore processors through data and code transformations. Stratton et al. [70] developed a framework MCUDA that executes CUDA language on shared memory multicore processors. In contrast to the above approaches, our technique focuses on embedded multicore processors that incorporate SPMs. In addition to actor to PE mapping and double buffering, we also face the challenge of dynamic management of the limited on-chip SPM for program code and data.

There have been approaches that concentrate on automatic compilation of stream applications onto multicore processors. Gordon et al. [31] explored the trade-offs between data and task level parallelisms and developed a heuris-

tic to generate multi-threaded code for the RAW architecture. Hormati et al. [34] [35] proposed compiler frameworks for mapping stream languages onto GPUs and heterogeneous architectures. Kudlur et al. [48] came up with an ILP that unfolds and partitions a stream application onto multicore processors. An improved version of this work that addresses memory constraint was later presented by Choi et al. [24]. Our approach is distinguished from the above approaches in that we explore the trade-offs between double buffering, code overlay and data overlay, thus efficiently address the SPM constraint.

The previous work that comes closest to us is the CSMP [20] and RETM [15] approaches proposed by Che et al.. The CSMP approach utilizes batch fusion and fission operations to map an SDF model onto embedded multicore processors. In this work, a cycle is treated as a high-level actor and the resulting software pipeline stages of the schedule is uncontrollable. Our URSEM approach on the contrary handles cycles inherently and can accept an upper bound on the resulting software pipeline stages. The proposed RTEM approach compiles a stream program onto SPM based multicore processors through retiming. Compared with this work, our URSEM performs unrolling and retiming simultaneously. As a result, we can achieve better performance and scalability. Further, we implemented our code overlay with pre-fetching, which reduces the overall overhead. In the case when the SPM capacity is extremely restricted, we also introduce data overlay.

6.3 URSEM Heuristic Approach

Prior to entering our URSEM heuristic approach, we first discuss the preprocessing steps that are performed to construct a single appearance SDF.

6.3.1 Pre-processing

In our URSEM heuristic, we require a single appearance SDF. Given a regular SDF without loop structures, we can simply combine all executions of the same actor in a PASS into a high-level execution to derive a single appearance SDF. In the case when loop structures are present, we block process an actor as many times as permitted by the delays on its feedback loop edge. The resulting delay in the single appearance SDF is given by $\lfloor w(e)/N_v \rfloor$, where $w(e)$ denotes the delays/tokens on feedback loop edge e and N_v denotes the tokens consumed by actor v (consumer of edge e) in the steady-state execution. If $w(e) < N_v$ the entire loop is treated as a one high-level actor. The resulting actor is stateful and the delays on e becomes a state variable. For all stateful actors in the program, we add a self-loop with one delay to ensure that two executions of a stateful actor never overlap. Section 5.5 discusses how we resolves cycles that might be present in a stream program.

6.3.2 URSEM heuristic algorithm

In this section, we discuss the high level routine of our algorithm as illustrated in AlgorithmURSEM. It iteratively performs unrolling and retiming to schedule a stream format G onto an embedded multicore processor P. It terminates when no further performance improvement can be achieved or the unroll factor exceeds $|P|$, Line 19. II_f in the condition indicates the minimum II achieved by scheduling G with an unroll factor f. We divide II_f by f to derive the corresponding II of the original program G. At each iteration, we first unroll the stream program by the given factor and store the unrolled graph in G_f. We utilize the graph unrolling algorithm by Chao et al. [51]. Then the minimum II

```
1  II_f ← +∞, f ← 0;
2  repeat
3      II_{f−1} ← II_f, f ← f + 1, G_f ← Unroll(G, f);
4      /* retime with min{|P|, N_user} pipeline stages */
5      l ← 0, db ← 0, N ← min{|P|, N_user} ;
6      II_f ← BinarySearch(G_f, P, N, l, db);
7      |RGs| ← r_max(v) − r_min(v) + 1;
8      if |RGs| < |P| then
9          /* retime with list scheduling */
10         l ← 1, db ← 0, N ← |RGs| ;
11         II_f ← BinarySearch(G_f, P, N, l, db);
12     else
13         /* retime with double buffering */
14         l ← 0, db ← 1, N ← N_user, stages ← 0;
15         while stages < min{|P|, N_user − |P|} do
16             Identify RG (r) that results in min II_f by calling
               II_f ← BinarySearch(G_f, P, N, l, db);
17             Update II_f and set_db(r, 1);
18             stages ← stages + 1;
19 until II_f/f ≥ II_{f−1}/(f − 1) or f > |P|;
20 return II_{f−1}/(f − 1);
```

Algorithm 16: AlgorithmURSEM(G, P)

achieved at unroll factor f (II_f) is calculated through a binary search. Binary search is conducted within the range of $\{\sum_{v \in V_f} C(v)/|P|, \sum_{v \in V_f} C(v)\}$, and AlgorithmRDL is invoked to check whether a given II is achievable. The parameters G_f, P, N, l, and db passed to the binary search capture the unrolled graph, the multicore architecture, the maximum number of retiming groups (RGs) to be generated, and whether list scheduling and double buffering are enabled, respectively. RG is defined as a group of actors that have the same retiming delay ($r(v)$). In the remainder of this paragraph, we focus on the high-level overview of our URSEM heuristic[1]. The actual number of RGs generated by the retiming procedure is given by $|RGs| \leftarrow r_{max}(v) - r_{min}(v) + 1$,

[1]The discussion of AlgorithmRDL is provided in Section 6.3.3.

where $r_{max}(v)$ and $r_{min}(v)$ are the maximum and the minimum retiming delays. $|RGs|$ could be less than P due to inter-iteration dependencies or user specified limitation on software pipeline stages. In AlgorithmURSEM, a list scheduling is implemented to improve on the initial solution if $|RGs| < |P|$, Line 11. Otherwise if $|RGs| = |P|$, a smart double buffering scheme is implemented, Line 13-18. We greedily introduce double buffering to each RG. The RG that provides us with the most significant performance improvement is selected at each iteration. The process terminates when all RGs are double buffered or there is no extra pipeline stage left, Line 15.

6.3.3 AlgorithmRDL

AlgorithmRDL determines whether a given II is achievable for scheduling graph G on P through retiming. The retiming delay of each actor is set to zero in the initialization. Then the algorithm enters an iterative procedure where we construct G_0 by preserving all actors from G_r and exactly those edges with $w_r(e) = 0$, Line 3. A scheduling order is generated from G_0 such that if there is an edge directing from actor u to v in G_0, then u must be scheduled before v. When the scheduling sequence of u and v is immaterial, we schedule them based on their priorities. The priority of an actor is given by the maximum priority among all its children plus its own computation delay ($d(v)$). There is no cycle in G_0, thus a fixed priority can be generated for each actor following a bottom-up manner. We calculate the completion time of each actor $\Delta(v)$ by applying AlgorithmDeltaCD. AlgorithmDeltaCD schedules a retimed graph G_r onto a multicore processor P with code and data overlays. We discuss it in Section 6.3.4. Starting from the 5th line of AlgorithmRDL, we compare the completion time of each actor with a given II. If its completion time is larger

than II, we increase the retiming delay of v. In the algorithm, $get_db(r(v))$ returns one if $r(v)$ is double buffered (zero otherwise). If double buffering is enabled for $r(v)$ then $r(v) \leftarrow r(v) + 2$, indicating that one addition delay is allocated for DMA transfers. Otherwise, we increase $r(v)$ by 1, Line 8. We only alter the retiming delay of an actor when there are enough delays left to be scheduled, Line 7. If an actor's completion time is larger than II and its retiming delay cannot be altered due to lack of pipeline stages (captured by $r(v) - r_{min}(v) \geq N_{user}$, where $r_{min}(v) \leftarrow \min_{v \in V} r(v)$) then we immediately return -1 (failure), Line 10. At each iteration, after the retiming process, we compute the new retimed graph G_r by setting $w_r(e) \leftarrow w(e) + r(v) - r(u)$ for each edge e in the unrolled graph. For the purpose of double buffering, we occasionally increase the retiming delay of an actor by two instead of one. In this case, the validity of the retiming needs to be verified. If an invalid retiming is found, we return -1, indicating that double buffering cannot be introduced. This scenario could happen when we have loop structures with limited delays on their feedback edges. Upon termination of the iterative retiming procedure, we apply AlgorithmDeltaCD to calculate the resulting II and store it to II_{min}. Finally, we return II_{min} if $II_{min} \leq II$ and return -1 (failure), otherwise.

6.3.4 AlgorithmDeltaCD

In this section we discuss AlgorithmDeltaCD which schedules a retimed graph onto embedded multicore processor with code pre-fetching and data overlay.

6.3.4.1 Construction of RG to PE mapping

In AlgorithmDeltaCD, $P(r(v))$ denotes the set of processors that an actor with retiming $r(v)$ could be scheduled on. If $l = 0$, then $|P(r(v))| = 1$. In this case,

```
1  ∀v ∈ V_f set r(v) ← 0;
2  for i = 0 to |V_f| − 1 do
3      Construct G_0 and a scheduling order S;
4      ∀v ∈ V_f, apply AlgorithmDeltaCD to calculate Δ(v);
5      forall the v ∈ S do
6          if Δ(v) > II then
7              if r(v) − r_min(v) < N_user − (get_db(r(v)) + 1) then
8                  r(v) ← r(v) + (get_db(r(v)) + 1);
9              else
10                 return −1;
11     Compute G_r based on the retiming r of each actor v;
12     ∀e ∈ E_f, if w_r(e) < 0 then return −1;
13 II_min ← AlgorithmDeltaCD(G_r, P);
14 if II_min ≤ II then return II_min else return −1;
```

Algorithm 17: AlgorithmRDL(G_f, P, II, N, l, db)

```
1  ∀v ∈ V, set Δ(v) ← d(v);
2  Calculate RG to PE mapping (P(r(v)));
3  forall the v ∈ S do
4      Schedule v on p ∈ P(r(v)) (list scheduling if l = 1);
5      Update code/data memory, memory state of p;
6      Calculate Δ(v) and set Δ(p) ← Δ(v);
7  return Max_{v∈V} Δ(v);
```

Algorithm 18: AlgorithmDeltaCD(G_r, P, S, l, db)

each RG is mapped to exactly one processor. Otherwise ($l = 1$), the RG with the maximum parallelism[2] is scheduled on $|P(r(v))| = |P| - |RGs| - 1$ processors with list scheduling [55] and the remaining RGs are scheduled on one processor each. We schedule each actor v following S and update the completion time of actor v ($\Delta(v)$) and the workload of processor p ($\Delta(p)$) accordingly. The calculations of $\Delta(v)$ and $\Delta(p)$ requires the knowledge of code/data memory usage and the memory state of processor p. Their calculations are provided below.

[2]RG that has the maximum width following a BFS search.

6.3.4.2 Calculation of code, data memory usage

The calculation of code memory of processor p after scheduling v is given by,

$$C_{code}(p) \leftarrow C_{code}(p) + C(v)$$

The processor data memory after scheduling v on p is given as follows. For every edge e that has v as a consumer

$$\begin{aligned} &\text{if } get_state(p) \neq \text{DATA_OVERLAY} \\ &\qquad C_{data}(p) \leftarrow C_{data}(p) + w_r(e) * C(e) \\ &\text{else} \quad C_{data}(p) \leftarrow C_{data}(p) + C(e) | e : u \rightarrow v, u \notin p, v \in p \end{aligned} \tag{6.1}$$

For every edge e that has v as a producer

$$C_{data}(p) \leftarrow C_{data}(p) + (1 + get_db(r(v))) * C(e) \tag{6.2}$$

In Equation (6.1), $get_state(p)$ returns the memory stage of processor p. The memory state of a processor p could be SF (sufficient), CO (code overlay), DO (data overlay), and IF (infeasible). The transitions of memory states and their conditions are illustrated in Figure 6.2. The memory state of each processor is first initialized to be SF. Then as we keep on scheduling actors on p, the processor memory state changes to CO when the code and data size becomes larger than $C(p)$. In memory state CO, $\tau_o(v)$ is recalculated for every v that has been scheduled since an actor that has been mapped to the on-chip SPM may be relocated to the off-chip memory when we try to schedule another actor on p. If the SPM is only able to accommodate the program internal data and two overlay buffers $(2 * C_{max}(v))$, the memory state changes to DO. $C_{max}(v)$ denotes the largest actor code size. We conservatively allocate the overlay buffer size to be $C_{max}(v)$ such that every actor can be placed in it.

The data memory and $\tau_o(v)$ for each actor being scheduled is recalculated in this case with data overlay enabled. Finally when the data memory (with data overlay, denoted as $data_min$) plus an overlay buffer is larger than the SPM, the memory enters IF state, indicating that this actor cannot be scheduled on the p.

6.3.4.3 Calculation of processor workload

The calculation of the workload after scheduling actor v on p is given by

$$\Delta(p) \leftarrow d(v) + Max\{\tau_c(v), \Delta(p) + \tau_o(v)\} \tag{6.3}$$

In Equation(6.3), $d(v)$ is the computation delay of actor v. $\tau_c(v)$ models the earliest start time of v due to data dependencies. $\Delta(p)$ indicates the workload of p before scheduling v on it. $\tau_o(v)$ indicates the code overlay overhead of scheduling v on p. $\Delta(p) + \tau_o(v)$ models the earliest start time of v due to limited PEs and code overlay. The calculation of $\tau_c(v)$ and $\tau_o(v)$ are discussed in the following,

6.3.4.4 Calculation of $\tau_c(v)$

There could be two categories of data dependencies, namely intra-pipeline dependencies and inter-pipeline dependencies in our schedule. For an intra-pipeline dependency, the edge that connects the producer and the consumer has no delay on it. Therefore the consumer can only start after its producer finishes execution and transfers its data to the consumer side. For an inter-pipeline dependency, the producer and consumer have at least one delay between them. The consumer can execute with the producer simultaneously in a pipelined manner. Given the above discussion, the calculation of $\tau_{intra}(v)$ is

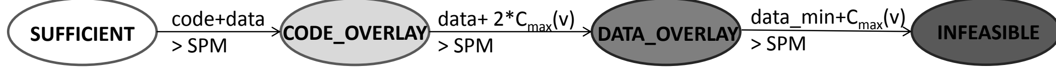


Figure 6.2: Processor memory state transitions.

given by,

$$\tau_{intra}(v) \leftarrow \max_{e:u\rightarrow v, v\in p, u\notin p, w_r(e)=0} \Delta(u) + T_c(C(e)) \tag{6.4}$$

In Equation (6.4), $v \in p, u \notin p$, and $w_r(e) = 0$ indicate the condition for edge e to have an intra-pipeline communication. $\Delta(u)$ indicates the completion time of producer u. Since we schedule actors following the scheduling order S, by the time we schedule v on p, $\Delta(u)$ is known to us. $T_c(C(e))$ computes the cost of transferring data from the producer to the consumer. Function $T_c(x)$ is a hardware feature and is defined in Section 6.1. The calculation of $\tau_{inter}(v)$ is given by,

$$\tau_{inter}(v) \leftarrow \begin{cases} T_c(C(e)), \text{if } get_db(r(v)) = 0, \\ \max\{0, T_c(C(e)) - d(u)\}, \text{otherwise.} \end{cases} \tag{6.5}$$

where $e : u \rightarrow v, v \in p, u \notin p, w_r(e) \geq 1$.

In Equation (6.5), the condition for an edge to have inter-pipeline communication overhead is given by $v \in p, u \notin p$, and $w_r(e) \geq 1$. When double buffering is disabled, the communication overhead equals the DMA transfer cost. Otherwise, DMA transfer is overlapped with actor computation and the effective cost is given by $\max\{0, T_c(C(e)) - d(u)\}$.

6.3.4.5 Calculation of $\tau_o(v)$

When the on-chip SPM of p is not able to accommodate all its code and data, code overlay overhead is encountered. The calculation of $\tau_o(v)$ is given by,

$$\tau_o(v) \leftarrow \begin{cases} 0, & \text{if } get_state(p) = \text{SF}, \\ \max\{0, T_c(C(v)) - d(u)\}, & \text{if } get_state(p) = \text{CO}, \\ T_c(C(v)), & \text{if } get_state(p) = \text{DO}, \\ +\infty, & \text{if } get_state(p) = \text{IF}. \end{cases} \tag{6.6}$$

In Equation (6.6), $T_c(C(v))$ captures the code overlay overhead without code pre-fetching and $\max\{0, d(u) - T_c(C(v))\}$ captures the code overlay overhead with code pre-fetching, where u is the actor scheduled immediately before v on the same PE.

6.3.5 Algorithm Complexity

Without unrolling, AlgorithmDeltaCD runs in $O(|E| + |V|)$. AlgorithmFDL wraps AlgorithmDeltaCD within a loop of $|V|$. Therefore AlgorithmFDL runs in $O(|V|(|E| + |V|))$. The binary search adds another complexity of $O(log_2 U)$ ($U = \sum_{v \in V} d(v)$). As a result, URSEM without unrolling runs in $O(|V|(|E| + |V|)log_2 U)$. Since the unroll factor is bounded by $|P|$ in our technique, the overall algorithm complexity is given by $O(|V||P|^2(|E|+|V|)(log_2 U + log_2 |P|))$.

6.4 Experimental Results

We adopted StreamIt language as our input specification. URSEM was implemented as an optimization pass in the StreamIt compiler 2.1.1. The hardware

platform we experimented with is a PlayStation3 system running Fedora 9 at 3.2 GHz. In this platform, 1 PPE and 6 SPEs are available to the programmer. Table 1.1 presents the benchmark details.

6.4.0.1 Overall performance comparison

We first compared the performance of our URSEM with two existing approaches, namely CSMP [20] and RTEM [15]. CSMP has no control over the number of software pipelines being generated. For a fair comparison, we set N_{user} to be infinity for RTEM and URSEM. Figure 6.3 presents their performance results. The x-axis and y-axis provide the benchmark names and their steady-state execution time normalized to the lower bound. The lower bound is given by $L = \max\{\max_{v \in V_s} d(v)/f, \sum_{v \in V} d(v)/|p|\}$, where V_s is the set of stateful actors. Our URSEM outperforms RTEM due to the fact that we perform unrolling and retiming simultaneously. A better load balancing is expected as the unroll factor increases. Compared with RTEM, we also implemented code prefetching and data overlay that further improves the overall performance. Our URSEM also performs better than CSMP in most cases. This is due to the fact that in CSMP, as the fusion operation proceeds, the granularity that the algorithm can operate on becomes larger and larger. Whereas, our URSEM always works on the granularity of a single actor. Overall our URSEM outperforms CSMP by 21% and RTEM by 6%. The algorithm run time of our URSEM, based on the benchmark size, is hundreds of seconds. Whereas CSMP and RTEM finishes in less than ten seconds. The increased algorithm run time is due to the Unrolling operation. Nevertheless, a user can provide an upper bound on the unroll factor for a shorter algorithm run time, or terminate the unrolling as soon as an acceptable solution is achieved.

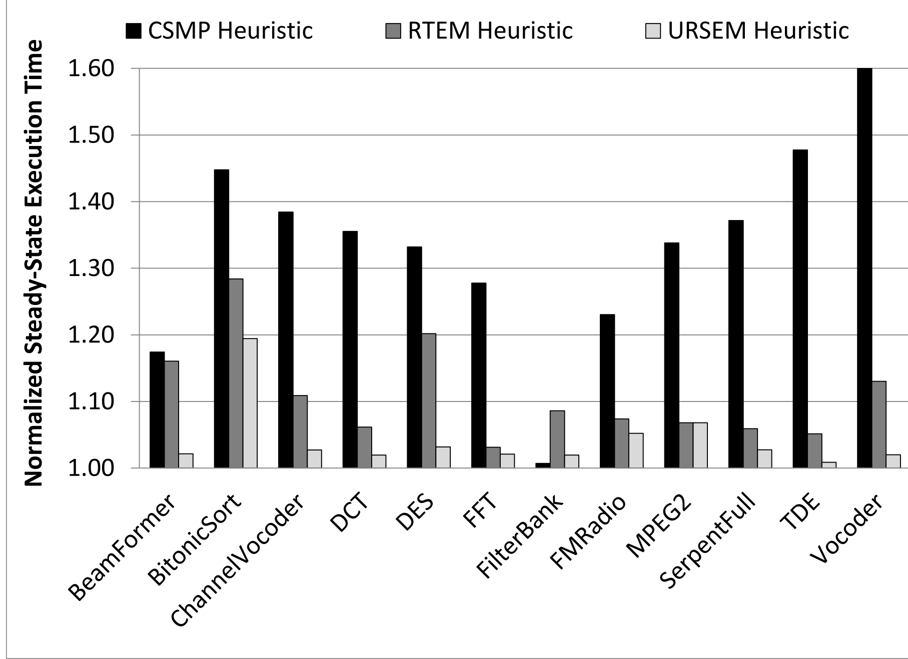


Figure 6.3: Overall performance comparison.

6.4.0.2 Impact of optimizations

We examine the impact of each optimization in URSEM in Figure 6.4. N_{user}, $|P|$ and SPM size are set to infinity, 28, and 4K respectively. We applied Retiming, Unrolling, Double Buffering, Code Overlay, Code Pre-fetching, and Data Overlay incrementally. We didn't show the results for FilterBank, FM-Radio, and Serpentfull because they reduce to the mapping of scheduling every actor on PPE. As observed from Figure 6.4, Unrolling delivers the most significant performance gain due to improved parallelism in an unrolled graph. Double buffering is another optimization that has a significant impact. It takes effect when the code and data memory is still tolerable compared to the

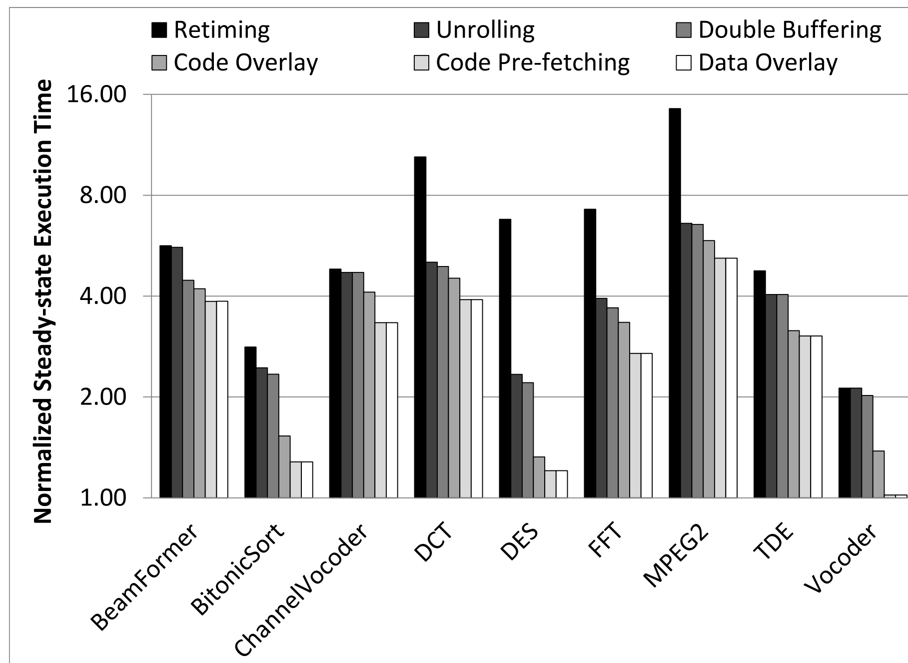


Figure 6.4: Impact of optimizations.

SPM size. Code Overlay, Code Prefetching, and Data Overlay further reduce the steady-state execution time whenever they can be applied. Overall, the performance improvement by applying all six optimizations is over 47%.

6.4.0.3 Performance scaling with PEs

In this section, we examine the scalability of URSEM with different number of PEs. N_{user} and SPM are set to infinity and 256K respectively so that they do not become the limiting factors. The number of processors are set to 7, 14, 21, and 28 respectively. The experimental results are shown in Figure 6.5. The y axis in the figure presents the normalized steady-state execution time scaled to 7 PEs. Overall our approach scales with the number of processors as

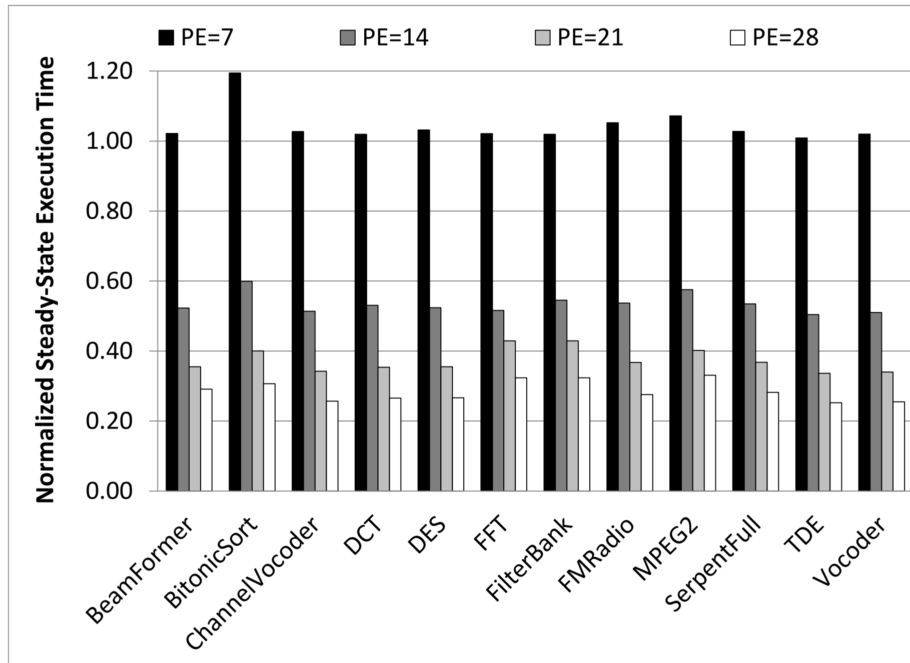


Figure 6.5: Performance scaling with PEs.

observed from Figure 6.5. This property results from the fact that we perform iterative unrolling in our algorithm. When the number of processors increases, the algorithm will search for a larger unroll factor. As a result, the unrolled graph that is to be scheduled with retiming actually scales with the number of PEs. Overall the performance improvement is around 70% when we increased the number of PEs from 7 to 28. The results validate the scalability of our approach.

6.4.1 Performance scaling with Delays

Figure 6.6 examines the scalability of our URSEM approach with various user specified software pipeline stages. N_{user} are set to be 7, 14, 21, and 28 in

this setup. The y-axis in the figure shows the steady-state execution time of each benchmark normalized to its performance achieved with $N_{user} = 7$. For most of the benchmarks, we observe a significant performance improvement as we increase the pipeline stages from 7 to 28. This is because with 7 pipeline stages, there is no double buffering due to the lack of delays. Further, when we unroll a graph, the retiming groups that can be generated could decrease due to inter-iteration data dependencies. As observed from Figure 6.6, we achieved a performance gain of 6% on average as we increased N_{user} from 7 to 28. For benchmarks BeamFormer, BitonicSort, and DES we observe a huge latency reduction from 7 to 14 pipeline stages. The major contributing factor for the performance gain of BitonicSort is double buffering that could be introduced when we have more delays. For BeamFormer and DES, the reduction results from the reduction of intra-pipeline communication.

6.4.1.1 Performance scaling with SPMs

In this experimental setup, we examine the performance of our URSEM algorithm under tight SPM constraints. The size of each SPM is set to be 2K, 4K, 16K, and 256K respectively. N_{user} and $|P|$ are set to infinity and 7 respectively. Figure 6.7 provides the experimental results under this setup. As observed from Figure 6.7, our URSEM always generates a valid solution. When the SPM size is extremely limited, many solutions reduce to mapping all actors to PPE. From Figure 6.7, when the SPM size is set to 2K, 6 out of 12 benchmarks map everything to PPE. When the memory increases from 2K to 16K, the steady-state execution time of each benchmark drops down dramatically. This behavior suggests that the on-chip SPM is very precious when the code and data memory are comparatively large. The results validate

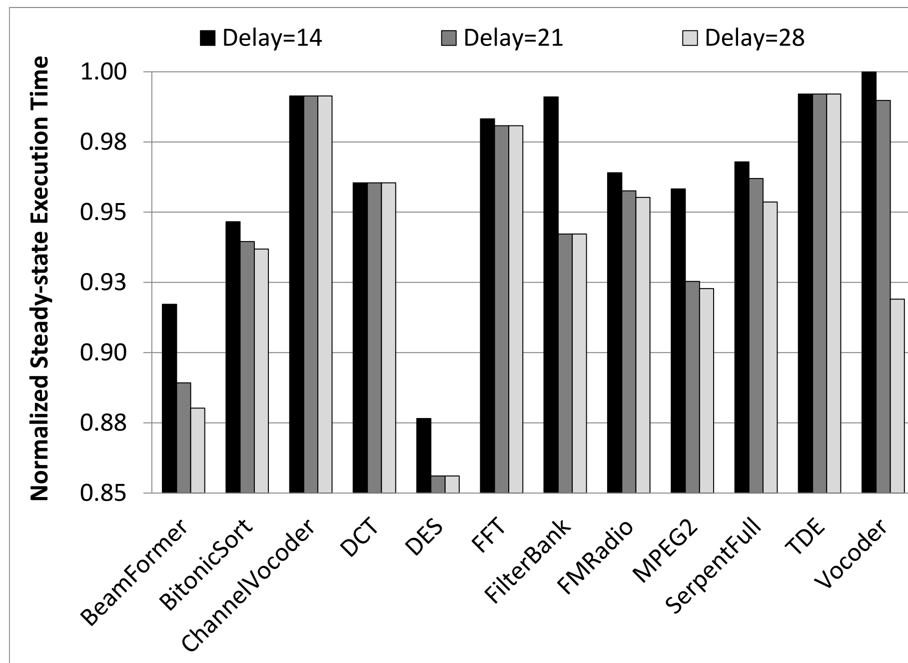


Figure 6.6: Performance scaling with Delays.

the rationale for introducing code overlay and data overlay in our technique.

6.5 Summary

In this paper, we propose an unrolling and retiming approach for scheduling stream applications onto embedded multicore processors. In our technique, a user specified number of software pipeline stages can be imposed. Compared to the existing approaches, our URSEM algorithm efficiently unrolls and schedules a stream application with loop structures. Our URSEM scales well over a wide range of PEs, delays, and SPMs. Further, our heuristic performs code pre-fetching and data overlay under tight SPM constraints, thus is able to handle extreme cases with tolerable performance results. Our future work will

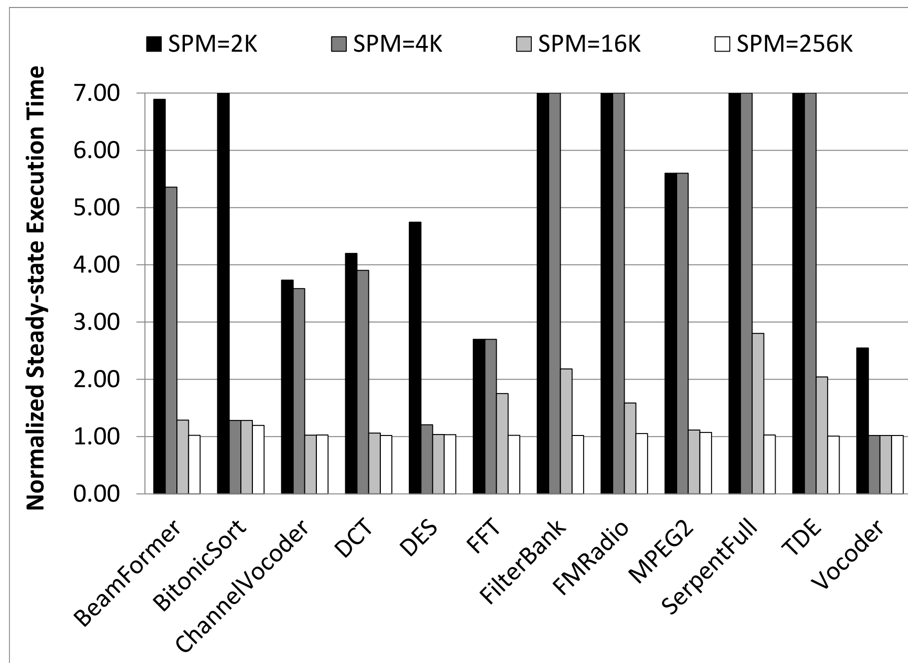


Figure 6.7: Performance scaling with SPMs.

address stream applications with dynamic behavior and execution time.

Chapter 7

CONCLUSION AND FUTURE WORK

In this dissertation, we present optimization techniques that automatically compile stream programs onto SPM enhanced architectures. As an initial effort, we manually implement an ATR algorithm on the IBM Cell BE. Eight optimizations that exploit both the specific algorithm constructs of the ATR algorithm and the architectural features of the Cell BE are implemented. The manual implementation provides us with both the design trade-offs of programming on embedded multicore processors and the optimizations that are applicable when programming stream applications on embedded multicore. Then we provide a three-stage ILP and a heuristic for scheduling stream programs on an SPM based embedded core with code overlay. The three-stage ILP approach extensively explores the design alternatives with different schedules, code/data partitions, and actor to region/segment assignments. Although the ILP approach is able to explore various design trade-offs and generate high quality solutions, it takes a very long time to run for large inputs. For a faster algorithm time, a heuristic algorithm is later provided. The heuristic deliberately evolves the schedule from a minimum buffer schedule to a minimum actor switching schedule. It generates a minimum code overlay schedule that balances the data buffer usage and actor switches.

In the next phase of our work, we propose CSMP ILP and heuristic optimizations that schedule stream programs on SPM based embedded multicore processors. CSMP ILP utilizes fusion and fission operators to combine actors into batches, and then batches to PEs. Communication overheads and code overlays are modeled under the given SPM capacity and DMA transfer de-

lays. Experimental results show that CSMP ILP is able to efficiently trade-off between computation and communication and provide close to optimal solutions. To amortize the algorithm run time of CSMP ILP, CSMP heuristic is provided. CSMP heuristic is able to generate solutions comparable to CSMP ILP in a matter of seconds. CSMP ILP and heuristic approaches do not optimize for cycles. Neither do they have control over the number of software pipeline stages being generated. We next present RTEM heuristic that schedules stream programs onto SPM based multicore processors through retiming. Trade-offs between double buffering and code overlay are explored intensively in this approach. More importantly, the retiming approach inherently handles cycles and also can accept an upper bound on the resulting number of software pipeline stages. Although RTEM heuristic generates high quality solutions for the Cell processor with 8 SPEs, it has the limitation of relying on the existing parallelism in a stream program for parallelization. Consequently, it may not scale with a large number of processing engines. As our last optimization, we extend RTEM heuristic with unrolling and propose URSEM technique. URSEM performs unrolling and retiming simultaneously, thus is able to achieve much better scalability compared with RTEM heuristic. To address the increased code and data size due to unrolling, we also incorporate code pre-fetching and data overlay. Experimental results show that URSEM approach is able to generate high quality solutions over a wide range of PEs, pipeline stages, and SPMs.

The dissertation can be extended from two directions, namely support of industry oriented, open royalty-free standard general purpose parallel programming languages, such as Open Computing Language (OpenCL) [58] and support of cache/SPM mixed embedded multicore architectures such as TI

TMSC320C6472 [73]. In the remainder of this chapter, we adopt OpenCL and TMSC320C6472 as representatives to discuss the challenges that must be addressed.

7.1 Future Work on OpenCL

OpenCL is a framework for writing programs that execute across heterogeneous platforms. The target platform typically consists of central processing units (CPUs), graphics processing units (GPUs), and several other DSPs. An OpenCL program is constructed with kernels that execute on OpenCL devices and application programming interfaces (APIs) that define the control and code/data transfers. There have been some initial efforts that investigate automatic compilation of OpenCL programs onto FPGAs [39]. To apply our optimizations, we can treat a kernel in an OpenCL program as an actor. Consequently, data communications among different kernels correspond to edges. The challenges of incorporating OpenCL in our optimizations lie in the implicit data communications/dependencies, the vastly different computing devices, and the many levels of memory hierarchies that OpenCL supports. In stream programs, data communications are explicitly specified with pop, peek, and pop operators. In an OpenCL program, input and output of a kernel are passed in/out through kernel arguments (pointers). Data dependencies are implicitly specified. Further, since there are many different computing devices supported, a kernel will have different run time/code size for each device. Our optimizations can be adjusted to take these variations into account for actor to processing engine mapping. Compared to SPM based multicore processors where we have on-chip SPMs and off-chip main memory, the memory hier-

archies of the heterogeneous platforms that OpenCL supports are also more complicated. Our optimizations can be extended to handle these platforms by appropriately adjusting costs of code and data placements/transfers, and memory access delays.

7.2 Future Work on TI Multicore

TI TMSC320C6472 is a representative embedded multicore architecture from TI with heterogeneous cache/SPM mixed memory hierarchy. TMSC320C6472 has six processing engines/DSP subsystems that are based on C64x DSPs. Each processing engine has 32 KB L1 instruction memory, 32 KB L1 data memory, and 608 KB L2 memory that can be configured as either SPM or cache[1]. There is also a 768 KB RAM that is shared by all six processing engines. To simplify the extension of our optimizations to cache/SPM mixed architecture, we first impose a few restrictions on the architecture itself. First, let us assume instructions that mapped to an SPM operates on data that also resides in the same SPM. Second, an actor is the smallest granularity that our optimizations operate on, meaning all instructions within one actor are either all mapped to an SPM or they are all mapped to instruction cache. Third, instructions mapped to an instruction cache or SPM can fetch data from the same processing engine either from its data cache, or from the local SPM. Last, caches are assumed to be coherent across all processing engines. We can extend our optimizations to support cache/SPM mixed architectures by addressing the following challenges. First, separate actors mapped to an SPM from actors mapped to instruction caches. Intuitively, an SPM should

[1]When configured as caches, L1 instruction memory is directed mapped, L1 data memory is 2-way set associative, and L2 memory is 4-way set associative.

be used for actors that exhibit stable and predictable data access patterns. Since the on-chip SPMs are typically very small, they should be used very cautiously. Then separate data mapped to an SPM from data mapped to data cache. Data mapped to data cache are subject to cache misses and data mapped to an SPM may require explicit DMA transfers. Since code and data mapped to an SPM shares the same physical memory, over utilization of an SPM could result in high code/data overheads. The calculation of computation/communication costs in our optimizations can be modified to adjust to various actor to cache/SPM mappings, including i) actor code and data both reside in an SPM, ii) actor code and data both mapped to caches, iii) actor code resides in an SPM with data mapped to data cache, and iv) actor code mapped to instruction cache with actor data resides in an SPM. Our optimizations can be enhanced to accept cache/SPM mixed architectures following the guideline of extracting code and data from caches to SPMs to improve performance.

REFERENCES

[1] Federico Angiolini, Luca Benini, and Alberto Caprara. Polynomial-time algorithm for on-chip scratchpad memory partitioning. In Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems, CASES '03, pages 318–326, New York, NY, USA, 2003. ACM.

[2] Federico Angiolini, Francesco Menichelli, Alberto Ferrero, Luca Benini, and Mauro Olivieri. A post-compiler approach to scratchpad mapping of code. In Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems, CASES '04, pages 259–267, New York, NY, USA, 2004. ACM.

[3] Oren Avissar, Rajeev Barua, and Dave Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. ACM Trans. Embed. Comput. Syst., 1:6–26, November 2002.

[4] Michael A. Baker, Pravin Dalale, Karam S. Chatha, and Sarma B.K. Vrudhula. A scalable parallel h.264 decoder on the cell broadband engine architecture. In Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis, CODES+ISSS '09, pages 353–362, New York, NY, USA, 2009. ACM.

[5] Michael A. Baker, Amrit Panda, Nikhil Ghadge, Aniruddha Kadne, and Karam S. Chatha. A performance model and code overlay generator for scratchpad enhanced embedded processors. In Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, CODES/ISSS '10, pages 287–296, New York, NY, USA, 2010. ACM.

[6] S Bandyopadhyay. Automated memory allocation of actor code and data buffer in heterochronous dataflow models to scratchpad memory. Technical Report No. UCB/EECS-2006-105, August 2006.

[7] Shamik Bandyopadhyay, Thomas Huining Feng, Hiren D. Patel, and Edward A. Lee. A scratchpad memory allocation scheme for dataflow models, 2008.

[8] Shuvra S. Battacharyya, Edward A. Lee, and Praveen K. Murthy. Software Synthesis from Dataflow Graphs. Kluwer Academic Publishers, Norwell, MA, USA, 1996.

[9] B. Bhanu. Automatic target recognition: State of the art survey. Aerospace and Electronic Systems, IEEE Transactions on, AES-22(4):364 –379, july 1986.

[10] B. Bhanu and T.L. Jones. Image understanding research for automatic target recognition. Aerospace and Electronic Systems Magazine, IEEE, 8(10):15 –23, oct 1993.

[11] S.S. Bhattacharyya, J.T. Buck, S. Ha, and E.A. Lee. A scheduling framework for minimizing memory requirements of multirate dsp systems represented as dataflow graphs. In VLSI Signal Processing, VI, 1993., [Workshop on], pages 188 –196, oct 1993.

[12] Filip Blagojevic, Dimitris S. Nikolopoulos, Alexandros Stamatakis, and Christos D. Antonopoulos. Dynamic multigrain parallelization on the cell broadband engine. In Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming, PPoPP '07, pages 90–100, New York, NY, USA, 2007. ACM.

[13] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. ACM Trans. Graph., 23(3):777–786, August 2004.

[14] Weijia Che and K. Chatha. Compilation of stream programs onto scratchpad memory based embedded multicore processors through retiming. In Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE, pages 122 –127, june 2011.

[15] Weijia Che and K. Chatha. Compilation of stream programs onto scratchpad memory based embedded multicore processors through retiming. In Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE, pages 122 –127, june 2011.

[16] Weijia Che and K. Chatha. Unrolling and retiming of stream applications onto embedded multicore processors. In Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE, june 2012.

[17] Weijia Che and Karam S. Chatha. Design of an automatic target recognition algorithm on the ibm cell broadband engine. In Application-specific Systems Architectures and Processors (ASAP), 2010 21st IEEE International Conference on, pages 21 –28, july 2010.

[18] Weijia Che and Karam S. Chatha. Scheduling of stream programs onto spm enhanced processors with code overlay. In Embedded Systems for Real-Time Multimedia (ESTIMedia), 2011 9th IEEE Symposium on, pages 9 –18, oct. 2011.

[19] Weijia Che and K.S. Chatha. Scheduling of synchronous data flow models on scratchpad memory based embedded processors. In Computer-Aided Design (ICCAD), 2010 IEEE/ACM International Conference on, pages 205 –212, nov. 2010.

[20] Weijia Che, Amrit Panda, and Karam S. Chatha. Compilation of stream programs for multicore processors that incorporate scratchpad memories. In Proceedings of the Conference on Design, Automation and Test in Europe, DATE '10, pages 1118–1123, 3001 Leuven, Belgium, Belgium, 2010. European Design and Automation Association.

[21] Michael K. Chen, Xiao Feng Li, Ruiqi Lian, Jason H. Lin, Lixia Liu, Tao Liu, and Roy Ju. Shangri-la: achieving high performance from compiled network applications while enabling ease of programming. SIGPLAN Not., 40:224–236, June 2005.

[22] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata. Cell broadband engine architecture and its first implementation–a performance view. IBM Journal of Research and Development, 51(5):559 –572, sept. 2007.

[23] Kang-Ngee Chia, Hea Joung Kim, S. Lansing, W.H. Mangione-Smith, and J. Villasensor. High-performance automatic target recognition through data-specific vlsi. Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, 6(3):364 –371, sept. 1998.

[24] Yoonseo Choi, Yuan Lin, Nathan Chong, Scott Mahlke, and Trevor Mudge. Stream compilation for real-time embedded multicore systems. In Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '09, pages 210–220, Washington, DC, USA, 2009. IEEE Computer Society.

[25] Alain Darte and Guillaume Huard. Loop shifting for loop compaction. International Journal of Parallel Programming, 28:499–534, 2000. 10.1023/A:1007506711786.

[26] Bernhard Egger, Chihun Kim, Choonki Jang, Yoonsung Nam, Jaejin Lee, and Sang Lyul Min. A dynamic code placement technique for scratchpad memory using postpass optimization. In Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems, CASES '06, pages 223–233, New York, NY, USA, 2006. ACM.

[27] A. E. Eichenberger, J. K. O'Brien, K. M. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. K. Gschwind, R. Archambault, Y. Gao, and R. Koo. Using advanced compiler technology to exploit the performance of the cell broadband engine^{TM}; architecture. IBM Systems Journal, 45(1):59 –84, 2006.

[28] A.E. Eichenberger, K. O'Brien, Peng Wu, Tong Chen, P.H. Oden, D.A. Prener, J.C. Shepherd, Byoungro So, Z. Sura, A. Wang, Tao Zhang, Peng Zhao, and M. Gschwind. Optimizing compiler for the cell processor. In Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on, pages 161 – 172, sept. 2005.

[29] Johan Eker and JAorn W. Janneck. Cal language report. 2003.

[30] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: programming the memory hierarchy. In Proceedings of the 2006 ACM/IEEE conference on Supercomputing, SC '06, New York, NY, USA, 2006. ACM.

[31] Michael I. Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. SIGOPS Oper. Syst. Rev., 40:151–162, October 2006.

[32] Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. A stream compiler for communication-exposed architectures. SIGARCH Comput. Archit. News, 30:291–303, October 2002.

[33] M. Gschwind, H.P. Hofstee, B. Flachs, M. Hopkin, Y. Watanabe, and T. Yamazaki. Synergistic processing in cell's multicore architecture. Micro, IEEE, 26(2):10 –24, march-april 2006.

[34] A.H. Hormati, Yoonseo Choi, and M. Kudlur et al. Flextream: Adaptive compilation of streaming applications for heterogeneous architectures. In PACT '09, pages 214 –223, sept. 2009.

[35] Amir H. Hormati, Mehrzad Samadi, Mark Woh, Trevor Mudge, and Scott Mahlke. Sponge: portable stream programming on graphics engines. In Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems, ASPLOS '11, pages 381–392, New York, NY, USA, 2011. ACM.

[36] IBM. Cell broadband engine resource center. http://www.ibm.com/developerworks/power/cell/pkgdownloads.html.

[37] Andhi Janapsatya, Aleksandar Ignjatović, and Sri Parameswaran. A novel instruction scratchpad memory optimization method based on concomitance metric. In Proceedings of the 2006 Asia and South Pacific Design Automation Conference, ASP-DAC '06, pages 612–617, Piscataway, NJ, USA, 2006. IEEE Press.

[38] Axel Jantsch. Modeling Embedded Systems and SoC's: Concurrency and Time in Models of Computation (Systems on Silicon). Morgan Kaufmann, 1 edition, June 2003.

[39] P.O. JaÌĹ andaÌĹ andskelaÌĹ andinen, C.S. de La Lama, P. Huerta, and J.H. Takala. Opencl-based design methodology for application-specific processors. In Embedded Computer Systems (SAMOS), 2010 International Conference on, pages 223 –230, july 2010.

[40] Erik J. Johnson and Aaron R. Kunze. Ixp2400-2800 Programming: The Complete Microengine Coding Guide. Intel Press, 2003.

[41] Seung chul Jung, Aviral Shrivastava, and Ke Bai. Dynamic code mapping for limited local memory systems. In Application-specific Systems Architectures and Processors (ASAP), 2010 21st IEEE International Conference on, pages 13 –20, 2010.

[42] N. Kato, K. Takeuchi, S. Maeda, M. Shimbayashi, R. Sakai, H. Nozue, and J. Amemiya. Digital media applications on a cell software platform. In Consumer Electronics, 2006. ICCE '06. 2006 Digest of Technical Papers. International Conference on, pages 347 – 348, jan. 2006.

[43] Emmett Kilgariff and Randima Fernando. The geforce 6 series gpu architecture. In ACM SIGGRAPH 2005 Courses, SIGGRAPH '05, New York, NY, USA, 2005. ACM.

[44] M. Kistler, M. Perrone, and F. Petrini. Cell multiprocessor communication network: Built for speed. Micro, IEEE, 26(3):10 –23, may-june 2006.

[45] M. Kistler, M. Perrone, and F. Petrini. Cell multiprocessor communication network: Built for speed. Micro, IEEE, 26(3):10 –23, may-june 2006.

[46] Timothy J. Knight, Ji Young Park, Manman Ren, Mike Houston, Mattan Erez, Kayvon Fatahalian, Alex Aiken, William J. Dally, and Pat Hanrahan. Compilation for explicitly managed memory hierarchies. In Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming, PPoPP '07, pages 226–236, New York, NY, USA, 2007. ACM.

[47] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: a 32-way multithreaded sparc processor. Micro, IEEE, 25(2):21 – 29, march-april 2005.

[48] Manjunath Kudlur and Scott Mahlke. Orchestrating the execution of stream programs on multicore platforms. SIGPLAN Not., 43:114–124, June 2008.

[49] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. Proceedings of the IEEE, 75(9):1235 – 1245, sept. 1987.

[50] Charles Leiserson and James Saxe. Retiming synchronous circuitry. Algorithmica, 6:5–35, 1991.

[51] Chao Liang-Fang. Scheduling And Behavioral Transformations For Parallel Systems. PhD thesis, Princeton University, 1993.

[52] Shih-wei Liao, Zhaohui Du, Gansha Wu, and Guei-Yuan Lueh. Data and computation transformations for brook streaming applications on multiprocessors. In Proceedings of the International Symposium on Code Generation and Optimization, CGO '06, pages 196–207, Washington, DC, USA, 2006. IEEE Computer Society.

[53] S. Maeda, S. Asano, T. Shimada, K. Awazu, and H. Tago. A real-time software platform for the cell processor. Micro, IEEE, 25(5):20 – 29, sept.-oct. 2005.

[54] Michael D. McCool, Zheng Qin, and Tiberiu S. Popa. Shader metaprogramming. In Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, HWWS '02, pages 57–68, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[55] Giovanni De Micheli. Synthesis and Optimization of Digital Circuits. McGraw-Hill Higher Education, 1st edition, 1994.

[56] MIT. Streamit benchmarks. http://groups.csail.mit.edu/cag/streamit/shtml/benchmarks.shtml.

[57] MIT. Streamit compiler source code. http://groups.csail.mit.edu/cag/streamit/ restricted/files.shtml.

[58] A. Munshi. The OpenCL Specification, version 1.0. Khronos OpenCL Working Group, 2009.

[59] Nghi Nguyen, Angel Dominguez, and Rajeev Barua. Memory allocation for embedded systems with a compile-time-unknown scratch-pad size. In Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems, CASES '05, pages 115–125, New York, NY, USA, 2005. ACM.

[60] NVIDIA. Compute Unified Device Architecture Programming Guide. NVIDIA: Santa Clara, CA, 2007.

[61] Chris Ostler, Karam S. Chatha, Vijay Ramamurthi, and Krishnan Srinivasan. Ilp and heuristic techniques for system-level design on network processor architectures. ACM Trans. Des. Autom. Electron. Syst., 12, September 2007.

[62] Amit Pabalkar, Aviral Shrivastava, Arun Kannan, and Jongeun Lee. Sdrm: simultaneous determination of regions and function-to-region mapping for scratchpad memories. In Proceedings of the 15th international conference on High performance computing, HiPC'08, pages 569–582, Berlin, Heidelberg, 2008. Springer-Verlag.

[63] F. Petrini, G. Fossum, J. Fernandez, A.L. Varbanescu, N. Kistler, and M. Perrone. Multicore surprises: Lessons learned from optimizing sweep3d on the cell broadband engine. In Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International, pages 1 –10, march 2007.

[64] D.C. Pham, T. Aipperspach, D. Boerstler, M. Bolliger, R. Chaudhry, D. Cox, P. Harvey, P.M. Harvey, H.P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Pham, J. Pille, S. Posluszny, M. Riley, D.L. Stasiak, M. Suzuoki, O. Takahashi, J. Warnock, S. Weitzel, D. Wendel, and K. Yazawa. Overview of the architecture, circuit design, and physical implementation of a first-generation cell processor. Solid-State Circuits, IEEE Journal of, 41(1):179 – 196, jan. 2006.

[65] J.L. Pino, S.S. Bhattacharyya, and E.A. Lee. A hierarchical multiprocessor scheduling system for dsp applications. In Signals, Systems and Computers, 1995. 1995 Conference Record of the Twenty-Ninth Asilomar Conference on, volume 1, pages 122 –126 vol.1, oct-1 nov 1995.

[66] J.L. Pino and E.A. Lee. Hierarchical static scheduling of dataflow graphs onto multiple processors. In Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on, volume 4, pages 2643 –2646 vol.4, may 1995.

[67] M. Rencher and B.L. Hutchings. Automated target recognition on splash 2. In FPGAs for Custom Computing Machines, 1997. Proceedings., The 5th Annual IEEE Symposium on, pages 192 –200, apr 1997.

[68] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. ACM Trans. Graph., 27:18:1–18:15, August 2008.

[69] S. Steinke, L. Wehmeyer, Bo-Sik Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. In Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings, pages 409–415, 2002.

[70] John Stratton, Sam Stone, and Wen-mei Hwu. Mcuda: An efficient implementation of cuda kernels for multi-core cpus. In JosÃľ Amaral, editor, Languages and Compilers for Parallel Computing, volume 5335 of Lecture Notes in Computer Science, pages 16–30. Springer Berlin / Heidelberg, 2008.

[71] William Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit: A language for streaming applications. In R. Horspool, editor, Compiler Construction, volume 2304 of Lecture Notes in Computer Science, pages 49–84. Springer Berlin / Heidelberg, 2002.

[72] TILERA. TILE64 Processor: http://www.tilera.com/products/processors /TILE64, accessed Dec., 2011.

[73] Loc Truong. White paper: Low power consumption and a competitive price tag make the six-core tms320c6472 ideal for high performance applications. High-Performance Multicore, Processing Business, oct. 2009.

[74] M. Verma and P. Marwedel. Overlay techniques for scratchpad memories in low power embedded processors. Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, 14(8):802 –815, 2006.

[75] Manish Verma, Lars Wehmeyer, and Peter Marwedel. Dynamic overlay of scratchpad memory for energy minimization. In Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, CODES+ISSS '04, pages 104–109, New York, NY, USA, 2004. ACM.

[76] Scott Wasson. Ageia's physx physics processing unit. The tech report, PC hardware explored, Last accessed July 2008.

[77] Shih wei Liao, Zhaohui Du, Gansha Wu, and Guei-Yuan Lueh. Data and computation transformations for brook streaming applications on multiprocessors. In Code Generation and Optimization, 2006. CGO 2006. International Symposium on, page 12 pp., march 2006.

[78] Q. Zhao and J.C. Principe. Support vector machines for sar automatic target recognition. Aerospace and Electronic Systems, IEEE Transactions on, 37(2):643 –654, apr 2001.

APPENDIX A

A Compiler Backend for the IBM Cell Broadband Engine

In this appendix, we provide a compiler framework that automatically generates multi-threaded IBM Cell Broadband Engine (BE) code. The compiler framework takes StreamIt [71] programs as input and automatically generate executables that runs concurrently on the IBM Cell BE. In the following, the file structure of our code base, the execution pattern of the generated code, the environment setup, the available commands, and interpretation of results are discussed in details.

File Structure

The compiler framework is built upon the StreamIt compiler [57] that is developed by MIT. The MIT StreamIt project can be found in the homepage at `http://groups.csail.mit.edu/cag/streamit/`. The StreamIt compiler can take StreamIt program as input and generate simple C code. We instrument the existing StreamIt compiler in three places to enable automatic generation of multi-threaded IBM Cell BE Code.

We first insert an optimization pass that operations on the intermediate format of a StreamIt program and generates mapping and scheduling of the given program on the IBM Cell BE. This code base can be found under the directory of "STREAMIT_HOME/src/at/dms/kjc/cell/heuristic". The files under this directory include

- CellPPU.java: The CellPPU class stores the information of a PPU architecture and its helper functions. The basic information includes the memory size of PPU, the batch and filters that are mapped to the PPU.
- CellSPU.java: The CellSPU class stores the information of a SPU ar-

chitecture and its helper functions. The basic information includes the index of the current SPU, its memory size, the batch and filters that are mapped to the current SPU. To facilitate code generation, there are also some other information that are produced, for example the incoming edges, outgoing edges, intra edges of the current SPU, and information that helps implementation of split-join structure and batch fission.

- HeuFilter.java: The HeuFilter class stores the information of a filter and its helper functions. The basic information includes the filter ID, the number of peek tokens, the number of pop tokens, the number of push tokens, the number of steady-state executions of the current filter, work estimate of the current filter, code size estimate of the current filter, whether the current filter is stateful. Some other information that are stored include the parents and children of the current filter, it incoming and outgoing edges, and more information that helps implementation of code generation.

- HeuEdge.java: The HeuEdge class stores the information of an edge between two filters and its helper functions. The basic information includes the edge ID, its producer and consumer, the edge weight and other information that helps implementation of code generation.

- HeuGraph.java: The HeuGraph class stores our intermediate format of a StreamIt program. The structure is a graph as indicated by the class name. A node in the graph represents a filter in the StreamIt program. Correspondingly, an edge represents data communications between two filters. The HeuGraph also implements helpr functions that operates on it.

- HeuMain.java: The HeuMain class is the main routine of our optimization. It implements the algorithm that is published in [20].

Then we insert code base that generates the IBM Cell BE code, including header files, code for each filter, the main thread code that runs on PPE, and makefile that compiles and links everything together. The code base can be found under the directory "STREAMIT_HOME/src/at/dms/kjc/cell". We describe this code base in more detail below.

- EmitCellCode.java: The EmitCellCode class takes our optimization output and generate IBM Cell BE code accordingly. It implements helper functoins that generates the makefile, ppu header, ppu code, spu header, spu code.

- CellBackend.java: The CellBackend class implements the main routine that calls our optimiation pass and code generation pass. It is invoked when the target architecture is specified as IBM Cell BE.

Finally, we also implement a static library that iteratives parses the code generated by our optimization and backend, and generates output information. This code base can be found under the directory of "STREAMIT_HOME/library/cell". The details of the code base are discussed in the following.

- include: This direcory defines all the header files, common micro definitions that is used through out the library.

- lib: This directory stores the compiled library as "spulib_spu.a".

- src: This directory contains the code that implements the main routine of the static library.

Execution Pattern

During the program execution, the PPU of the IBM Cell BE first initializes all the control blocks, filter constructs, and edge constructs. A control block stores the infomation of the SPU index, the scaling factor of the current graph, the number of total iterations to run, the number of filers assgined to the current SPU and several other variables that help the implementation of peek and fission. A filter struct stores all the information that is needed to execute a filter. An edge struct stores all the information that is required for data commmunication.

After the initialization is completed, the PPU sends out mailbox messages to inform all SPUs that the data are ready. As soon as an SPU receives the confirmation message, it starts DMA and gets its control block from the PPU. Upon completion of the DMA, each SPU starts its memory allocation according to the content of the control block. The SPU local memory (local store) is partitioned into global data, library funtion, code memory, data memory, heap and statck.

Then upon completion of memory allocation, each SPU starts another DMA to initialize all the filter structs and edge structs. After initializaion, the library checks whether the is initial schedule required for peek operations. If initialization of peek buffers are needed, the library fires an initial schedule that runs each filter just enough to produce the correct amount of data.

Finally, the library implements an iterative routine that at each iteration parses the filter structs and edge structs and carry out the following actions.

- For each edge struct check whether a DMA command should be issued. The edge struct constains the information of the iterations when the DMA should start and stop, the source and destination of the DMA, and the size of the DMA. The starting and ending iterations of an DMA is assigned such that all data dependencies are respected and double buffering is enabled. After the DMA, the source and destination address of each edge is updated such that it is ready to start the next data communication.

- For each filter struct check whether it should be fired at the current iteration. If it should be fired, then gather all the data it requires and orginize the data into the correct format. Fire the filter and then scatter the data to each of its outgoing edges. After each firing, the actor also needs to update its incoming data address and outgoing address such that it is ready for the next execution.

Environment Setup

Inorder to execute the IBM Cell Backend, there are several environmental variables that needs to be set up. Since our IBM Cell Backend generates extended multi-threaded C code and relies on the IBM SDK to compile the generated code into executable binary, a working compilation tool chain for the IBM Cell BE is required. The verion tested for our implementaion is IBM SDK 3.0 which can be downloaded from the “IBM Cell Broadband Engine resource center” [36]. We also need to set up the environmental variables for executing the StreamIt compiler [57] and our IBM Cell backend. The variables that should be added are listed in the following,

- STREAMIT_HOME: The home directory of the StreamIt compiler and our backend code.
- ANTLRJAR: The directory that contains antlr.jar
- CLASS_PATH: The CLASS_PATh should include ANTLRJAR, STREAMIT _HOME/src, STREAMIT_HOME/3rdparty, STREAMIT_HOME/3rdparty /JFlex/jflex.jar.
- SPULIB_TOP_MYLIB: The directory that contains our static library implementation.

Commands

To faciliate the execution and debug of our IBM Cell backend, we also implement a perl script that sets up the directories in the local PC, runs the backend and generates IBM Cell BE code, compiles the code, connects to the remote Play Station 3 via net-ssh, set up the directories in the remote Play Station 3, connects to the remote Play Station 3 vis net-sftp, upload the inputs and executables into the corresponding directories, execute the program, download the output from remote Play Station 3 to the local PC, disconnect net-ssh, net-sftp, run the program in local PC, and compares the results produced by Play Station 3 and local PC. The perl script takes several arguments as shown below,

Usage: run inputfile benchmark iterations [load_input] [disable_regenerate]

A sample command is given by "run BeamFormer1 beamformer 100 true false". This command will compile the BeamFormer1.str under the directory "STREAMIT_

HOME/apps/benchmarks/asplos06/beamformer/streamit" for 100 iterations. The "load_input" and "disable_regenerate" arguments are optional. If load_input is specified as true, then a new input is uploaded to the Play Station 3 before the program execution. Otherwise the old input is adopted. If disable_regenerate is specified as true, then all previous data are cleared. Otherwise, the previous executable is used.

Outputs

There are several different categories of outputs being produced by our IBM Cell backend. The first category is our optimization outputs. The are presented as dot files and can be viewed graphically. To be more specific

- HeuInputGraph.dot: This dot file contains the graphical representation of the input program and all the information that is needed by our optimizatio.
- HeuOutputGraph.dot: This dot file contains the graphical representation of the solution our optimization, including filter to batch mapping, batch to processing mapping, iteration number of each filter etc..
- HeuFilterBatches.dot: This dot file contains the graphical representation of filter to batch mapping, the work distributation of each batch, the incoming edges, the outgoing edges, the intra dedges of each batch.
- HeuBufAllocation: This dot file contains the memory layout of each batch. The data memory is partitioned for each incoming edge, outgoing edge and intra-edge.

- HeuSPUs.dot: This dot file contains the graphical representation of filter to batch and batch to processor mapping.

The second category of outputs contains the multi-threaded C code for each filter (heu_str_spuN.c, N is positive integer), the main thread PPU code (heu_strppu.c), the header files and makefile. These files can be found in the same directory of the program code (the .str file). After compilation, there will be a build directory created that contains the executables for the PPU, all SPUs and a combined executable that links everything together (strppu).

The last category of outputs contains the output results from the Play Station 3 and local PC. The output file name for the results from remote Play Station 3 is "PS3.out.txt" and the output file name for the results from local PC is given by "$benchmark.out.txt". A comparison of the two files can validate the correctness of our multi-threaded IBM Cell BE executable.

APPENDIX B

Control Block Definitions, Helper Functions, and Cell BE Library Main Routine

Control Block Definitions

In this section we provide the code for various control block definitions that are utilized to implement the Cell BE library, namely SPU control block, Filter control block, edge control block.

SPU Control Block

The control block for an SPU is described in below.

```
typedef struct _control_block {
    /* current spu id */
    unsigned int spuId;
    /* number of times to run the program in one iteration */
    unsigned int scale;
    /* total number of stages of the entire application,
    including prolog, steady-state, epilog */
    unsigned int NUM_TOTAL_STAGES;
    /* number of filters of current batch */
    unsigned int NUM_FILTERS;
    /* address for macros of current batch */
    unsigned int filters_addr;
    /* buffer addr for file read */
    unsigned int input_array_addr;
    /* buffer addr for file writer */
    unsigned int output_array_addr;
    /* offset of split_join buffer */
    unsigned int split_join_start;
     /* size of split_join buffer */
```

```
    unsigned int split_join_size;
    /* the least common multiple of all fission factors */
    unsigned int fission_lcm;
    /* fission factor of the current batch */
    unsigned int fission_factor;
    /* fission executions of the current batch */
    unsigned int fission_exec;
    /* the index of current copy */
    unsigned int fission_index;
    /* whether peek_buf needs to be initialized */
    unsigned int init_peek_buf;
    /* padding for DMA */
    char pad[72];
} control_block;
```

Filter Control Block

The control block for a filter is described in below.

```
typedef struct _filter_s
{
    /* the id of a filter */
    unsigned int filterId;
    /* the execution start stage of a filter */
    unsigned int exec_start_stage;
    /* the execution end stage of a filter */
    unsigned int exec_end_stage;
    /* the base address of the input data */
    unsigned int exec_in_start;
```

```
/* the offset of the input data */
unsigned int exec_in_offset;
/* the base address of the output data */
unsigned int exec_out_start;
/* the offset of the output data */
unsigned int exec_out_offset;
/* the temporary buffer size required */
unsigned int input_buffer_sj_size;
/* the temporary buffer size required */
unsigned int output_buffer_sj_size;
/* the number of input data buffers */
unsigned int num_input_buffers;
/* the number of output data buffers */
unsigned int num_output_buffers;
/* the number of incoming edges of a filter */
unsigned int num_incoming_edges;
/* the number of outgoing edges of a filter */
unsigned int num_outgoing_edges;
/* the starting address of incoming edge pointer */
unsigned int incoming_edges_addr;
/* the starting address of outgoing edge pointer */
unsigned int outgoing_edges_addr;
/* the address of the init work function */
unsigned int work_func_init;
/* whether the output is a split */
unsigned int is_out_split;
/* whether the input is a joint */
unsigned int is_in_join;
```

```
    /* the address of the work function */
    unsigned int work_func;
    /* whether the init function for peek is completed */
    unsigned int init_done;
    /* the pointer to the incoming edges of the filter */
    edge_s * incoming_edges_ptr;
    /* the pointer to the outgoing edges of the filter */
    edge_s * outgoing_edges_ptr;
    /* the unroll factor of the entire graph due to fission */
    unsigned int lcm_index;
    /* the fission index of the current copy */
    unsigned int exec_index;
    /* incoming edges weight adjustor due to fission */
    unsigned int in_weight_scale;
    /* outgoing edges weight adjustor due to fission */
    unsigned int out_weight_scale;
    /* DMA padding */
    unsigned char pad[24];
} filter_s;
```

Edge Control Block

The control block for an edge is described in below.

```
typedef struct _edge_s
{
    /* the number of tokens popped from edge */
    unsigned int pop;
    /* the number of tokens pushed to edge */
```

```
unsigned int push;
/* the number of tokens peeks from edge */
unsigned int peek;
/* the weight pushed to edge in steady-state */
unsigned int in_weight;
/* the weight popped from edge in steady-state */
unsigned int out_weight;
/* the DMA start stage for the edge */
unsigned int DMA_start_stage;
/* the DMA end stage for the edge */
unsigned int DMA_end_stage;
/* the base start address for input data */
unsigned int DMA_in_start;
/* the offset for input data */
unsigned int DMA_in_offset;
/* the DMA step size for input data */
unsigned int DMA_in_step_size;
/* the base start addrerss for output data */
unsigned int DMA_out_start;
/* the offset for output data */
unsigned int DMA_out_offset;
/* the DMA step size for output data */
unsigned int DMA_out_step_size;
/* the execution offset for input data */
unsigned int exec_in_offset;
/* the execution offset for output data */
unsigned int exec_out_offset;
/* the offset for input data if it is a join */
```

```
unsigned int join_offset;
/* the step size for input data if it is a join */
unsigned int join_step_size;
/* the offset for output data if it is a split */
unsigned int split_offset;
/* the step size for output data if it is a split */
unsigned int split_step_size;
/* the DMA size */
unsigned int DMA_size;
/* the number of producer buffers */
unsigned int num_producer_buffers;
/* the number of consumer buffers */
unsigned int num_consumer_buffers;
/* the source index of the input data for a split */
unsigned int data_src_index;
/* the destination index of the output data for a join */
unsigned int data_dest_index;
/* type: 0 incoming to spu, 1 intra to spu,
2 outgoing from spu */
unsigned int type;
/* whether is reading or writing to PPU */
unsigned int is_read_write_ppu;
/* number of steady-state executions */
unsigned int steady_state_executions;
// used to implement data communication for fission
/* number of possible DNA_out_starts */
unsigned int num_in_targets;
 /* the target DMA in start address */
```

```
    unsigned int DMA_in_targets[7];
    /* number of possible DMA_out_starts */
    unsigned int num_out_targets;
    /* the target DMA out start address */
    unsigned int DMA_out_targets[7];
    /* the fission factor of the producer */
    unsigned int producer_fission_factor;
    /* the fission index of the producer */
    unsigned int producer_fission_exec;
    /* the fission factor of the consumer */
    unsigned int consumer_fission_factor;
    /* the fission index of the consumer */
    unsigned int consumer_fission_exec;
    /* the size of the peek buffer */
    unsigned int peek_buf;
    /* DMA padding */
    unsigned char pad[64];
} edge_s;
```

Helper Functions

The helper functions for reformatting the input data for a join filter and output data for a split data are described. The helper functions for mailbox communications are also provided.

```
/* SPU wait for incoming msg */
int spu_wait_mailbox()
{
```

```
    /* wait for ppe to inform spe to do the next step */
    do{ ;}while(!spu_stat_in_mbox ());
    return spu_read_in_mbox();
}

/* SPU write outgoing msg */
void spu_write_mailbox(unsigned int n)
{
    /* inform ppe the job is done*/
    do{ ;}while(!spu_stat_out_mbox());
    spu_write_out_mbox(n);
}

/* Gather data from all incoming edges to
split_join in buffer in the correct format */
void gather(filter_s * filter_ptr, edge_s * incoming_edges_ptr,
int num_incoming_edges, int data_start)
{
    int i, j, k;
    int cur_ptr=cb.split_join_start;
    edge_s * incoming_edge_ptr=incoming_edges_ptr;
    /* Initiate all join_offsets to be zero */
    for(i=0; i<num_incoming_edges; i++) {
        incoming_edge_ptr->join_offset=0;
        incoming_edge_ptr++;
    }

    /* Gathering data according to join rules */
```

```
    for(i=0; i<cb.scale*(filter_ptr->in_weight_scale
    * cb.fission_exec); i++) {
        incoming_edge_ptr=incoming_edges_ptr;
        for(j=0; j<num_incoming_edges; j++) {
            for(k=0; k<incoming_edge_ptr->in_weight; k++) {
                *((int*)cur_ptr) = *(int*)(data_start+
                incoming_edge_ptr->DMA_out_start+incoming_edge_ptr
                ->exec_in_offset+incoming_edge_ptr->peek_buf+
                incoming_edge_ptr->join_offset);

                cur_ptr += 4;
                incoming_edge_ptr->join_offset += 4;
            }
            incoming_edge_ptr++;
        }
    }
}

/* Reorgnize data in split_join out buffer into the correct
format, scatter data to all outgoing edges */
void scatter(filter_s * filter_ptr, edge_s * outgoing_edges_ptr,
int num_outgoing_edges, int data_start)
{
    int i, j, k;
    edge_s * outgoing_edge_ptr = outgoing_edges_ptr;
    /* Initiate all split_offsets to be zero */
    for(i=0; i<num_outgoing_edges; i++) {
        outgoing_edge_ptr->split_offset=0;
```

```
        outgoing_edge_ptr++;
    }
    int cur_ptr = cb.split_join_start+cb.split_join_size;
    /* Scattering data according to split rules*/
    for(i=0; i<cb.scale*(filter_ptr->out_weight_scale)
    *cb.fission_exec; i++) {
            outgoing_edge_ptr=outgoing_edges_ptr;
            for(j=0; j<num_outgoing_edges; j++) {
                for(k=0; k<outgoing_edge_ptr->out_weight; k++) {
                    *(int*)(data_start+outgoing_edge_ptr->
                    DMA_in_start+outgoing_edge_ptr->exec_out_offset
                    +outgoing_edge_ptr->peek_buf+outgoing_edge_ptr
                    ->split_offset) = *((int*)cur_ptr);

                    cur_ptr += 4;
                    outgoing_edge_ptr->split_offset += 4;
                }
                outgoing_edge_ptr++;
            }
    }
}
```

IBM Cell BE Library Main Routine

```
/* -------------------------------------------------- */
/*                 Start of main loop
/* -------------------------------------------------- */

filter_ptr = filters;
```

```
for(i=0; i<cb.NUM_FILTERS; i++){
    /* DMA in incoming edges of filter i */
    mfc_get(filter_ptr->incoming_edges_ptr, filter_ptr->
    incoming_edges_addr, sizeof(edge_s)*(filter_ptr->
    num_incoming_edges), tag_id, 0, 0);

    mfc_write_tag_mask(1<<tag_id);
    mfc_read_tag_status_all();

    /* DMA in outgoing edges of filter i */
    mfc_get(filter_ptr->outgoing_edges_ptr, filter_ptr->
    outgoing_edges_addr, sizeof(edge_s)*(filter_ptr->
    num_outgoing_edges), tag_id, 0, 0);

    mfc_write_tag_mask(1<<tag_id);
    mfc_read_tag_status_all();

    /* Move pointer to the next filter */
    filter_ptr ++;
}

/* Move filter pointer to the first filter */
filter_ptr=filters;

/* Fission factor/index of the batch */
int fission_lcm = cb.fission_lcm;
int fission_factor = cb.fission_factor;
```

```
int fission_exec = cb.fission_exec;
int fission_index = cb.fission_index;

/* For each filter we carry out the following operation */
if(cb.init_peek_buf == 0){
    for(i=0; i<cb.NUM_FILTERS; i++){
        /* Move incoming edge pointer to incoming edges */
        edge_s * incoming_edge_ptr =
                        filter_ptr->incoming_edges_ptr;
        /* Move outgoing edge pointer to outgoing edges */

        edge_s * outgoing_edge_ptr =
                        filter_ptr->outgoing_edges_ptr;

        /* ----------- Execution of a filter --------- */
        /* Initiate input data pointer */
        void * input_data_ptr = (void *)(data_start+
        incoming_edge_ptr->DMA_out_start+
                          incoming_edge_ptr->exec_in_offset);

        /* Initiate output data pointer */
        void * output_data_ptr = (void *)(data_start+o
        utgoing_edge_ptr->DMA_in_start+
                         outgoing_edge_ptr->exec_out_offset);

        /* A pointer to input_data_ptr */
        void ** input_data_dptr = &input_data_ptr;
        /* A pointer to output_data_ptr */
```

```
        void ** output_data_dptr = &output_data_ptr;

        /* execute init work function */
        (*(work_func *)filter_ptr->work_func_init)
        (input_data_dptr,output_data_dptr, cb.scale,
                              (void **)cb.split_join_start);
        /* move pointer to the next filter */
        filter_ptr ++;
    }
}

/* Determine whether a certain operation should take place */
unsigned int cur_stage=0;

/* NUM_TOTAL_STAGES is basically the prolog executions
+ steady-state executions + epilog executions */

for(cur_stage=0; cur_stage<cb.NUM_TOTAL_STAGES; cur_stage++) {
/* Wait for mbox, it should read the filter address */
spu_wait_mailbox();

/* Move filter pointer to the first filter */
filter_ptr=filters;

/* For each filter we carry out the following operation */
for(i=0; i<cb.NUM_FILTERS; i++) {

    /* Move incoming edge pointer to incoming edges pointer */
```

```
edge_s * incoming_edge_ptr=filter_ptr->incoming_edges_ptr;

/* ------------- DMA input from PPU ------------------- */
/* For filter reads from PPU, we issue mfc_get to get the
 * data from PPU. All other data communications are carried
 * out by the producer issue mfc_put (examing outgoing
 * edges). In this way, any DMA operation is done by some
 * DMA engine of a SPU (most efficient). */
for(j=0; j<filter_ptr->num_incoming_edges; j++){
    /* If edge reads from PPU */
    if(incoming_edge_ptr->is_read_write_ppu==1){
    if((cur_stage >= incoming_edge_ptr->DMA_start_stage)&&
    (cur_stage <= incoming_edge_ptr->DMA_end_stage)){

    unsigned int ls_addr = data_start+incoming_edge_ptr->
    DMA_out_start+incoming_edge_ptr->DMA_out_offset+
    incoming_edge_ptr->peek_buf;

    unsigned int remote_addr = incoming_edge_ptr->
    DMA_in_targets[0]+incoming_edge_ptr->DMA_in_start
    +incoming_edge_ptr->DMA_in_offset;

    int exec_idx = 0;
    for(exec_idx = 0; exec_idx < incoming_edge_ptr->
    consumer_fission_exec; exec_idx ++){
        int global_index = fission_index * fission_exec +
                           exec_idx;
        int target_offset_index = global_index %
```

```
    incoming_edge_ptr->producer_fission_exec;

    int    consumer_offset = exec_idx * incoming_edge_ptr
                             ->DMA_size;
    int    producer_offset = target_offset_index *
                             incoming_edge_ptr->DMA_size;

    mfc_get(ls_addr+consumer_offset,
    remote_addr+producer_offset,
    incoming_edge_ptr->DMA_size, tag_id, 0, 0);
    }

/* Update DMA_in_offset */
  incoming_edge_ptr->DMA_in_offset += incoming_edge_ptr->
  DMA_in_step_size*incoming_edge_ptr->producer_fission_exec;

/* Update DMA_out_offset */
  incoming_edge_ptr->DMA_out_offset += (incoming_edge_ptr->
  DMA_out_step_size+incoming_edge_ptr->peek_buf)*
  incoming_edge_ptr->consumer_fission_exec;

/* When DMA_in_offset exceeds the producer buffers */
  if(incoming_edge_ptr->DMA_in_offset >= incoming_edge_ptr->
  DMA_in_step_size * incoming_edge_ptr->num_producer_buffers
  * incoming_edge_ptr->producer_fission_exec) {
         incoming_edge_ptr->DMA_in_offset=0;
        }
 /* When DMA_out_offset exceeds the consumer buffers */
```

```
        if(incoming_edge_ptr->DMA_out_offset >= (incoming_edge_ptr
        -> DMA_out_step_size+incoming_edge_ptr->peek_buf) *
        incoming_edge_ptr->num_consumer_buffers *
        incoming_edge_ptr->consumer_fission_exec) {
                incoming_edge_ptr->DMA_out_offset=0;
                }
            }
        }
    /* Move pointer to next edge */
    incoming_edge_ptr ++;
    }
/* Move pointer to the next filter */
filter_ptr ++;
}

/* Move filter pointer to the first filter */
filter_ptr=filters;

/* For each filter we carry out the following operation */
for(i=0; i<cb.NUM_FILTERS; i++) {

    /* Move outgoing edge pointer to outgoing edges of
    the current filter */
    edge_s * outgoing_edge_ptr=filter_ptr->outgoing_edges_ptr;

    /* DMA out for each outgoing edge */
    outgoing_edge_ptr=filter_ptr->outgoing_edges_ptr;
```

```
for(j=0; j<filter_ptr->num_outgoing_edges; j++) {

/* If it is an outgoing edge from the current spu */
if(outgoing_edge_ptr->type==2) {

    if((cur_stage >= outgoing_edge_ptr->DMA_start_stage)&&
    (cur_stage <= outgoing_edge_ptr->DMA_end_stage)) {
    unsigned int ls_addr = data_start+outgoing_edge_ptr->
    DMA_in_start+outgoing_edge_ptr->DMA_in_offset+
    outgoing_edge_ptr->peek_buf;

    int exec_idx = 0;
    for(exec_idx = 0; exec_idx < outgoing_edge_ptr->
    producer_fission_exec; exec_idx ++) {
        int global_index = fission_exec * fission_index +
                           exec_idx;
        int target_index = global_index / outgoing_edge_ptr->
                           consumer_fission_exec;
        int target_offset_index = global_index %
                   outgoing_edge_ptr->consumer_fission_exec;
        int    producer_offset = exec_idx * outgoing_edge_ptr
                                 ->DMA_size;
        int    consumer_offset = target_offset_index *
                                  outgoing_edge_ptr->DMA_size;
        unsigned int remote_addr = outgoing_edge_ptr->
        DMA_out_targets[target_index]+outgoing_edge_ptr->
        DMA_out_start+outgoing_edge_ptr->DMA_out_offset
        +outgoing_edge_ptr->peek_buf;
```

```
    mfc_put(ls_addr+producer_offset, remote_addr+
    consumer_offset, outgoing_edge_ptr->DMA_size,
    tag_id, 0, 0);
}
/* Update pointer */
outgoing_edge_ptr->DMA_in_offset += (outgoing_edge_ptr->
DMA_in_step_size+outgoing_edge_ptr->peek_buf) *
outgoing_edge_ptr->producer_fission_exec;

outgoing_edge_ptr->DMA_out_offset += (outgoing_edge_ptr->
DMA_out_step_size+outgoing_edge_ptr->peek_buf) *
outgoing_edge_ptr->consumer_fission_exec;

if(outgoing_edge_ptr->DMA_in_offset >= (outgoing_edge_ptr
->DMA_in_step_size+outgoing_edge_ptr->peek_buf) *
outgoing_edge_ptr->num_producer_buffers *
outgoing_edge_ptr->producer_fission_exec){
    outgoing_edge_ptr->DMA_in_offset = 0;
    }

if(outgoing_edge_ptr->DMA_out_offset >= (outgoing_edge_ptr
->DMA_out_step_size+outgoing_edge_ptr->peek_buf) *
outgoing_edge_ptr->num_consumer_buffers *
outgoing_edge_ptr->consumer_fission_exec){
    outgoing_edge_ptr->DMA_out_offset = 0;
    }
}
```

```
    }
    /* Move pointer to next edge */
    outgoing_edge_ptr ++;
    }
    /* Move pointer to the next filter */
    filter_ptr ++;
}

/* Move filter pointer to the first filter */
filter_ptr = filters;

/* For each filter we carry out the following operation */
for(i=0; i<cb.NUM_FILTERS; i++) {
    /* Move incoming edge pointer to incoming edges pointer */
    edge_s * incoming_edge_ptr=filter_ptr->incoming_edges_ptr;

    /* Move outgoing edge pointer to outgoing edges pointer */
    edge_s * outgoing_edge_ptr=filter_ptr->outgoing_edges_ptr;

    /* --------- Execution of a filter ----------- */
    if((cur_stage >= filter_ptr->exec_start_stage)&&(cur_stage
    <= filter_ptr->exec_end_stage)) {
    /* Initiate input data pointer */
    void * input_data_ptr = (void *)(data_start+incoming_edge_ptr
    ->DMA_out_start+incoming_edge_ptr->exec_in_offset+
    incoming_edge_ptr->peek_buf-4*(incoming_edge_ptr->
    incoming_edge_ptr->pop));
```

```
/* Initiate output data pointer */
void * output_data_ptr = (void *)(data_start+outgoing_edge_ptr
->DMA_in_start+outgoing_edge_ptr->exec_out_offset
+outgoing_edge_ptr->peek_buf);

/* A pointer to input_data_ptr */
void ** input_data_dptr = &input_data_ptr;

/* A pointer to output_data_ptr */
void ** output_data_dptr = &output_data_ptr;

int exec_idx = 0;
/* Go to different branches depending on the input
(split?) and output (join?) */
if( filter_ptr->is_in_join || filter_ptr->is_out_split) {

    /* Case: join input single output/duplicate output */
    if(filter_ptr->is_in_join && !filter_ptr->is_out_split) {
    input_data_ptr = (void *)(cb.split_join_start);
    input_data_dptr = &input_data_ptr;

    /* Gather input from all incoming edges */
    gather(filter_ptr, incoming_edge_ptr, filter_ptr->
    num_incoming_edges, data_start);

    /* Start filter execution */
    for(exec_idx = 0; exec_idx < fission_exec; exec_idx ++) {
        (*(work_func *)filter_ptr->work_func)(
```

```
    input_data_dptr, output_data_dptr, cb.scale,
    (void **)cb.split_join_start);
    }
} else if(!filter_ptr->is_in_join && filter_ptr->
is_out_split) { /* Case: single input split output */

output_data_ptr = (void *)(cb.split_join_start+
cb.split_join_size);
output_data_dptr = &output_data_ptr;
/* Start filter execution */
for(exec_idx = 0; exec_idx < fission_exec; exec_idx ++) {
    (*(work_func *)filter_ptr->work_func)(input_data_dptr,
    output_data_dptr, cb.scale,
    (void **)cb.split_join_start);
    }

/* Scattering output from split_join_start+
split_join_size to all outgoing edges */
    scatter(filter_ptr, outgoing_edge_ptr,
    filter_ptr->num_outgoing_edges, data_start);
} else { /* Join input, split output */
input_data_ptr = (void *)(cb.split_join_start);
input_data_dptr = &input_data_ptr;
output_data_ptr = (void *)(cb.split_join_start+
cb.split_join_size);
output_data_dptr = &output_data_ptr;

/* Gather input from all incoming edges */
```

```
        gather(filter_ptr, incoming_edge_ptr,
        filter_ptr->num_incoming_edges, data_start);

        /* Start filter execution */
        for(exec_idx = 0; exec_idx < fission_exec; exec_idx ++) {
            (*(work_func *)filter_ptr->work_func)(input_data_dptr,
            output_data_dptr, cb.scale,
            (void **)cb.split_join_start);
        }
        /* Scattering output to all outgoing edges */
        scatter(filter_ptr, outgoing_edge_ptr,
        filter_ptr->num_outgoing_edges, data_start);
        }
    } else {
        for(exec_idx = 0; exec_idx < fission_exec; exec_idx ++) {
        (*(work_func *)filter_ptr->work_func)(input_data_dptr,
        output_data_dptr, cb.scale, (void **)cb.split_join_start);
        }
    }

    /* Keep tracking execution pointers of each incoming edge */
    incoming_edge_ptr=filter_ptr->incoming_edges_ptr;
    for(j=0; j<filter_ptr->num_incoming_edges; j++) {
        incoming_edge_ptr->exec_in_offset += (incoming_edge_ptr->
        DMA_size+incoming_edge_ptr->peek_buf)*incoming_edge_ptr->
        consumer_fission_exec;
        if(incoming_edge_ptr->exec_in_offset==(incoming_edge_ptr->
        DMA_size+incoming_edge_ptr->peek_buf)*incoming_edge_ptr->
```

```
num_consumer_buffers*incoming_edge_ptr->
consumer_fission_exec) {
    incoming_edge_ptr->exec_in_offset=0;
}

if(incoming_edge_ptr->peek_buf != 0){
   if(incoming_edge_ptr->exec_in_offset != 0){
       unsigned int dest = data_start+incoming_edge_ptr->
       DMA_out_start+incoming_edge_ptr->exec_in_offset+
       incoming_edge_ptr->peek_buf-4*(incoming_edge_ptr->
       peek-incoming_edge_ptr->pop);

    unsigned int src = (unsigned int)(*input_data_dptr);
    memcpy((void*)dest, (void*)src, 4*(incoming_edge_ptr
    ->peek-incoming_edge_ptr->pop));

} else {
    unsigned int dest = data_start+incoming_edge_ptr->
    DMA_out_start+incoming_edge_ptr->peek_buf-4*(
    incoming_edge_ptr->peek-incoming_edge_ptr->pop);

    unsigned int total_buf_size = (incoming_edge_ptr->
    DMA_size+incoming_edge_ptr->peek_buf)*
    incoming_edge_ptr->num_consumer_buffers;

    unsigned int src = (unsigned int)(*input_data_dptr);
    memcpy ((void*)dest, (void*)src, 4*(incoming_edge_ptr
    ->peek-incoming_edge_ptr->pop));
```

```
        }
    }
    incoming_edge_ptr ++;
}
/* Keep tracing execution pointers of each outgoing edge */
outgoing_edge_ptr=filter_ptr->outgoing_edges_ptr;
for(j=0; j<filter_ptr->num_outgoing_edges; j++)
{
    outgoing_edge_ptr->exec_out_offset += (outgoing_edge_ptr
    ->DMA_size+outgoing_edge_ptr->peek_buf)*outgoing_edge_ptr
    ->producer_fission_exec;

    if(outgoing_edge_ptr->num_producer_buffers==0 &&
    outgoing_edge_ptr->exec_out_offset==(outgoing_edge_ptr->
    DMA_size+outgoing_edge_ptr->peek_buf)*outgoing_edge_ptr->
    num_consumer_buffers*outgoing_edge_ptr
    ->producer_fission_exec){
        outgoing_edge_ptr->exec_out_offset=0;
    } else if(outgoing_edge_ptr->exec_out_offset==
    (outgoing_edge_ptr->DMA_size+outgoing_edge_ptr->peek_buf)
    *outgoing_edge_ptr->num_producer_buffers*outgoing_edge_ptr
    ->producer_fission_exec){
        outgoing_edge_ptr->exec_out_offset=0;
        }
    outgoing_edge_ptr ++;
    }
}
/* Move pointer to the next filter */
```

```
    filter_ptr ++;
}

mfc_write_tag_mask(1<<tag_id);
mfc_read_tag_status_all();

/* Write mbox */
spu_write_mailbox(cb.spuId);
```