ENERGY EFFICIENT COMPUTING THROUGH COMPILER ASSISTED DYNAMIC SPECIALIZATION

By

Venkatraman Govindaraju

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2014

Date of final oral examination: 07/29/14

The dissertation is approved by the following members of the Final Oral Committee:
Karthikeyan Sankaralingam, Associate Professor, Computer Sciences
David Wood, Professor, Computer Sciences
Jignesh Patel, Professor, Computer Sciences
Somesh Jha, Professor, Computer Sciences
Mikko Lipasti, Professor, Electrical and Computer Engineering

© Copyright by Venkatraman Govindaraju 2014

All Rights Reserved

To my family

ENERGY EFFICIENT COMPUTING THROUGH COMPILER ASSISTED DYNAMIC SPECIALIZATION

Venkatraman Govindaraju

Under the supervision of Associate Professor Karthikeyan Sankaralingam
At the University of Wisconsin-Madison

Due to the failure of threshold voltage scaling, per-transistor switching power is not scaling down at the pace of Moore's Law, causing the power density to rise for each successive generation. Consequently, computer architects need to improve the energy efficiency of microarchitecture designs to sustain the traditional performance growth. Hardware specialization or using accelerators is a promising direction to improve the energy efficiency without sacrificing performance. However, it requires disruptive changes in hardware and software including the programming model, applications, and operating systems. Moreover, specialized accelerators cannot help with the general purpose computing. Going forward, we need a solution that avoids such disruptive changes and can accelerate or specialize even general purpose workloads.

This thesis develops a hardware/software co-designed solution called *Dynamically Specialized Execution*, which uses compiler assisted dynamic specialization to improve the energy efficiency without radical changes to microarchitecture, the ISA or the programming model. This dissertation first develops a decoupled access/execute coarse-grain reconfigurable architecture called DySER: Dynamically Specialized Execution Resources, which achieves energy efficiency by creating specialized hardware at runtime for hot code regions. DySER exposes a well defined interface and execution model, which makes it easier to integrate DySER with an existing core microarchitecture. To address the challenges of compiling for a specialized accelerator, this thesis develops a novel compiler intermediate representation called the Access/Execute Program Dependence Graph (AEPDG), which accurately models DySER and captures the spatio-temporal aspects of its execution. This thesis shows that using this representation, we can implement a compiler that generates highly optimized code for a coarse-grain reconfigurable architecture without manual intervention for programs written in the traditional programming model.

Detailed evaluation shows that automatic specialization of data parallel workloads with DySER provides a mean speedup of $3.8\times$ with 60% energy reduction when compared to a 4-wide out-of-order processor. On irregular workloads, exemplified by SPECCPU, DySER provides on average speedup of 11% with 10% reduction in energy consumption. On a highly relevant application, database query processing, which has a mix of data parallel kernels and irregular kernels, DySER provides an $2.7\times$ speedup over the 4-wide out-of-order processor.

Acknowledgments

I would like to thank everyone who helped to complete this dissertation.

I would like to thank my advisor, Prof. Karthikeyan Sankaralingam for his support and guidance throughout my graduate school. He gave me the opportunity to research as a member of the Vertical research group to develop both the Copernicus architecture and DySER. He provided an environment in which I could be productive without much of the usual graduate school stress. He taught me how to read, write, present research ideas and how to do research in computer architecture.

I would also like to thank my committee members, Prof. David Wood, Prof. Mikko Lipasti, Prof. Somesh Jha, and Prof. Jignesh Patel for all of their comments and feedback to make this dissertation better. I would also like to thank Prof. Guri Sohi, and Prof. Mark Hill, for their encouragement, support and feedback on my papers and talks.

I would also extend a special thanks to Lena Olson and Tony Nowatzki for their help in proof-reading this dissertation.

I would like to thank the members of Vertical research group throughout the years for their patience and feedback on my various research ideas. In particular, I thank Marc de Kruijf, Tony Nowatzki, Matt Sinclair, and Vijay Thiruvengadem for being my office mates and enduring both my discussions and digressions. I would also like to thank Emily Blem, Jai Menon, Raghu Balakrishnan and Newsha Ardalani for productive discussions on computer architecture research. Also, I thank the DySER project members, Chen-han Ho, Ryan Coffell, Chris Frericks, Jesse Benson, Zachary Marzec, Preeti Agarwal, Ranjini Nagaraju, and Harsha Sutaone for their contributions to DySER research.

I would like to thank students who attended the architecture reading group and the architecture

lunch for the productive discussions and their feedback on my research: Luke Yen, Jayaram Bobba, Dan Gibson, Yasuko Watanabe, Polina Dudnik, Dana Vantrease, Matthew Allen, James Wang, Derek Hower, Arkaprava Basu, Srinath Sridharan, Somayeh Sardashti, Hamid Ghasemi, Rathijit Sen, Gagan Gupta, Lena Olson, Jayneel Gandhi, Nilay Vaish, Jason Power, Joel Hestness, Muhammad Shoaib, and Hongil Yoon.

I am grateful for the internship opportunities that I had with Intel and Oracle. I thank Kevin Moore, ChangKyu Kim and Sai Santhosh for providing me these wonderful opportunities and letting me explore my ideas in an industrial setting.The research at industry provided me more insights into the computer architecture that are otherwise impossible to get. I would also thank the UW industrial affiliates for their insights and interest in my work.

I would like to thank Stefan Westman, Manuel de la Pena, Jill Sheridan, and Jackie Westphal for their friendship in my early years in Wisconsin. I also like to thank my college friends Kumaran Rajaram, Sivakumar Ramu, Navaneethan Sundaramoorthy and Lavanya for their friendship and encouragement in all my endeavors.

I thank my parents, Prof. S. Govindaraju and S. Sakunthala, for always supporting me even when my decisions caused them hardships. They supported me throughout my undergraduate college in Chennai, and sent me to the US even though it was not financially prudent at that time. I thank them for their sacrifice, support and trust in me. I would also like to thank my sister Dr. G. Amutha and my brother Dr. G. Sundararajan for their patience and support throughout my life.

This thesis would not have been possible without the support, both financially and emotionally, of my loving wife, Pritha. She shielded me from various distractions and took full charge of caring for our newborn twins, which allowed me to pursue graduate studies in the fullest sense possible. She prioritized my education and career advancement over her career more than once. Without her sacrifice and fullest support, I surely would not have succeeded. I thank her for her sacrifice, love and her unwavering support. I would like to thank my kids Vanathi and Surya for their unconditional love. I thank both of you for reminding me that life is full of little wonders and mysteries and it is not just work.

Contents

Contents vi

Abstract x

1 Introduction 1

1.1 Dynamically Specialized Execution 2

1.2 Thesis Contributions 6

1.3 Thesis Organization 9

2 A Case for Dynamically Specialized Execution 10

2.1 Motivation 11

2.2 Dynamically Specialized Execution Model 14

2.3 Benefits of Dynamically Specialized Execution Model 16

2.4 Challenges 17

2.4.1 Microarchitecture Challenges 17

2.4.2 Compilation Challenges 18

2.4.3 Application Challenges 19

2.5 Chapter Summary 21

3 DySER Architecture 22

3.1 ISA Extensions for DySER 23

3.2 DySER Microarchitecture 25

3.2.1 Functional Unit 27

3.2.2 Switch Network 28
3.2.3 Pipelined Execution and Flow Control 29
3.2.4 Configuration 30
3.3 Processor Interface 30
3.3.1 Input Interface 32
3.3.2 Output Interface 32
3.3.3 Integration with pipeline 33
3.3.4 Page Faults, Context-switches etc., ... 34
3.4 Integration Feasibility Study 34
3.5 Hardware Mechanisms for Efficiency 35
3.5.1 Control Flow: Predication and ϕ-functions 35
3.5.2 Vectorization: Flexible Vector Interface 36
3.5.3 Virtualization: Fast Configuration Switching 39
3.6 Chapter Summary 41

4 Compiling for DySER 42
4.1 Overview of the DySER Compiler 43
4.2 Access Execute PDG 45
4.2.1 A Case for a New Program Representation 45
4.2.2 Definition and Description 47
4.2.3 An Example 48
4.2.4 Characteristics of good AEPDGs 49
4.3 Initial AEPDG Formation 50
4.3.1 Region Selection for Acceleration 50
4.3.2 AEPDG Construction 51
4.4 Compilation Strategies to Form Ideal Execute-PDG 52
4.4.1 Types of Execute-PDG 53
4.4.2 Transformation Flow 54
4.5 Algorithms for AEPDG Transformations 58

4.5.1 Region Growing 58
4.5.2 Region Virtualization 59
4.5.3 Vectorized DySER Communication 61
4.6 Scheduler and Code Generation 62
4.6.1 Scheduling Execute PDGs 63
4.6.2 Code Generation 64
4.7 Implementation 65
4.8 Case Study 66
4.8.1 Reduction/Induction 66
4.8.2 Control Dependence 67
4.8.3 Strided Data Accesses 68
4.8.4 Carried Dependencies 69
4.8.5 Partially Vectorizable 70
4.9 Chapter Summary 71

5 Experimental Evaluation and Analysis 72
5.1 Overview 72
5.2 Evaluation Methodology 74
5.2.1 Simulation Environment 74
5.2.2 Compiler Implementation 75
5.2.3 Baseline Machine 76
5.2.4 Benchmarks 76
5.2.5 DySER Microarchitecture Details 77
5.3 Workload Characterization 78
5.3.1 Execute-PDG Region Size 79
5.3.2 Phase Behavior 81
5.4 Compiler Evaluation 82
5.4.1 Evaluation methodology 83
5.4.2 Automatic vs Manual DySER Optimization 84

5.4.3 Automatic DySER vs SSE Acceleration 87
5.5 Performance and Energy Evaluation 88
5.5.1 Data Parallel Workloads 88
5.5.2 General Purpose Workloads 91
5.5.3 Source of Improvements and Bottlenecks 94
5.6 Sensitivity Study 101
5.7 Evaluation on Database Kernels 102
5.7.1 Database Primitives 102
5.7.2 Full Query Evaluation 108
5.7.3 Database Evaluation Summary 111
5.8 Chapter Summary 111

6 Related Work 113
6.1 Specialized Architectures 113
6.2 Compilation Techniques 124

7 Conclusion 129
7.1 Summary of Contributions 129
7.2 Closing Remarks 130

Bibliography 132

Abstract

Due to the failure of threshold voltage scaling, per-transistor switching power is not scaling down at the pace of Moore's Law, causing the power density to rise for each successive generation. Consequently, computer architects need to improve the energy efficiency of microarchitecture designs to sustain the traditional performance growth. Hardware specialization or using accelerators is a promising direction to improve the energy efficiency without sacrificing performance. However, it requires disruptive changes in hardware and software including the programming model, applications, and operating systems. Moreover, specialized accelerators cannot help with the general purpose computing. Going forward, we need a solution that avoids such disruptive changes and can accelerate or specialize even general purpose workloads.

Power distribution on typical general purpose processors shows that the execute pipeline stage, which is the actual workhorse of the processor, consumes only about 20% of overall energy. The instruction supply and operand delivery consume most of the overall core power. We observe first that by eliminating the energy consumed on these pipeline stages, we can substantially improve the energy efficiency of the whole processor. Second, we observe that programs execute in phases, and by creating specialized hardware for the computations in the code regions of the most frequently executing phases, we can reduce the energy consumption of these overhead pipeline stages. Third, we observe that we can construct specialized hardware dynamically at run-time by interconnecting a set of heterogeneous functional units with a circuit switched network. Using these observations, in this thesis, we propose a hardware/software co-designed solution called *Dynamically Specialized Execution*, which dynamically creates specialized hardware datapaths for the most frequently executing code regions of the program, to improve the energy efficiency of the processor.

Dynamically specialized execution poses several challenges. First, creating specialized hard-

ware at runtime requires extra microarchitectural mechanisms, which leads to additional design complexity and introduces runtime overheads. Second, compiling and creating optimized code for dynamically specialized architectures is a difficult process because they expose more degrees of freedom for the compiler to optimize for. Finally, it is unknown whether programs written in the traditional sequential programming model can be compiled and optimized for these architectures.

This dissertation tackles these challenges and develops a decoupled access/execute coarse-grain reconfigurable architecture called DySER: Dynamically Specialized Execution Resources, to mitigate the design complexity. DySER exposes a well defined interface and execution model, which makes it easier to integrate DySER with an existing core microarchitecture. To address the challenges of compiling for a specialized accelerator, this thesis develops a novel compiler intermediate representation called the Access/Execute Program Dependence Graph (AEPDG), which accurately models DySER and captures the spatio-temporal aspects of its execution. This thesis shows that using this representation, *we can implement a compiler that generates highly optimized code for a coarse-grain reconfigurable architecture without manual intervention for programs written in the traditional programming model*.

Detailed evaluation shows that automatic specialization of data parallel workloads with DySER provides a mean speedup of $3.8\times$ with 60% energy reduction when compared to a 4-wide out-of-order processor. On irregular workloads, exemplified by SPECCPU, DySER provides on average speedup of 11% with 10% reduction in energy consumption. On a highly relevant application, database query processing, which has a mix of data parallel kernels and irregular kernels, DySER provides an $2.7\times$ speedup over the 4-wide out-of-order processor.

1 Introduction

As predicted by Moore's Law [89], the number of transistors available on a chip is continuing to double every two years, and transistors are getting smaller with each successive generation. Historically, computer architects took these bountiful, smaller, faster and more power efficient transistors and discovered innovative techniques like pipelining, branch prediction, super-scalar processors, out-of-order execution, multilevel cache hierarchy, and memory disambiguation to improve the processor's performance. Since these innovations are mainly invisible to the higher levels of the software stack, even to the compilers and operating systems, they have transparently improved the performance of unmodified applications and software developers reaped the benefits of these innovations without any need of expert knowledge.

Traditionally, even as the frequency and density of transistors increased, the power per unit area in CMOS technology process stayed near constant because of the supply voltage scaling, also known as Dennard Scaling [28]. Unfortunately, voltage scaling is effectively stalled as the leakage current, process variations, and noise impose limits on the threshold voltage and thus preventing the switching voltage to scale down [13]. This failure of Dennard scaling leads to per-transistor switching power not scaling down at the pace of Moore's Law, causing the power density to rise with successive generations. In addition, this implies that in current and future processors, higher performance comes with the cost of higher energy use. Consequently, power and energy have become the first class architectural design constraints for achieving high performance [90, 1, 40, 2, 6, 106, 15, 34, 54, 124, 105].

Given these technological constraints, architects need to improve the energy efficiency of the microarchitecture to sustain the traditional performance growth historically enjoyed by programmers. Thus far, the architectural response to the need for improving energy efficiency is to use

multicores [74, 19, 29, 126, 18], heterogeneous cores [75, 82, 65], and accelerators or specialization [113, 124, 53, 64, 44].

Using accelerators or specialized hardware can improve the energy efficiency of the microarchitecture because it eliminates the need for power hungry structures. With the failure of Dennard scaling, energy efficiency is needed across all computing domains, from warehouse scale computers in data centers to hand-held smart phones. However, the applicability of specialization is limited. For example, SIMD accelerators are ill-suited for programs with irregular control-flow and data accesses. Also, programming for specialized accelerators is a difficult task for the non-expert programmers, as they usually require programmers to have hardware specific knowledge to optimize the code for the specialized architecture. They also require programmers to either use raw assembly or lower level compiler specific constructs to manually optimize the code for them. Finally, these current solutions require disruptive manual changes to software and cannot be easily targetable with a compiler.

Going forward, architects need a simple solution that is applicable to general purpose workloads, but avoids disruptive changes to existing microarchitecture, architecture, programming models and applications and simultaneously improves the overall energy efficiency of the microarchitecture. This dissertation strives to improve the energy efficiency on a variety of workloads by finding and reducing the energy wasted in general purpose processors.

The rest of this chapter is organized as follows. Section 1.1 describes the overview of the hardware/software codesigned solution proposed in this dissertation to improve the energy efficiency of the microarchitecture. Section 1.2 lists the contributions of this thesis and Section 1.3 presents the organization of this dissertation.

1.1 Dynamically Specialized Execution

The goal of this dissertation is to demonstrate that we can improve energy efficiency of general purpose processors on diverse sets of workloads without radical changes to the existing hardware/software stack through compiler assisted automatic dynamic specialization. To achieve that goal, this dissertation proposes a hardware-software codesigned solution, called *Compiler Assisted*

Dynamic Hardware Specialization, which improves the energy efficiency of the microarchitecture by eliminating per-instruction overheads such as Fetch, Decode and Writeback. The main idea is to dynamically specialize the hardware such that it can efficiently execute the application's computation without using the power hungry structures in general purpose processors. In this execution model, which we call *Dynamically Specialized Execution (DySE)*, at compile-time, the compiler partitions the application into code regions and creates configuration bits that the hardware substrate uses at run-time to create specialized hardware. Specialization improves energy efficiency because it eliminates per-instruction overheads. In addition, this architecture is applicable to a wide range of application because it can dynamically create specialized datapath at runtime. Since the DySE execution model uses a compiler to specialize applications written in the traditional programming model, it is transparent to the average programmer and does not require any radical software changes.

However, the dynamically specialized execution model presents several challenges to microarchitects, architects, and compiler writers. The challenges are:

- **Architectural Challenges:** Specializing datapaths dynamically at runtime requires extra mechanisms in the existing microarchitecture, which can lead to additional complexity in hardware and can introduce unnecessary runtime overheads. Recognizing which datapaths to specialize at runtime by hardware efficiently is difficult without assistance from the compiler and/or the ISA.

- **Compiler Challenges:** The compiler for this execution model needs to manage the flexible hardware substrate that dynamically creates arbitrary hardware datapath. These flexible substrates enable additional compiler optimizations. However, the complexity of the compiler increases substantially, since these flexible substrates have more degrees of freedom than that of other compiler managed resources such as registers. Like compilers for VLIW, compilers for the dynamically specialized execution model suffer from a lack of dynamic information about the runtime.

- **Application Challenges:** It is difficult to actually use dynamically specialized datapaths and

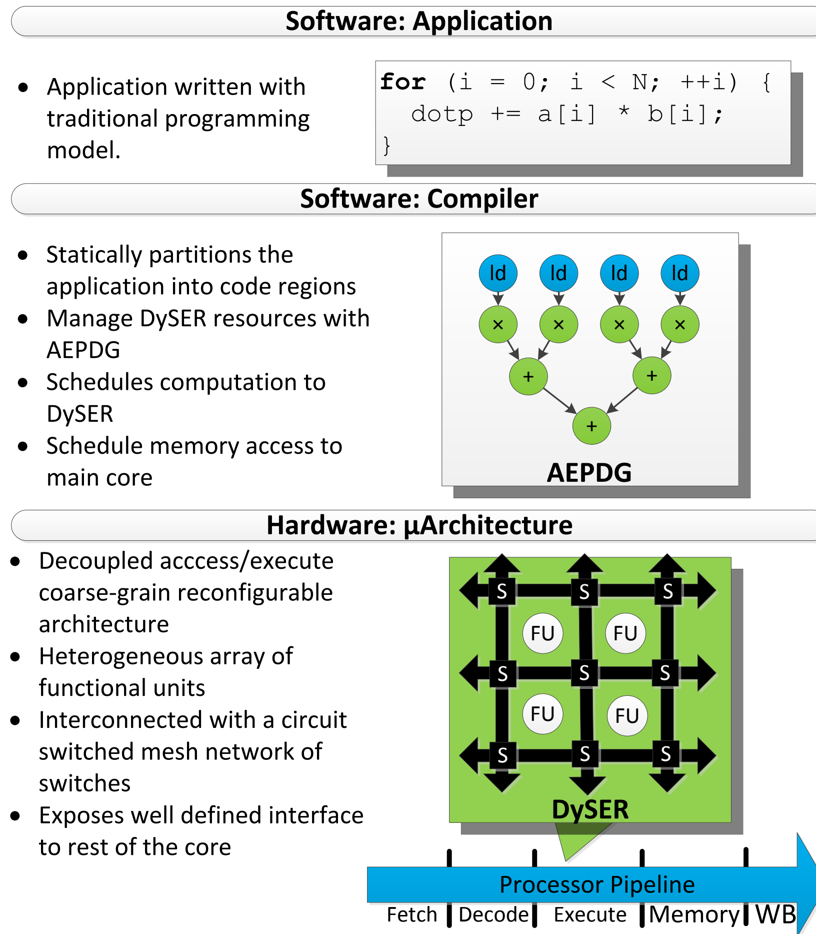


Figure 1.1: Overview of the Compiler Assisted Dynamically Specialized Execution

be more energy efficient or competitive with existing solutions because there are overheads in identifying and creating specialized datapaths at runtime.

This dissertation addresses these challenges of dynamically specialized execution model by proposing a microarchitecture, called ***Dy**namically **S**pecialized **E**xecution **R**esources (DySER)*, based on decoupled access/execute (DAE) architectures [116]. It subsequently addresses the compiler challenges by designing and implementing a compiler for DySER using a suitable abstration and demonstrates that a variety of workloads can benefit from specialization with DySER.

- **Architectural Solution:** DySER dynamically specializes the execute pipeline of decoupled access execute architecture for arbitrary computations by creating specialized hardware datapaths. DySER is integrated to a general purpose processor as a long latency functional

unit. It is a heterogeneous array of functional units interconnected with a circuit-switched mesh network, without any additional state or other overhead resources. It communicates with the rest of the processor pipeline through a well-defined interface. DySER requires only small changes to existing microarchitectures, and thus it is easy to manage the additional hardware design complexity introduced. DySER relies on its compiler to identify and create the hardware datapath at compile time so that it can specialize the hardware at runtime and avoid adding any extra overheads. DySER offers a coarse-grain reconfigurable substrate to build efficient specialized hardware datapaths to offload work from the energy-hungry processor pipeline. The microarchitecture of DySER presented here is published in HPCA 2011 [47] and it is a joint work with my collobarator, Chen-Han Ho.

- **Compiler Solution:** Since DySER is a decoupled access/execute architecture, the memory address calculation, loads and stores execute in the main processor pipeline and the remaining computations execute inside DySER. In order to model this decoupled nature, this thesis first develops a novel compiler intermediate representation called the *Access/Execute Program Dependence Graph (AEPDG)*, a graph based representation that is a suitable abstraction for DySER's spatial computation. The AEPDG captures the spatio-temporal aspects of memory accesses and computation and accurately models the dynamically specialized execution. Using this graph based intermediate representation, a compiler can easily manage the spatial resources that DySER provides and the relationship between the spatial computation of DySER to the memory accesses and can generate highly optimized code for DySER.

  Second, this thesis designs and implements a compiler that uses the AEPDG to compile programs written in C/C++ using the traditional programming model for DySER with minor changes to the source code. This compiler is codesigned with the microarchitecture. Although the compiler lacks dynamic information about the run-time, which may prevent the compiler from generating optimized code, the design of the DySER hardware allows it to dynamically adapt and efficiently compute even with compiler scheduled code. The DySER compiler not only infers useful information about the execution using the AEPDG, but also performs transformations and optimizations on the AEPDG. Figure 1.1 shows an overview of the DySER

architecture, which uses the dynamically specialized execution model to efficiently specialize general purpose workloads.

- **Addressing Application Challenges:** To address the application challenges and to demonstrate that the solution presented here is efficient, this thesis evaluates the proposed full system stack with DySER integrated with an out-of-order processor and show that it improves the efficiency of the microarchitecture on diverse sets of workloads ranging from highly data parallel to heavy control-dependent applications. This evaluation uses the DySER compiler that is developed in this thesis to compile data parallel benchmarks from Intel Throughput kernels [111] and Parboil [99] and general purpose benchmarks from SPECCPU 2006 [60] and PARSEC [10]. We also evaluate DySER on database query processing to demonstrate that DySER's effectiveness. This demonstrates that full-fledged legacy applications can actually use DySER and be energy efficient without changing their source code.

1.2 Thesis Contributions

This thesis designs and studies a full solution, spanning microarchitecture, compiler and application, that avoids disruptive changes to the existing hardware/software stack and achieves energy efficiency irrespective of the sequential or parallel nature of the application. It presents a practical way to use a coarse grain reconfigurable accelerator automatically for diverse sets of workloads written in the traditional programming model.

Specifically this thesis makes the following contributions:

Hardware/Software design: It presents a hardware/software codesign approach to achieve energy efficient computing with dynamically specialized execution. In particular,

- It observes that the applications execute in phases, and dynamically creating specialized hardware datapaths to match the phases improves the overall efficiency of a general purpose processor.

- It develops a decoupled access/execute reconfigurable architecture that does not lead to additional hardware design complexity to the existing microarchitecture design.
- It describes the microarchitecture of DySER: Dynamically Specialized execution resources, which can dynamically create specialized datapaths for arbitrary sequences of computation.

Compiler Design: It presents a complete source-to-binary compiler toolchain for automatic specialization of programs written in the traditional programming model. Specifically,

- It develops a novel intermediate representation called the Access Execute Program Dependence Graph (AEPDG), a variant of Program Dependence Graph, to capture the temporal and spatial nature of computation.
- It describes the overall design and implementation of a compiler that constructs the AEPDG and applies optimizations on AEPDG to generate high quality code for DySER.
- It performs a detailed analysis on two data parallel benchmark suites to show how close the performance of automatically compiled DySER code comes to its manual counterpart's performance, and how our automated compiler generated code outperforms ICC compiled code for SSE by 80% and consumes 50% less energy.

Application Analysis:

- It demonstrates the ease and utility of codesigned architecture-compiler approach by doing detailed studies on scientific/throughput workloads (Intel Throughput kernels and PARBOIL), and database kernels.
- It evaluates and analyzes the full system stack (architecture + compiler) and presents a detailed characterization of specialization on general purpose workloads (SPECCPU 2006) and emerging workloads (PARSEC).
- It presents quantitative results on a highly relevent application, database query processing, which shows that DySER can provide efficiency when the application has mix of data parallel and irregular code patterns.

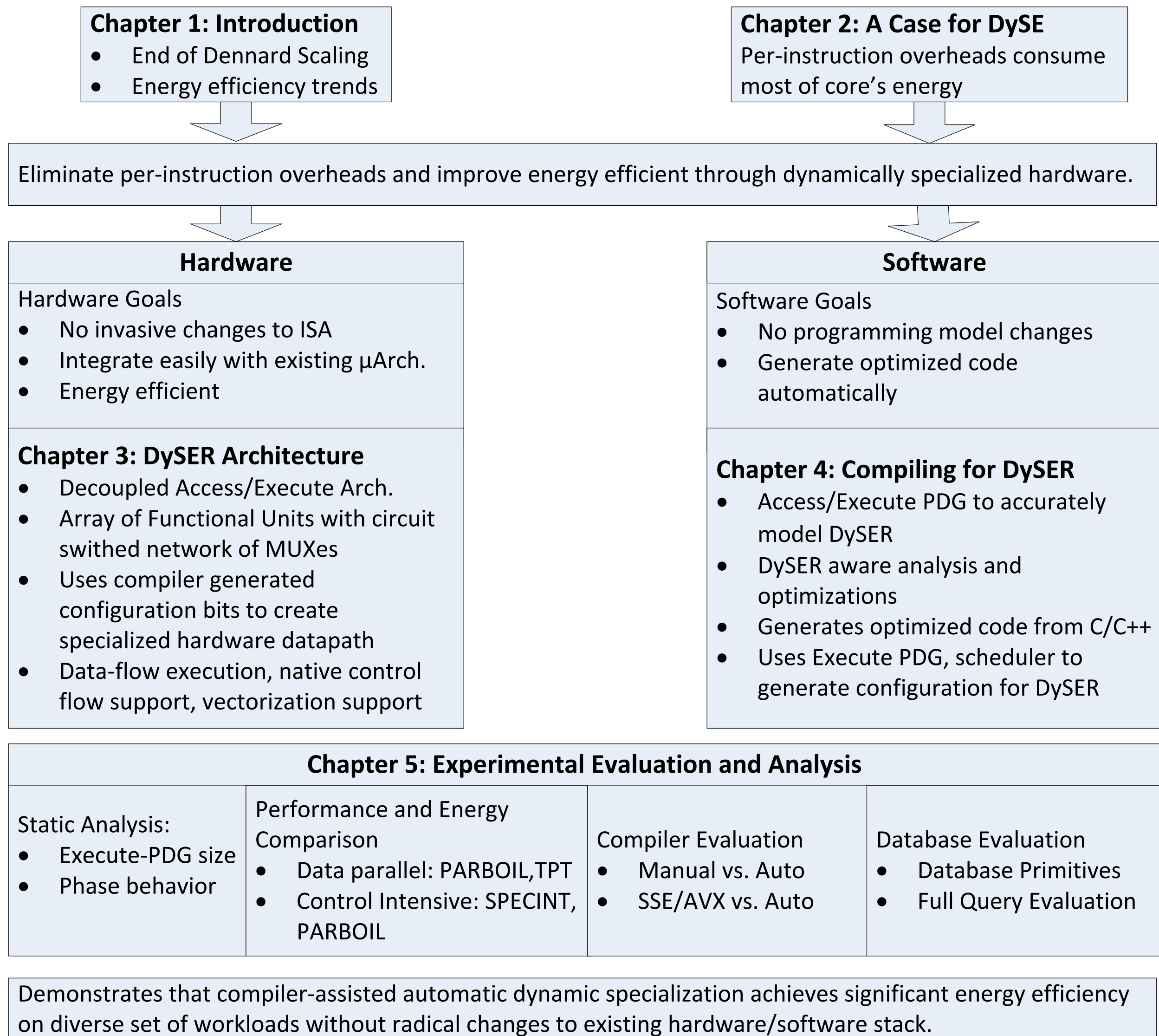


Figure 1.2: Organization of the Thesis

1.3 Thesis Organization

The organization of this thesis is shown in Figure 1.2. Section 2 presents a case for the dynamically specialized execution model as well as the challenges in realizing the benefits of dynamically specialized architecture in more detail. The main contributions of this dissertation are organized into three parts: architecture design, compiler design and experimental evaluation and analysis. Chapters 3 – 5 cover these contributions.

DySER Architecture: Chapter 3 presents the DySER architecture, a decoupled access execute coarse-grain reconfigurable architecture, that dynamically creates specialized hardware for arbitrary sequences of computation. It also describes how to integrate DySER with an existing core's microarchitecture. In addition, it elaborates on additional mechanisms required to attain high performance and efficiency with DySER specialization.

Compiling for DySER: Chapter 4 develops the novel compiler intermediate representation, the Access Execute Program Dependence Graph (AEPDG), which accurately models DySER and captures the spatio-temporal aspects of memory access and the computation. Chapter 4 also describes the compilation tasks that are required to compile programs written using the traditional programming model in a high level language to generate optimized code for DySER. It also presents case studies to illustrate how the DySER compiler generates code for five challenging cases.

Experimental Evaluation and Analysis: Chapter 5 describes the evaluation methodology, benchmarks that this thesis uses to evaluate DySER and its compiler. It evaluates the DySER compiler by comparing compiled code's performance with manually optimized code's performance. It presents quantitative results to show how efficient DySER specialization is with data parallel workloads and general purpose workloads. It also describes the evaluation of the DySER on a more relevent application, database on data analytic query processing. It first presents the evaluation of database kernels and then presents the results for a query from TPC-H benchmark suite.

Chapter 6 describes the related work in detail, and Chapter 7 concludes this dissertation.

2 A Case for Dynamically Specialized Execution

In order to understand the opportunities for specialization to improve the energy efficiency of microarchitectures, it is necessary to study quantitatively the energy consumed by the stages in a typical processor. In this chapter, we first present quantitative data to show that only a small portion of energy consumed by a processor is for the execute stage, which is the one that does useful work. Second, we present an overview of a compiler assisted dynamically specialized execution model. Third, we describe the key benefits of this approach over other specialized architectures or accelerators. Finally, we discuss the key challenges in realizing the potential of the dynamically specialized execution model, specifically with DySER and its compiler.

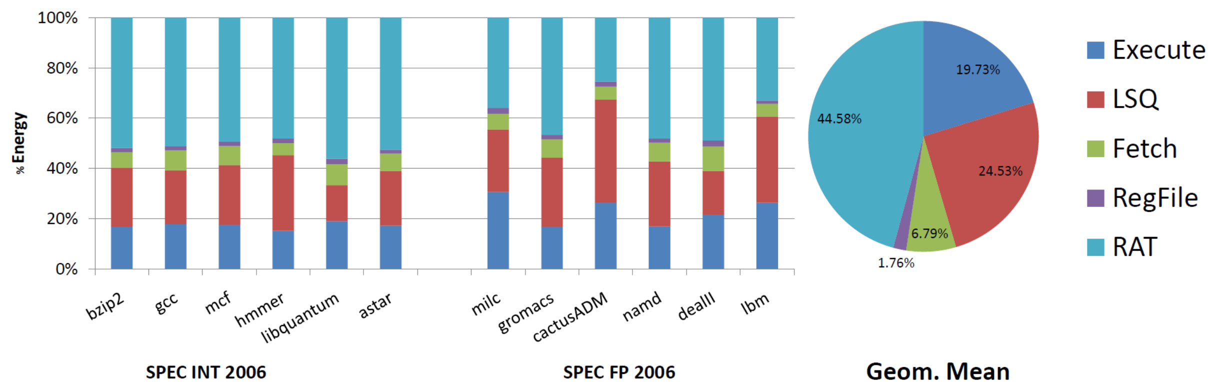


Figure 2.1: Percentage of energy consumed by various pipeline stages of a typical out-of-order processor. This data is from McPAT [80]. Fetch = Fetch + Decode + Branch Predictor, LSQ = Load Store Queue, RegFile = register file access, Execute = ALU, RAT = Renaming + Issue + Retire.

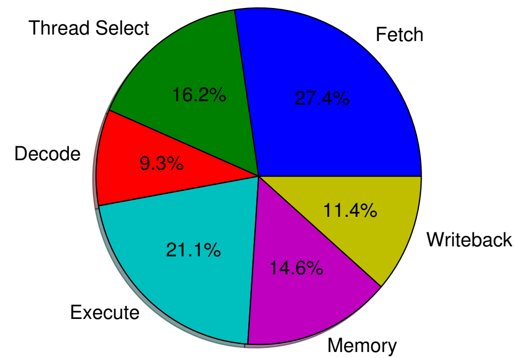


Figure 2.2: Percentage of power consumed by the pipeline stages of OpenSPARC

2.1 Motivation

Figure 2.1 shows the percentage of energy consumed by the pipeline stages in a typical out-of-order processor while executing a subset of SPECCPU 2006 benchmarks [60]. We use the gem5 simulator [12] to collect the microarchitecture statistics for 200 million instructions after skipping the program initialization by fast forwarding 1 billion instructions. Then, we use McPAT [80] to gather the energy breakdown by various pipeline states of a typical out-of-order processor.

The execute pipeline stage, which is the actual workhorse of the processor, consumes only about 20% of overall energy consumed by the processor core. While other pipeline stages support the execute stage by supplying instructions and allowing it to speculatively execute instructions, the energy consumed by these supporting pipeline stages dwarfs the energy consumed by the execute stage.

Figure 2.2 shows the power consumed by various pipeline stages for the OpenSPARC processor. Similar to the out-of-order core, the execution stage in OpenSPARC, an inorder processor, consumes about 27% of overall power. Similar study on other benchmark suites and different configurations of the general purpose processors show a similar core power breakdown that to running SPECINT on an out-of-order processor [110, 118].

Reducing the energy consumed by these supporting pipeline stages will dramatically increase the overall energy efficiency of the processor. Specializing the hardware datapath in the general purpose microarchitecture to match the computation sequence can eliminate most of the energy consumed in the supporting pipeline stages. For example, once a specialized datapath is created,

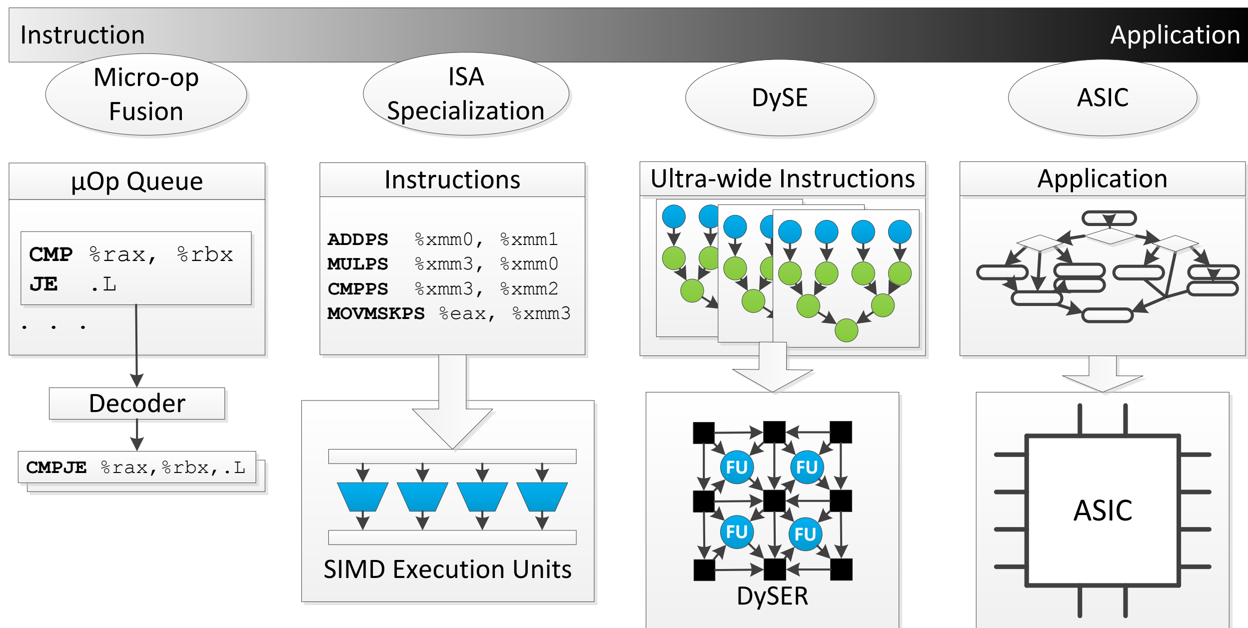


Figure 2.3: Specialization Spectrum

there is no need to fetch instructions continuously. This eliminates most of the energy for the instruction supply, represented in the Figure 2.1 as Fetch. Similarly, since execution using the specialized datapath does not require reading or writing the register files, the energy for operands delivery, represented in the Figure 2.1 as RegFile and WriteBack, can be reduced substantially. Also, since specialized datapaths do not require traditional speculation support based on out-of-order issue or checkpoints for misspeculation recovery, they can also eliminate most of the energy consumed by speculation support, represented in the Figure 2.1 as RAT.

Figure 2.3 shows a spectrum of hardware specialization increasing in granularity. On the left is the implicit microarchitecture specialization. An example is macro-op fusion, where the microarchitecture fuses the sequence of instructions to amortize the per-instruction overheads like register reads, writebacks and renaming etc., For example, in Intel Core i7, macro-op fusion takes instruction combinations such as compare, followed by branch, and fuses them into a single operation [59]. Although this specialization does not eliminate the fetch stage, it eliminates decode, register renaming and other overheads without any visible changes to the ISA or to the compiler.

On the other end of the spectrum is the application specific integrated circuit(ASIC). ASICs

eliminate most of the overheads associated with the general purpose processor and realize the full potential of computational energy and performance efficiency of hardware [53]. However, it is impractical and expensive to design and implement a custom chip for every application.

The second column from the left is the instruction set specialization which is visible to the compiler. Examples include encryption accelerators in Niagara processor [113] and the multimedia ISA extensions such as SSE [119] and GPU instruction sets. However, they do not easily generalize outside their specific domain and require extensive changes to the hardware/software toolchain including the applications and operating systems. Also, they require new programming models; for example, GPUs require CUDA or OpenCL and are usually harder to program than in the traditional programming model.

Using accelerators that are specific to a particular domain, such as SSE/AVX for data parallel workloads, along with a general purpose core improves energy efficiency and performance because they eliminate per instruction overheads and unnecessary temporary states in computation. However, these accelerators are not flexible and cannot accelerate arbitrary code. First, they expose a rigid interface and compiling for them is hard, as evident by the compilers' inability to generate optimized code for short-vector extensions such as SSE/AVX. Second, designing and implementing multiple specialized accelerators integrated to a processor is area inefficient and uneconomical.

As shown in Figure 2.3, the dynamically specialized execution model specializes the application phases with "ultra wide instructions" described in the next section and provides efficiency close to an ASIC, but flexible enough to specialize diverse sets of applications. It uses insight from domain-driven accelerators to dynamically specialize hardware datapaths to create specialized accelerator to match the characteristics of the computation. Creating efficient specialized hardware suited for the computation and using it provides efficiency and eliminates most of the energy consumed by the supporting pipeline stages such as Fetch, Decode, and Renaming. The ability to do specialization dynamically provides the necessary flexibility and generality and makes this approach applicable to a wide variety of workloads.

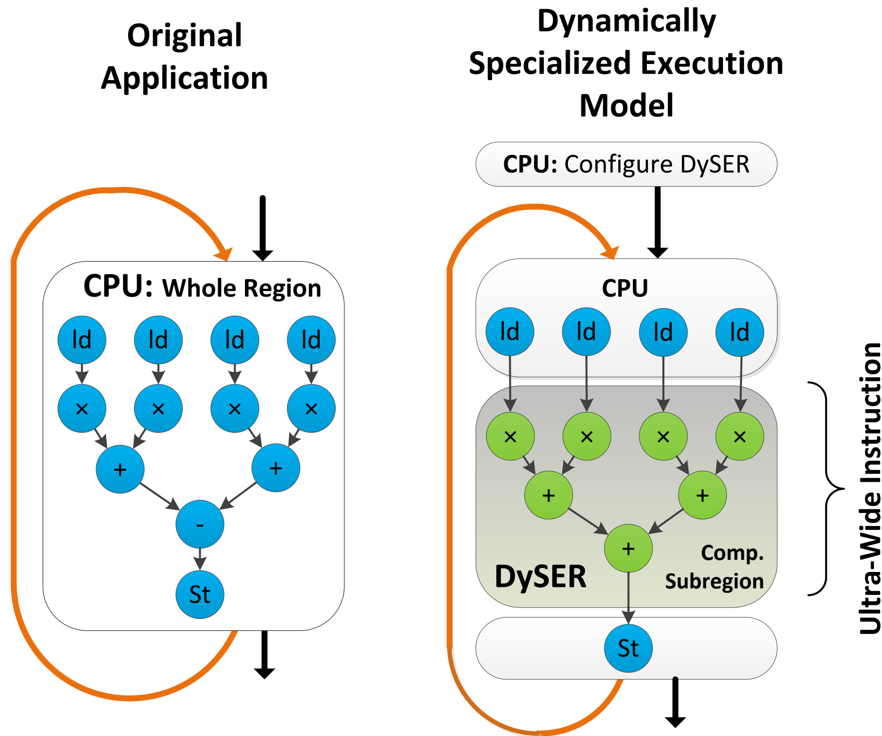


Figure 2.4: Dynamically specialized execution model

2.2 Dynamically Specialized Execution Model

In the DySE execution model, the application is abstracted as a sequence of "ultra-wide instructions". Each of these ultra-wide instructions represents a large sequence of computation that the application needs to perform to accomplish its tasks. In this thesis, we develop and use Dynamically Specialized Execution Resources (DySER) to execute these ultra wide instructions. DySER is a decoupled access/execute reconfigurable accelerator that is tightly integrated to a main processor. The main processor injects data values to DySER and DySER acts a compound functional unit that performs the sequence of operations in the ultra wide instruction, and delivers the results to the main processor. DySER consists of a heterogeneous array of functional units such as adder, logical units, shifters, comparators etc., interconnected with a mesh network of simple switches. These ultra-wide instructions encode the physical routes on the substrate and connect the functional units to form a specialized hardware datapath that matches the sequence of computation that ultra-wide instructions represent.

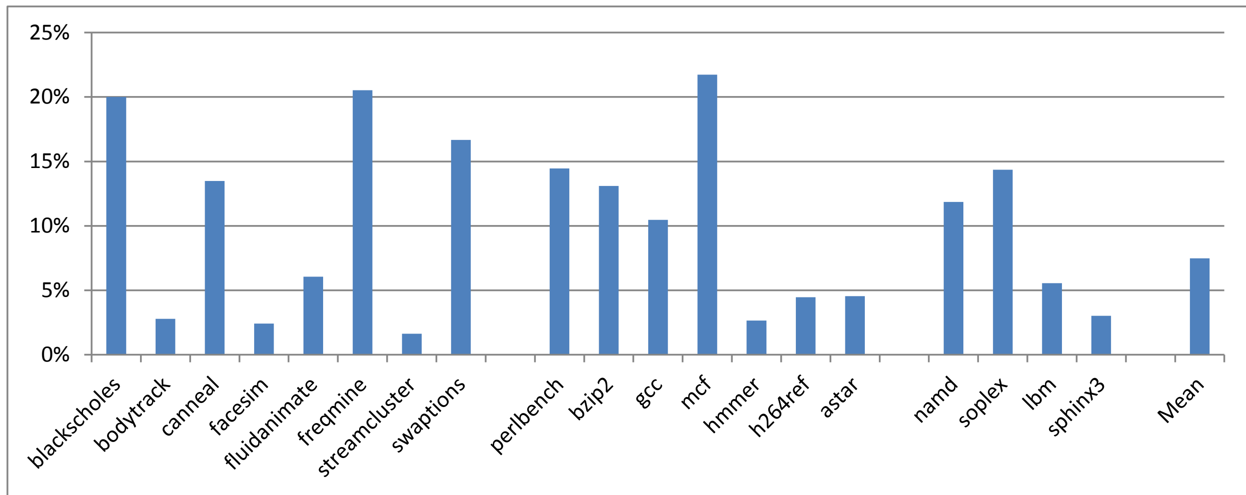


Figure 2.5: Percent of code regions that contribute to 90% of dynamic instruction code

Figure 2.4 shows the Dynamically Specialized Execution (DySE) model. Before an application uses DySER to specialize, it needs to set up the physical routes corresponding to the ultra wide instruction. Once DySER is configured, the operands to the computation are fed to DySER from the main processor. Since DySER is a decoupled access/execute architecture, the inputs from memory are directly fed to DySER using special load instructions that execute in the main processor pipeline. DySER performs the computation within its heterogeneous array of functional units and switches and generates the output. The output is then stored to memory directly using special store instructions executing in the main processor pipeline. The rationale for using the main processor pipeline to load and store data from memory is presented in subsection 2.4.3.

This execution model hinges on the assumption that only a few such ultra wide instructions are active during a phase of an application. Figure 2.5 shows the percentage of code regions from PARSEC and SPECCPU that contribute to 90% of the dynamic instruction count. From the graph, we observe that on average only 8% of code regions contribute to 90% of the dynamic instruction code. Hence, DySER does not need to specialize each and every code region with a ultra-wide instruction. As long as it specializes the most frequently executing code regions, DySER can execute the computation with efficiency closer to the ASIC because it can reuse the physical routes set up in the circuit switched network. The execution of these ultra wide instructions over many invocations of the DySER can amortize the cost of setting up the static routes.

2.3 Benefits of Dynamically Specialized Execution Model

Below, we highlight the reasons why the dynamically specialized execution model, especially DySER, is not only energy efficient but also comparable to other domain specific architectural solutions. The potential benefits of this execution model are:

Energy Efficiency With hardware datapath specialization, DySE removes the per-instruction overheads and improves energy efficiency of the microarchitecture. Since power and thermal effects will allow only a small portion of the transistors to be active at any given time [34, 124], using specialized accelerators, which require only a subset of transistors to be active, to improve the energy efficiency of microarchitecture is prudent.

Compilability Because it allows specialization at a coarse grained level, i.e., functional unit level, instead of fine grained like FPGA with LUT, DySER is a good compiler target. This is because the existing compiler technology and hardware/software toolchain are accustomed to operating at functional unit level rather than at bit level. In addition, their analysis and transformation passes can be reused without much modification to compile and optimize code for coarse grain reconfigurable accelarators.

Design Complexity: The execution model of ultra-wide instructions, which communicate through the memory, calls for a stateless compound functional unit. The implemenation of DySER as a decoupled access/execute coarse grain reconfigurable accelerator lends itself to easy integration with the processor pipeline.

Flexible Execution Model The execution model of ultra-wide instructions, that makes use of dynamically specialized hardware implementation, unifies different specialization techniques like SIMD-execution, instruction specialization, and loop-accelerators with potentially little efficiency loss.

Area Efficiency and Programmability Dynamically creating specialized hardware for application phases, instead of designing accelerators for each phase, provides area efficiency. Even when

multiple accelerators are integrated, they expose an inflexible interface to compilers and accelerate applications only from their chosen domains.

2.4 Challenges

In this section, we discuss the key challenges in realizing the potential benefits of dynamically specialized execution model. Moreover, this section also describes how DySER and its codesigned compiler tackle these challenges.

2.4.1 Microarchitecture Challenges

Design Complexity: Augmenting an existing processor pipeline with additional mechanisms to dynamically specialize datapath for computation increases the complexity of the design and makes the verification of the microarchitecture hard. Also, if the accelerator is not carefully integrated to the pipeline, it may also introduce extra runtime overheads. However, DySER integrates to a processor pipeline through a well defined interface in the execute pipeline stage of the processor. Since DySER interacts with the main processor pipeline as a stateless, long latency functional unit, it can be integrated to the existing general purpose processor easily and any additional complexity introduced because of DySER is manageable. Chapter 3 describes the DySER architecture in detail. Also, to demonstrate the simplicity of the DySER architecture, we developed a FPGA prototype that integrates an implementation of DySER to OpenSPARC and more details are in the following publications [8, 7].

Energy Efficient Dynamically Specializing Substrate: There are several challenges in designing and implementing an energy efficient dynamically specializing substrate.

First, the substrate should allow us to specialize most sequences of computations. To achieve that, it should have a set of functional units to perform most operations and a way to route data between any two functional units. Although the functional unit mix is highly application dependent, it is possible to build a common-case array of heterogeneous units [47]. Using a crossbar, or packet switching interconnect network to route data between functional units consumes energy and

introduces extra overheads. DySER uses a circuit switching interconnect instead. Circuit switching is energy efficient because the switches route data to its destination without any unnecessary work.

Second, this substrate should allow us to utilize the functional units as much as possible. To increase the utilization, DySER pipelines the computation. DySER uses a simple credit based flow control to stall the pipeline of computation if the destination of a data cannot accept new inputs.

Finally, when a computation sequence requires more functional units than the number of functional units available, we can only map a portion of the sequence. This leads to suboptimal performance and energy inefficiency. Through modulo scheduling and hardware mechanisms to switch configurations fast, DySER emulates a large DySER and maps the entire computation sequence [45], and mitigates the effect of the lack of computational resources needed.

Data-independent Data flow and Routing: If the values flowing between the functional units are data dependent, then a router or a switch needs to examine the data before routing it to its destination. Data flow becomes data-dependent only if there are buffers being used to share a functional unit among multiple operations. Instead of sharing functional units among multiple operations, DySER takes the radical approach of providing a single computational unit for each primitive operation in the "ultra wide instruction". This makes the routing values no longer data-dependent. This also provides the opportunity to use energy efficiency circuit-switched static routing instead of power-hungry packet switching, where it needs to examine the data constantly. Since the DySE execution model reuses the "ultra wide instruction" many times, the circuit-switched network, which does not need to change its static routing between the invocations of these wide instruction, is more efficient.

2.4.2 Compilation Challenges

Compilers traditionally manage architectural registers and schedule instructions while considering a small region of code and ignoring the pipelining and other temporal aspects of the execution. In order to use a dynamically specialized architecture effectively, a compiler also must manage the internal operations of the dynamically specialized architectures.

First, it should be aware of the capability of the substrate. Second, it should identify computation sequences that the substrate can map successfully. Third, it should map any identified computation sequences to the substrate and create the datapath at compile-time. Fourth, while optimizing, it should be aware of the pipeline execution and dataflow execution of the dynamically specialized architectures. Otherwise, it will generate suboptimal code. Finally, it should generate code such that communication between the processor pipeline and the specializing substrate is small and they amortize the cost of creating the datapath in run-time by reusing the specialized datapath.

To be an effective compiler, the DySER compiler should be aware of DySER's ability to pipeline multiple invocations of DySER. To manage the flexibility exposed by DySER, this thesis develops a new program representation called the Access/Execute Program Dependence Graph (AEPDG), which models the decoupled access/execute nature of DySER architecture. Using the AEPDG, the DySER compiler generates optimized code for DySER.

2.4.3 Application Challenges

Irregular Loads and Stores Related work on specialized architectures has found irregular memory accesses to be a problem. Previous works sidestep the problem by restricting their domains to where memory accesses are regular [23], or restricting the program scope [66, 23, 130], or by enforcing specialized languages [52, 51]. The resulting architectures are unscalable and highly domain-specific and cannot be used to specialize and accelerate general purpose workloads.

Instead, DySER exploits a simple insight, which some may feel is counter-intuitive: Use a general purpose processor to perform memory operations. Driven by sophisticated advances in memory disambiguation [120], prefetching [70], and streaming [21, 79], general-purpose processors with short physical paths to hardware managed caches provide effective low-latency access to memory. Quantitatively, the PARSEC benchmarks typically have L1 data-cache miss-rates less than 2% [9] and the SPECCPU benchmarks typically have 30 data-cache misses per thousand instructions [68]. Hence, our solution is to utilize general-purpose processors as a load/store engine to feed a specialized datapath. This provides sufficient support to provide practical computation specialization without disrupting the hardware/software stack.

In the DySE execution model, the compiler partitions the control-flow graph (CFG) of a program into multiple code-regions that have no loops or back-edges. The code region is further sliced into memory subregion and compute subregion. The memory subregion includes all computations of memory address and the compute subregion includes all other computation. The instructions in the memory subregion compute the load address and load the values to the DySER. The instructions in the compute sub-region consume the values loaded from memory, perform computations and send store values back to the main processor. This observation that the application can be partitioned into the memory slice and computatation slice provides the generality and freedom to investigate the large code-region for specialization and a hardware block simple enough to integrate with processors like a functional unit. Since loads and stores are performed in the processor pipeline, it naturally maintains the load-store ordering and allows the processor's memory disambiguation optimizations to proceed unhindered. Recently, building upon these insights, Ho et al. have developed the Memory Access Dataflow (MAD) architecture that plays the role of the host processor [63].

Control-flow: As dynamically specialized execution is a spatial computation, control-dependence is usually difficult to manage. DySER provides three hardware/software codesign solutions to handle control-flow in the programs. First, DySER has a select functional unit which selects one of its output based on a third input. It is similar to conditional move after using the if-conversion. The compiler can transform the control-flow into data-flow and easily map to the hardware substrate that the DySER provides. Second, the functional units in DySER can be predicated, i.e, depending upon a control signal, the functional units can generate "invalid output". Third, it has a functional unit called the ϕ-function that operates similar to the select function unit with the exception that it selects its valid input as its output and discards the invalid output. The behavior of this ϕ-function unit is similar to the ϕ-function of the Single Static Assignment (SSA) form of the program representation. With these mechanisms, DySER handles control-flow natively.

Data Parallel Workloads: Although data parallel workloads are not highly challenging for spatial computation, when compared to data parallel specific accelerators such as SIMD accelerator, spatial computation is not efficient. However, DySER, with its ability to dynamically specialize for the

application phase, can emulate a SIMD accelerator by creating independent lanes of computation and achieve similar if not better efficiency than SIMD accelerator. But, in order to achieve, it also needs wide memory interface to caches or memory and vector instructions to perform the loads and stores.

2.5 Chapter Summary

This chapter provided the motivation for Dynamically Specialized Execution and described the benefits of doing dynamically specialized architecture. It also presents the challenges of achieving energy efficiency with the dynamically specialized architecture and how DySER and its compiler developed in this dissertation tackle the challenges.

3 DySER Architecture

Dynamically Specialized Execution Resources (DySER) is meant to be integrated as a long latency compound functional unit into a processor pipeline, as shown in Figure 3.1. The compiler and processor view DySER as a block of computational units that consume inputs from memory or from registers and produce outputs, which can be stored directly to memory or to registers.

This chapter presents the architecture of DySER, which creates a specialized hardware datapath dynamically at run-time. It is organized as follows. Section 3.1 presents the instructions required to communicate with DySER. Section 3.2 describes the microarchitecture of DySER in detail and Section 3.3 explains the processor interface of DySER and how DySER integrates with existing processor pipelines. Section 3.5 describes the mechanisms and optimizations that make DySER an efficient accelerator for data parallel workloads and also for control intensive workloads.

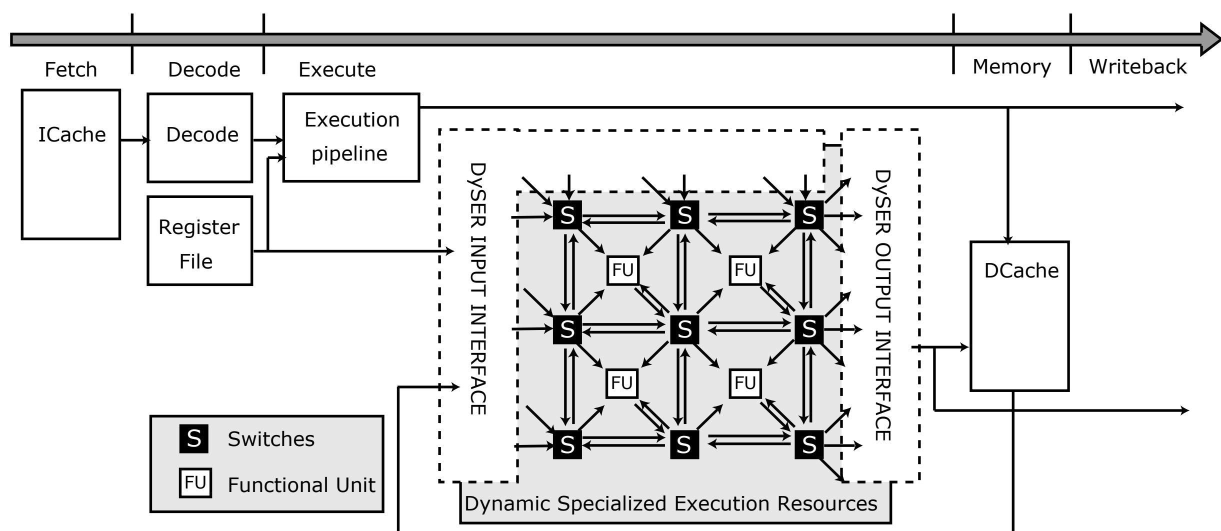


Figure 3.1: DySER Integration on a processor pipeline

3.1 ISA Extensions for DySER

Since DySER is a decoupled access/execute architecture and integrates to the processor pipeline as a compound functional unit, it requires the processor to explicitly send operands to and receive values from DySER. Figure 3.1 shows the integration of DySER with a processor pipeline. DySER is connected to the processor pipeline through a simple logical First-In-First-Out (FIFO) based interface. The processor communicates to DySER through a set of named input and output ports, which correspond to FIFOs in the input and output interface respectively. Execution with DySER follows the dynamically specialized execution model described in Section 2.2 and proceeds as follows: When the program reaches a code-region that can be executed efficiently on DySER, the processor configures DySER by sending configuration bits from memory to DySER. After DySER is fully configured, the main processor starts injecting register values and memory values into DySER. DySER gets the values from its input interface and execution proceeds in data-flow fashion with values routed between functional units through a circuit-switched network. Outputs are delivered to the output interface and written back to registers or stored to memory.

Table 3.1 lists the ISA extensions for DySER and describes the operation of each instruction. DySER requires five instructions to manage the operations of DySER. It needs an instruction to send configuration bits, an instruction to send register values, an instruction to receive register values, an instruction to load values directly to DySER, and an instruction to store values directly from DySER.

The instruction `dyserinit` initializes DySER with configuration bits. It reads the configuration bits from the instruction cache and sends the bits to the input interface of DySER. Multiple `dyserinit` instructions may be required to fully configure DySER. It stalls any other following DySER instructions such as `dysersend` and `dyserload`. Also, this instruction needs to be executed as a non-speculative instruction in an out-of-order processor, because in order to rollback this instruction, DySER needs to reconfigure itself back to the previous configuration, which is very expensive.

The instruction `dysersend` reads a register and sends the register value to the specified DySER port. If the DySER port, a FIFO, is full, this instruction will stall the pipeline. Similarly, the

Instruction	Operation	Comments
`dyserinit`	Sends configuration bits to DySER	Stalls other DySER instructions until it retires
`dysend`	Sends data from a register to DySER	may stall if the DySER port is full
`dyserload`	Loads data from memory directly to a DySER	Stalls if the DySER port is full
`dyserrecv`	Receives data from DySER and writes it to a register	Stalls if the DySER port is empty
`dyserstore`	Receives data from DySER port and stores the value to the memory	Stalls if the DySER port is empty

Table 3.1: Basic DySER Instructions

`dyserload` instruction loads a value from memory and sends the value to the specified DySER port. The `dyserrecv` instruction reads a value from DySER's output port and writes the value to a register and the `dyserstore` instruction stores the value from DySER's output port to memory. Both of these instructions will stall the pipeline if the specified output port is empty.

Figure 3.2 shows a code snippet and the corresponding DySER code. Before the code uses DySER, it configures DySER using multiple `dyserinit` instructions to provide configuration bits, as shown in Figure 3.2c. Then, it sends data to DySER using `dyserload` instructions, which load the data from memory. Once data has arrived to DySER's input FIFO, it follows the configured path through the switches. When the data reaches the functional units, the functional units perform the computation in data-flow fashion. Finally, the results of the computation are delivered to output FIFOs, from which the processor fetches the outputs and sends them to memory using a `dyserstore` instruction.

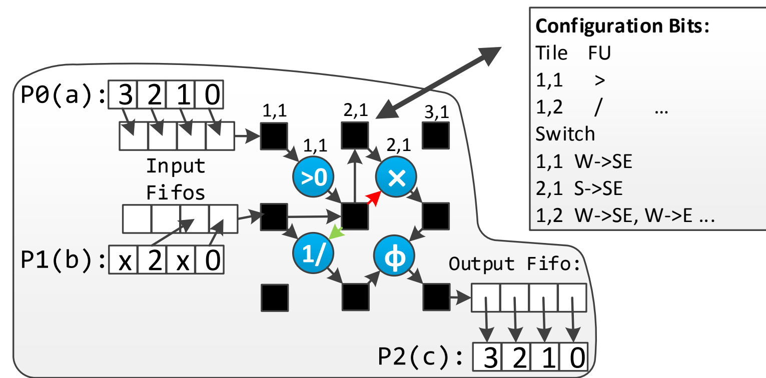
Although these instructions are sufficient to communicate with DySER, additional instructions to send and receive floating point values and other data types are required. For example, we may need different DySER instructions to send and receive floating point values, as they are usually stored in a separate register file. If the core's microarchitecture has a wide memory interface that has the capability to load and store vector values, it may be necessary to have vector instructions

```
for(i=0; i<n; ++i) {
 if(a[i]>0)
   c[i] = 1/b[2i];
 else
   c[i] = b[2i]*2;
}
```

(a) Original Loop

```
dyserinit <Config>
...
for(i=0; i<n; ++i) {
  dyserload a[i]->P0;
  dyserload b[2i]->P1;
  dyserstore P2->c[i];
}
...
```

(b) DySER Code



(c) DySER Configuration

Figure 3.2: ISA extensions for DySER

for DySER that load and store multiple data values from DySER with a single instruction. Table 3.2 describes additional instructions that are necessary to support sending and receiving floating point values, vectors, to perform conditional branches on values computed inside DySER, and to send constants to DySER. All of these instructions reduce the number of instructions required to execute in the processor pipeline to support execution with DySER. In addition, vector instructions, which send multiple data elements to DySER with a single instruction, help to feed the functional units faster and hence achieve high energy efficiency.

3.2 DySER Microarchitecture

This section describes the microarchitecture design of DySER that makes it an energy efficient hardware substrate that can create a specialized hardware datapath for a sequence of computations. The main source of DySER's energy efficiency is the design of the functional units and switches,

Instruction	Operation	Equivalent Sequence of operations
dysersendf	Sends data from floating point register to DySER	movfloat2int,dysersend
dyserrecvf	Receives data from floating point register to DySER	dyserrecv,movint2float
dysersend_vec	Sends data from a vector register file to DySER	A sequence of dysersends
dyserrecv_vec	Receives data from DySER and writes it to a vector register file	A sequence of dyserrecvs
dyserload_vec	Loads vector from memory and send directly to a DySER	A sequence of dyserloads
dyserstore_vec	Receives data from DySER port and stores the value to the memory	A sequence of dyserstores
dyserbranch	Conditional branch based on a boolean value received from DySER	dyserrecv, cmp, jmp
dysersend_const	Sends a constant to DySER, which does not change for multiple invocations	A sequence of dysersends with same value to same port

Table 3.2: Additional DySER Instructions for Efficiency

which execute in data flow fashion and consume little dynamic power when they are not operational. DySER's pipelining capability and its ability to amortize the configuration cost also help to achieve high energy efficiency.

Figure 3.3 shows the high level schematic of DySER. DySER consists of an input interface, a compute fabric, and an output interface. The input interface consists of a set of First-In-First-Out (FIFOs) queues that take inputs from outside and deliver the values into the compute fabric. The compute fabric consists of a heterogeneous array of functional units and switches. With these functional units and switches, one can create a specialized hardware data path to perform a sequence of computations. Similar to the input interface, the output interface consists of another set of FIFOs that take the values computed in the compute fabric and deliver them as output. The processor

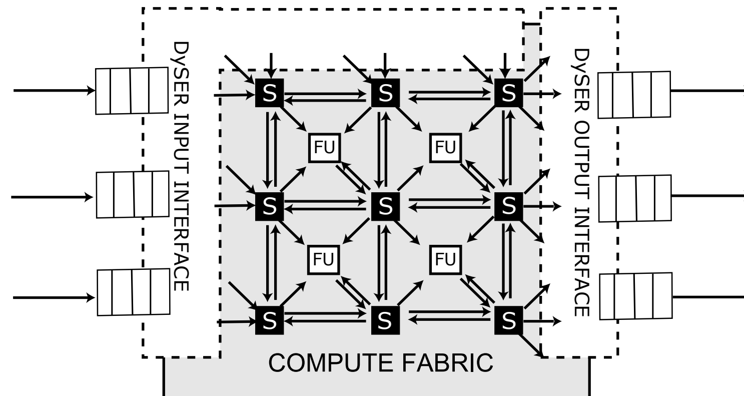


Figure 3.3: Highlevel Schematic of DySER

pipeline communicates to DySER through a set of named input and output ports, which correspond to FIFOs in the input interface and FIFOs in the output interface respectively.

The remainder of this section describes DySER's compute fabric in detail. Subsection 3.2.1 describes the functional units and subsection 3.2.2 describes the switch network. Section 3.2.3 explains the flow control in the switch network that allows pipeline execution and finally section 3.2.4 describes the mechanisms that DySER uses to reconfigure the functional units and switches to create different hardware datapaths.

3.2.1 Functional Unit

Each functional unit is connected to its four neighboring switches in the mesh network as shown in Figure 3.3. It gets its input values from these neighboring switches and delivers its output to a neighboring switch. Figure 3.4 shows the details of a functional unit in DySER.

Each functional unit includes a configuration register that specifies which function to perform and which input values to use as operands. For example, an integer-ALU functional unit in DySER can perform addition, subtraction, and a few logical operations and can use values from any of its neighboring switches as its operands. Using bits in the configuration register as control signals, the functional unit selects its inputs and performs the configured operation.

Each functional unit also has a data and a status register for each of its four input switches. The data registers match the word-size of the machine, and the status register stores two status bits.

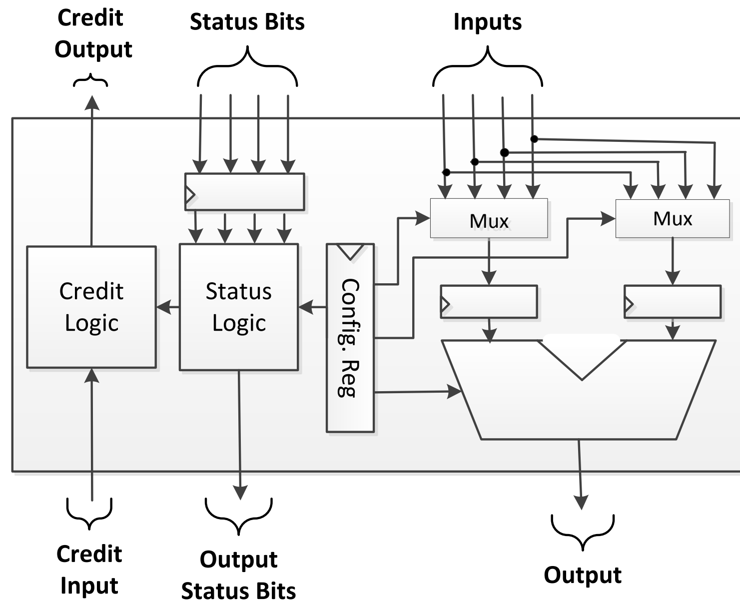
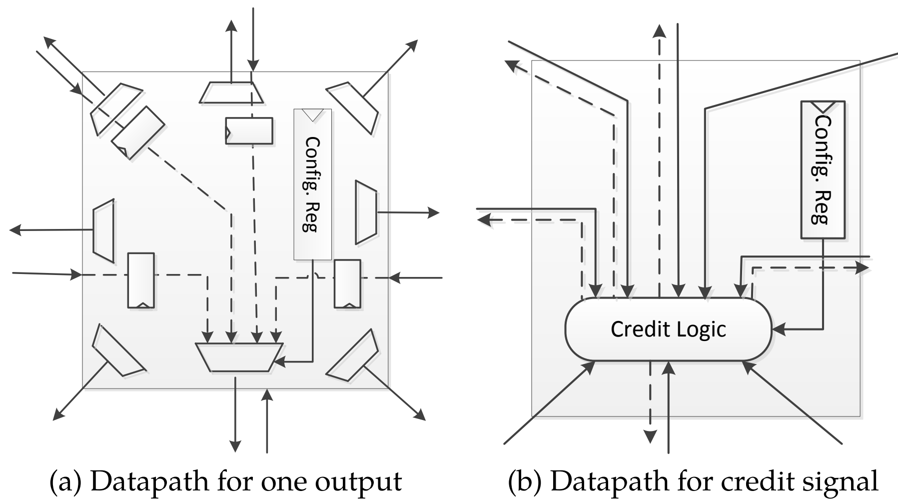


Figure 3.4: Functional Unit in DySER

One bit indicates whether the data is predicated and another indicates whether the data is ready to be consumed. These status bits help to support pipelining and control-flow as described below in subsection 3.2.3 and in subsection 3.5.1 respectively. A functional unit is not enabled until all of its operands are ready. Once all its operands are ready, it performs the configured operation and produces its output. In addition to producing the output, it also generates the status bits for the output. If any of its inputs is invalid but ready, it produces an invalid output, i.e. the ready bit in the output status bit is set but not the valid bit. This logic, represented as the Status Logic in Figure 3.4, is purely computational logic and consumes little dynamic power. For the flow-control, as explained in subsection 3.2.3, it also generates a credit signal for the input switches that indicates the functional unit can accept new inputs.

3.2.2 Switch Network

Switches in DySER allow datapaths to be dynamically specialized, represented by black squares in the Figure 3.3. Switches form a circuit-switched mesh network that creates explicit hardware paths from input FIFOs to the functional units, between functional units, and from functional units to output FIFOs. Figure 3.5a shows the basic switch with the dotted lines representing the possible



(a) Datapath for one output (b) Datapath for credit signal

Figure 3.5: Switches in DySER

connections from all possible inputs to one output. This forms the crux of DySER's capability to dynamically specialize computations. Similar to the functional units, each switch includes a configuration register, which specifies the input to output port mappings, data registers and status registers. Switches in DySER have eight outputs for eight directions, four inputs from neighboring switches and one input from one of its neighboring functional unit. In addition, switches forward the status bits of the inputs as the status bits of outputs without any changes. However, they generate a credit signal for an input only when all credits for its corresponding outputs are set. Figure 3.5b shows the credit signal path through a switch.

3.2.3 Pipelined Execution and Flow Control

The basic execution inside DySER is data-flow driven by values arriving at a functional unit. When the ready bits for both left and right operands are set, the functional unit consumes its operands, and a fixed number of cycles later produces output, which is written directly into the output switch's corresponding input register. Similar to the functional units in DySER, a switch in DySER only consumes the input when it is ready. If so, it forwards the input to a neighboring functional unit or to a neighboring switch. This data-flow execution continues until the values reach DySER's output interface.

Similar to pipelining long-latency functional units, specialized datapaths inside DySER can be

pipelined with multiple invocations executing simultaneously. Since DySER receives the inputs at arbitrary times from the input interface, some form of flow-control is required to prevent values from a new invocation clobbering values from a previous invocations. DySER implements a simple credit based flow control optimized for the statically switched network by a signal called “credit”. Figure 3.4 and Figure 3.5b shows the details of the credit signal datapath in a functional unit and a switch respectively. Physically, the credit signal is routed in the opposite direction of the data signal and the status bits. A stage in the pipeline, either a functional unit or switch, needs a credit signal from the next stage to send data. After sending the output to the next stage, it sends a credit signal to its predecessor in the pipeline. If a stage is processing or waiting for data, the ready bit is cleared, and the credit is not passed to the previous stage.

3.2.4 Configuration

DySER is configured to create specialized datapaths for computation by writing into configuration registers at each functional unit and switch. DySER uses the data network itself to transmit configuration bits as shown in Figure 3.6. Every switch also includes a decoder and a path from the switch’s inputs to its configuration register. The data routed to the switch is interpreted as a 3-bit target and 29-bit payload data when the switch is in configuration mode. The switch uses the decoder to check if the message is meant for the current node by examining the target field, and if so, the value is written into the configuration registers. In addition, all configuration messages are forwarded to the next switch. When switches are in the configuration mode, they forward the input from the east port to the west port and the input from the north port to south port. With this design, DySER is configured using datapath wires without any dedicated configuration wires. DySER is configured once and re-used many times and thus it amortizes the configuration cost.

3.3 Processor Interface

This section describes how DySER integrates with an existing processor microarchitecture. Since DySER is internally stateless and exposes a well-defined interface, it can be integrated with an existing microarchitecture as a long latency functional unit. In addition, the changes to the microar-

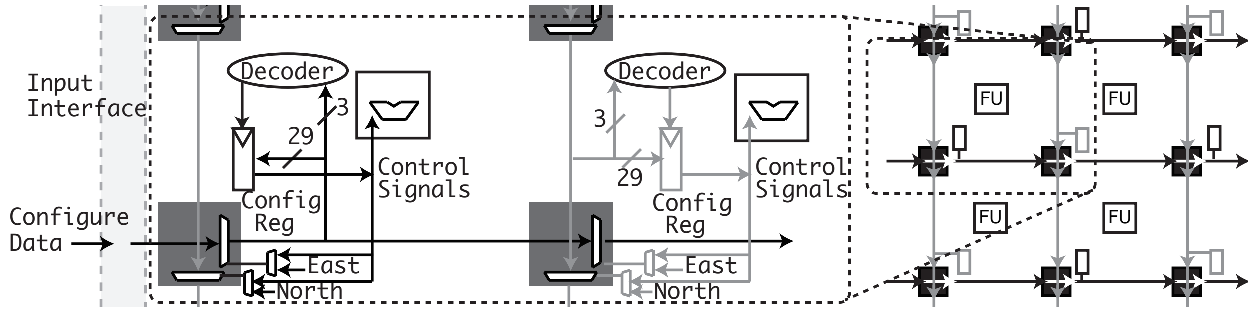


Figure 3.6: Configuration path: switch and functional unit's configuration registers combined

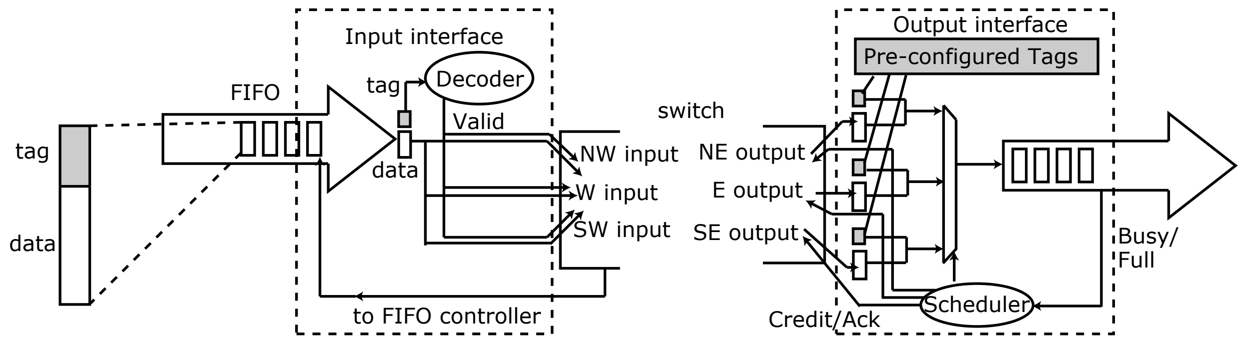


Figure 3.7: DySER interface to Processor pipeline

chitecture are small and manageable, because the co-designed compiler specializes the program and creates DySER configuration bits that encode the custom datapath at compile-time.

Figure 3.7 shows a logical FIFO-based processor interface to DySER. The processor pipeline communicates to DySER through a set of named input and output ports, which correspond to FIFOs. These FIFOs deliver or receive data from the switches at the edge of the network. All of the inputs to DySER are fed through a logical FIFO, which delivers register inputs and memory values. Each entry specifies a port, which effectively decides where the value will be delivered in the array, because DySER uses circuit-switched routing. Outputs follow a similar procedure. Each port of the output switches corresponds to one possible DySER output. From the output port, the processor can fetch the data and send it to either the register file or store the data to memory.

The rest of this section is organized as follows: Subsection 3.3.1 presents the input interface and the subsection 3.3.2 describes the output interface in detail. Subsection 3.3.3 presents the integration of DySER with a processor pipeline. Finally, subsection 3.3.4 explains how DySER handles page

faults and other exceptions that may occur during a program execution.

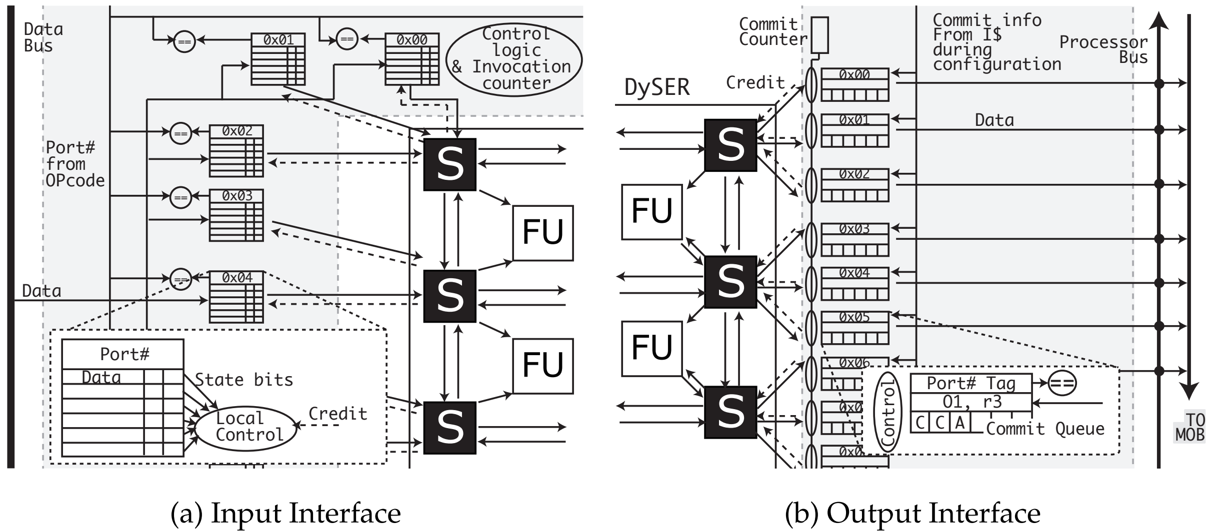
3.3.1 Input Interface

Figure 3.8a shows the details of the input interface and how it is connected to the switches in the compute fabric of DySER. The logical FIFO that receives values from the processor is physically partitioned into multiple banks that correspond to a row and a column of the array. Each bank is implemented as a circular buffer to support multiple invocations. Each buffer also includes a port number and a decoder which is used to obtain values from the data bus. Since each switch in the two sides of the array has two inputs, a total of $4n$ inputs can be injected into an $n \times n$ array of functional units.

Each buffer entry consists of two state bits and data. The four possible states of a buffer entry are: *i*) ready indicates input data is ready for DySER to consume, *ii*) invalid-but-ready indicates the data is invalid, but ready to be consumed. This data will be discarded either by DySER itself or at the output interface if the data or the results that are computed with the invaid data reached the output interface, *iii*) busy indicates the input is issued in the processor, but delayed by a cache miss and hence cannot be consumed by DySER yet, and *iv*) empty indicates no data. Switches check these status bits and consume data if available and propagate the status signals “ready” and “valid” accordingly.

3.3.2 Output Interface

Logically, output values from DySER are held at output ports until a `dyserrecv` or `dyserstore` instruction is issued for that invocation. The network flow control guarantees that they will not be overwritten by values from successive invocations. Figure 3.8b shows the implementation of the output interface. The output interface consists of a commit counter, which counts the values committed for the current invocation, and control logic for each port. Each output control logic consists of a queue that maintains the status of the output port for each invocation. There are three possible values for the status: *i*) *commit* denotes that the output is consumed by a instruction that is already committed, *ii*) *abort* denotes that the output values are ready but must be removed



(a) Input Interface (b) Output Interface

Figure 3.8: DySER's Processor Interface

from the port and ignored and *iii*) *done* indicates that the output is consumed by an instruction. When a `dyserrecv` or a `dyserstore` instruction read the value from the output port, the status is changed to *done*. When the instruction that received the value is committed, the output port is marked as *commit* and the commit counter for the current invocation is advanced. If the instruction that received the output is squashed and reexecuted, the output interface guarantees that it will still get the correct value using the status bits. If the status of an output port is abort, the internal control in output interface automatically removes the output. When the commit counter reaches the number of outputs for current invocation, the input circular buffer is advanced, because values in the head of the buffer are no longer needed.

3.3.3 Integration with pipeline

DySER can be easily integrated into a conventional in-order or out-of-order pipeline as a compound functional unit. With an in-order pipeline, the integration is relatively simple. DySER simply needs to interface with the instruction fetch stage for obtaining the configuration bits, the register file stage and the memory stage of the pipeline.

DySER integration with an out-of-order pipeline requires a more careful design. The processor views DySER as a functional unit, but the input ports must be exposed to the issue logic to ensure

two `dysersend`s to a port are not executed out-of-order. Since loads can cause cache misses, when a `dyserload` executes in the processor, the corresponding input port is marked busy in the input buffers. When the data arrives from the cache, the input port is marked ready, which prevents a subsequent `dyserload`'s value from entering DySER earlier.

When a branch is mispredicted and misspeculated values are sent to DySER, the values computed in DySER must be squashed. Since DySER modifies the architecture state only through its output, we can simply ignore the outputs of DySER for the misspeculated inputs. However, DySER may need additional inputs to compute all of its outputs. This is implemented by sending invalid data as inputs to complete the invocation of DySER. This ensures that all inputs are available for DySER to compute and produce outputs. In order to restart the computation with correct values, the non-speculated inputs for an invocation are stored in an input buffer until the invocation is completed. DySER restarts the computations using the values in the input buffer and injecting new, correct values.

3.3.4 Page Faults, Context-switches etc.,

Since all memory accesses execute in the main processor, page faults can only be raised by instructions executing in the processor and almost no changes are required to the processor's existing mechanisms. The processor services the page-fault and resumes execution from the memory instruction that caused the fault. The OS routine to handle the page-faults is assumed not to use DySER. To handle context-switches, the processor waits until all DySER invocations are complete before allowing the operating system to swap in a new process. Operating systems consider the configuration in DySER and the data in the input and output ports of DySER as architectural state and store them as part of process context. Since DySER itself is stateless, restarting DySER after a context switch is the same as restarting DySER after a pipeline squash.

3.4 Integration Feasibility Study

To demonstrate the feasibility of DySER integration to an existing conventional microarchitecture, I jointly worked with six fellow graduate students to integrate a prototype of DySER into the

Configuration	Operation	Comments
Default	out = in1 <op> in2 out = <op> in1	Binary operation Unary operation
Predication	out = in1 <op> in2, if in3 out = <invalid>, otherwise	
Inverted Predication	out = in1 <op> in2, if !in3 out = <invalid>, otherwise	
ϕ-$function$	out = (in1.valid) ? in1 : in2	
Select	out = (in3) ? in1 : in2	Conditional Move

Table 3.3: Functional Unit operations for control-flow

OpenSPARC processor. It supports SPARC ISA extensions for DySER to send and receive values from DySER. We verified the implementation on an off-the-shelf Virtex-5 FPGA board booting unmodified Linux and running applications. The leasons learned and other implementation details were presented in the following publications [8, 7, 62]

3.5 Hardware Mechanisms for Efficiency

The major source of efficiency in DySER is its flexible hardware substrate that can create a specialized hardware datapath and execute computation in data flow fashion. In addition, DySER has several mechanisms that make it an efficient accelerator for both control intensive workloads and data parallel workloads. This section elaborates on these mechanisms. We describe how the circuit switch network and a special functional units work together to natively support control-flow inside DySER. We then elaborate on vectorization support of DySER, which uses the wide memory interface and flexible I/O to make DySER a suitable substrate for specializing data parallel workloads. Finally, we present the additional mechanisms and changes to DySER's baseline microarchitecture to virtualize the resources in DySER to map large code regions to DySER and improve the utilization of DySER.

3.5.1 Control Flow: Predication and ϕ-functions

Many accelerators do not allow internal control-flow in the specialized code. DySER allows control-flow and simplifies the mapping of a large region of code to DySER. DySER natively supports control flow using two mechanisms. First, functional units accept an extra input that predicates the

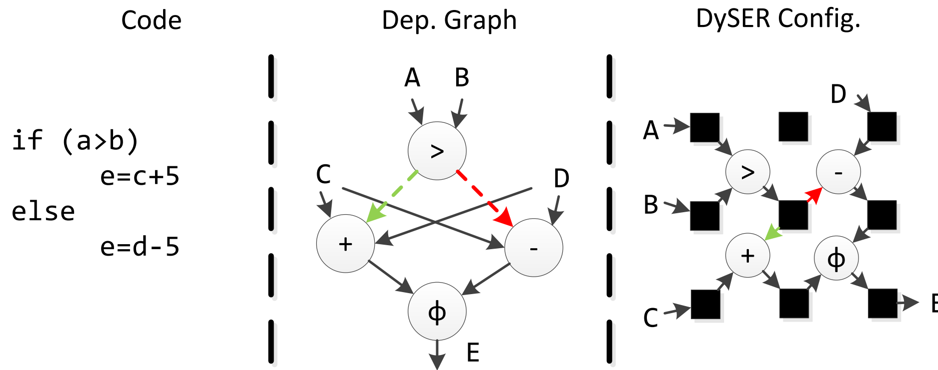


Figure 3.9: Control flow support in DySER

operation of the functional unit. If the predicate input is not asserted, the functional unit generates an invalid output, which means the ready bit is set but not the valid bit. The DySER compiler uses if-conversion to convert control-flow into data-flow and map the predicated instructions also to DySER. Second, DySER provides the ability to perform select operations depending upon the validity of the input data. This select operation is implemented in a separate functional unit and is similar to the ϕ-functions in the Single Static Assignment (SSA) [26] form of the code. (A ϕ-function in SSA controls which value to use when the value has multiple producers.) Using predication and ϕ-functions, DySER can specialize code with internal control flow. Finally, it also has a functional unit that perform a selection operation, which corresponds to a conditional move. Table 3.3 summarizes the control-flow operations of a functional unit in DySER.

Figure 3.9 shows a simple code snippet that has control-flow and how DySER uses predication and a ϕ-function to specialize the code. The compare functional unit in DySER generates a boolean output. Using the result of the compare, the functional units that perform addition and subtraction generate their output. This guarantees that at most one operand to the ϕ-$function$ is valid. Then, the ϕ-function selects its valid operand as its output, which is the desired result.

3.5.2 Vectorization: Flexible Vector Interface

In order to specialize data parallel workloads efficiently, we add DySER vector instructions to send and receive vectors. With the aid of vector instructions, the main processor can send multiple data elements to DySER with just one instruction and increase the utilization of DySER resources.

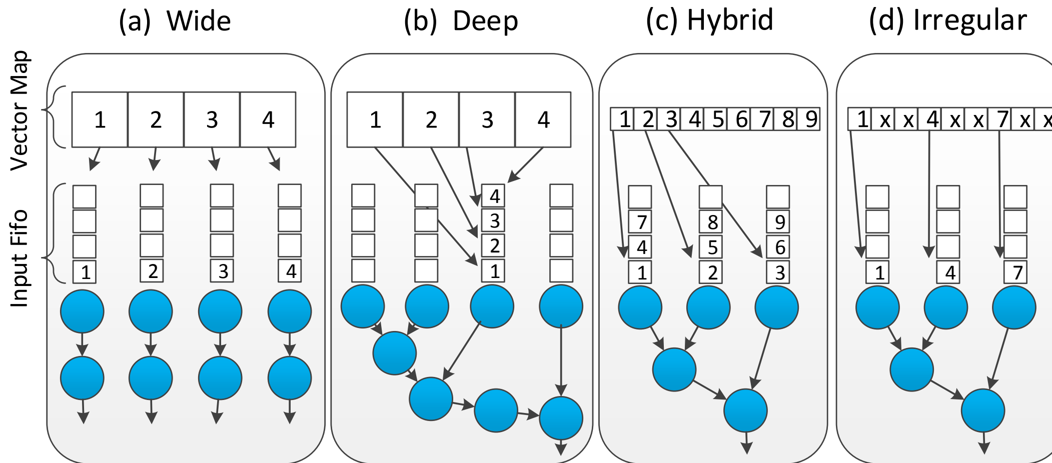


Figure 3.10: Flexible Vector I/O Mechanisms in DySER

SIMD instructions or vector instructions, without the support of scatter/gather units, as is the case for most modern SIMD implementations like SSE/AVX, can only load and store contiguous sections of memory. This simplifies hardware for fetching vectors from the memory system by only requiring one cache line fetch, or perhaps two if unaligned access is supported. DySER retains this simplicity by requiring contiguous memory on vector loads, but provides a flexible vector I/O mechanism, which maps locations in a vector to arbitrary ports in DySER. To support this, DySER's configuration includes vector port definitions, which map sequences of DySER's ports to virtual vector ports [45, 46].

This mapping mechanism allows DySER to utilize vector instructions for communication in different paradigms, as shown in Figure 3.10. First, when the elements of a vector correspond to different elements of the computation, this is a "wide" communication pattern (Fig. 3.10a). This is most similar to SIMD's vector interface. When the elements of a vector correspond to the same element of the computation, this is a "deep" communication pattern (Fig. 3.10b). This corresponds to explicitly pipelining a computation. The combination of the wide communication pattern and deep communication pattern results in a "hybrid" pattern (Fig. 3.10c). Finally, when certain vector elements are masked-off or ignored, this is an "irregular" pattern (Fig. 3.10d). The flexibility of DySER's I/O interface, in part, gives rise to the need for a sophisticated compiler intermediate representation, described in the next chapter.

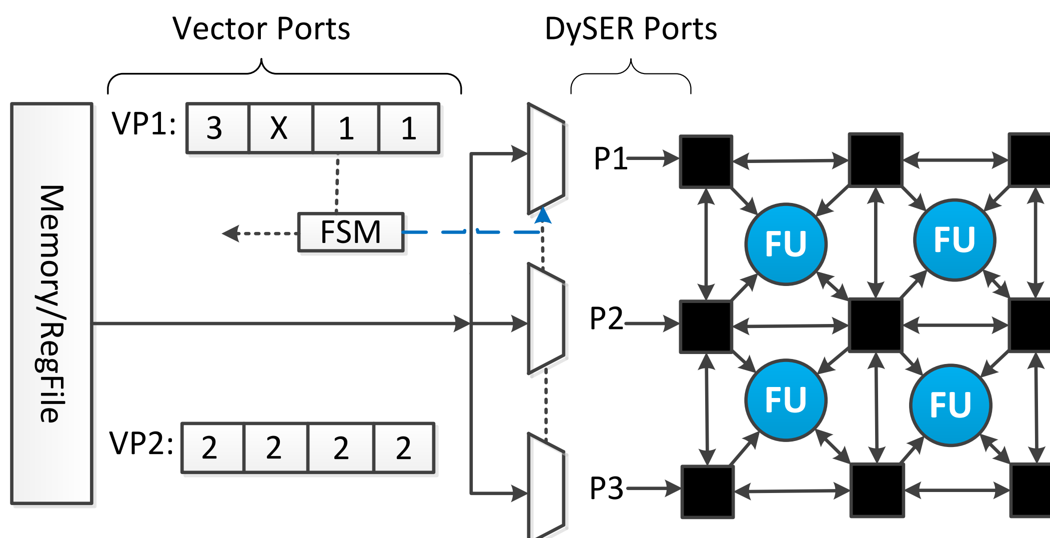


Figure 3.11: Implementation of vector ports in DySER

Implementation: To implement vectorized communication in hardware, we first require a wide memory interface similar to that of SSE, and a number of named vector ports in DySER. Additionally, DySER needs a mechanism to map vector input/output values to or from DySER's internal input/output ports. We call the information that conveys this correspondence the "vector map". The configuration is augmented with additional bits, which specify a vector map for each vector port. When a vectorized DySER instruction accesses these ports, a finite state machine (FSM) in DySER's I/O interface coordinates the transfer of data between the incoming values from the register file or memory and DySER's internal ports. Figure 3.11 shows the implementation of vectorized communication. The FSM in DySER's I/O interface uses the vector map bits to generate control signals which select the appropriate DySER port. In each subsequent cycle, the next value in the vector map is utilized. In the example shown, vector port 1 routes the first and second memory values to DySER port 1, the third memory value is ignored because it is masked off, and finally the fourth memory value is sent to DySER port 3. A similar vector mapping FSM is required on the output interface as well. In this implementation, it takes N cycles to map an N element vector. Though faster implementations are possible, and can have an impact on performance, their description and evaluation is beyond the scope of this thesis.

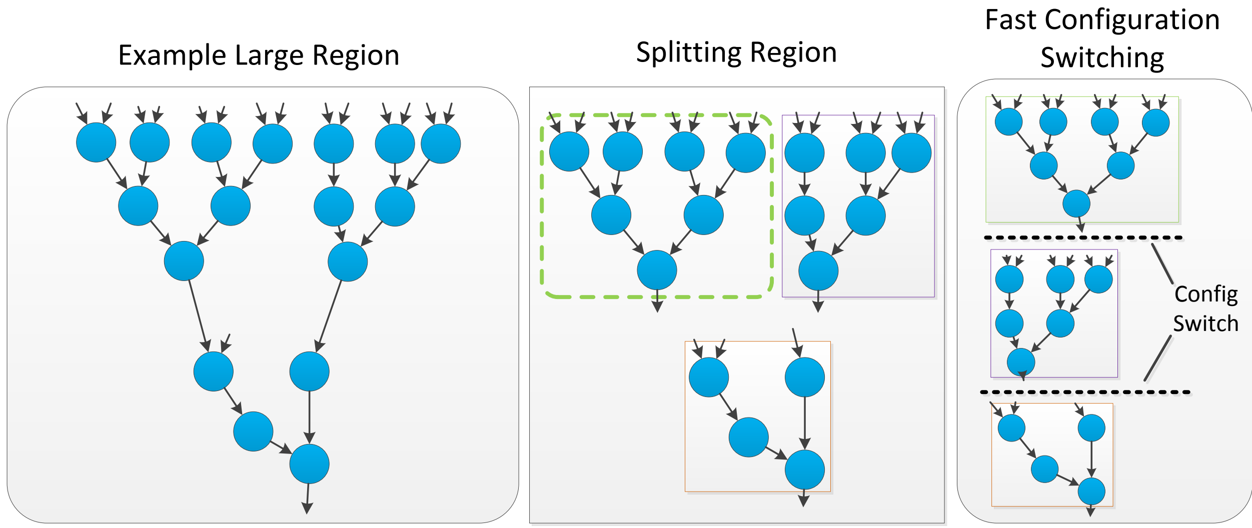


Figure 3.12: Large region with Subgraph Matching and Fast Configuration Switching in DySER

3.5.3 Virtualization: Fast Configuration Switching

To achieve high utilization of DySER and amortize the cost of reconfiguration, DySER should map computation regions with appropriate size. When a overly large region of computation is mapped to DySER, only a portion of the computation can be mapped to DySER because of resource limitation and remaining computation executed in the main processor pipeline. To avoid using the main processor for computation, overly large regions must be "split" into multiple regions to fit inside DySER to achieve high utilization. Compared to instruction-level acceleration, DySER's dynamic customization introduces resource limitation challenges, which DySER overcomes by using the technique described below.

Fast Configuration Switching: When the code region is huge, the region is split into multiple small regions. Figure 3.12 shows how a large computation subregion can be cut into components of appropriate size and mapped to DySER by using multiple configurations. To reduce the configuration penalty when switching between the configuration, DySER uses a hardware technique to reduce the configuration penalty, but with additional hardware cost.

DySER enables fast configuration switching through two hardware mechanisms. First, it augments every DySER tile (a functional unit or a switch) with the ability to store multiple configurations.

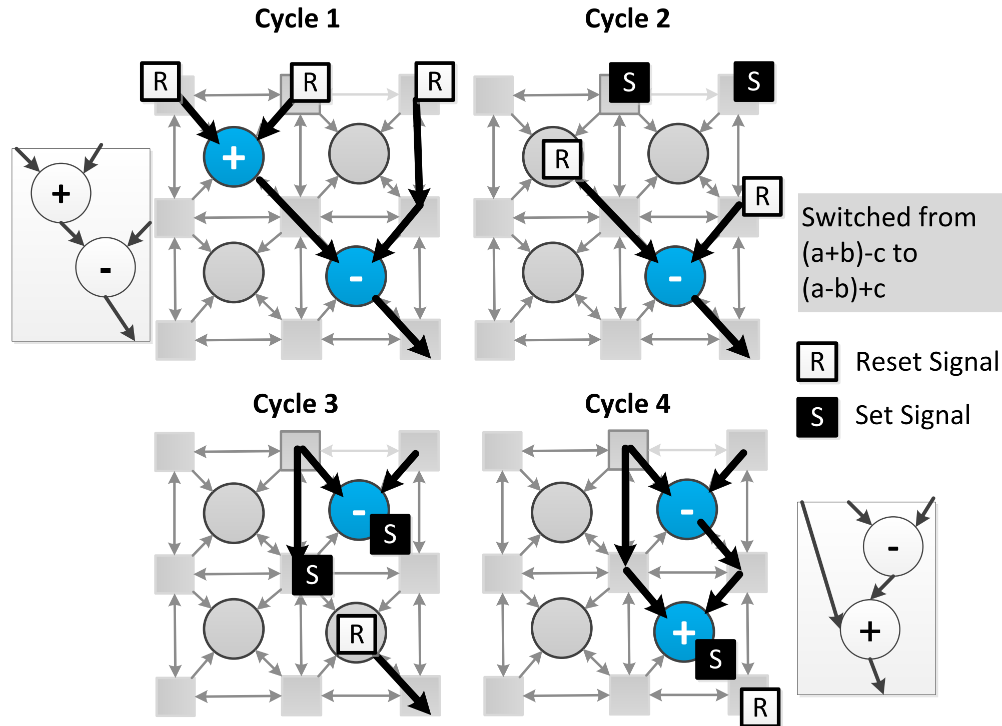


Figure 3.13: Fast Configuration Switching Example and Implementation

Second, DySER employs a configuration switch protocol that relies on each functional unit and switch being either in an active or off state. It adds a 1-bit "free" signal to the network, which is set from the eight neighbors of a DySER tile. In addition, it adds a DySER instruction that switches the configurations that are already stored inside DySER. This instruction sends reset signals through the *old* configuration and set signals through the *new* configuration's input ports. The reset signal forces every tile that has finished computation into the off state, triggering them into sending "free" signals to neighbors. The set signals that follow the reset signals then change any off-state tile into the new configuation. Each set signal propagates to all neighbors in the *new* configuation after receiving their free signals.

This protocol explicitly reuses the dataflow in the two regions to synchronize the set and reset signals without any additional networks or compiler requirements. If a DySER tile is used in the previous configuration, it will not send a free signal until it is explicitly resetted by reset signal. Since set signal will not propagate until the neighboring tile sends the free signal, the reset and set signal will automatically synchorize themselves, and set follows the reset signal. This and the fact

that the protocol do not allow multiple sets and resets make this protocol deadlock free. As soon as all data for an invocation has been sent, reset and set signals can also be sent. Figure 3.13(a) shows an example of the set and reset signals performing the configuration switching.

3.6 Chapter Summary

This chapter presented the architecture of DySER, microarchitecture and the processor interface of DySER. With DySER, it is possible to a create specialized hardware datapath dynamically at run time by sending configuration bits to switches and functional units. The functionality of DySER is exposed through a well defined interface to the existing processor microarchitecture. This chapter also elaborated on the hardware mechanisms in DySER that makes it an efficient accelerator for both data parallel and control intensive workloads. The mechanisms from this chapter have appeared in HPCA-2011 [47], HPCA-2012 [8], IEEE Micro Sep/Oct-2012 [45], Hot Chips 2012 [7] and will also appear in Ho's dissertation [61].

4 Compiling for DySER

As described in the previous chapter, the Dynamically Specialized Execution Resources (DySER) architecture provides a flexible, reconfigurable substrate of functional units and switches, which offers the possibility of high performance with energy efficiency. In the dynamically specialized execution model, the compiler identifies code regions that can be specialized and partitions them into two subregions: the memory subregion, which consists of the address calculations, loads, and stores; and the compute subregion, which consists of other computations. The memory subregion executes in the main processor pipeline, and the compute subregion executes inside DySER. The instructions that are executing inside DySER communicate directly through the circuit switched network, unlike the instructions that are executing in the processor pipeline, which communicate through shared registers.

DySER imposes several restrictions on the execute subregion to simplify its interface to the processor and its microarchitecture. The execute subregion cannot be more than the size of DySER. For example, if the DySER has 64 functional units, then it cannot have more than 32 inputs and 32 outputs. Also, the execute subregion should map to the circuit switched mesh network of DySER. The memory subregions should not only load and store data to memory, but also manage the DySER properly. They should amortize the configuration cost by reusing the specialized datapath multiple times and feed DySER with operands as quickly as possible to utilize the resources in DySER better.

The goal of the DySER compiler is to generate the execute subregion with useful instructions while optimizing the memory subregion to continue feeding inputs quickly to the instructions in the execute subregion. This chapter presents the DySER compiler, which meets these goals by using a set of code transformations, including an algorithm that maps the execute subregion effectively to

the DySER substrate. In order to do optimizations on both the memory subregion and the compute subregion, this chapter develops an intermediate representation called the Access/Execute Program Dependence Graph (AEPDG). The DySER compiler models DySER's pipelined execution and the memory access through the instructions executing in the main processor pipeline using the AEPDG. It performs optimizations on AEPDG to make the execute subregion optimized for DySER and finally schedules the execute region to DySER.

4.1 Overview of the DySER Compiler

This section presents an overview of the DySER compiler. The DySER compiler takes a program written in a high level language like C/C++ and generates optimized code for DySER that uses DySER instructions to communicate to DySER. Figure 4.1 shows the high level compilation flow of the DySER compiler, which has four main phases. The figure also shows the intermediate representation the compiler uses at each phase to model the program or the function under compilation.

Phase I of the DySER compiler is the C/C++ frontend, which takes C/C++ source code as input and generates the control-flow graph (CFG). In addition to generating the control-flow graph, it also performs standard scalar optimizations such as Dead Code Elimination, Constant Propagation, and Loop Invariant Code Motion. Phase II takes the CFG as the input and generates the Access/Execute Program Dependence Graph (AEPDG), an intermediate representation that is more suited for generating optimized code for DySER, by slicing the program dependence graph into two subregions called access-PDG and execute-PDG. Phase III performs optimizations on the AEPDG, including loop unrolling, region cloning, vectorization, and applying peephole optimizations that are aware of DySER's resource's capability and availability. Finally, phase IV schedules the portion of the AEPDG (execute-PDG) that executes inside DySER to the functional units and switches in DySER. It also inserts DySER instructions to communicate to DySER and compiles the portion of AEPDG that executes in the processor pipeline to assembly code.

The rest of this chapter is organized as follows: Section 4.2 describes the Access/Execute Program Dependence Graph and explains why the DySER compiler needs a new intermediate representation. Section 4.3 describes the phase II and presents an algorithm to construct on AEPDG

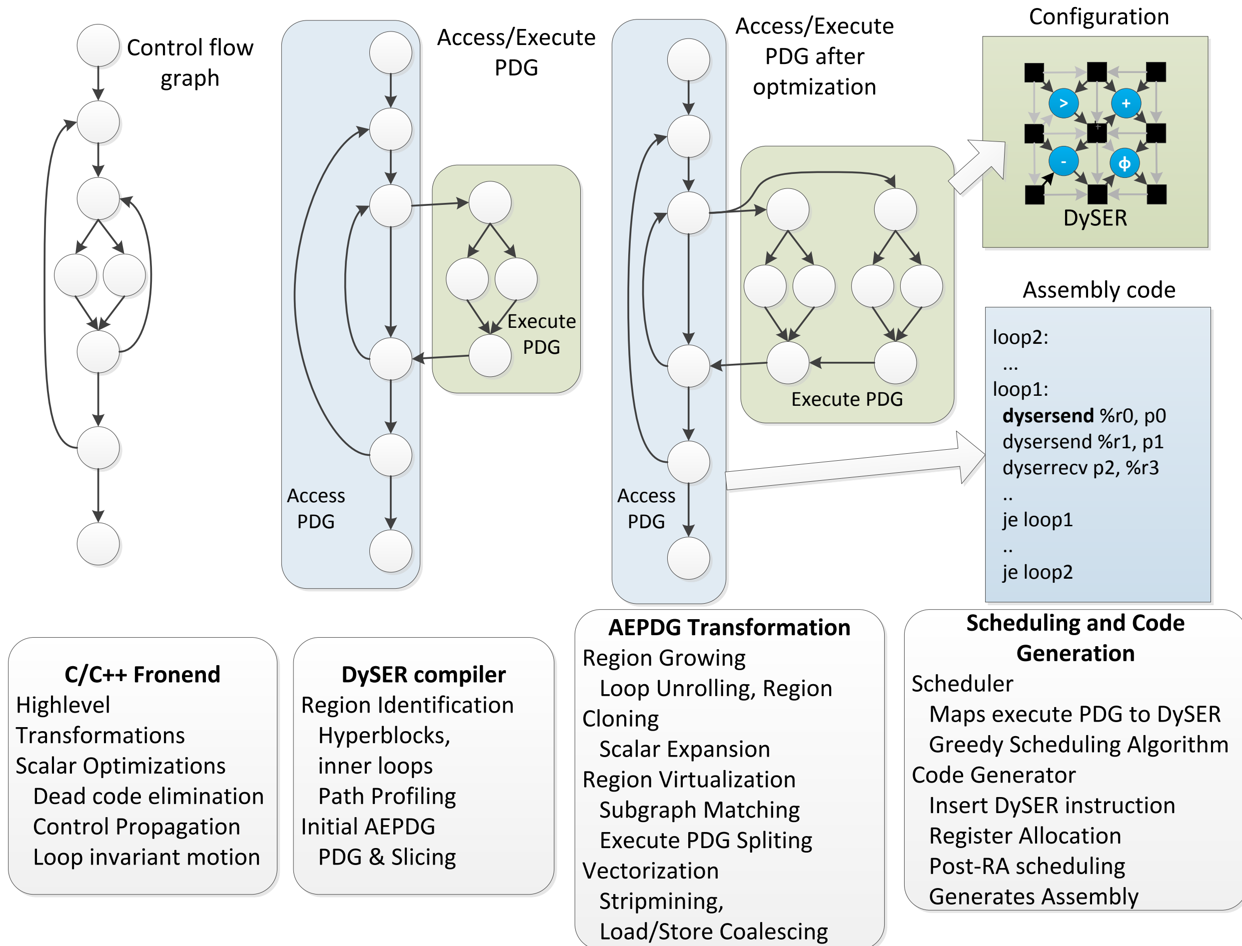


Figure 4.1: DySER Compiler Overview

from the program dependence graph of the code region. Section 4.4 presents the transformations required to transform an arbitrary AEPDG so that it will have an ideal execute-PDG that can be specialized with DySER efficiently. Section 4.5 describes algorithms required to transform the AEPDG, and Section 4.6 presents the scheduler and code generator of the DySER compiler and Section 4.7 describes the implementation of the DySER. Section 4.8 presents a case study on challenging code-regions and shows how the transformations on the AEPDG naturally tackles them. Section 4.9 concludes the chapter.

4.2 Access Execute PDG

Optimizing compilers should capture the aspects of the underlying architecture accurately to generate efficient code for the architecture. With an appropriate model of the architecture, the compiler can make legal transformations to an unoptimized program and generate optimized code for the architecture. The motivation to develop a new compiler intermediate representation for DySER is that the intermediate representation must model the internal operation of DySER through both spatial (via contiguous memory access using vector instructions in main processor pipeline) and temporal (via pipelined execution inside DySER) dimensions.

This section first presents the Access/Execute Program Dependence Graph (AEPDG), which extends traditional PDGs with special nodes and edges to capture spatial and temporal aspects of the program. Then, we enumerate the characteristics of AEPDGs which facilitate efficient mapping to DySER. Section 4.3 describes an algorithm that takes a program dependence graph as input and constructs the Access/Execute Program Dependence Graph. Section 4.4 presents the strategies to generate the optimized AEPDG from the initial AEPDG.

4.2.1 A Case for a New Program Representation

The DySER compiler's intermediate representation should be rich enough to express DySER's capabilities, especially the flexible mechanisms that are available in DySER to achieve high efficiency. Specifically, the compiler should model the decoupled access/execute execution model of DySER and the following three flexible mechanisms that DySER provides: i) configurable pipelined datapaths; ii) native control capability; and iii) a flexible vector I/O interface. Below, we elaborate on these mechanisms and how they define the role of the compiler and its intermediate representation.

Configurable Datapath DySER provides a way to create a specialized datapath to perform multiple operations in parallel. Complex dependencies can be expressed inside the DySER hardware substrate, which is an array of heterogeneous functional units and switches. *Depending upon the characteristics of the computation, the compiler should create independent lanes to exploit fine grain data parallelism or to exploit instruction level parallelism.*

Native Control Mapping As described in subsection 3.5.1, DySER provides the support for control instructions by augmenting the internal datapath with a status bit called "valid", and provides the ability to perform select operations depending upon the value of this bit using the ϕ-functional unit. *The DySER compiler should map the control instructions to the predicate based control-flow*.

Flexible Vector I/O In order to feed the functional units quickly with useful data and achieve high utilization of resources in DySER, it is necessary to use vector instructions to load and store vectors directly from DySER's ports. Subsection 3.5.2 describes the flexible I/O mechanism in DySER, which can map locations in an I/O vector to arbitrary ports in DySER using vector port definitions. Using the flexible I/O, DySER can consume the vector for different communication patterns: wide, deep, hybrid, and irregular. *The DySER compiler and its intermediate representation should model this flexible vector I/O and the communication patterns to generate optimized code for data parallel workloads*.

In order to be an effective compiler, the DySER compiler should model the three flexible mechanisms described above. The first two, the configurable datapath and the control capability, can be sufficiently represented by the well known Program Dependence Graph (PDG) [38]. This representation makes explicit the data and control dependencies between instructions, which incidentally closely match DySER's internal execution model. This graph can be partitioned into a subgraph which is executed on the main processor, and another subgraph which executes purely on the DySER hardware. When targeting applications which can only utilize *scalar* access to the memory and register files, the PDG aptly captures program behavior. However, since DySER supports the flexible I/O interface, the PDG fails to capture the decoupled nature of DySER execution.

The PDG does not explicitly capture the notion of spatial access, meaning that it is unaware of the potentially contiguous access for a computation. Also, it does not have a notion of temporal execution, which corresponds to pipelining computations through the accelerator. More fundamentally, the PDG lacks a representation for the relationship between spatial access to the memory and temporal execution in the accelerator, i.e. the PDG is unaware of the correspondence between the contiguity of inputs and outputs of a particular computation through subsequent iterations. To address this shortcoming, we develop the AEPDG, which captures exactly this relationship with special edges in the PDG. Figure 4.2 shows the mechanisms in DySER architecture and how the

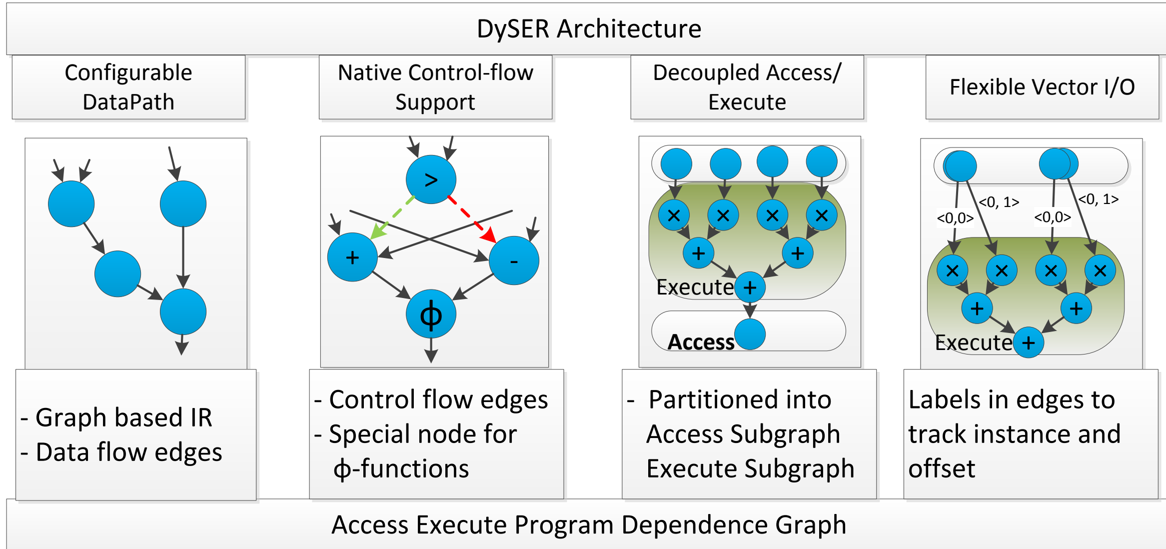


Figure 4.2: A Case for Access Execute Program Dependence Graph

AEPDG models the mechanisms.

4.2.2 Definition and Description

In this subsection, we define the Access/Execute Program Dependence Graph and describe how it differs from the traditional program dependence graph.

The AEPDG is simply a traditional PDG, partitioned into an *access-PDG* and an *execute-PDG*. The execute-PDG is defined as a subgraph of the AEPDG which is executed purely on the DySER hardware substrate. The access-PDG is simply the remaining portion of the AEPDG. Additionally, the AEPDG is augmented with one or many (instance, offset) pairs at each interface edge between the access and execute PDGs. The "instance" identifies the ordering of the value into the computation, and the "offset" describes the distance from the base of the memory address. This decoupling and added information allows the compiler to efficiently coordinate pipelined instances of DySER invocations through the DySER's flexible vectorized interface. Using these pairs, the compiler creates a vector map, which associates a virtual vector port to a sequence of DySER's ports automatically.

More formally, an access/execute program dependence graph G is defined as an 8-tuple $G =$

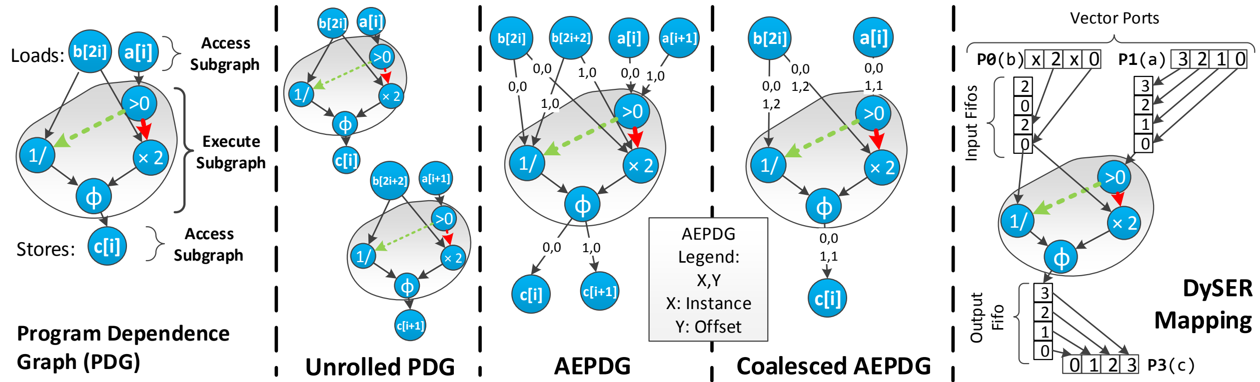


Figure 4.3: AEPDG Example

$(V, E_{df}, E_{cf}, C, A, X, I, L)$ where

- V is a finite, non-empty set of operations or instructions that represent the vertices of the graph.
- $E_{df} \subseteq V \times V$ is a set of edges that represent the data flow edges.
- $E_{cf} \subseteq V \times V$ is a set of edges that represent the control-flow edges.
- $C : E \rightarrow \{true, false\}$ is a function that maps the control flow edges to a boolean label.
- $A \subseteq V$ is the set of instructions in the access program dependence graph.
- $X \subseteq V$ is the set of instructions in the execute program dependence graph.
- $I \subseteq E_{df} \cup E_{cf}$ is the set of edges in the interface between the access and execute PDG. i.e, $(v_1, v_2) \in I$ if and only if $(v_1 \in A \wedge v_2 \in X) \vee (v_1 \in X \wedge v_2 \in A)$
- $L : I \times \{1, 2, ..., n\} \rightarrow \{(instance, offset) | instance \in N \wedge offset \in N\}$ is a function that maps the interface edges to n labels that specifies the instance and the offset.

4.2.3 An Example

Figure 4.3 shows a situation where the PDG fails to capture the program behavior we need to model and illustrates the usefulness of the AEPDG. It shows the traditional PDG on the left pane, which corresponds to the original loop in Figure 3.2. In order to exploit the data parallelism in the loop, we

can perform unrolling, which results in the "Unrolled PDG" in the second pane of figure 4.3. Note how this traditional PDG representation lacks awareness of the relationship between contiguous inputs and pipelineable computations. We construct the AEPDG by determining the relationship between memory accesses through iterations of the loop. Here each edge between the access and execute PDGs has an instance number and offset number. In the "AEPDG" pane of Figure 4.3, all offset numbers are 0, because the loads and stores have not been coalesced. The next pane shows how the AEPDG keeps track of multiple instances of the computation through subsequent iterations. Some edges have multiple pairs, indicating multiple loads from the same address, and some computations are for two separate instances, indicating pipelined execution. The final pane, "DySER Mapping", shows how it is now simple to configure DySER's flexible I/O interface using the AEPDG. Each access pattern is simply given a vector port, which, when utilized by an I/O instruction, initiates a hardware mapping between the vector port and the corresponding DySER port(s).

4.2.4 Characteristics of good AEPDGs

The partitioning of the AEPDG into access-PDG and execute-PDG is not necessarily fixed. The Execute-PDG can be selected differently by simply choosing different instructions, or it can be influenced by transformations performed on the access PDG. Since the execute-PDG is the candidate for the specialization with DySER, it must carefully transform the AEPDG in a way that creates execute-PDGs which are amenable to achieving efficiency when scheduled to DySER. Because of the DySER architecture's spatial computation and its decoupled access/execute model, in order to achieve high efficiency, the compiler should be aware of the following properties of execute-PDGs:

1. **Size:** The number/type of operations in the execute-PDG should be proportional to the accelerator's resources. Too large means many reconfigurations are necessary, while too small prevents a high utilization.

2. **Schedulability:** The compiler should schedule the execute-PDG to the hardware substrate efficiently, meaning that the extra latency introduced by the scheduling of dependencies in the routing network should be as small as possible.

3. **Interface:** The execute-PDG should connect to the access-PDG with the least possible edges, in order to minimize the cost of excess communication instructions. Also, if possible, the compiler should create memory inputs and outputs with spatial locality so that vectorized I/O can amortize the overheads of communication.

Section 4.4 presents strategies and optimizations to transform an arbitrary AEPDG to an optimized AEPDG so that that the execute-PDG will have the above properties. The next section describes how the DySER compiler identifies the region to accelerate and an algorithm that forms the initial AEPDG from the control-flow graph.

4.3 Initial AEPDG Formation

This section describes the Phase II of the DySER compiler, which takes an optimized control-flow graph of a function and constructs the access/execute program dependence graph. First, the DySER compiler needs to select the candidate code region for specialization. Then, it builds the AEPDG for that code region by slicing the code region into the memory subregion and compute subregion.

4.3.1 Region Selection for Acceleration

Applications have many candidate code regions that can be specialized such as functions, loops, basic blocks, a set of basic blocks etc., However, since DySER specializes the code region dynamically using configuration bits at run-time, it is beneficial to identify the most frequently executed regions and map them to DySER. If DySER specializes the most frequently executed regions, then the cost of reconfiguration can be amortized over many invocations of the same configuration. Many conventional techniques can be repurposed for identifying the regions, including programmer inserted pragmas, or using static analysis techniques like hyperblocks [84], superblocks [39] or inner loops. We can also use dynamic approaches using profiling, especially path profiling [5], or pathtrees [47], or loops with high trip counts to identify the candidates to specialize with DySER as they help to select large code regions to specialize. The DySER compiler uses a combination of these approaches to select a region. It first identifies the regions with programmer inserted pragmas to select the code region to specialize. If no pragmas are provided, it tries to use the profiling

information to select the region. If no profiling information is available, the DySER compiler selects inner loop bodies as a candidate for specialization.

4.3.2 AEPDG Construction

Once the region for specialization is identified, the DySER compiler constructs the corresponding PDG using existing techniques [38]. Forming the AEPDG means partitioning the PDG into the access-PDG and execute-PDG. This task is important because it influences the effectiveness of the acceleration. DySER's configurable datapath and control capability give the compiler great flexibility in determining the execute-PDG. Overall, we want to chose regions which are as large as possible, so that they can use DySER's efficient datapath. However, we also want to choose graphs which are tightly connected, because communication with DySER requires instruction overhead. Also, since memory operations still must be executed by the main processor, the potential subgraphs for selection are constrained.

The compiler employs two heuristics to perform the partitioning. In the first approach, it finds the backward slices of all address calculations from loads and stores inside the candidate region, and places them in the access-PDG. The remaining instructions form the execute-PDG. This works well for many data parallel applications, where each loop loads many data elements, performs a computation, and stores the result. Algorithm 1 shows the algorithm that constructs the AEPDG given the program dependence graph for the code region, which uses the backward slice of the address calculations to get access-PDG and execute-PDG.

For applications where the primary computation is to compute the address of a load or a store, the method described above places almost all instructions in the access-PDG. Instead, we employ a method which first identifies loads/stores that are dependent on prior loads/stores. Then, we find the backward slices for the non-dependent loads/stores as we did in our first approach. For the dependent loads/stores, we identify the forward slices and make them also part of the access-PDG, leaving the address calculation of the dependent loads/stores as the execute subregion.

To select between these techniques, we simply choose the one which provides the largest execute-PDG. It is possible to develop more advanced techniques or selection heuristics, but in practice,

Algorithm 1 FORM-INITIAL-AEPDG(pdg)

```
1: worklist ← ∅
2: apdg ← ∅ {Initialize worklist for slicing: Loads, Stores}
3: for all node ∈ pdg such that node ∈ LOADS(pdg) do
4:    worklist ← worklist ∪ node
5: end for
6: for all node ∈ pdg such that node ∈ STORES(pdg) do
7:    apdg ← apdg ∪ node
8:    worklist ← worklist∪ GET-ADDRESS(node)
9: end for{Add branches that are latches to loops to apdg}
10: for all node ∈ pdg such that node ∈ IS-LATCH(pdg) do
11:    apdg ← apdg ∪ node
12: end for
   {Slice PDG}
13: while HAS-ELEMENT(worklist) do
14:    node ← POP(worklist)
15:    apdg ← apdg ∪ node
16:    for all op ∈ GET-OPERANDS(node) such that op ∉ apdg do
17:       worklist ← worklist ∪ op
18:    end for
19: end while
20: epdg ← pdg − apdg
   {Populate IO edges}
21: dyio_edges ← ∅
22: for all node ∈ epdg do
23:    for all op ∈ GET-OPERANDS(node) such that op ∉ epdg do
24:       dyio_edges ← dyio_edges ∪ (op, node, < 0, 0 >)
25:    end for
26:    for all use ∈ GET-USES(node) such that use ∉ epdg do
27:       dyio_edges ← dyio_edges ∪ (node, use, < 0, 0 >)
28:    end for
29: end for
30: aepdg ← (apdg, epdg, dyio_edges)
31: return aepdg
```

these two approaches are sufficient.

4.4 Compilation Strategies to Form Ideal Execute-PDG

To accelerate effectively, the compiler should create execute-PDGs whose number of operations is proportional to the accelerator's resources, and whose interface with the access-PDG has few connections, minimizing the I/O cost. Also, the compiler should schedule the execute-PDG to the

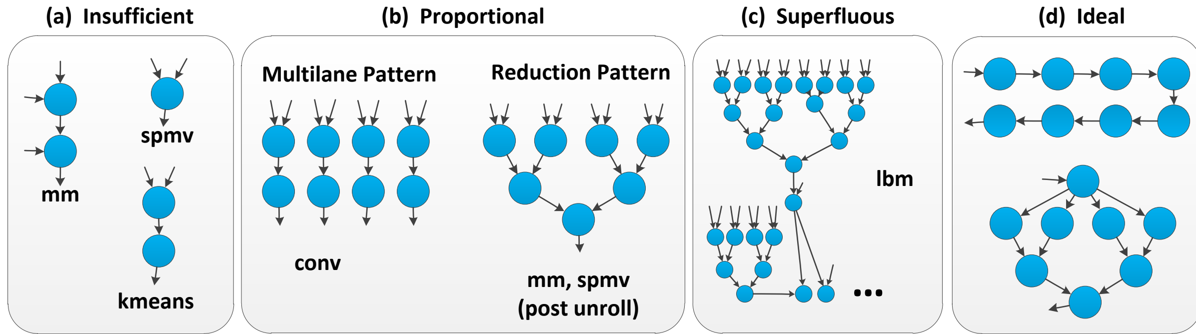


Figure 4.4: Types of computation subregions.

accelerator with high throughput and low latency as the primary concerns. The initial AEPDG will need transformations to achieve the above goals.

The goal of this section is to describe the AEPDG transformations and optimizations that take the inital AEPDG and generate an optimized version, which is more suited for DySER and achieve high efficiency. It first describes the types of code-regions that are in the general purpose workloads and what strategy to use to attain the ideal region to specialize. The next section describes the transformations and optimizations in detail and provides algorithms to accomplish the transformations.

4.4.1 Types of Execute-PDG

The execute-PDG of the programs considered, while providing opportunity for specialization, are mostly ill-suited for DySER due to their size and shape. Figure 4.4 shows four types of computation subregions that the DySER compiler faces during the compilation of general purpose workloads. Section 5.2 describes the benchmarks that we used in the figures in more detail.

Insufficient Regions: Figure 4.4(a) shows example execute-PDGs that are small in relation to DySER's resources. Here, DySER speedup and utilization will be fundamentally limited by the number of operations that can be performed in a single invocation. These small regions limit DySER's potential speedup and specialization, because they do not fully utilize the available resources.

Proportional Regions: Figure 4.4(b) shows execute-PDGs that are appropriately sized for DySER. These generally come in the form of the multilane and reduction patterns. Even though the potential for these regions is high, these patterns have a high communication/computation ratio, which limits speedups, since the instructions in execute-PDG wait for operands to arrive. Another way to view the problem is that the memory subregion simply cannot feed the execute-PDG fast enough to attain high utilization of DySER's resources. In the absence of other mechanisms, DySER performance will be similar to that of an out-of-order processor.

Superfluous Regions: Figure 4.4(c) shows an execute-PDG which is very large. When the execute-PDG is large, only a portion of the execute-PDG can be mapped to DySER and instructions in the remaining portions will be scheduled to execute in the main processor pipeline. This limits the potential benefits from DySER because most of the instructions from the execute-PDG will use power-hungry structures in the processor.

Ideal Regions: Figure 4.4(d) depicts some best-case scenarios of execute-PDG, distinguished by small numbers of inputs and outputs with numerous computations. Assuming pipelinable invocations, these patterns are ideal because they have very little communication overhead. However, these are rare in most workloads. In order to get this type of region, we develop techniques to transform other regions into this type.

4.4.2 Transformation Flow

The transformation flow in the Figure 4.5 shows the DySER compiler's overall strategy for transforming an arbitrary execute-PDG to act like an ideal execute-PDG. In addition to the transformation flow, the figure also includes the hardware mechanisms in DySER that the compiler exploits to achieve high efficiency. The rest of this subsection describes the transformation flow in more detail and the next section presents algorithms for these transformation and details how these transformations can be implemented.

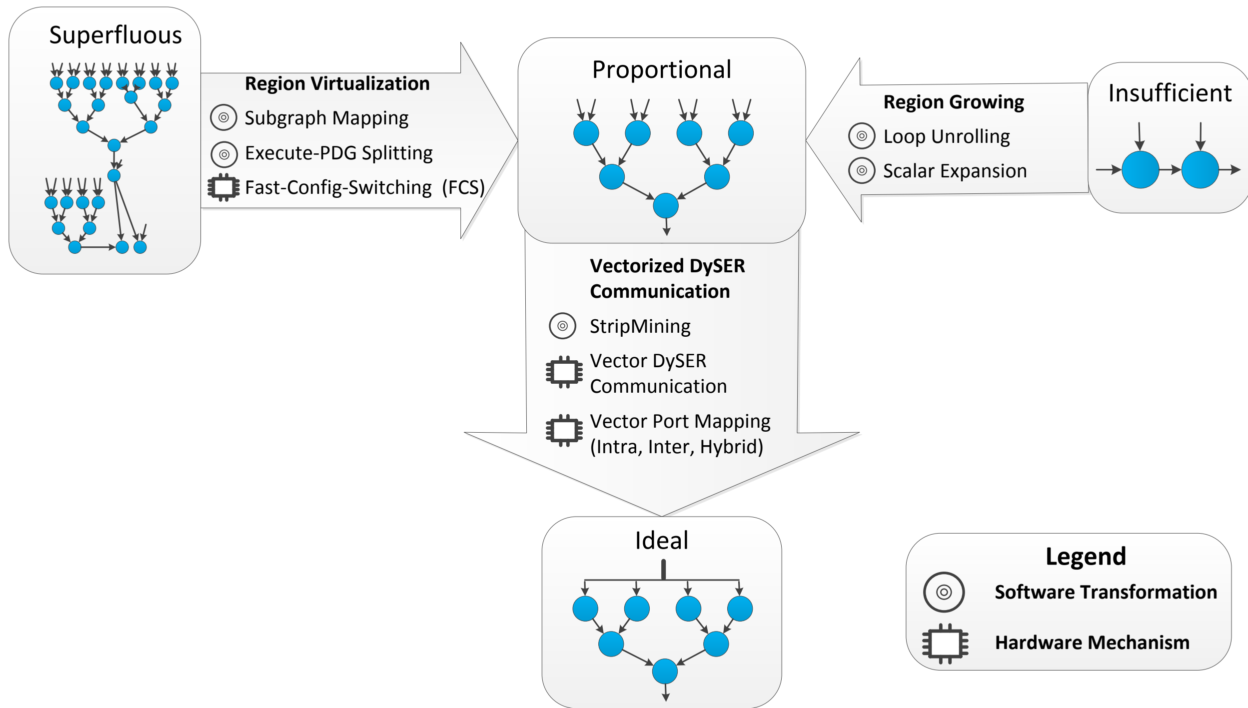


Figure 4.5: Transformation flow and hardware support

4.4.2.1 Region Growing

Execute-PDGs that are too small to attain high utilization must be expanded, which can be achieved by transforming the loops. Specifically, we apply loop unrolling until an appropriately sized computation subregion is formed, as shown in Figure 4.6(a). If the loops are independent, we create a multilane pattern. With a single loop carried dependence, we create a reduction pattern, if possible. When profitable, we alternatively employ scalar expansion 4.6(b), which enables loop parallelization by providing temporary storage for dependent variables. Scalar expansion allows us to break some reduction patterns into multilane patterns, which can be beneficial depending on the use of the region's outputs. Note that we cannot rely on standard compiler passes, as they are unaware of DySER, and could create inappropriate regions.

4.4.2.2 Vectorizing DySER

Vectorized DySER instructions can load and store only contiguous words. In order to vectorize send and load instructions efficiently, we must provide mechanisms to handle arbitrary relationships

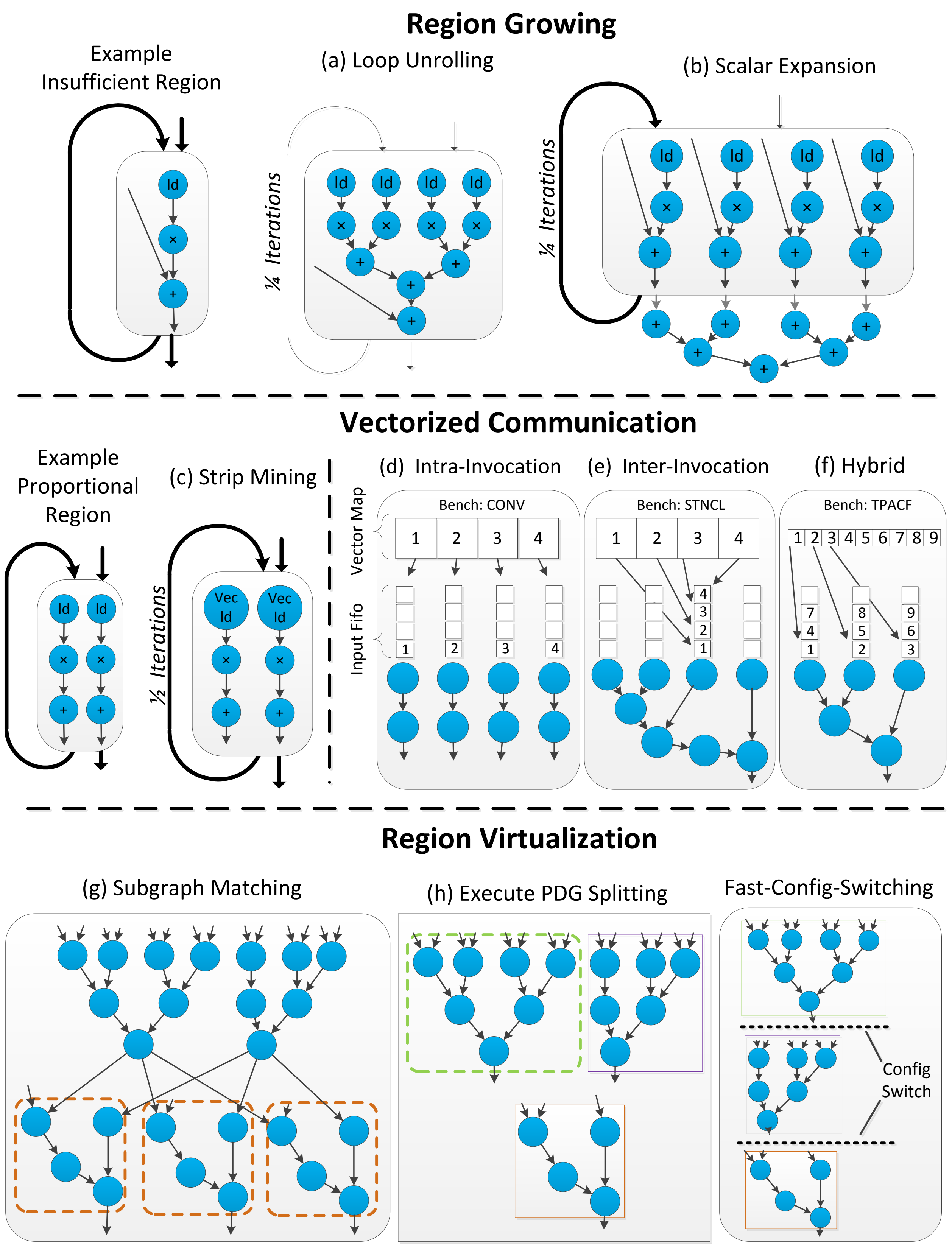


Figure 4.6: AEPDG Transformations

between contiguous memory and the interface to the regions. We explain several communication patterns with examples, and describe the mechanisms which make vectorization possible.

Intra-invocation Communication (“Wide”): Figure 4.6(d) shows the computation subregion, where each contiguous memory word is mapped to a different input port of DySER and used by a single invocation. This type of wide communication pattern converts DySER into a vector unit.

Inter-invocation Communication (“Deep”): Figure 4.6(e) shows the computation subregion, where each contiguous memory word is mapped to the same port since subsequent invocations use contiguous memory addresses, thus allowing multiple invocations to be explicitly pipelined.

Hybrid Communication: Figure 4.6(f) shows a computation subregion from the TPACF benchmark. Neither inter-invocation nor intra-invocation is sufficient to perform a vector load more than 3 words wide. Our strategy is to use a hybrid, where each word triplet is sent to the same invocation, and subsequent triplets are pipelined to subsequent invocations. This example is 3 “wide” and 3 “deep”.

Stripmining: Employing these communication patterns requires a transformation called stripmining, as shown in Figure 4.6(c). Both stripmining and loop unrolling reduce the loop trip count, but doing both is usually possible for data parallel workloads, because the loops in these benchmarks have high bounds.

4.4.2.3 Region Virtualization

Similarly to insufficient regions, overly large regions must be "resized" to fit inside DySER to achieve high utilization. Compared to instruction-level acceleration, DySER’s dynamic customization introduces resource limitation challenges, which we overcome by employing two primary techniques.

Subgraph Matching: First, we attempt to reduce the computational region by identifying similar computational structures, which we call Subgraph Matching as shown in Figure 4.6(g). The trans-

formation is essentially to cut dataflow edges from a common subgraph, and combine all common subgraphs together. These cut edges will be reconnected through the memory subregion.

Execute-PDG Spliting: If Subgraph Matching cannot reduce the computation subregion sufficiently, we employ a further technique that splits the execute-PDG into multiple execute-PDGs that can fit inside the DySER substrate. Figure 4.6(h) shows how a superfluous computation subregion can be cut into components of appropriate size and mapped to DySER by using multiple configurations. However, since the DySER needs to switch between these configurations, the configuration penalty cannot be amortized over multiple invocations. As described in subsection 3.5.3, DySER uses fast configuration switching to mitigate the efficiency lost of reconfiguration.

4.5 Algorithms for AEPDG Transformations

As described in the previous section, to accelerate effectively, the DySER compiler should create execute-PDGs whose number of operations is proportional to the DySER's resources, and whose interface with the access-PDG has few connections, minimizing the I/O cost. We also presented several transformations to get an ideal execute-PDG. In this section, we present algorithms for the transformations described in the previous section.

4.5.1 Region Growing

Loop Unrolling for PDG Cloning: If the execute-PDG underutilizes DySER, the potential performance gains will be suboptimal. To achieve high utilization for a loop which has no loop-carried dependencies, we need to grow the execute-PDG until a sufficiently large execute-PDG is created. This corresponds to unrolling the computations in the access-PDG, and reduces the trip count of the loop.

Algorithm 2 shows the PDG cloning and strip mining algorithm, which creates edges to track the links between the access and execute subgraphs. First, it uses the size and types of instructions in the execute-PDG and the DySER model, which includes the quantity of and capabilities of the functional units in DySER, to determine the number of execute-PDG clones and number of times

the access-PDG should be unrolled. After cloning the execute-PDG and unrolling the access-PDG, it creates edges between appropriate nodes in each. These interface edges are labeled to track the spatio-temporal information.

This transformation exploits the spatial aspect of the AEPDG to track the links among the access subgraph nodes and cloned execute subgraph nodes. In the load/store coalescing transformation, described later, the compiler uses these links to combine consecutive accesses.

Strip Mining for Vector Deepening In addition to parallelizing the loop to achieve the correct execute-PDG size, the compiler can further parallelize the loop by pipelining computations. This transformation, called strip mining, means that additional loop iterations are performed in parallel by pipelining data through the execute-PDG. The effective "depth" of the vectors, if the memory access is vectorizable, is increased as an effect of this transformation.

4.5.2 Region Virtualization

Subgraph Matching: If the size of an execute-PDG is "larger" than the size of DySER, many configurations will be required, resulting in excess overhead per computation performed. If computations in the execute-PDG share a common structure, or formally an isomorphic subgraph, this can be exploited to pipeline the computations through this subgraph. This transformation, called subgraph matching, merges the isomorphic subgraphs, and modifies the access nodes to use temporal information encoded in the AEPDG to pipeline data.

Since computing the largest matching subgraph is an NP-complete problem, we use a greedy algorithm that grows the subgraph by incrementally adding instructions to a seed instruction. Algorithm 3 lists the algorithm that uses the function COMPUTE-EPDG-COVERING to find the subgraphs or the covers. This algorithm then uses the provided covers to insert edges in the AEPDG to accomplish the subgraph matching.

We can also adapt the previously proposed approach by Clark et al. [24] called unate covering selection which does full enumeration of the bounded search space to find the covering. However, most program regions we considered do not have common subgraphs, as they most commonly arise when the code is unrolled or functions are inlined before AEPDG formation.

Algorithm 2 CLONE-EPDG-STRIPMINE-APDG($aepdg, dyser_model, vec_len$)

1: $epdg \leftarrow$GET-EPDG($aepdg$)
2: $apdg \leftarrow$GET-APDG($aepdg$)
3: $MaxEPDGs \leftarrow$ GET-NUM-EPDG-IN-DYSER($epdg, dyser_model$)
4: $NumClones \leftarrow x$, **where** $x \leq MaxEPDGs \wedge x | vec_len$
5: $NumUnroll \leftarrow vec_len$
{Copy execute-PDG $NumClones$ times}
6: **for** $i := 1$ **to** $NumClones$ **do**
7: $clonedEPDG[i] \leftarrow$ CLONE($edpg$)
8: **end for**
{Unroll access-PDG $NumUnroll$ times}
9: **for** $i := 1$ **to** $NumClones$ **do**
10: $unrolledAPDG[i] \leftarrow$ UNROLL($adpg$)
11: **end for**
{Insert I/O edges for execute-PDG inputs}
12: **for all** $node \in$ GET-INPUTS($epdg$) **do**
13: **for** $i := 1$ **to** $NumUnroll$ **do**
14: $idx \leftarrow i \mod NumClones$
15: $clonedNode \leftarrow$ GET-CLONED-NODE($ClonedEDPG[idx], node$)
16: **for all** $pred \in$ GET-OPERANDS($clonedNode$) **do**
17: $unrolledNode \leftarrow$GET-UNROLLEDNODE($UnrolledAPDG[i], pred$)
18: $dyio_edges \leftarrow dyio_edges \cup (unrolledNode, clonedNode, < i, 0 >)$
19: **end for**
20: **end for**
21: **end for**
{Insert I/O edges for execute-PDG outputs}
22: **for all** $node \in$ GET-OUTPUTS($epdg$) **do**
23: **for** $i := 1$ **to** $NumUnroll$ **do**
24: $idx \leftarrow i \mod NumClones$
25: $clonedNode \leftarrow$ GET-CLONED-NODE($ClonedEDPG[idx], node$)
26: **for all** $use \in$ GET-USES($clonedNode$) **do**
27: $unrolledNode \leftarrow$GET-UNROLLEDNODE($UnrolledAPDG[i], use$)
28: $dyio_edges \leftarrow dyio_edges \cup (clonedNode, unrolledNode, < i, 0 >)$
29: **end for**
30: **end for**
31: **end for**
32: $out_apdg \leftarrow \bigcup_{i=1}^{NumUnroll} UnrolledAPDG[i]$
33: $out_epdg \leftarrow \bigcup_{i=1}^{NumClones} ClonedEPDG[i]$
34: $out_aepdg \leftarrow (out_apdg, out_epdg, dyio_edges)$
35: **return** out_aepdg

Algorithm 3 SUBGRAPH-MATCHING-AND-SPLITTING($aepdg, dyser_model$)

1: $epdg \leftarrow$ GET-EPDG($aepdg$)
2: $apdg \leftarrow$ GET-APDG($aepdg$)
3: $dyio_edges \leftarrow$ GET-DYIO-EDGES($aepdg$)
4: $covering \leftarrow$ COMPUTE-EPDG-COVERING($epdg, dyser_model$)
5: **for all** $g \in covering$ **do**
6:     **for all** $node \in g$ **do**
7:         **for all** $op \in$ GET-OPERANDS($node$) **such that** $op \notin g$ **do**
8:             $send \leftarrow$ SPLIT-EDGE-AND-CREATE-SEND($op, node$)
9:             $apdg \leftarrow apdg \cup send$
10:             $dyio_edges \leftarrow dyio_edges \cup (send, node, <1, 0>)$
11:         **end for**
12:         **for all** $use \in$ GET-USES($node$) **such that** $use \notin g$ **do**
13:             $recv \leftarrow$ SPLIT-EDGE-AND-CREATE-RECV($node, use$)
14:             $apdg \leftarrow apdg \cup recv$
15:             $dyio_edges \leftarrow dyio_edges \cup (node, recv, <1, 0>)$
16:         **end for**
17:     **end for**
18:     INSERT-DYSER-CONFIG(g)
19: **end for**
20: **return** $(epdg, epdg, dyio_edges)$

Execute PDG Splitting: When subgraph matching is insufficient to reduce the size of the execute-PDG, or when there is not an isomorphic subgraph, it becomes necessary to split the execute-PDG and insert nodes in the access-PDG to orchestrate the dependencies among the newly created execute-PDGs. Although this introduces extra configuration switches and cannot amortize the cost of reconfiguration, DySER can use the Fast Configuration Switching mechanism, which is described in subsection 3.5.3, to mitigate the efficiency loss.

4.5.3 Vectorized DySER Communication

Unrolling for Loop Dependence: When the AEPDG represents loops without data dependence, we can use it to trivially unroll and vectorize the nodes in the access-PDG just like traditional SIMD compilers. When loops have memory dependencies across iterations, SIMD compilers usually fail or use complex techniques, such as the polyhedral model [49, 37], to transform the loop such that they can be vectorized. In contrast, we simply unroll the loop multiple times and combine the dependent computation with the execute-PDG. This can accelerate the loop considerably since the

execute-PDG is pipelined using DySER. Again, the AEPDG tracks the links between the unrolled nodes in the access-PDG, which can be used to combine the nodes in the load/store coalescing transform.

Traditional Vectorization: The DySER compiler leverages several techniques developed for SIMD compilers to vectorize loops when the iterations are independent [96]. These include loop peeling, scalar expansion, and loop interchange. Loop peeling is used to maintain correctness in the presence of non-divisible loop termination bounds. Scalar expansion, where the reduction variables are split into multiple copies to eliminate unnecessary dependence, improves the amount of available parallelism. Loop interchange, which switches the order of nested loops, can improve memory locality. The DySER compiler implements these traditional vectorization techniques on the AEPDG, and these techniques are designed not to interfere with its temporal and spatial properties.

Load/Store Coalescing: DySER's flexible I/O interface enables the compiler to combine multiple DySER communication instructions which have the same base address, but different offsets. We use the order encoded in the interface edges between access and execute PDGs and leverage existing alias analysis to find whether multiple access nodes can be coalesced into a single node.

Algorithm 4 and algorithm 5 show the load/store coalescing algorithm, which tracks the offset information between the coalesced loads and the computation in the execute-PDG. It iterates through the memory instructions in program order and attempts coalescing with nodes of the same type (i.e both loads or stores) which also access addresses with a constant offset (relative to the loop induction variable). Then, if any of the coalesced nodes are dependent on other memory nodes in the AEPDG, it discards the memory dependent loads from coalescing. Coalesced nodes are split into vector-sized groups, and for each group a new node is created with updated instance and offset information.

4.6 Scheduler and Code Generation

After the DySER compiler optimizes the AEPDG with the transformations described in the last section, it generates the DySER configuration for the execute-PDG and assembly code for the

Algorithm 4 COALESCE-LOADS-STORES($aepdg$)

```
1: apdg ← GET-APDG(aepdg)
2: dyio_edges ← GET-DYIO-EDGES(aepdg)
3: S ← GET-LOADS-STORES-IN-PROGRAM-ORDER(apdg)
4: while HAS-ELEMENT(S) do
5:    candidate ← GET-FIRST-NODE(S)
6:    coalescedNodes ← candidate
7:    for all node ∈ S in order do
8:       if IS-LOAD(candidate) == IS-LOAD(node) and HAS-CONST-OFFSET(Candidate, node) then
9:          coalescedNodes ← coalescedNodes ∪ node
10:      end if
11:   end for
      {Check dependences}
12:   AliasedNodes ← GET-ALIASED-NODES(S, coalescedNodes)
      {Cannot coalesce nodes}
13:   if AliasedNodes ≠ ∅ then
14:      S ← S − (AliasedNodes ∪ coalescedNodes)
15:      continue
16:   end if
17:   apdg, dyio_edges ← COALESCE-NODES-IN-APDG(apdg, dyio_edges, coalescedNodes)
18:   S ← S − coalescedNodes
19: end while
20: return (apdg, GET-EPDG(aepdg), dyio_edges)
```

access-PDG. To generate the DySER configuration, it uses a greedy algorithm to schedule the execute-PDG spatially to the DySER hardware substrate. Before it generates the assembly code for the access-PDG, it inserts DySER instructions to communicate between the main processor and DySER through the DySER's ports.

4.6.1 Scheduling Execute PDGs

Once the final execute PDG has been determined, we need to create a mapping between the execute-PDG and the DySER hardware itself. We use a greedy algorithm that places instructions in DySER with the lowest additional routing cost. This greedy algorithm is similar to other spatial architecture scheduling algorithms, completes quickly, and is suitable for use in production compilers. This algorithm usually succeeds in scheduling all nodes in the execute-PDG to the DySER hardware. When it fails, we "spill" problematic nodes to the access-PDG and schedule the execute-PDG again. Algorithm 6 lists the algorithm for scheduling execute-PDG to DySER.

Algorithm 5 COALESCE-NODES($apdg, dyio_edges, coalescedNodes$)

```
1: cn ← SORT-BY-OFFSET(coalescedNodes)
2: vecNodes ← GET-SPLITTED-LIST(cn, maxVecSize)
3: if IS-LOAD(vecNodes) then
4:    for all vecLoad ∈ vecNodes do
5:       apdg ← apdg ∪ vecLoad
6:       for all load ∈ vecLoad do
7:          offset ← GET-OFFSET-FROM(vecLoad)
8:          for all edge ∈ GET-DYIO-EDGES(aepdg, load) do
9:             dyio_edges ← dyio_edges ∪ (vecLoad, edge.dest, < edge.instance, offset >)
10:         end for
11:      end for
12:   end for
13: else
14:   for all vecStore ∈ vecNodes do
15:      if IS-CONTIGUOUS(vecStore) then
16:         apdg ← apdg ∪ vecStore
17:         for all store ∈ vecStore do
18:            offset ← GET-OFFSET-FROM(vecStore)
19:            for all edge ∈ GET-DYIO-EDGES(aepdg, store) do
20:               dyio_edges ← dyio_edges ∪ (edge.dest, vecStore, < edge.instance, offset >)
21:            end for
22:         end for
23:      end if
24:   end for
25: end if
26: return apdg, dyio_edges
```

Another potential approach is to use the recently proposed general constraint centric scheduler to map the execute-PDG to the DySER hardware [93, 109].

4.6.2 Code Generation

In this final phase of compilation, the compiler generates the accelerator configurations corresponding to the scheduled execute-PDG. It also inserts instructions in the access-PDG to transfer data to and from the accelerator. The compiler uses the information in the interface edges of the AEPDG to generate vectorized DySER instructions. Also, it eliminates any dead code introduced by mapping the execute-PDG to DySER.

Algorithm 6 SCHEDULE-EXECUTE-PDG($epdg, dyser$)

1: $Config \leftarrow \emptyset$
2: $Spilled \leftarrow \emptyset$
3: $SortedList \leftarrow$ TOPOLOGICAL-SORT($epdg$)
4: **for all** $node \in SortedList$ **do**
5: $BestCost \leftarrow \infty$
6: $BestRoute \leftarrow \emptyset$
7: $Slots \leftarrow$ GET-AVAILABLE-SLOTS($dyser, node$)
8: **for all** $slot \in Slots$ **do**
9: $Route, Cost \leftarrow$ CREATE-ROUTES-NODE-FROM-OPS($node, slot, config$)
10: **if** $Cost < BestCost$ **then**
11: $BestCost \leftarrow Cost$
12: $BestRoute \leftarrow Route$
13: MAKE-SLOT-AS-OCCUPIED($dyser, slot$)
14: **end if**
15: **end for**
16: **if** $BESTCOST \neq \infty$ **then**
17: $Config \leftarrow$ ADD-ROUTE-TO-CONFIG($Config, Route$)
18: **else**
19: $Spilled \leftarrow node$
20: **end if**
21: **end for**
22: **return** $(config, spilled)$

4.7 Implementation

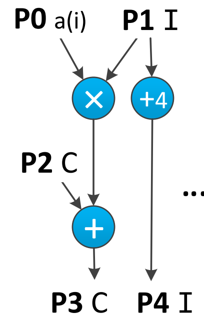
To implement our compiler, we leverage the LLVM compiler framework and its intermediate representation (LLVM IR). First, we implement an architecture independent compiler pass that processes LLVM IR and constructs the AEPDG. Second, we develop a series of optimization passes that transform the AEPDG to attain high quality code for DySER. Third, we implement a transformation pass that creates LLVM IR with DySER instructions from the access-PDG. Finally, we extend the LLVM X86 code-generator to generate DySER configuration bits from the execute-PDG. With this compiler, we can generate executables that target DySER from C/C++ source code. Our implementation is publicly released and more documentation is available here [30].

```
for (i=0; i<n; ++i) {
    c += a[i] * i;
}
```

(a) Original Loop

```
I={0,1,2,3}, C={0,0,0,0};
for(i=0; i<n-n%4; i+=4){
  dyserload_vec a[i:i+3] => P0;
  dysersend_vec I => P1;
  dysersend_vec C => P2;
  dyserrecv_vec P3 => C;
  dyserrecv_vec P4 => I;
}
...
c = C[0]+C[1]+C[2]+C[3];
...
```

(b) DySER Code



(c) Execute-PDG

Figure 4.7: Loop with Reduction/Induction (Peeled Loop not shown)

4.8 Case Study

In this section, we illustrate how the DySER compiler with the AEPDG representation and previously described compiler transformations enables the automatic specialization of DySER. We demonstrate the ability of the compiler by doing a case study on a set of challenging loops with characteristics that are usually hard to specialize with a coarse grain reconfigurable architecture.

4.8.1 Reduction/Induction

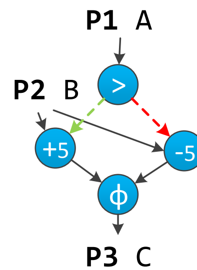
Loops which have contiguous memory accesses across iterations and lack control-flow or loop dependencies are easily specializable with vector instructions. Figure 4.7a shows an example reduction loop with an induction variable use. This creates artificial dependencies that may prevent the DySER compiler from specializing the region with DySER vector instructions. However, the DySER compiler vectorizes the variable `c` by using scalar expansion to vectorize the induction

```
for (i=0; i<n; ++i) {
  if (a[i] > 0) {
    c[i] = b[i] + 5;
  } else {
    c[i] = b[i] - 5;
  }
}
```

(a) Original Loop

```
for(i=0; i<n-n%4; i+=4){
  dyserload_vec a[i:i+3]->P1;
  dyserload_vec b[i:i+3]->P2;
  dyserstore_vec P3->c[i:i+3];
}
...
```

(b) DySER Code



(c) Execute-PDG

Figure 4.8: Loop with control dependence (Peeled Loop not shown)

variable by hoisting initialization out of the loop, and performing loop peeling to do vector-size loop iterations. Figure 4.7b shows the transformed code after DySER acceleration, and Figure 4.7c shows the uncloned version of the execute-PDG.

4.8.2 Control Dependence

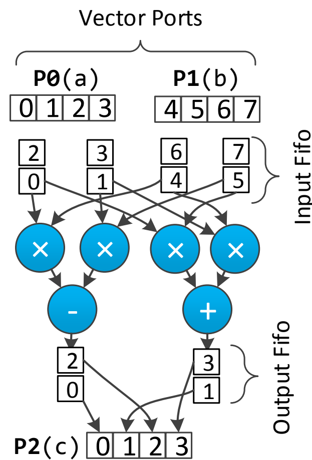
The DySER compiler leverages the AEPDG structure to represent control flow inside the execute-PDG. The example in Figure 4.8(a) shows how the DySER compiler can trivially observe that the control is entirely in the execute-PDG, enabling this control decision to be offloaded from the main processor. This eliminates the need for any masking instructions or conditional moves in the main processor, reducing overhead significantly. Since the control dependence is moved to the execute PDG, the analysis for the access PDG easily finds out the feasibility of doing vectorization on the memory access and vectorizes the code. This is an additional benefit of the AEPDG. Formerly diffcult cases for vectorization became easier to perform because of the decoupled nature of AEPDG [48].

```
for  (i=0; i<n; ++i) {
 c[2i] = a[2i]*b[2i]
     - a[2i+1]*b[2i+1];
 c[2i+1]=a[2i]*b[2i+1]
     + a[2i+1]*b[2i];
}
```

(a) Original Loop

```
for(i=0; i<n-n%4; i+=4){
  DyLd_Vec a[2i:2i+3] => P0;
  DyLd_Vec a[2i+4:2i+7] => P0;
  DyLd_Vec b[2i:2i+3] => P1;
  DyLd_Vec b[2i+4:2i+7] => P1;
  DySt_Vec P2 => c[2i:2i+3];
  DySt_Vec P2 => c[2i+4:2i+7];
}
...
```

(b) DySER Code



(c) Execute-PDG

Figure 4.9: Loop with strided data access (Peeled Loop not shown)

4.8.3 Strided Data Accesses

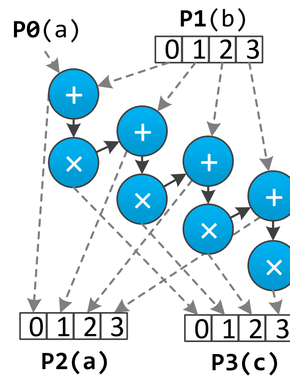
Strided data accesses can occur for a variety of reasons, commonly for accessing arrays of structs (AOS). Vectorizing compilers can sometimes eliminate the strided accesses by transforming the data structure into a struct of arrays (SOA). However, this transformation requires global information about the data structure usage, and is not always possible. Figure 4.9 shows an example loop that does complex multiplication, which cannot benefit from AOS to SOA transformation because of the way the data is consumed. When non-contiguous memory prevents straight-forward loop vectorization, the DySER compiler can leverage the spatio-temporal information in the AEPDG to configure DySER's flexible I/O hardware to perform this mapping. For the code in Figure 4.9(b), the compiler creates interleaved wide ports to coordinate the strided data movement across loop iterations, as shown in Figure 4.9(c). Since the DySER port configuration is used throughout the

```
for(i=1; i<n; ++i) {
  c[i] = a[i-1]+b[i];
  a[i] = c[i]*k;
}
```

(a) Original Loop

```
for(i=1; i<=n-n%4; i+=4){
  dyserload_vec a[i-1] => P0;
  dyserload_vec b[i:i+3] => P1;
  dyserstore_vec P2 => a[i:i+3];
  dyserstore_vec P3 => c[i:i+3];
}
...
```

(b) DySER Code



(c) Execute-PDG

Figure 4.10: Loop with loop-carried dependence (Peeled Loop not shown)

loop's lifetime, this is more efficient than issuing shuffle instructions on each loop iteration.

4.8.4 Carried Dependencies

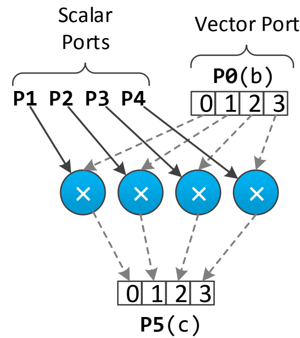
Usually, optimizing compilers attempt to break loop-carried memory dependencies by reordering loops after loop fission, or reordering memory operations inside a loop, or traversing the loop in a different order. However, these compilers use heavy weight analysis framework like polyhedral analysis to find out whether the transformations are legal or not. Figure 4.10 shows a loop with a carried dependence which cannot be broken with traditional techniques. The statements cannot be re-ordered or separated because of the forward flow dependence through `c[i]`, and the backwards loop anti-dependence on `a[i]`. However, the DySER compiler naturally handles the loop carried dependencies and map them to the DySER configurable datapath. The DySER compiler unrolls the loop until the execute-PDG uses a proportional number of resources to the hardware. This exposes the contiguous memory accesses, which the load/store coalescing algorithm take advantage of

```
for(i=0; i<n; i++){
  d1 = a[i];
  index = ind[i];
  d2 = b[index];
  c[i] = d1*d2;
}
```

(a) Original Loop

```
for(int i=0; i<n-n%4; i+=4) {
  DyLd_Vec a[i:i+3] => P0;
  DyLd b[ind[i+0]] => P1;
  DyLd b[ind[i+1]] => P2;
  DyLd b[ind[i+2]] => P3;
  DyLd b[ind[i+3]] => P4;
  DySt_Vec P5 => c[i:i+3];
}
...
```

(b) DySER Code



(c) Execute-PDG

Figure 4.11: Loop with Partial Vectorization (Peeled Loop not shown)

and makes use of DySER vector instructions for efficiency. The loop dependencies, which are now explicit in the execute-PDG, become part of DySER's internal datapath, enabling efficient execution of the loop body.

The drawback to this approach is that the dependencies can create long latency "chains" in the execute-PDG. However, in many cases the computation is not on the critical path, or can be amortized by pipelining multiple invocations through DySER.

4.8.5 Partially Vectorizable

When contiguous memory patterns occur only on some streams in a loop, an optimizing compiler must carefully weigh the benefits of using vector instructions against the drawbacks of excessive shuffling. One example is in Figure 4.11, where the loop has two streaming access patterns coming from the arrays "a" and "b". The accesses from "a" are contiguous, but "b" is accessed indirectly through the "index" array. Here, the compiler can chose to perform scalar loads for non-contiguous

accesses and combine these values using additional instructions. This transformation's profitability relies on the number of instructions required to construct a vector with "d2" values.

Though partially vectorizable loops pose complex tradeoffs for vectorizing compilers, the DySER compiler represents these naturally with the AEPDG, which is made possible by the flexible I/O interface that the DySER hardware provides. For the loop in Figure 4.11(a), accesses to the "a" array are vectorized, and scalar loads are used for "b". Compared to a vectorized version, the DySER compiler eliminates the overhead of additional shuffle instructions without any need for complex analysis or tradeoffs.

4.9 Chapter Summary

In this chapter, we described the DySER compiler design that uses a novel compiler intermediate representation called the Access/Execute Program Dependence Graph (AEPDG) to generate optimized code for DySER. We have designed and implemented the transformations described in this chapter on a LLVM based compiler. We publicly released the source code of this compiler and more details are here [30]. Although AEPDG enables the compiler to generate optimized code for DySER or DySER like architectures, it can also help generate code for SIMD architectures since it decouples the data access from the computation itself. However, it is an open question whether it also makes code generation for SIMD architecture easier and better than existing approaches.

5 Experimental Evaluation and Analysis

An important goal of this thesis is to demonstrate that DySER and its compiler can automatically specialize a set of diverse workloads that are written in the traditional programming model with high level languages like C/C++. To that end, the previously described compiler design has been implemented using the LLVM compilation framework to compile C/C++ programs and generate executables optimized for DySER. To evaluate the DySER architecture, we have implemented a simulator for DySER integration with an out-of-order processor. With this implementation of the DySER compiler and the simulator, we performed experiments on a variety of benchmarks to evaluate the effectiveness of DySER and its codesigned compiler by performing simulations representing the integration of DySER with a general purpose core.

5.1 Overview

In this section, we describe the motivations for the experiments conducted in this chapter. The quantitative evaluation presented in this chapter centers around five studies.

1. **Workload Characterization:** DySER's efficiency depends upon the DySER compiler's ability to identify code-regions that can be specialized with DySER. With this study, we strive to answer the following questions:

   - Do the applications have specializable regions that are identifiable by the compiler?
   - Can DySER amortize the configuration cost?

   To answer the first question, we use the DySER compiler to identify code regions and then we measure the average number of instructions in the execute-PDG using a static analysis pass.

To answer the second question, we must consider the overhead of the compiler generated configuration bits that DySER uses to specialize the region. If the configuration is not reused multiple times, the cost of reconfiguring will lower overall efficiency. To illustrate that DySER can amortize configuration cost, we quantitatively measure the number of times a configuration is reused during the execution. Section 5.3 describes these characterization studies in more detail.

2. **Compiler Evaluation:** This study evaluates the AEPDG-based compiler implementation for DySER. We compare the performance of the compiler generated code to the manually optimized code. We also compare the DySER compiler's ability to do vectorization to the auto vectorizer in ICC. Section 5.4 elaborates on the evaluation of the compiler and presents the results.

3. **Performance and Energy Evaluation:** This study quantitatively evaluates the speedup attained and energy reduction with DySER over a 4 wide issue out-of-order general purpose processor. The efficiency of DySER comes from its ability to dynamically create a specialized datapath for a sequence of computations and its tight integration to the processor pipeline. DySER's main source of improvements is its ability to execute concurrently with the processor pipeline. To understand the source of improvements, we measured the average number of functional units that are activated per cycle inside DySER and instructions per cycle committed in the main processor, which shows that DySER integration effectively emulates a wider issue window. There are potential bottlenecks for achieving high performance with DySER. If the latency of computation inside DySER is large, it may not perform better. Similarly, if it requires too many DySER instructions to communicate, the energy efficiency from DySER is reduced substantially because the main processor pipeline executes these overhead instructions. Section 5.5 presents this evaluation and the analysis.

4. **Sensitivity Study:** The efficiency attained with DySER varies with the main processor that it integrates with, because wider processors can send operands to DySER faster and utilize DySER better; however they consume more power. To understand the performance sensitivity

of DySER to the main processor, we vary the issue width of the main processor and compare the performance gain with DySER. Section 5.6 describes this sensitivity study.

5. **Database Kernel Evaluation:** For the previous studies, we use benchmarks that can be broadly classified as data parallel and control intensive. To understand the opportunities available in a real world scenario where both data parallel and control intensive code regions exist in the same program, we evaluate DySER on a database query processing engine processing a decision support query. We evaluate DySER on database primitives or operators and on a full query. Section 5.7 describes the database kernel and presents the result of the study.

5.2 Evaluation Methodology

In this section, we describe the experimental framework, the parameters used to configure the hardware simulator and the benchmarks used in the evaluation of DySER and its compiler.

5.2.1 Simulation Environment

The data presented in this chapter for DySER was collected with an X86 architecture simulator derived from gem5 [12]. gem5 is a cycle level architecture simulator, which we configured to simulate the microarchitecture of an out-of-order processor. This simulator can execute the X86 instruction set including SSE, SSE2 and SSE3 instructions. For energy analysis, we collect various activity count and statistics about the execution. The activity counts are then inputted to McPAT [80], which estimates the energy consumption of the processor during the execution.

This simulator is able to simulate out-of-order execution in timing mode, in which it tracks the cycles of the instruction execution along with the memory system. Each X86 instruction is cracked into a sequence of microops, which are instructions from gem5's internal RISC ISA. The simulator then simulates these microops through the processor pipeline. It simulates the fetch, decode, rename, dispatch, issue, execute, writeback and commit pipeline stages. A rename table is used to map the architected registers to the physical registers, which removes output dependencies and anti-dependencies. When branches are mispredicted, the younger instructions in the pipeline

are squashed and restarted. The simulator also simulates the load/store queue and the memory dependency predictor. When the memory ordering is mispredicted, it uses a pipeline flush as the recovery mechanism.

We have implemented a simulator for DySER and integrated it with the detailed out-of-order model in gem5. As expected, the integration of DySER with the pipeline required only modest changes to the simulator. It mainly required changes in the decoder and issue logic. The X86 decoder in gem5 is modified to decode the DySER instructions described in section 3.1. If the specified input port of DySER is full, the processor cannot issue `dysersend`. Similarly, if the output port of DySER is empty, the processor cannot issue `dyserrecv`. The issue logic is modified to check the status of DySER's ports before issuing the DySER instructions. A major engineering effort was needed to perform squashing and restarting DySER invocations when `dysersend` and `dyserrecv` instructions are squashed in the main processor pipeline due to the misprediction of branches or load/store ordering.

This simulator executes all instructions of the benchmarks except for system calls, which are emulated using the host machine's system calls.

5.2.2 Compiler Implementation

We have implemented a C/C++ compiler for DySER using the LLVM compilation framework [76]. This compiler takes the benchmark source files that are written in C/C++ as inputs and produces ELF executables.

It first compiles C/C++ source code into an LLVM bitcode file (.bc file) using a LLVM frontend called `dragonegg`. This frontend is actually a plugin to the GCC's 4.7 C/C++ frontend and uses GCC to compile and optimize the C/C++ source code. `dragonegg` hijacks the usual machinery in GCC, which lowers its internal intermediate representation called gimple into assembly code. Instead `dragonegg` enables GCC to generate LLVM-IR. Using the LLVM's middle-end optimizer tool `opt`, which operates on LLVM-IR, programs are further optimized. We implemented a static analysis to characterize the benchmarks using this optimized LLVM-IR. This static analysis also identifies the regions to specialize and slices the programs statically into the memory subregion

and compute subregion.

In the second stage of the compilation, the DySER compiler takes the output of the static analysis and generates the Access/Execute Program Dependence Graph for the regions identified. The transformation passes, as described in the previous chapter, optimize the AEPDG to attain a good candidate execute-PDG to specialize. The DySER code generator then takes the optimized AEPDG and schedules the execute-PDG to DySER and the access-PDG and other instructions to the main processor pipeline. An existing LLVM tool called `llc` then lowers the LLVM-IR and DySER's configuration bits into machine assembly code.

Finally, we modified the GNU assembler `gas` so that it can assemble DySER instructions and DySER configuration bits into object code along with other processor instructions. After the assembly code is assembled into object code, the system linker links the object file and generates the final executable.

5.2.3 Baseline Machine

We considered an out-of-order processor as our baseline to compare against the DySER architecture. Table 5.1 presents the machine configuration of the baseline processor. The baseline processor is configured to match currently available out-of-order processors.

5.2.4 Benchmarks

Because DySER specializes computation, the most meaningful workloads for this study should have sufficient computation and be representative of emerging areas. To avoid selection bias, we pick two existing benchmark suites: Intel throughput kernels [111] and Parboil [99]. Both of these suites have benchmarks with large data parallelism available to exploit. We choose these benchmark suites because while they are complex and challenging, they are small enough to perform detailed simulation studies to extract out bottlenecks and get insights. Table 5.2 list the description and qualitative characterization of the main kernel in these benchmarks.

However, in order to demonstrate the effectiveness of DySER on general purpose workloads, we choose SPECINT 2006 [60], a suite targeted at conventional single threaded workloads. The

Attribute	Values
Decode/Issue/Commit Width	4/4/4
Branch Predictor	Tournament predictor with 4K BTB
Instruction Queue	64 entries with speculative scheduling. Uses squashing to recover
Reorder Buffer	192
Physical Register File	256 Integer, 256 FP
Load/Store Queue	32/32
Functional Units (Latency)	6 INT ALU(1) 1 INT MULT(3) 1 INT DIV(20) 2 L1D load(2) 1 L1D store(1) 4 FP ADD/CMP(2) 1 FP MULT(4) 1 FP DIV(12) 1 FP SQRT(24)
L1 Caches	I-Cache: 32 KB, 2 way, 64B lines D-Cache: 64 KB, 2 way, 64B lines
L2 Caches	2 MB, 8-way unified, 64B lines
Memory	512MB

Table 5.1: Baseline Machine Configuration

SPEC CPU 2006 integer benchmarks suite provides a collection of standardized programs and inputs intended to represent commonly used applications and programming constructs. We also evaluate a subset of benchmarks from the PARSEC benchmark suite, which represents emerging multi-threaded workloads. Table 5.3 shows the description of these benchmarks.

5.2.5 DySER Microarchitecture Details

The data parallel workloads we considered are mainly floating point workloads and the general purpose workloads we considered are mainly integer workloads. In order to effectively specialize these workloads, we consider two variants of DySER. For floating point workloads, we consider a 64-tile heterogeneous (16 INT-ADD, 16 FP-ADD, 12 INT-MUL, 12-FP-MUL, 4 FP-DIV, 4 FP-SQRT) functional-unit DySER array. It takes 64 cycles to reconfigure DySER with 64 functional units, assuming that the L1I cache contains the configuration bits for DySER and can sustain a bandwidth of 128 bits/cycle. Area analysis comparing to SSE and AVX shows this configuration has similar

Benchmark	Application	Characterization of main kernel
Throughput Kernels		
conv	2D Image convolution	Regular computation and data access
merge	Merge phase of bitonic sorting	Small kernel with unpredictable data dependent control-flow
nbody	Nbody Simulation	Large kernel with regular access pattern
radar	Complex 1D convolution	Small kernel with regular access pattern
treesearch	Tree Search	Irregular data accesses prevents vectorization
vr	Volume rendering	Nested loop with lots of control-flow
PARBOIL Benchmarks		
cutcp	3D Grid & Point Calc.	Small kernel with control flow
fft	Fast Fourier Transform	Regular Memory Access with varying vector width
kmeans	K-Means clustering	Regular memory access
lbm	Fluid Dynamics	Extremely large computation region with control-flow
mm	Dense Matrix Mult.	Small kernel - multiple blocking opportunity
mri-q	Mag. Res. Imaging	Regular memory access, but uses of Sin/Cos function calls
needle	Dynamic Programming	loop carried dependence
nnw	Neural Networks	Indirect and Strided memory access
stencil	3D Matrix Jacobi	Small comp/mem ratio
spmv	Spare Matrix Vector Mult.	Indirect memory access
sad	Sum-of-abs. diff.	Extremely High Comp/Mem Ratio. Good Memory Locality
tpacf	Angular Correlation	Irregular memory access due to histograming

Table 5.2: Data Parallel Benchmark Characterization

area to an AVX unit and twice the area of a SSE unit. For integer workloads, we consider a 64-tile heterogeneous integer functional unit DySER array with 40 INT-ADD, 8 INT-CMP, 8 SHIFT, and 8 INT-MUL.

5.3 Workload Characterization

As described in the previous chapters, the DySER compiler needs to find the regions where DySER can achieve efficiency with specialization. Also, the regions identified should exhibit phase reuse behavior. Otherwise, DySER cannot amortize the cost of reconfiguration and will not be efficient compared to a general purpose processor. To understand the compiler's ability to choose these regions, we evaluate the compiler's ability to identify code-regions that are suitable for DySER with

Benchmark	Appliction Area	Description
	SPECINT 2006	
astar	Path-finding Algorithms	Path finding library for 2D maps, including the well known A* algorithm
bzip2	Compression	bzip2 version 1.0.3, modified to do most work in memory
gcc	C compiler	based on gcc version 3.2
gobmk	Artificial Intelligence: go	Plays a game of go
h264ref	Video compression	encodes a video stream
hmmer	Search gene sequence	Protein sequence analysis using profile hidden Markov models
libquantum	Physics/Quantum Computing	Simulates a quantum computer, running Shor's polynomial-time factorization algorithm.
mcf	Combinatorial Optimization	Vehicle scheduling
omnetpp	Discrete event simulation	Uses the OMNet++ discrete event simulator to model a large Ethernet campus network.
perlbench	Programming Language	Derived from Perl V5.8.7
sjeng	Artificial Intelligence: chess	A highly-ranked chess program
xalancbmk	XML Processing	A modified version of Xalan-C++
	PARSEC	
blackscholes	Scientific	Option pricing with Black-Scholes Partial Differential Equation
fluidanimate	Fluid dynamics	Fluid Dynamics for animation purposes
freqmine	Data mining	Frequent itemset mining
swaptions	Financial	Pricing of a portfolio of swaptions
streamcluser	Kernel	Online clustering of an input stream

Table 5.3: Descriptions of general purpose workloads

the following two questions.

1. How large are the code regions that the compiler is able to select for specialization?

2. Can DySER amortize the cost of reconfiguration?

5.3.1 Execute-PDG Region Size

To create specialized code for DySER, the DySER compiler is heavily dependent on certain application characteristics and must be able to identify specializable regions. In particular, the size of the execute-PDG directly affects the efficiency we can get from DySER. Generally, proportional code

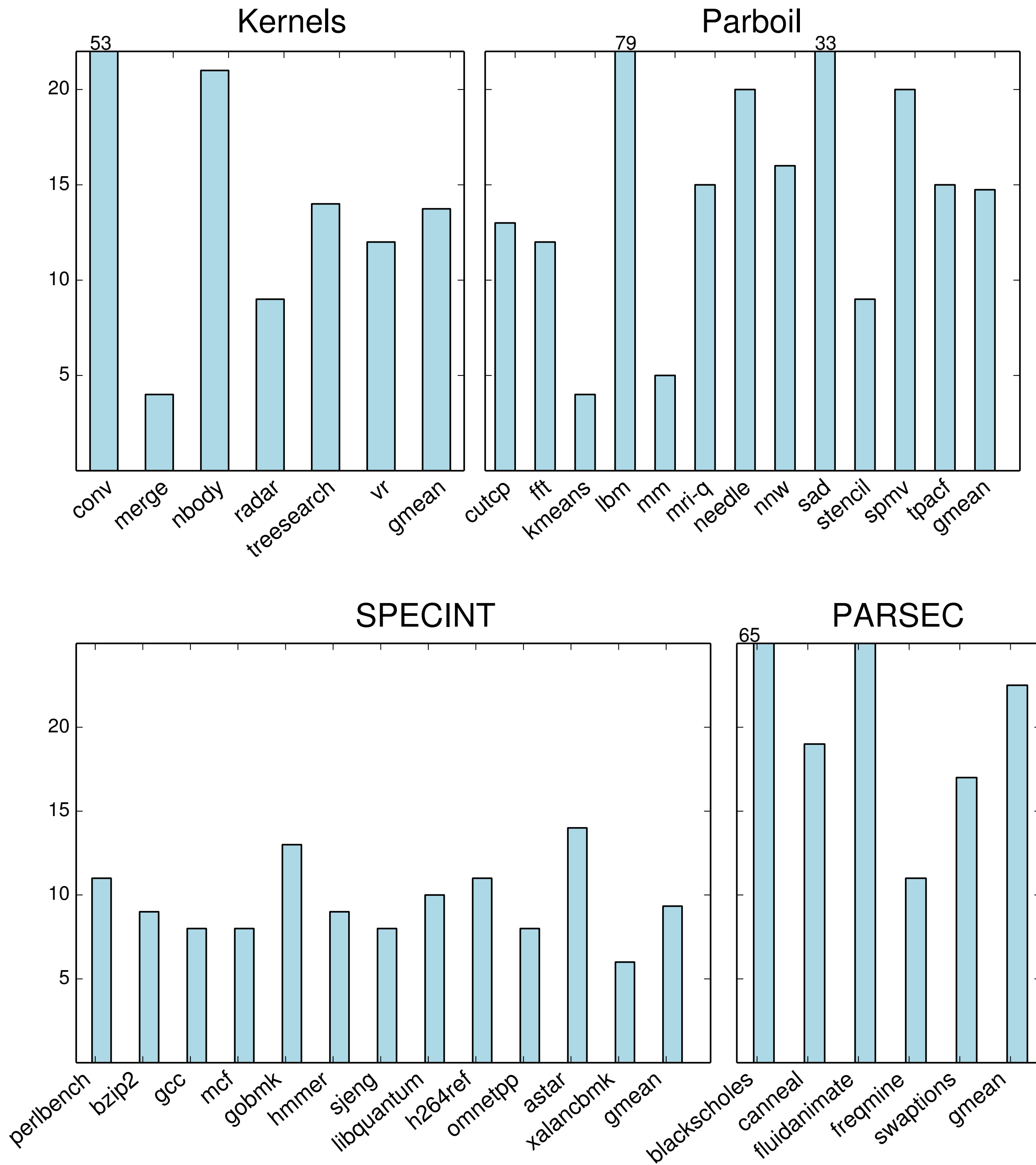


Figure 5.1: Mean number of instructions in execute-PDG

regions that can match DySER's resources are desirable. Smaller code regions tend to underutilize DySER, and larger code regions require extra reconfiguration and may lead to efficiency loss.

In order to measure the code regions, we implemented a static analysis pass in the LLVM compilation framework that measures the number of instructions in acyclic code regions which may be targeted for specialization with DySER. Figure 5.1 shows the mean number of instructions in the acyclic code regions from the respective benchmarks. If the execute-PDG is smaller than 4 instructions and the compiler cannot unroll the loop or clone the execute-PDG, it does not specialize the code region as it is not beneficial. Thus, we omit them from this static analysis as well. First, we observe that the average size of code-regions is about 20 instructions for data parallel workloads and about 10 for SPECINT. For PARSEC benchmarks, the average size of the region is 23 instructions, as they are mostly compute intensive benchmarks. For some data parallel workloads (eg. `merge`, `radar`, `kmeans`, `mm`, `stencil`), the compiler identifies code regions with less than 10 instructions. However, using loop unrolling for the region cloning transformation, the region is cloned to make it larger, and we then use DySER vector instructions to reduce the number of instructions in the processor pipeline.

Observation 5.1. *Programs have code-regions with enough instructions to specialize with DySER. The DySER compiler can identify these specializable regions and target these regions for specialization with DySER.*

5.3.2 Phase Behavior

In order to effectively specialize general purpose workloads, DySER dynamically creates hardware datapaths to match the executing code regions. However, this leads to extra overheads during runtime as the configuration bits need to be fetched from memory and then used to configure DySER by selecting physical routes through the circuit switched network and selecting which function to perform for functional units. In order to amortize the configuration cost, DySER exploits the phase behavior in programs. i.e. the programs reuse a DySER configuration multiple times before reconfiguring DySER for another region. To evaluate the phase behavior and the DySER compiler's ability to amortize the cost of reconfiguration by identifying code regions that are suitable for

DySER, we measure the number of times the program configures DySER and the number of times the program reuses the configuration. Table 5.4 lists the average number of times a configuration is reused for the benchmarks considered.

Since throughput kernels and the PARBOIL benchmarks are mainly designed to study and evaluate a kernel, most benchmarks configure DySER once for the kernel and reuse it until they finish executing. However, `nbody` and `radar` in the throughput kernels have the reduction pattern and the compiler uses an extra configuration to reduce the temporary variables that the scalar replacement optimizations uses to accumulate the values. This makes the number of times a configuration is reused lower in those benchmarks. Similarly, in the PARBOIL benchmark suite, `kmeans` and `sad` use the reduction pattern, thus lowering the average number of times the configuration is reused in these kernels.

For SPEC, the number of times a configuration is used is lower than that of the other benchmark suites. On average, it reuses the configuration about 60 times before switching to a different configuration. For benchmarks like SPEC, it is important to profile the application and specialize only the regions that are the most frequently executing regions to DySER, so less repetitive regions do not introduce extra overheads. Similar to data parallel workloads, the benchmarks from the PARSEC benchmark suite have large configuration reuse except for `freqmine`.

Observation 5.2. *Most benchmarks reuse each DySER configuration multiple times before switching to another configuration. For data parallel workloads, DySER can amortize the cost of reconfiguration. For SPECINT type benchmarks, it is important to choose the most frequently executing code regions as a candidate for specialization with DySER so that it can avoid reconfiguration.*

5.4 Compiler Evaluation

This section quantitatively evaluates the AEPDG-based compiler implementation for DySER and is organized around two main questions:

1. How close to the performance of manually-optimized code does our automatically-compiled code reach?

Benchmarks	#Config. Reuse	Benchmarks	#Config. Reuse
Througput Kernels		**SPECINT 2006**	
conv	634,884	perlbench	12
merge	131,071	bzip2	28
nbody	1,049	gcc	8
radar	8,170	mcf	44
treesearch	61,439	gobmk	8
vr	158,855	hmmer	2,230
		sjeng	3
Parboil		libquantum	3,329
cutcp	11,726,169	h264ref	29
fft	786,559	omnetpp	1
kmeans	99	astar	423
lbm	6,756	xalancbmk	31
mm	179,644,850		
mri-q	4,259,871	**PARSEC**	
needle	1,349,265	blackscholes	66,178
nnw	642,195	fluidanimate	343
sad	3,808	freqmine	4
stencil	86,315	swaptions	509
spmv	8,194,295	streamcluster	357,595
tpacf	429,108		

Table 5.4: Average number of times configuration is reused

2. How does the DySER compiler auto-vectorization and specialization compare to the acceleration provided by the auto vectorizer in Intel's ICC compiler generated code for SSE/AVX extensions?

5.4.1 Evaluation methodology

Compilers: As described above, we have implemented the DySER compiler in LLVM v3.2 and use it generate an executable optimized for DySER. We compare against SSE/AVX code generated with Intel's compiler ICC 12.1, which has a state of art auto-vectorizer. We use ICC compiler flags `-fast -xSSE4.2` to enable auto-vectorization for SSE and `-fast -xavx` for AVX. All benchmarks include the `__restrict__` keyword on array pointers, where appropriate, to eliminate the need for interprocedural analysis of array aliasing.

Loop Clasification	Affected Benchmarks
Regular Data and Control	CONV, RADAR, NBODY, MM, STENCIL, KMEANS
Loop Body Control Flow	TSRCH, VR, CutCP, LBM
Strided Data Access	FFT, MRI-Q, NNW, TPACF, LBM
Loop-Carried Dependence	NEEDLE, MERGE
Partially Vectorizable	SPMV, NEEDLE
Impossible Vectorization	None

Table 5.5: Classification of Loops Evaluated

Metric: The performance metric of interest is speedup, which we always report with respect to the baseline scalar code generated with the respective compiler, with `-O3` level optimizations turned on.

Benchmarks: We evaluate our compiler on the two sets of data parallel workloads described in subsection 5.2.4. First, we use the suite of throughput kernels written to provide high data parallelism, which are easier to analyze. Second, we consider the PARBOIL benchmark suite, because the code is complex enough to be challenging for the compiler's automatic analysis, but small enough to be manually optimized for DySER. We chose these benchmarks because they have good data level parallelism and hence are good candidates for acceleration. For both cases, we also implemented hand-optimized DySER code. Table 5.5 classifies the benchmarks according to the five challenges from the case studies presented in Section 4.8.

5.4.2 Automatic vs Manual DySER Optimization

Figure 5.2 shows the speedup of manually-optimized and compiler-generated DySER code relative to the baseline.

Kernels For throughput kernels, manually-optimized DySER code achieves a harmonic mean speedup of $3.5\times$, while automatic DySER compilation yields $2.8\times$. As expected, manually-optimized code is faster than the auto generated code, since programmers can apply application specific knowl-

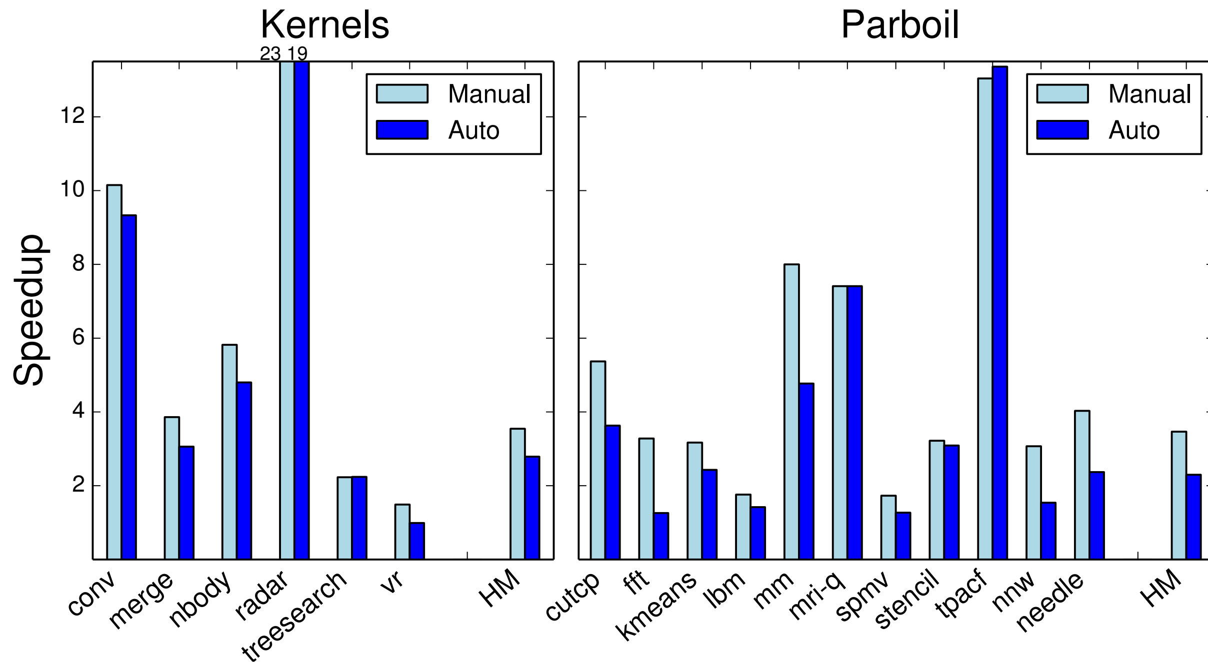


Figure 5.2: Manual vs. Automatic DySER Performance

edge when utilizing the accelerator. What is notable in these results is the number of cases where the DySER compiler generates code that achieves comparable performance to the manually optimized code. For five out of six kernels, our flexible mechanisms give the compiler enough leverage to create a datapath for the loop bodies, and also provide an efficient interface to memory. The only exception from this suite is Volume Rendering (VR), which is difficult to automatically parallelize with DySER because it requires indirect data access and cannot use DySER's flexible vector I/O. The manual version, however, computes multiple rays in parallel using the loop flattening [123] transformation on the outer loop to expose parallelism. This could be an additional optimization for our DySER compiler. However, with a little help from the programmer, the DySER compiler can automatically perform the unroll and use load/store coalescing to achieve the same effect of the loop flattening. Since it requires programmer intervention, we do not report the speedup number here.

PARBOIL Benchmarks. For PARBOIL benchmarks, Automatic DySER compilation provides $2.3\times$, which comes close to the manually-optimized speedup of $3.4\times$.

Compiler Behavior	Benchmarks
Compiler effective	All kernels (except VR) MRI-Q, STENCIL, TPACF, KMEANS
Heuristic Tuning Reqd.	MM
Missing optimization	VR, FFT, NEEDLE, CutCP
Architecture ineffective	LBM, SPMV

Table 5.6: Summary of DySER compiler effectiveness

These provide a spread of behavior and we analyze the results for the four categories in Table 5.6.
Compiler effective (4 of 11): MRI-Q, STENCIL, KMEANS and TPACF perform equally as well in both manual and automatic compilation. This is because the flexible-IO enables the strided pattern in MRI-Q, the "deep" access pattern in STENCIL, and load coalescing in TPACF. KMEANS also attains high performance, but does not reach that of the manual version because it uses an outer-loop unrolling technique to expose extra parallelism.
More heuristic tuning required (1 of 11): For MM, our compiler implementation fails to recognize when mapping reduction to DySER is better than mapping scalar expansion, and it sub optimally chooses scalar expansion.
Missing optimizations (4 of 11): FFT, NEEDLE, NNW, and CutCP achieve less than 70% of manually-optimized code due to missing optimizations. In FFT, the vector length needs to be dynamically chosen. NEEDLE has a long dependence chain caused by unrolling, and CutCP uses long latency functional units, causing long latency execute-PDGs for both. These benchmarks would benefit from software pipelining invocations using a outer loop. Also, the NNW benchmark uses a constant memory lookup table, which makes it hard for the compiler to reason about contiguous accesses. The manual version exploits the patterns in this lookup table, while the DySER compiler falls back on only partial vectorization.
Architecture ineffective (2 of 11): LBM has a large code region, which requires multiple configuration and SPMV has indirect memory accesses. These characteristics mean the DySER architecture is ill-suited for LBM and SPMV, since even manually-optimized code provides speedup of less than 80%.

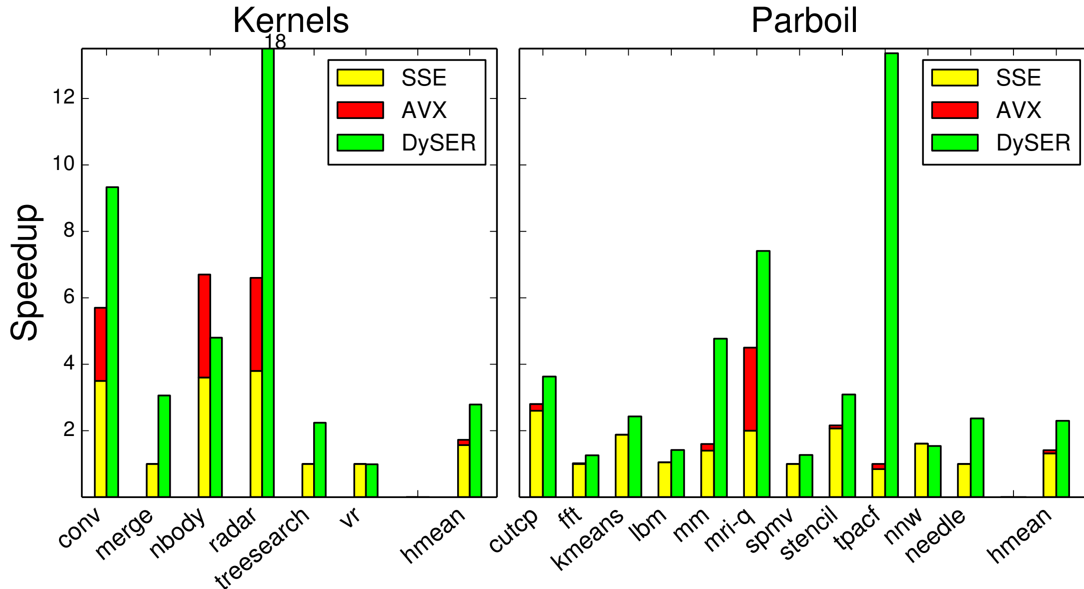


Figure 5.3: Performance of DySER compiled code vs. SSE/AVX and GPU

Observation 5.3. *The DySER compiler generates optimized code that can perform close to manually optimized code, with 30% average performance difference for some benchmarks. Analysis show that much of this difference is because of simply heuristic tuning or algorithmic changes.*

5.4.3 Automatic DySER vs SSE Acceleration

We now compare the compiler+architecture performance of DySER to SSE/AVX. Figure 5.3 shows the speedup of auto-vectorized SSE and AVX and the speedup of compiler generated code for DySER, both measured against the same baseline.

Auto-vectorization provides only about $1.3\times$ mean speedup with SSE and $1.4\times$ mean speedup with AVX, whereas compiler generated code for DySER provides about $2.5\times$ mean speedup. In 3 of 6 kernels, and in 4 of 11 parboil benchmarks, DySER is 2× faster than AVX.

Auto-vectorization is generally effective in the presence of regular memory accesses and no control flow. For example, the automatic compilation of CONV and NBODY performs well for either SIMD or DySER. With complex access patterns or complex control flow, SIMD compilers provide no speedup (6 of 11 Parboil, and 2 of 6 kernels). DySER compilation, on the other hand, shows speedup in all but two cases. These results indicate DySER's AEPDG based compilation for

flexible architectures is more effective than SIMD compilers.

Although both techniques reduce per-instruction overheads, energy reduction from DySER is significantly better than SSE, because DySER is able to handle more types of code and produce more speedups. At times, SSE increases energy consumption because of meager speedups and extra power consumption by the SIMD register file and functional units.

Observation 5.4. *If data level parallelism is available, the DySER compiler exploits it even in the presence of control-flow and irregular memory access. Auto-vectorization with SIMD accelerators is only effective when there are regular accesses and no control-flow.*

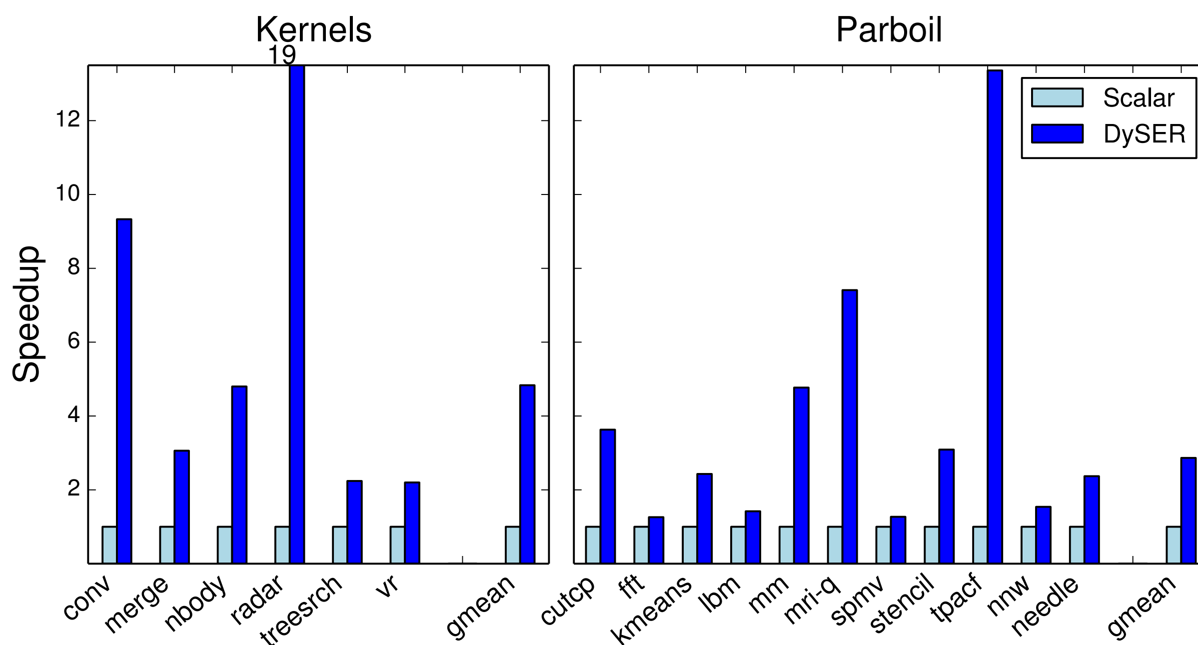
5.5 Performance and Energy Evaluation

This section presents quantitative results on the performance gained and energy efficiency achieved with DySER on the benchmarks considered. For performance, we use speedup relative to the baseline processor as the metric. For energy efficiency, we use the percentage of energy reduction with respect to the baseline processor's energy as the metric. The baseline code is optimized with the highest level optimizations (`-O3`) available in the compiler. The code for DySER is fully optimized with the DySER compiler as described in the previous chapter. Subsection 5.5.1 presents the results for data parallel workloads and subsection 5.5.2 presents the results for control intensive general purpose benchmarks. Section 5.5.3 describes the source of improvements with DySER and potential bottlenecks preventing the DySER architecture from achieving efficiency.

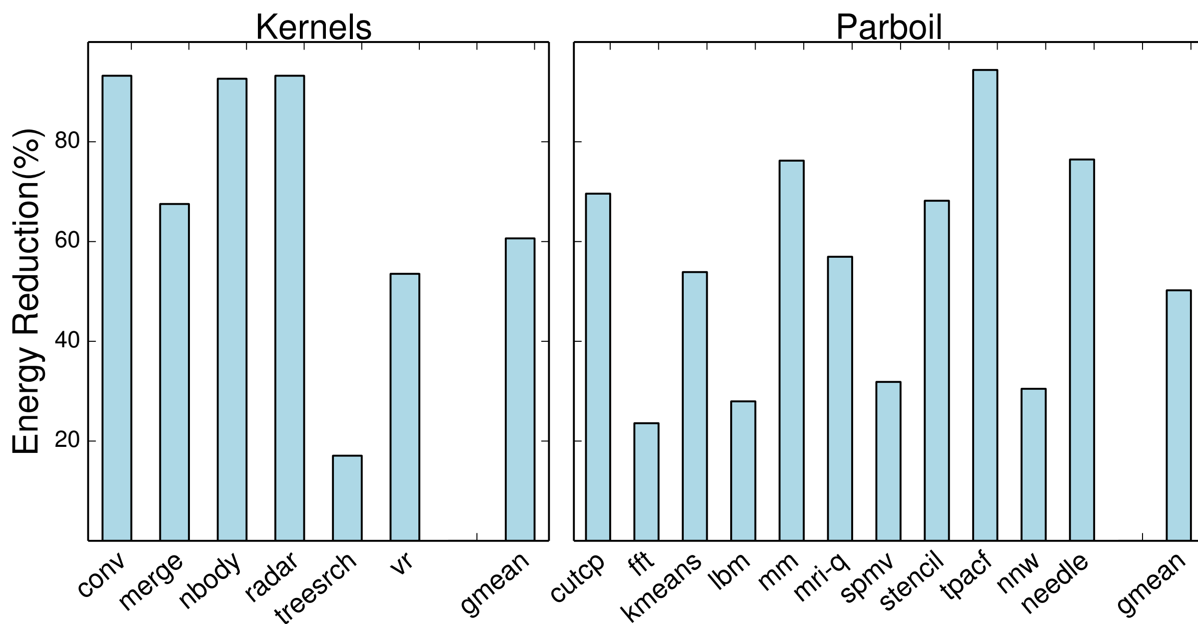
5.5.1 Data Parallel Workloads

Figure 5.4a shows the speedup from DySER integration when compared to 4-wide issue out-of-order core for data parallel workloads. The speedup achieved with DySER is determined by how well its resources are utilized and how many DySER instructions are required for communication.

For the throughput kernels, DySER performs significantly better on average when compared to the baseline processor. The benchmarks `conv` and `radar` have highly regular data accesses and control flow. For such workloads, DySER effectively emulates a SIMD unit and accelerates using vectorized DySER loads and stores. The benchmarks `vr` and `treesearch` have irregular control



(a) Speedup relative to the baseline



(b) Energy reduction with DySER over baseline

Figure 5.4: Speedup and energy comparison for data parallel workloads

flow. Here, we identify and extract independent computations in them and use DySER to specialize the computations. However, we cannot use vectorized DySER instructions beneficially because of data dependent control flow and irregular memory accesses. The benchmark `nbody` has a large code region with reduction. Using the scalar replacement transformation, we exploit the data level parallelism available in this benchmark. However, this requires an additional DySER configuration to DySER to reduce the temporary values outside the inner loop. We use the fast configuration switching mechanism to attain performance improvement. *DySER provides a mean speedup of* $2.8\times$ *over the baseline, with a range of* $1.5\times$ *to* $15\times$ *on highly data parallel workloads*.

Similarly, DySER provides speedup with Parboil benchmarks. On most benchmarks, the benchmarks are $2\times$ faster with DySER than the scalar version on the baseline processor. For `fft`, the input source code is manually unrolled so that the auto-vectorizer can easily vectorize for a SIMD accelerator. Although we can use the unrolled code for vectorized DySER instructions, manually unrolled loops lead to multiple reconfigurations and we cannot amortize the configuration cost, which prevents DySER from providing further speedup. LBM has a large region with control flow. We break the large region into three subregions that it can effectively specialize with DySER. However, configuration switching introduces unnecessary latency through DySER. The benchmarks `spmv` and `nnw` have an indirect memory access pattern and do not have a large computation to memory ratio. For these benchmarks, we specialize the address calculation of the code, which makes the memory subregion dependent on the computation subregion curtailing speedup gain from DySER. DySER provides a geometric mean speedup of $2.2\times$ over our baseline with a range of $1.2\times$ to $13.1\times$.

Observation 5.5. *On data parallel workloads, DySER provides significant performance improvement over the scalar version, because DySER's resources are utilized well with vectorized DySER instructions.*

Figure 5.4b shows the energy reduction when using DySER integrated with an out-of-order processor. On average, DySER consumes 61% less energy for the throughput kernels and 50% less energy for PARBOIL benchmarks. Most of the energy savings for the data parallel workloads comes from the speedup and vector DySER instructions. Since the vector DySER instructions reduce the number of instructions in the processor pipeline to support DySER, they eliminate the energy spent

on various power hungry structures in the processor front end such as th register file, renaming tables, and instruction queue. Even though DySER does not provide huge speedups for TreeSearch and VR, it provides energy saving because it eliminates per-instruction overheads such as fetch, decode, rename, and register reads and writes for instructions that are specialized with DySER.

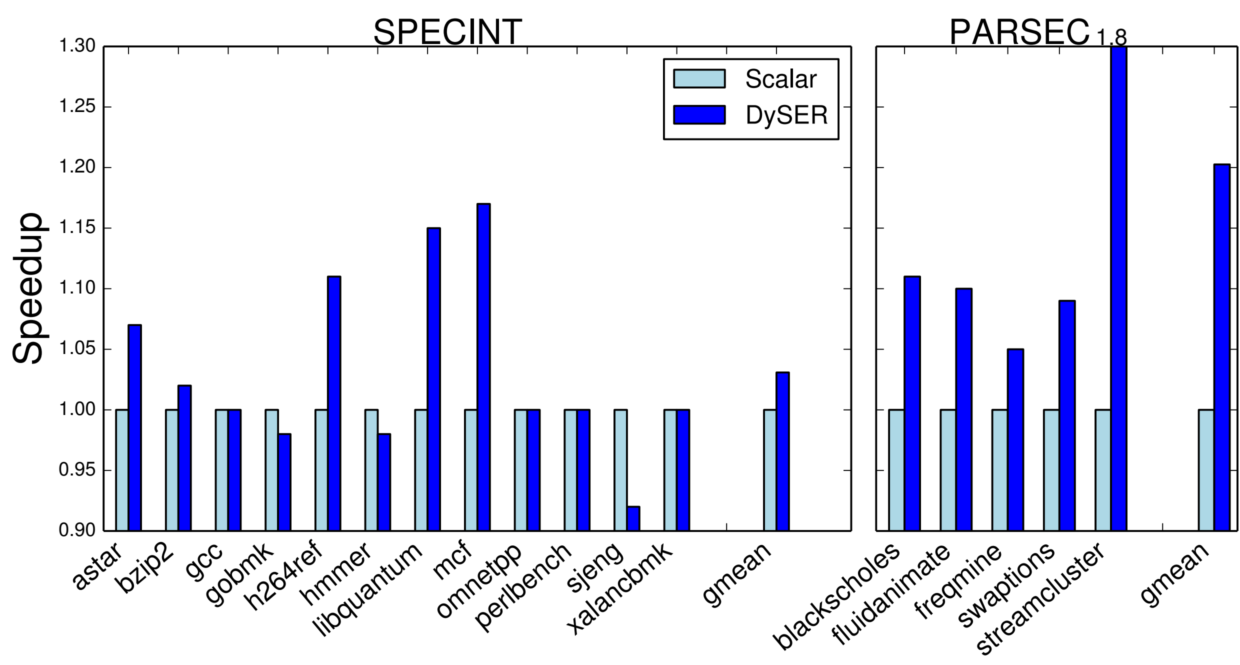
Observation 5.6. *On data parallel workloads, DySER reduces energy consumption significantly over the baseline processor, because it accelerates the computation, uses DySER vector instructions effectively and eliminates per-instruction overheads.*

5.5.2 General Purpose Workloads

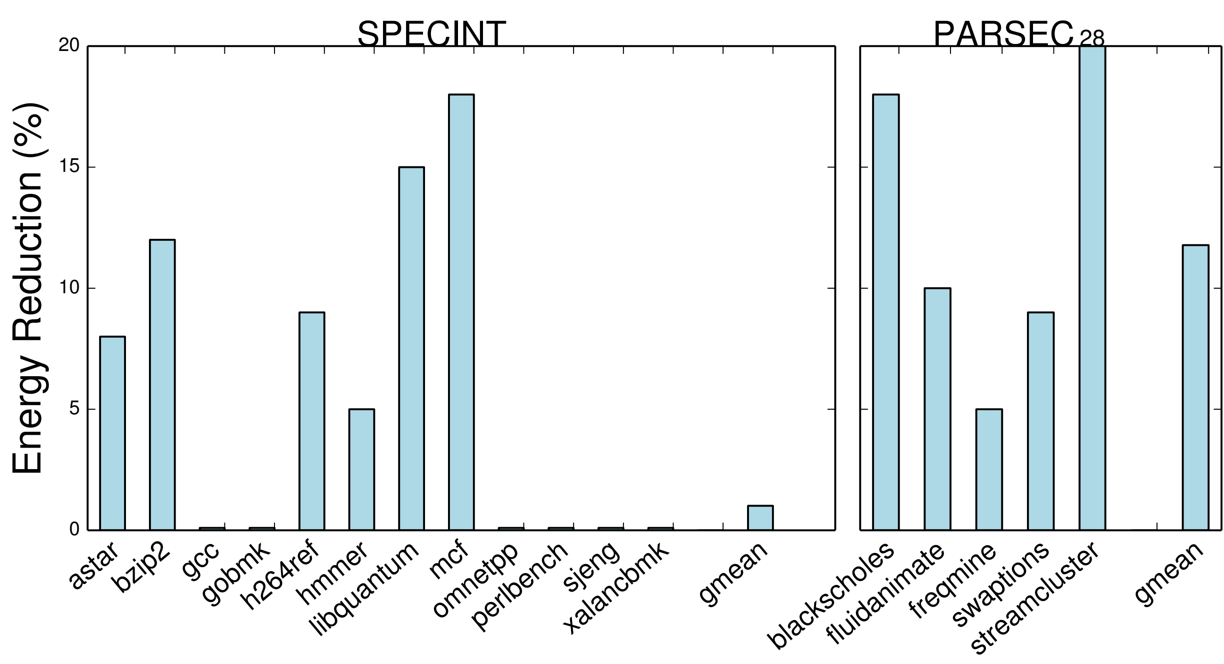
Figure 5.5a shows the speedup attained for SPECINT and PARSEC. We use the DySER compiler to compile these benchmarks. DySER speeds up the SPECINT benchmarks by only 3% on average. To analyze the performance, we classify the SPECINT benchmarks into three categories. The first category contains benchmarks, where the DySER compiler deemed it not worthwhile to use DySER. The second category are the benchmarks that the DySER compiler chooses to specialize with DySER, but when it does not provide any performance benefits. The third category is comprised of benchmarks where we see performance improvements with DySER.

The benchmarks `gcc`, `omnetpp`, `perlbench` and `xalancbmk` belong to the first category. For `gcc`, the heuristics in the DySER compiler chooses not to use DySER to specialize as the most frequently executing regions are either in the libraries or very small. For `omnetpp`, the compiler does not find large enough regions to specialize with DySER, as the benchmark uses small methods in a class heirarchy to perform its computation. This prevents the compiler from identifying the regions that can be specialized with DySER. Similar to `gcc`, the DySER compiler skips `perlbench` and `xalancbmk` due to lack of good regions.

The second category has the benchmarks `gobmk`, `sjeng` and `hmmer`. For `gobmk` and `sjeng`, the most frequently executing regions are functions with multi-loops. i.e., they have nested loops with multiple inner loop and switch/case statements to select different inner loops. The DySER compiler identifies the inner loops as candidate regions for specialization. However, during run-time, configuration switching and small region sizes hurt performance, as they add extra overheads



(a) Speedup relative to the baseline



(b) Energy reduction with DySER over baseline

Figure 5.5: Speedup and energy comparison for data parallel workloads

over the out-of-order execution. For `hmmer`, the most frequently executing code region requires 12 inputs from memory, which causes DySER to mostly wait for the operands from memory to arrive.

The benchmarks `astar`, `bzip2`, `h264ref`, `libquantum` and `mcf` are in the third category, where DySER provides speedup. Although they have control intensive code, the DySER compiler is able to identify computation that can be specialized with DySER. For example, in `h264ref`, the most frequently executing function `SetupFastFullPelSearch` has a large code region, which computes the sum of absolute difference over four blocks concurrently to exploit instruction level parallelism. The DySER compiler identifies the code region as a candidate for specialization and uses subgraph matching to split the large region into appropriatly sized regions for DySER. In `mcf`, the most frequently executing function `primal_bea_mpp` computes a cost from a data structure and performs multiple checks on the value of cost. Depending upon the output of the checks, it updates the data structure that it uses to compute the cost. These branches are not easily predictable as they are data dependent, and therefore cause frequent pipeline flushes. In the DySER case, the compiler offloads the computation of the cost and decision to update the data structure to DySER. In effect, it replaces multiple branches with a single DySER branch instruction, which prevents most of the pipeline flushes. This allows the main processor pipeline to run ahead and perform better than the baseline processor without DySER.

Figure 5.5 also shows the speedup provided by DySER on PARSEC benchmarks. On all benchmarks considered, DySER provided speedup, with a mean speedup of 20%. The DySER compiler correctly identified the most frequently executing code regions as a candidate for DySER. However, the code regions in these benchmarks require more resources than DySER provides which causes them to either use processor resources or be split into multiple code regions, limiting overall gain. For example, in the `blackscholes` benchmark, the kernel that is responsible for more than 80% of the dynamic instruction is the function that computes the cumulative normal distribution (CNDF). However, it has a call to the `exp()` library function, which prevent the DySER compiler from using vector instructions. This curtails the performance gain from DySER.

Observation 5.7. *For highly irregular programs, which have data dependent control flow and irregular data accesses, DySER does not improve the performance as it does with data parallel workloads. This type of*

benchmark may not be suitable for DySER if speedup is the primary goal.

Figure 5.5b shows the energy reduction provided by DySER. For SPECINT, the geometric mean energy reduction is 10% when we do not count the benchmarks that do not have any energy reduction. For PARSEC, the geometric mean energy reduction is 11%. For some benchmarks like `hmmer`, DySER does not speed up their computation, but improves energy efficiency. The reason is that the number of register reads and writes are reduced because computation is offloaded to DySER and its efficient circuit switched network. For other benchmarks like `sjeng` and `gobmk`, DySER not only slows down the program, but also consumes more energy than the original processor. This is because of heavy configuration switching inside DySER, which causes subsequent DySER instructions to stay in the instruction queue for longer than necessary.

Observation 5.8. *For general purpose workloads, DySER reduces energy consumption by a modest amount compared to an out-of-order processor. The energy benefits for general purpose workloads are mainly from removing per-instruction overhead rather than from speedup.*

5.5.3 Source of Improvements and Bottlenecks

In this subsection, we identify the source of DySER's efficiency and potential bottlenecks that prevent it from providing more performance gain. The efficiency provided by DySER comes from its ability to concurrently execute many operations with its array of heterogeneous functional units in data flow fashion. The potential bottlenecks are the DySER instructions that need to communicate with DySER from the processor pipeline, and the latency through the DySER substrate. We use the throughput kernels and Parboil to study the sources of efficiency and bottlenecks.

5.5.3.1 Source of Improvements

DySER's major source of performance improvements over an out-of-order processor is its ability to concurrently execute many operations in its heterogeneous array of functional units and emulate a wider processor than the baseline processor. Figure 5.6 shows the instruction per cycle (IPC) for the baseline processor and for the DySER integration with the same baseline processor. The gray colored `Baseline Core IPC` bar shows the IPC for the baseline processor and the blue

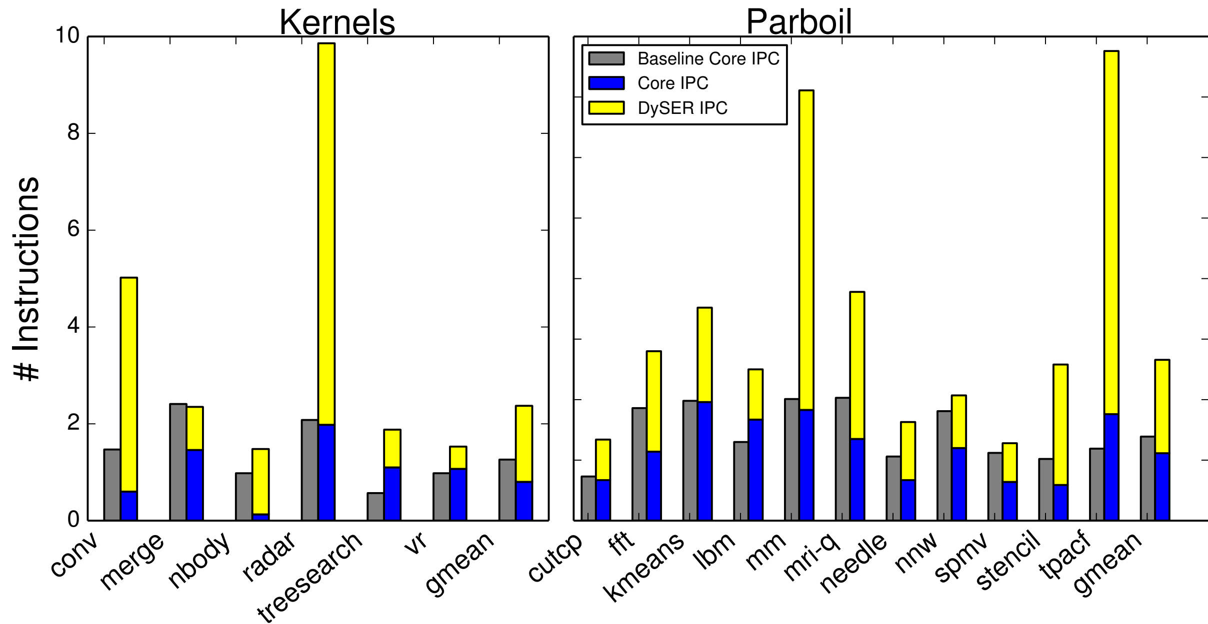


Figure 5.6: Effective IPC comparison

colored `Core IPC` bar shows the IPC of the main processor of the DySER integration, and the yellow colored `DySER IPC` shows the average number of operations executed per cycle.

For throughput kernels, the baseline processor has a mean IPC of 1.26, whereas DySER integration's effective IPC is 2.4. The effective IPC is the sum of DySER's average operation per cycle and the baseline processor's IPC. Similar to the throughput kernels, for PARBOIL, the baseline processor has a mean IPC of 1.38, and DySER has an effective IPC of 2.56. This shows that, when integrated to a core, DySER integration effectively emulates a wider issue processor. For all benchmarks, DySER's effective IPC is higher than the baseline processor. However, the mean IPC of the main processor integrated with DySER is lower than that of the baseline processor. For benchmarks `treesearch`, `vr`, `lbm`, `tpacf`, the IPC for main processor integrated with DySER is higher than that of the baseline processor. For `treesearch` and `vr`, and `tpacf`, data dependent branch instructions are mapped to DySER, which eliminates branch mispredictions. This causes the number of useful instructions committed per cycle to decrease in the baseline processor. For most of the benchmarks, the main processor, when integrated to DySER, executes fewer instructions than the baseline because the

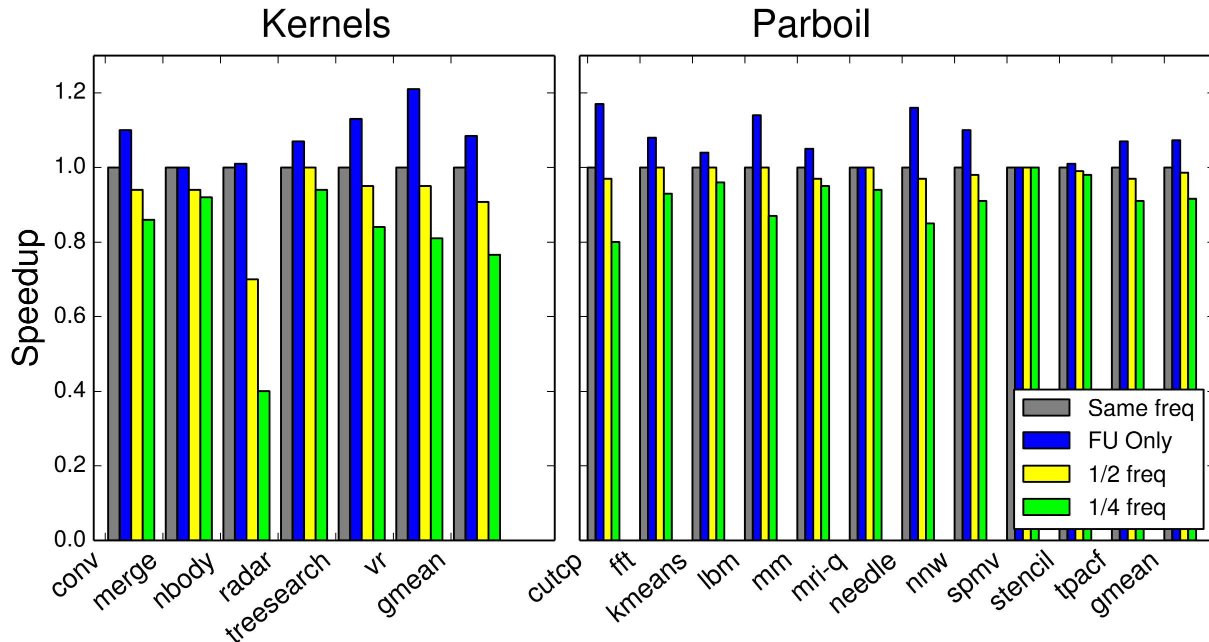


Figure 5.7: Speedup due to DySER latency

computation slice is offloaded to DySER. Further, as shown in Figure 5.6, because it executes fewer instructions per cycle, the power consumed by the main processor is lower than that of the baseline processor. Since DySER executes the instructions from the computation region with its energy efficient substrate, the overall energy consumption of the processor is reduced.

Note that since the number of instructions executed on the baseline processor is different from the number of instructions executed on the DySER integration, the performance gained with DySER is not directly proportional to the effective IPC presented in the Figure 5.6.

5.5.3.2 Potential Bottlenecks

Although the DySER architecture is efficient and provides performance gain along with energy efficiency, it introduces two main overheads over an out-of-order implementation. First, it introduces extra latency between the operations in DySER as they need to communicate through the circuit switched network. Second, DySER requires extra instructions in the processor pipeline to send and receive register values.

Potential Bottleneck: Latency We identify the latency through the DySER substrate as one of its bottlenecks. In a general purpose processor's pipeline, the results can be consumed within a cycle or two because of forwarding to a dependent instruction in the pipeline. However, for DySER, if dependent operations are mapped spatially further away from the source functional units, the latency of the computation increases. To understand this bottleneck, we simulate DySER without any latency from switches. Also, to understand the impact of slowness due to the latency of DySER, we simulate DySER with half the frequency of the processor and a quarter the frequency of the processor. This effectively increases the latency of the computation in the DySER by two or four times. Figure 5.7 shows the relative speedup of the overall performance due to latency of DySER. As known in the second bar in the Figure 5.7, the functional unit latency actually dominates the overall latency of DySER as not counting latency from switches improves speedup only by 8%. Except `nbody`, all benchmarks do not slow down beyond 80%. With `nbody`, as we slow down DySER, the overall performance also drops. This benchmark has two long latency functional units (FP-DIV and FP-SQRT) in series, which makes this kernel sensitive to latency. Apart from this outlier benchmark, other benchmarks are not sensitive to the latency of DySER. They continue to provide speedup over the scalar version with DySER. This shows that pipelining with multiple invocations effectively hides the functional units and network latency inside DySER.

Potential Bottleneck: DySER Instructions Overheads As described before, DySER requires the main processor pipeline to feed operands for the computation in DySER. This is done so that the interface between DySER and rest of the microarchitecture is simple and easier to integrate. However, having extra instructions to communicate the values between the processor pipeline and DySER introduces extra run-time overheads. In this study, we quantify the overhead by measuring the number of DySER instructions executed and other memory access instructions like address calculation, loads and stores executed in the processor pipeline and compare that to the original instructions.

Figure 5.8 shows the percentage of dynamic instructions executed relative to the number of dynamic instructions executed when original program executed in the baseline processor. The first bar shows the dynamic instruction count for the original program while executing in the baseline

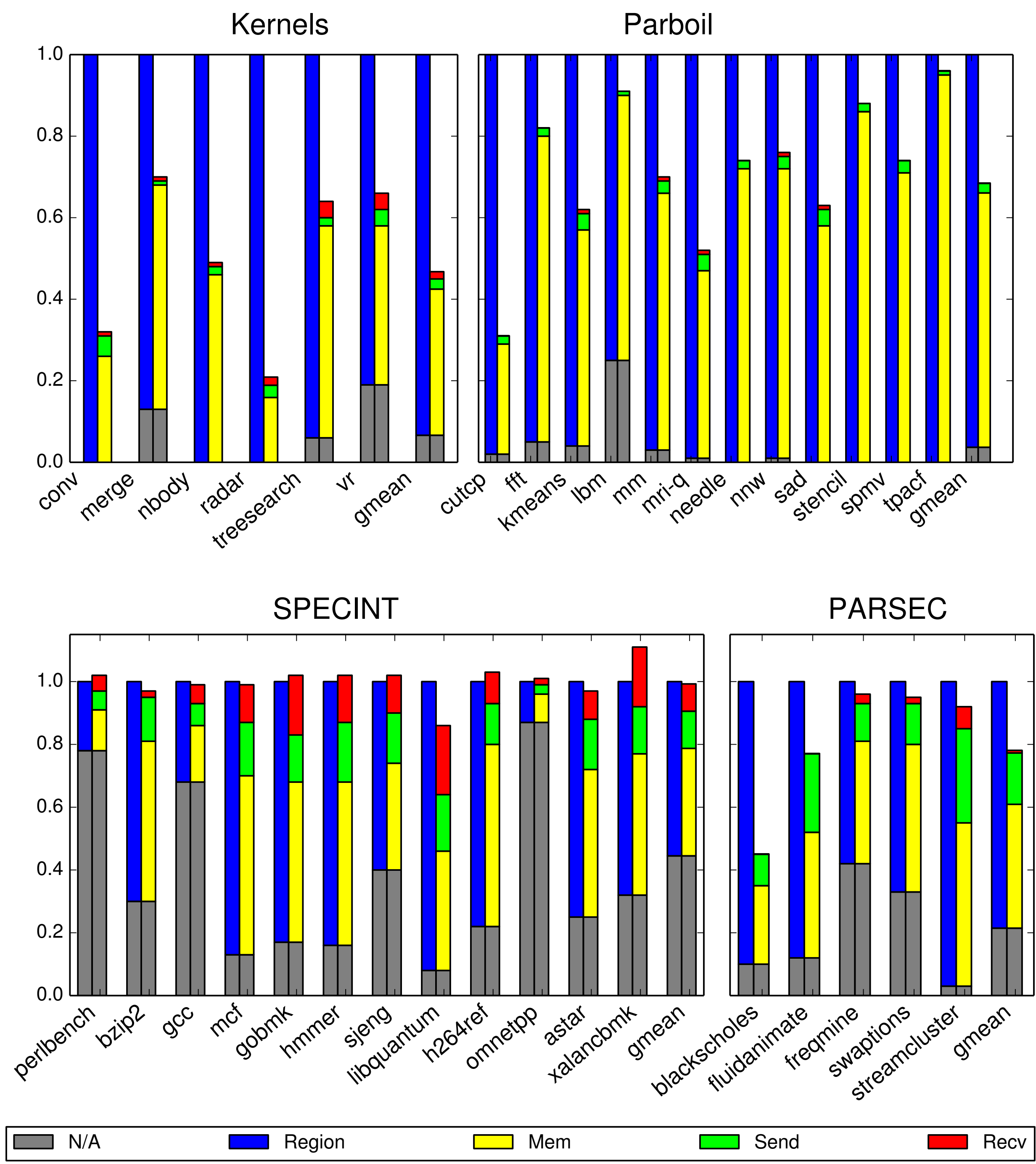


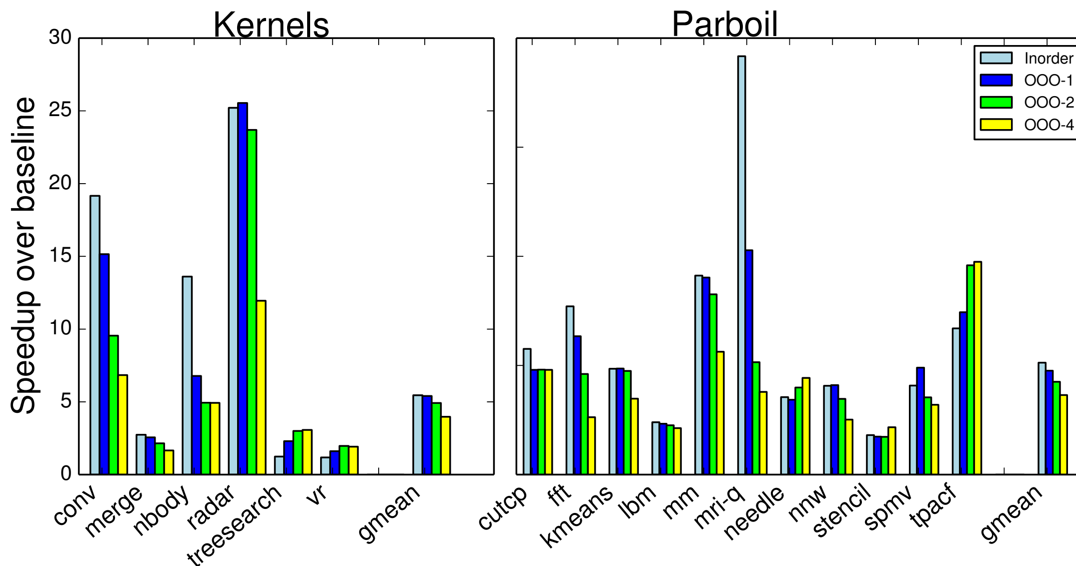
Figure 5.8: Relative Dynamic Instructions Count

processor. The second bar shows the dynamic instruction count for the program compiled with the DySER compiler. The gray shaded `N/A` bar represents the regions that the compiler chose not to specialize with DySER. This includes outer loops, recursion calls, and functions without any loop. The blue shaded `Region` bar represents the number of dynamic instructions from the code region that the compiler chose to specialize. The yellow shaded `Mem` bar represents the number of dynamic instructions that are in the memory access subregion. The `send` bar and `recv` bar represent the number of dynamic instructions executed to send and receive values to DySER.

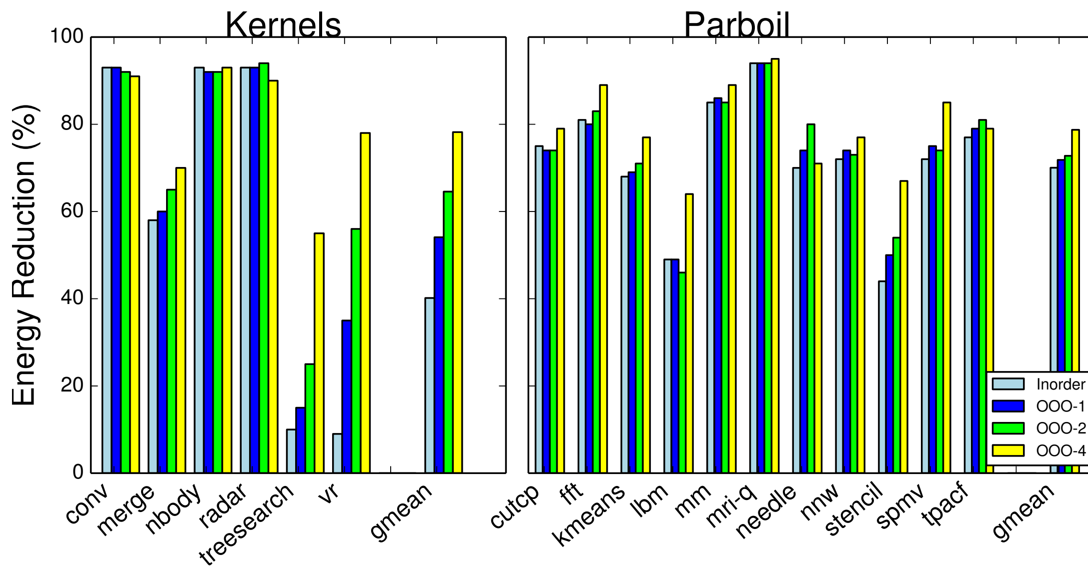
Even though the DySER instructions introduce overheads for communicating with DySER, the overall dynamic instruction count is reduced in the processor pipeline for data parallel workloads because the compute region is offloaded to DySER. For data parallel workloads, the number of dynamic instructions is reduced by 57%, and only a few DySER instructions are required to communicate with DySER. The reason is that for data parallel workloads, the DySER compiler enables vectorized communication.

For the general purpose workloads, the DySER compiler does not reduce the dynamic instruction significantly, and in fact for some benchmarks, it exceeds the original dynamic instruction count. For SPEC, about 98% of the original dynamic instructions are required with DySER, and for PARSEC, 80% of original instructions are needed. The reasons for this behavior are two fold. First, the DySER compiler does not find specializable regions that are amenable to DySER. For example, in omnetpp, it only finds that 22% of dynamic regions are specializable because it is implemented with C++ and it performs its computations by using a class hierarchy. The DySER compiler does not identify this type of code region as specializable. Second, SPEC INT code is irregular and cannot use vector instructions to reduce the overhead of DySER instructions.

Observation 5.9. *The overhead of DySER instructions is not high for data parallel workloads as they can exploit data level parallelism with DySER vector instructions. For control intensive general purpose workloads, the overhead of DySER instructions may hurt the performance gain and energy efficiency. Further mechanisms may be required to reduce their effects on performance and energy.*



(a) Speedup relative to the baseline



(b) Energy reduction with DySER over baseline

Figure 5.9: Senstivitiy Study

5.6 Sensitivity Study

In this section, we present a sensitivity study on DySER by varying the width of the baseline processor. We integrate DySER with an inorder processor, a dual issue, 4-wide and 8-wide out-of-order processor and evaluate the data parallel workload to understand the performance sensitivity to the width of the processor.

Figure 5.9a shows the relative speedup of the data parallel workloads. The speedups shown are relative to their respective baseline. We perform this study using the manually optimized code instead of compiler generated code in order to study the raw effectiveness of DySER and the processor without any compiler inefficiencies.

For these benchmarks, the performance benefit of having DySER decreases as the main processor can issue more instructions. The reason is that the wider processor can execute more instructions speculatively and exploit instruction level parallelism better. This matches DySER's benefits, as it exploits the instruction level parallelism to provide performance improvements. However, for some benchmarks, like `treesearch`, `vr`, and `tpacf`, the DySER actually provides more performance gain for the wider issue processor. These benchmarks have data dependent branches in their kernel, which make the baseline processor mispredict more often. These mispredictions cause the processor pipeline to squash and restart. However, with DySER, these data dependent branches are mapped to DySER and the memory access slice without these branches executes in the processor pipeline. Since a wider processor executes faster, it injects the data values faster to DySER. As the width of the processor increases, the processor sends the operands faster and DySER can perform the operations faster and deliver the outputs.

Figure 5.9b shows the energy reduction from DySER to their respective baseline. Although the performance gain from DySER diminishes as the width of the processor increases, the energy reduction due to DySER increases. For throughput kernels, DySER integrated with an inorder processor provides about 40% energy reduction due to the speedup. But with an 8-wide out-of-order processor, it reduces energy by as much as 80%, as the DySER does not require power hungry structures to achieve speedup. Parboil also shows a similar benefit in energy reduction.

Observation 5.10. *As width of the processor increases, speedup from DySER diminishes because the base-*

line processor itself can perform better. However, the energy reduction due to DySER increases as it eliminates power hungry structures that a wider processor may require.

5.7 Evaluation on Database Kernels

The goal of this study is to evaluate DySER on a highly relevant application, where both data parallel and irregular code patterns coexist. In this section, we first present the evaluation study on main memory database primitives or operators that are generally used in decision support queries. Then, we describe how DySER helps to achieve speedup on a full query from TPC-H benchmarks. For each database primitive, we describe the high level implementation of the primitive and how DySER can accelerate it. Then, we evaluate the primitives with DySER and present results on the speedup obtained compared to scalar code. With the advent of SIMD units in commodity processors, main memory databases exploit the fine grain parallelism available through vectorization [14]. To understand whether DySER can compete with SIMD acceleration, we also present the speedup of these primitives when the code is vectorized with SSE, as it is a widely adopted SIMD architecture, and compare that to DySER.

5.7.1 Database Primitives

In this subsection, we describe the database kernels that we intended for DySER to specialize. For each database primitive, we describe its operations and how SSE units can exploit the fine grain parallelism available. We then describe how we can specialize these primitives with DySER. In order to evaluate the database query engines, we identify the following relational database primitives: `scan`, `sort`, `strcmp`, `aggregation`.

Scan: An important database primitive is the `scan` of the full table, since this is typical of ad hoc queries for business intelligence. Figure 5.10 shows the pseudo code for `scan` for a column oriented database. The inputs to the scan are a column of data, a key to scan the column data, and an input bitvector that filters the input column. The output is a bitvector that represents the scan result. In this study, we consider an equality scan over a large number of columns. This primitive has high

```
inputs: in_mask:bitvector, col, key
output: out_mask:bitvector
 for (i = 0; i < LEN; i += SZ) {
   for (j = 0; j < SZ; ++j) {
     data = col[i*SZ + j];
     out |= (data == key) << j;
   }
   out_mask[i] = in_mask[i] & out;
 }
```

Figure 5.10: Scan

data-level parallelism because the processing for subsequent rows is independent of the previous row.

For SSE, the inner loop can be easily vectorized, and using the SSE instruction `movmskps`, four bits can easily be packed together to create the output mask. Similarly, for DySER, we vectorize the inner loop with the DySER's vector instruction. Since DySER has far more functional units than SSE that we can control, it emulates a wider SIMD unit that compares and packs 8 bits instead.

A variant of `scan` is the `scan` on compressed data with run-length encoding (RLE). The run length encoded data is a very simple form of data compression in which the sequence of same data is stored as a single data and count rather than the full sequence of data. In order to exploit fine grain data level parallelism, the RLE data is blocked such that the run length of the data are aligned.

DySER uses its vectorized DySER instructions to specialize the scan primitive which scans the blocked run length encoded data. However, SSE and its compiler fails to vectoize this primitive. In order to vectorize this primitive, a vector instruction which shifts the elements in a vector with variable shift amount is needed. Since SSE does not have such a instruction, it does not vectorize it. However, DySER creates a specialized datapath in its hardware substrate to perform that operation and uses DySER vector instructions to fetch multiple data elements at once, and then send them directly to DySER.

Sort To exploit the fine grain parallelism with SSE better, `sort` is implemented with the bitonic sort algorithm. Figure 5.11 shows the bitonic sort merge network that merges two sorted sequences of length 4 and creates a sorted sequence of length 8. Bitonic merge requires that one sequence be sorted in ascending order and the other be sorted in descending order. The order of the inputs A

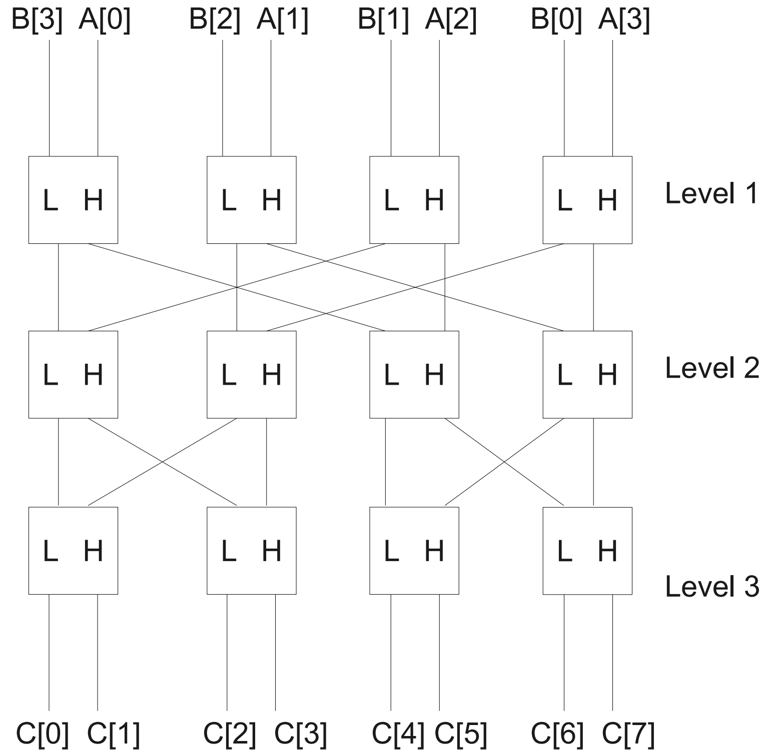


Figure 5.11: The bitonic merge network for merging sequence of data of length 4. A, B are the input sequences and C is the output sequence of length 8.

and B in the figure are shown after the sorted order of B has been reversed. Each level examines the elements in parallel using min and max operations as denoted as L and H in the figure respectively. Using this merge operation as a primitive, a full array of columns can be sorted using the algorithm presented in Chhugani et al [20].

For DySER acceleration, we schedule the bitonic merge network shown in the figure 5.11 to DySER. The inherent shuffling in the merge network is mapped to the physical routes inside the DySER and does not require extra shuffle instructions before using min/max operations at every level. In the case of SSE, it needs to use extra shuffle instructions to align the values between the level.

Strcmp In this kernel, we evaluate variable length string compare on DySER. In this kernel, DLP is readily available. Without special instructions such as PCMPISTR, which is a SSE4 instruction that

```
Inputs: K, keys
        V, values
Outputs: A, aggregated values

for (i = 0; i < LEN; i ++):
  A[K[i]] += V[i]
```

Figure 5.12: Aggregation

performs string comparison, SSE cannot speed up this kernel. Even with this special instruction, SSE requires the string to be aligned which makes the special SSE instructions harder to use.

Similarly, with naive mapping of the string compare kernel to DySER, it actually slows down the computation slightly (about 5% slowdown with DySER). There are two reasons for this behavior.

- The String compare kernel has a high memory to computation ratio. It loads two values and does only one compare. The overheads in terms of DySER sends and receives dominate and eliminate any potential benefit from DySER.
- It has data dependent loop control which renders most vectorization techniques ineffective.

Instead, first we stripmine the STRCMP loop and map the stripmined loop to DySER. This improves the performance by 10%. In this workload, about 1/3 of the keys are identical. Second, we map this primitive to DySER such that it aggregates the compare functional units' output into a 16 bit bitmask and receives the whole 16 bit bitmask as one integer value using `dyserrecv` instruction. With this change, the processor receives the compare results in batches, which reduces the overheads in `dyserrecv` instructions. This provides another 20% performance improvement. With both of these strategies, the string compare kernel with DySER is 27% faster than the baseline for X86.

Aggregation: Figure 5.12 shows the code for the aggregation primitives. It performs aggregation of the values from the V array to the A array using the keys in the K array. Since the keys in K can have aliases, this code cannot be vectorized easily and we cannot use SSE instructions to accelerate this code beneficially. This operator represents the worst case for DySER because it has only one computation and others are just address calculations.

```
Inputs: K, keys
        V, values
Outputs: A, aggregated values

for (i = 0; i < LEN; i +=2 ): {
  # Can vectorize this with DySER instructions
  K_i0   = K[i], K_i1 = K[i+1];
  A_K_i0 = A[K_i0];
  A_K_i1 = A[K_i1];
  # Can vectorize this with DySER instructions
  V_i0   = V[i], V_i1   = V[i+1]

  // Inside DySER
  int O0 = A_K_i0 + V_i0;
  int O1 = ((K_i0 == K_i1) ? O0: A_K_i1 + V_i1

  // Outside DySER
  A[K[i]] = O0;
  A[K[i+1]] = O1
}
```

Figure 5.13: Aggregation

However, we can unroll the loop, and vectorize the loading of the V array using the partial vectorization technique similar to the case study performed in section 4.8. However, to map the addition inside DySER, we need to perform the aliasing check on K. Figure 5.13 shows the unrolled loop before mapping the compute subregion to DySER. Although it increases the number of operations to be performed by DySER, it decreases the number of instructions required to load and store instructions because of vectorizable loads.

With aliasing check performed with in DySER, we achieve speedup of 56% over baseline when integrated with an out-of-order processor, because DySER can exploit the consecutive accesses to memory and uses the out-of-order processor's ability to do memory disambiguation effectively for non-consecutive access to memory.

As a variant of `aggregation`, we also implemented and evaluated Jenkin's Hash with aggregation. This primitive computes the keys in the aggregation kernel with Jenkin's Hash function and then aggregates the values. Since the `aggregation` does not have a computation subregion for DySER, computing Jenkin's hash in the kernel provides DySER with more operations to be mapped.

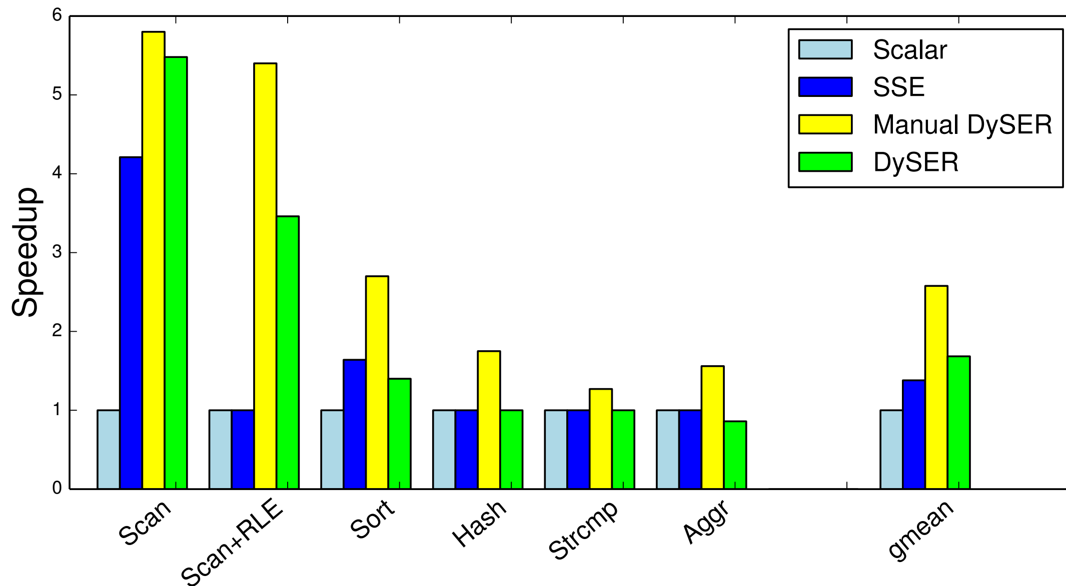


Figure 5.14: Database kernels speedup comparison

Quantitative Evaluation Figure 5.14 shows the speedup of the database primitives on DySER and SSE compared to the baseline processor. The second bar shows the speedup of the SSE over the scalar processor. The third bar shows the speedup provided by DySER when the primitives are manually optimized with compiler intrisics just like the SSE. The fourth bar shows the relative speedup of DySER when the primitives are compiled with the DySER compiler. SSE provides a geometric mean speedup of 40% over the scalar version, whereas DySER provides a gemeotric mean speedup of 70% over the scalar version. However, if we optimize the kernels manually, DySER can provide a mean speedup of $2.5\times$.

For `scan`, both DySER and SIMD exploit fine grain data level parallelism and provide similar speedup for the same memory bandwidth (16 bytes per cycle). The reason DySER performs slightly better than SSE is that it emulates a larger SIMD unit and exploits the pipeline parallelism in DySER better. For SCAN+RLE, SSE fails to vectorize and does not provide any benefit. However, DySER uses it flexible hardware substrate to create the special operation described above to exploit the data level parallelism.

Similarly for `sort`, DySER and SSE use the bitonic sort merge network to exploit data level

```
SELECT l_returnflag, l_linestatus,
    sum(l_quantity) as sum_qty,
    sum(l_extendedprice) as sum_base_price,
    sum(l_extendedprice * (1 - l_discount))
        as sum_disc_price,
    sum(l_extendedprice * (1 - l_discount) *
         (1 + l_tax)) as sum_charge,
    avg(l_quantity) as avg_qty,
    avg(l_extendedprice) as avg_price,
    avg(l_discount) as avg_disc,
    count(*) as count_order
FROM lineitem
WHERE l_shipdate < date '1998-09-01'
GROUP BY l_returnflag, l_linestatus
```

Figure 5.15: Query 1 of TPC-H

parallelism. However, DySER performs better than SSE because the shuffle operations that are needed to perform bitonic merge are mapped to the DySER network itself, thereby eliminating extra instructions from the processor pipeline.

Since there is no beneficial way to use SSE to accelerate `hash`, `strcmp` and `aggregation` primitives, SSE fails to provide speedup. However, DySER maps the computations from the primitives described earlier and uses the flexible substrate and native mapping of control flow to accelerate the kernels.

Observation 5.11. *When data level parallelism is available in the database primitives, both DySER and SSE improve the performance significantly. When DLP is limited or not available, DySER provides an average of 50% speedup over the baseline, whereas SSE fails to improve.*

5.7.2 Full Query Evaluation

As described in the previous section, DySER provides significant speedup over the baseline when DLP is available and provides modest improvement when DLP is not available. In the query processing, both types of primitives will be used to achieve the task. In this section, we evaluate a full query on DySER to understand the trade offs and the effect of DySER when both types of primitives are needed to achieve the task.

Figure 5.15 shows query 1 of the TPC-H benchmark [103]. We chose query 1, because it is CPU-bound and hence a good candidate for specializing with DySER. Also, this query's plan

```
SELECT
    sum(l_extendedprice * (1 - l_discount))
    sum(l_extendedprice * (1 - l_discount) *
          (1 + l_tax))
FROM lineitem
WHERE l_shipdate < date '1998-09-01'
GROUP BY l_returnflag, l_linestatus
```

Figure 5.16: Simplified query 1 of TPC-H

is simple and easy to understand and does not require any complex optimizations or joins. It is basically a scan on the `lineitem` table of 6 million tuples, that selects most tuples. It then computes a number of fixed point decimal expressions and performs eight aggregates. Since the aggregate grouping is on two single character columns, and only has four unique combinations, the hashing computation can be performed easily, requiring no additional computation resources.

As explained before, main memory databases exploit the fine grain parallelism through SIMD units in commodity processors [14]. They create vectorized query plans, in which the query is partitioned into a sequence of primitives and each primitive operates on data with simple computations so that the compiler can easily auto vectorize these primitives.

However, if we use the vectorized query plan, this does not fully exploit DySER's pipeline parallelism and has a high memory to compute ratio. We evaluate the performance of TPC-H query 1 with three looping strategies to understand the tradeoffs involved with DySER. They are

1. **Single Loop or JIT:** In this strategy, we process the whole query with a single loop. The advantage of this approach is that the loop consumes the data fully and there are no extra loads and stores. The disadvantage is that the loop is not vectorizable since it contains multiple aggregation kernels.

2. **Multiple Loops or Vectorized**: In this strategy, we partition the query processing into multiple loops in which we perform only one operation and materialize the intermediate values. The advantage is that some of the loops are vectorizable, but requires additional loads and stores. However, if the temporary storage is small enough to be in the caches, the extra loads and stores will not affect the overall performance.

```
for (i = 0; i < N; i += BLOCK_SIZE) {
  # compute projections
  for (j = 0; j < BLOCK_SIZE; ++j) {
     base_price = table['price'][i]
     disc   = table['discount'][i]
     tax    = table['tax'][i]
     disc_price[j] =  base_price * (1-disc)
     charge[j]     =  disc_price[j] * (1+disc)
   }
  # compute hash
  for (j = 0; j < BLOCK_SIZE; ++j) {
     hash[j] = compute_hash(table['returnflag'][i+j],
    table['linestatus'][i+j])
  }
   #aggregation
   for (j = 0; j < BLOCK_SIZE; ++j) {
     result.sum_disc_price[hash[j]] += disc_price[j]
     result.sum_charge[hash[j]]     += charge[j]
   }
}
```

Figure 5.17: Hybrid loop or Partitioned looping strategy for the simplified query

3. **Partitioned Loop or Hybrid**: In this strategy, we partition the loops such that we can exploit the data level parallelism and the data locality simultaneously. Figure 5.17 shows the pseudocode for this strategy for a simplified variant of query 1. The simplified query is shown in Figure 5.16

Figure 5.18 shows the performance results for query with DySER and SSE. It shows the speedup relative to scalar version of the JIT compiled loop. i.e. using one loop to perform the full query. In all cases, the vectorized version is slower than its JIT or Hybrid counterparts because of the extra instructions required to load from and store to the memory. SSE does not provide any performance improvement on the vectorized loops because of the aggregation primitive which does not lend itself to vectorization. However, DySER with the JIT version is 2.7× faster than the scalar version because it exploits its pipeline parallelism to accelerate the computation in the full query.

From the performance results for DySER, we observe that DySERization of the single loop implementation performs better than other approaches. The reason is that all operations from the single loop can be mapped to DySER without overflowing, which creates a customized hardware accelerator in the DySER. By exploiting the pipeline parallelism, DySER performs better than the

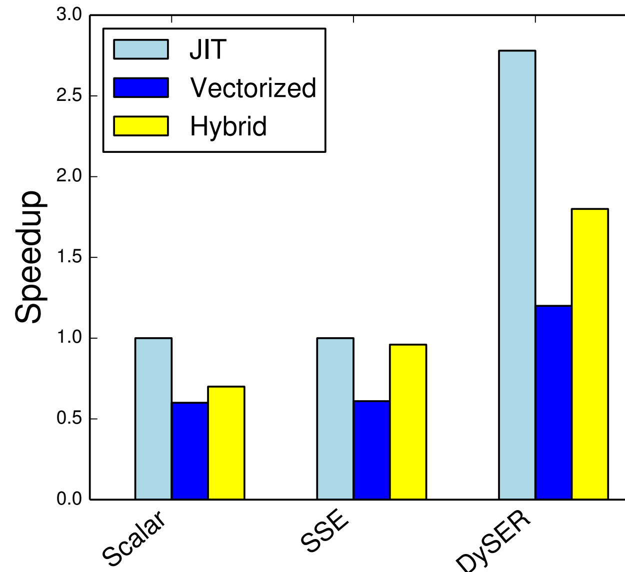


Figure 5.18: Performance results for TPC-H query with SSE and DySER on three looping strategy

other approaches. If the computation does not fully map to the DySER, the hybrid version may be the better option. To fully utilize DySER's potential, a careful tradeoff between looping strategy, appropriately computation size, and exposing the DLP available are required.

5.7.3 Database Evaluation Summary

DySER exploits both pipeline parallelism and data level parallelism to improve performance. With database kernels that have DLP, such as SCAN, SORT, and PROJECT, DySER provides more than 2× speedup, which is similar to SIMD. When the database kernels have limited DLP or no DLP, DySER speeds up by 50%, whereas SIMD fails. Combining multiple database kernels to exploit pipeline parallelism also improves performance, but requires careful looping strategies to utilize the DySER efficiently.

5.8 Chapter Summary

In this chapter, we first presented a detailed workload characterization to demonstrate that the DySER compiler can specialize programs written with traditional programming languages. We then described the performance and energy evaluation of the DySER implementation and showed that the

DySER provides significant improvements for data parallel workloads and modest improvements for control intensive workloads. We show that the compiler generated code performs as well as the manually optimized code. We also demonstrate that vectorization using the AEPDG and the DySER execution model perform better than the state of art auto vectorization in ICC. Sensitivity studies show that as the width of the processor increases, the performance gain from DySER decreases, but energy reduction due to DySER increases. Finally, we evaluated DySER on database primitives and demonstrated that DySER can specialize a real world application where both data parallel and control intensive kernels work together.

6 Related Work

Using specialized architectural designs to improve performance and energy has been an active area of research for decades. Section 6.1 describes the related work in architecture designs that use specialization to accelerate or to achieve energy efficiency. It also discusses compilers for these architectures, if any. The DySER architecture requires advanced support from its compiler in order to exploit the resources available. Section 6.2 discusses the related work on the compilation techniques used in the DySER compiler.

6.1 Specialized Architectures

One of the closest works to DySER from the classical era of supercomputing is the Burroughs Scientific Processor (BSP) [73]. BSP uses highly specialized arithmetic elements that are fully pipelined to accelerate vectorized FORTRAN code. The evolution of three important insights from BSP leads to the DySER architecture. First, to achieve generality, both BSP and DySER utilize compiler support to generate configurations that map computations to an execution substrate. DySER further expands the flexibility and efficiency by introducing a circuit-switched network in the execution substrate. This improvement needs several additional supporting mechanisms in the architecture such as flow control and reconfigurability. Second, both BSP and DySER identify the critical role of intermediate value storage for performance and efficiency. The arithmetic elements in the BSP have dedicated register files which are not part of the architectural state. Unlike this "centralized" design, which is not energy efficient, DySER provides distributed storage in its network using pipeline registers. Third, to generate useful code, the BSP compiler maps vectorized FORTRAN code to a set of prebuilt templates called *vector forms*, which in turn have

efficient mapping to the pipelined arithmetic elements. In contrast, DySER uses a co-designed compiler that can identify arbitrary code-regions and map them to DySER's hardware substrate. A final difference is in the implementation. While the BSP spends much effort on building a fast storage system (register, I/O, special memory), DySER uses a conventional core for efficient data management to achieve the same goal.

We broadly classify the related work from the recent literature into four categories: application specific accelerators, coarse grain reconfigurable accelerators, tiled architectures, and data parallel architectures. Table 6.1 lists the related works and their characteristics compared to DySER. Column 6 of Table 6.1 lists the characteristics of DySER. The design of DySER achieves three goals simultaneously: generality in software, low design complexity in hardware, and efficiency. Basically, it strives to provide efficiency on diverse sets of workloads written in the traditional programming model without increasing design complexity.

Application Specific Accelerators: Application specific accelerators are custom hardware units that speed up or improve energy efficiency for a specific application. They are usually implemented as an Application Specific Integrated Circuit (ASIC) and integrated to a core as an intellectual property (IP). Column 2 of Table 6.1 lists the characteristics of these accelerators.

Architects have long used application specific accelerators for energy efficiency in the embedded space. Examples include accelerated signal processing [81], cryptography [129], video encoding [53], and lossless image compression [97]. Since ASICs implement the algorithm in hardware directly, they provide more than an order of magnitude in energy efficiency. However, they are application specific and are not flexible. i.e., these accelerators usually cannot be used to speed up another application. In addition to not being flexible, their integration with a processor core increases design complexity because they usually need dedicated resources and glue logic. In embedded processing, to achieve flexibility and efficiency close to an ASIC, we use application specific instruction set processors (ASIP). The instruction set of these processors can be customized to benefit a specific application. For example, with the Xtensa processor [42] from Tensilica/Cadence and OpenCores [95], we can extend a baseline RISC processor by adding new instructions, registers, and datapath before synthesizing the processor. This provides flexibility during synthesis, but the

	Application Specific accelerators [81, 129, 53, 97, 42, 95]	CGRAs [31, 57, 41, 115, 130, 22, 25, 23, 85, 21, 87, 88, 27].	Tiled [122, 121, 16]	DLP [117, 43, 91, 72, 78, 108, 112, 101]	DySER [47, 48, 45, 46, 8, 7]
Software					
Generality	Application specific	Loop specific	General Purpose	Loop specific	General Purpose
Scope	Application	Inner Loop	Full	Kernels/ Loops	Code Regions
Flexibility	None	Limited	Yes	Limited	Yes
Hardware					
Overall Complexity	High	Medium	High	Low	Low
Integration	Dedicated	coprocessor/ incore	Dedicated	coprocessor/ incore	incore
Area	Large	Large/ Small	Large	Large/ Medium	Medium
Performance	High	Low	Medium	High/ Medium	Medium
Mechanisms					
ISA	New	Co-designed	New	New/ Extension	Extension
Compute Elements	Custom Logic	Functional Units	Cores, RF, buffers	SIMD Units	FU/Switches
Network	Custom	Custom	Packet Switch	Custom	Circuit Switch

Table 6.1: Related Work on Specialized Architecture

flexibility is lost after synthesis.

In contrast, DySER provides flexibility by dynamically creating the hardware datapath at runtime. With DySER, we can create custom instructions to adapt the datapath for applications that are running on the processor. Hence, DySER provides more flexibility than ASIPs and their custom instruction set. In addition, it can target general purpose workloads, whereas ASIPs cannot. In addition, DySER can be more easily integrated with existing general purpose cores than these custom ASIPs.

Recently, Conservation cores (C-Cores) [124] have been proposed to mitigate the effect of dark silicon by creating application specific hardware circuits for the most frequently executing functions to reduce the energy consumption. The functionality of stable applications, which do not change drastically, is extracted at the compiler IR level, and then a synthesis and mapping process statically specializes this functionality into multiple C-Cores, which are essentially application specific circuits, at system design time. To accommodate modifications through the lifetime of the chip, a patching mechanism is provided, which allows limited modification to the circuits to evolve the hardware with the software. Conservation cores use an exception handling mechanism to transfer control between the main processor and the conservation cores. By doing static domain-specific specialization, the energy efficiency is maximized. The energy efficiency benefit from conservation cores is close to the ASIC acceleration, but the performance gain over baseline inorder processors is very low, if any, because of the overhead of exception handling and lack of pipelining through the substrate. Also, this approach requires an independent co-processor for each application and cannot target diverse sets of workloads without being overly area inefficient. In contrast, DySER can target a diverse set of workloads automatically and create custom hardware dynamically to provide energy efficiency along with performance. Even though they are tightly integrated to the processor, C-Cores cannot execute computation speculatively and do not exploit fine grain data level parallelism. Since DySER is internally stateless, it can execute computation speculatively and exploit fine grain DLP by effectively emulating a SIMD unit with its vectorized instructions and many functional units.

Coarse Grain Reconfigurable Accelerators To achieve both full flexibility and efficiency close to ASICs, coarse grain reconfigurable accelerators have been proposed [56]. Coarse grain reconfigurable accelerators (CGRAs) consist of a sea of processing elements (PE), usually interconnected with a mesh network. They provide the ability to create specialized units. Unlike FPGAs, which allow reconfigurability at gate level, these accelerators allow reconfigurability at word level and are more area efficient than FPGAs. Column 3 of Table 6.1 lists the characteristics of these accelerators.

Coarse grain reconfigurable architectures are integrated to the core either as coprocessors or functional units. Coarse grain reconfigurable coprocessors from the past include RaPID [31], Garp [57], PipeRench [41], MorphoSys [115], Chimaera [130] and Tartan [87]. Coarse grain reconfigurable compound functional units include CCA [22, 25], SoftHV [27], and VEAL [23].

RaPID uses a linear array of functional units that can communicate with their nearest neighbors over a bus. It can be configured to form a linear computational pipeline that can perform repetitive computation efficiently. Programming for RaPID is done with an architecture specific language called Rapid B programming language which makes it easier to map applications to the RaPID substrate [32]. Since its substrate is a linear array, it cannot map computations with multiple flows. Unlike DySER, it uses a specialized language and hence cannot target legacy applications easily.

Garp uses an FPGA-like substrate to accelerate tight inner loops. However, it suffers when loop iterations are small because the cost of configuration for Garp is high. The Garp C compiler [17] takes unmodified C code and generates configurations for its substrate. However, the Garp architecture does not map control flow and its scope is limited to basic blocks. In contrast, the DySER compiler targets larger regions than basic blocks using the AEPDG to generate configurations for DySER. In addition, DySER's ability to quickly switch configurations allows it to provide efficiency even when the loop iterations are small.

PipeRench [41] is a domain specific reconfigurable coprocessor for stream based media applications. It uses a compiler for a domain specific language and maps the application to its substrate. With DySER, a variety of applications including media applications can be targeted for acceleration. The DySER compiler compiles directly from programs written in traditional languages like C/C++ to DySER.

Morphosys [115] is a coarse grain reconfigurable coprocessor with a Mips-like "TinyRISC"

processor with an extended instruction set for communicating to the coprocessor. It uses direct memory access (DMA) instructions in the host processor to communicate with the CGRA substrate. Programming for Morphosys is done through manual assembly programming.

The Tartan architecture [87, 88] and its compiler memory analysis explore spatial computing at a fine granularity with entire applications laid out on reconfigurable substrates on a co-processor. Procedure calls, library calls, and system calls require switching to the main processor and need extensive context saving. Also, the Tartan substrate supports multi level configurations which makes configuration switching a costly operation. DySER's specialized execution model is far less disruptive and achieves efficiency by dynamically reconfiguring the DySER datapaths as the application changes phases with fast configuration switching.

In summary, previously proposed coarse grain reconfigurable coprocessors provide specialized datapaths for efficiency, but it is difficult to program them, and they suffer from inefficiency due to the granularity of their specialization. In contrast, DySER integrates with the processor pipeline and exposes a set of flexible mechanisms that make it a good compiler target.

It has also been proposed to integrate coarse grain reconfigurable architectures with a processor as a compound functional unit, similar to DySER. Examples include CCA [22, 25], SoftHV [27] and VEAL [23]. Like DySER, these CGRAs integrate tightly with the processor pipeline. However, they do not support diverse application domains, and have memory limitations. For example, CCA [22, 25] uses a feed forward cross bar network connecting consecutive rows, which limits scalability and generality. With its circuit-switched interconnection network and decoupled access execute execution model, DySER can specialize a variety of code patterns with arbitrary memory access patterns. Unlike DySER, CCA lacks support for control flow.

SoftHV [27] has a co-designed virtual machine which reorders and fuses micro ops and executes the fused microops on accelerators to achieve energy efficiency over an inorder processor. It uses two accelerator units called ICALU and VLDU to accelerate simple arithmetic operations and consecutive memory operations respectively. Instead of fusing operations at run-time, which incurs additional overheads and design complexity, DySER exposes the flexible microarchitecture to the compiler and specializes over a larger region than SoftHV without additional overheads during runtime.

VEAL [23] tries to accelerate inner loops by designing loop accelerators with a compound functional unit (ex. CCA) to exploit data level parallelism. However, these loop accelerators do not support control flow. In contrast, DySER uses predication and ϕ-functions to support control flow inside the substrate and can map loops with internal control flow to DySER. VEAL is limited to innermost loops that must be modulo-schedulable, and does not allow the specialization region to extend across load/stores. Also, these accelerator implementations have limited size, supporting only a small number of functional units. For example, VEAL exploits the loop's modulo-schedulability with a novel design to generalize to multiple applications, but is limited to a small number of functional units: 2 INT, 2 FP and one compound unit. While VEAL focused on loop accelerators that can be used for multiple applications, Yehia et al. [131] and Fan et al. [36] seek to design loop accelerators with compound functional units to accelerate inner loops for an application. Unlike these approaches, DySER is more general and can used to accelerate code regions with unpredictable control flow and memory accesses.

With their configurable computation fabric, CGRAs can be used to create specialized datapaths for efficient computation, just like DySER. However, DySER's execution model, control flow support, fast reconfigurability and ability to exploit fine grain data level parallelism with vector instructions make it a light weight solution for achieving efficiency when integrated with a general purpose core.

Tiled Architectures The regular pattern of the mesh network and the array of functional units of the DySER hardware substrate resemble tiled architectures like RAW [122], Wavescalar [121], and TRIPS [16]. Column 4 of Table 6.1 lists the characteristics of these architectures. These architectures have been proposed to extend the von Neumann paradigm and try to replace non-scalable centralized structures in the traditional superscalar processors with regular scalable distributed structures.

Although DySER's hardware substrate resembles these architectures' tile based microarchitecture, DySER is a pure computation fabric and thus has no extra buffering, instruction storage, or data-memory storage in the tiles. In addition, DySER implements circuit-switched static routing of values, thus making the network far more energy efficient than dynamically arbitrated networks.

While these architectures distribute design complexity to different tiles, DySER does not add design complexity. The slicing of applications and the decoupled/access execution model require DySER only to perform computations, which simplifies the interface to the existing microarchitecture. Compared to these architectures, DySER achieves similar performance without massive ISA changes and at significantly smaller area and design complexity when compared to these full-chip solutions. However, it lacks their intellectual purity, since we relegate loads and stores to the processor pipeline.

Data Parallel Architectures In addition to being an efficient specialized architecture, DySER uses vector instructions to access multiple data elements from memory with a single instruction. Data parallel architectures use single instruction multiple data (SIMD) to exploit data level parallelism. Using its flexible hardware mechanism, DySER can emulate a SIMD unit and exploit available data level parallelism [45]. DLP architectures can be broadly classified into three broad categories: SIMD extensions, GPUs, and Other DLP architectures.

SIMD Extensions: Most modern processors include ISA extensions for vector operations like SSE/AVX, Altivec or Neon, which are designed to accelerate computations by exploiting data-level parallelism. The presence of control-flow prevent these architectures from exploiting fine grain data level parallelism because they lack masking registers and predicate registers. Also, strided data accesses forces these architectures to use scalar loads and stores which hurts efficiency. Several extensions have been proposed to mitigate these problems [117, 43]. In contrast, DySER natively maps the control flow instructions to the DySER hardware substrate using predication and ϕ-function operations, and handles the strided memory access through its flexible vector interface.

GPUs: Another approach is to use alternative architectures focused on data-level parallelism. GPUs [91] are the mainstream example. They use SIMT execution model to address some of the challenges of SIMD, providing significant performance through optimized data-parallel hardware and memory coalescing. The disadvantages are that programs in the traditional programming model have to be rewritten and optimized for a specific GPU architecture. From a hardware and system integration standpoint, the design integration of a GPU with a general purpose core is highly

disruptive, introduces design complexity, requires a new ISA, and adds the challenges associated with a new system software stack. However, the DySER compiler, described in this thesis, compiles programs written with the traditional programming model to automatically target DySER and achieve efficiency.

Other DLP Architectures: The Vector-Thread architecture is a research example that is even more flexible than the GPU approach, but it is difficult to program [72, 78]. Sankaralingam et al. develop a set of microarchitectural mechanisms designed for data-level parallelism, which are not inherently tied to any underlying architecture [108]. One of the recent DLP architectures is Intel's Xeon Phi, which accelerates data parallel workloads through wide SIMD and hardware support for scatter/gather [112]. In general, these DLP architectures do not perform well outside the data parallel domain and there are additional issues when integrating them with a processor core. In contrast, DySER can accelerate data parallel workloads and irregular workloads with control flow and unpredictable memory accesses. In addition, when the application has a mix of both data parallel and control intensive code regions, as in the case of databases, it is harder for these data parallel architectures to provide acceleration. However, as demonstrated in this thesis, DySER can dynamically create specialized hardware datapaths to achieve efficiency.

Similar to the DySER approach, which uses the flexible hardware substrate to exploit both data level parallelism and instruction level parallelism, the recently proposed LIBRA architecture also uses the principles of heterogeneity and dynamic reconfigurability to build a flexible accelerator [101]. It augments a SIMD architecture with a flexible network to improve the scope of SIMD acceleration. Though this approach shows promise, effective compilation techniques have not been fully explored.

Recent Proposals In the last few years, several works have been published on using specialized hardware to achieve energy efficiency or performance with an execution model similar to DySER. Examples include HARP [127], NPU [35], BERET [50], Convolution engine [104], Index Traversal Acceleration [71], Q100 [128], LEAP [55] and SGMF [125]. We describe the connections and influence of DySER in their principles.

HARP [127] seeks to improve the throughput and energy efficiency of large scale data partitioning, especially range partitioning, with a domain specific accelerator and stream buffers. Similar to DySER decoupled access/execute architecture, the HARP accelerator is decoupled from the rest of the microarchitecture with an input and an output stream buffer. Like DySER, ISA extensions are used to manage data transfers from memory to the stream buffers. The accelerator pulls its data from the input stream buffer and delivers its output to the output stream buffer. We can configure DySER to partition the data and use its flexible vector interface to achieve efficiency similar to HARP. However, HARP's dedicated data path to memory, dedicated stream buffers and dedicated hardware is more energy efficient than DySER's general purpose circuit switched interconnect.

NPU [35] proposes an accelerator, called the Neural Processing Unit, which accelerates applications with inexact computation. Many modern applications such as image rendering, signal processing, augmented reality, and data mining have approximatable computation, i.e., they can tolerate a certain degree of error in their outputs. The NPU approach exploits these characteristics by replacing a large code region with an invocation of neural network in the NPU. Similar to DySER invocation, the main processor communicates with the NPU through input and output FIFOs. Unlike DySER, which creates specialized data paths for the exact computation, NPU accelerates the learned model of the neural network with a specialized sigmoid functional unit and dedicated constant broadcast network. DySER can be adapted to accelerate the neural network model instead of the computation to mimic NPU. However, NPU's dedicated sigmoid functional unit and constant broadcast network provide more efficient support for computing the neural network than the resources available in DySER.

BERET [50] specializes only code-regions without any internal control-flow using its subgraph execution blocks (SEB), which are customizable cluster of functional units. Lack of divergent control-flow support limits the number of potential code-regions that can mapped to SEBs. DySER's ability to map control-flow natively helps more code regions to be accelerated with DySER. BERET is integrated with an inorder processor as a coprocessor and does not lend itself to integrate with an out-of-order processor, as it does not have mechanisms to rollback misspeculated computation. Also, implementing SEBs and integrating them to an existing microarchitecture pipeline is hard, since the BERET architecture allows memory operations to be performed from SEBs themselves. In

contrast, DySER decoupled access execute model makes it easier to integrate with an out-of-order processor.

The convolution engine [104] targets image processing kernels and stencil computations by exploiting the key data flow patterns in the kernels. It uses custom load/store units, custom shift registers, map and reduce logic, a complex graph fusion unit, and custom SIMD registers to accelerate convolution and other filter kernels. The programming for the convolution engine is done through compiler specific intrinsics unlike DySER. Since convolution type kernels have more fine grain data level parallelism, we can specialize these kernels with DySER and use its vectorized instruction to achieve high throughput and efficiency.

The work in Meet the Walkers [71] presents an on-chip accelerator called Widx, for indexing operations in big data analytics. Widx uses a set of programmable hardware units to achieve high performance by accessing multiple hash buckets concurrently and hashing input keys in advance and hence removing hashing from the critical path. Widx itself is implemented with a custom RISC processor that supports fused instructions to accelerate hash functions. The accelerator is programmed with a limited subset of C, without any dynamic memory allocation, no stack and with one output. DySER can specialize the indexing operations using its substrate. However, the "Walkers" architecture achieves high throughout by decoupling hashing and hash table walking with a dedicated buffer. Without this dedicated buffer, the DySER architecture stores and loads from memory and consumes memory bandwidth, which may lead to loss of efficiency. As with other domain specific accelerators, the applicability of Walkers outside its chosen domain is limited. In contrast, DySER accelerates a variety of workloads.

The Q100 [128] architecture accelerates database processing tasks with a collection of heterogeneous ASIC tiles that can efficiently perform database primitives like sort, scan etc., As we described in Section 5.7, DySER can specialize database primitives and achieve significant energy efficiency. Compared to Q100, which needs separate ASIC tiles for each primitive, DySER can dynamically specialize for each primitive and hence be more area efficient. However, for each specific primitive, Q100 is more energy efficient than DySER because DySER uses its general purpose circuit-switched network to route data.

Recently, to eliminate the artificial inefficiencies due to program counter sequencing of ALU

operations, Triggered Instructions [98] has been proposed. It presents a novel execution model which performs computation and transitions to different states without explicit branching instructions, but with rules or conditions. However, programming the triggered instruction architecture is difficult. It requires significant compiler support, and a new programming model and new algorithms. However, using the DySER compiler, we can target programs written in the traditional programming model.

6.2 Compilation Techniques

In order to achieve high efficiency with coarse grain reconfigurable architectures, a good compiler is essential to manage and exploit the available heterogeneous computing resources available. In addition to being a CGRA compiler, the DySER compiler also borrows vectorization techniques to generate DySER vector instructions to exploit fine grain data level parallelism. The DySER compiler represents the candidate code regions with the Access/Execute Program Dependence graph for optimization. Below, we present related work on the techniques that are used to construct the AEPDG and optimizations that are performed on the AEPDG.

Identifying Code-Regions: In order to compile for DySER, the DySER compiler first identifies code regions without loops or back edges as a candidate region and construct the AEPDG for that region. There are many alternative methods to identify acyclic code regions: Basicblocks [3], Traces [33], Superblocks [67], Hyperblocks [84] , Treegions [58] and Pathtrees [47]. Basic blocks are straight line block of code without any control flow. Traces are linear paths through the code, but they can have multiple entrances and exits. Superblocks are traces with single entrances, i.e. they are single entry multiple exit regions. However, they cannot have any form of control flow. Hyperblocks are single-entry, multiple exit code regions that can have internal control flow. Treegions are the tree of basic block within the control flow graph(CFG). Pathtrees are set of basic blocks identified with Path Profiling to represent an application phase. The DySER compiler uses Hyperblocks to identify the acyclic code regions to specialize when there is no dynamic profiling is available. When dynamic profiling is available, it identifies acyclic code regions that are similar to Pathtrees. It then

uses the AEPDG to represent the programs to generate optimized code for DySER.

Region Optimizations: Depending upon the methods to identify code regions, compilers use several techniques to alter the region size that can be fit into DySER. The oldest and simplest region enlargement technique is loop unrolling. Other techniques such as Tail duplication, loop peeling are also used. Subregion identification is a common technique to improve accelerator efficiency [24, 100]. These approaches either require modulo-schedulable loops or proportional sized loops. In addition to subregion identification with subgraph matching, the DySER compiler uses subregion splitting on the AEPDG and utilizes hardware support, especially fast configuration switching, to achieve efficiency.

Scheduling: The DySER compiler statically schedules the instructions in the execute-PDG to the DySER substrate. Previously, static scheduling has been explored in VLIW processors [33] and RAW [77]. However, it works on the fine granularity of a few instructions within a basic block or trace. The DySER compiler schedules tens of instructions in the execute-PDG to DySER and also does the explicit routing of data between the instructions.

Traditional instruction schedulers are not suitable for scheduling operations to coarse grain accelerators because they do not take into account the explicit routing of operands between the operations. Mei et al. use simulated annealing techniques to schedule a loop body to a reconfigurable substrate [86]. They start with a random placement of operations to the substrate and then operations are randomly moved between functional units until a valid schedule is achieved. Although this may succeed, in practice, the simulated annealing algorithm takes a long time to converge for loops with a large number of operations. The DySER compiler takes a simple approach to scheduling operations to DySER. It uses the list based scheduling algorithm, which either succeeds or fails. When it fails, it spills the operations to the access-PDG and continues. Although this may result in a suboptimal schedule, in practice, it maps large regions to DySER quickly and generates good quality schedules. Another approach is to use modulo graph embedding [100]. However, this requires the region to be a modulo schedulable loop.

Recently, Nowatzki et al. [93, 92] described a general scheduling framework for spatial archi-

tectures using integer linear programming which can target a variety of specialized architectures including DySER. Also, they discussed the vast literature from this area.

The mapping of the execute-PDG to DySER is related to VLSI place-and-route problems [83]. However, the DySER scheduler is significantly smaller scale than VLSI problems and the placement in the case of DySER is fixed.

Intermediate Representation for CGRAs: There have been several works on compilers for coarse grain reconfigurable architectures that are related to the work presented in this thesis. Past studies and efforts at compilation tools for coarse grain reconfigurable architectures have focused mainly on exploiting instruction level parallelism (ILP) [17, 77] and modulo schedulable loops [86, 100]. These works uses data flow graph (DFG) as their intermediate representation to capture the execution through their reconfigurable substrate. Since they usually do not support control flow inside their substrate, this is sufficient. Also, the specializable regions are usually selected separately and the scheduler maps the computation to the substrate. In contrast, the DySER compiler captures the spatio-temporal properties of the computation with the AEPDG and uses the AEPDG itself to select code regions for specialization.

CGRA Compiler Implementations: There have been compilers implemented for coarse grain reconfigurable accelerators. One example CGRA compiler is the Garp C compiler. The Garp C compiler [17] takes unmodified C code and generates the configuration for its substrate. However, the Garp architecture does not map control flow and its scope is limited to basic blocks. PipeRench [41], a domain specific reconfigurable architecture for stream based media applications, uses a compiler for a domain specific language to achieve efficiency. In contrast, the DySER compiler identifies code-regions across basic blocks and compiles programs written with the traditional programming model.

Vectorizing Compilation Techniques: Autovectorizing compilers have been around for decades, beginning with perhaps the High Performance Fortran (HPF) compilers of the 1980s. Many of the standard loop-vectorization techniques that we utilize were already well known in the mid 1980s [96]. To get high performance from data parallel code with control flow and unpredictable

memory access patterns, compiler writers have been working around the limitations imposed by SIMD's rigid model of homogeneous execution in various ways [96, 11]. We discuss how vectorizing compilers deal with control flow, strided memory access and loop carried dependences.

To cope with control-flow, vectorizing compilers usually first convert control dependence to data dependence through if conversion [4] and then use standard vectorizing techniques. However, when nested conditions are present, these technique usually do not work as the number of predication required grow exponentially. Instead the DySER compiler exploits the predication and ϕ-function support in DySER to map the if-converted control dependences easily to DySER. Recently, Shin et al. [114] describes how to generate efficient vector codes in the presence of control with branch-on-superword-condition-code instructions in the ISA. It uses a novel reverse-implies graph and uses a heavy weight analysis to generate the code. In contrast, the DySER compiler uses a simplified analysis and maps control flow natively to DySER.

To overcome interleaved or strided data access, Nuzman et al. describe techniques, using common vector primitives, to perform these access patterns at fixed stride lengths [94]. Gang et al. provided a more universal methodology for optimizing data permutations [107]. These techniques will necessarily incur overheads that the DySER compiler and DySER hardware mechanisms that provide a flexible vector interface seek to avoid.

For handling loop-carried dependence and partially vectorizable loops, one solution has been to use loop-fission [69] to break the vectorizable portion and non-vectorizable portion of the loops. However, loop-fission may not be possible for all cases and it may be less cache friendly. A general solution to handle loop carried dependence is to use polyhedral models [49, 37]. A polyhedral model is a mathematical framework to perform loop nest transformations to eliminate loop carried dependences, if possible. It maps the iterations of the loops to lattice and performs affinity transformations to eliminate loop carried dependences.

Other Vectorizing Compiler Approaches: The ispc compiler tries to solve the challenges with SIMD by adopting a new language semantics and trying to overcome compiler problems [102], whereas the DySER approach operates on C/C++ source code and makes the architecture more flexible. Intel Xeon Phi [112], a recent SIMD architecture, and its compiler help programmers

tackle the challenges of SIMD through algorithmic changes such as struct-of-arrays to array-of-structs, blocking, and SIMD friendly algorithms, compiler transformations such as parallelization, vectorization, and with scatter/gather hardware support [111]. However, to successfully use them, these changes require heavy programmer intervention and application specific knowledge.

7 Conclusion

Computer architects are increasingly using coarse grain reconfigurable architectures for specializing general purpose workloads to gain performance and energy efficiency in the post Dennard scaling era. This thesis has presented a novel design with a hardware/software solution, spanning the microarchitecture, compiler and application, that avoids disruptive changes to the existing hardware/software stack while achieving energy efficiency. It presented a practical way to use coarse grain reconfigurable accelerators automatically for diverse sets of workloads written in traditional programming languages such as C/C++.

7.1 Summary of Contributions

The key contributions of this thesis are:

DySER Microarchitecture: It presented a hardware codesigned approach to achieve energy efficient computing with a dynamically specialized architecture. Specifically, it described the microarchitecture of DySER: Dynamically Specialized Executed Resources. DySER can dynamically create specialized datapaths for arbitrary sequences of computation with its heterogeneous array of functional units and circuit switched network. It described hardware mechanisms to map control flow to the DySER hardware substrate. It also described the flexible vector interface that DySER uses to exploit fine grain parallelism.

DySER Compiler: It presents a source to binary compiler toolchain for automatic specialization of programs written with the traditional programming model in C/C++. It developed a novel intermediate representation called the Access Execute Program Dependence Graph (AEPDG), a variant

of the Program Dependence Graph, to capture the temporal and spatial nature of computation. It also presented compiler optimizations and transformations using the AEPDG, to produce high quality code for dynamically specialized architectures.

Application: It demonstrated the ease and utility of a codesigned architecture-compiler approach to accelerate a variety of workloads automatically by doing evaluation studies on data parallel workloads, on general purpose workloads and on emerging workloads. It also presented quantitative results on a highly relevant application, database query processing, which shows that DySER can provide efficiency even when the application has a mix of data parallel and irregular code patterns.

7.2 Closing Remarks

We have shown how the DySER architecture and its compiler achieve high performance and energy efficiency with diverse sets of workloads. It unifies disparate attempts on specialized architectures for functionality like encryption accelerators in the Niagara processor [113], and SIMD accelerators to exploit fine grain data level parallelism like SSE/AVX. Our quantitative results show DySER is competitive with or outperforms SIMD accelerators, and provides energy efficiency with irregular code. Also, it is a feasible design, easily integrable with a processor. The evaluation results of DySER, a coarse grain reconfigurable accelerator, on data parallel workloads have implications for future accelerators, especially SIMD. Below, we elaborate on these implications.

Programming tradeoffs: SIMD accelerators or short-vector extensions can provide speedup, but compilers have difficulty targeting SIMD well. Programmers typically must use compiler intrinsics, which creates severe portability and maintainability problems. Although there have been successful GPGPU programming languages like CUDA, GPUs pose their own set of programming challenges. Not only must the user learn a new language, they must learn the massively multi-threaded thinking paradigm, give up on familiar sequential program debugging, and apply GPU specific optimizations. DySER programming is relatively simple, uses sequential C++ code, and uses established debugging methodologies.

SIMD Evolution: Even though the SSE family is SIMD, many extensions to SSE (SSE3 and later) have instructions that are not purely word parallel. For example, the instruction `HADDPD` and its variants operate on elements from the same vector. Also, there are instructions with specialized functionality like `MPSADBW`, which computes the sum of absolute differences. This exemplifies a trend towards providing functionality specialization in data parallel accelerators. SIMD evolution, by increasing width, does not provide scalable performance benefits across workloads, whereas DySER scalably adapts. Hence, we feel DySER is the natural evolution of these instructions sets.

GPU Evolution: Conversely, GPUs are leaning toward the CPU side by providing caches and eliminating redundant work with their scalarization approach which effectively creates a "control" core and a set of compute-cores, much like DySER's organization. Again, we feel DySER-like integration is the direction GPUs are headed. Inspired by DySER, researchers are already proposing alternative designs for GPGPUs that uses DySER like specialization hardware to be more energy efficient [125].

Replacing SIMD or GPU: In summary, we feel DySER is a viable candidate for replacing SIMD short vector instruction sets. With some simple extensions, DySER can be augmented to emulate existing instruction sets like SSE, thus providing backward compatibility. Clearly, DySER is not a GPU replacement, since it cannot perform graphics tasks well. It is a promising alternative for "design-constrained" environments like Tilera, and ARM in servers to target high-performance computing. In these cases, a completely new processor design like a GPU, or integration of a GPU with a core, and adoption of a new software ecosystem may be prohibitively complex. In contrast, DySER's hardware and software ecosystem are non-disruptive.

Broadly, DySER's unifying data parallel mechanisms with a flexible substrate that can natively map control flow provides a platform for energy efficient computing.

Bibliography

[1] The International Technology Roadmap for Semiconductors (ITRS), System Drivers, 2011, http://www.itrs.net/. 2011.

[2] 21st century computer architecture: A community white paper, http://cra.org/ccc/docs/init/21stcenturyarchitecturewhitepaper.pdf.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[4] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *POPL '83*, 1983.

[5] T. Ball and J. R. Larus. Efficient Path Profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29, pages 46–57, Washington, DC, USA, 1996. IEEE Computer Society.

[6] L. A. Barroso and U. Hölzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, Dec. 2007.

[7] J. Benson, R. Cofell, C. Frericks, V. Govindaraju, C.-H. Ho, Z. Marzec, T. Nowatzki, and K. Sankaralingam. Prototyping the DySER Specialization Architecture with OpenSPARC. In *Hot Chips 24*, August 2012.

[8] J. Benson, R. Cofell, C. Frericks, C.-H. Ho, V. Govindaraju, T. Nowatzki, and K. Sankaralingam. Design Integration and Implementation of the DySER Hardware Accelerator into OpenSPARC. In *Proceedings of 18th International Conference on High Performance Computer Architecture (HPCA)*, 2012.

[9] M. Bhadauria, V. M. Weaver, and S. A. McKee. Understanding PARSEC performance on contemporary CMPs. In *IISWC, 2009*, pages 98–107, Austin, TX.

[10] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM.

[11] A. J. C. Bik. *Software Vectorization Handbook, The: Applying Intel Multimedia Extensions for Maximum Performance*. Intel Press, 2004.

[12] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, Aug. 2011.

[13] M. Bohr. A 30 Year Retrospective on Dennard's MOSFET Scaling Paper. *Solid-State Circuits Newsletter, IEEE*, 12(1):36 –37, winter 2007.

[14] P. Boncz, M. Zukowski, and N. Nes. MonedDB/X100: Hyper-Pipelining Query Execution. In *Proceedings of the 2005 CIDR Conference*, 2005.

[15] S. Borkar and A. A. Chien. The future of microprocessors. *Commun. ACM*, 54(5):67–77, May 2011.

[16] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, W. Yoder, and t. T. Team. Scaling to the End of Silicon with EDGE Architectures. *Computer*, 37(7):44–55, July 2004.

[17] T. Callahan, J. Hauser, and J. Wawrzynek. The Garp architecture and C compiler. *Computer*, 33(4):62–69, Apr 2000.

[18] K. Chakraborty. *Over-provisioned Multicore Systems*. PhD thesis, Department of Computer Sciences, Madison, WI, USA, 2008.

[19] K. Chakraborty, P. M. Wells, G. S. Sohi, and K. Chakraborty. A case for an over-provisioned multicore system: Energy efficient processing of multithreaded programs. Technical report, Department of Computer Sciences, University of Wisconsin-Madison, 2007.

[20] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey. Efficient implementation of sorting on multi-core SIMD CPU architecture. *Proc. VLDB Endow.*, 1(2):1313–1324, Aug. 2008.

[21] S. Ciricescu, R. Essick, B. Lucas, P. May, K. Moat, J. Norris, M. Schuette, and A. Saidi. The Reconfigurable Streaming Vector Processor (RSVPTM). In *MICRO 36*, page 141, 2003.

[22] N. Clark, J. Blome, M. Chu, S. Mahlke, S. Biles, and K. Flautner. An Architecture Framework for Transparent Instruction Set Customization in Embedded Processors. In *Proceedings of the 32nd annual international symposium on Computer Architecture*, ISCA '05, pages 272–283, Washington, DC, USA, 2005. IEEE Computer Society.

[23] N. Clark, A. Hormati, and S. Mahlke. VEAL: Virtualized Execution Accelerator for Loops. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 389–400, Washington, DC, USA, 2008. IEEE Computer Society.

[24] N. Clark, A. Hormati, S. Mahlke, and S. Yehia. Scalable subgraph mapping for acyclic computation accelerators. In *CASES '06*, 2006.

[25] N. Clark, M. Kudlur, H. Park, S. Mahlke, and K. Flautner. Application-Specific Processing on a General-Purpose Core via Transparent Instruction Set Customization. In *MICRO 37*, pages 30–40, 2004.

[26] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, Oct. 1991.

[27] A. Deb, J. M. Codina, and A. González. SoftHV: A HW/SW Co-designed Processor with Horizontal and Vertical Fusion. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, pages 1:1–1:10, New York, NY, USA, 2011. ACM.

[28] R. H. Dennard, F. H. Gaensslen, H. N. Yu, V. L. Rideout, E. Bassous, and A. LeBlanc. Ion implanted MOSFET's with very short channel lengths. *Solid-State Circuits Newsletter, IEEE*, 12(1):36 –37, winter 2007.

[29] D. Donofrio, L. Oliker, J. Shalf, M. F. Wehner, C. Rowen, J. Krueger, S. Kamil, and M. Mohiyuddin. Energy-efficient computing for extreme-scale science. *Computer*, 42(11):62–71, Nov. 2009.

[30] Slicer - Compiler for DySER. http://research.cs.wisc.edu/veritcal/dyser-compiler.

[31] C. Ebeling, D. C. Cronquist, and P. Franklin. RaPiD - Reconfigurable Pipelined Datapath. In *Proceedings of the 6th International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*, FPL '96, pages 126–135, London, UK, UK, 1996. Springer-Verlag.

[32] C. Ebeling, D. C. Cronquist, P. Franklin, J. Secosky, and S. G. Berg. Mapping Applications to the RaPiD Configurable Architecture. In *Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines*, FCCM '97, pages 106–, Washington, DC, USA, 1997. IEEE Computer Society.

[33] J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures (Parallel Computing, Reduced-instruction-set, Trace Scheduling, Scientific)*. PhD thesis, New Haven, CT, USA, 1985. AAI8600982.

[34] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM.

[35] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural Acceleration for General-Purpose Approximate Programs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 449–460, Washington, DC, USA, 2012. IEEE Computer Society.

[36] K. Fan, M. Kudlur, H. Park, and S. Mahlke. Increasing Hardware Efficiency with Multifunction Loop Accelerators. In *Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '06, pages 276–281, New York, NY, USA, 2006. ACM.

[37] P. Feautrier. Automatic Parallelization in the Polytope Model. In *The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications*, pages 79–103, London, UK, UK, 1996. Springer-Verlag.

[38] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[39] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. Comput.*, 30(7):478–490, July 1981.

[40] S. H. Fuller and E. C. o. S. G. i. C. P. N. R. C. Lynette I. Millett. *The Future of Computing Performance: Game Over or Next Level?* The National Academies Press, 2011.

[41] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. R. Taylor. PipeRench: A Reconfigurable Architecture and Compiler. *Computer*, 33(4):70–77, Apr. 2000.

[42] R. E. Gonzalez. Xtensa: A Configurable and Extensible Processor. *IEEE Micro*, 20(2):60–70, Mar. 2000.

[43] C. Gou, G. Kuzmanov, and G. N. Gaydadjiev. SAMS Multi-layout Memory: Providing Multiple Views of Data to Boost SIMD Performance. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 179–188, New York, NY, USA, 2010. ACM.

[44] N. Goulding-Hotta, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, P.-C. Huang, M. Arora, S. Nath, V. Bhatt, J. Babb, S. Swanson, and M. Taylor. The GreenDroid Mobile Application Processor: An Architecture for Silicon's Dark Future. *IEEE Micro*, 31(2):86–95, Mar. 2011.

[45] V. Govindaraju, 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, Sept. 2012.

[46] V. Govindaraju, C.-H. Ho, T. Nowatzki, and K. Sankaralingam. Mechanisms for Parallelism Specialization for the DySER Architecture. Technical Report TR-1773, UW-Madison, June 2012.

[47] V. Govindaraju, C.-H. Ho, and K. Sankaralingam. Dynamically Specialized Datapaths for Energy Efficient Computing. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, pages 503–514, Washington, DC, USA, 2011. IEEE Computer Society.

[48] V. Govindaraju, T. Nowatzki, and K. Sankaralingam. Breaking SIMD Shackles with an Exposed Flexible Microarchitecture and the Access Execute PDG. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 341–352, Piscataway, NJ, USA, 2013. IEEE Press.

[49] M. Griebl, C. Lengauer, and S. Wetzel. Code Generation in the Polytope Model. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, PACT '98, pages 106–, Washington, DC, USA, 1998. IEEE Computer Society.

[50] S. Gupta, S. Feng, A. Ansari, S. Mahlke, and D. August. Bundled execution of recurring traces for energy-efficient general purpose processing. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, pages 12–23, New York, NY, USA, 2011. ACM.

[51] T. R. Halfhill. Ambric'S New Parallel Processor - Globally Asynchronous Architecture Eases Parallel Programming. *Microprocessor Report*, October 2006.

[52] T. R. Halfill. MathStar Challenges FPGAs. *Microprocessor Report*, 20(7):29–35, July 2006.

[53] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding sources of inefficiency in general-purpose chips. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 37–47, New York, NY, USA, 2010. ACM.

[54] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Toward Dark Silicon in Servers. *IEEE Micro*, 31(4):6 – 15, 2011.

[55] E. Harris, S. Wasmundt, L. D. Carli, K. Sankaralingam, and C. Estan. LEAP: Latency- Energy- and Area-optimized Lookup Pipeline. In *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, October 2012.

[56] R. Hartenstein. Coarse grain reconfigurable architecture (embedded tutorial). In *Proceedings of the 2001 Asia and South Pacific Design Automation Conference*, ASP-DAC '01, pages 564–570, New York, NY, USA, 2001. ACM.

[57] J. R. Hauser and J. Wawrzynek. Garp: a MIPS processor with a reconfigurable coprocessor. In *Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines*, FCCM '97, pages 12–, Washington, DC, USA, 1997. IEEE Computer Society.

[58] W. Havanki, S. Banerjia, and T. Conte. Treegion scheduling for wide issue processors. In *High-Performance Computer Architecture, 1998. Proceedings., 1998 Fourth International Symposium on*, pages 266–276, Feb 1998.

[59] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[60] J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, Sept. 2006.

[61] C.-H. Ho. *Mechanisms towards energy-efficient dynamic hardware specialization*. PhD thesis.

[62] C.-H. Ho, V. Govindaraju, T. Nowatzki, Z. Marzec, R. Nagaraju, P. Agarwal, C. Frericks, R. Cofell, and K. Sankaralingam. Performance Evaluation of a DySER FPGA Prototype System Spanning the Compiler, Microarchitecture, and Hardware Implementation. In *Submission*.

[63] C.-H. Ho, S. J. Kim, and K. Sankaralingam. Memory access dataflow. In *Submission*.

[64] H. P. Hofstee. Power Efficient Processor Architecture and The Cell Processor. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, HPCA '05, pages 258–262, Washington, DC, USA, 2005. IEEE Computer Society.

[65] S. Huang, S. Xiao, and W. Feng. On the energy efficiency of graphics processing units for scientific computing. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–8, Washington, DC, USA, 2009. IEEE Computer Society.

[66] I. Hur and C. Lin. Memory prefetching using adaptive stream detection. In *MICRO 39*, pages 397–408.

[67] W.-M. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: an effective technique for VLIW and superscalar compilation. *J. Supercomput.*, 7(1-2):229–248, May 1993.

[68] A. Kejariwal, A. V. Veidenbaum, A. Nicolau, X. Tian, M. Girkar, H. Saito, and U. Banerjee. Comparative architectural characterization of SPEC CPU2000 and CPU2006 benchmarks on the intel core2 duo processor. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 132–141.

[69] K. Kennedy and J. R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[70] D. Kim and D. Yeung. Design and evaluation of compiler algorithms for pre-execution. *SIGPLAN Not.*, 37(10):159–170, 2002.

[71] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan. Meet the Walkers: Accelerating Index Traversals for In-memory Databases. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 468–479, New York, NY, USA, 2013. ACM.

[72] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic. The Vector-Thread Architecture. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA '04, pages 52–, Washington, DC, USA, 2004. IEEE Computer Society.

[73] D. J. Kuck and R. A. Stokes. The Burroughs Scientific Processor (BSP). *IEEE Trans. Comput.*, 31(5):363–376, May 1982.

[74] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 81–, Washington, DC, USA, 2003. IEEE Computer Society.

[75] R. Kumar, D. M. Tullsen, and N. P. Jouppi. Core architecture optimization for heterogeneous chip multiprocessors. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, PACT '06, pages 23–32, New York, NY, USA, 2006. ACM.

[76] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[77] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. Amarasinghe. Space-time Scheduling of Instruction-level Parallelism on a RAW Machine. *SIGPLAN Not.*, 33(11):46–57, Oct. 1998.

[78] Y. Lee, R. Avizienis, A. Bishara, R. Xia, D. Lockhart, C. Batten, and K. Asanović. Exploring the Tradeoffs Between Programmability and Efficiency in Data-parallel Accelerators. *SIGARCH Comput. Archit. News*, 39(3):129–140, June 2011.

[79] J. Leverich, H. Arakida, A. Solomatnikov, A. Firoozshahian, M. Horowitz, and C. Kozyrakis. Comparing memory systems for chip multiprocessors. In *ISCA '07*, pages 358–368.

[80] S. Li, 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, pages 469–480, New York, NY, USA, 2009. ACM.

[81] Y. Lin, H. Lee, M. Woh, Y. Harel, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner. SODA: A Low-power Architecture For Software Radio. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ISCA '06, pages 89–101, Washington, DC, USA, 2006. IEEE Computer Society.

[82] Q. Liu and W. Luk. Heterogeneous systems for energy efficient scientific computing. In *Proceedings of the 8th international conference on Reconfigurable Computing: architectures, tools and applications*, ARC'12, pages 64–75, Berlin, Heidelberg, 2012. Springer-Verlag.

[83] J. Luu, I. Kuon, P. Jamieson, T. Campbell, A. Ye, W. M. Fang, and J. Rose. VPR 5.0: FPGA Cad and Architecture Exploration Tools with Single-driver Routing, Heterogeneity and Process Scaling. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '09, pages 133–142, New York, NY, USA, 2009. ACM.

[84] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th annual international symposium on Microarchitecture*, MICRO 25, pages 45–54, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[85] B. Mathew and A. Davis. A loop accelerator for low power embedded VLIW processors. In *CODES+ISSS '04*, pages 6–11.

[86] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. *Computers and Digital Techniques, IEE Proceedings -*, 150(5):255–61–, Sept 2003.

[87] M. Mishra, T. J. Callahan, T. Chelcea, G. Venkataramani, S. C. Goldstein, and M. Budiu. Tartan: evaluating spatial computation for whole program execution. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 163–174, New York, NY, USA, 2006. ACM.

[88] M. Mishra and S. Goldstein. Virtualization on the Tartan Reconfigurable Architecture. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 323–330, Aug 2007.

[89] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[90] T. Mudge. Power: a first-class architectural design constraint. *Computer*, 34(4):52 –58, Apr 2001.

[91] J. Nickolls and W. Dally. The GPU Computing Era. *Micro, IEEE*, 30(2):56–69, March 2010.

[92] T. Nowatzki, M. Ferris, K. Sankaralingam, C. Estan, N. Vaish, and D. A. Wood. Optimization and Mathematical Modeling in Computer Architecture. Synthesis Lectures on Computer Architecture, September 2013.

[93] T. Nowatzki, M. Sartin-Tarm, L. De Carli, K. Sankaralingam, C. Estan, and B. Robatmili. A General Constraint-centric Scheduling Framework for Spatial Architectures. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 495–506, New York, NY, USA, 2013. ACM.

[94] D. Nuzman, I. Rosen, and A. Zaks. Auto-vectorization of Interleaved Data for SIMD. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 132–143, New York, NY, USA, 2006. ACM.

[95] Opencores project home. http://opencores.org/.

[96] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Commun. ACM*, 1986.

[97] M. Papadonikolakis, V. Pantazis, and A. P. Kakarountas. Efficient High-performance ASIC Implementation of JPEG-LS Encoder. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '07, pages 159–164, San Jose, CA, USA, 2007. EDA Consortium.

[98] A. Parashar, M. Pellauer, M. Adler, B. Ahsan, N. Crago, D. Lustig, V. Pavlov, A. Zhai, M. Gambhir, A. Jaleel, R. Allmon, R. Rayess, S. Maresh, and J. Emer. Triggered Instructions: A Control Paradigm for Spatially-programmed Architectures. *SIGARCH Comput. Archit. News*, 41(3):142–153, June 2013.

[99] Parboil Benchmark suite, http://impact.crhc.illinois.edu/parboil.php.

[100] H. Park, K. Fan, M. Kudlur, and S. Mahlke. Modulo Graph Embedding: Mapping Applications Onto Coarse-grained Reconfigurable Architectures. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '06, pages 136–146, New York, NY, USA, 2006. ACM.

[101] Y. Park, J. J. K. Park, H. Park, and S. Mahlke. Libra: Tailoring SIMD Execution using Heterogeneous Hardware and Dynamic Configurability. In *MICRO '12*, 2012.

[102] M. Pharr and W. R. Mark. ispc: A SPMD Compiler for High-Performance CPU Programming. In *InPar 2012*, 2012.

[103] M. Poess and C. Floyd. New TPC Benchmarks for Decision Support and Web Commerce. *SIGMOD Rec.*, 29(4):64–71, Dec. 2000.

[104] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz. Convolution Engine: Balancing Efficiency & Flexibility in Specialized Computing. *SIGARCH Comput. Archit. News*, 41(3):24–35, June 2013.

[105] A. Raghavan, Y. Luo, A. Chandawalla, M. Papaefthymiou, K. P. Pipe, T. F. Wenisch, and M. M. K. Martin. Computational sprinting. *High-Performance Computer Architecture, International Symposium on*, 0:1–12, 2012.

[106] P. Ranganathan. Recipe for efficiency: principles of power-aware computing. *Commun. ACM*, 53(4):60–67, Apr. 2010.

[107] G. Ren, P. Wu, and D. Padua. Optimizing data permutations for SIMD devices. In *PLDI '06*, 2006.

[108] K. Sankaralingam, 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, pages 303–, Washington, DC, USA, 2003. IEEE Computer Society.

[109] M. Sartin-Tarm, T. Nowatzki, L. De Carli, K. Sankaralingam, and C. Estan. Constraint centric scheduling guide. *SIGARCH Comput. Archit. News*, 41(2):17–21, May 2013.

[110] J. Sartori, B. Ahrens, and R. Kumar. Power balanced pipelines. *High-Performance Computer Architecture, International Symposium on*, 0:1–12, 2012.

[111] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey. Can Traditional Programming Bridge the Ninja Performance Gap for Parallel Computing Applications? In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 440–451, Washington, DC, USA, 2012. IEEE Computer Society.

[112] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: A Many-core x86 Architecture for Visual Computing. In *ACM SIGGRAPH 2008 Papers*, SIGGRAPH '08, pages 18:1–18:15, New York, NY, USA, 2008. ACM.

[113] M. Shah, R. Golla, G. Grohoski, P. Jordan, J. Barreh, J. Brooks, M. Greenberg, G. Levinsky, M. Luttrell, C. Olson, Z. Samoail, M. Smittle, and T. Ziaja. Sparc T4: A Dynamically Threaded Server-on-a-Chip. *IEEE Micro*, 32:8–19, 2012.

[114] J. Shin. Introducing control flow into vectorized code. In *PACT '07*, 2007.

[115] H. Singh, M.-H. Lee, G. Lu, N. Bagherzadeh, F. J. Kurdahi, and E. M. C. Filho. MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications. *IEEE Trans. Comput.*, 49(5):465–481, May 2000.

[116] J. E. Smith. Decoupled access/execute computer architectures. In *Proceedings of the 9th annual symposium on Computer Architecture*, ISCA '82, pages 112–119, Los Alamitos, CA, USA, 1982. IEEE Computer Society Press.

[117] J. E. Smith, G. Faanes, and R. Sugumar. Vector instruction set support for conditional operations. In *ISCA '00*, 2000.

[118] A. Sodani. Race to exascale: Opportunities and challenges, keynote. In *Micro-44*, 2011.

[119] Intel streaming simd extensions 4 (sse4), http://www.intel.com/technology/architecture-silicon/sse4-instructions/index.html.

[120] S. Subramaniam and G. H. Loh. Fire-and-Forget: Load/Store Scheduling with No Store Queue at All. In *MICRO 39*, pages 273–284, 2006.

[121] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. WaveScalar. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 291–, Washington, DC, USA, 2003. IEEE Computer Society.

[122] M. B. Taylor, 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, Mar. 2002.

[123] R. v. Hanxleden and K. Kennedy. Relaxing SIMD control flow constraints using loop transformations. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, PLDI '92, pages 188–199, New York, NY, USA, 1992. ACM.

[124] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor. Conservation cores: reducing the energy of mature computations. In J. C. Hoe and V. S. Adve, editors, *ASPLOS*, pages 205–218. ACM, 2010.

[125] D. Voitsechov and Y. Etsion. Single-Graph Multiple Flows: Energy Efficient Design Alternative for GPGPUs. In *Proceedings of the 41st annual international symposium on Computer architecture*, ISCA '14, June 2014.

[126] P. M. Wells, K. Chakraborty, and G. S. Sohi. Dynamic heterogeneity and the need for multicore virtualization. *SIGOPS Oper. Syst. Rev.*, 43(2):5–14, Apr. 2009.

[127] L. Wu, R. J. Barker, M. A. Kim, and K. A. Ross. Navigating big data with high-throughput, energy-efficient data partitioning. *SIGARCH Comput. Archit. News*, 41(3):249–260, June 2013.

[128] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross. Q100: The Architecture and Design of a Database Processing Unit. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 255–268, New York, NY, USA, 2014. ACM.

[129] L. Wu, C. Weaver, and T. Austin. CryptoManiac: A Fast Flexible Architecture for Secure Communication. *SIGARCH Comput. Archit. News*, 29(2):110–119, May 2001.

[130] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee. CHIMAERA: a high-performance architecture with a tightly-coupled reconfigurable functional unit. In *Proceedings of the 27th annual international symposium on Computer architecture*, ISCA '00, pages 225–235, New York, NY, USA, 2000. ACM.

[131] S. Yehia, S. Girbal, H. Berry, and O. Temam. Reconciling specialization and flexibility through compound circuits. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pages 277–288, Feb 2009.