StreamWorks: An Energy-efficient Embedded Co-processor for Stream Computing

by

Amrit Panda

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

Approved August 2014 by the
Graduate Supervisory Committee:

Karam S. Chatha, Co-Chair
Carole-Jean Wu, Co-Chair
Chaitali Chakrabarti
Aviral Shrivastava

ARIZONA STATE UNIVERSITY

December 2014

ABSTRACT

Stream processing has emerged as an important model of computation especially in the context of multimedia and communication sub-systems of embedded System-on-Chip (SoC) architectures. The dataflow nature of streaming applications allows them to be most naturally expressed as a set of kernels iteratively operating on continuous streams of data. The kernels are computationally intensive and are mainly characterized by real-time constraints that demand high throughput and data bandwidth with limited global data reuse. Conventional architectures fail to meet these demands due to their poorly matched execution models and the overheads associated with instruction and data movements.

This work presents StreamWorks, a multi-core embedded architecture for energy-efficient stream computing. The basic processing element in the StreamWorks architecture is the StreamEngine (SE) which is responsible for iteratively executing a stream kernel. SE introduces an instruction locking mechanism that exploits the iterative nature of the kernels and enables fine-grain instruction reuse. Each instruction in a SE is locked to a Reservation Station (RS) and revitalizes itself after execution; thus never retiring from the RS. The entire kernel is hosted in RS Banks (RSBs) close to functional units for energy-efficient instruction delivery. The dataflow semantics of stream kernels are captured by a context-aware dataflow execution mode that efficiently exploits the Instruction Level Parallelism (ILP) and Data-level parallelism (DLP) within stream kernels.

Multiple SEs are grouped together to form a StreamCluster (SC) that communicate via a local interconnect. A novel software FIFO virtualization technique with split-join functionality is proposed for efficient and scalable stream communication across SEs. The proposed communication mechanism exploits the Task-level parallelism (TLP) of the stream application. The performance and scalability of the

communication mechanism is evaluated against the existing data movement schemes for scratchpad based multi-core architectures. Further, overlay schemes and architectural support are proposed that allow hosting any number of kernels on the StreamWorks architecture. The proposed oevrlay schemes for code management supports kernel(context) switching for the most common use cases and can be adapted for any multi-core architecture that use software managed local memories.

The performance and energy-efficiency of the StreamWorks architecture is evaluated for stream kernel and application benchmarks by implementing the architecture in 45nm TSMC and comparison with a low power RISC core and a contemporary accelerator.

DEDICATION

To my parents, sister and Aditi.

ACKNOWLEDGEMENTS

As I started my journey as a Ph.D student under the able guidance of Prof. Karam Chatha, little did I know how much more I would learn and how much further I would be able to push the boundaries of existing knowledge. He taught me how to identify key research challenges and come up with creative solutions to address the same. Not only did he provide me with technical guidance, but also helped me identify my strengths, work on my weaknesses and grow immensely as a person. It was due to his constant motivation, I enjoyed every moment of my Ph.D, not a single day during my Ph.D did I feel that this was not meant for me. I want to thank Prof. Chatha for everything; his guidance has led me to evolve as a researcher and as a person.

I would like to extend a special word of thanks to my co-chair, Prof. Carole Jean-Wu for her guidance and motivation. I really appreciated the effort and interest that you took in my work, always making sure to read over my proposals paying attention to every detail and providing very valuable and timely feedback. I am very grateful to my committee for their continued guidance and valuable input to this thesis. I would like to thank Prof. Chaitali Chakrabarti and Prof. Aviral Shrivastava for the helpful pointers that enabled critical evaluation of my work. In addition to the committee, I really appreciate the support of the faculty at the ASU Computer Science department that I could not have completed my thesis without. Specifically, I would like to thank Prof. Rida Bazzi for developing my skills in compiler engineering, Prof. Partha Dasgupta for driving me to take a practical approach to distributed operating systems and Prof. Andrea Richa for introducing me to combinatorial optimizations. It was extremely helpful to have your teachings in the arsenal of my research tools.

During my years at ASU, I was very fortunate to work with a number of people who have added great value to my work. In addition to learning technical skills, these people have greatly enriched my life. I would like to thank Michael Baker, Glenn Leary, Weijia Che, Nikhil Ghadge, Anil Chunduru, Haeseung Lee, Jyothi Swaroop Arlagadda, Zisong Zhao, Akhil Arunkumar, Shin-Ying Lee, Sushil Kumar and Manoj Venkat for their valuable feedback and constructive criticism during discussions in the lab.

I would like to thank my friends in brickyard Vinay Hanumaiah, Reiley Jeyapaul, Niranjan Kulkarni, Yooseong Kim, Mahdi Hamzeh, Sujogya Banerjee, Digant Desai, Nishant Nukula and Pritam Gundecha for the stimulating discussions during the afternoon coffee breaks and sharing experiences that helped me take the right decisions and steered me through my Ph.D in several occasions.

A very special thanks to my academic advisors Ms. Martha Vander Berg and Ms. Cynthia Donahue for making sure that I'm on track for my degree. I really appreciate your time and consideration in processing the numerous forms and taking care of all the paperwork to make my experience in graduate school a pleasant one.

In our busy work lives, it is easy to take for granted the support that we get from the administration. The timely support of Ms. Pamela Dunn, Ms. Monica Dugan and Ms. Theresa Chai with expenses, submissions, scheduling of defense/talks greatly helped take those tasks off my plate so that I could focus better on my research.

After gaining some rudimentary research skills, I am grateful to Ashu Razdan and Rudy Beraha for having given me a chance to gain industry experience as part of my summer internships with the OOTCS and CRD departments at Qualcomm Research, San Diego. Thank you very much for having confidence in my research work and giving me a chance to work alongside your very able team.

Last, but certainly not the least, I want to thank my parents for their unwavering support, guidance, encouragement and love at every step of the long winding stairway that I have taken to be here today. It is because of them that I have been able to realize my dream. With a sister like mine, Atasi Panda, I felt like an army of cheerleaders was with me the whole time! I must express my gratitude to my wife and best friend Aditi Singh for her continuous support, encouragement, quiet patience and unyielding love. It's her love that has become the warp and woof of the last ten years of my life. I also want to thank my in-laws for keeping faith in my professional and personal goals and extending their unconditional love and support at every stage. The inspiration that I have received from my family has been my strength and made my journey easy and fun. This thesis would not have been possible without the love and support that I have received. I want to dedicate this thesis to my family.

TABLE OF CONTENTS

Page

LIST OF TABLES . . . x

LIST OF FIGURES . . . xii

CHAPTER

1 INTRODUCTION . . . 1

1.1 Contributions . . . 7

2 THE STREAMENGINE ARCHITECTURE . . . 12

2.1 Introduction . . . 12

2.2 Related Work . . . 13

2.3 Context-aware Dataflow Execution and Instruction Locking . . . 21

2.3.1 Token Distribution Network . . . 23

2.3.2 Conditional Dataflow . . . 25

2.3.3 Dataflow Monitor and Token regulation . . . 30

2.3.4 Stream Index Generator, Iteration Decoupling and Loop Merging . . . 32

2.4 SE Program Execution Example . . . 33

2.5 Executing Nested Conditionals . . . 40

2.6 Executing Nested Loops . . . 46

2.7 Evaluation . . . 54

2.7.1 Experimental Set-up . . . 54

2.7.2 Instruction Delivery Energy Savings . . . 61

2.7.3 Impact of CDE on IPC . . . 63

2.7.4 SE Energy-efficiency . . . 64

2.7.5 Impact of Transforming Control-flow to Dataflow . . . 67

2.7.6 Performance Scaling with SE Functional Units . . . 69

2.7.7 VLSI Implementation, Area and Power ... 70
2.7.8 Comparison with ELM ... 70
3 THE MULTI-CORE DATAPLANE ... 73
3.1 Introduction ... 73
3.2 Related Work ... 74
3.3 FIFO Virtualization ... 78
3.3.1 The Input Channel ... 79
3.3.2 The Push Unit ... 80
3.3.3 Credit-based Back Pressure ... 83
3.4 Evaluation ... 84
3.4.1 Experimental Setup ... 86
3.4.2 Results ... 88
4 KERNEL OVERLAY AND SCALABILITY ... 90
4.1 Limitations of the Current Architecture ... 90
4.2 Kernel Overlay ... 91
4.2.1 Kernel Fission ... 92
4.2.2 Overlay Schedule ... 93
4.2.3 Implementation ... 97
4.3 Evaluation ... 100
4.3.1 Experimental Setup ... 101
4.3.2 Results ... 102
4.3.3 Discussion ... 106
5 CONCLUSION ... 109

CHAPTER Page

REFERENCES 111

APPENDIX

A EVALUATION WITH GPU 116

A.1 EVALUATION OF SE WITH GPU SM AS BASELINE 117

A.1.1 Experimental Setup 117

A.1.2 Results 118

A.2 EVALUATION OF SW WITH GPU AS BASELINE 121

A.2.1 Experimental Setup 121

A.2.2 Results 121

LIST OF TABLES

Table Page

1.1 Comparison of SE with VLIW, SIMD and GPU(SM) 10
2.1 Comparison of SE with Conventional Dataflow Architectures 17
2.2 StreamEngine Configuration 34
2.3 Configuration of Stream (s-RSB), Compute (c-RSB) and Memory (m-RSB) RSBs 34
2.4 Configuration of Stream (s-RSB), Compute (c-RSB) and Memory (m-RSB) RSBs 35
2.5 Configuration of Stream Index Generators 36
2.6 Configuration of Stream Index Generators 43
2.7 Configuration of Stream (s-RSB), Compute (c-RSB) and Memory (m-RSB) RSBs 43
2.8 Configuration of Stream (s-RSB), Compute (c-RSB) and Memory (m-RSB) RSBs 44
2.9 Configuration of Stream Index Generators 51
2.10 Configuration of Stream (s-RSB), Compute (c-RSB) and Memory (m-RSB) RSBs 51
2.11 Configuration of Stream (s-RSB), Compute (c-RSB) and Memory (m-RSB) RSBs 52
2.12 Benchmarks Representing Stream Kernels 54
2.13 StreamEngine Configuration 56
2.14 SPARC V8-based LEON Configuration 59
2.15 ARM Cortex A-15 Configuration 60
3.1 Description of Push Unit Reservation Station Fields 80
3.2 Push Unit Contents for Weighted Round-robin Split E.g. 81

Table Page

3.3 Push Unit Contents at Producer SE for Weighted Round-robin Join Example 82

3.4 Description of Credit Reservation Station Block 83

3.5 Stream Application Benchmarks 86

4.1 Kernel Deployment Round 0 95

4.2 Kernel Deployment Round 1 95

4.3 Kernel Deployment Round 0 97

4.4 Kernel Deployment Round 1 97

4.5 Kernel Deployment Round 2 98

4.6 StreamEngine Configuration 101

LIST OF FIGURES

Figure		Page
1.1	Streaming Specification of 8 x 8 Inverse Discrete Cosine Transform Derived from StreamIt Benchmark Suite	1
1.2	Syncronous Data Flow	5
1.3	Cyclo-static Data Flow	6
1.4	Boolean Data Flow	6
1.5	Venn Diagram Showing the Classification of Dataflow Models	7
2.1	StreamEngine with Communication Unit	13
2.2	Static Dataflow Architecture	14
2.3	Tagged-token Dataflow Architecture	15
2.4	Structure of a Basic Reservation Station	22
2.5	Conversion of If-then-else Construct	26
2.6	Structure of the Extended Reservation Station	27
2.7	Nested Conditional Example	28
2.8	Example of a Dataflow Graph that Causes Token Accumulation	31
2.9	Dataflow Graph (DFG) of the Example Kernel	36
2.10	DFG Mapping and RS/SIG/TDN State during Execution	37
2.11	Dataflow Graph of the Sample Kernel with Nested Conditional.	42
2.12	DFG Mapping and RS/SIG/TDN State During Execution	46
2.13	Dataflow Graph of the iDCT Kernel After Loop Merging.	50
2.14	DFG Mapping and RS/SIG/TDN State During Execution of the iDCT Kernel	53
2.15	Power Characterization Methodology for SE and LEON	58
2.16	Instruction Delivery Energy Comparison with Cache-based and SPM-based LEON	61

Figure Page

2.17 Instruction Delivery Energy Comparison with ARM and ARM NEON . 62
2.18 Impact of CDE on IPC 63
2.19 Total Energy per Kernel of the LEON RISC Core Normalized to that of a SE 64
2.20 Energy-efficiency of SE and SPM-based LEON Normalized to that of Cache-based LEON 65
2.21 Total Energy per Kernel of the ARM Cortex A-15 Core Normalized to that of a SE 66
2.22 Energy-efficiency of SE and ARM Cortex A-15 with SIMD Extensions Normalized to that of ARM Cortex A-15 67
2.23 Average Stalls per Instruction with Predicate in a SE 68
2.24 SE Performance Scaling: Stream and Compute Instructions per Cycle . 69
2.25 Layout and Area Breakdown of SE 71
2.26 Power Breakdown of SE 71
3.1 StreamWorks Architecture 74
3.2 Weighted Round-robin Split and Join Examples 81
3.3 Buffer Requirement for the 3 DMA Schemes Normalized to the StreamWorks Buffer Usage and Software Pipeline Stages 87
3.4 Normalized Throughput (in cc) for DMA Schemes 87
3.5 Performance as a Function of StreamWorks Clusters 89
4.1 Structure of the Shadow RS 99
4.2 Instruction Delivery Energy Comparison 103
4.3 Total Energy Comparison of Various Overlay Schemes with RISC 104
4.4 Energy-efficiency of SE Configurations Normalized to that of RISC 105

Figure Page

4.5 Instruction Delivery Energy of the Overlay Schemes Normalized to SE without Overlay . . . 106

4.6 Total Energy of the OS-I and OS-II Normalized to that of SE without Overlay . . . 107

4.7 Energy-efficiency of the OS-I and OS-II Normalized to that of SE without Overlay . . . 108

A.1 Performance (Throughput) Comparison of SE with Kepler GK110 . . . 119

A.2 Energy-efficiency Comparison of SE with Kepler GK110 . . . 120

A.3 Performance Comparison of StreamWorks with GPU (Kepler GK110) . . . 122

A.4 Energy-efficiency Comparison of StreamWorks with GPU (Kepler GK110) 123

Chapter 1

INTRODUCTION

In the past decade stream computing has emerged as an important model of computation. It is particularly suitable for expressing computation in communication (network processing, wireless communication standards) and multimedia (graphics, video/audio codec) application domains. Streaming applications demonstrate dataflow behavior at the application level. They can be most naturally expressed as set of concurrent kernels (or actors) that iteratively operate on streams of data, and communicate with each other through software FIFOs. They exhibit a stable communication and computation pattern with occasional modification of stream flow. Streaming applications are typically computation intensive, and have to satisfy high throughput requirements.

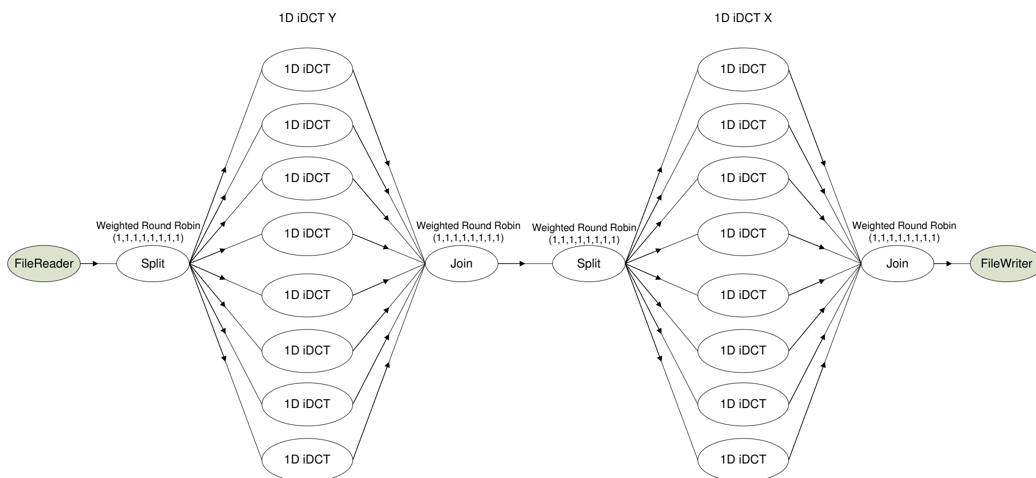


Figure 1.1: Streaming Specification of 8 x 8 Inverse Discrete Cosine Transform Derived from StreamIt Benchmark Suite

Figure 1.1 depicts the mapping of a 8x8 inverse Discrete Cosine Transform (iDCT) to the stream model. Each node (oval) in the figure corresponds to a stream kernel,

while each arrow represents a data stream transfer. In this case, each data stream is composed of floating point elements. The *FileReader* and *FileWriter* are special kernels that are responsible to stream data in and out for the application respectively. Also, note the special actors *Split* and *Join* that have a fan-out and and fan-in greater than one respectively. The *Split* and *Join* actors can implement a number of algorithms (e.g. duplicate, weighted round robin etc.) for the divergence and convergence of data streams. In this case, both the *Split* and *Join* actors implement weighted round robin with a weight of 1. Apart from the special actors viz. *FileReader, FileWriter, Split, Join*, the overall iDCT application requires 2 stages of 8 iDCT kernels each. Each iDCT kernel belonging to a stage communicates with each of the iDCT kernel from the other stage. The *Join* and *Split* in the center of the stream graph enable this communication pattern.

A number of programming languages and formats have emerged in academia (Brook Buck (2001), StreamIt Thies *et al.* (2002), KernelC, StreamC) and industry (CUDA, OpenCL, OpenGL) for specifying streaming applications.

Due to their inherent dataflow characteristics, stream applications demonstrate large amounts of parallelism. Streaming applications exhibit task-level parallelism as the operations within a kernel are performed independently on each stream element. Therefore, two kernels can be concurrently executed as long as the input/output conditions are satisfied on their respective FIFOs (input FIFO not empty and output FIFO not full). A kernel is repeatedly executed on independent sets of data items, and thereby demonstrates data-level parallelism (DLP). One can easily exploit the DLP exhibited by stateless[1] kernels using a SIMD-like execution model allowing multiple iterations in flight. The kernels themselves are compute intensive, and typically

[1] A kernel (or actor) is called stateless if its current iteration is independent on any of its previous iteration(s). A kernel with inter-iteration dependencies is called stateful.

show large amounts of instruction-level parallelism. The applications do not demonstrate data locality as they operate on streaming inputs. However, they do posses large amounts of instruction locality due to their repetitive execution. Another key characteristic of stream actors is their compact code footprint (typically less than 50 for each actor in the StreamIt benchmark suite).

The stream programming model can be efficiently implemented in hardware because in addition to exposing the parallelism offered by stream applications, it also exposes the communication behavior that demands high data bandwidth; thus exhibiting characteristics that are well matched to the capabilities of modern VLSI. Given the target architecture, the stream programming model also allows the compiler to perform several high level optimizations. However, without proper architectural support, it presents new challenges to programming tools. In recent past, several architectures have been proposed Khailany *et al.* (2001); Taylor *et al.* (2002); Lin *et al.* (2006); Yu *et al.* (2008); Woh *et al.* (2009); Dally *et al.* (2008) and commercialized Pham *et al.* (2006); Johnson and Kunze (2003); Baines and Pulley (2004) that support stream computing. As many of these architectures are aimed at embedded computing domains they are implemented as heterogeneous multi-core or System-on-Chip (SoC) designs. The top-level architecture consists of a control plane or application processor (for example an ARM core) that executes the more conventional workloads such as the operating system, JavaVM and so on. The stream processors that serve as data plane co-processors implement the computation intensive workloads. Low power stream computing has received considerable attention in recent years due to the convergence of communication and multimedia applications on the same smart mobile device (cell phone or tablet).

In the past, several models of computation have been proposed to capture the execution semantics of streaming applications. Following are some of the important

streaming computation models that have gained popularity due to their relevance to more frequent streaming workloads.

- **Kahn Process Network (KPN):** KPN is arguably the fundamental computation model for all other streaming computation models Geilen and Basten (2003). A KPN program is a set of processes that communicate through a network of unidirectional infinite FIFO queues. A read from the queue is blocking (when queue is empty) while a write is non-blocking (since FIFO is of infinite size). The key feature of KPNs is that they are determinate i.e. the outcome does not depend on the execution order of the processes and therefore they can be scheduled sequentially or in parallel. However, one major drawback of KPNs is they cannot be scheduled statically. Moreover special care has to be taken for bounded execution of KPNs.

- **Data Flow Networks :** A typical implementation of a KPN involves a process executing until its input FIFO is empty and gets context switched by its consumer process. This is mainly because of the lack of the notion of a quantum of computation. Dataflow process networks define a quantum of computation by decomposing processes into repeated firings of *actors*. Data Flow networks are executed by scheduling the actor firings as opposed to context switching. The following are a few data flow networks relevant to streaming:

    1. **Synchronous Data Flow (SDF):** SDF Lee and et al. (1987); Lee and Messerschmitt (1987) is a restriction of KPN in which the number of tokens produced or consumed by each process (node) on each firing is fixed. This a priori knowledge of the number of tokens allows the processes to be scheduled statically. Also, SDF programs guarantee memory boundedness and can therefore be scheduled on limited memory embedded processors.

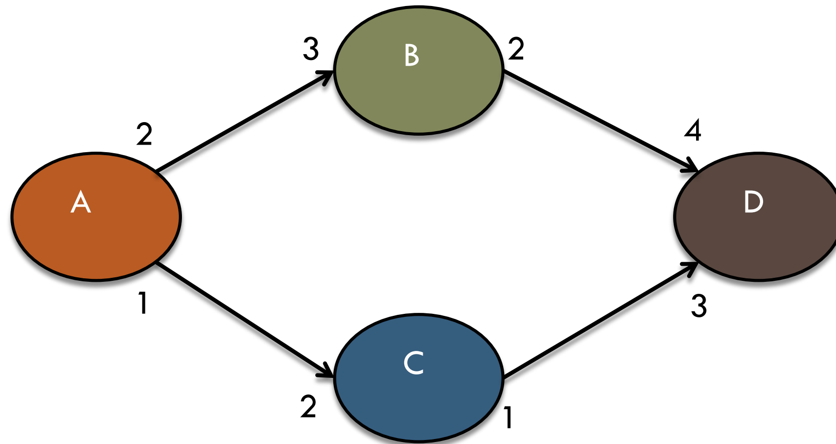


Figure 1.2: Syncronous Data Flow

2. **Cyclo-static Data Flow (CSDF) :** CDSF Bilsen *et al.* (1995); Parks *et al.* (1995) is an extension of SDF with additional support for processes that implement algorithms which are critically changing in a cyclic manner with a predefined behavior. To accommodate such processes, the nodes in CSDF have a set of tokens associated with their input and output channels. For example, if a node has the set x, y, z associated with its output channel, this node produces x tokens during its first invocation, y tokens during the second, z tokens during the third invocation, then again x during the fourth and so on. Basically the process cycles through this set. For SDF, this set is a singleton set and thus every invocation produces (and consumes) the same number of tokens. The weighted round-robin actors in the StreamIt benchmark suite is best captured by CSDF.

3. **Boolean Data Flow (BDF) :** Boolean Data Flow Buck and Lee (1993) networks extend SDF networks to efficiently implement conditionals. BDF allows nodes in the dataflow graph to have an additional control port. The actor reads the control port and depending on the value of the token present

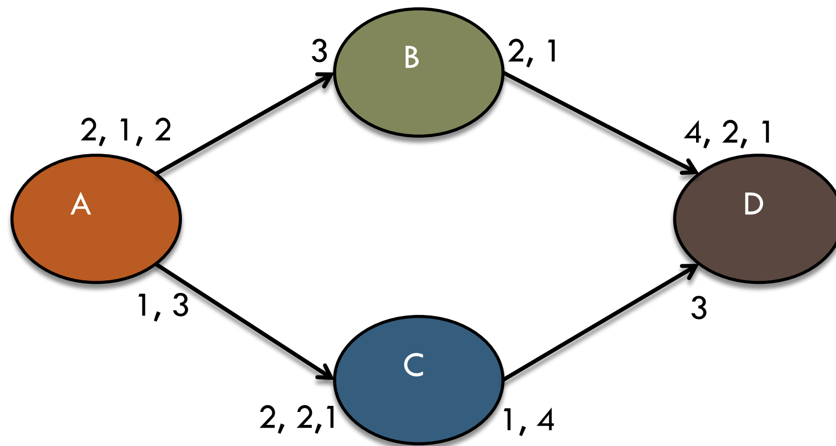


Figure 1.3: Cyclo-static Data Flow

in the control either selectively reads from or writes to one of two input or output channels respectively. The BDF captures the dynamic behavior of certain actors that are not known at compile time. Note that it still allows static scheduling by considering the WCET and worst case memory usage.

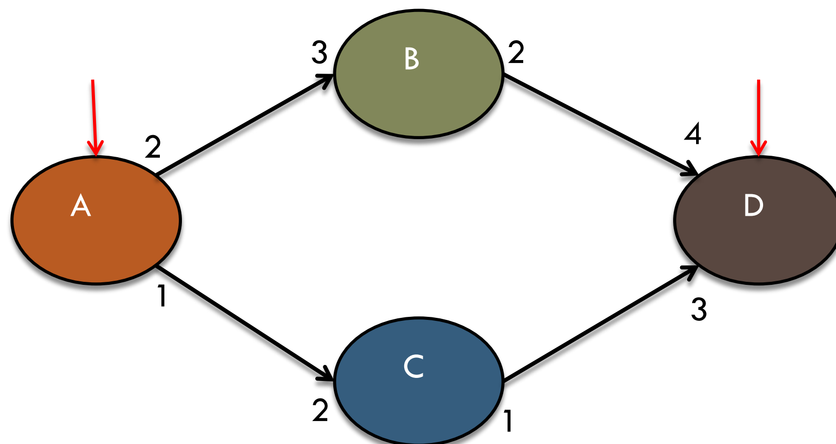


Figure 1.4: Boolean Data Flow

4. **Dynamic Data Flow (DDF):** DDF allows both static and dynamic actors and thus the schedule of actor firings is decided at runtime. Unlike

SDF, CSDF and BDF, deadlock and boundedness are not decidable in DDF. However, note that unlike process networks, an actor can be scheduled as soon as its firing rules are satisfied.

Note: $SDF \subset CSDF \subset BDF \subset DDF$

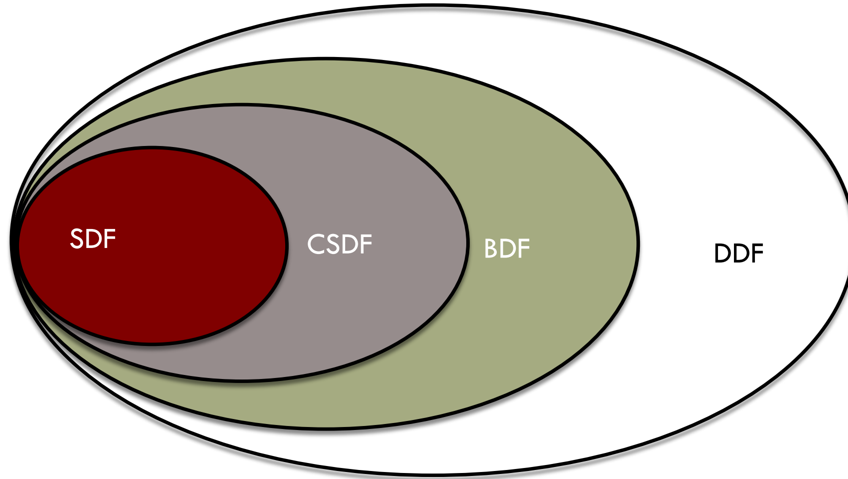


Figure 1.5: Venn Diagram Showing the Classification of Dataflow Models

The StreamWorks architecture is best suited for the SDF and CSDF computational models. However, it doesn't rely on static scheduling and therefore inherits some properties of DDF which are beyond the scope of SDF. Essentially, my architecture captures a special case of deadlock-free and bounded DDF.

1.1 Contributions

My thesis offers different mechanisms to enable energy-efficient stream computing in the context of dataplane co-processors by identifying the key characteristics of stream kernels that can be exploited with appropriate architectural support. Specifically, my thesis addresses the following research questions

1. How to improve the energy-efficiency of instruction hosting and delivery in stream processors by exploiting the high instruction reuse Panda and Chatha (2014) prevalent among stream kernels?

2. What execution model can efficiently exploit the ILP and DLP within stream kernels at runtime Panda and Chatha (2014) and establish a software pipeline to exploit the high TLP exhibited across kernels?

3. How to efficiently distribute data tokens across stream kernels with high fan-in/fan-out communication patterns?

4. How to enable efficient synchronization across stream kernels without the programmer having to schedule the communication patterns?

5. How to schedule large streaming applications without significantly impacting the throughput of the application?

To the best of my knowledge, the Stream-Engine (SE) is the only architecture that supports fine-grain instruction reuse; previous reuse mechanisms follow a coarse-grain reuse approach that limits the parallelism across invocations.

Table 1.1 compares the StreamEngine with other execution models that are used for stream architectures.

In terms of the different levels parallelism exploited, SE exploits Instruction, Data and Thread level parallelism. VLIW architectures exploit ILP and can exploit DLP using explicit unrolling (affecting code size). SIMD architectures can exploit only the inherent DLP of stream kernels and GPUs exploit ILP and DLP within a kernel and the thread-level parallelism. Note that ILP exploited using pipeline is not accounted for in the table.The table reports ILP only for architectures that *execute multiple instructions concurrently*.

In terms of scheduling, VLIW uses a compile-time static schedule (accounting for WCETs) and therefore cannot benefit from run-time behavior that might result in better scheduling opportunities. SIMD, GPU and SE all use dynamic scheduling.

VLIW, SIMD and GPUs use wide register files for intermediate operands and intra-PE data movement. SE distributes the operand store across reservations stations and uses the broadcast token distribution network to move data. Note that since the instructions are locked to RS, the mapping is fixed that enables the circuit-switched token distribution network and therefore is not as power-hungry as conventional broadcast infrastructures.

VLIW architectures use compile-time analysis to schedule instruction and therefore doesn't require synchronization at run-time. SIMD architectures usually lack synchronization across SIMD lanes and need software methods to implement the same. GPUs offer atomic operations (atomicCAS etc.) to allow synchronization. SE uses the broadcast bus for synchronization.

In terms of control-flow, VLIW uses predication to execute both the branch paths and then discard one of the paths when the predicate value is available. Conventional SIMD architectures doesn't support control-flow and GPUs use mask bits to select active lanes. SE transforms control-flow to data flow and executes a branch path only after the predicate is available.

In this work, I present architectural support for energy-efficient stream computing. I envision an entire stream application with multiple kernels mapped across a multi-core dataplane co-processor. I first present the microarchitecture of the StreamEngine (SE), a processing element (PE) for efficiently executing stream kernels. Each kernel of the stream application can be mapped to a StreamEngine. I then present StreamWorks, the multi-core co-processor with architectural support for effi-

	VLIW	**SIMD**	**GPU (SM)**	**SE**
Parallelism	ILP, DLP (w Unrolling)	ILP, DLP TLP	DLP, TLP	ILP, DLP, TLP
Scheduling	Static	Dynamic	Dynamic	Dynamic
Synchronization	NA	Software	Atomic ops	Broadcast bus
Data movement	RF	wide-RF	wide-RF	Broadcast bus
Control-Flow	Predication	No support	Masking	Dataflow predication

Table 1.1: Comparison of SE with VLIW, SIMD and GPU(SM)

cient stream communication. The contributions are presented in further detail in the following chapters.

1. In Chapter 2, we present the SE architecture. The SE features the following innovations:

    - A novel instruction locking mechanism that allows fine-grain instruction reuse and eliminates the overheads associated with instruction delivery.
    - A scalable Context-aware Dataflow Execution (CDE) model that efficiently exploits ILP and DLP within stream kernels.

2. In Chapter 3, StreamWorks, the multi-core dataplane for stream applications is presented. StreamWorks introduces a special communication unit in the form of input channel and push unit pair that enables software FIFO virtualization and supports the split/join functionality that is abundant in stream applications. StreamWorks efficiently extends the CDE across SEs and exploits the TLP in stream applications.

Finally, in Chapter 4 I identify the current limitations of the StreamWorks architecture and propose mechanisms to address the limitations.

Chapter 2

THE STREAMENGINE ARCHITECTURE

2.1 Introduction

In this chapter, the architectural features and execution model of the StreamEngine are presented. The StreamEngine, or SE, is a single-precision floating-point streaming core that forms the basic processing element (PE) of the StreamWorks architecture. Each SE is aimed at implementing a single kernel in the streaming application. The SE implements a context-aware dataflow architecture that is particularly conducive for executing stream kernels. The idea is to exploit the iterative nature and abundant parallelisms exhibited by stream kernels.

The iterative nature of the stream kernels accounts for extremely high instruction locality. Instruction locality coupled with compact code footprint of the kernels allows the SE to host an entire kernel close to the functional units; thus enabling instruction reuse. Instruction reuse amortizes the fetch and decode cost for instructions over multiple kernel iterations and achieves tremendous energy savings.

The block architecture of a SE along with its communication unit is shown in Figure 2.1. Each instruction along with its operands (intermediate results) is hosted within a Reservation Station (RS) close to functional units for efficient instruction and operand delivery. These RSs are distributed across multiple RS Banks (RSBs) for improved energy efficiency. The Stream RSBs host streaming instructions that read from input streams and write into output streams. The Compute RSBs host the core computation instructions meant for the ALUs and MULs. The LD/ST RSBs host memory instructions for loading from and writing into the SPM. Each SE also has

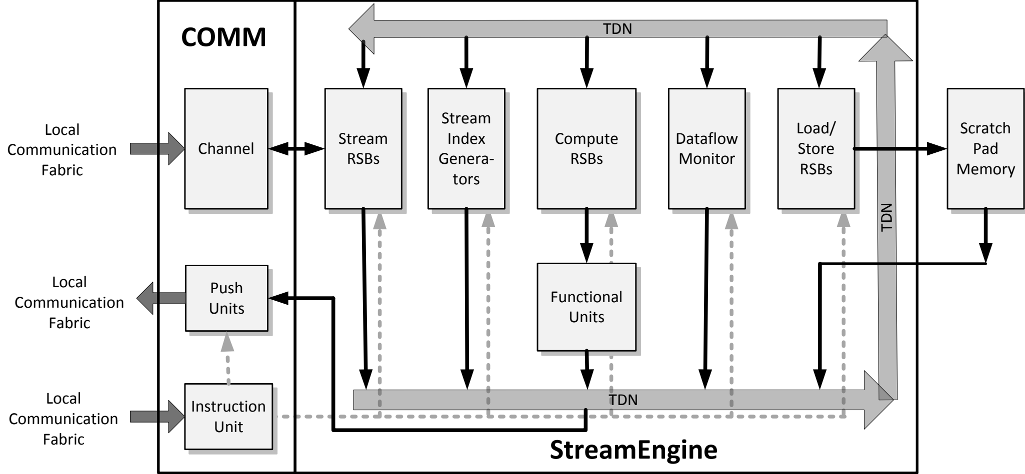


Figure 2.1: StreamEngine with Communication Unit

a bunch of Stream Index Generators (SIGs) that generate the indices and addresses for the stream and LD/ST RSBs respectively. Additionally, the SIGs also decouple consecutive iterations of kernels or loop within kernels by implementing what we call *loop merging* that will presented later in this chapter. All the RSBs are tightly coupled with each other and the functional units via a low-power token distribution network (TDN). The SE also has a central Dataflow Monitor (DFM) that regulates the flow of tokens across the RSBs within a SE.

In Section 2.2, previous work on dataflow architectures and instruction reuse is presented. Section 2.3 presents the execution model and the architectural support that enable the SE to efficiently execute a stream kernel. Finally, in Section 2.7 the evaluation of the SE is presented.

2.2 Related Work

Dataflow architectures have been studied as early as 1970's and capture a very powerful formalism for concurrent execution at different levels of granularity. At instruction level, the abstract dataflow model Kahn (1974) describes a placeholder for data values as *token*, which circulates along the arcs connecting instructions in a

program graph. These arcs are further assumed to be infinite FIFOs. The way in which tokens are managed on arcs has been identified as one of the key factors that impacts not only the organization of a dataflow architecture but also the amount of parallelism that can be exploited in programs Arvind and Culler (1986).

Static and dynamic (Tagged-token) dataflow architectures encode dependencies of an instruction in the form of a list of destination instruction tags. In other words, every producer instruction maintains a list of its consumers. Upon execution, a separate token is created for each of the consumer and added to a common memory pool for distribution. Each token carries its destination instruction tag along with the data value.

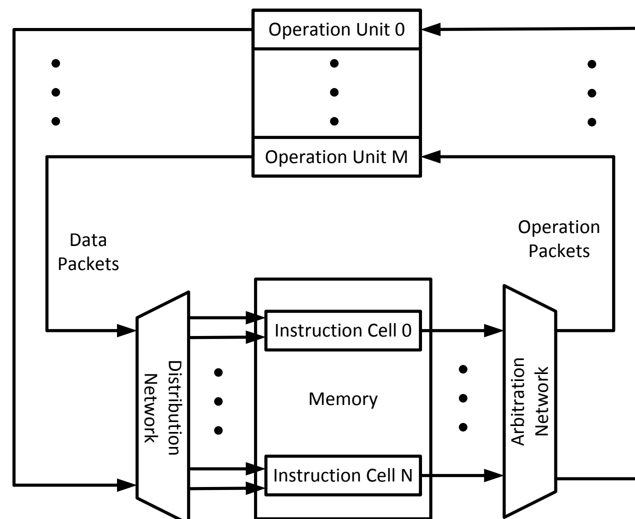


Figure 2.2: Static Dataflow Architecture

Static dataflow restricts the number of tokens per arc to be one; thus an instruction cannot execute unless there is no token present on any output arc of that instruction Dennis and Misunas (1975). Such a restriction allows only a single invocation of any routine to be in flight. When an instruction executes, a token corresponding to each of its destination instructions is formed and added to the distribution network as shown in Figure 2.2. Since only a single invocation of any instruction can be in flight,

the instruction addresses can be static and therefore conventional memory addresses can be used as the destination tags associated with the tokens. Static instruction addresses greatly simplifies the token distribution. The distribution network now becomes equivalent to a memory interface for writing to a conventional directly-addressed memory. However, the one-token-per-arc restriction seriously limits the parallelism in the program Arvind and Culler (1986).

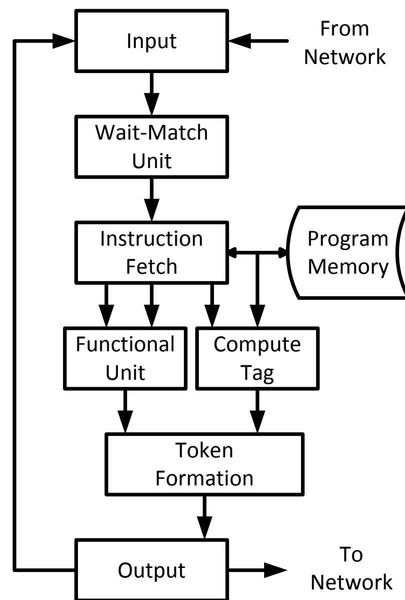


Figure 2.3: Tagged-token Dataflow Architecture

Tagged-token dataflow allows unbounded queues on the arcs without any ordering. It provides dynamic allocation of token storage out of a common pool and allows multiple invocations in flight (as long as space is available in the common pool). However, this complicates the structure of the tag associated with each token. In the MIT Tagged-token dataflow architecture (TTDA)Arvind and Nikhil (1990), the tags have an invocation ID, an iteration ID, code-block and instruction address. Code-block refers to either a routine (acyclic) or a single loop; nested loops are treated as several code blocks. The invocation ID further designates a triple of registers indicating the base address of code-block, the base address of local data (similar to frame pointer)

and mapping information. Upon execution, multiple tokens corresponding to destination instructions are formed and dispatched to a common pool in the Wait-Match (WM) unit as shown in Figure 2.3. The WM unit decides when a consumer instruction is ready to execute by ensuring availability of all its operands belonging to same tag are present in the pool. In TTDA, the WM unit is an associative memory which is slow and expensive. In addition to circulating a complex tag structure, the primary overhead comes from the fact that the memory pool is a centralized unit which needs to perform parallel tag matches for any combination of tokens present in the pool. Another major drawback of dynamic dataflow architectures is that if the WM unit ever gets full the machine will immediately deadlock. This arises from the fact that Tagged-token dataflow execution doesn't ensure any ordering of tokens.

Both static and dynamic dataflow lack scalability in the tag-match and token-supply mechanisms. StreamEngine uses a distributed tag-match mechanism and a circuit switched token-supply network that allow scalability.

Tomasulo Tomasulo (1967) takes a different approach by not storing consumer dependencies but storing the producer dependencies as tags in the instruction RS. When a producer instruction executes, the result and the producer tag is broadcasted in the Common Data Bus (CDB) and all the waiting consumers update their operands within their respective RS. However, Tomasulo has limitations on performance arising from the fact that only one result, tag pair can be broadcasted on the CDB. Scaling the CDB renders it as a power-hungry CAM. We propose a highly scalable low power data bus for token supply. Another major drawback of Tomasulo is that it lacks the notion of a context and thus the only way to allow multiple invocations in flight is by replicating instructions and thereby increasing the RS pressure.

Table 2.1 summarizes the key differences among the conventional dataflow architectures and SE on the following grounds.

Dataflow Architecture	**Context Aware**	**Tag-Match scalability**	**Token supply scalability**	**Deadlock Free**
Static	✘	✘	✘	✓
TTDA	✓	✘	✘	✘
Tomasulo	✘	✓	✘	✓
SE	✓	✓	✓	✓

Table 2.1: Comparison of SE with Conventional Dataflow Architectures

- *Context Awareness* refers to the ability of an architecture to distinguish among multiple sets of operands belonging to different invocations of the same instruction. This is extremely relevant for dataflow architectures as correctness is ensured only when instructions execute with operands belonging to the same invocation context. Static dataflow architectures allow only one data token per edge (between any pair of producer-consumer instruction). In this case, context-awareness is not required as only one set of operands are available at any time. However, restricting the number of tokens to only one seriously limits the parallelism in the program. The architecture proposed by Tomasulo also lacks context-awareness as there is no way to distinguish between operands belonging to 2 different invocations of the same instruction. The only way to allow multiple invocations in flight is to replicate the instruction and rename the registers thus increasing both RS and register pressure. TTDA and SE can distinguish among multiple contexts by using the tag and context field of the tokens respectively.

- *Tag-Match scalability:* Static dataflow and TTDA use a central wait-match unit for matching the tags from producer to consumer instructions before dis-

patching the operands. This central unit is implemented as associative memory and is not scalable for large number of instructions. Tomasulo and SE use distributed tag-match mechanism within RS which is scalable with the number of instructions (RSs).

- *Token supply scalability:* Static dataflow and TTDA again use a central approach for token supply which doesn't scale very well with number of tokens in flight. Tomasulo uses the Common Data Bus (CDB) for token supply. However, since the mapping from producer to consumer RS is not known a priori (the RS contains only a moving window of instructions) only one token is allowed on the CDB at a time. Allowing multiple tokens at the same time renders the CDB to a power-hungry CAM which doesn't scale well with the number of tokens in flight. In case of SE, as the instructions are locked to RS, the mapping for each instruction is known at compile-time. The token distribution network (TDN) uses this mapping information to implement a circuit switched token supply network which scales very well with the number of tokens in flight. In the case of SE, the TDN uses a broadcast bus with a slot corresponding to each token from a Functional unit. At each RS site, the RS scans exactly one slot per operand using a simple switch. Not having to scan all slots for a tag-match allows a scalable low-power token supply network.

- *Deadlock:* The TTDA allows multiple tokens in flight belonging to different invocation contexts and uses tag to distinguish between them. It uses a central wait-match unit for storing tokens and waits until all operands of an instruction belonging to the same context arrive. Only then it dispatches the set of operands to the consumer instruction. Allowing multiple contexts in flight exploits the parallelism in the program. However, since a central token store is used, the

machine can deadlock when the store gets full with only one operand for each instruction available in the store. SE overcomes this problem by maintaining the order of invocation and ensuring a placeholder for each operand at the RS site. Static dataflow and Tomasulo do not deadlock as they allow only a single token in flight.

The comparison suggests that the SE offers a number of distinct features that other dataflow architectures lack.

Levo Uht *et al.* (2003) uses time-tagged Active Stations (AS) organized as a large Execution Window (EW) to realize speculative data-flow execution. Time-tags capture the age of an instruction in the EW, allowing dependent instructions to snarf values from more recent (higher time-tag) instructions in the EW. It is important to note that time-tags only capture the program order to eliminate register renaming. In contrast to SE, Levo doesn't support instruction reuse and uses an expensive CAM-based Register Forwarding Bus (RFB) for token broadcast. Moreover, speculative execution introduces the overhead of re-execution upon mis-speculation.

Instruction reuse at coarser granularity has been proposed by Sankaralingam *et al.* (2003a) and Swanson *et al.* (2003). Sankaralingam Sankaralingam *et al.* (2003a), as part of TRIPS Burger *et al.* (2004), proposed instruction revitalization at block level. The block, which is a loop body, resides on the execution core and decrements a special counter register (CTR) at the end of each iteration. A setup block needs to execute *repeat* instruction that initializes the CTR with a loop bound before the loop starts executing. A block control logic decides the end of iteration and broadcasts a global revitalize signal when CTR > 0. Wavescalar Swanson *et al.* (2003) executes instructions in-place in memory and uses *wave numbers* to identify contexts. Since the atomic unit of execution in Wavescalar is a *wave* (hyper-block), each instruction in the wave executes at most once per wave execution. Both Sankaralingam *et al.* (2003a)

and Swanson *et al.* (2003), in contrast to SE, do not allow fine-grain instruction reuse and therefore limit the parallelism across multiple invocations of loop body and wave respectively. In Section 2.7 we demonstrate how the SE execution model achieves higher IPC by exploiting the parallelism across successive kernel iterations.

More recently, Balfour Balfour *et al.* (2008) proposed the use of instruction registers in the ELM Dally *et al.* (2008) architecture to reduce the cost of instruction delivery. These registers are located close to functional units and capture instruction reuse and locality. However, in contrast to the SE architecture, ELM still uses a register organization and incorporates a fetch-decode pipeline.

The DySER (Dynamically Specializing Execution Resources) architecture Govindaraju *et al.* (2012) attempts to unify parallelism and functionality specialization in a single architecure.

Parallelism specialization refers to techniques that exploit data-level parallelism for example vector processors, Streaming SIMD extensions, Advanced Vector Extensions (AVX) and GPUs. Parallelism specialization uses lanes of hardware resources that are homogeneous and are independent of each other. There is no routing among these lanes. Functionality specialization, on the other hand, uses custom hardware that target the hotspots of applications. It uses task-specific hardware resources and custom routing of operands.

The functionality specialization of DySER can be compared to a SE. DySER achieves functionality specialization by dynamically creating specialized data paths for only frequently executed regions (hotspots). The configuration requires 64 cycles and eliminates per-instruction decode, commit and register read/write overheads. This feature is similar to the instruction locking mechanism of SE.

DySER uses dataflow execution semantics with a fine-grain credit-based mechanism for flow control and context-awareness. Whenever a data token is produced,

the producer routes the token to its consumers and asserts a forward *valid* signal. An instruction executes when all its inputs are valid. Upon consuming the data, the instruction asserts a back *credit* signal to its producers. There are 3 key differences between DySER and SE

1. If an instruction has multiple dependents (consumers) the configured datapath realised by switches must be able to deliver tokens to all the consumers. A high fan-out data dependency might be hard to realize using switches. SE uses broadcast mechanism and thus overcomes the problem of high fan-out dependencies.

2. In DySER, if the operations in the application hot-spot have variable latencies, the operation with the highest latency will be a bottleneck as it cannot send a credit back to its producer thus stalling the datapath. Operand buffers in SE allow variable latency operations to keep firing by providing placeholders for the output of the lower latency operations.

3. Mapping the data path to the reconfigurable DySER fabric depends heavily on the compiler and requires number of transformations as explained in the DySER manuscript. Although the authors claim that such transformations can be implemented in a compiler, the evaluation uses manually optimized benchmarks. SE does not encounter any routing constraints. Moreover, in my evaluation I have implemented the instruction to RS mapping in the runtime framework.

2.3 Context-aware Dataflow Execution and Instruction Locking

StreamEngine exploits the iterative nature and compact code size of stream actors to host the entire actor code within Reservation Stations (RSs) and lock them

across multiple actor iterations if not for the entire program lifetime. We introduce a context-aware dataflow model that allows multiple invocations of an instruction in flight. The tokens carry a context ID tag which in the case of a stream kernel can be as simple as the kernel's iteration number. Context-awareness implies an instruction can execute only upon receiving tokens belonging to the same context. Rather than using a centralized tag-match mechanism similar to tagged-token dataflow, we implement a fast and energy-efficient distributed tag-match mechanism. Each instruction in a RS is responsible for matching the token tags which include the context as well as the instruction dependency. This can be achieved by maintaining the source dependencies of an instruction as opposed to a list of its destination dependencies suggested by static and dynamic dataflow architectures. The source dependencies can be easily restricted to two by enforcing three address generation; thus simplifying the instruction structure. Each RS has an identifier associated with it that becomes the tag of the instruction residing in the RS. The instruction tag can be used to encode the source dependencies.

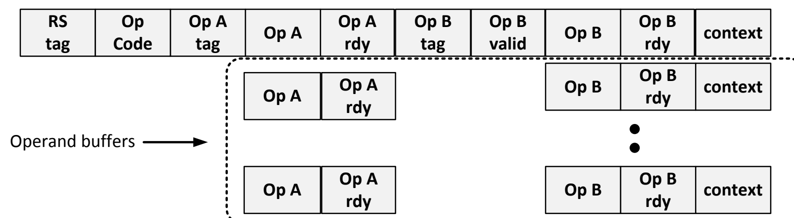


Figure 2.4: Structure of a Basic Reservation Station

The structure of a RS is shown in Figure 2.4. Each instruction has an identifier (Tag) and maintains its source dependencies (Op A tag, Op B tag) along with its context[1]. To begin with, all the instructions belong to the same context 0. We preserve the order of invocation by executing an instruction only upon receiving tokens

[1]The *Op B Tag valid* field in the RS indicates if the instruction depends on a second operand or uses an immediate value.

that belong to its own context. Upon receiving a token, the instruction matches the source tag of the token with its Op tags and the context tag of the token with its own context. If both match then the instruction extracts the data from the token and stores it as the operand for the corresponding source dependency (If Op A/B tag matches the data is stored in Op A/B.) and the ready bit for that operand (Op A/B rdy) is set. If only the source address matches with an Op tag with the token context being higher than the instruction context, the data of the token is stored in the corresponding Op buffer along with its context and the buffer valid bit is set. Thus the SE efficiently distributes not only the tag-match logic but also the token store memory. Preserving the invocation order, avoids the SE from deadlocking when the RS buffers get filled.

An instruction is ready to execute when both the operand ready bits (Op A rdy and Op B rdy) in the RS are set. Upon execution, the instruction updates its context (increments by one) and resets the operand ready bits. It then searches the RS buffer for operands belonging to its new context and accordingly sets the Op rdy bits. If both operands were already present in the buffer the instruction is again ready to fire in the following cycle. If the buffers do not contain both operands, the instruction waits on the TDN to deliver the matching token. Thus an instruction never retires from a Reservation Station.

2.3.1 Token Distribution Network

The locking scheme not only eliminates the overhead of repeatedly fetching the same set of instructions from program memory but also allows static instruction mapping. SE utilizes the static instruction mapping to implement an energy-efficient Token Distribution Network (TDN). The TDN is responsible for delivering tokens

from executing producer instructions to the consumer instructions. A token can be generated at any of the following sites:

- *A Functional Unit* A functional unit executes an instruction from a RS and forms a token using the result and the tag and context of the instruction. More than one instructions in a RSB-Functional Unit pair can be ready at the same cycle but the Functional Unit is arbitrated and exactly one RS fires its operands producing one token per compute RSB.

- *A Stream Index Generator* A Stream Index Generator (SIG) is always one of the root nodes in a dataflow graph. It fires the ISIG instruction which generates stream indices. The token is formed by using the tag of the ISIG instruction and its context along with the generated stream index.

- *A Stream RSB* A Stream RSB hosts instructions that read from the input stream. The token is formed by using the tag and context of the read instruction and the data item that was read from the input stream.

- *A Load RSB* A Load RSB hosts LD instructions that load data items from the SPM. It can form a token using the tag and context of the Load inxtruction along with the data item read from the SPM.

The TDN is partitioned into slots corresponding to each token generation site. Since the instruction to RS mapping is static for a stream kernel, the TDN implements a circuit-switched network to route the entering tokens to their respective destinations. For example, consider a MUL instruction that is mapped to a RS X in the RSB paired with one of the multipliers (say MUL0). Let one of the operands of the MUL comes from an ADD instruction that is mapped to a RS in the RSB corresponding to ALU0. The TDN routes the tokens generated at ALU0 site to RS

X. It is important to note that tokens generated at site ALU0 are a result of execution of any of the ALU instructions mapped to the RSB corresponding to ALU0. Thus all such tokens will be routed to RS X. The tag matching logic at RS X will selectively store only the relevant tokens. Of course, tokens from a generation site are routed to multiple RSs. Thus the dependency fan-out is captured by the TDN and not by maintaining a list of consumer instructions within a RS.

2.3.2 Conditional Dataflow

Dataflow execution demands the conversion of control dependencies to data dependencies. Static dataflow uses special T-gate, F-gate and merge operators to direct the flow of data tokens based on control tokens which can either be true or false Dennis and Misunas (1975). A T-gate (F-gate) moves an input data token to its output arc only when the control token received at the control input has a true (false) value. Otherwise the input data token is simply absorbed. A merge operator has three inputs corresponding to true, false and control. The token on the input arc corresponding to the value of control input is passed to the output arc.

Tagged-token dataflow uses a merge operator similar to that of static dataflow. However, it uses a special switch operator that captures the functionality of both T-gate and F-gate Arvind and Nikhil (1990). The switch operator has two outputs corresponding to the true and false path. Depending on the control input, it steers the input data token into either of the outputs.

The conversion of a control dependence (if-then-else construct) to data dependence using gate, switch and merge operators is illustrated in Figure 2.5.

As can be observed from Figure 2.5, one of the major drawbacks of using special control operators is that for every live variable being used in a control path, the compiler needs to introduce a switch operator or a T-gate, F-gate operator pair. Use of

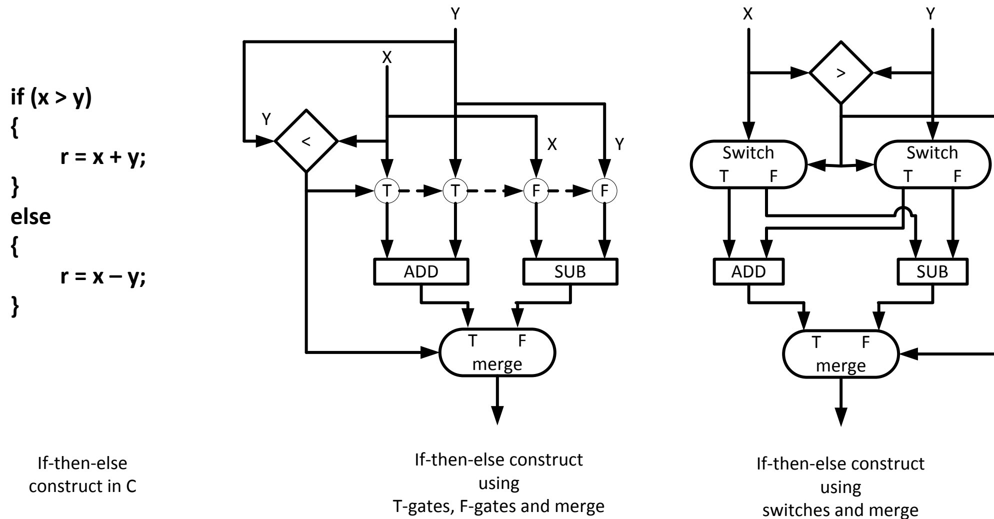


Figure 2.5: Conversion of If-then-else Construct

switches or T-gate, F-gate pairs for every input to a control path results in significant overhead. TTDA calls this set of input tokens as a wave of inputs. Assuming, that the conditionals are well-behaved, a single wave of tokens will eventually arrive at the data input of appropriate side of the merge. Since dynamic dataflow allows multiple invocations in flight, special care needs to be taken for loops in order to distinguish between two iterations. TTDA employs the D and D_reset operators to change the context of a input token and reset it back to the context of the loop. This is a serious overhead which is incurred in every iteration of the loop. Also, each nested loop body requires a distinct code-block ID.

Clearly, one cannot apply these classical dataflow conversions to context-aware dataflow. Every instruction in a context-aware dataflow expects a token belonging to its context. However, the above conversions steer the tokens to a single branch path which is the *taken* (branch taken) path. The *not-taken* path will never receive a token belonging to its present context and therefore will never execute even when it becomes the *taken* path for following kernel iterations.

In order to efficiently transform control dependencies to data dependencies for context-aware dataflow we extend the RS with the following:

- *Ctrl Tag*: The Ctrl tag or Control tag is the tag (RS identifier) of the instruction that evaluates which branch path is taken.
- *Op Ctrl*: The Op Ctrl represents the control operand which indicates whether the path to which the RS belongs was taken or not. It is the result of executing the instruction at Ctrl Tag. Possible values can be 00 (branch not taken), 01 (branch taken) or 11 (branch skip, discussed later in the section)
- *Op Ctrl rdy*: This bit indicates whether the control operand is available for the present context.
- *Ctrl Valid*: This is a single bit which indicates whether the instruction in ths RS belongs to a branch path or is independent of any predicate.
- *Path*: The path bit indicates the branch path to which the instruction belongs.

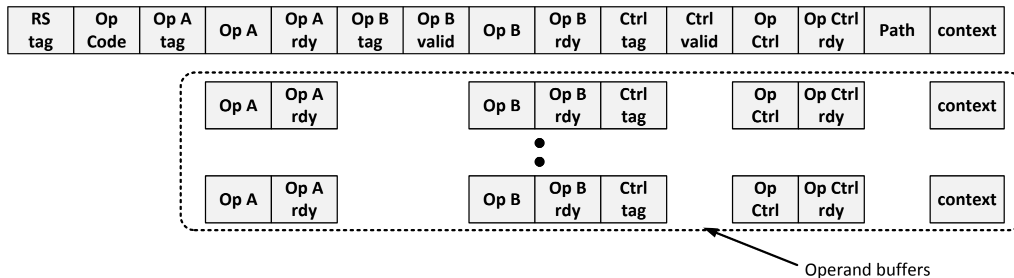
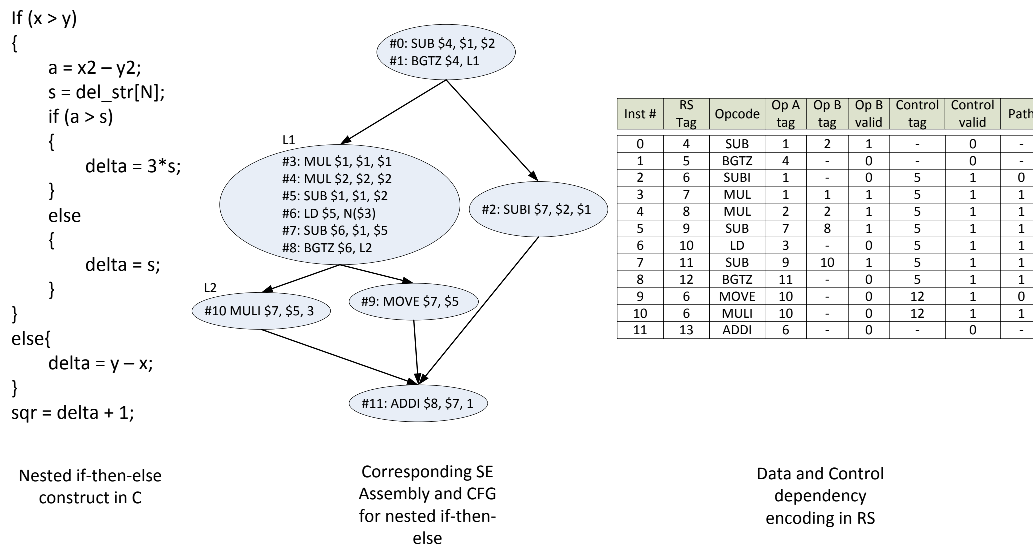


Figure 2.6: Structure of the Extended Reservation Station

Figure 2.6 shows the structure of the extended RS. The RS Op buffers are also extended to allow multiple control tokens in flight. If the *Control Tag valid* bit is set, then an instruction can execute only when all the operand valid bits (Op A rdy, Op B rdy, Op Ctrl rdy) are set. If the *Op ctrl* value matches with the 0-prepended *Path*

value, then the instruction follows the same procedure for firing as in the case without the control extensions. It dispatches the operands to the functional unit, resets the operand valid bits (including control op valid) and updates its context. However, if the *Op ctrl* value is not the same as *Path* then the instruction doesn't dispatch any operands but simply resets all the operand valid bits and updates the context. Thus even when a branch path is not taken, the instructions still update their context thus retaining the context-awareness of the stream kernel.



Inst #	RS Tag	Opcode	Op A tag	Op B tag	Op B valid	Control tag	Control valid	Path
0	4	SUB	1	2	1	-	0	-
1	5	BGTZ	4	-	0	-	0	-
2	6	SUBI	1	-	0	5	1	0
3	7	MUL	1	1	1	5	1	1
4	8	MUL	2	2	1	5	1	1
5	9	SUB	7	8	1	5	1	1
6	10	LD	3	-	0	5	1	1
7	11	SUB	9	10	1	5	1	1
8	12	BGTZ	11	-	0	5	1	1
9	6	MOVE	10	-	0	12	1	0
10	6	MULI	10	-	0	12	1	1
11	13	ADDI	6	-	0	-	0	-

Figure 2.7: Nested Conditional Example

Nested conditionals can easily be handled by using the introduced control extensions. However, the firing mechanism needs a slight modification as we discuss below. Figure 2.7 illustrates the use of control extensions for a nested if-then-else construct. The *Path* field is set for the true path. The operand valid bits and operand values are not marked as their states keep changing at runtime. Figure 2.7 shows the C-code along with the equivalent assembly. For the assembly code, lets assume that the values of variables x and y are available in the registers $1 and $2 respectively. Also, the base address of the array *del_str* is available in $3. The different fields of the RSs corresponding to the code are also shown in Figure 2.7. The *Inst#* field is added just

for the discussion here. For the RS configurations, it is assumed that the producers of x, y and base address of *del_str* array have 1, 2 and 3 as their RS Tags respectively. Depending on the branch path taken during an invocation, the *ADDI* instruction just outside the if-then-else construct can receive its token from either of the three instructions (*Inst# 2, 9* and *10*). However, all the three instructions that produce the operand for the *ADDI* have the same RS Tag (=6). In a single invocation, exactly one RS with Tag 6 will execute to produce the operand for *ADDI*. The control token for the instructions in body of the inner if-then-else (*Inst# 9* and *10*) is produced by the *BLTZ* with Tag 12 which in turn waits for its control token to be produced by *BLTZ* with Tag 5. Thus nested control structures are supported by cascading the control dependencies. Now consider an invocation (say with context ID n) where the *BLTZ* with Tag 5 evaluates to *false*. Instructions from *Inst# 2* through *Inst# 8* will receive their control operands. *Inst# 2* with Tag 6 will execute and update its context to $n + 1$. Instructions from *Inst# 3* through *Inst# 8* will not execute but just update their contexts. *Inst# 9* and *10* will never receive thier control token for context n and thus will never be able to move their contexts forward. We solve the above problem by introducing an opcode *BR_SKIP*. The ALU on evaluating a branch instruction normally produces either a 0 or 1 (binary). The *BR_SKIP* instruction doesn't take any operands and always evaluates to 11 (binary). Any nested branch instruction that lies in the not-taken path of its outer branch construct sends the *BR_SKIP* opcode to the ALU. The nested instructions will now receive their control token (with the proper tag) and the value 11 which will never match any of their 0-prepended path values and therefore all such nested instructions will simply update their context.

When the branch paths are unbalanced as in our example, the instruction at the join of branch paths (*Inst# 11* in the example) will receive its input tokens out of

order. However, the operand buffers in the RS already take care of preserving the order of the tokens as discussed in the previous subsection. Therefore no additional architectural support is required to deal with unbalanced branch paths.

As we saw in the example, across contexts, an instruction at the join of branch paths can receive its data tokens from multiple RSs (which have the same tag). If these RSs are mapped to different RSBs, the TDN now has to route the results of more than one slot to the operand of the instruction at the join. This cannot be achieved using a static network. We specify mapping same tag instructions to the same RSB as a constraint during mapping instructions to RSs. If the same tag instructions have opcodes that require different functional units (e.g. *SUBI, MULI* in our example), we introduce a *MOVE* instruction per such pair to migrate the result of one of the instructions to the RSB where the other resides. Both the ALU and the Multiplier implement the *MOVE* instruction to allow migration of any instruction including stream instructions, Ld/St instructions and index generation instructions to either a ALU RSB or a Multiplier RSB.

2.3.3 Dataflow Monitor and Token regulation

One direct consequence of locking instructions and repeatedly executing them to operate on a continuous stream of input tokens is *Token accumulation.* Token accumulation results from the mismatch in the rate at which data or control operands are received at a RS. In Figure 2.8 Arvind and Culler (1986), for example, the mismatch is evident from the datapath. Lets assume that the multiplier has a two-stage pipeline and the ALU is not pipelined. Assuming that the *a,b* and *c* operands (belonging to a new context) are available every cycle, by the time the divide instruction receives its operand from the preceding multiply, five tokens corresponding to *c* would have accumulated at the RS hosting the divide instruction. Unbalanced branch paths, as

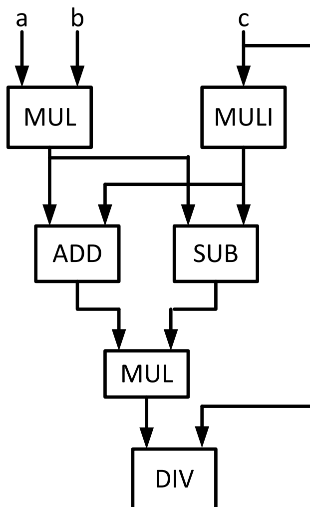


Figure 2.8: Example of a Dataflow Graph that Causes Token Accumulation

in Figure 2.8. could be another such source of token accumulation. For example if the longer path is taken for the first invocation (context ID 0) and then the shorter path is taken for the next k invocations. These k tokens will get accumulated in the operand buffer of the instruction at the join of the paths since it is yet to receive the token for context 0. Eventually, the operand buffers at the RS will be full and consequent tokens of higher contexts will be dropped.

We use a simple yet elegant back-pressure mechanism to stall an instruction in order to bound the forwarding of its context. The Dataflow Monitor (DFM) maintains all the data and control dependencies among the RSs. Once the operand buffer corresponding to a certain operand is on the verge of a spill, the RS signals the DFM to stall the producer of the corresponding operand. A tricky case is when an instruction at the join of branch paths, wants to selectively stall a producer belonging to a particular branch path. The only dependency encoding available in the system is in terms of RS Tag. However all the producers of this instruction have the same tag. If the DFM stalls all the producers it will create a deadlock because the instruction requesting the stall is awaiting a token belonging to its context. Since all the producers

are stalled, this token will never arrive. This can be resolved by monitoring the context of the RSs. The rule for the DFM to stall a RS is as follows : "If RS X requests to stall RSs with Tag A, stall only the RSs with Tag A that have context ID higher than that of RS X."

2.3.4 Stream Index Generator, Iteration Decoupling and Loop Merging

Consider the execution of a loop in any dataflow machine. The invocation of consequent iterations of a loop depends on the evaluation of a predicate that in turn depends on the instrucion that updates the trip count of the loop (e.g. $i++$, $i = i - 1$ etc.). The evaluation of trip-count implicitly infers a data dependency between consecutive loop iterations. At first, one may tend to overlook the effects of this inter-iteration data dependency as the datapath for the trip count update (across iterations) is very simple. However, in modern processors, where each functional unit is heavily pipelined, this dependency on previous iteration may seriously limit the parallelism by restricting the number of loop iterations in flight[2]. This is because the instruction that updates the trip count cannot execute before it receives the result of its own previous invocation corresponding to the previous loop iteration. For a N-stage pipelined ALU that is used to execute the trip count update, the pipeline will never reach a steady-state unless the loop is unrolled $N - 1$ times resulting in N copies of the loop body.

Streaming workloads (multimedia for example) mainly consists of loops that have bounded trip counts which is known at compile time. StreamEngine exploits this characteristic of stream actors to implement what we call *Iteration decoupling* using a Stream Index Generator (SIG). Iteration decoupling allows multiple loop iterations in flight by eliminating the inter-iteration dependency due to trip count update without

[2]Unless, power-hungry branch predictors are used

unrolling. A SIG hosts a special instruction *ISIG* that has a format as follows $ISIG$ $\$R_dst$, *base*, *stride*, *length*. The *base* is the initial trip count of a loop, *stride* is the increment and *length* is the target trip count. The SIG is a fast adder that can produce a new index every clock cycle to R_dst. The ISIG instruction doesn't wait on any instruction to forward the trip count of a loop. Of course, the assumption is that the loop has no other inter-iteration data dependencies The SIG unit also features special offset registers for *peek* operations.

2.4 SE Program Execution Example

In this section, a sample kernel execution on the SE architecture is presented. For the ease of explaining the CDE model, we consider an integer benchmark and a simpler instance of SE with the configuration listed in Table 2.2. Let the ALU instances be ALU0 and ALU1 and the MUL instance be MUL0.

Listing 2.1 shows the SE assembly for the sample kernel. Lines 4 and 5 in the SE assembly correspond to the one time configuration of the SIGs and implement a loop with a trip count of 64 (0 through 127 with a stride of 2). All the other instructions get mapped to RS; instructions 8 and 9 get mapped to stream RSs, 11-14 to compute RSs and 16 to a St RS. Note that the instruction *rd $dst, $src* is similar to a load instruction the only difference being the source of the load is the channel and not the SPM.

As evident from the SE assembly, the sample kernel calculates the difference of squares of two consecutive input tokens and stores the value in the SPM. The *isig* instructions are used to generate the indices of the channel from which the kernel reads the input tokens. The store address is determined by the sum of the tokens followed by a bitwise *and* operation with 63 (effectively a modulus operation with

	Configuration
ALUs	2
MULs	1
Compute RSBs	3
RS per compute RSB	2
Stream RSBs	2
RS per stream RSB	2
Ld/St RSBs	1
RS per Ld/St RSB	2
Operand buffers per RS	1
SIGs	2

Table 2.2: StreamEngine Configuration

RSB	RS Tag	Op	Op A Tag	Op A	Op A rdy	Op B Tag	Op B valid	Op B	Op B rdy	Ctrl Tag
STR RSB 0	3	rd	1	?	0	-	0	-	1	-
STR RSB 1	4	rd	2	?	0	-	0	-	1	-
ALU0 RSB	5	add	3	?	0	4	1	?	0	-
ALU1 RSB	6	sub	3	?	0	4	1	?	0	-
ALU0 RSB	7	andi	5	?	0	-	0	63	1	-
MUL0 RSB	8	mul	5	?	0	6	1	?	0	-
ST RSB0	9	st	8	?	0	7	1	?	0	-

Table 2.3: Configuration of Stream (s-RSB), Compute (c-RSB) and Memory (m-RSB) RSBs

64). A single invocation of this kernel consumes 128 tokens and produces 64 tokens as the result in the SPM.

RSB	Ctrl valid	Op Ctrl	Op Ctrl rdy	Pred- -icate	Context
STR RSB 0	0	-	1	-	0
STR RSB 1	0	-	1	-	0
ALU0 RSB	0	-	1	-	0
ALU1 RSB	0	-	1	-	0
ALU0 RSB	0	-	1	-	0
MUL0 RSB	0	-	1	-	0
ST RSB0	0	-	1	-	0

Table 2.4: Configuration of Stream (s-RSB), Compute (c-RSB) and Memory (m-RSB) RSBs

Listing 2.1: SE Assembly for Sample Kernel

```
 .text
main:
# Configure SIGs
  isig  $i0, 0, 2, 127, 0   #SIG0
  isig  $i1, 1, 2, 127, 0   #SIG1
# read 2 tokens from channel
# indices generated by SIGs
  rd    $r4, $i0
  rd    $r5, $i1
# Compute
  add   $r6, $r4, $r5
  sub   $r7, $r4, $r5
  andi  $r8, $r6, 63    # r8 = r6 % 64
  mul   $r9, $r6, $r7
# Store into SPM
  st    $r9, [$r8]
```

Each SE is configured by the control-plane processor. During configuration phase, the registers corresponding to base, stride, length and offset of the SIGs are set by

the *isig* instruction. The format of the *isig* instruction is *isig* $\$dst$, $\langle$*base value*$\rangle$, $\langle$*stride value*$\rangle$, $\langle$*length value*$\rangle$, $\langle$*offset value*$\rangle$. During execution, each SIG broadcasts an index starting from its base and incrementing by the stride value till it encounters the length value. It then adds the offset value both to the base and to the length and starts generating indices starting from the new base till it encounters the new length. The destination registers of SIGs ($\$i0$,$\$i1$ etc.) are strictly for encoding dependencies of other instructions that are dependent on the output of SIGs (For example the *rd* instructions in this case). Table 2.5 lists the configuration of the SIGS for the kernel example.

Figure 2.9 shows the dataflow graph (DFG) of the sample kernel. The nodes of the DFG are labeled with the instruction numbers corresponding to Listing 2.1.

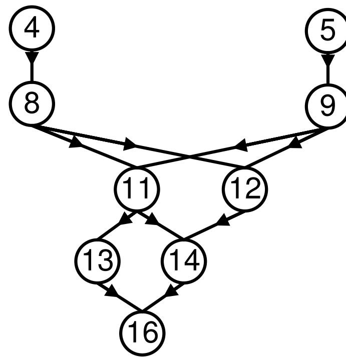


Figure 2.9: Dataflow Graph (DFG) of the Example Kernel

SIG Num	RS Tag	Base Reg	Stride Reg	Length Reg	Offset Reg
0	1	0	2	127	0
1	2	1	2	127	0

Table 2.5: Configuration of Stream Index Generators

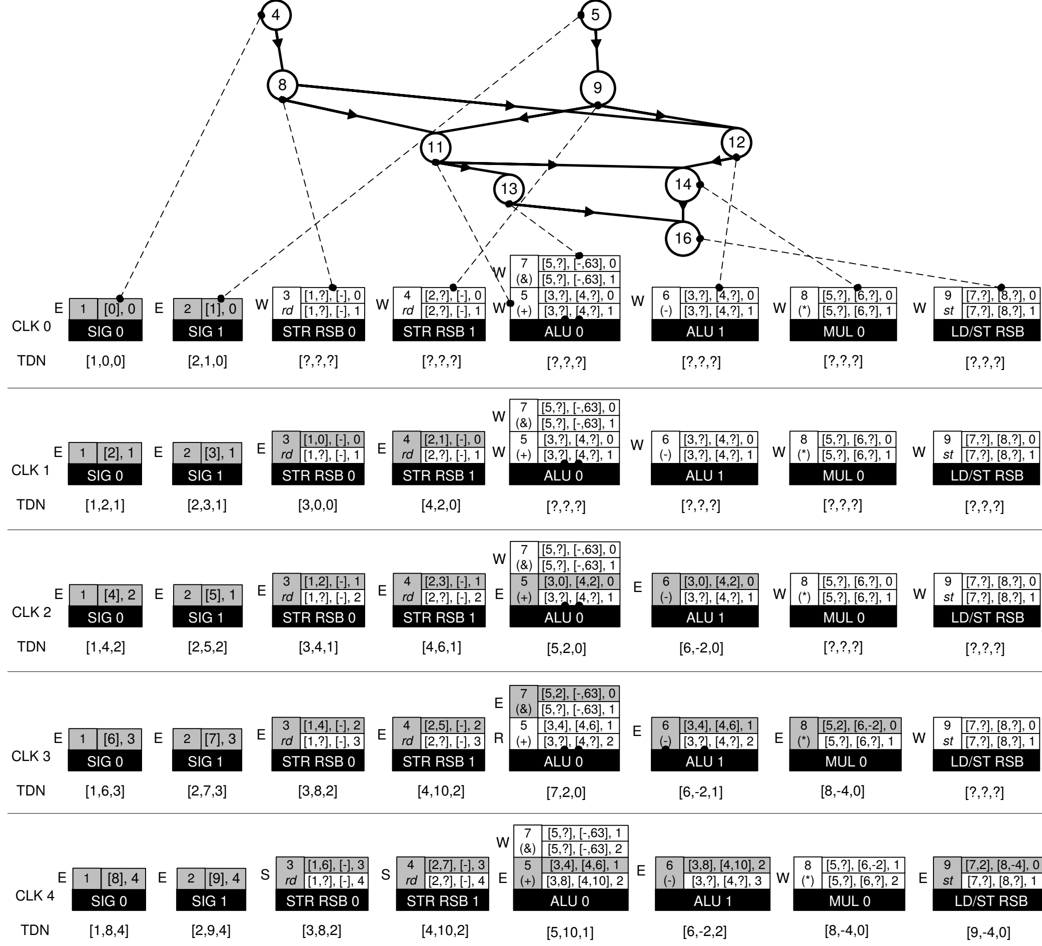


Figure 2.10: DFG Mapping and RS/SIG/TDN State during Execution

The RSBs are also configured during the configuration phase. The RS tags are assigned and the data and control dependencies are encoded in the *Op A Tag*, *Op B Tag* and *Ctrl Tag* fields. The configurations of the stream, compute and memory RSBs are shown in Table 2.3 and Table 2.4. The RSB (first column) indicates the static mapping of instructions to RSBs. STR RSBs represents the RSBs associated with the read ports of the channel. Since the channel has 2 read ports, each of the port is associated with a STR RSB. As Table 2.3 and Table 2.4 suggests instructions 11 (RS Tag 5) and 13 (RS Tag 7) are both mapped to ALU0 RSB and will therefore contend for ALU0 when both are ready to execute. The RSB entries that are unknown during configuration phase are represented by '?'.

A RS is ready to fire when all the *rdy* bits (Op A rdy, Op B rdy and Ctrl Op rdy) are set and there is no stall arising from resource contention or from a dependent RS. Each RS increases its context and resets its rdy bits upon firing. Note that in Table 2.3 and Table 2.4, the instructions where the Op B and Ctrl are not valid (*Op B valid* and *Ctrl valid* not set), Op B rdy and Op Ctrl rdy are always set. These RS never reset the respective bits upon firing. Also note that the all the RS are initialized with the same context 0.

We now show the activity in the TDN and each of the RSs and SIGs during the execution phase. Figure 2.10 shows the mapping of the DFG to the RSBs and the state of all the RSs, SIGs and the TDN during the execution phase. The contents of each of the RSs and their respective operand buffers are shown alongside the RS Tag and the operation it hosts. The contents are represented as *[Op A Tag, Op A], [Op B Tag, Op B], Context*. As explained in Section 2.3, the operand buffers are used to store operands belonging to higher contexts. Note that the context of an operand buffer is always one higher than its respective RS. If a RS is waiting on an operand, the corresponding placeholder is represented as '?'. For RSs hosting instructions that require only a single operand, the placeholder corresponding to the second operand is represented by '-'. Note that since instruction 13 uses the value 63 as the second operand, the placeholder for the second operand will always be occupied by 63. For each SIG, its index and context are represented as *[index], Context* beside the Tag of the SIG.

The activity at each RS is indicated by either **W, E, R** or **S**. **W** denotes the RS is waiting on one or more operands. **E** denotes that the RS is firing thus executing the instruction it hosts. **R** denotes the RS is ready to fire (execute) but is stalled due to a structural hazard (in our example when instructions mapped to same RSB are

contending for the functional unit). **S** denotes the RS is stalled by one or more of its consumers. The executing RSs are shaded in the figure.

For each token placed on the TDN, we follow the triplet convention of [Rs Tag, Value, Context]. Also, assume that the value of tokens in channel at address x is $2x$ ($channel[x] = 2x$).

As can be seen from Figure 2.10, SE can achieve a compute IPC of upto 3 (Clk cycle 3) with an average compute IPC of 2.5 across successive kernel iterations when compared to the inherent IPC of 2 for a single kernel invocation.

Clk cycle 3 demonstrates the structural hazard due to mapping instructions 11 and 13 to the same RSB. As the figure illustrates, RS Tag 5 is ready to fire but cannot fire as ALU 0 is busy in executing instruction in RS Tag 7. Thus, RS Tag 5 signals all its producers (RS Tag 3 and RS Tag 4) to stall in the following cycle. In the following cycle, both RS Tag 3 and RS Tag 4 stall. However, the results broadcasted by the producer RSs in the previous cycle are now stored in the operand buffer of RS Tag 5 with context 2.

2.5 Executing Nested Conditionals

SE uses predication to execute conditional statements as described in Section 2.3.2 of the proposal. Note that as opposed to conventional predication (for e.g. in VLIW architectures), SE does not execute both the branch paths only to reject the not-taken path once the predicate is available. Instead, the correct branch path is executed only upon the availability of the predicate. However, since SE maintains the context awareness, the not-taken path must advance its context without executing. The context-awareness, in particular, can become tricky when executing nested conditional statements.

I will demonstrate, using the example shown in Listing 2.2, how nested conditionals are handled by the SE . The sample kernel in the listing takes 2 streams of length 16 each, it then compares the tokens from the two streams and outputs the greater of the two. In the case when the 2 tokens are equal, the kernel outputs the sum of the tokens. Let us assume that the 2 input streams are interleaved in the channel (stream 0 occupies channel indices 0, 2, 4 and so on and stream 1 occupies channel indices 1, 3, 5 and so on). Listing 2.3 shows the equivalent SE assembly and Figure 2.11 captures the dataflow graph of the kernel. The numbers on the nodes represent the instruction number in the SE assembly. The red edges represent control dependencies.

```c
void kernel (int* in_stream, int* out_stream){
    int i = 0;         // loop index

    for (i = 0; i < 32 ; i+=2)
    {
        if (in_stream[i] >= in_stream[i+1]) {
          if (in_stream[i] == in_stream[i+1]) {
            out_stream[i/2] = in_stream[i] + in_stream[i+1];
          }
          else {
            out_stream[i/2] = in_stream[i];
          }
        }
        else {
          out_stream[i/2] = in_stream[i+1];
        }
    }
}
```

Listing 2.2: C Code for Sample Kernel with Nested Conditional

```
 .text
main:
# Configure SIGs
    isig    $i0, 0, 2, 31, 0    #SIG0
    isig    $i1, 1, 2, 31, 0    #SIG1
# read the tokens from the channel
    rd $r2, $i0
    rd $r3, $i1
# branch
    bge $r2, $r3, L1
    move $r4, $r3
    b L2
L1: beq $r2, $r3, L3
    move $r4, $r2
    b L2
L3: add $r4, $r2, $r3
# Push the result
L2: push $r4
```

Listing 2.3: SE Assembly for Sample Kernel with Nested Conditional

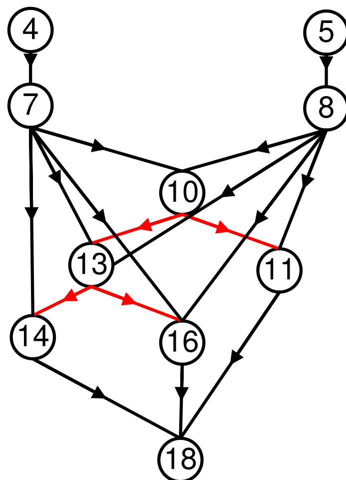


Figure 2.11: Dataflow Graph of the Sample Kernel with Nested Conditional.

The configuration of the SIGS and RSBs is shown in Table 2.6 and Table 2.8 respectively. Note that in dataflow execution, the unconditional branches (line 12 and 15) do not need to be hosted by any RS. Also, note that the two *MOVE* instructions and the *ADD* instruction that produce the final result ($\$r4$), belong to mutually exclusive control paths and have the same RS Tag. This is because at the convergence point of branch paths, the dependent instruction will have to maintain only one tag dependency (as opposed to having maintain list of all instructions that can produce its token).

SIG Num	RS Tag	Base Reg	Stride Reg	Length Reg	Offset Reg
0	1	0	2	31	0
1	2	1	2	31	0

Table 2.6: Configuration of Stream Index Generators

RSB	RS Tag	Op	Op A Tag	Op A	Op A rdy	Op B Tag	Op B valid	Op B	Op B rdy	Ctrl Tag
STR RSB 0	3	rd	1	?	0	-	0	-	1	-
STR RSB 1	4	rd	2	?	0	-	0	-	1	-
ALU0 RSB	5	bge	3	?	0	4	1	?	0	-
ALU1 RSB	7	move	4	?	0	-	0	-	1	5
ALU2 RSB	6	beq	3	?	0	4	1	?	0	5
ALU0 RSB	7	move	3	?	0	-	0	-	1	6
ALU1 RSB	7	add	3	?	0	4	1	?	0	6

Table 2.7: Configuration of Stream (s-RSB), Compute (c-RSB) and Memory (m-RSB) RSBs

Figure 2.12 shows the mapping of the DFG to the RSBs and the state of all the RSs, SIGs and the TDN during the execution phase. The contents of each of the RSs and their respective operand buffers are shown alongside the RS Tag and the

RSB	Ctrl valid	Op Ctrl	Op Ctrl rdy	Pred- -icate	Context
STR RSB 0	0	-	1	-	0
STR RSB 1	0	-	1	-	0
ALU0 RSB	0	-	1	-	0
ALU1 RSB	1	?	0	0	0
ALU2 RSB	1	?	0	1	0
ALU0 RSB	1	?	0	0	0
ALU1 RSB	1	?	0	1	0

Table 2.8: Configuration of Stream (s-RSB), Compute (c-RSB) and Memory (m-RSB) RSBs

operation it hosts. The contents are represented as *[Op A Tag, Op A], [Op B Tag, Op B], Context*. For simplicity, I have shown only one operand buffer per RS. Note that the context of an operand buffer is always one higher than its respective RS. If a RS is waiting on an operand, the corresponding placeholder is represented as '?'. For RSs hosting instructions that require only a single operand, the placeholder corresponding to the second operand is represented by '-'. For each SIG, its index and context are represented as *[index], Context* beside the Tag of the SIG.

The activity at each RS is indicated by either **W, E, I** or **S**. **W** denotes the RS is waiting on one or more operands. **E** denotes that the RS is firing thus executing the instruction it hosts. **I** denotes the RS is ready to fire (execute) but it belongs to the not-taken branch path and therefore just advances its context without executing. **S** denotes the RS is stalled by one or more of its consumers. The ready to fire RSs are shaded in the figure.

Figure 2.12 demonstrates 2 iterations of the sample kernel that take different branch paths. In the first iteration, the data token from stream 0 is less than that of stream 1 and therefore only the *MOVE* instruction corresponding to line 11 is executed. The *BGE* instruction evaluates to false (0) in CLK 2. Once the value of

the control evaluation is available in CLK 3. There are 2 key operations that take place

1. The *MOVE* instruction in RS 0 of ALU 1 which corresponds to instruction 11, fires because the result from *BGE* matches its predicate

2. The *BEQ* instruction in RS 0 of ALU 2 (corresponding to instruction 13) fires with a special result token "11"

Typically, since the *BEQ* instruction belongs to the not-taken path, it should have just advanced its context without executing. However, since it is a branch instruction itself, there are instructions that have the *BEQ* (instruction# 13) as their control dependency (instruction# 14 and instruction# 16 in this case.) If the branch instruction in the not-taken path doesn't execute. The two dependent instructions will never receive their control operand corresponding to iteration 0 and can never advance to iteration 1.

SE uses a special opcode *BR_SKIP* to handle the nested conditionals. *BR_SKIP* doesn't take any operands and always evaluates to binary "11". The control dependent instructions receive "11" as their control operand which is compared against 0-prepended predicate of the dependent instructions. Since "11" is a mismatch for both predicate 0 and 1 ("00" and "01" after 0-prepending) both the branch paths of the *BEQ* instruction will advance their context without executing. This mechanism is demonstrated in CLK 4.

Note that by CLK 4, the result of *BGE* for iteration 1 is already available and since it evaluates to *true* (1), the *MOVE* corresponding to instruction# 11 in RS 0 of ALU 1 must also advance its context (in state *I*) in CLK 4 as shown in Figure 2.12. For iteration 1, the *BEQ* is in the branch-taken path and evaluates to true eventually allowing the *ADD* operation to fire in CLK 5.

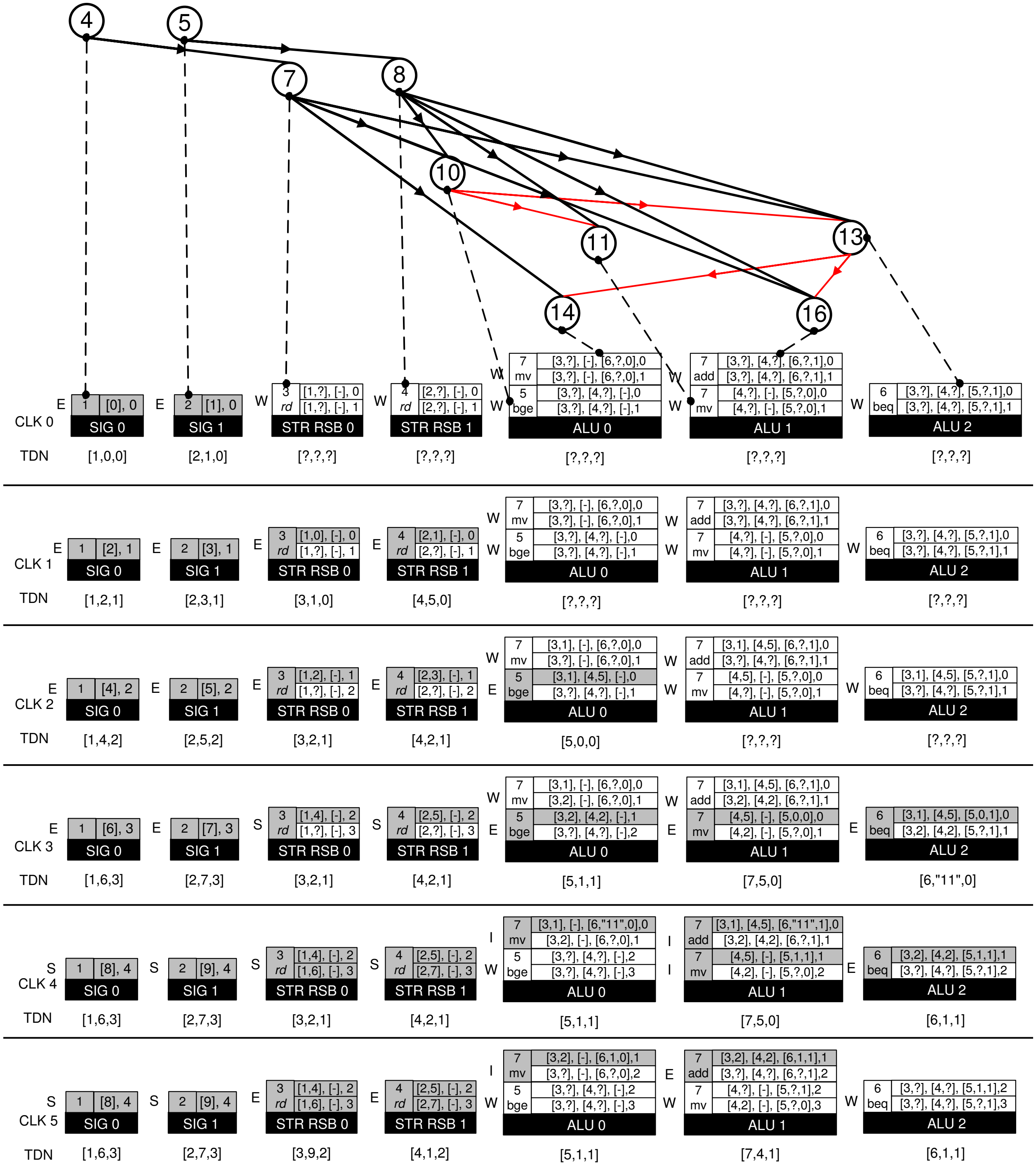


Figure 2.12: DFG Mapping and RS/SIG/TDN State During Execution

2.6 Executing Nested Loops

One of the key characteristics of stream kernels is that loops have known trip counts and therefore allows several compile-time optimizations. The SE compiler

takes advantage of the available trip counts to merge nested loops into a single loop using a transformation known as *Loop merging*. The merged loop has a trip count which is equal to the number of times the instructions in the innermost loop are invoked (product of all the trip counts in the nesting). For instance, if a kernel has a 3-level nested loop, each level with a trip count of 5, *Loop merging* results in a single loop with a trip count of 125.

The execution semantics post-*Loop merging* is intuitive when only the innermost loop has instructions and the outer loops are merely used to index the correct memory addresses. However, if there are instructions at each level to be executed, SE uses predication to conditionally execute the instructions at each level.

The *Loop merging* transformation is demonstrated by the following example. Listing 2.4 shows the C-code for the iDCT kernel from the StreamIt benchmark suite. The iDCT kernel has a 2-level nested loop each with a trip count of 8. *Loop merging* would result in a single loop with a trip count of 64 (8×8). However, note that the instruction in line 8, where the *temp_sum* is initialized to 0, needs to be executed every 8th iteration of the new merged loop. Also, the indexing of the coefficient array and the input stream array needs to be changed.

```
void kernel (float* in_stream, float* coeff, int* spm_start){
    float temp_sum;
    int i = 0;        // outer loop index
    int j = 0;        // inner loop index

    for (i = 0; i < 8 ; i++)  // begin outer loop
    {
        temp_sum = 0;     // initialize the accumulate reg
        for (j = 0; j < 8; j++) // begin inner loop
        {
          temp_sum += coeff[i*8 + j] * in_stream[j]   // MAC
              operation
        }
        out_stream[i] = temp_sum;
    }
}
```

Listing 2.4: C Code for iDCT 8X8 Kernel

Listing 2.5 shows the C-code for the same iDCT kernel after loop merging. The following key transformations can be observed

- Single loop with a trip count of 64.
- The coefficient array is now indexed from 0 through 63 using only single index.
- The input stream array is indexed iteratively from 0 through 7 using the loop index.
- On iterations 0 and every multiple of 8, the running sum is discarded by initializing the *temp_sum* to the product of the input and the coefficient. For all other iterations the MAC operation is performed.
- The output stream is written to, conditionally.

```c
void kernel (float* in_stream, float* coeff, int* spm_start){
    float temp_sum;
    int i = 0;        // merged loop index
    for (i = 0; i < 64 ; i++) // begin merged loop
    {
        if (i%8 == 0) {
          temp_sum = coeff[i] * in_stream[i%8];
        }
        else {
          temp_sum += coeff[i] * in_stream[i%8];
        }
        if (i%8 == 7) {
          out_stream[i>>3] = temp_sum;
        }
    }
}
```

Listing 2.5: C Code for iDCT 8X8 Kernel After *Loop merging*

Listing 2.6 shows the SE assembly code corresponding to Listing 2.5. Note that the *push* instruction is implemented by the push unit which takes care of the conditional execution using the push table. Figure 2.13 shows the DFG of the transformed kernel. The number on the nodes represent the instruction corresponding to line numbers in Listing 2.6. The red edges represent control dependencies.

```
 .text
main:
# Configure SIGs
    isig    $i0, 0, 1, 63, 0    #SIG0
# load the coefficient from SPM
    ld $r2, $i0
# read the appropriate token from the channel
    andi $r3, $i0, 7
    rd $r4, $r3
# decide whether to initialize accumulator reg
    bnez $r3, L1
    mult $r5, $r2, $r4
    b L2
L1: madd $r5, $r2, $r4
# Push the result
L2: push $r5
```

Listing 2.6: SE Assembly for iDCT Kernel

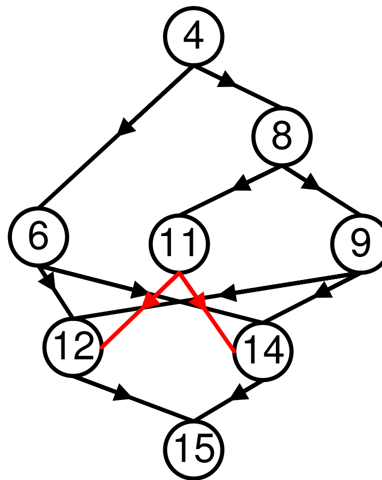


Figure 2.13: Dataflow Graph of the iDCT Kernel After Loop Merging.

During configuration phase, the registers corresponding to base, stride, length and offset of the SIGs are set by the *isig* instruction. The format of the *isig* instruction is *isig $dst, ⟨base value⟩, ⟨stride value⟩, ⟨length value⟩, ⟨offset value⟩*. One SIG is

used for the iDCT kernel. Table 2.9 list the configuration of the SIGS for the iDCT kernel example. The RSBs are also configured during the configuration phase. The RS tags are assigned and the data and control dependencies are encoded in the *Op A Tag*, *Op B Tag* and *Ctrl Tag* fields. The configurations of the stream, compute and memory RSBs are shown in Table 2.10 and Table 2.11. Note that the unconditional branch instruction (line 13 in Listing 2.6) doesn't need to be hosted by any RS in the dataflow execution model.

SIG Num	RS Tag	Base Reg	Stride Reg	Length Reg	Offset Reg
0	1	0	1	63	0

Table 2.9: Configuration of Stream Index Generators

RSB	RS Tag	Op	Op A Tag	Op A	Op A rdy	Op B Tag	Op B valid	Op B	Op B rdy	Ctrl Tag
LD RSB 0	2	ld	1	?	0	-	0	-	1	-
ALU0 RSB 0	3	andi	1	?	0	-	0	7	1	-
STR RSB 0	4	rd	3	?	0	-	0	-	1	-
ALU1 RSB	5	bnez	3	?	0	-	0	-	1	-
MUL0 RSB	6	madd	2	?	0	4	1	?	0	5
MUL0 RSB	6	mult	2	?	0	4	1	?	0	5

Table 2.10: Configuration of Stream (s-RSB), Compute (c-RSB) and Memory (m-RSB) RSBs

The execution of this kernel now follows the semantics of the regular predicated dataflow. Figure 2.14 shows the execution and the state of RSBs, SIGS and the TDN for the first 4 clock cycles. The representation used is the same as in response to question 1.

Ctrl valid	Op Ctrl	Op Ctrl rdy	Pred- -icate	Context
0	-	1	-	0
0	-	1	-	0
0	-	1	-	0
0	-	1	-	0
1	?	0	1	0
1	?	0	0	0

Table 2.11: Configuration of Stream (s-RSB), Compute (c-RSB) and Memory (m-RSB) RSBs

Note: In cases where the trip count is unknown or the loop is based on runtime values (e.g. while loop), *Loop merging* cannot be used to transform the original kernel. I propose to used predicated SIGs along with conditionals to handle such loops.

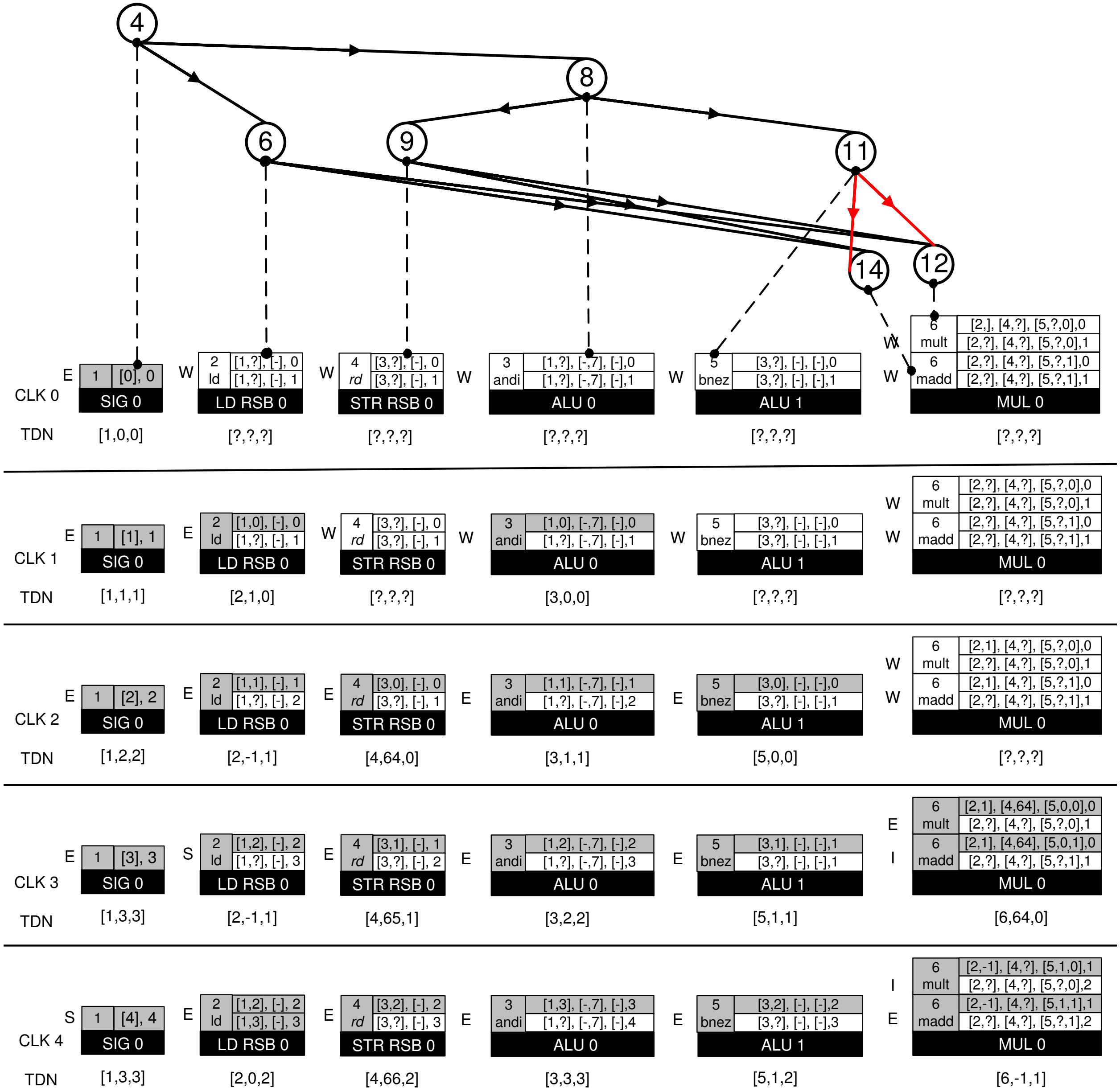


Figure 2.14: DFG Mapping and RS/SIG/TDN State During Execution of the iDCT Kernel

2.7 Evaluation

This section presents the evaluation of the StreamEngine. The impact of instruction locking is evaluated by comparing the energy expended in instruction delivery by SE with that of embedded RISC cores. The overall impact of the CDE model and instruction locking on the energy-efficiency is evalauted by comparing the total energy spent and energy-efficiency (in GFLOPS/W) across a set of stream kernel benchmarks. Next, the impact (if any) of transforming control dependencies to data dependencies is evaluated. Finally the performance scaling and VLSI implementation of SE are discussed.

2.7.1 Experimental Set-up

Benchmark	**Description**
LPF	128-tap Low pass filter
MVD	Motion vector decode
Autocor	Produces an autocorrelation series of length 1 for a series of vectors of length 32
FFT	Single stage of a 256-point single precision floating point Fast Fourier Transform
DCT	Single stream of a 8 x 8 2D Discrete Cosine Transform
RGB2YUV	RGB to YUV conversion algorithm used in MPEG
CRC32	Cyclic Redundancy Check using 32-bit CRC-32 used in IEEE 802.3

Table 2.12: Benchmarks Representing Stream Kernels

.

Table 2.12 lists the stream kernel benchmarks that were utilized for the evaluation. All the benchmarks belong to the StreamIt suite except for CRC32. CRC32 was included as it is a stateful kernel. A cycle-accurate model was constructed in SystemC for the performance characterization of the StreamWorks architecture. The power consumption characteristics of SE were obtained by generating the VLSI layout of the design. A register-transfer level (RTL) VHDL description of the SE was synthesized in the TSMC 45nm technology for 500 MHz using Synopsys Design Compiler and the layout was obtained using Cadence Encounter. The power characterization of each of the SE components was done by performing a VCD-based power analysis using Synopsys Primetime PX. This power characterization was used in conjunction with average component occupancies obtained from the cycle-accurate simulator to estimate the SE power consumption. The configuration of the synthesized SE is shown in Table 2.13. The SE has 48 RS, 3 ALUs, 2 MULS and 4 SIGS. Each RS has 5 operand buffers and each of the ALUs and MULs has a 4-stage pipeline. The SE also includes a 2 kB SPM and a 2 kB channel for input streams.

The SystemC model takes as input the SE assembly for individual stream kernels. We utilize the StreamIt compiler that has been enhanced for generating multi-threaded C code for heterogeneous architectures to compile the StreamIt specification of the benchmarks. The SE assembly is generated manually from the C code. In other words, all the system-level optimizations are automated through the compiler and no hand optimizations are used for translating C code to assembly.

The embedded RISC cores utilized as the baselines for our comparative studies are the LeonHabnic and Gaisler (2007) and the ARM Cortex A-15.

The Leon RISC core has a SPARC V8 compliant 32-bit integer unit and is available as open source from the European Space Agency. It comes with GNU tool chain and a Bare-C cross compiler (collectively called as the GRLIB toolchain). The core was

	Configuration
Bitwidth	32
Input Channel ports	2
Input Channel capacity	2kB
SPM capacity	2kB
SIGs	4
ALUs	3
Multipliers	2
ALU/Multiplier pipeline depth	4
Stream RSBs	2
Compute RSBs	5
LD/ST RSBs	2
RS per ALU RSB	8
RS per MUL RSB	4
RS per Stream RSB	4
RS per LD/ST RSB	4
Operand buffers per RS	5

Table 2.13: StreamEngine Configuration

synthesized in the TSMC 45nm technology with two different configurations - one with an instruction cache and the other with an instruction local store (SPM).

For SE and LEON, the register-transfer level (RTL) VHDL description of the processors were synthesized in the TSMC 45nm technology for 500 MHz and 200 MHz respectively. We designed the RTL for SE whereas the RTL for LEON is available as open source. Synopsys Design Compiler was used for the synthesis to obtain the gate-level netlist. This netlist was then used to obtain the layout using Cadence Encounter. The power characterization was done by performing a VCD-based power analysis using Synopsys Primetime PX. As the LEON comes with the entire toolchain including the compiler, I was able to compile the benchmarks for LEON and directly run them on the gate-level netlist using Cadence NCSim. The switching activity was dumped into a VCD file and Primetime PX was used for the power analysis.

In case of SE, since I do not have the full fledged compiler, I could not run the benchmarks directly on the gate-level netlist of the SE. I characterized the netlists for different components with various testbench workloads again using Cadence NCSim for netlist simulation and Primetime PX for power analysis. I built a cycle-accurate SystemC simulator of the SE to obtain the average component occupancies while running the kernel benchmarks. I used the power characterization of the netlist in conjunction with the component utilizations to obtain the power characterization for SE.

The power characterization methodology is is shown in Figure 2.15.

The configuration of the Leon core with instruction cache is shown in Table 2.14. The StreamIt compiler was again used to generate C code which is then compiled for Leon using the supporting tool chain. The GRLIB tool chain enabled us to perform VCD based power analysis using gate-level simulations of benchmark workloads to obtain the power consumption.

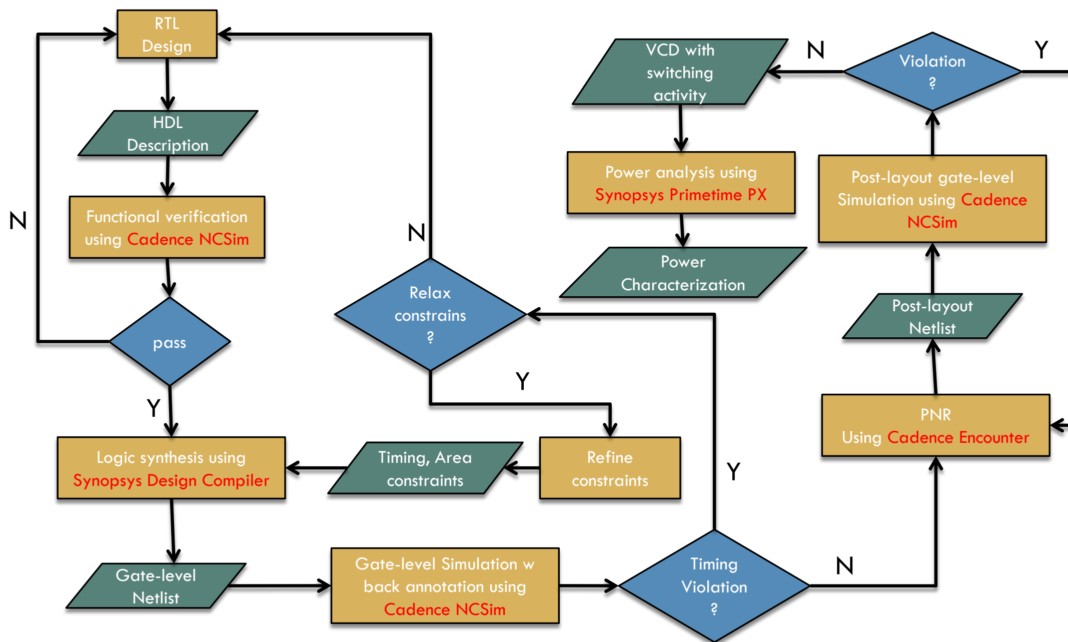


Figure 2.15: Power Characterization Methodology for SE and LEON

Since the ARM core is not open source and neither is the netlist made available, the ARM Cortex A-15 core was modeled using the gem5 simulator Binkert *et al.* (2011). The *arm_detailed* CPU in gem5 models a modern OoO ARM v7 architecture and can be configured to Cortex A-15 using appropriate parameters. Table 2.15 lists the configuration of the baseline ARM Cortex-A 15 used for the evaluation.

The major differences between the modeled baseline and commercial ARM Cortex are the size of the L1 caches, the technology node and the operating frequency. For a fair comparison between the ARM core and SE, minimal I-cache (1 kB) was used for ARM that could accommodate the entire stream kernel with only compulsory misses (commercial ARM Cortex A-15 has 32kB I-cache). Also, the D-cache used for ARM was 4 kB (equivalent to the sum of capacities of the SPM and input channel on the SE) as opposed to 32 kB D-cache in the commercial version. The working data set for ARM was restricted to 4 kB to eliminate any performance hits due to capacity

misses. Thus, without sacrificing any performance, the ARM core was evaluated for a low power configuration. While the commercial ARM is at 32nm operating at 1.5 GHz, power and area characterization was performed for 45nm at 500 MHz (same as SE). McPAT Li *et al.* (2009) was used for power and area characterizations of the ARM core. The stream kernel benchmarks were also SIMDized[3] to utilize the NEON SIMD unit in Cortex A-15. Evaluation with and without SIMDization is presented. The CodeSourcery Graphics (2009) ARM GNU/Linux tool chain was used to cross-compile the kernel benchmarks for the ARM core.

[3]CRC32 and MVD were not SIMDized due to inter-iteration dependencies and conditional execution respectively.

	Configuration
Integer ALUs	1
Integer MUL/DIV Unit	1
Branch Predictor	0
LD/ST Unit	2
Floating-Point Units	0
Integer pipeline depth	7
Integer Registers	32
L1 I-cache	1 kB
L1 D-cache	4 kB
L1 Associativity	4
Cacheline size	32

Table 2.14: SPARC V8-based LEON Configuration

	Configuration
Integer ALUs	2
Integer MUL/DIV Unit	1
Branch Unit	1
LD/ST Unit	2
Floating-Point Units	2
Branch Predictor	Tournament
BTB entries	256
Integer pipeline depth	15
Integer Registers	128
Single-precision FP Regs	32
Double-precision FP Regs	16
Issue Width	8
L1 I-cache	1 kB
L1 D-cache	4 kB
L1 Associativity	2
Cacheline size	64

Table 2.15: ARM Cortex A-15 Configuration

2.7.2 *Instruction Delivery Energy Savings*

This section evaluates the impact of fine-grain instruction reuse enabled by instruction locking by comparing the energy expended in instruction delivery for SE against that of LEON and ARM Cortex A-15.

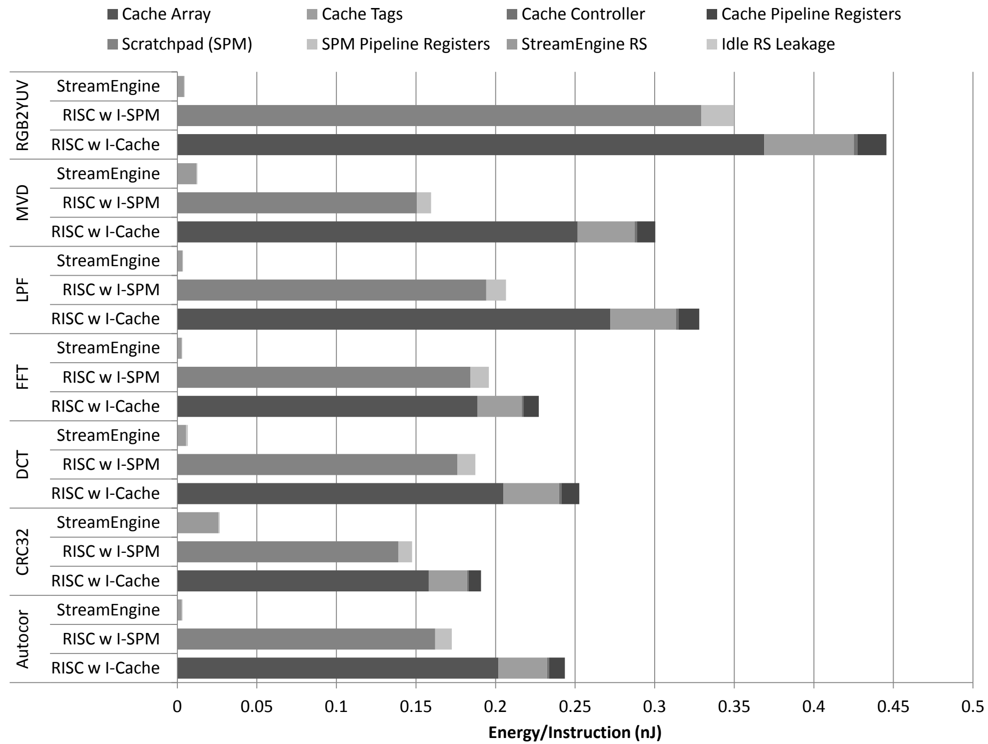


Figure 2.16: Instruction Delivery Energy Comparison with Cache-based and SPM-based LEON

Figure 2.16 demonstrates the energy savings obtained due to instruction locking by the SE when compared to LEON cores with instruction cache and instruction SPM. The different components that were included towards the instruction delivery energy in the cache-based LEON were the cache array, the cache tags, the cache controller and the pipeline registers. In case of SPM-based LEON, only the scratchpad and pipeline registers were included. The RSBs are the instruction store for SE and hence all the RSBs account towards the total instruction delivery energy. As Figure 2.16

suggests, the SPM-based LEON core, because of its software managed instruction memory, requires lesser energy than the cache-based LEON core. The SE gives upto 95× energy savings with an average of 58.5× and 41.9× compared to cache-based and SPM-based LEON cores respectively.

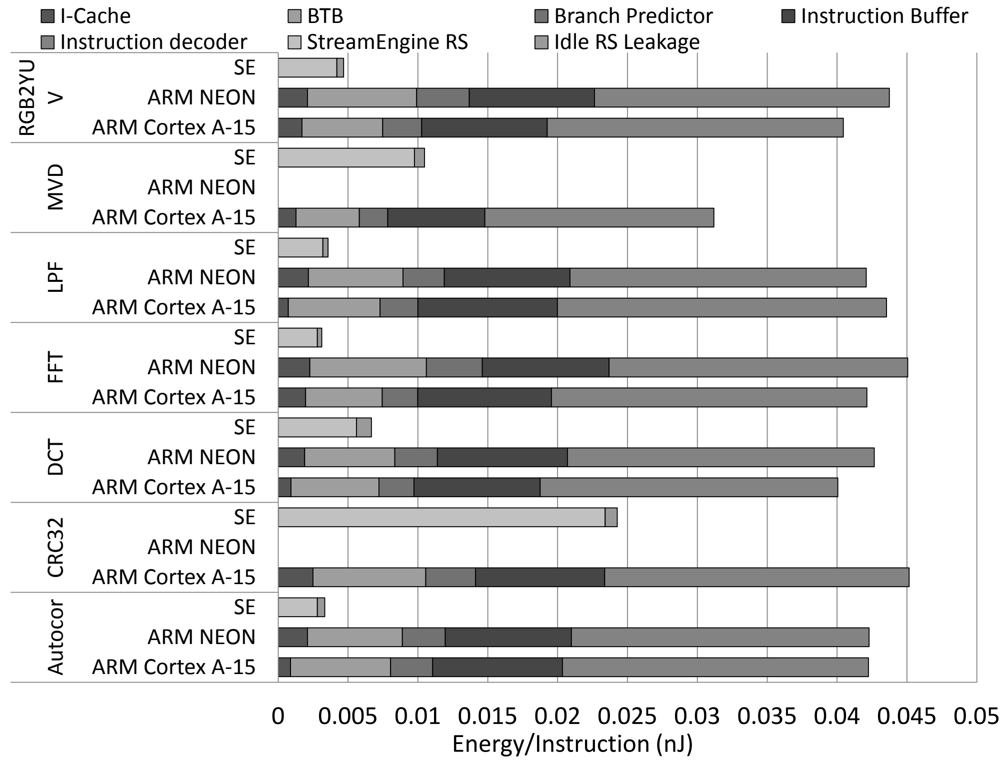


Figure 2.17: Instruction Delivery Energy Comparison with ARM and ARM NEON

Figure 2.17 presents the energy expended in instruction delivery when executing kernel benchmarks in SE and the ARM cores without and with NEON SIMD extensions. The different components that were accounted for towards the delivery energy in ARM were instruction cache, BTB, branch predictor, instruction buffer and instruction decoder. Figure 2.17 demonstrates that instruction locking achieves upto 93% reduction in delivery energy when compared to ARM cores.

It is evident that preserving the instruction locality at the functional units thereby bypassing the fetch and decode stages results in these tremendous energy savings.

2.7.3 Impact of CDE on IPC

Context-aware Dataflow Execution exploits the inherent DLP across stream kernel iterations by overlapping the execution of successive iterations; thus, essentially converting DLP to ILP. The respective IPC numbers were utilized to compare the inherent ILP of a stream kernel with that acheived by SE using CDE. The inherent ILP was obtained by executing one iteration of the kernel to completion before initiating the next iteration.

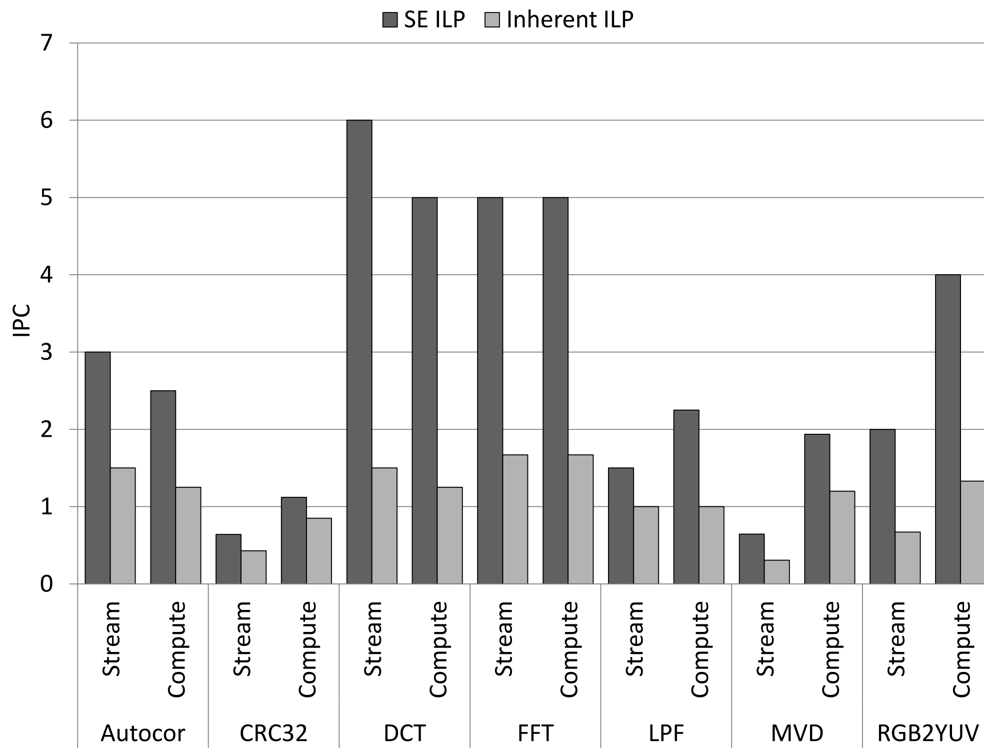


Figure 2.18: Impact of CDE on IPC

Figure 2.18 plots stream and compute IPC for the kernel benchmarks due to the inherent ILP and extracted (SE) ILP. The instructions hosted by the Stream and Load/Store RSBs were accounted for by the stream IPC, and the compute IPC includes instructions on the compute RSBs alone. The *ISIG* instruction was not

included for any of the IPC calculations. As it is evident from the figure, CDE coupled with instruction locking efficiently exploits DLP to boost the IPC for all kernels. As expected, there is no significant boost in the case of CRC32 on account of its statefulness. For FFT and DCT, the SE achieves a compute IPC of 5 which is equal to the number of execution units on the SE; thus, achieving 100% utilization. The average increase in stream and compute IPC is 2.44 and 2.45 respectively with a standard deviation of 0.92 and 0.94 respectively. It is important to note that unlike VLIW architectures, this boost in parallelism is achieved purely by the CDE model without any explicit unrolling.

2.7.4 SE Energy-efficiency

This section evaluates the overall energy savings and efficiency at the core level.

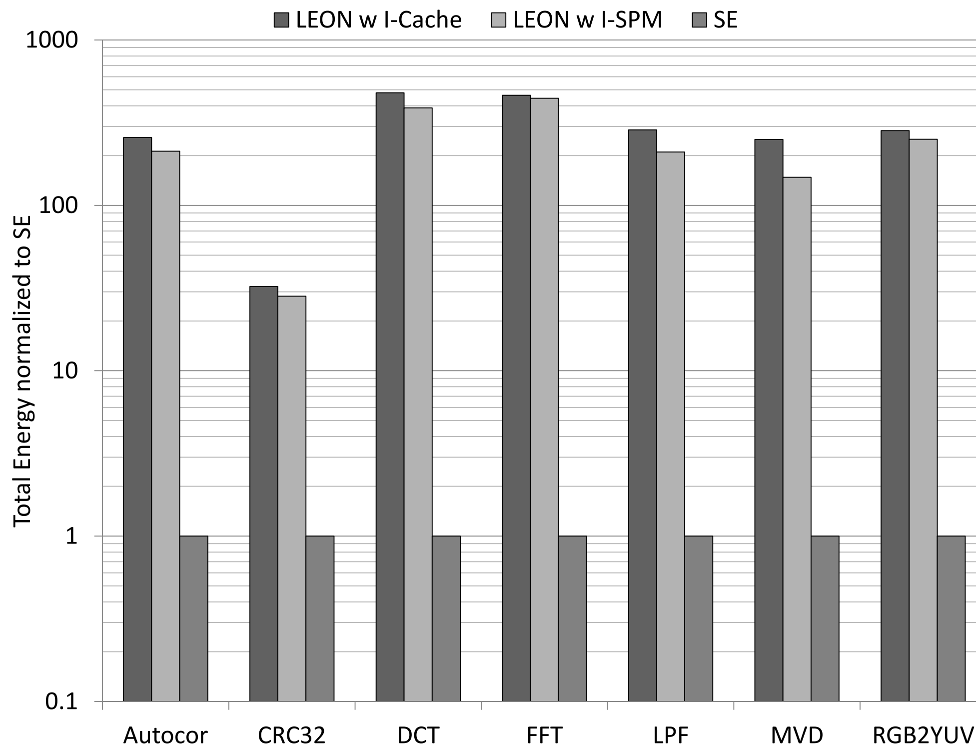


Figure 2.19: Total Energy per Kernel of the LEON RISC Core Normalized to that of a SE

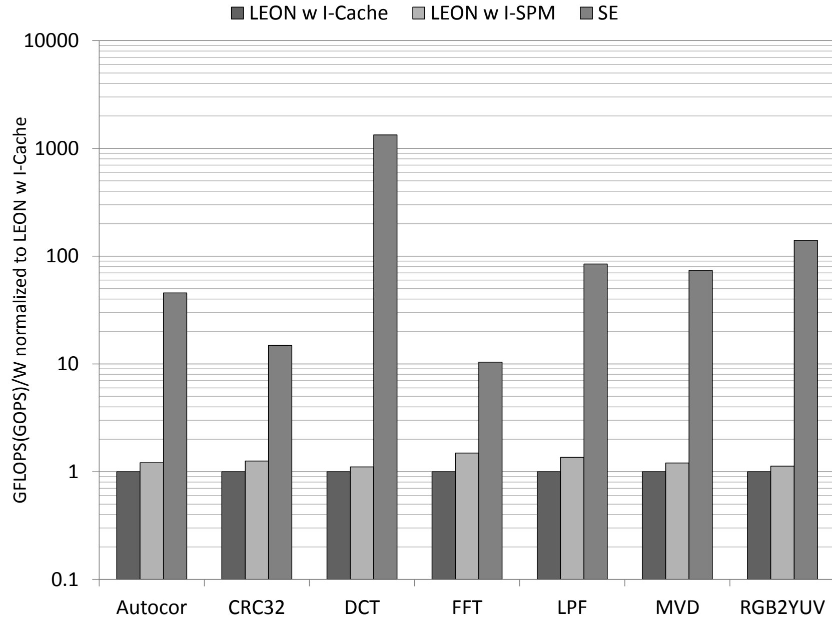


Figure 2.20: Energy-efficiency of SE and SPM-based LEON Normalized to that of Cache-based LEON

Figure 2.19 presents the total energy of the LEON cores normalized to that of an SE. As demonstrated by the figure, with the exception of the stateful kernel CRC32, the SE expends between two to three orders of magnitude less energy than the cache-based and SPM-based LEON cores. In the case of CRC32, the number of kernel invocations in flight is limited by the inter-iteration dependencies and therefore the SE is unable to fully utilize the available functional units. However, even for CRC32 the SE achieves more than an order of magnitude energy savings. The SE expends, on an average, 293.12 and 240.42 times less energy when compared to cache-based and SPM-based LEON respectively.

Figure 2.20 depicts the energy-efficiency in GFLOPS/W (GOPS/W for integer kernel benchmarks CRC32 and MVD) of the SPM-based LEON core and SE normalized to that of a cache-based LEON core. Memory operations are not included in the GFLOPS calculation. As demonstrated by the figure, with the exception of DCT, the SE is one to two orders of magnitude more energy-efficient than the cache-based

LEON core. In case of DCT, the SE is more than three orders of magnitude more energy-efficient than LEON. On an average, the SE is 243 times more energy-efficient than LEON.

Figure 2.21 presents the total energy of the ARM cores normalized to that of an SE. As demonstrated by the figure, SE expends between one to two orders of magnitude less energy than the ARM cores.

Figure 2.22 depicts the energy-efficiency of the ARM NEON core and SE normalized to that os ARM. The SE delivers. on an average, 62X and 40X more GFLOPS/W in comparison with ARM without and with SIMD extensions respectively.

It is worth mentioning that even with a low power implementation of SPARC V8 in LEON which consumes power in the range of 15 mW, an ARM core consuming $\tilde{0}.5$ W is much more energy-efficient than LEON due to the better performance resulting from its out-of-order execution logic.

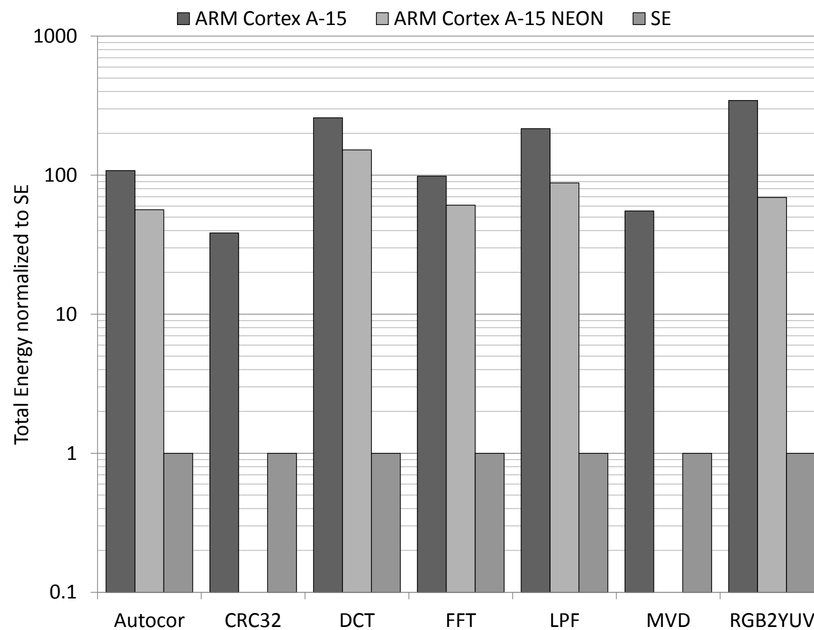


Figure 2.21: Total Energy per Kernel of the ARM Cortex A-15 Core Normalized to that of a SE

2.7.5 Impact of Transforming Control-flow to Dataflow

This section evaluates the impact of transforming control dependency to data dependency via the predicate bit in the RS. As discussed in Section 2.3.2, by control dependency we refer to the branches encountered at *for* and *while* loop boundaries and any *if-then-else* constructs.

Figure 2.23 shows the different types of stalls for a range of SE configurations across the kernel benchmarks. The Y-axis plots the average stalls in clock cycles per RS in the SE. The operand stall is the number of clock cycles between the firing of an instruction and the availability of all its data operands corresponding to the next iteration. The control stall is the number of clock cycles between the firing and the availability of control information that decides whether the instruction needs to be executed in the next iteration. It should be noted that operand and control stalls are

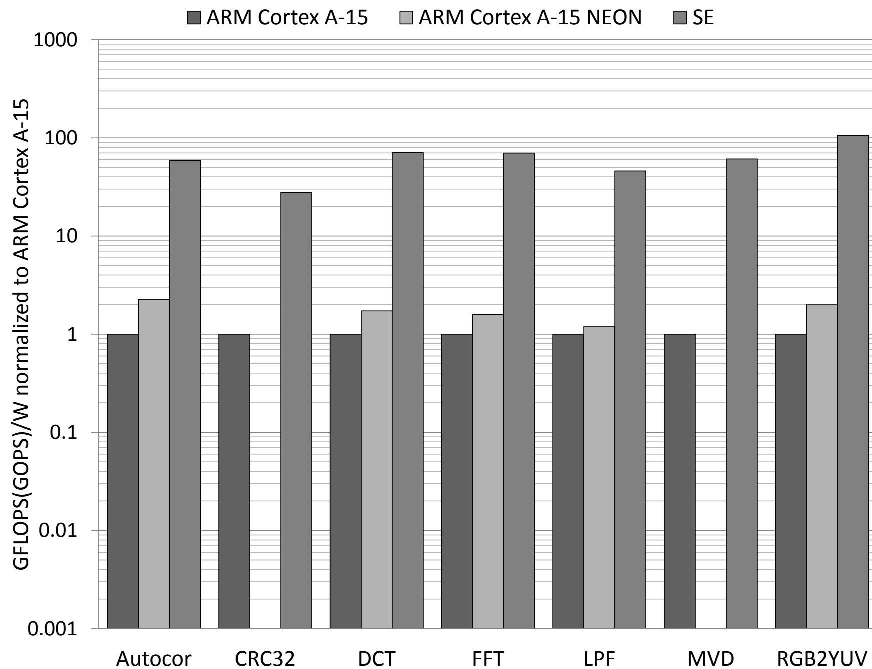


Figure 2.22: Energy-efficiency of SE and ARM Cortex A-15 with SIMD Extensions Normalized to that of ARM Cortex A-15

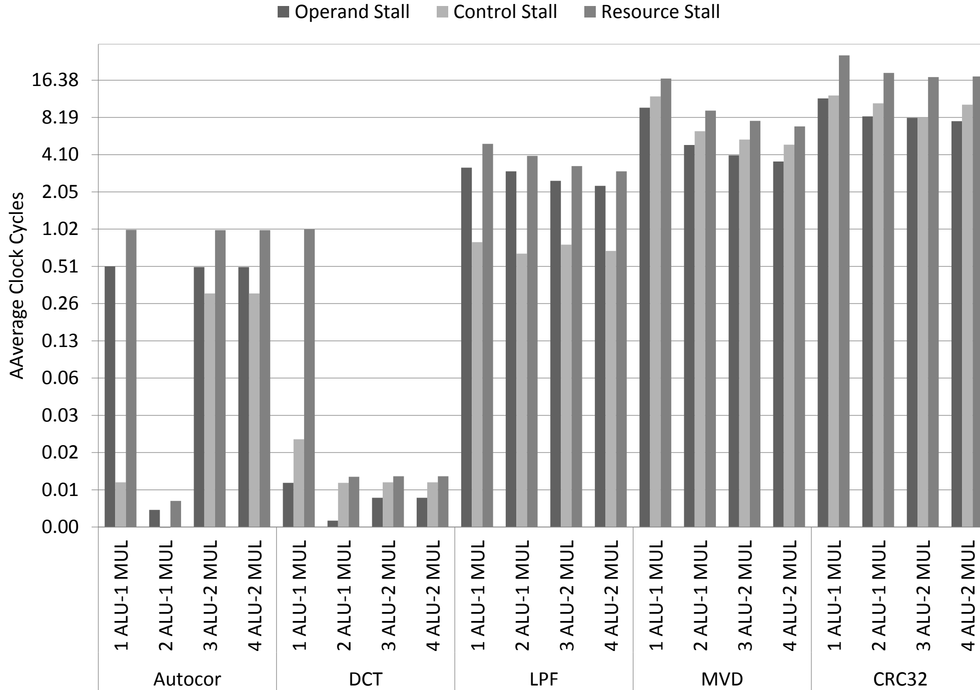


Figure 2.23: Average Stalls per Instruction with Predicate in a SE

exclusive, and can be overlapped in time. The resource stall is the number of clock cycles between consecutive firings. The resource stall can be broken down as the sum of the greater of operand and control stalls and stall due to resource arbitration. The X-axis lists different configurations of the SE for all the kernel benchmarks.

The results demonstrate that our transformation is an efficient way to preserve the dataflow semantics in kernels with control instructions without incurring any penalty in performance. With the exception of DCT, MVD ,and CRC the control stall is always less than the operand stall which implies that the control evaluation is overlapped and hidden by the operand evaluation. In the case of DCT, MVD and CRC, we notice that the instruction does not fire immediately upon the availability of its execution path which implies that the execution is limited by resource availability and not by the control availability. The FFT and RGB2YUV kernels do not have any branch instructions, and therefore are not included in the figure.

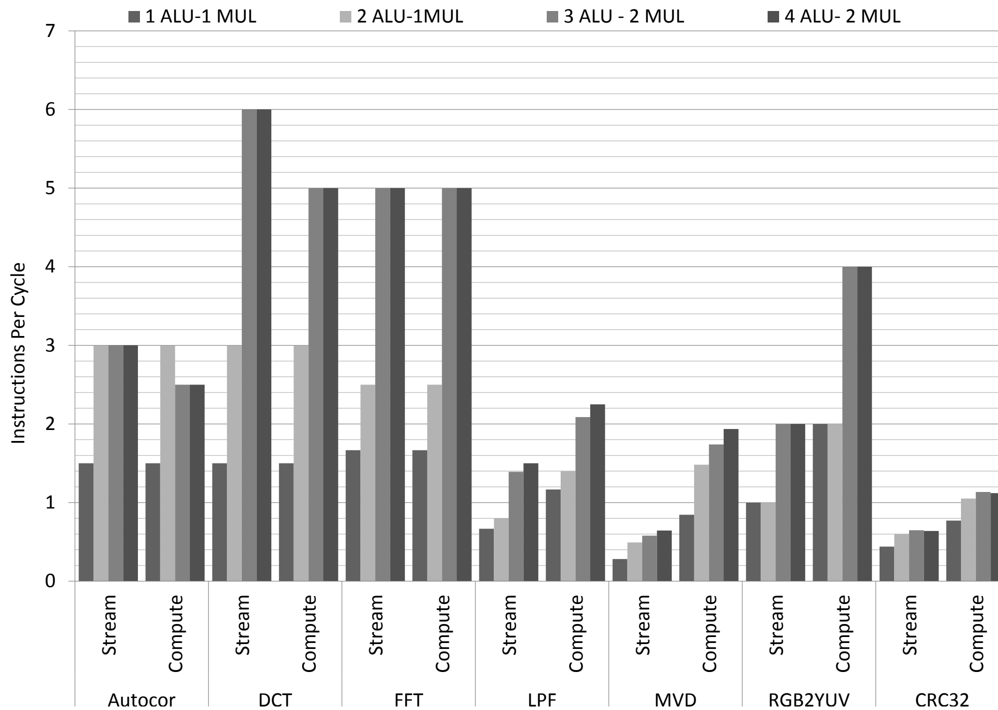


Figure 2.24: SE Performance Scaling: Stream and Compute Instructions per Cycle

We evaluate the scalability of a single SE by examining the performance of each of the kernel benchmarks as a function of the compute resources available in the SE. Figure 2.24 plots the stream and compute instructions per cycle (IPC) achieved for all the kernel benchmarks. The stream operations that read from the input channel and write in consumer SE channel (RD, RMV and $PUSH$, discussed in detail in Chapter 3) account towards the stream IPC. All other operations account towards the compute IPC. The *ISIG* operations were excluded for IPC calculation. Since each multiplier is associated with a single accumulator, the kernels that use accumulator do not run the same code for 1-MUL and 2-MUL configurations. Autocor, DCT and LPF are the kernels that can operate on data from successive iterations of the actor in the presence of two accumulators but stay limited to a single iteration with one multiplier. The

results demonstrate that the SE scales very well for all the benchmarks other than Autocor. The 3-ALU and 2-MUL configuration seems to be the sweet spot. In the case of Autocor, the scaling is limited due to the number of input channel ports as observed from the IPC of stream instructions. Autocor consumes two data tokens from the input channel per iteration. In the case of one multiplier, when the kernel is operating on a single iteration, it continuously receives input tokens and thus each of the functional units have new operands every cycle. However, when it operates across 2 iterations with two multipliers (or greater), the computation on each iteration gets interleaved with the another because only two data tokens can be available from the channel in one cycle. Hence, the performance scaling is limited.

2.7.7 VLSI Implementation, Area and Power

We created a VLSI implementation of the SE in TSMC 45 nm technology. The area of the SE was 1 mm^2 and the average power consumption was 40 mW at 500 MHz. Figure 2.25 presents the layout and percentage area of the StreamEngine as obtained from Cadence Encounter.

Figure 2.26 depicts the percentage power consumption of the various units within the SE. As the figure suggests, maximum power is spent in useful computation in the datapath. Note that the reservation stations contribute to the datapath as they host operands in addition to instructions thus playing the role of a Register file.

2.7.8 Comparison with ELM

The ELM processor operates at 200 MHz in TSMC 130nm technology. Compared to LEON3 with an instruction cache, ELM reduces the cost of supplying instructions by 49× and acheives an energy-efficiency that is 23× greater than LEON3 Balfour *et al.* (2008).

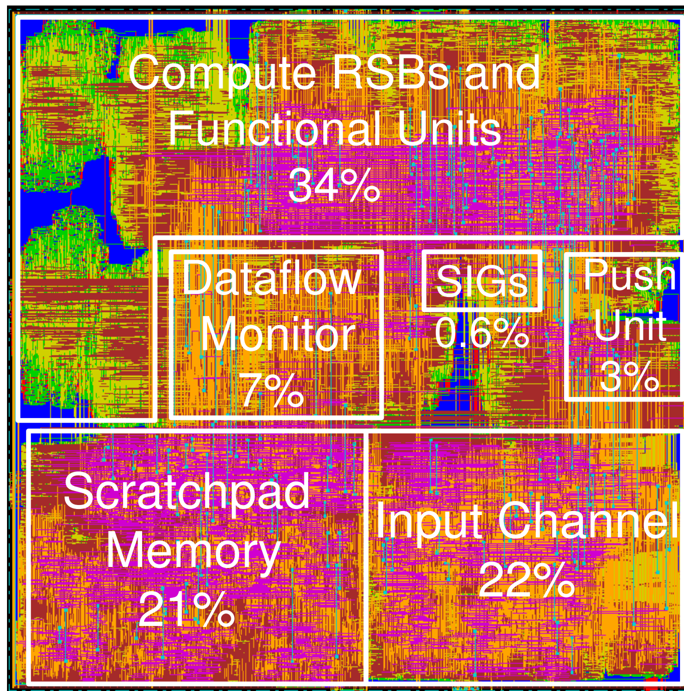


Figure 2.25: Layout and Area Breakdown of SE

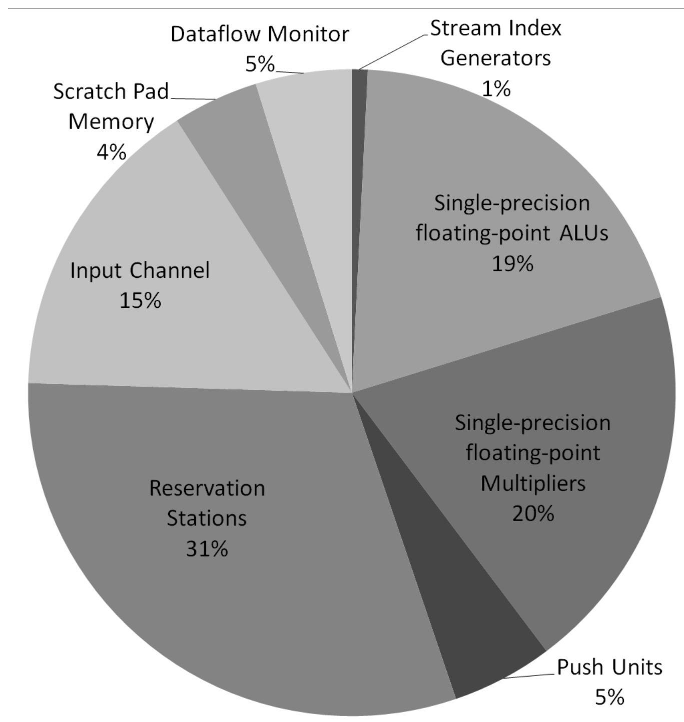


Figure 2.26: Power Breakdown of SE

Using LEON3 with instruction cache as the common baseline, I attempt to compare the energy-efficiency of StreamEngine and ELM. I would like to highlight that the energy saving numbers reported are an average across a number of kernel benchmarks. Among the benchmarks used for the evaluation of ELM and SE, three are common benchmarks (crc, dct and rgb2yuv). Also, note that there can be differences in the implementation of the same benchmark as they are not derived from a common benchmark suite. Based on the respective benchmarks, StreamEngine achieves 20% more energy savings in instruction delivery and expends 10$\times$ less total energy than the ELM processor.

Chapter 3

THE MULTI-CORE DATAPLANE

3.1 Introduction

In this Chapter, we introduce StreamWorks, a multi-core embedded co-processor for streaming applications. StreamWorks utilizes StreamEngine (SE) proposed in Chapter 2 as processing elements to realize a scalable multi-core stream architecture. StreamWorks is aimed at high performance dataplane processing in System-on-Chip based heterogeneous multiprocessor architectures.

The top-level view of the StreamWorks architecture is shown in Figure 3.1. Multiple SEs and scratch-pad memory (SPM) pairs are grouped into a StreamCluster (or SC). Each intra-cluster SPM is visible within the address range of every SE that is present within the cluster. An SE can perform inter-SE communication through either a high bandwidth local communication fabric or through the SPM.

We also introduce the channel and push unit pair that constitute the communication unit of the SE for inter-SE communication. The input channel and push unit pair implements specialized stream input/output operations and supports *FIFO virtualization*, a technique that is used for high fan-in/fan-out communication patterns. A credit-based back pressure scheme is used for inter-SE communication, which extends the dataflow semantics across SEs. The channel and push unit pairs are connected to the local communication fabric, and therefore the fabric is the preferred means for stream communication. Several SC together along with the a Network-on-Chip (NoC) fabric define the StreamWorks architecture.

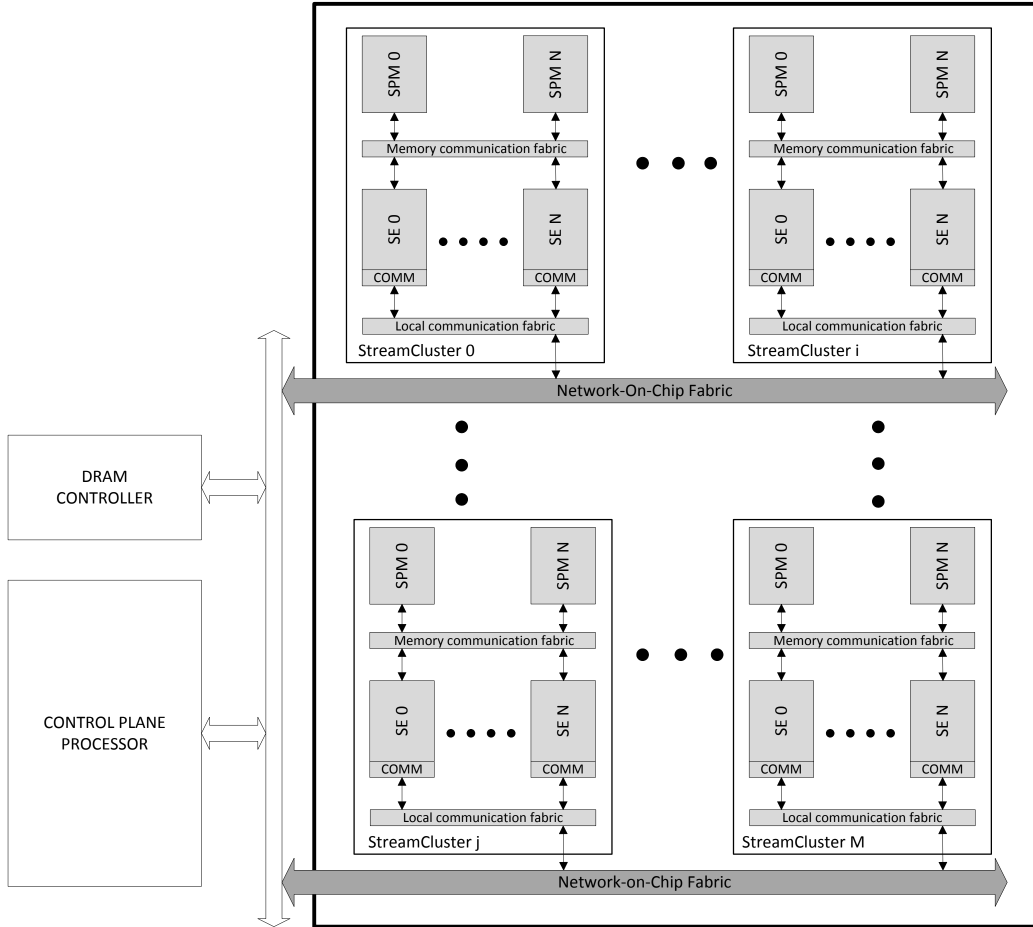


Figure 3.1: StreamWorks Architecture

In Section 3.2, previous work on multi-core data plane co-processors aimed at streaming applications is presented. Section 3.3 presents the FIFO virtualization scheme and its implementation that enables SEs to efficiently communicate across stream kernel boundaries. Finally, in Section 3.4 the evaluation of StreamWorks is presented.

3.2 Related Work

Stream processor architectures have gained considerable attention over the last decade. The Imagine stream processor Khailany *et al.* (2001); Kapasi *et al.* (2002) can be regarded as one of the forerunners in programmable streaming architectures. It

builds upon several ideas including vector processors, VLIW media processors, DSPs and sub-word SIMD parallel instruction sets. The main insight of Imagine is that the execution flow is effectively deterministic and can be managed by software. Only memory accesses are non-deterministic, and therefore memory accesses are decoupled. One of the architectural innovations that Imagine introduces is the enormous data bandwidth supported by a streaming memory system local to the co-processor and a large amount of on-chip intermediate storage for streams in the form of a wide Stream Register File (SRF) Rixner *et al.* (2000). Multiple functional units are grouped in an arithmetic cluster and 8 such arithmetic clusters form the Imagine compute subsystem. Each of the arithmetic clusters executes the same VLIW schedule. This can be regarded as 8-wide SIMD execution where each instruction is a VLIW. Thus each cluster exploits the ILP exposed by the compiler that forms the VLIW schedule and the clusters together exploit the DLP across streams. Unlike StreamWorks, Imagine cannot exploit either Task or Thread level parallelism since all the clusters execute the same kernel and a VLIW schedule is executed until completion. Imagine also requires a complicated compiler that can generate an efficient VLIW schedule and the communication needs to be scheduled by the programmer which is a huge programming overhead, the authors propose a new programming language (StreamC) in order to capture the communication pattern. StreamWorks provides hardware support for inter-kernel communication without the programmer having to schedule the communication.

MerrimacDally *et al.* (2003), the successor of Imagine, is aimed at high-performance scientific applications. The main idea in Merrimac, similar to GPUs, is to hide the memory latencies using useful computations. This is more suited for applications with high computation to memory bandwidth ratios. The core of Merrimac is a single-chip stream processor which is architecturally similar to Imagine. This processor chip along

with 16 DRAM chips form a single Merrimac node. Merrimac employs a high-radix interconnection network to connect 16 such nodes to form the entire co-processor. One can easily draw parallels between Merrimac and present day GPUs. Again, only instruction and data-level parallelism are exploited by Merrimac at the expense of a complex compiler and programming overhead. In contrast to StreamWorks, all the communication in Merrimac is scheduled in software by the programmer. Also, the static scheduling of VLIW instructions cannot exploit run-time opportunities.

Adres Mei *et al.* (2003) and TRIPS Sankaralingam *et al.* (2003b); Burger *et al.* (2004) provide a 2-D array of functional units in the processor data path that can be utilized to execute the compute intensive loops of an application. Morphosys Singh *et al.* (2000) integrates an 2-D array of functional units with a RISC core in a SoC configuration. The SEs in the StreamWorks architecture are much coarser in granularity than individual functional units in Adres, TRIPS and Morphosys. An array of SE executes independent threads while an array of functional unit in Adres, TRIPS and Morphosys typically execute a single thread. Further, both Adres and TRIPS do implement a fetch/decode/execute pipeline while in Morphosys the RISC core broadcasts the instructions to the functional units. In contrast, StreamWorks extensively utilizes instruction reuse and minimizes power consumption due to instruction movement.

Soda Lin *et al.* (2006), Ardbeg Woh *et al.* (2008) and AnySP Woh *et al.* (2009) support wide-issue instructions (including SIMD) to exploit ILP and DLP that is present in computation intensive loops of signal processing applications. While Soda and Ardbeg mainly targets software defined radio, AnySP can be regarded as a domain-specific programmable co-processor for signal processing. Among other architectural support for signal processing, one of the key features of AnySP is the introduction of flexible functional units. The proposed flexible functional units allow reconfigu-

ration and effectively turn two SIMD lanes into a 2-deep execution pipeline; thus, exploiting pipeline parallelism for workloads that under-utilize the SIMD width. The major drawback of these architectures is that they support only 16-bit datapaths. The main insight of using a 16-bit datapath is to eliminate the overheads associated with data alignment for SIMD execution. However, they still have the overhead of moving data from scalar to vector register files and vice-versa. StreamWorks, on the other hand, supports 32-bit integers and single-precision floating point datapaths and doesn't incur any of the overheads associated with packing/unpacking data for DLP extraction.

The *T architecture Nikhil *et al.* (1992) addresses latency and synchronization issues with remote loads. Fine grained multi-threading, and memory status bits along with synchronization co-processor are presented as solutions for the two problems respectively. Synchronization between kernels is implemented in StreamWorks through a credit based scheme as will be described in detail in Section 3.3 whereas *T only implements a blocking read. Further StreamWorks does not require a dedicated/centralized synchronization co-processor.

Raw Taylor *et al.* (2002) is a statically scheduled tiled multi-core architecture that includes specialized instructions that directly write to the buffers of the router. The push unit that is discussed in Section 3.3.2 in the StreamWorks architecture also implements instructions that write to the local interconnection network of the cluster. The StreamWorks push unit provides additional hardware support for credit based communication and hardware support for synchronization.

AsAP Yu *et al.* (2008) and Intel communication processor Chen *et al.* (2004) include a 2-dimensional (2D) array of simple processing elements (PEs) along with instruction memory for data plane processing. The SEs in StreamWorks do not utilize a separate instruction memory, and use RSB with instruction reuse for kernel

execution. Also, contrary to the dataflow model in SE, each PE in AsAP implements a simple pipeline that can only exploit limited ILP.

In recent years a number of embedded processor architectures have been proposed in industry Johnson and Kunze (2003); Baines and Pulley (2004); Pham *et al.* (2006) that are aimed at multimedia and communication applications both of which demonstrate streaming characteristics. The commercial embedded processor architectures primarily integrate either an array of RISC Johnson and Kunze (2003) (with SIMD support in Cell BE Pham *et al.* (2006)) or VLIW Baines and Pulley (2004) cores for high performance data plane processing. Elm Dally *et al.* (2008) from Stanford utilizes instruction registers and operand forwarding in the fetch-decode-execution pipeline to minimize the power consumption due to instruction and data movement. Elm requires the application to be written in Elk which exposes the parallelisms to the compiler. All of the above mentioned architectures primarily address ILP and DLP but do not provide any architectural support for exploiting TLP in streaming workloads which is offered by the StreamWorks architecture.

3.3 FIFO Virtualization

The inter-core communication mechanism is a key factor that determines the performance of any multi-core architecture. In the context of streaming architectures, inter-core communication becomes crucial for realising the software pipeline to exploit the task-level parallelism.

Since streaming applications can be best expressed as Synchronous Data Flow graphs with well-defined communication patterns, the number of tokens produced and consumed per kernel firing is predetermined. This a priori knowledge enables StreamWorks to implement hardware FIFOs as bounded memory arrays. However, stream applications also commonly exhibit high fan-in and fan-out communication

patterns that require a kernel to have access to multiple input and output FIFOs. FIFO virtualization enables the mapping of a single hardware FIFO to multiple *virtual* FIFOs.

StreamWorks implements FIFO virtualization using the input channel and the push unit. The input channel and push unit together constitute the inter-SE communication unit of StreamWorks. They support stream-specific FIFO operations and enable the extension of the dataflow execution model across SEs. The realisation of FIFO virtualization is discussed in detail in the following sub-sections.

3.3.1 The Input Channel

The input channel is implemented as a memory array that is operated upon by the stream instructions in the stream RSBs. The stream RSBs host two specialized instructions, *RD mem* and *RMV mem*. Both the instructions perform a read on a particular address location in the channel that is specified by *mem*. The *RMV* operation is similar to the *pop* operation for FIFO. The distinction is that while the *pop* operation operates upon the head of FIFO. The channel as implemented in SE architecture is not a FIFO, but rather a memory array. Thus, the *RMV* operation can perform a read on any location of the channel. Similar to *pop*, the *RMV* instruction "consumes" the data in the memory location by invalidating its contents. The *RD* instruction is identical to the FIFO *peek* operation. It reads the contents of a channel memory location without invalidating its contents. Both the instructions on execution broadcast the contents of the channel location to all other RSBs.

Each location in the channel is extended with a valid bit that is used to determine whether or not the data in the location has been invalidated by a *RMV* instruction following a write. Clearly, a write to a location sets the valid bit and a *RMV* instruction resets it.

Field	Description
RS tag	Tag of instruction producing the data in the producer SE
Iteration low	Lower limit of actor iteration for which push instruction is active
Iteration high	Upper limit of actor iteration for which push instruction is active
Consumer SE	Address of consumer SE
Channel Start	Start address of the consumer SE channel where data is written
Channel End	Start address of the consumer SE channel where data is written
Stride	The stride of the index at which the data is written
Index	Current index from channel start address at which data is written
Credit	Start address of the consumer SE channel where data is written

Table 3.1: Description of Push Unit Reservation Station Fields

In the stream computation model, the *peek* and *pop* instructions follow a regular pattern across successive iterations of the actor. Therefore, in a vast majority of the cases it is possible to statically determine the address of the RD and RMV instructions at compile time. Consequently, many of the read instructions can be executed in parallel, and the stream RSBs exploit this parallelism.

3.3.2 The Push Unit

The push unit, as the name suggests, implements the FIFO *push* operation. The push unit is also composed of tagged RSBs where each RS entry holds a push instruction. The push instruction implements split/join operations which are quite common in streaming applications. The split/join operations in the stream computation model depict regular patterns, and push instructions provide support for them. The structure of the push unit RSB is different than the other blocks. The various fields of the

RS tag	Iteration	Iteration	Consumer	Channel	Channel	Stride	Index	Credit
	low	high	SE	start	end			
16	0	63	1	0	255	1	31	4
16	64	127	2	0	255	1	0	4
16	128	191	3	0	255	1	0	4
16	192	255	4	0	255	1	0	4

Table 3.2: Push Unit Contents for Weighted Round-robin Split E.g.

push unit RSB and their descriptions is given in Table 3.1. The push instructions are explained with the help of two examples in the following paragraphs.

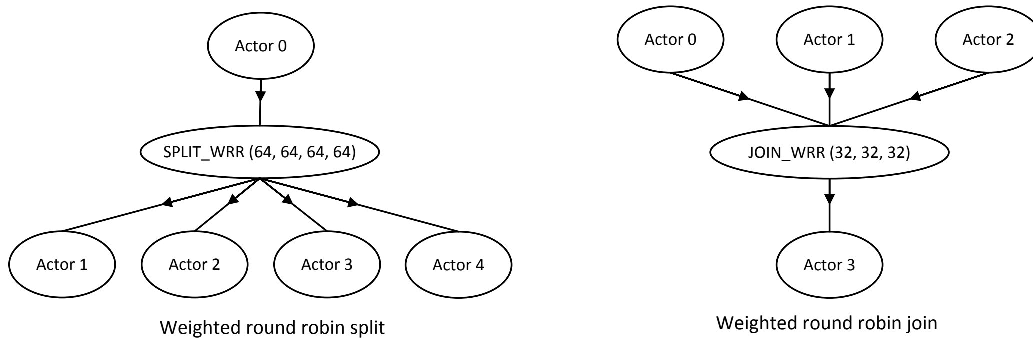


Figure 3.2: Weighted Round-robin Split and Join Examples

Figure 3.2 depicts examples of stream programs that include weighted round robin split and join operations, respectively. In the split operation the output stream from actor 0 is split across actors 1 through 4 in a weighted round robin manner. The first 64 data items of the stream from actor 0 are sent to actor 1, the next 64 data items to actor 2 and so on. Correspondingly, in the join operation the first 32 data items are consumed from actor 0, the next 32 data items from actor 1 and so on. We assume that the kernels are programmed on SE whose ID is identical to the actor ID. That is, actor 1 is mapped to SE 1 and so on.

Table 3.2 shows the contents of the push unit RSB along with the multiple push instructions for the weighted round robin split example. The producer instruction

	RS tag	Iter low	Iter high	Consumer SE	Channel start	Channel end	Stride	Index	Credit
Producer SE 0	8	0	31	3	0	31	1	0	1
	8	32	63	3	96	127	1	96	1
Producer SE 1	12	0	31	3	32	63	1	32	1
	12	32	63	3	128	159	1	128	1
Producer SE 2	9	0	31	3	64	95	1	64	1
	9	32	63	3	160	191	1	160	1

Table 3.3: Push Unit Contents at Producer SE for Weighted Round-robin Join Example

for the various push instructions has RS tag as 16. The first row represents the push instructions for actor 1 that is mapped to SE 1 (denoted by entry in consumer SE field). This particular push instruction is valid for iterations 0 to 63 (inclusive of the limits) of actor 0 as denoted by iteration low and iteration high fields of the first RS. The push instruction writes its data starting from channel address 0 of the consumer SE. Index field in the example denotes the current location (31) to which the push instruction will write. After each execution of the push instruction, index is incremented by stride (1 in our example). Finally, index wraps around to the start channel address when its value becomes greater than the end channel address (255 in the example). The last entry in the RS represents the credit that the particular push instruction has, and it is utilized to implement a credit based inter SE communication scheme. The scheme itself is discussed in the following section. The weighted round robin split operation is performed by 4 push instructions in the RSB shown in Table 3.2. Notice that the four instructions are valid for non-overlapping iterations of actor 0 and the target SE are also different for each instruction.

Table 3.3 depicts the contents of the weighted round robin join example shown on the right hand side of Figure 3.2. As the join operation is performed by the three

Field	Description
Start address	Start address of channel
End address	End address of channel
RMV and *RD* count	Number of *RMV* or *RD* instructions performed
Maximum count	Total number of *RMV* or *RD* instructions to be performed for sending the credit back
Producer SE	Address of producer SE
RS tag	RS tag of push instruction in producer SE

Table 3.4: Description of Credit Reservation Station Block

actors mapped on distinct SEs, the table gives the RS entries for push units on each SE (as denoted by the first column). The push instructions in SE 0 (which hosts actor 0) write to the channel of SE 3 (actor 3) at addresses 0 to 31 and 96 to 127. Similarly SEs 1 and 2 write at address ranges (32 to 63, 128 to 159) and (64 to 95, 160 to 191), respectively. Notice that the total size of the FIFO as virtualized on the SE 3 channel is 192, and each actor writes at two distinct ranges within the 0 to 191 address space. The push unit along with the channel is effectively able to virtualize multiple software FIFOs belonging to the streaming application on limited hardware space.

3.3.3 Credit-based Back Pressure

The push unit also implements a credit-based push back scheme for inter-SE streaming communication. The credit based schemes effectively extends the dataflow execution model across multiple SEs. Each push instruction in the push unit RSB is initialized with a given amount of credits. Once the push instruction index rolls over from iteration high to iteration low, the credit is decremented by one. In other words the credit is not decremented for each execution of the push instruction. Rather, it is decremented after the index has covered the entire range once. The coarse granularity of credit deduction reduces the inter-SE credit communication bandwidth. Further,

it also enables the various actors to execute in largely asynchronous manner, and effectively limits the impact of isolated instruction stalls and stream data bursts. The push RS stalls the push instruction once credit reaches zero. It also sends a signal to the dataflow monitor to stall the instruction that is the producer instruction for the push RS.

The input channel unit for the consumer SE is responsible for sending a credit back once the data has been consumed at the channel. The channel also maintains a RSB whose contents mirror those at the push unit. Table 3.4 describes the fields in the channel credit RSB. The start and end address denote the range of locations that constitute the credit. The channel RSB keeps count of the successful RMV or RD instructions that are performed on the locations within the address range. Once the total number of instructions is equal to the maximum count value, the channel RSB sends a credit back to the push instruction (specified by the RS tag) in the producer SE (whose address is also stored).

3.4 Evaluation

In this section, we evaluate the FIFO virtualization implemented by the push unit and input channel pairs, and its impact on inter-SE communication. We evaluate our communication mechanism against the IBM Cell BE Pham *et al.* (2006) architecture which also utilizes SPMs for the cores in the dataplane (also called synergistic processing units or SPU). Each SPU in Cell BE is equipped with a DMA engine to offload the inter-core communication. We evaluate our communication mechanism against three of the widely used software managed and DMA based inter-core communication schemes. Performance optimization on the IBM Cell BE requires that the application be software pipelined across the SPUs. Hence, the three schemes described below dis-

cuss the execution of the actor and the corresponding DMA to the consumer actor with respect to the steady state of the software pipeline. The three schemes are:

- *Scheme 1:* The kernel or actor executes for a particular iteration, writes the result to a buffer and then initiates the DMA to the destination core. Thus, the kernel execution and DMA occur sequentially in the same iteration of the software pipeline.

- *Scheme 2:* This scheme also known as double buffering attempts to hide the DMA overhead associated with a particular transfer. In this scheme twice the amount of memory required by the previous scheme is allocated for a particular DMA operation. In the steady state of the pipeline, the kernel execution for one iteration overlaps with the DMA operation that transfers data produced by the actor in the previous iteration.

- *Scheme 3:* This scheme, also known as block processing, attempts to amortize the DMA overhead by transferring data belonging to several iterations of the kernel. Thus, in the steady state of the software pipeline the kernel will fire several times. The DMA will also concurrently transfer data produced by the multiple executions of the kernel. However, similar to double buffering the DMA transfers data produced in the previous iteration of the software pipeline.

Scheme 1, 2 and 3 are arranged in ascending order of the buffer usage, and in descending order of typically observed throughput. We obtained optimized implementations of the application level benchmarks on the IBM Cell BE architecture (with 1 PowerPC and 6 SPUs)[1] and a cluster of 8 SEs.

[1]We utilized the Playstation3 platform which allows the programmer to only access 6 SPUs

3.4.1 Experimental Setup

Table 3.5 lists the stream application benchmarks that were utilized for the evaluation. All the evaluations are based on cycle-accurate simulations using the SystemC model. In addition to the SE modeled in the previous chapter, routers and the NoC was also modeled to capture the system-level behavior. Eight SEs were grouped together to form a StreamCluster. The router modeled was a 9 by 9 router that connects 8 SEs with each other within a single hop. The 9th port is used to connect to the next hierarchy of router that connects clusters together within 2 hops.

The DMA schemes were implemented on the Playstation 3 (IBM Cell BE architecture) running Fedora.

Benchmark	**Description**	**Kernels**	**Split/join pairs**
Autocor	Produces an autocor series of length 1 for a series of vectors of length 32	8	1
FFT	256-point single precision floating point Fast Fourier Transform	15	0
iDCT	8 x 8 2D inverse Discrete Cosine Transform	16	2
FM	Frequency modulation	26	2
MPEG	MPEG-2 Standard block decoding and motion vector decoding	21	4
DES	Data encryption standard	31	8

Table 3.5: Stream Application Benchmarks

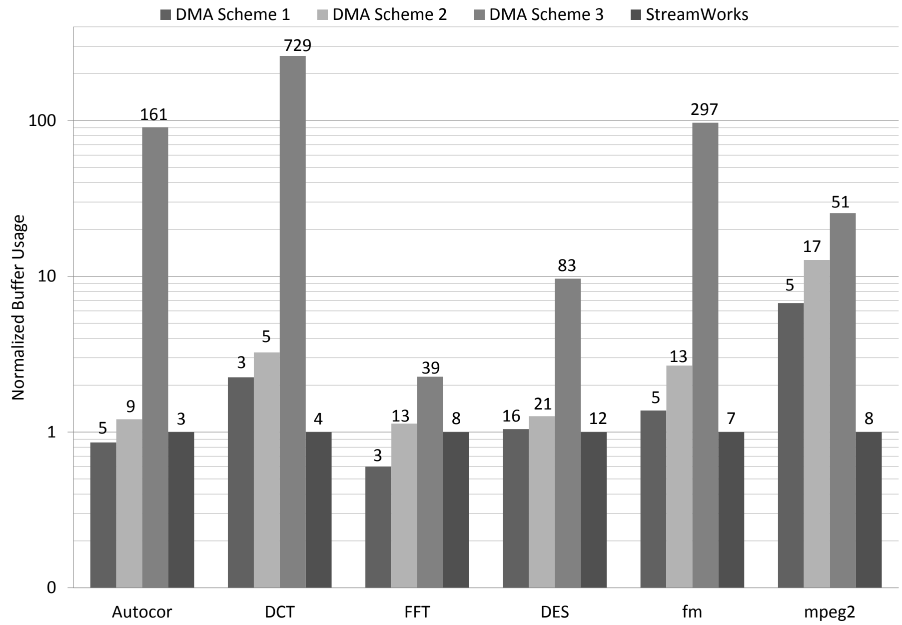


Figure 3.3: Buffer Requirement for the 3 DMA Schemes Normalized to the StreamWorks Buffer Usage and Software Pipeline Stages

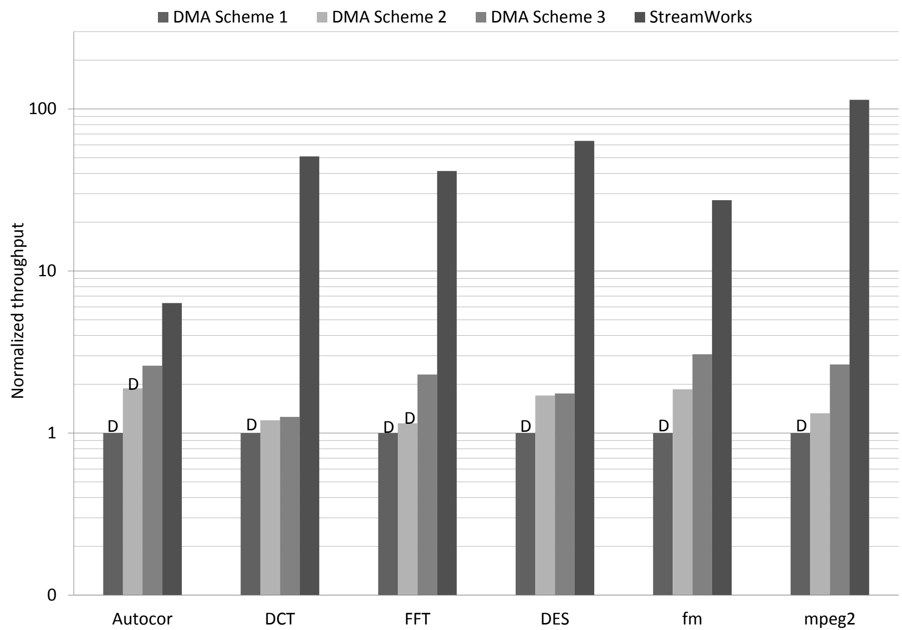


Figure 3.4: Normalized Throughput (in cc) for DMA Schemes

3.4.2 Results

Figure 3.3 shows the communication buffer requirement of the different DMA schemes normalized to the buffers utilized for communication in StreamWorks. The numbers on each of the bars denote the total number of software pipeline stages in the steady state of the implementation. A large number implies very deep pipelining and consequently larger design latency. It is evident from the figure that the StreamWorks' hardware assisted communication scheme enables efficient buffer management due to the finer granularity in data transfers.

Figure 3.4 depicts the normalized throughput (with base in data tokens produced per clock cycle) for the various applications when implemented on the Cell BE and the StreamWorks cluster. The 'D' on the various bars in the plot denote that the performance is limited by the DMA overhead of the design. As can be observed from the figure even with the most aggressive DMA amortization scheme, the StreamWorks cluster is able to outperform the Cell BE by 24.8X (standard deviation of 17.3)[2].

Finally, we evaluate the scaling of StreamWorks architecture across stream application-level benchmarks. We selected the 3 ALU 2 MUL configuration for each SE from the results obtained in Section 2.7.6. Each cluster had eight SEs and a single 9-port router whose free port was attached to the neighboring router. Figure 3.5 shows the normalized throughput of the applications as they scale with the number of clusters. It is evident from the results that the performance of the StreamWorks architecture for the set of benchmark applications scales up quite well as the number of clusters are increased. In the case of DCT the performance is super-linear as the congestion in the routers reduces due to introduction of additional clusters.

[2]The comparison is included only to highlight the benefit of FIFO virtualization. A thorough comparison with IBM Cell BE micro-architecture requires further studies

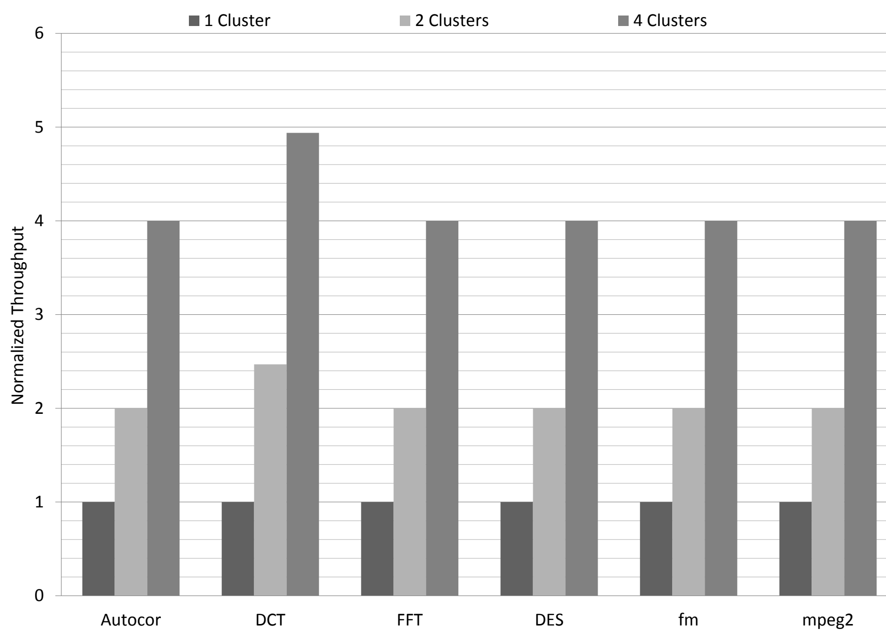


Figure 3.5: Performance as a Function of StreamWorks Clusters

Chapter 4

KERNEL OVERLAY AND SCALABILITY

In this chapter, the scalability of the StreamWorks architecture for large streaming applications is addressed. I first identify the limitations of the current architecture that impact scalability and then propose architectural support and software approaches to address the identified limitations. I then re-evaluate the StreamWorks architecture using the proposed techniques.

4.1 Limitations of the Current Architecture

One of the key factors that contributes towards the energy-efficiency of StreamWorks is the instruction delivery mechanism. The instruction locking mechanism is built upon the idea that the instruction fetch energy is amortized across several iterations of the kernel. This model involves fetching instructions from the next level of memory hierarchy (instruction store) just once and locking them close to functional units for the lifetime of the stream application. While this certainly holds for stream kernels that fit into the RSBs in SE, the case of over-sized kernels/applications still needs to be addressed.

The size of a RSB plays a key role in the energy-efficiency of the SE. The prototype of the StreamEngine that was synthesized contained only 48 reservation stations. Such a compact instruction store can handle most of the stream kernels because of their small code footprint. However, consider the following scenarios where the stream graphs that are mapped to execute on StreamWorks cannot be contained within the limited RSBs.

- The number of instructions in a stream kernel exceeds the total number of RSs in a SE
- The number of stream kernels in the stream graph exceed the total number of SEs in StreamWorks

Thus, without an efficient mechanism to support code size greater than the RSBs, the limited RSB size can restrict the scalability of the StreamWorks architecture.

4.2 Kernel Overlay

In the first of the scenarios described in Section 4.1, preserving the one-time RSB configuration could restrict the natural mapping of a stream kernel to a SE; larger kernels must be split across multiple SEs at appropriate boundaries. This can be achieved by redefining the kernel boundaries in such a way that each kernel now fits in a SE.

The second scenario, however, is trickier. One or more SEs will have to swap the kernel that is currently executing on the SE with other kernels that result from the spilling of the application. This violates the fundamental idea of locking instructions for the lifetime of an application.

The objective of the kernel overlay proposed in this section is to address the scalability of the StreamWorks architecture by allowing stream applications of any size to be mapped to StreamWorks without significantly sacrificing the energy-efficiency.

Kernel overlay for StreamWorks works in two phases, each addressing one of the two limitations highlighted in Section 4.1. The first phase uses a simple kernel fission mechanisms to split larger kernels into sub-kernels and map the resulting sub-kernels to SEs. The fission step ensures that each of the sub-kernels can be treated as an independent kernel for all mapping and RS allocation purposes. The second phase

identifies the kernesl that must be swapped and derives a schedule for the kernel overlay.

4.2.1 Kernel Fission

The kernel fission algorithm is applied to the entire stream graph (data flow graph of the stream application) and not to each of the over-sized kernels in isolation. The idea is to maximize RSB occupancy (improve average SE utilization) in each SE and minimize inter-SE communication (lower network traffic). The native stream graph that a programmer wants to map onto StreamWorks can have multiple over-sized kernels that might need fission. Upon fission of each candidate kernel, the ones that are marginally over-sized will result in two kernels; one of which utilizes very few RSs and therefore impacts the average SE utilization. Also, given a stream graph with n nodes, k of which are over-sized and fission candidates, a naive fission algorithm at kernel level will yield at least $n + 2 * k$ kernels, which requires $n + 2 * k$ SEs to map onto. The utilization problem arising from kernel-level fission can be addressed either by following a *fission-fusion* approach or by merging all the native kernels and then identifying new kernel boundaries.

The *fission-fusion* approach first splits all the over-sized kernels into two or more sub-kernels and then searches for candidate sub-kernels that can be fused together with other sub-kernels or native kernels that did not undergo fission.The kernel-based fission algorithm is heuristic in nature. The key objective is to identify fission points such that the intermediate data between the newly created sub-kernels is minimal subject to the constraint that the instruction count of each type of instructions is within the reservation station budget of SE.[1] Kernel fission essentially introduces new communication edges in the stream graph. Minimizing the intermediate data

[1]

that flows between the newly created sub-kernels attempts to avoid congestion in the system NoC.

Once the over-sized kernels are split using the fission algorithm, the *fission-fusion* approach attempts to fuse kernels to maximize RS occupancy. This can be solved by formulating an appropriate ILP Che and Chatha (2010); Choi *et al.* (2009); Kudlur and Mahlke (2008). However, it has been shown that simple and fast heuristics can achieve sub-optimal solutions that are close to optimal Che and Chatha (2011a) and thus can be incorporated in the runtime environment.

4.2.2 Overlay Schedule

In order to address the second limitation, the kernel overlay scheme determines a schedule for executing the kernels. In other words, it determines the point at which the SE should be re-configured with a kernel and change its context to the new kernel.

As discussed in Section 4.1, one of the major contributors towards the energy-efficiency of StreamWorks is the set of RSBs that eliminate instruction delivery overheads. The primary assumption is that the kernel instructions are loaded into the RSBs just once and the streaming nature allows infinite iterations of the kernel. Thus, there is only a one time instruction delivery (RSB configuration) cost that gets amortized across multiple kernel iterations. With such an assumption, delays and instruction movement energy associated with the one-time RSB configuration are acceptable. In the current implementation of the instruction unit (refer to Figure 2.1), which follows a serial approach, it can take up to 48 clock cycles (one clock per RS) to configure a 100% utilized StreamEngine. In general, the instruction unit takes x clock cycles to configure the SE, where x is the number of instructions in the kernel mapped to the SE.

Depending on the amount of DLP in a stream application, the programmer can choose to either run the same kernel concurrently on multiple SE's or establish a software pipeline among all the kernels. I first discuss the case of software pipeline that exploits the task level parallelism in the application and then address the case of executing same kernel on multiple SEs.

Consider the case where the total number of kernels (say n) is greater than the total number of SEs (say m) available. In order to preserve the software pipeline, the kernel deployment now needs to happen in *rounds*. Each *round* involves mapping all the kernels of the stream graph to SEs for a fixed number of kernel iterations. Note that since $n > m$, not all kernels can concurrently be mapped to SEs. However, the kernel to SE mapping can be determined based on the *round*. For example, in round 0, the kth kernel will be mapped to SE $k\%m$. In general, in round i, the kth kernel will be mapped to SE $((k\%m) + i \times s)\%m$, where s is the spill $n\%m$ of the kernels on StreamWorks. Note that the kernels in this context could be sub-kernels resulting from a kernel fission.

Table 4.1 and Table 4.2 demonstrate the mappings of 10 kernels to 4 SEs in Round 0 and Round 1 of deployment respectively, where k_i denotes the ith kernel. The stream graph has 10 kernels 0 through 9 and the available SEs are only 4 in this example.The numbers in rows 0, 2, 4 and 6 denote the iteration count in steady state execution of the kernel mapped to SE 0, SE 1, SE 2 and SE 3 respectively. As the table indicates, the steady-state in the software pipeline is preserved across *round* boundaries. Of course, there will be a prologue (iterations 0, 1 and 2 in this case) and epilogue (not shown) as with any other pipeline.

In steady-state execution, the throughput is dictated by the kernel with the highest latency. Going back to our example with n kernels on m SEs, by the time kernel $m-1$ is ready to execute iteration 0, kernel 0 has already executed $m - 1$ iterations. This

Iter	0	1	2	3	0	1	2	3	0	1
SE 0	k_0	k_0	k_0	k_0	k_4	k_4	k_4	k_4	k_8	k_8
Iter	-	**0**	**1**	**2**	**3**	**0**	**1**	**2**	**3**	**0**
SE 1	-	k_1	k_1	k_1	k_1	k_5	k_5	k_5	k_5	k_9
Iter	-	-	**0**	**1**	**2**	**3**	**0**	**1**	**2**	**3**
SE 2	-	-	k_2	k_2	k_2	k_2	k_6	k_6	k_6	k_6
Iter	-	-	-	**0**	**1**	**2**	**3**	**0**	**1**	**2**
SE 3	-	-	-	k_3	k_3	k_3	k_3	k_7	k_7	k_7

Table 4.1: Kernel Deployment Round 0

Iter	2	3	4	5	6	7	4	5	6	7
SE 0	k_8	k_8	k_2	k_2	k_2	k_2	k_6	k_6	k_6	k_6
Iter	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**4**	**5**	**6**
SE 1	k_9	k_9	k_9	k_3	k_3	k_3	k_3	k_7	k_7	k_7
Iter	**4**	**5**	**6**	**7**	**4**	**5**	**6**	**7**	**4**	**5**
SE 2	k_0	k_0	k_0	k_0	k_4	k_4	k_4	k_4	k_8	k_8
Iter	**3**	**4**	**5**	**6**	**7**	**4**	**5**	**6**	**7**	**4**
SE 3	k_7	k_1	k_1	k_1	k_1	k_5	k_5	k_5	k_5	k_9

Table 4.2: Kernel Deployment Round 1

could be a potential overlay point for SE0, where SE 0 could be reconfigured with the instructions of the mth kernel.

Note that the configuration overhead is still amortized across multiple iterations. However, the span of iterations across which the kernel code must be swapped with

subsequent kernels is now limited by the number of SEs. In other words, each SE needs to be configured for every m iterations of the stream kernels. Note that m iterations of the stream kernels translates to the $m \times c$ clock cycles, where c is the number of cycles required to complete one iteration of the slowest kernel that is currently mapped onto StreamWorks.

The above overlay schedule preserves the software pipeline of the application at the cost of reduced instruction reuse. When the number of SEs is low, the above scheme might not be able to achieve significant energy savings from instruction locking. However, one consequence of fewer SEs is the time to reach steady state execution in the software pipeline. This can be exploited to improve the instruction reuse. For StreamWorks configurations with fewer SEs, a different strategy is followed to determine the overlay point. Going back to our example of n kernels and m SEs, SE 0 now needs to be reconfigured only when the buffer of SE 0 gets full as a result of accumulating output tokens of the kernel mapped on to SE $m - 1$. This will significantly increase the number of iterations for which the SEs can lock the instructions in the RSBs. Note that unlike the previous scheme where the reconfiguration point for each SE is different, all the SEs must be reconfigured before the pipeline with the new kernels is established. This scheme can clearly be applied only for configurations with low SE count. This overlay scheme can also be applied to the case when the programmer chooses to deploy the same kernel on all the SEs.

Table 4.3, Table 4.4 and Table 4.5 show the deployment of the 10 kernels k_0 through k_9 on the 4 available SEs. Note that for each round there is a prologue and an epilogue for the software pipeline that was setup among the mapped kernels.

Iter	**0**	**1**	**2**	**3**	**4**	**5**	**6**	-	-	-
SE 0	k_0	k_0	k_0	k_0	k_0	k_0	k_0	-	-	-
Iter	-	**0**	**1**	**2**	**3**	**4**	**5**	**6**	-	-
SE 1	-	k_1	k_1	k_1	k_1	k_1	k_1	k_1	-	-
Iter	-	-	**0**	**1**	**2**	**3**	**4**	**5**	**6**	-
SE 2	-	-	k_2	k_2	k_2	k_2	k_2	k_2	k_2	-
Iter	-	-	-	**0**	**1**	**2**	**3**	**4**	**5**	**6**
SE 3	-	-	-	k_3	k_3	k_3	k_3	k_3	k_3	k_3

Table 4.3: Kernel Deployment Round 0

Iter	**0**	**1**	**2**	**3**	**4**	**5**	**6**	-	-	-
SE 0	k_4	k_4	k_4	k_4	k_4	k_4	k_4	-	-	-
Iter	-	**0**	**1**	**2**	**3**	**4**	**5**	**6**	-	-
SE 1	-	k_5	k_5	k_5	k_5	k_5	k_5	k_5	-	-
Iter	-	-	**0**	**1**	**2**	**3**	**4**	**5**	**6**	-
SE 2	-	-	k_6	k_6	k_6	k_6	k_6	k_6	k_6	-
Iter	-	-	-	**0**	**1**	**2**	**3**	**4**	**5**	**6**
SE 3	-	-	-	k_7	k_7	k_7	k_7	k_7	k_7	k_7

Table 4.4: Kernel Deployment Round 1

4.2.3 Implementation

Efficient kernel overlay will not only allow mapping large applications, but also enable temporal and spatial sharing of the StreamWorks architecture among concurrent stream applications. As discussed in Section 4.2.2, depending on the configuration

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	-
SE 0	k_8	k_8	k_8	k_8	k_8	k_8	k_8	k_8	k_8	-
	-	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
SE 1	-	k_9	k_9	k_9	k_9	k_9	k_9	k_9	k_9	k_9
	7	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	-
SE 2	k_0	k_0	k_0	k_0	k_0	k_0	k_0	k_0	k_0	-
	-	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
SE 3	-	k_1	k_1	k_1	k_1	k_1	k_1	k_1	k_1	k_1

Table 4.5: Kernel Deployment Round 2

of StreamWorks that is under consideration, one of the two overlay schemes can be applied to dynamically change the mapping of kernels on StreamWorks. For the convenience of the following discussion, lets name the overlay schemes as follows

- **Overlay Scheme I (OS-I)**: Preserve the software pipeline created among the mapped kernels by replacing only the first mapped kernel.(Illustrated in Table 4.1 and Table 4.2)
- **Overlay Scheme II (OS-II)**: Continue executing all the mapped kernels until the buffer of the first kernel gets filled by the results of the last kernel in the pipeline. (Illustrated in Table 4.3, Table 4.4 and Table 4.5)

It is worth mentioning at this point that the implicit assumption for OS-II is that the output of all the kernels 0 through $m-2$ mapped on the $m-$ SEs is consumed by a following kernel in the pipeline. However, when OS-II is used for data parallel execution of the same kernel on all SEs, the condition of the full buffer applies to each SE.

Let us now examine the architectural support (if any) required for the overlay schemes along with their pros and cons. As alluded to in Section 4.2.2, OS-I is useful when the pipeline established among the mapped kernels is deep and therefore both the epilogue and prologue penalties cannot be ignored especially when incurred multiple times across several deployments. Consequently, the number of SEs available is considerably high that enable such a deep pipeline. Given there are a significant number of SEs available to establish a deep pipeline, it is highly likely that the number of spilled kernels is less than the number of SEs.The assumption that the number of spilled kernels is less than the number of SEs only allows for optimizations. Later, overlay implementation without any assumptions is discussed that can be applied to arbitrary number of kernels.. In other words, all the kernels of the stream graph can be mapped onto $2 \times m$ SEs, where m is the total number of SEs which is equivalent to each SE hosting twice its size of kernel. This can be achieved by introducing what is known as *shadow reservation stations* or shadow RS. The idea is to host two kernels within a SE, only one of them being active at any time.

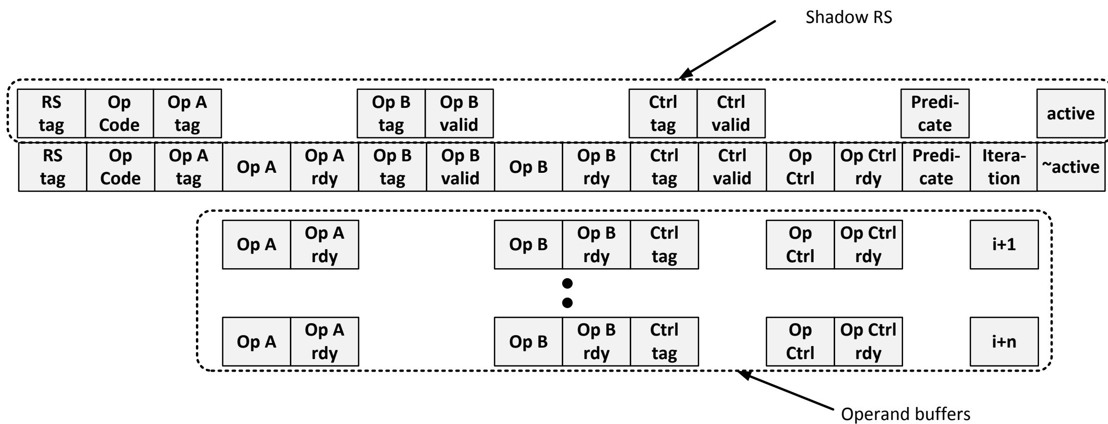


Figure 4.1: Structure of the Shadow RS

Figure 4.1 shows the structure of the enhanced reservation station along with its corresponding shadow RS. Note that only the configuration fields of the RS need to be replicated for a shadow RS and therefore, incurs very low area and power overheads. The *active* bit indicates which of the 2 RS is currently executing. Within a SE, the active bit can be considered as the kernel ID. Driving the active to 0, enables execution of kernel configured into the RS (kernel 0) while driving *active* to 1 enables the kernel residing in the shadow RS (kernel 1). Allowing 2 kernel configurations residing concurrently enables instantaneous context switch. As pointed out above, the key assumption is that $n <= 2 \times m$, where n and m are the number of kernels in the stream graph and number of available SEs in StreamWorks. This, however, is a reasonable assumption given we have enough SEs and would prefer to maintain the steady-state in the pipeline.

The OS-II is a more generic scheme that sacrifices the steady state of the established kernel pipeline in order to achieve kernel switch. The idea is to utilize the local buffer (scratchpad memory) to maximize the number of iterations any kernel can execute before being switched by another kernel. This amortizes the kernel(context) switch cost across number of iterations. Clearly, this is a function of the local buffer size. Conceptually, this is similar to the DMA Scheme 3 discussed in Section 3.4, the difference being, here the kernel fetch cost is amortized across iterations as opposed to the DMA cost being amortized in the case of DMA Scheme 3. OS-II scheme doesn't require any additional architectural support within the SE.

4.3 Evaluation

In this section, I demonstrate how the overlay schemes impact the energy-efficiency of the SE architecture. For OS-I, the shadow RS was implemented as an extension of the RS for power characterization. For OS-II, the assumption is that the StreamWorks

shares the LLC with the host processor. In present day architectures, this is a valid assumption as heterogeneous architectures are moving towards sharing the memory space.

4.3.1 Experimental Setup

	Configuration
Bitwidth	32
Input Channel ports	2
Input Channel capacity	2kB
SPM capacity	2kB
SIGs	4
ALUs	3
Multipliers	2
ALU/Multiplier pipeline depth	4
Stream RSBs	2
Compute RSBs	5
LD/ST RSBs	2
RS per ALU RSB	8
RS per MUL RSB	4
RS per Stream RSB	4
RS per LD/ST RSB	4
Operand buffers per RS	5

Table 4.6: StreamEngine Configuration

Power characterization of the shadow RS was done using Synopsys Primetime PX on the gate-level netlist generated from synthesis of the RTL VHDL description. The design flow followed is similar to the one illustrated in Figure 2.15. The configuration of SE used for the characterization of shadow RS as well as characterizing OS-II is given in Table 4.6. As the table suggests, the SE has 48 RSs, each with its own shadow RS. Only the Input Channel was considered as the buffer for OS-II. Consequently, only 2kB was used as the buffer size that needs to be filled at which point the kernel is swapped.Since the configuration energy of the SE depends on the kernel that is being mapped onto it, for worst case analysis, it was assumed that for each reconfiguration of the SE in OS-II, all the 48 RSs needed to be reconfigured. Further, each RSS configuration word was assumed to be 128-bits with 32-bit bus used for reconfiguration. Essentially, 192 ($48 \times (128/32)$) clock cycles were assumed for every SE reconfiguration.

4.3.2 Results

Figure 4.2 shows the energy expended to deliver a single instruction to the compute pipeline. The different components that were included towards the instruction delivery energy in the cache-based LEON were the cache array, the cache tags, the cache controller and the pipeline registers. In case of SPM-based LEON, only the scratchpad and pipeline registers were included. The RSBs are the instruction store for SE and hence all the RSBs account towards the total instruction delivery energy. For OS-II (SE w LLC access), additional cache access energy was included to account for the kernel fetch across multiple iterations. As the figure suggests, introducing shadow RS doesn't increase the energy significantly. This was expected, as the shadow RS are light weight extension of the RS involving only the static configuration bits. For OS-II, however, the access to LLC does have a significant impact on the

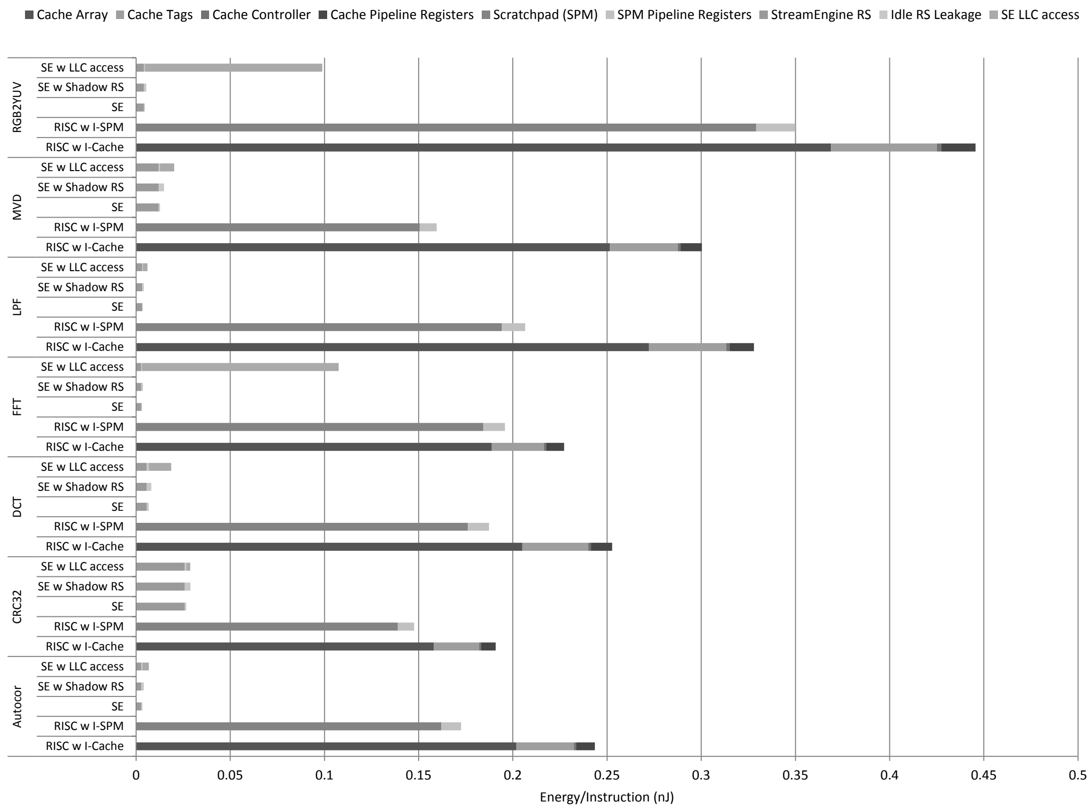


Figure 4.2: Instruction Delivery Energy Comparison

energy expended. When compared to LEON3, the SE performs better on both the overlay schemes. SE with shadow RS expends up to 82× less energy than LEON3 with cache with an average of 49.1×. With SE implementing OS-II, the SE spends up to 54.6× less energy than the LEON3 w cache with an average of 18.9×

The total energy consumption of LEON3 and SE with different configurations is shown in Figure 4.3. The total energy of the LEON cores and SE with overlay schemes is normalized to that of SE without any overlay. Compared to LEON3 with instruction cache, SE with shadow RS expends, on an average, 280× less energy while SE with LLC access expends about 220× less energy. Note that in the case of FFT, SE with access to LLC expends only 87× less energy when compared to LEON3.

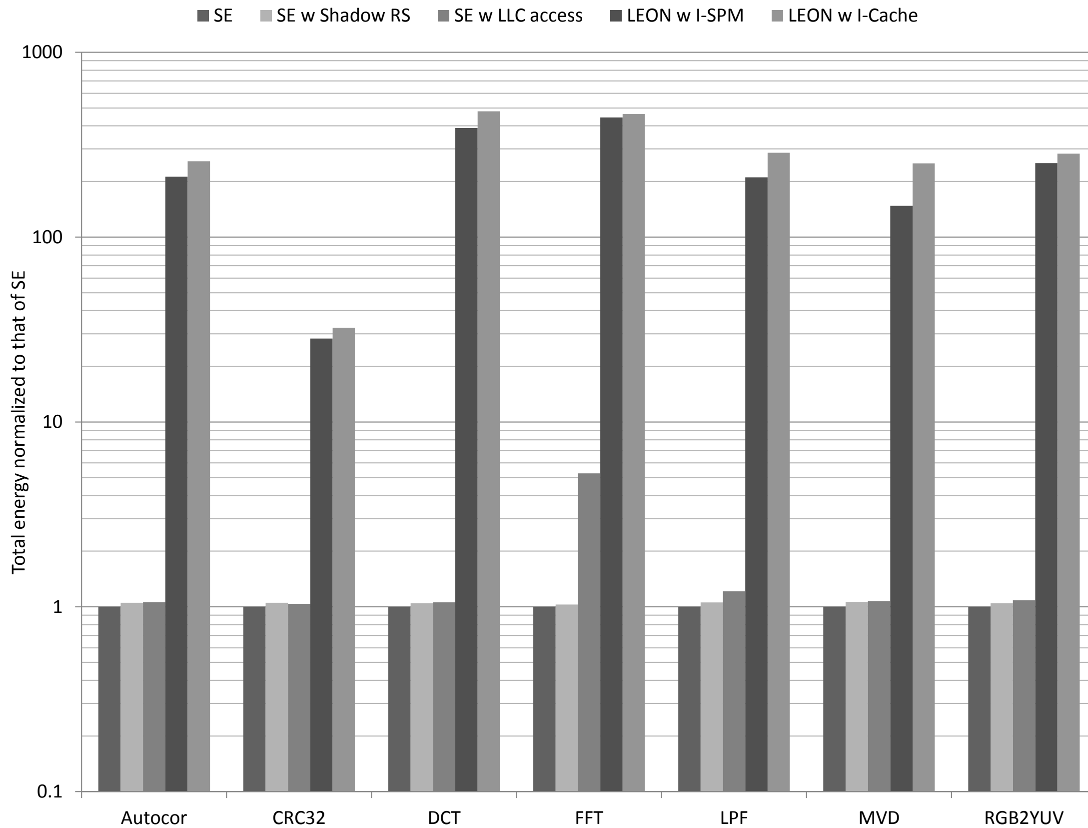


Figure 4.3: Total Energy Comparison of Various Overlay Schemes with RISC

This is particularly low when compared to the average. The reason behind the poor performance of SE using OS-II for FFT is the high throughput of the FFT kernel. In general , for high throughput kernels, the buffer gets filled much sooner and therefore restricts the total number of iterations across which the kernel switch energy can be amortized.

Figure 4.4 depicts the energy-efficiency in GFLOPS/W (GOPS/W for integer kernel benchmarks CRC32 and MVD) of the SPM-based LEON core and SE with and without overlay implementation normalized to that of a cache-based LEON core. Memory operations are not included in the GFLOPS calculation. As demonstrated by the figure, on an average, the SE with shadow RS is 232 times more energy-efficient

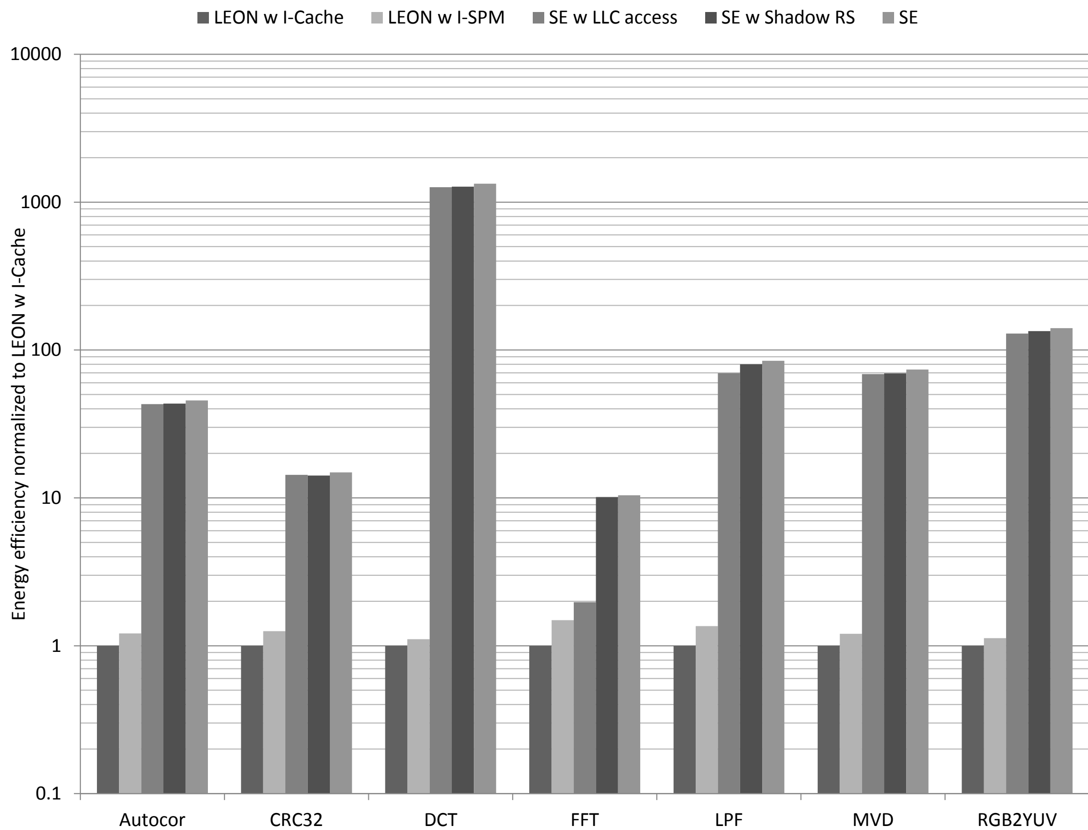


Figure 4.4: Energy-efficiency of SE Configurations Normalized to that of RISC

than LEON3 while SE implementing OS-II is 226× more energy efficient than the cache based LEON3. Again, SE with access to LLC performs worst for the high throughput FFT kernel.

Figure 4.5, Figure 4.6 and Figure 4.7 compare the instruction delivery energy, the total energy and the energy efficiency of the SE implementing the overlay schemes against the SE without overlay. As expected, the overhead of shadow RS is minimal and therefore SE with shadow RS marginally differs from SE without any overlay. Accessing LLC for OS-II, on the other hand, is much less energy-efficient than the OS-I scheme. I would like to highlight that the experimentation uses only 2kB of local memory and as pointed out earlier, the efficiency of OS-II is linearly dependent on

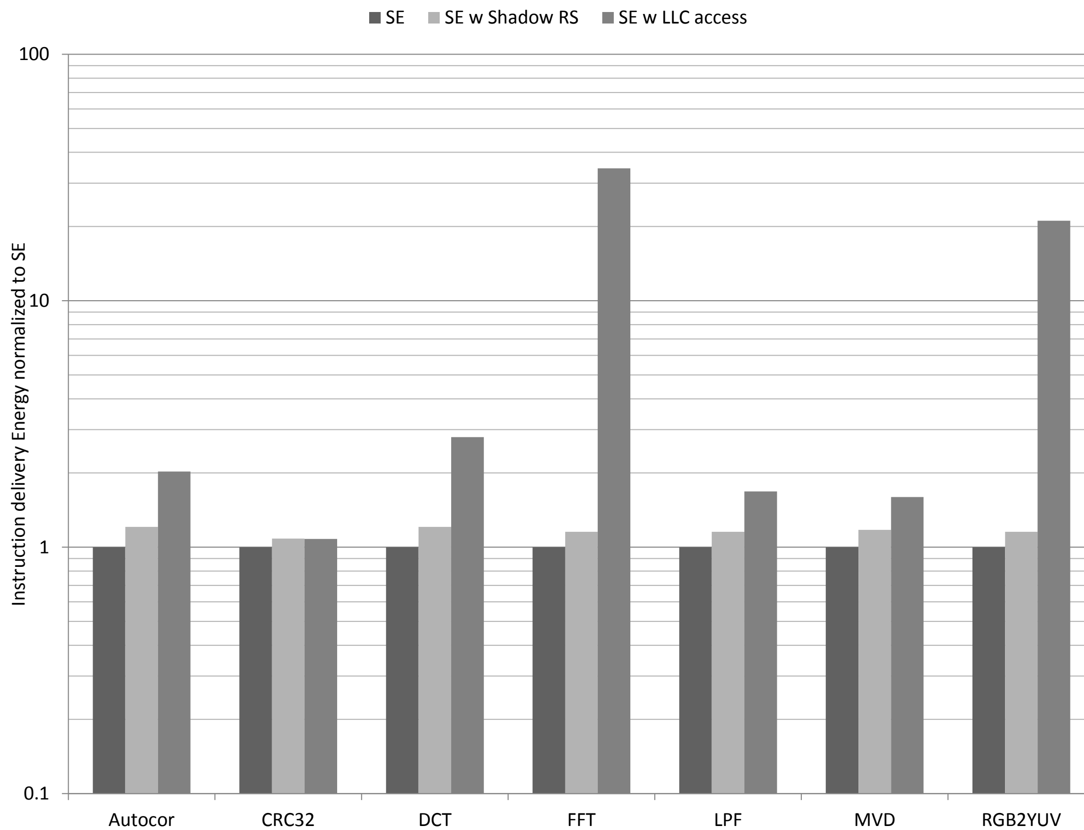


Figure 4.5: Instruction Delivery Energy of the Overlay Schemes Normalized to SE without Overlay

the amount of buffer (local storage). OS-II when adapted for embedded accelerators with higher local store will yield better energy efficiency. Also, OS-I comes with the limitation of restricting kernel migration whereas OS-II is a more general overlay scheme that can be applied under any circumstances.

4.3.3 Discussion

Implementing the OS-I scheme with a shadow RS restricts the flexibility of kernel migration from one SE to the other and consequently may limit the load balancing opportunities. The kernel mapping is statically decided and the SEs configured with their respective kernels. The runtime environment is responsible to utilize fusion and

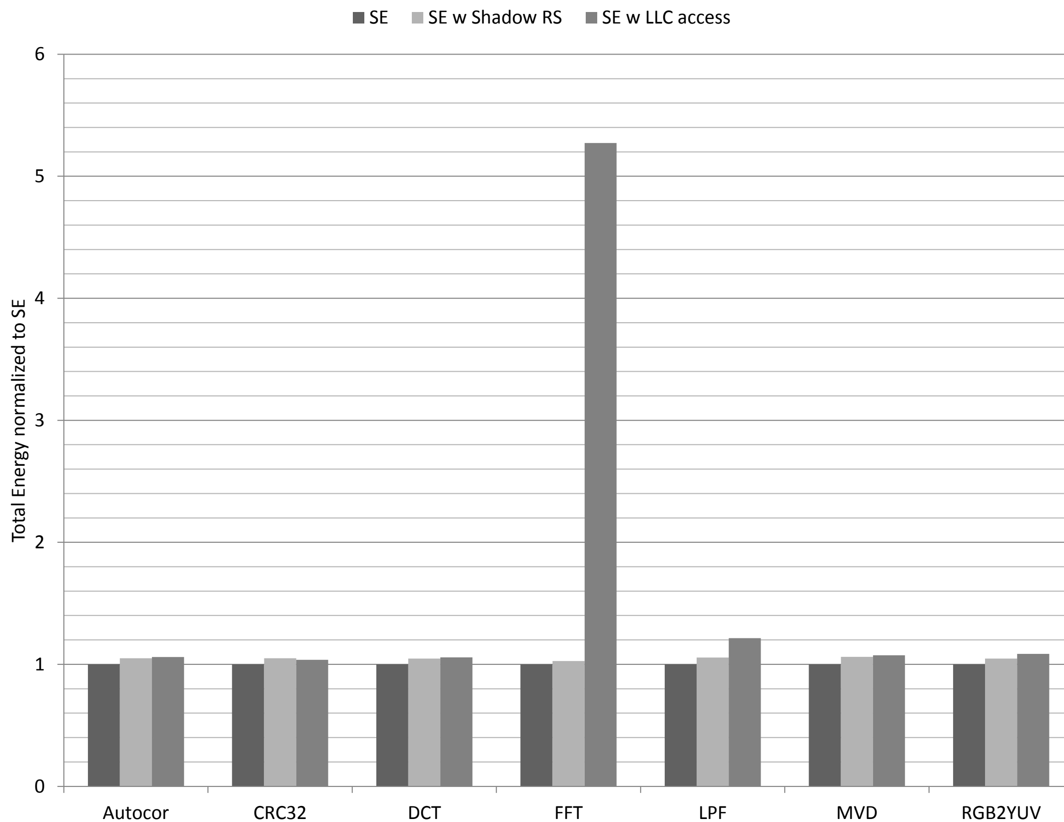


Figure 4.6: Total Energy of the OS-I and OS-II Normalized to that of SE without Overlay

fission mechanisms to create a stream graoh that can match the rate of the kernels with acceptable margins. This is a well studied problem and there are several ILP and heuristics that have been proposed to address the same Che *et al.* (2010); Che and Chatha (2012); Lee *et al.* (2012); Che and Chatha (2011b); Jung *et al.* (2010). Several light-weight schedulers Baker and Chatha (2010); Baker *et al.* (2010) have also been proposed that satisfy the aforementioned criterion. The programmer can adapt any of the existing schemes to iteratively fuse and split kernels to achieve rate matching and $2 \times m$ kernels for maximum utilization.

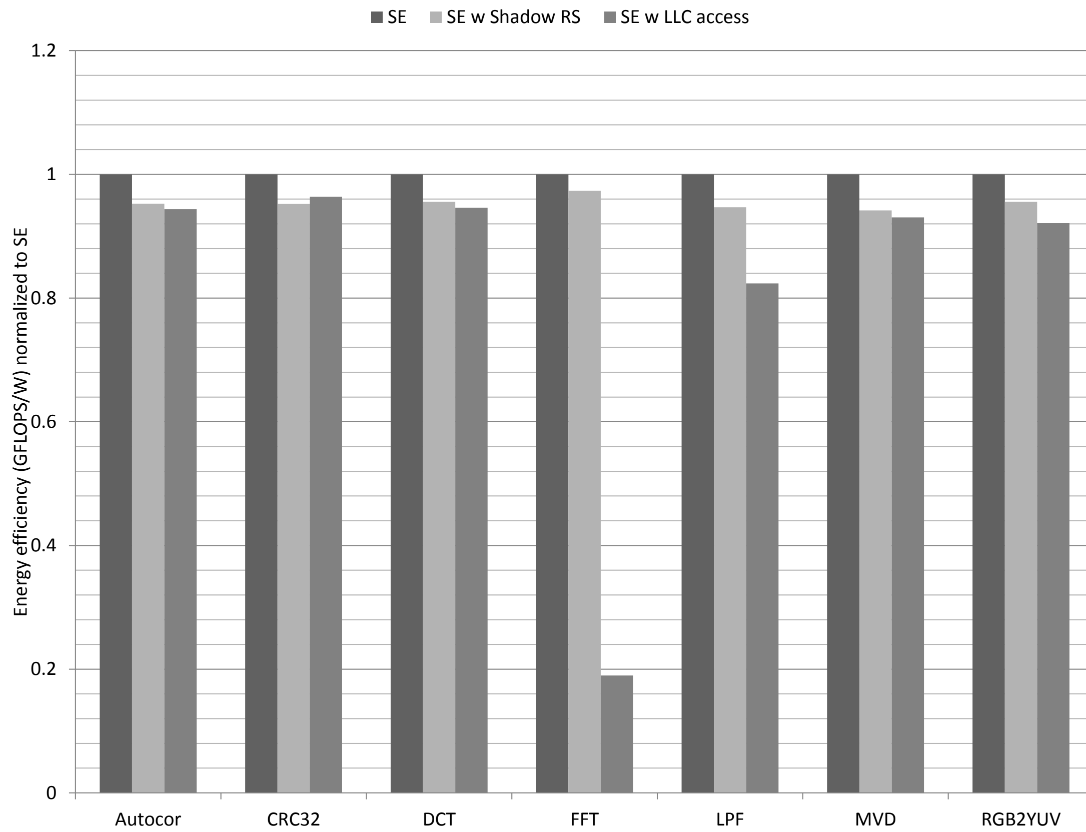


Figure 4.7: Energy-efficiency of the OS-I and OS-II Normalized to that of SE without Overlay

Chapter 5

CONCLUSION

The failure of Dennard scaling has not only curtailed single-core scaling but is soon limiting multi-core scaling as well. Given the constant chip-level power budgets, the percentage utilization of a chip has been decreasing with each technology generation and has led to the creation of dark silicon. Heterogeneous architectures that introduce specialized cores, each of which are tuned for specific application domains and $10-100\times$ more energy-efficient, have been identified as one of the promising solutions for dark silicon. With the emergence of streaming workloads for mobile devices, energy-efficient stream computing has become the need of the hour.

In this dissertation, I presented the StreamWorks architecture for energy-efficient stream computing. The processing element in StreamWorks is the StreamEngine that implements a context-aware data flow execution model and utilizes fine-grain instruction reuse to exploit the instruction locality in stream kernels. Instruction reuse coupled with the dataflow execution model eliminates the fetch and decode stages and thus achieves higher energy-efficiency. The energy-efficiency of StreamEngine was demonstrated against simple and contemporary RISC cores across a set of stream kernel benchmarks. The StreamWorks architecture also introduces a novel push/channel unit pair that includes specialized instructions for stream processing. These units include a credit-based communication scheme that can be utilized to scale the data flow architecture across multiple processing elements. As can be observed from the experimental results, StreamWorks architecture makes a compelling case for adopting a dataflow oriented approach for implementing streaming workloads.

Finally, the scalability of the StreamWorks architecture is addressed by discussing kernel fission and overlay schemes that enables mapping stream graphs of arbitrary size on to StreamWorks. The overlay techniques discussed are not specific to StreamWorks and can be adapted for any multi-core accelerator with software controlled local memories.

Arvind and D. E. Culler, “Dataflow architectures”, chap. Annual review of computer science vol. 1, 1986, pp. 225–253 (Annual Reviews Inc., Palo Alto, CA, USA, 1986).

Arvind, K. and R. S. Nikhil, “Executing a program on the mit tagged-token dataflow architecture”, IEEE Trans. Comput. **39**, 3, 300–318 (1990).

Baines, R. and D. Pulley, “The picoarray and reconfigurable baseband processing for wireless basestations”, in “Software Defined Radio”, (2004).

Baker, M. and K. Chatha, “A lightweight run-time scheduler for multitasking multicore stream applications”, in “Computer Design (ICCD), 2010 IEEE International Conference on”, pp. 297–304 (2010).

Baker, M., A. Panda, N. Ghadge, A. Kadne and K. Chatha, “A performance model and code overlay generator for scratchpad enhanced embedded processors”, in “Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2010 IEEE/ACM/IFIP International Conference on”, pp. 287–296 (2010).

Balfour, J., W. Dally, D. Black-Schaffer, V. Parikh and J. Park, “An energy-efficient processor architecture for embedded systems”, IEEE Comput. Archit. Lett. **7**, 1, 29–32 (2008).

Bilsen, G., M. Engels, R. Lauwereins and J. Peperstraete, “Cyclo-static data flow”, in “Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on”, vol. 5, pp. 3255–3258 vol.5 (1995).

Binkert, N., B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill and D. A. Wood, “The gem5 simulator”, SIGARCH Comput. Archit. News **39**, 2, 1–7, URL `http://doi.acm.org/10.1145/2024716.2024718` (2011).

Buck, I., “Brook: A Streaming Programming language”, (2001).

Buck, J. and E. Lee, “Scheduling dynamic dataflow graphs with bounded memory using the token flow model”, in “Acoustics, Speech, and Signal Processing, 1993. ICASSP-93., 1993 IEEE International Conference on”, vol. 1, pp. 429–432 vol.1 (1993).

Burger, D., S. W, K. S, M. Dahlin, L. K. John, C. Lin, C. R, J. Burrill, R. G and Yoder, “Scaling to the end of silicon with edge architectures”, IEEE Computer **37**, 44–55 (2004).

Che, W. and K. Chatha, “Scheduling of synchronous data flow models on scratchpad memory based embedded processors”, in “Computer-Aided Design (ICCAD), 2010 IEEE/ACM International Conference on”, pp. 205–212 (2010).

Che, W. 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", pp. 122–127 (2011a).

Che, W. and K. 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", pp. 9–18 (2011b).

Che, W. and K. S. Chatha, "Unrolling and retiming of stream applications onto embedded multicore processors", in "Proceedings of the 49th Annual Design Automation Conference", DAC '12, pp. 1272–1277 (ACM, New York, NY, USA, 2012), URL http://doi.acm.org/10.1145/2228360.2228598.

Che, W., A. Panda and K. Chatha, "Compilation of stream programs for multicore processors that incorporate scratchpad memories", in "Design, Automation Test in Europe Conference Exhibition (DATE), 2010", pp. 1118–1123 (2010).

Chen, I., A. Chun, E. Tsui, H. Honary and V. Tsai, "Overview of intel's reconfigurable communication architecture", in "International Workshop on Application Specific Processors", (2004).

Choi, Y., Y. Lin, N. Chong, S. Mahlke and T. Mudge, "Stream compilation for real-time embedded multicore systems", in "Code Generation and Optimization, 2009. CGO 2009. International Symposium on", pp. 210–220 (2009).

Dally, W., J. Balfour, D. Black-Shaffer, J. Chen, R. Harting, V. Parikh, J. Park and D. Sheffield, "Efficient embedded computing", Computer **41**, 7, 27 –32 (2008).

Dally, W. J., F. Labonte, A. Das, P. Hanrahan, J.-H. Ahn, J. Gummaraju, M. Erez, N. Jayasena, I. Buck, T. J. Knight and U. J. Kapasi, "Merrimac: Supercomputing with streams", in "Proceedings of the 2003 ACM/IEEE conference on Supercomputing", SC '03, pp. 35– (ACM, New York, NY, USA, 2003), URL http://doi.acm.org/10.1145/1048935.1050187.

Dennis, J. B. and D. P. Misunas, "A preliminary architecture for a basic data-flow processor", in "Proceedings of the 2nd annual symposium on Computer architecture", ISCA '75, pp. 126–132 (ACM, New York, NY, USA, 1975).

Geilen, M. and T. Basten, "Requirements on the execution of kahn process networks", in "Proc. of the 12th European Symposium on Programming, ESOP 2003", pp. 319–334 (Springer Verlag, 2003).

Govindaraju, V., C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam and C. Kim, "Dyser: Unifying functionality and parallelism specialization for energy-efficient computing", IEEE Micro **32**, 5, 38–51, URL http://dx.doi.org/10.1109/MM.2012.51 (2012).

Graphics, M., "Sourcery Codebench", URL http://www.mentor.com/embedded-software/sourcery-tools/sourcery-codebench/overview/ (2009).

Habnic, S. and J. Gaisler, “Status of the leon2/3 processor developments”, in “DASIA ’07: DAta Systems In Aerospace 2007”, (2007).

Johnson, E. J. and A. Kunze, *IXP2400/2800 Programming The Complete Microengine Coding Guide* (Intel Press, 2003).

Jung, S. C., A. Shrivastava and K. Bai, “Dynamic code mapping for limited local memory systems”, in “Application-specific Systems Architectures and Processors (ASAP), 2010 21st IEEE International Conference on”, pp. 13–20 (2010).

Kahn, G., “The semantics of a simple language for parallel programming”, in “Information processing”, edited by J. L. Rosenfeld, pp. 471–475 (North Holland, Amsterdam, Stockholm, Sweden, 1974).

Kapasi, U., W. J. Dally, S. Rixner, J. D. Owens and B. Khailany, “The Imagine stream processor”, in “Proceedings 2002 IEEE International Conference on Computer Design”, pp. 282–288 (2002).

Khailany, B., W. J. Dally, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, A. Chang and S. Rixner, “Imagine: Media processing with streams”, IEEE Micro (2001).

Kudlur, M. and S. Mahlke, “Orchestrating the execution of stream programs on multicore platforms”, in “In Proc. of the SIGPLAN 08 Conference on Programming Language Design and Implementation”, pp. 114–124 (2008).

Lee, E. A. and et al., “Synchronous data flow”, (1987).

Lee, E. A. and D. G. Messerschmitt, “Static scheduling of synchronous data flow programs for digital signal processing”, IEEE Trans. Comput. **36**, 1, 24–35, URL `http://dx.doi.org/10.1109/TC.1987.5009446` (1987).

Lee, H., W. Che and K. Chatha, “Dynamic scheduling of stream programs on embedded multi-core processors”, in “Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis”, CODES+ISSS ’12, pp. 93–102 (ACM, New York, NY, USA, 2012), URL `http://doi.acm.org/10.1145/2380445.2380465`.

Li, S., J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen and N. P. Jouppi, “Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures”, in “Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture”, MICRO 42, pp. 469–480 (ACM, New York, NY, USA, 2009), URL `http://doi.acm.org/10.1145/1669112.1669172`.

Lin, Y., H. Lee, M. Woh, Y. Harel, S. Mahlke and T. Mudge, “Soda: A low-power architecture for software radio”, in “Proceedings of International Symposium on Computer Architecture”, (2006).

Mei, B., S. Vernalde, D. Verkest, H. De Man and R. Lauwereins, “Adres: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix”, in “Field Programmable Logic and Application”, edited by P. Y. K. Cheung and G. Constantinides, vol. 2778 of *Lecture Notes in Computer Science* (2003).

Nikhil, R. S., G. M. Papadopoulos and Arvind, “*t: A multithreaded massively parallel architecture”, in “ISCA”, pp. 156–167 (1992).

Panda, A. and K. Chatha, “An embedded architecture for energy-efficient stream computing”, Embedded Systems Letters, IEEE **6**, 3, 57–60 (2014).

Parks, T., J. Pino and E. Lee, “A comparison of synchronous and cycle-static dataflow”, in “Signals, Systems and Computers, 1995. 1995 Conference Record of the Twenty-Ninth Asilomar Conference on”, vol. 1, pp. 204–210 vol.1 (1995).

Pham, D., T. Aipperspach, D. Boerstler, M. Bolliger, R. Chaudhry, D. Cox, P. Harvey, P. Harvey, H. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Pham, J. Pille, S. Posluszny, M. Riley, D. 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 (2006).

Rixner, S., W. Dally, B. Khailany, P. Mattson, U. Kapasi and J. Owens, “Register organization for media processing”, in “High-Performance Computer Architecture, 2000. HPCA-6. Proceedings. Sixth International Symposium on”, pp. 375–386 (2000).

Sankaralingam, K., S. W. Keckler, W. R. Mark and D. Burger, “Universal mechanisms for data-parallel architectures”, in “Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture”, MICRO 36, pp. 303– (IEEE Computer Society, Washington, DC, USA, 2003a).

Sankaralingam, K., R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler and C. R. Moore, “Exploiting ilp, tlp, and dlp with the polymorphous trips architecture”, in “Proceedings of the 30th annual international symposium on Computer architecture”, ISCA ’03, pp. 422–433 (ACM, New York, NY, USA, 2003b), URL `http://doi.acm.org/10.1145/859618.859667`.

Singh, H., M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh and E. M. C. Filho, “Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications”, in “IEEE Transactions on Computers”, vol. 99 (2000).

Swanson, S., K. Michelson, A. Schwerin and M. Oskin, “Wavescalar”, in “Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture”, MICRO 36, pp. 291– (IEEE Computer Society, Washington, DC, USA, 2003), URL `http://dl.acm.org/citation.cfm?id=956417.956546`.

Taylor, M. B., J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman,

V. Strumpen, M. Frank, S. Amarasinghe and A. Agarwal, "The raw microprocessor: A computational fabric for software circuits and general-purpose programs", IEEE Micro **22**, 2, 25–35, URL http://dx.doi.org/10.1109/MM.2002.997877 (2002).

Thies, W., M. Karczmarek and S. Amarasinghe, "Streamit: A language for streaming applications", in "Proceedings of International Conference on Compiler Construction", (2002).

Tomasulo, R. M., "An efficient algorithm for exploiting multiple arithmetic units", IBM J. Res. Dev. **11**, 1, 25–33 (1967).

Uht, A., D. Morano, A. Khalafi and D. Kaeli, "Levo-a scalable processor with high IPC", Journal of Instruction-Level Parallelism **5**, 1–28 (2003).

Woh, M., Y. Lin, S. Seo, S. Mahlke, T. Mudge, C. Chakrabarti, R. Bruce, D. Kershaw, A. Reid, M. Wilder and K. Flautner, "From soda to scotch: The evolution of a wireless baseband processor", in "Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture", MICRO 41, pp. 152–163 (IEEE Computer Society, Washington, DC, USA, 2008), URL http://dx.doi.org/10.1109/MICRO.2008.4771787.

Woh, M., S. Seo, S. Mahlke, T. Mudge, C. Chakrabarti and K. Flautner, "Anysp: Anytime anywhere anyway signal processing", in "Proceedings of International Symposium on Computer Architecture", (2009).

Yu, Z., M. Meeuwsen, R. Apperson, O. Sattari, M. Lai, J. Webb, E. Work, T. Mohsenin and B. Baas, "Architecture and evaluation of an asynchronous array of simple processors", Journal of VLSI Signal Processing Systems (2008).

APPENDIX A

EVALUATION WITH GPU

A.1 EVALUATION OF SE WITH GPU SM AS BASELINE

A state-of-art GPU with Kepler GK110 architecture has been added as another baseline for comparison with the SE. I implemented the StreamIt kernel benchmarks on the GPU adopting different parallelization strategies and chose the implementation with highest performance and energy-efficiency for the comparison.

The parallelization strategy is important particularly for kernels like *Autocor* and *LPF* where the kernel uses MAC operations to maintain a running sum across the loop iterations. One approach is to assign each iteration of the loop to a thread and then use a single thread to sum the resulting product from each thread. This approach requires a *syncthreads()* operation before taking the sum.

A second approach is to assign each iteration to a thread and also to assign the final sum to multiple threads. With the first approach, the sum is done by a single thread which takes n cycles for n add operations (assuming one cycle per add). In this approach, the sum can be done in $log(n)$ cycles. However, this approach requires $log(n)$ *syncthreads()* and thus adversely affects the performance.

Another approach (bulk synchronous model) is to assign the entire kernel to a single thread and execute multiple kernel instances spanning across all the SIMT lanes in the GPU. This approach doesn't require any synchronization and was experimentally found to deliver maximum throughput.

I want to highlight that due to the throughput-oriented nature of GPU architectures, they are not suited for real-time streaming applications where only limited number of input streams are available at any time. SE, on the other hand, can operate as efficiently on single input stream as with multiple streams. In my experiment, I have used sufficient number of input streams to ensure maximum GPU utilization and thus maximum performance and energy-efficiency.

A.1.1 Experimental Setup

The kernel benchmarks were implemented in CUDA for evaluating the GPU. The GPU used was a Nvidia Tesla K20m card which is based on the Kepler GK110 architecture. The Kepler architecture is among the higher energy-efficient GPU architectures available. The kernel execution time was obtained using CUDA events and the NVML API was used to obtain the power and temperature readings available as hardware counters on the GPU card. Only the kernel execution time was taken into account for GOPS calculation, time taken for moving data to and from the host and GPU were excluded. The kernel execution time was averaged across multiple $(2k - 10k)$ runs.

As discussed above, maximum threads were used to ensure maximum GPU utilization. Allowing maximum threads resulted in the working data set to exceed the L1 cache in some cases. For such cases, I conducted another set of experiments using only as many threads that require data that fits into the L1 cache. The kernel execution latency certainly dropped, however, the throughput was higher in the former case even when the data exceeds the local cache. This again confirms the GPU's massively parallel execution model and its capability to hide memory latencies by exploiting thread -level parallelism.

With the objective of comparing the StreamEngine with a single Streaming Multiprocessor (SM, also known as SMX), a single thread block with maximum threads (1024) was used to ensure only one SM is active. One limitation in the exprimentation arises from the fact that the power number obtained using the hardware counters includes power drawn by SMs, memory and the memory controller. Since no data is availabe for the power breakdown in Kepler GPU cards, I used the power breakdown for the previous generation GT200 architecture as reported in a ISLPED 2012 publication. As per the breakdown, 40% of the total power is drawn by SMs and cache and the rest is drawn by memory and the memory controller. This breakdown assumes 35% memory bandwidth utilization.

I carried out a simple validation procedure to ensure that my estimates are reliable. The validation procedure involves running 2 sets of experiments for each kernel benchmark. One set spawns only one thread block and the other eight thread blocks thereby utilizing 1 SM and 8 SMs respectively. I used the following simple breakdown formula to estimate the power of 1 SM

$Total\ Power = x + 13 * y + N * z$, where x is the total power drawn by the memory and the memory controller, y is the static power of each SM, z is the dynamic power of each SM and N is the number of SMs utilized by the experiment (number of thread blocks spawned). x was calculated using the ISLPED estimate (60% of total power). For the estimate to be reliable, I expected the SM static power to be constant across the different benchmarks. The SM static power (averaged across the 7 benchmarks) was found to be 746mW with a standard deviation of only 3.5mW.

A.1.2 Results

Figure A.1 shows the performance of SE normalized to that of the Kepler GK110. As the figure demonstrates, SE delivers on an average 4.8× more GOPS than the GPU. SE scores over the GPU mainly due to the lack of explicit synchronization among threads. A typical stream kernel starts with input and output stream base pointers and maintains a pointer to the data token that it needs to read from the input stream and write to the output stream. In case of GPU, the base pointers passed to all the threads is the same (which is an argument to the kernel during invocation from the host). The threads then add their respective thread ids (as offsets) to read and write data. However before modifying the base pointer to the new base pointer of the streams all the threads need to reach a synchronization point after all the reads and writes are performed. In case of SE, no such synchronization is required as the SIGs take care of the correct index of the input stream and the push unit takes care of the output stream pointer.

As expected, SE doesn't perform well when compared to GPU in the case of CRC32 due to limited parallelism arising from the statefulness. However, GPU takes advantage of the thread level parallelism and its very wide architecture to execute 1024 threads operating on 1024 streams.

Figure A.2 shows the energy efficiency of SE normalized to that of the GPU. The SE is on an average 103.8 times more energy-efficient than the GPU with a standard deviation of 58.5. As expected, the wide SIMD lanes with multiple functional units in GPU account for the high power drawn thus affecting the energy-efficiency.

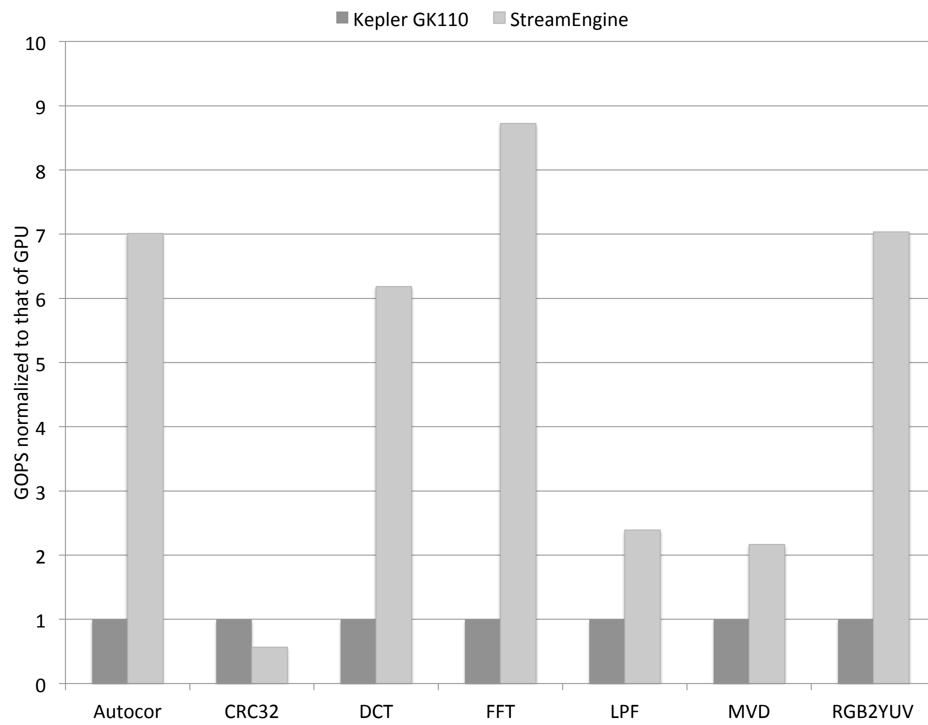


Figure A.1: Performance (Throughput) Comparison of SE with Kepler GK110

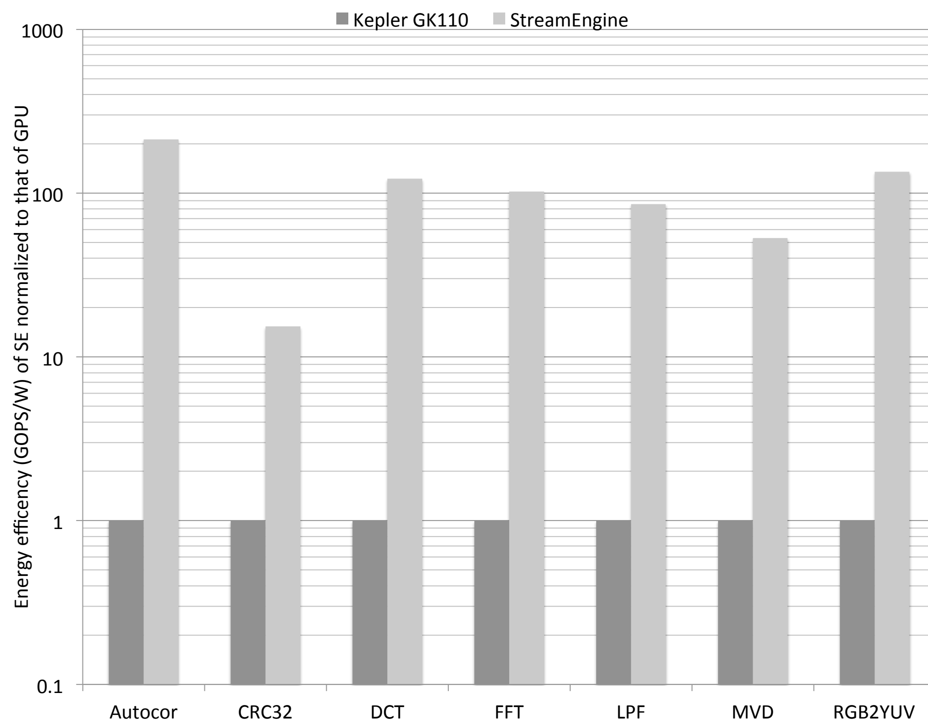


Figure A.2: Energy-efficiency Comparison of SE with Kepler GK110

A.2 EVALUATION OF SW WITH GPU AS BASELINE

For energy-efficiency comparison, the evaluation of StreamWorks against the Kepler GK110 based GPU with 13 SMs is presented below. This evaluation was performed during the comprehensive examination and due to time constraints, only 3 of the 6 application benchmakrs were evaluated. However, the 3 benchmarks exhibit different communication patterns and represent most of the other application benchmarks. *Autocor* benchmark uses 8 concurrent kernels without inter-kernel communication. *FFT* has 15 kernels with each kernel communicating to at most two other kernels (one input and one output). In other words, each kernel in FFT has input/output ports with single fan-in/fan-out. The third benchmark *iDCT* has 16 kernels and exhibits a point-to-point communication pattern. 8 of the 16 iDCT kernels have a fan-in of 8 and the other 8 have a fan-out of 8. The dataflow of the *iDCT* can be found in Figure 1.1 in the introduction of the proposal.

A.2.1 Experimental Setup

The benchmarks were implemented in CUDA to allow software pipelining on the Kepler architecture. Since there is no explicit hardware support for synchronization, atomic operations were used to implement global locks. The synchronization mechanism was optimized for minimum overhead. Instead of all the threads spinning on the global lock, a hierarchical locking mechanism was used. All the threads in a thread block spin on a local lock in the shared memory and only the thread with thread id 0 spins on the global lock competing with threads from other thread blocks (one thread per thread block).

Note: When concurrent thread blocks are spawned on the GPU, the Nvidia scheduler attempts to schedule as many thread blocks as possible until it runs out of resources (the order in which the blocks are scheduled is not deterministic). For example, in the Kepler GK110 architecure with 13 SMs, if I spawn 27 thread blocks with 1024 threads each, not all of them can be scheduled concurrently because the architecture limits the MAX threads on a SM to be 2048. In this case the number of threads becoms the bottleneck for scheduling. Similarly, the bottleneck could be the register usage, shared memory usage etc. The Nvidia scheduler always executes a thread block to completion and threfore might lead to deadlocks and timeout when some thread blocks are spinning on global locks and all the spawned thread blocks cannot be scheduled concurrently on the available SMs. For the evaluation, deadlocks have been avoided by ensuring no resouce spill occurs when scheduling all thread blocks.

A 2-cluster (16 SEs) configuration was used for the StreamWorks evaluation. However, for a fair comparison, all the kernels were scheduled only on 13 SEs. All the 13 SMs were used in the case of the GPU. The power characterization of the StreamWorks and GPU followed the same methodology as used in StreamEngine evaluation.

A.2.2 Results

Figure A.3 shows the performance (GFLOPS) comparison of the StreamWorks and the Kepler GK110. The performance of StreamWorks has been normalized to that of the GPU. As expected, the performance scales very well for *Autocor* as no

synchronization is used. However, for *iDCT* and *FFT*, a performance degradation of 28% and 70% was observed respectively as compared to when executing only a single kernel. The higher degradation in the case of *FFT* is most likely because of higher number of synchronization stages. Each kernel in the 15-stage pipeline of the FFT needs to synchronize once with its producer and again with its consumer (a total of 14 synchronization points) as opposed to *iDCT* which is implemented as a 2-stage pipeline with 8 concurrent (independent) kernels in each stage and requires only 8 synchronization points. StreamWorks, on an average, delivers 11.6× more GFLOPS than the GPU.

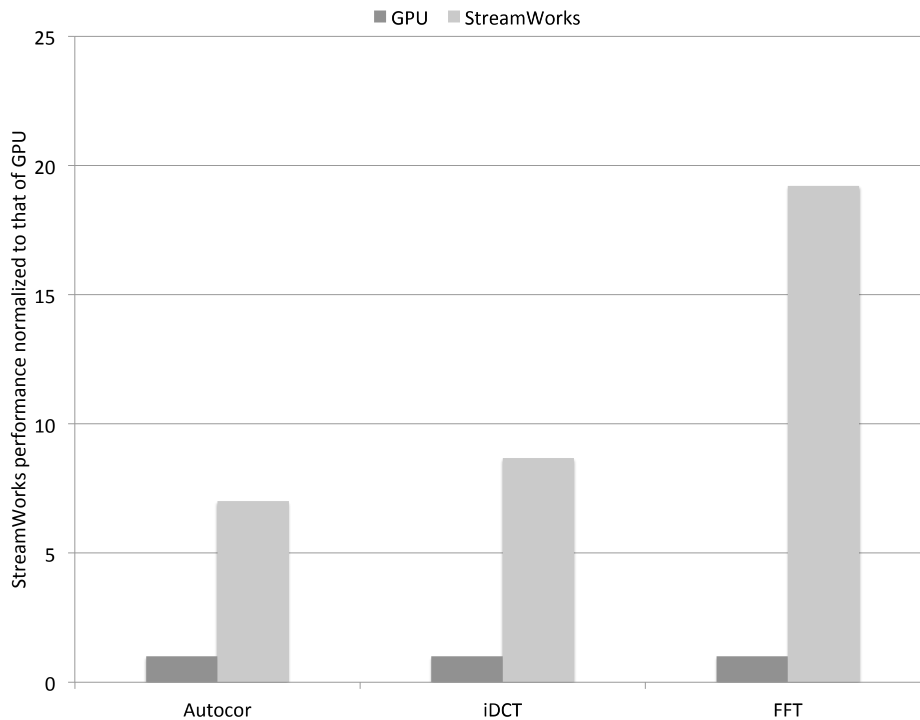


Figure A.3: Performance Comparison of StreamWorks with GPU (Kepler GK110)

Figure A.4 shows the energy-efficiency (GFLOPS/W) comparison of the StreamWorks and the Kepler GK110. As the figure demonstrates, StreamWorks is over a couple of orders (average 263.22×) more energy-efficient than the GPU.

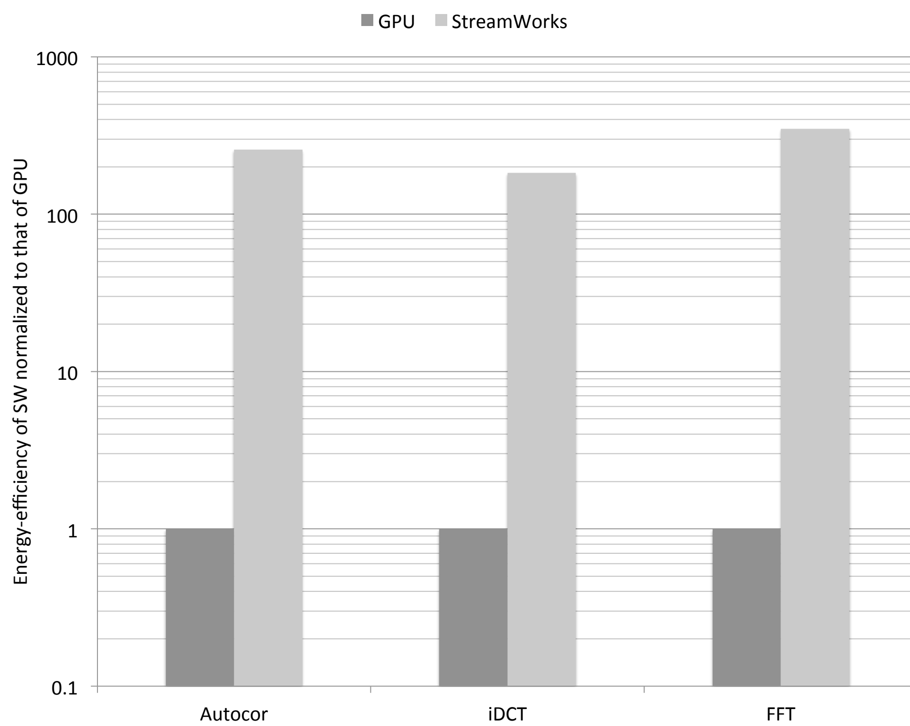


Figure A.4: Energy-efficiency Comparison of StreamWorks with GPU (Kepler GK110)