Mechanisms Towards Energy-Efficient Dynamic Hardware Specialization

by

Chen-Han Ho

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: 10/30/2014

The dissertation is approved by the following members of the Final Oral Committee:
Gurindar Sohi, Professor, Computer Science
Mark Hill, Professor, Computer Science
David Wood, Professor, Computer Science
Nam Sung Kim, Associate Professor, Computer Science
Karu Sankaralingam, Associate Professor, Computer Science

© Copyright by Chen-Han Ho 2014

All Rights Reserved

ACKNOWLEDGMENTS

First I would like thank my advisor Karu Sankaralingam, who had continuously supported my study and research. Without your patience, tutelage, enthusiasm and knowledge this work would have never been written. Thank you, Karu.

Besides my advisor, I would like to thank the rest of my graduation committee members: Dr. Gurindar Sohi, Dr. Mark Hill, Dr. David Wood, and Dr. Nam Sung Kim. Guri pushed me to think critically and to revisit the insights in past ideas when formulating new solutions. Mark gave me many inspiring comments, which are not only for research but also for the life and career; his teaching and mentoring style was one of my pursuits. David was a great listener (with year-round warm and smiling face), and he found many new possibility to very specific things in my research. I especially thank to Nam for encouraging me after an exhaustive preliminary exam.

I thank the CS and ECE faculty for so many wonderful courses, with special thanks to Dr. Mary Vernon and Dr. Susan Horwitz, who taught me patiently and sparked my interests in different areas. A special acknowledgement goes to Dr. Azadeh Davoodi for her teaching, mentoring and support; I greatly appreciate her recognition in my first few years in Madison. I would also like to thank Dr. Doug Burger at Microsoft Research, who is an alumnus of UW-Madison, and Dr. Hadi Esmaeilzadeh at Georgia Tech for the support on Neural Processing Units.

Thanks to the Vertical group. In particular I have received significant help and learned many skills from my collaborator Dr. Venkatraman Gvindaraju. Thanks to the DySER prototyping team, Jesse, Chris, Ryan, Zach, Preeti, and Ranjini, who all did a great work and made DySER real. Amit, Shuou, Matt, Emily, Raghu, Jai, Jin, Newsha, Vinay, and Vijay: thank you for your feedback and support. Thanks to Ziliang for the great help on writing in English. Thanks to Tony for all the fun discussions and cultural knowledge. Special thanks to Marc de Kruijf, for being more than supportive and for showing me how to write and think with ease and elegance.

Thanks to everyone in the computer architecture group. Srinath and Gagan: thank you for many stimulating discussions. Jason and Marc: thank you for all the valuable feedbacks. Nilay

and Joel: thank you for the discussions and help on the simulator. Rathijit, Somayeh, Hamid, Shoaib, Srinath, Gagan, Hongil, Lena, and Jayneel: thank you and it is my pleasure to be one of the group.

Many thanks to the architecture alumni, with special thanks to Dr. Kevin Moore for his mentoring and support. Thank you Jichuan for all the favors and advices. Dan: I did use the SPARC machine for good, not evil; you may not know but I learned many from your work. Min, Luke, Derek, Polina, and Arka: thank you and I appreciate your help.

Thanks to Oracle Labs and Lawrence Livermore Laboratory for the internships. Special thanks to Erik Schlanger, my supervisor at Oracle Labs, for his kind help and many career advices. I would also like to thank Greg Wright at Qualcomm, who gave insightful comments to this work in different aspects.

Thanks to the faculty of National Taiwan University, with special thanks to Dr. Hsu-Chun Yen, Dr. Wanjiun Liao, and Dr. Farn Wang, who never hesitate to provide help and support. Without you I could never be prepared to be in a PhD program.

Finally, I would like to thank my family. You may not know what this work is about but you are always my strongest support.

CONTENTS

Contents iii

List of Tables vi

List of Figures vii

Abstract ix

1 Introduction 1

1.1 Building Specialized Hardware 2

1.2 Dynamically Specialized Execution 3

1.3 Contributions 5

1.4 Dissertation Organization 7

2 Hardware Specialization and Dynamically Specialized Execution (DySE) 8

2.1 Hardware Specialization, Background and Examples 8

2.2 Dynamically Specialized Execution 15

2.3 Manual and Compiler-Assisted DySE 21

2.4 Chapter Summary 26

3 Dynamically Specialized Execution Resources (DySER) 27

3.1 DySER Design Goals and Overview 27

3.2 DySER Internal Microarchitecture 29

3.3 Configuring DySER 33

3.4 Integrating DySER 35

3.5 Chapter Summary 43

4 SPARC-DySER Prototype 44

4.1 SPARC-DySER Integration 44

4.2 Incorporating DySER into OpenSPARC 47

4.3 Summary and Lessons Learned 49

5 Memory Access Dataflow (MAD) 50

5.1 MAD Design Goals 50

5.2 Memory Access Dataflow Overview 53

5.3 MAD Microarchitecture 59

5.4 Complex Scenarios 65

5.5 Integration 68

5.6 Chapter Summary 70

6 Evaluation Methodology 71

6.1 Architectural Models 71

6.2 Benchmarks 74

6.3 Measurements and Metrics 75

7 Evaluation 79

7.1 DySER with Host General Purpose Processor as The Access Engine 79

7.2 DySER with MAD 88

7.3 MAD Driving Other Execute Accelerators 93

7.4 MAD Executing Access-Only Benchmarks 97

7.5 Chapter Summary 98

8 Related Work 100

8.1 DySER and Execute Architectures 100

8.2 MAD and Access Architectures 103

8.3 Chapter Summary 105

9 Conclusions and Future Work 106

9.1 Contributions and Conclusions 106

9.2 Future Work 107

9.3 Reflections 108

A The Encoding of MAD ISA 112

A.1 Dataflow Graph Node 112

A.2 ECA Rules 113

B The Set/Reset Protocol 114

Bibliography 117

LIST OF TABLES

4.1 OpenSPARC RTL modifications (comments included) 45
4.2 A stylized listing of the DySER instructions 46
4.3 Design of the DySER instruction extensions for OpenSPARC 46
4.4 Summary of the performance evaluation works 49

5.1 Dataflow patterns: Exec Comp.: Execute component, Acc. I/F: Accelerator interface instructions in the access component code, SP: Stack pointer, Off.: Offset, Base: Base address 55

6.1 General purpose host processor models 72
6.2 Architectural models 72
6.3 Benchmarks in evaluation 76

7.1 Power breakdown of DySER's sub-modules 86

8.1 Comparison to related work 104

A.1 The encoding of the dataflow graph 113
A.2 The encoding of the ECA rules 113

LIST OF FIGURES

2.1 Hardware Specialization Taxonomy . . . 9
2.2 Hardware specialization execution model . . . 10
2.3 An out-of-order general purpose processor pipeline . . . 13
2.4 Power breakdown of 2 and 4-issue out-of-order processor pipeline, modeling Intel Silvermont and Sandy Bridge . . . 14
2.5 Decoupled Access/Execute . . . 16
2.6 Decoupled Access/Execute: microarchitecture . . . 17
2.7 Performance of 2-issue OOO vs. 1-issue in-order with accelerator Speedups normalized to non-accelerated 2-OOO for both bars . . . 20
2.8 Manual DySE programming . . . 22

3.1 Dynamically Specialized Execution Resources (DySER) . . . 28
3.2 DySER microarchitecture: the network of switches and functional Units . . . 29
3.3 DySER microarchitecture: switch . . . 30
3.4 DySER microarchitecture: functional unit . . . 31
3.5 DySER microarchitecture: Phi-function . . . 32
3.6 DySER microarchitecture: configuration mode . . . 33
3.7 Fast config switching example . . . 34
3.8 Fast config switching in a DySER switch . . . 35
3.9 Simple queue-based DySER interface . . . 36
3.10 OOO DySER interface . . . 37
3.11 OOO DySER interface with speculative execution . . . 38
3.12 OOO DySER interface: An alternative solution . . . 40
3.13 Vector port mapping . . . 41
3.14 Vector port mapping: microarchitecture . . . 42

4.1 SPARC-DySER integration . . . 48

5.1 The block diagram and the execution of an out-of-order host processor with an accelerator . . . 51

5.2 The MAD ISA . . . 54

5.3 An example execution of the MAD hardware . . . 57

5.4 MAD microarchitecture . . . 60

5.5 Detailed MAD execution with a simple code . . . 63

5.6 MAD execution with a complex example . . . 64

5.7 The integration of accelerators and the MAD hardware . . . 67

7.1 The reduced dynamic instructions in percentages . . . 80

7.2 Execute component size in operations . . . 81

7.3 Speedup over 2-OOO baseline . . . 81

7.4 Energy reduction over 2-OOO baseline . . . 82

7.5 Instruction and operation level parallelism . . . 84

7.6 Instruction window size . . . 84

7.7 Dynamic power . . . 85

7.8 DySER's sensitivity on different access hardware . . . 87

7.9 Speedup over 2-OOO baseline . . . 88

7.10 Energy reduction over 2-OOO baseline . . . 89

7.11 Parallelism in MAD/DySER . . . 90

7.12 Dynamic power . . . 91

7.13 MAD's sensitivity on hardware resources . . . 92

7.14 Performance of MAD/Accelerators . . . 94

7.15 Energy reduction of MAD/Accelerator . . . 95

7.16 Energy reduction of MAD/Accelerator . . . 96

7.17 MAD (Access-only) performance and energy . . . 97

ABSTRACT

In the past few decades, Von Neumann superscalar processors have been the prevalent approach for general purpose processing. Hardware specialization, as a complementary technique, offers superior performance, power or energy efficiency on specific tasks. Today, with an increased focus on energy critical platforms such as datacenters and mobile devices, hardware specialization are becoming an important and widely used approach to improving the overall efficiency.

Our work is motivated by observing that in frequent program phases, using specialized hardware could eliminate the conventional "instruction processing" in a superscalar pipeline. To this end, we propose two supporting architectures, for both computation and data acquisition, under a hardware-software co-designed execution model– Dynamically Specialized Execution (DySE). This model leverages re-configurable hardware and decoupled access/execute for energy efficiency, generality and flexibility. The two architectures discussed in this dissertation are: Dynamically Specialized Execution Resources (DySER) and Memory Access Dataflow (MAD). Decoupling access and execute components in a program phase enables different optimization opportunities in hardware. DySER, the supporting architecture for the execute component, is a circuit-switched functional unit fabric that can be viewed as a long-latency, multi-input and asynchronous unit. MAD, on the other hand, is an event-driven dataflow memory access engine. It efficiently performs two primitive tasks found in a superscalar processor: (1) computations that generate recurring address patterns/branches; (2) event-condition evaluations that trigger resulting data movements. By turning off the host, using MAD to drive the accelerators delivers energy improvement compared to an out-of-order host processor.

This dissertation has the following findings: First, we prove that DySER is a viable approach by building a SPARC-DySER prototype, which integrates DySER into OpenSPARC. In the evaluation of DySER, we observe 80% saving in dynamic instruction count, 3.47× speedup and 3.55× energy reduction over a power-efficient 2-issue out-of-order processor; DySER increases the parallelism for such speedup through its hardware, vector interface, and decoupled access/execute. Second, we support DySER with MAD and increase the overall speedup and energy reduction over the same base to 5.3× and 5.6×, respectively. MAD can also drive other

execute accelerators or perform access-only codes for energy-efficiency. Compared to a 2-issue superscalar host, MAD increases the parallelism and lowers the power consumption through its microarchitecture, which benefits from the exposure of computation, dataflow events and actions in the MAD ISA.

1 INTRODUCTION

In the past few decades, process scaling has offered doubled transistors on a chip every generation as Moore's Law predicts. The performance of processors has consequently increased with constant power density by leveraging these smaller and faster transistors that consume less power. Conventionally, computer architects pipeline the microarchitecture for a faster clock frequency, and spend the raised transistor budget on speculative execution and dynamic scheduling (e.g., branch predictors and out-of-order execution). However, these microprocessor design techniques eventually hit the *Power Wall*. The frequency scaling ends because of practical thermal limitations; moreover, the power density per unit area has been increased over generations because of the breakdown of Dennard Scaling [41]. As a result, the industry shifted to Chip Multiprocessors (CMPs) and mitigated this problem by offering better performance through thread-level parallelism. Fundamentally, the Power Wall still exists, and the evidence of the incapacity of the traditional core architecture design has been discovered recently. Azizi et al. [14] show that microprocessor design techniques like changing issue width, improving prediction accuracy, etc. cannot provide significant improvements in energy efficiency as the . Specifically, they show that a dual-issue out-of-order processor, under voltage and frequency scaling, is within 3% of the energy-delay metric of any other out-of-order processing design. Esmaeilzadeh et al. [44] show that power limitations of conventional microprocessor techniques will severely curtail the performance, and significant changes are required to match the historical annual cumulative performance growth of 30% to 40%.

In general, the dynamic power consumption of a CMOS circuit could be represented as follows:

$$P = ACV_{dd}^2F$$

Where P is the dynamic power, A is the activity factor (the switching activity of the circuit), C is the capacitance of the circuit, V_{dd} is the supply voltage and F is the clock frequency. Among the above variables, the scaling on supply voltage V is not enough, and capacitance C and the frequency F are held constant for performance. Computer architects, consequently, turn to hardware specialization to lower the activity factor A [103, 45, 122, 57, 34, 47, 100, 88]. By

reducing the reliance on power hunger structure in a general purpose pipeline, hardware specialization decreases the overall switching activity yet provides similar or even improved performance. I begin this dissertation with a discussion on general-purpose hardware, from which I describe how hardware specialization improves energy efficiency. Next, I develop Dynamic Hardware Specialization (DySE) to address the demands found in the observations. I show how Dynamic Hardware Specialization and its supporting architectures tackle the issues in energy efficiency. I then briefly outline my contributions in this work and summarize this chapter with an overview of this dissertation.

1.1 Building Specialized Hardware

General purpose hardware, typically, executes an application in the following steps. First, the execution pipeline fetches the application as instructions from the instruction cache sequentially. Second, these instructions are decoded, where each of the decoded instructions carries a piece of information in the application. In a limited instruction window, the information from instructions is analyzed and scheduled to drive the execution units and storage inside the pipeline. The instruction window is either the same as the issuing width or is much larger in the case of out-of-order execution. To utilize the pipeline more aggressively, speculation is often used with out-of-order execution to schedule the speculated instructions and pay the potential recovery cost in mis-speculations. The above fetch, decode and issue stages are the front-end of the pipeline. Last, the back-end of the pipeline, which operates on the work of the application, is composed of the execution units and the data from storage. The execution units read from register file and data cache for operation, and write the resulting data values back. The controlling and forwarding signals of the execution units and storage are produced dynamically by the information from the front-end.

Specialization alters the general purpose hardware with specific optimizations. In the front-end, a specialization approach changes the general purpose execution model such that there is less dynamic scheduling (which saves power) in a given instruction window. For example, the Single-Instruction-Multiple-Data (SIMD) execution model operates on multiple data. Compared

to using multiple instructions, SIMD effectively reduce the overall instruction scheduling. In the back-end, the execution units may be specialized for some sophisticated application. Depending on the granularity, the execution units may be organized to reduce the overhead in dynamically producing the controlling and forwarding signals. For example, C-Cores [122] lays out the execution units in a region of the pipeline as a single piece in hardware; the control of its execution units, data access and forwarding datapath are all statically routed and performed to save power. Another optimization in the back-end is to optimize the data acquisition mechanism. Specialized memory [90, 30], registers [108], and/or scratch pad memory [67, 40] are examples of this optimization. There are also proposals only focused on data acquisition such that they use a simple in-order Von Neumann pipeline with optimized memory and memory interface [77] to achieve energy efficiency.

1.2 Dynamically Specialized Execution

This dissertation seeks to provide a comprehensive hardware specialization which eliminates the per-instruction overheads in both the front-end and the back-end of a general purpose processor pipeline. This approach, called *Dynamically Specialized Execution (DySE)*, leverages an alternative execution model based on Decoupled Access/Execute [112] to effectively create a huge instruction window with "ultra-wide" instructions. In the DySE execution model, the application is abstracted as a sequence of ultra-wide instructions (through compiler or programming techniques), each representing an application phase. Before entering a specialized application phase, the host general purpose processors *dynamically* configure the hardware to specialize for these application phases. In an application phase, the code is divided into access (memory loads and stores) and execute components (mainly computations), where the two components are executed on different hardware substrates. This decoupling thus allows different optimizing opportunities. By dynamically configuring and reusing of the configured phases, the specialized hardware substrate is able to target a variety of different applications.

Two supporting architectures are proposed under the DySE model, *Dynamically Specialized Execution Resources (DySER)* and *Memory Access Dataflow (MAD)*. They are hosted by a general

purpose processor pipeline and can be used separately or together to maximize the energy efficiency. First, DySER is proposed as a non-intrusive, multi-ported, long latency and asynchronous execution unit in the processor pipeline. It can be integrated into the host processor pipeline non-intrusively with instruction set extensions and interface buffers, and performs the execution component in an application phase. Second, MAD is motivated from the fact that driving specialized execution units, such as DySER, with host general purpose processor pipeline incurs significant overhead. In an application phase that executes with a specialized execution hardware substrate, the host general processor's role is merely delivering data from the cache to the substrate. As a result, a light-weight memory access dataflow engine, MAD, can be employed to access the cache and move the data between cache and the specialized hardware.

The design goals of DySE, DySER and MAD are as follows:

- **Energy efficiency:** The DySE model is proposed to remove the overheads in general purpose hardware. It aims at energy efficiency such that using DySER and MAD brings power efficiency (removing the overheads) with improved performance. DySE achieves power efficiency by decoupling the access and execute code, thus creating an extremely large instruction window without general purpose structures. DySER and MAD then perform the execution and data acquisition with their efficient dataflow microarchitecture. On the performance side, MAD and DySER extract instruction and data level parallelism in the access and execute code, utilizing a vectorized memory interface.

- **Area efficiency and programmability:** While Application-Specific Integrated Circuits (ASICs) can be built for different applications or application phases, DySE dynamically specializes the application phases on DySER and MAD and achieves area efficiency. To this end, DySER and MAD are designed for reconfigurability. In the DySE model, compiler or programming techniques can be used to identify the profitable application phases and create the configuration bits. This model removes the dynamic energy overhead of using hardware to analyze applications and phases at runtime for constructing specialized datapath.

- **Design complexity and flexibility:** The DySE model and the supporting architectures are

designed as a viable and flexible approach. A compiler framework was also developed for decoupling the access and execute component and generating optimized configuration automatically. This access/execute decoupling reduces the hardware design complexity, offering a well-defined flexible interface between host processor, DySER and MAD. With this interface, the host processor can use DySER, MAD or both of them. When using DySER alone, the access code can be executed on the host processor so that it is responsible for both configuration and data delivery. When applying both MAD and DySER, the host processor can be turned off during the specialized application phase to save power. In the case that the application phase is mostly composed of data accesses (e.g. a linked-list traversal), MAD can also be used alone with no execution specialization.

1.3 Contributions

This dissertation makes contributions by conceiving the Dynamically Specialized Execution model (DySE, Section 1.3.1) and the two supporting architectures, Dynamically Specialized Execution Resources (DySER, Section 1.3.2) and Memory Access Dataflow (MAD, Section 1.3.3). It studies the aspects of microarchitecture, architecture, the execution model, compiler and application to explore the benefits of dynamic specialization.

1.3.1 Dynamically Specialized Execution

The hardware specialization approaches can be roughly classified under two different building philosophies: specialization at fine granularity, focusing only on computation or execution units, or specialization at a coarse granularity that encapsulates the execution units, data access mechanism, and the control logic holistically. The first contribution of this work is the DySE model (Chapter 2), which captures both computation and memory access in application phases at a coarse-grained level but achieves fine-grain specialization efficiency by decoupling access and execute to enable efficient hardware optimization. Second, this dissertation supports DySE with a few observations about application phases and the energy efficiency of a general purpose superscalar pipeline. It develops the entire software/hardware stack and describes how to

decouple the access and execute components with compiler or programming techniques. This decoupled execution addresses the inefficiencies in a general purpose pipeline by leveraging specialized architectures.

1.3.2 Dynamically Specialized Execution Resources

This dissertation proposes the specialized hardware architecture for the execute component in the DySE model. It describes this hardware, Dynamically Specialized Execution Resources (DySER, Chapter 3), in terms of the architecture and microarchitecture details. DySER dynamically synthesizes specialized datapaths for computation operators in an application phase and relies on the host processor or MAD to deliver the data. To explain the latter, this dissertation discusses the integration of DySER with an out-of-order processor or MAD, as well as the microarchitectural interface and DySER's reconfigurability. It then presents a thorough analysis on Parboil [6], Throughput Kernels [110] and Rodinia [25] benchmarks to show the overall performance, power and energy efficiency (Chapter 7); this dissertation also conducts a series of microarchitectural analyses to reveal the tradeoffs and potential bottlenecks. DySE and DySER are joint works with my collaborator , Venkatraman Govindaraju [58].

To prove that DySER is a flexible and viable approach, this dissertation demonstrates SPARC-DySER, an OpenSPARC T1 processor integrated with DySER, with a FPGA prototype. It details the practical integration issues, and the lessons learned from bringing up a prototype. Chapter 4 gives a strong evidence that one can straightforwardly build DySER following its design specification and integrate it with a commercial processor.

1.3.3 Memory Access Dataflow

Last, this dissertation proposes the specialized hardware architecture for the access component in the DySE model, Memory Access Dataflow (MAD, Chapter 5). MAD is the first work to provide a unified interface for all accelerators and specialized hardwares that fit in DySE (or the Decoupled Access/Execute model). It introduces the event-action dataflow ISA to expose the data movement to the MAD hardware. This dissertation illustrates the execution flow of MAD, the architecture, microarchitecture, and the integration details of DySER and MAD. It

also discusses the integration between MAD and three other accelerators, x86 Streaming SIMD Extensions (SSE) unit, Neural Processing Unit [45], and Conversation Cores [122].

1.4 Dissertation Organization

Chapter 2 discusses the motivation, background material, and related work of Dynamically Specialized Execution. Chapter 3 elaborates DySER's architecture and microarchitecture designs, and Chapter 4 integrates DySER into OpenSPARC T1. Chapter 5 develops the MAD architecture with supporting examples. Chapter 6 discusses the experimental setup and evaluation methodology. Chapter 7 first evaluates DySER with the host processor delivering the data; it then evaluates MAD with DySER, MAD with other accelerators, and a special case where MAD is used alone, without execute accelerators, for access-only application phases. I conclude this dissertation in Chapter 9 and summarize the key insights, findings, and future work.

The contents of Chapter 2, 3 and 7 inherit from my prior publications, HPCA2011 [57] and IEEE-MICRO2012 [59]; they overlap with the dissertation of Govindaraju [58] , my collogue and the co-author of the above publications, but provide more detail and differ in evaluation. Regarding differences, Chapter 2 focus on hardware rationale and manually optimized data parallel benchmarks. Chapter 3 and 7 focuses on the microarchitecture and analyses that details the potential overheads of the microarchitecture design. The content of Chapter 4 is a follow-up work of published prototype at HPCA2012 [18]; this Chapter discusses the integration and design choices in-depth. Last, the MAD architecture described in Chapter 5 and its evaluation differ from the prior work [28] with a more developed design.

2 HARDWARE SPECIALIZATION AND DYNAMICALLY SPECIALIZED EXECUTION (DYSE)

This chapter discusses the background of hardware specialization and proposes the Dynamically Specialized Execution (DySE) model. Section 2.1 discusses different execution models and classifies the common specialization approaches based on programmability and granularity. It then investigates a traditional out-of-order (OOO) superscalar processor to understand how specialized hardware, with the above differences, can save power and increase the energy efficiency of an OOO processor pipeline. Through the above observation, Section 2.2 develops the DySE model as an alternative execution model based on decoupled access/execute and two supporting architectures, Dynamically Specialized Execution Resources (DySER) and Memory Access Dataflow (MAD). Section 2.3 then describes how to program for DySE and discusses the efforts in automating the transformation, from a conventional program to the codes and configurations that can be executed with DySE supporting architectures.

2.1 Hardware Specialization, Background and Examples

Today, the general propose processor design is dominated by Von Neumann superscalar processors. The idea of hardware specialization, however, is used extensively to support the general propose processors in improving performance, reducing power, or both. From a hardware viewpoint, these specializations can be characterized in two axises:

- **Programmability:** Specialized hardware can either be programmable or completely static like Application-Specific Integrated Circuits (ASICs). Among the programmable specializations, they can be pure-dynamic which reads instruction in-flight as superscalar processors do, or are synthesized by compiler or programming techniques before invoking the specialized hardware. Examples of the former include CCA [36], VEAL [31] and PipeRench [54]; the latter includes Field-Programming Gate Array (FPGA) based accelerators, GARP [68] and Chimaera [126]. The trade-off between different programmability is often the power efficiency; more dynamism in hardware in general consumes more power.

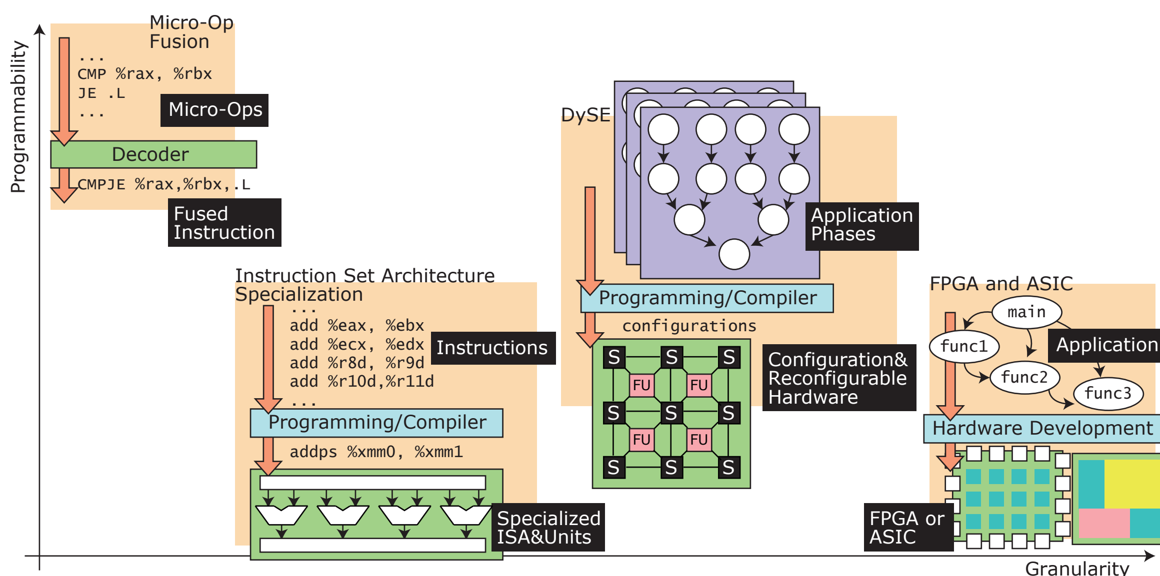


Figure 2.1: Hardware Specialization Taxonomy

- **Granularity:** Specialized hardware offloads work from a general purpose processor; the specialization granularity depends on the effective instructions in the work they offload. A fine-grain specialized example is Micro-Op Fusion, where a few micro-ops are bundled together and be executed in a specialized datapath. In contrast, coarse-grain specialization may effectively offload hundreds of instructions at one time such as in Graphics Processing Units. In this dissertation, *we do not consider an approach that differs from general purpose Von Neumann pipeline but "specializes" arbitrary applications, since they are essentially general purpose.* For instance, dataflow machines like RAW [119], TRIPS [109], and WaveScalar [117] are not considered as hardware specialization approaches.

Figure 2.1 presents the taxonomy of hardware specialization. The leftmost block illustrates Micro-Op fusion, which has the highest programmability. In the execution, the hardware (decoder) dynamically analyses a collection of instructions and produces fused instructions. No static programming or compiling effort is required before using the fused datapath, and the hardware can be used in any program that contains the target instruction sequence. Although their granularity limits the overall benefit, they can be very efficient in implementation and no modification is needed in the software stack.

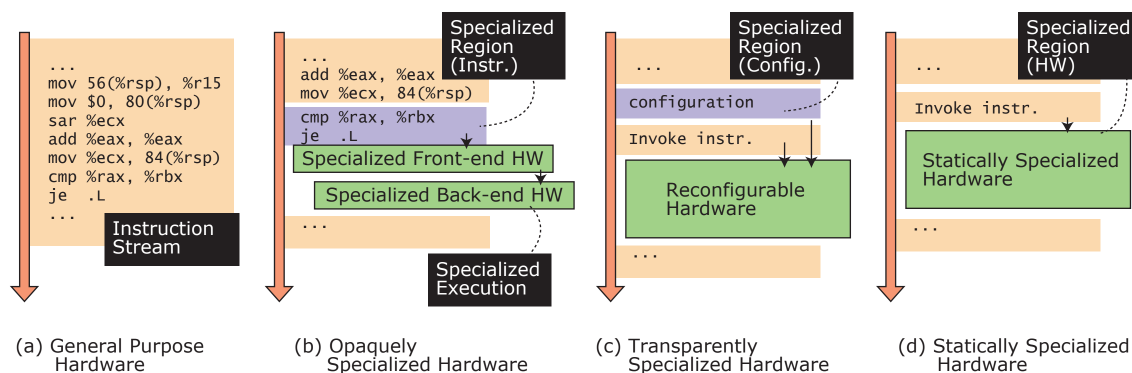


Figure 2.2: Hardware specialization execution model

Instruction set architecture extension specialization is illustrated at the next column; the examples are multimedia SSE [4] ISA extensions and cryptographic hardware accelerators [3, 1, 111]. In these cases, the specialized hardware are not programmable and executes pre-defined instruction extensions. Compiler or programming model changes have to be made in order to leverage the specialized hardware.

At the far right, FPGA and ASIC based accelerators [86, 23, 123] have the coarsest granularity and can offload a large code region or the entire application in hardware. The trade-offs here are the communication overhead, custom memory/storage design and software changes. Compared to ASICs, FPGAs are more programmable but less power efficient in hardware microarchitecture.

2.1.1 Employing Specialized Hardware in Different Execution Models

Orthogonal to the granularity and programmability, specialized hardwares are often integrated in a general purpose system with some alternation in the original execution model. Figure 2.2 compares a general purpose execution model and specialized execution models:

- **Model (a), General Purpose Hardware:** In the general purpose execution, the pipeline processes instruction streams sequentially through pipeline stages. The instructions are dynamically analyzed and assigned to execution units in the pipeline.

- **Model (b), Opaquely Specialized Hardware:** The first set of examples of the specialized hardware execution model includes a specialized front-end to analyze the instructions and program the specialized hardware dynamically. The specialized front-end constructs the

specialized region in-flight and controls back-end units for execution. This model matins the homogeneity in the software at the cost of a hardware front-end, which may consume much power when the back-end units are large and difficult to program. Micro-Op fusion and CCA [36] belong to this model.

- **Model (c), Transparently Specialized Hardware:** The second set of examples relies on the software to generate configurations for the specialized hardware. Analogous to the instructions, the configurations contain the information to control the specialized hardware; previous models differ in the fact that the configurations in this model have little or no abstraction. No dynamic analysis (such as decode, instruction queuing, and reordering) of configurations is needed. During the execution, the host processor sends the configuration bits to set up the specialized hardware before starting the specialized region. This model preserves the capability of targeting different tasks and improves the power efficiency with configuration cost and compiler/programming changes. Chimaera [126] and DySE [57, 59] belongs to this model.

- **Model (d), Statically Specialized Hardware:** In the specialized execution model, a region in the instruction stream is offloaded onto the specialized hardware. Static hardware specialization like ISA extension specialization utilizes integrated instructions to offload the work from the host. These specialized instructions invoke pre-defined (before silicon) functions in the specialized unit that is not programmable. Compared to other approaches, this model is not opaquely specialized because programmer have to invoke them explicitly; but it is also not fully transparent in terms of the programmability. By sacrificing the programmability, statically specialized hardware offers the best power efficiency.

2.1.2 Improving Efficiency with Specialized Hardware

With a modified execution model, the host general purpose hardware can exploit specialized hardware to reduce overheads. Figure 2.3 is a block diagram of a hypothetical general purpose out-of-order pipeline. It resembles state-of-the-art out-of-order processor pipeline designs such

as Intel's Haswell [66]. Logically, this general purpose pipeline has six stages: *Fetch, Decode and Dispatch, Issue, Execute, Memory,* and *Write-back* stages. [1] In particular:

- **Fetch:** The fetch stage is guided by the program counter and branch predictor to fetch instructions from the instruction cache. Often, an entire cache line is read from the instruction cache to lower the memory access overhead; buffers are used to hold these instructions temporarily. The fetch logic then selects the desired instructions following the program order and delivers them to the next stage. Because the fetch width (i.e. the number of instructions that fetch stage delivers) may not perfectly match the number of instructions in a cache line, dynamic scheduling of instruction access, buffering, ordering and selection are needed. While opaquely specialized hardware (model (b)) can hardly specialize this stage , static specialized hardware and transparent specialized hardware (model (c) and (d)) reduce this dynamic scheduling by fusing the instructions.

- **Decode and Dispatch:** The decode and dispatch stage decodes the instruction, buffers them in the instruction queue, searches for available resources that could execute the instruction, and allocates entries in the scheduler in the next (issue) stage. This stage also resolves the register structural hazards by register renaming, which dynamically assign a local register to an entry in the physical register file. The scheduling, buffering, and the size of the physical register file determines the length of the instruction window of this pipeline. Specialized hardware units usually have dedicated resources such as execution units and datapaths; even with shared resources, the specialized execution model can reduce the overhead of general purpose dynamic scheduling in many cases. For instance, the SIMD accelerators may read data values from a shared register file; compared with a general purpose processor, they can still reduce the overhead of scheduling because now only a single SIMD instruction is needed for scheduling multiple computation operations.

- **Issue:** The issue stage selects the instructions to drive the execution unit, tracks the execution status, and wakes-up the dependent instructions in the decode and dispatch stages for resource allocation. A scheduler (variant designs could be called issue queue,

[1] Each stage may be physically pipelined internally.

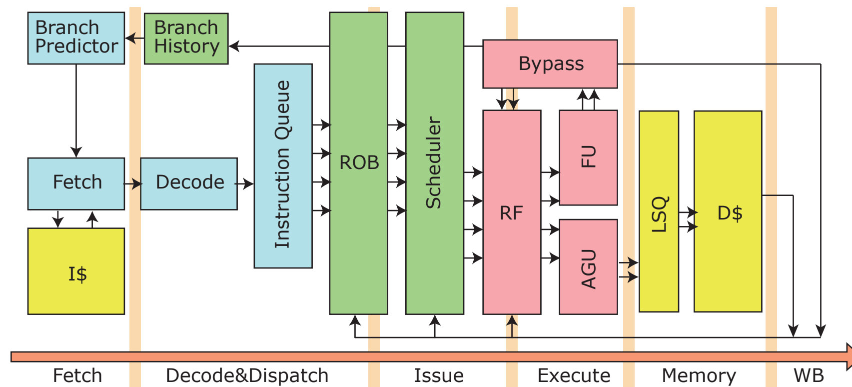


Figure 2.3: An out-of-order general purpose processor pipeline

reservation station, etc.) in this stage uses CAMs to match and wake up instructions and operands, uses arbiters to drive execution units, and uses buses to broadcast and distribute data. In addition, a reorder buffer (ROB) is accessed in parallel to hold the instructions in flight and maintain the program order. In the case of mis-speculation, exceptions and traps, a precise state can be restored by inspecting the ROB and flushing/re-executing the correct instructions. In static specialized hardware and transparent specialized hardware (model (c) and (d)) the operator scheduling is determined before execution by a dedicated datapath or configuration bits. In opaquely specialized hardware (model (a)), the specialized scheduler may have to cooperate with the general purpose scheduler in this stage.

- **Execute:** The execution stage comprises execution units (operators), datapath for operators, and bypass logic. Common execution units may include: integer and floating-point arithmetic units (ALUs), address generation units, multipliers and divider(s). When dependent instructions are scheduled in consecutive cycles, the data value can be forwarded through the bypass logic. In practice, the complexity of bypass logic scales quadratically with the number of execution units [69]. Specialized hardware units can significantly reduce the complexity in the bypass logic with a pre-determined static datapath. In addition, they may combine the bypass logic and execution units into fused execution units, thus enable circuit level optimization.

- **Memory:** The memory stage accesses the data cache and brings the operands into registers.

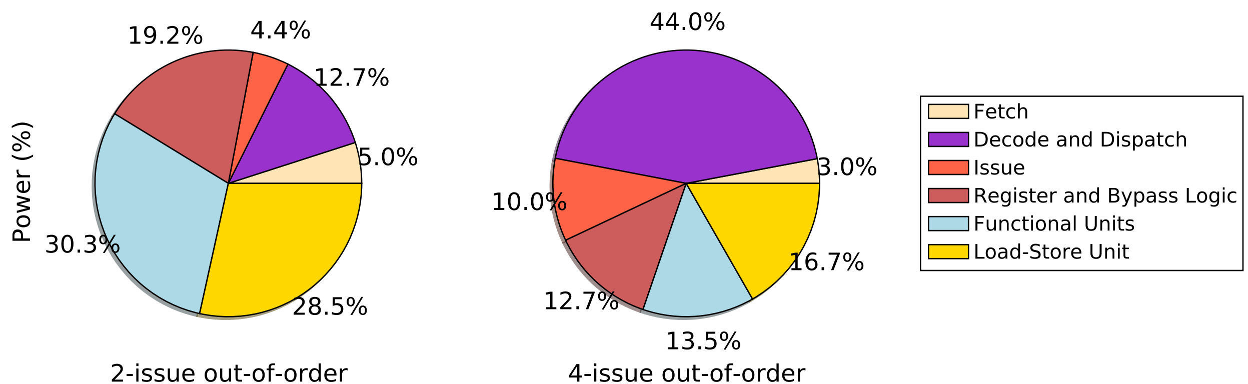


Figure 2.4: Power breakdown of 2 and 4-issue out-of-order processor pipeline, modeling Intel Silvermont and Sandy Bridge

It buffers the memory load and stores, performs memory disambiguation, and tracks the miss state by Miss Status Holding Registers (MSHR). Sophisticated mechanisms have been developed to tolerate memory latency [74, 27, 121] and reduce the complexity of the memory buffers [116] in a general purpose pipeline. Regarding specialized hardware, fine-grained proposals utilize the well-developed general purpose pipeline to deliver data from the cache. Coarse-grained proposals could use specialized memory [90, 30], registers [108], or scratch pad memory [67, 40] to reduce the dynamism in this stage. However, changing the memory system is intrusive and often demands a different programming model.

- **Write-back:** The write-back stage writes the resulting data values from the execute and memory stages back to the register file. Also, the ROB commits and retires instructions in this stage if it is safe. Specialized hardware without a centralized register file may lower the switching activity of accessing the multi-ported SRAMs.

Figure 2.4 quantifies the power consumed in each stage of an efficient 2-issue and a 4-issue out-of-order core, which models Intel's Silvermont and Sandy Bridge respectively [7, 5]. The figure reports the power from the execution of data parallel workloads; the experimental setup, and details of workloads and models can be found in Chapter 6. Here, this result presents a first-order observation–even with a power efficient out-of-order core, one-third of the overall power still goes to the front-end, register and bypass logic of the pipeline, where this power is *not* used for computation or accessing memory. In a high-performance core, the design is

even unbalanced, and the power-hungry out-of-order structures consume over 50% of the total power.

2.2 Dynamically Specialized Execution

The Dynamically Specialized Execution model, as shown in the third column in Figure 2.1, leverages reconfigurable hardware to provide programmability with coarse granularity. It adopts the reconfigurable specialized hardware execution model such that the application phases are compiled/programmed into configurations and sent to specialized hardware before executing the phase. These phases communicate with each other through memory. Overall, the execution proceeds in three phases: (1) host processor executes non-specialized phase; (2) host processor configures the specialized hardware into a specialized phase; and (3) host processor invokes specialized hardware to execute the specialized application phase. To understand the execution inside a specialized phase, we first revisit the Decoupled Access/Execute model first proposed in ISCA1982 [112].

2.2.1 Inside Specialized Application Phase: Decoupled Access/Execute

The Decoupled Access/Execute model (DAE) [112] was proposed to efficiently issue the instructions that are designated for access and execute. It hides the delay of the memory communication by decoupling these responsibilities. Figure 2.5 presents the basic concept of the DAE model with an example of specialized execution. Beginning from the left, an arrow represents the non-specialized execution of an application, and a purple box shows the application phase to be specialized [2]. When reaching the specialized application phase, the execution splits into two arrows, which represents the two decoupled components, the access component and the execute component. The access component is composed of loads, stores, and address calculations; the execute component is the computation— $a + b$ in this example. The two decoupled components can be executed independently and communicate asynchronously; as a result, a pipeline is formed between the access and execute components. The access component loads data values

[2]Originally, the DAE proposal exploits two instruction streams for general purpose execution; there is no transformation between non-specialized to specialized execution

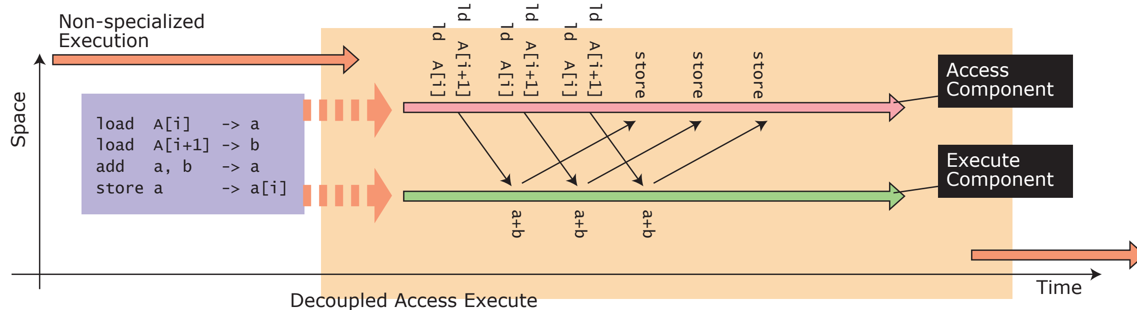


Figure 2.5: Decoupled Access/Execute

by iteration, and sends them to the execute component for computation. Originally, DAE was proposed to tolerate the memory access latency with such a decoupled pipeline (at that time, there was no cache).

The DAE model resembles the data streaming model [15, 20] when the access component works on subsequent items that can be modeled in data streams; however, they differ in the fact that DAE's access component can perform random memory access and trigger a different parts of the execute component. Figure 2.6a presents the original design of the DAE microarchitecture. Two processors, the Execute processor (E-processor) and the Access processor (A-processor), executes two different instruction streams, where the two processors communicate through data (AEQ, EAQ) and control (BranchQ) queues. The A-processor is responsible for issuing the memory reads and writes in the access component, and the E-processor is responsible for computing with the operands sent from the A-processor. In addition, the A-processor could send control signals through branch queue to drive the E-processor to a different branch when it completed the computation. Overall, DAE physically provides two decoupled microarchitectures to execute the decoupled instruction stream, thus allowing the two components to be optimized separately. It contrasts with pre-fetching schemes [74, 27, 121] apropos of the modification (decoupling) in the main instruction stream [3].

While originally the DAE model was proposed to mitigate the memory latency, this transformation concept influences superscalar processors [73] as well as many specialization pro-

[3]Recent pre-fetch proposals often leverage spare cycles to pre-load the data into cache non-bindingly (the data may not be used), where DAE explicitly alters the instruction stream and decouples it into two components that are performed in different hardware units.

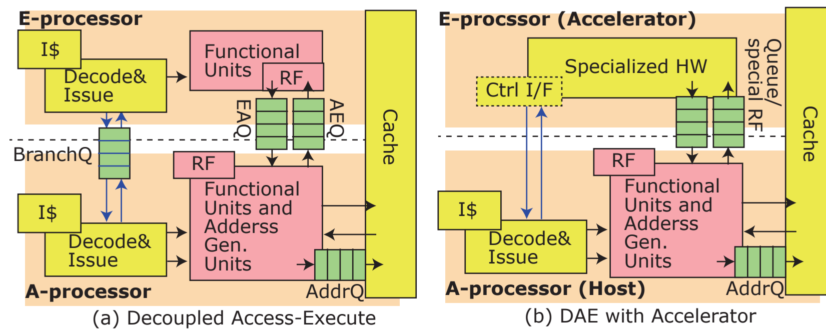


Figure 2.6: Decoupled Access/Execute: microarchitecture

posals [118, 92, 11]. Figure 2.6b describes a common DAE paradigm: an accelerator integrated into host processor, where the accelerator replaces the execute processor, and the host processor becomes the A-processor. This paradigm is widely used in various specialization proposals; if the accelerator is reconfigurable and fits in the Reconfigurable Specialized Hardware execution model (as described in Section 2.1.1), the host processor and accelerator can be viewed as using the DySE model as well. The profitability of DAE in the specialized phase (or the DySE model) largely depends on two characteristics: (1) the construction of a specialized phase, and (2) the phase behavior. In a prior work on DySE, "Energy Efficient Computing Through Compiler Assisted Dynamic Specialization" [58], Govindaraju conducted a series of analyses on the compiler assisted DySE to understand the effectiveness of such a model. The two major findings were:

- Across a wide variety of benchmarks, it is observed that the application phases of enough computation instructions (execute component) can be identified for specialization; and
- Most applications re-invoke the same phase multiple times before switching to another.

From the above, with efficient specialized hardware for execute and access components, the DySE model can be applied to a general purpose application and provide performance and energy efficiency on specific phases.

2.2.2 Supporting Architecture for Execute: DySER

To fully exploit the benefits of decoupled access/execute, two supporting architectures are conceived for each component. For the execute component, the hardware should address the

following design goals:

- It should be able to efficiently perform many computation operations, with some local branches but no memory loads or stores;
- It should be able to be programmed for different application phases; and
- It should be able to be interfaced with the access component, and be flexible or agnostic to the hardware implementation of the access component;

Dynamically Specialized Execution Resources (DySER, Chapter 3) is conceived to address the above design goals. First, DySER provides heterogeneous functional units for the primitive operators in an application phase. Between these function units, DySER leverages a circuit-switched mesh-like reconfigurable network to create datapaths. Although a packet-switched network allows dynamic resource allocation for data segments, it introduces more hardware overhead in decoding and forwarding data; DySER takes a radical approach that assigns every operation in an application phase to a dedicated functional unit, thus providing an opportunity to enable the circuited-switched network. In the network, dataflow is data-independent and follows a pre-configured datapath that is used many times before reconfiguration. This reuse avoids power-hungry structures like packet switching routers, centralized register files, and crossbars.

Second, while it is possible to create a common case array of frequently used functional units [57] for a phase, they have to be efficiently programmed for phases with different sizes. As a result, DySER utilizes the existing datapath to construct parallel configuration routes to the switches and functional units. The configuration of DySER is virtualized such that a phase with more primitive operations than available functional units can still be mapped to DySER.

Third, the configuration and data is delivered through a queue-based interface; DySER does not require any special front-end and all the control and data computation is driven by the readiness of the data in these queues. Via instructions extensions, host processor can send configuration to DySER, and the access hardware can also use exactly the same interface to send data and control. The primitive interface instructions are: (1) `dyserload` and `dyserstore`,

which loads data from memory to DySER and stores data from DySER output to memory; (2) `dyserinit`, which initializes DySER by sending the configuration bits.

2.2.3 Supporting Architecture for Access: MAD

Intuitively, one can use the host processor to drive DySER through the queue-based interface; in fact, many specialization proposals use a host processor and ISA extensions to control the accelerator and deliver data. In such a paradigm, the host processor is often one of the following:

- **Out-of-Order Processor:** In single-thread performance sensitive platforms, out-of-order (OOO) processors are often used as the host. However, because the dataflow between cache and accelerator in an application phase is usually dominated by a few access patterns, using an OOO processor becomes a less-desirable option. OOO's general purpose structures are power-hungry and tend to be an over-provisioned design for driving accelerators (as described in 2.1.2).

- **In-order Processor:** In-order processors are widely used in power critical platforms such as embedded systems. Some power-oriented specialization proposals [122] assume an in-order baseline so that the overall system offers best power efficiency. For other accelerators, an in-order integration may result in mediocre speedup even compared to a general purpose OOO processor. Figure 2.7 illustrates this issue with two different accelerators: DySER [57] and SSE [4] (details of the configuration and accelerator setting can be found in Chapter 6).The accelerators are modeled in the gem5 cycle-accurate simulator and executes the accelerated phases (kernels) of a mix of Parboil [6], Rodinia [25], and Throughput Kernel [110] Benchmarks. Compared to a 2-issue out-of-order with an accelerator, improvements from acceleration are severely reduced when using the in-order processor; this demonstrates that high-performance OOO processing capability is necessary for some accelerators. Our evaluation in Chapter 7 discusses the use of access component hardware with different performance in more depth.

- **In-order VLIW Processor:** One feasible approach of using in-order or low complexity processor to achieve high performance is the Very Long Instruction Word (VLIW) processors.

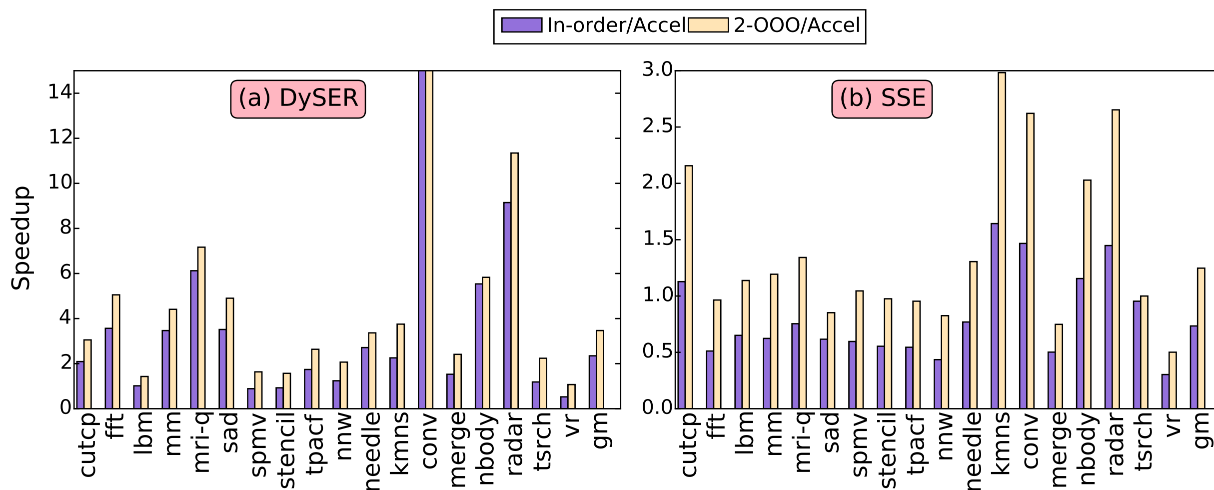


Figure 2.7: Performance of 2-issue OOO vs. 1-issue in-order with accelerator
Speedups normalized to non-accelerated 2-OOO for both bars

VLIW processors rely on the compiler to perform much of the work done in the front-end of the pipeline. The instructions are pre-scheduled and bundled in the compiler, and thus there is no dependence checking, OOO execution, and dynamic scheduling/forwarding within a bundle. Considering a general purpose host, using VLIW is cumbersome because the applications (even without specialization) have to be recompiled over hardware generations; compilers for VLIW processor may be overly complex in order to target general purpose irregular programs and achieve high performance. For our purpose, always assuming a VLIW host for the access component under DySE is unrealistic.

Our goal is to build an in-core data delivery architecture non-intrusively, which can be efficiently reconfigured for application phases, efficiently move data between cache and accelerator, and be non-intrusively integrated with a variety of execute component accelerators. To this end, we propose Memory Access Dataflow (MAD, Chapter 5); MAD translates instructions into a low-level event-driven ISA, and executes the translated events and actions on a reconfigurable dynamic dataflow substrate. By examining the access component, it can be observed that: (1) The program follows few dataflow patterns to compute the address and control behavior; (2) The outcome of these few dataflow patterns creates recurring “events“, such as values returning from cache; and (3) Based on these events, the program has a few “actions“ to move the data

between the accelerator and memory. These findings describe the fact that the access component of a specialized application phase program region is orchestrating a specialized dataflow memory movement for the computation component. An event-action/dataflow hybrid architecture can be the key towards efficient dataflow pattern computation, event triggering and action arbitration.

2.2.4 DySE Unprofitable Cases

As previously mentioned, the DySE model specializes profitable application phases–the frequent phases that have enough computation and few memory access patterns. For certain applications, however, they do not have such phases for DySE to specialize. In particular,

- **Flat phase profile:** For some applications, the phase profile is flat, and there are no prominent "frequent" phases. Specializing phases that are executed only one or two times results in no benefit, and may lower the performance because of the configuration cost.
- **Irregular and control-intensive phases:** While some applications may have frequent phases that have a reasonable compute to memory access ratio, they could be irregular and control-intensive such that it is not efficient to leverage DySER or MAD. Govindaraju et al. [60] have discovered some specific issues related to control-flow in legacy benchmarks, which limits the overall improvement from the DySE model.

These two cases, while they are not the target of this dissertation, may be solved with software techniques or different hardware designs. Supporting these phases in the DySE model is a planned future improvement.

2.3 Manual and Compiler-Assisted DySE

In the DySE model, application phases are transformed into decoupled access and execute components before running on the supporting architectures. Such transformation can be done manually by programming (manual DySE), or by compiler techniques (compiler assisted DySE). The transformation includes two steps: (1) phase analysis, which discovers the profitability of

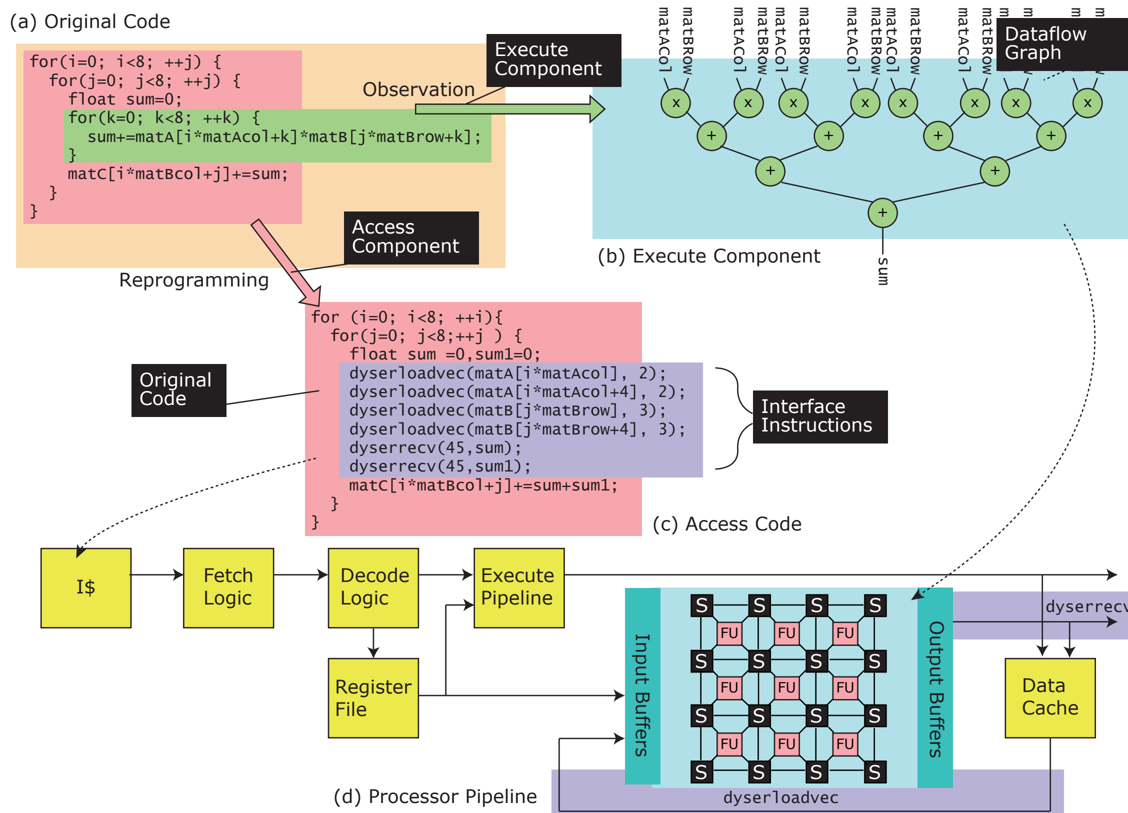


Figure 2.8: Manual DySE programming

each piece of the code and schedules them on to two different components; and (2) configuration generation, which generates the configuration for the access and execute architecture. The manual DySE transformation works on high-level code and assigns access and execute code manually. The compiler assisted transformation, on the other hand, automatically analyzes the internal representation of the phase and partitions it into access and execute components. This section presents the manual DySE transformation with an example, and discusses how the compiler assisted DySE can achieve the same transformation performance with automatic analysis and optimization techniques.

2.3.1 Manual DySE by Example

As a flexible execution model, DySE can leverage different supporting architectures to specialize the application phase. Figure 2.8 shows an example transformation on Matrix-Multiply (`mm` in the Parboil benchmark suite). For explanation purposes, the access component is performed on

host processor instead of MAD to simplify the configuration generation step. The details of the MAD ISA are introduced with the microarchitecture of the MAD hardware in Chapter 5.

First, Figure 2.8a shows the original non-specialized code of Matrix-Multiply. In the example code, three nested for loops walk through two matrices, A and B, to perform multiplication. Each element in the column of matrix A is first multiplied by one element in the row of matrix B, and then the results are summed together into matrix C. Here, the most frequently executed computations are the multiplications and additions in the inner-most loop. Through observations, a parallel structure of multiplications and additions can be discovered, as shown in Figure 2.8b. In this dataflow graph, the loop with induction variable `k` is unrolled eight times, and the multiplications can be executed in parallel; the results of the multiplication operations are merged by a tree of additions. The extracted graph meets two expectations of an execute component: (1) it has no memory operations and can be turned into pure dataflow; and (2) it is profitable to be specialized because of the operational parallelism within. With the above characteristic, this graph can be mapped onto DySER with scheduling tools that generate configuration; the scheduling tool incorporates DySER's physical hardware layout for routing and functional unit assignment.

The graph extraction and transformation has been explored in the literature; Program Dependence Graph (PDG) [48] can be used to describe the graph and its transformations in formal. In a prior work, Govindaraju et al. [60] developed Decoupled Access/Execute Program Dependence Graph (AEPDG) for compiler assisted DySE transformations. In this dissertation, the extracted execute component graph is referred to as Execute-PDG, and the access component graph is referred to as Access-PDG.

After the execute component is extracted, the rest of the code is merely preparing the memory addresses and accessing cache. The access component can be constructed by the following memory access codes and interface instructions. Figure 2.8c shows the access component of Matrix-Multiply in stylized C-style codes. In the access codes, the two outer loops remain unchanged, and the inner loop is replaced with a series of interface instructions (as stylized function calls) and an aggregation of the results. In the example, two interface instructions are used: a parallel `dyserloadPD` and `dyserrecv`. The former is the parallel variant of the primitive

`dyserload`, which loads eight elements from a cacheline, and the latter is a register variant of `dyserstore`, which retrieves a data value from the execute component (on DySER) to a register. The argument of the interface instructions are (1) register or memory address, and (2) a DySER port. The DySER port number represents an input of the execute component dataflow graph, and is physically routed to an input of the functional units (in this case, a multiplier).

2.3.2 Hardware Mechanisms to DySE

As many other specialization proposals, it is possible to completely offload the burden of identifying the profitable DySE phases to the hardware. Using the opaquely specialized hardware execution model (Section 2.1.1), we can build a front-ending for DySER to construct DySER configurations at run-time. Building such a front-end configuration engine, however, requires careful design such that this specialized front-end does not consume too much power when generating phase information and configuration. Two fundamental tasks have to be performed in the hardware in such case:

- **Phase identification:** Trace cache [105, 49] or the mechanisms in instruction reuse design [114] can be used for caching and analyzing the dynamic instruction from the host processor and create profitable phases.
- **Configuration generation:** Hardware supported binary translation or virtual machine [113] can be used for this purpose. In terms of the two supporting architecture in DySE, the DySER and MAD configuration may be generated with a greedy hardware translator that maps instructions to the functional units, switches, events or actions in their architecture.

CCA [36] and Chimaera [126] are two examples that leverage a specialized front-end to generate the configuration in hardware. In this dissertation, we do not use this approach mainly because of the overheads in power consumption.

2.3.3 Compiler-Assisted DySE

Many efforts have been made to automate the DySE transformation. Although the design and implementation of compiler assisted DySE transformation are not the focus of this dissertation [4], in this section we summarize the qualitative and quantitative findings from the implementation work, which includes the DySER Compiler published in PACT2013 [60], a spatial scheduler in PLDI2013 [95], and Govindaraju's dissertation [58].

Breaking SIMD shackles: Govindaraju et al. proposed a variant of Program Dependence Graph [48], called Access Execute Program Dependence Graph (AEPDG) to enable the compilation for DySER [60, 58]. The AEPDG is a PDG that is partitioned into two components: the access PDG and the execute PDG. After the front-end compilation, the compiler generates the control-flow graph (CFG) with standard scalar optimizations (Constant propagation, Loop Invariant Code Motion, and so forth.). Next, the compiler generates AEPDG from CFG and performs the DySE optimizations. These optimizations include: loop unrolling for PDG Cloning, strip mining for vector deepening, subgraph matching, execute PDG splitting, scheduling execute PDGs, unrolling for loop dependence, traditional loop vectorization, and load/store coalescing. Equipped with these optimizations, the compiler schedules the Execute-PDG onto DySER and inserts DySER interface instructions to the Access-PDG for the host processor.

The authors built a LLVM based compiler and compare the performance of the compiler (accelerated by DySER) with manually optimized programs and Intel's ICC (which generates SSE/AVX codes). With the flexibility in the DySE model, the compiler can employ many more optimizations and create significant parallelism for the underlying specialized hardware (DySER) than a conventional SIMD compiler. Overall, the compiler achieves a 1.8X speedup on DySER over conventional SIMD vectorization in simulation, and the compiler's performance is close to the ideal manually optimized code.

Spatial Scheduling for DySER: The DySER scheduling of the execute component, which is a graph mapping and spatial scheduling program, can be done in various ways from naive greedy

[4] All evaluations in other sections use manual transformation

algorithms [57] to more sophisticated approaches. Nowatzki et al. [95] proposed a general scheduling framework based on integer linear programming that solves this spatial scheduling problem on DySER and other spatial architectures. In the DySE-specific analysis, this ILP-based scheduler outperforms both the automated greedy approach and manual transformation by 36% and 2% in latency, respectively. It also improves overall performance and throughput in many cases.

2.4 Chapter Summary

In order to enable an efficient specialization, the execution must be freed from the original execution model; otherwise, any optimization is highly restricted so that only minor performance or power efficiency gains can be achieved over the general purpose processor. This chapter discussed the Dynamically Specialized Execution (DySE) model based on a discussion of hardware specialization. It developed DySE's application phase specialization, decoupled access/execute, and introduced the supporting architectures. It then presented the Matrix-Multiply as an example of manual DySE programming. Last, it summarized the work in automating the DySE code generation from a traditional program.

In summary, the DySE model is a specialized execution model that modularly enables different supporting architectures. The flexibility in the model and interface heavily reduces the design complexity of the design and integration of the supporting architectures, as well as the compiler optimizations.

3 DYNAMICALLY SPECIALIZED EXECUTION RESOURCES (DYSER)

The DySE execution model and transformation elaborated in Chapter 2 decouples an application phase into access and execute components, where the execute components are mapped onto the specialized hardware, DySER. Regrading power, DySER leverages static dataflow execution on a circuit-switched network to eliminate the excess dynamism in a superscalar out-of-order processor. This static dataflow, moreover, is dynamically reconfigurable between different application phases. The DySER network thus can efficiently offer operation and data level parallelism for performance with the functional units inside. This chapter discusses the hardware mechanisms that deliver performance, power efficiency, and programmability. It first outlines the design goals of DySER (Section 3.1) and gives an overview of the architecture of DySER. Next, it describes the internal microarchitecture of DySER (Section 3.2) and how to configure them with an efficient mechanism (Section 3.3). Third, Section 3.4 discusses the integration interface of DySER, and how DySER supports data level parallelism. Last, Section 3.5 summarizes this chapter.

3.1 DySER Design Goals and Overview

Dynamically Specialized Execution Resources (DySER) is proposed to perform the execute component in the DySE model. It executes the computations in a frequent application phase, represented by program dependence graphs (PDG) [48]. DySER supports the extracted execute component graph, Execute-PDG (described in Section 2.3.1), with an efficient fabric of functional units and a switch network. Specifically, DySER has several design goals:

- First, the microarchitecture of this fabric should eliminate the dynamic overheads of a general purpose processor, provided that a scheduled configuration is executed statically; during execution, DySER should exploit the pipeline and instruction (operation) level parallelism for similar or better performance as OOO processors;
- Second, DySE decouples the execute component of a specialized phase as a program dependence graph, which may have control-flow, and should efficiently support the

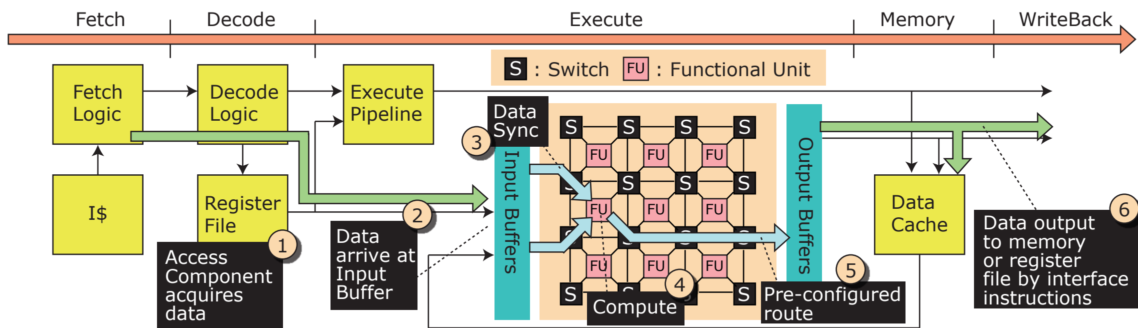


Figure 3.1: Dynamically Specialized Execution Resources (DySER)

control-flow within a phase;

- Third, DySER should be programmable to map different application phases;
- Forth, DySER should flexibly support large and small regions with hardware/software mechanisms;
- Fifth, the integration of DySER and the host processor should be flexible and non-intrusive, given the interface defined by the DySE model, and DySER should exploit the data level parallelism in the hardware for speedup over OOO.

These goals define the design of DySER; generally speaking, DySER leverages a reconfigurable circuit-switched network to remove the overhead of dynamic scheduling (goal 1). This network constructs routes that connect many functional units that can be activated in parallel for high performance (goal 2). Inside the network, DySER supports control flow with its functional units and the meta-bits (goal 3) and supports efficient configuration with the existing datapath in the network (goal 4). DySER is designed as a long latency, multi-ported execution unit that can be intuitively interfaced with queues and interface instructions (goal 5).

Figure 3.1 illustrates an overview of DySER. To simplify the explanation, the access component is performed on a host superscalar without specialization; DySER, in this case, logically resides in the execute stage. First in step 1, the access component executes interface instructions to load data from memory or the register file to DySER. In step 2, the data arrives at input buffers of DySER, where a hand-shake flow control protocol is implemented to notify the interconnect network in DySER to consume data. Next in step 3, through the protocol, the data

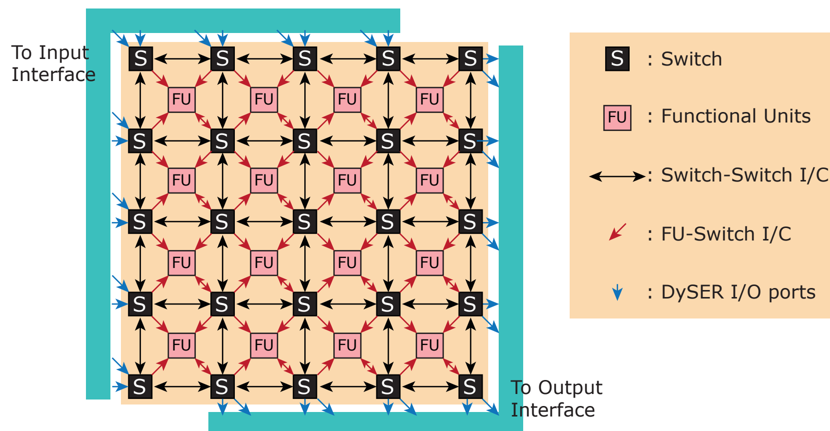


Figure 3.2: DySER microarchitecture: the network of switches and functional Units

move between switches along the preconfigured route, and they then arrive at the input of a functional unit. The data is buffered here until all inputs of this functional unit arrive; the inputs are then consumed by the functional units for computation (step 4). The result, following the preconfigured routes and passing the posterior switches and functional units, finally reaches the output buffers (step 5). Lastly, the access component executes another interface instruction to retrieve the result and move it to memory or its register.

Three major tasks in this figure are explained in the next few sections: (1) internal microarchitecture of the DySER network, which satisfies the goals 1 to 3 by design; (2) the configuration of this network, which satisfies the 4th goal with the reuse of DySER's internal microarchitecture; and (3) the integration of DySER, which fulfills the last goal via a non-intrusive interface.

3.2 DySER Internal Microarchitecture

DySER satisfies the first goal with a network of switches and functional units that can map a program dependence graph and execute it statically; this network, after configuration, does not dynamically analyze input data as a packet, but performs static dataflow execution on pre-configured routes. Figure 3.2 presents the network topology of a 4x4 DySER, which has 4x4 functional units and 5x5 switches. For the simplicity of explanation, all the functional units have two inputs and one output. The DySER network is a mesh network composed by switches (the black boxes with "S"), where the network is augmented with functional units in the "hole"

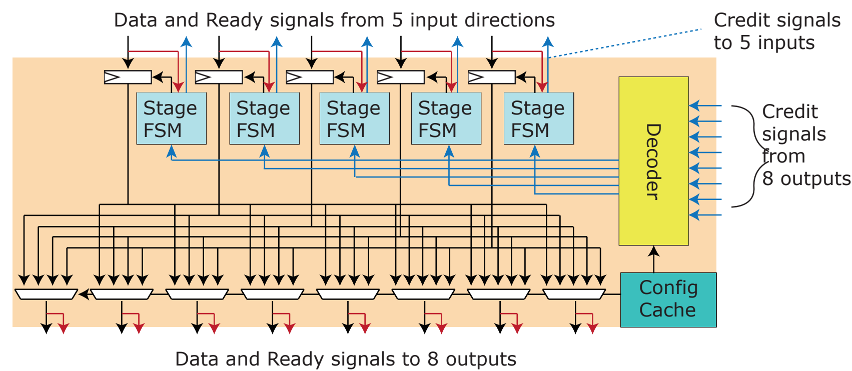


Figure 3.3: DySER microarchitecture: switch

of the mesh (Red boxes and arrows). Compared to the FPGA-like network, DySER has smaller fanins and fanouts at each switch and functional unit; this is because an FPGA tile tends to have many more input and output signals than a node (i.e., an operation, which often has few input/output data values) in DySER. At the edge of the mesh network, input and output ports of the switches are connected to DySER's input and output interface. These ports are exposed to the software such that a programmer or a compiler can write a DySE application phase that leverages the ports to send/receive data. For a 4x4 DySER, each side of the network has eight inputs or eight outputs; thus, a total of 16 input ports and 16 output ports can be used.

3.2.1 DySER Switch

The heart of the network is DySER switches, which route data values to functional units for computation. Figure 3.3 details the microarchitecture of a switch. The building blocks in a switch are multiplexers, flip-flop stages, a decoder and a configuration cache that stores the configuration. A DySER switch has five inputs (north, east, west, south, and northeast) and eight outputs (north, east, west, south, northeast, northwest, southeast, southwest) [1]. Flip-flop stages are used at each input direction to hold the data, and multiplexers are used at each output to select data from these stages. Controlling the flip-flop stages, the finite state machine (Stage FSM) is responsible for the flow-control along the configured routes. DySER implements a credit-base flow-control protocol in the stage FSM with two meta signals, a forward *ready* signal

[1] In the case of multi-output function, the switches have to support more inputs from a functional unit.

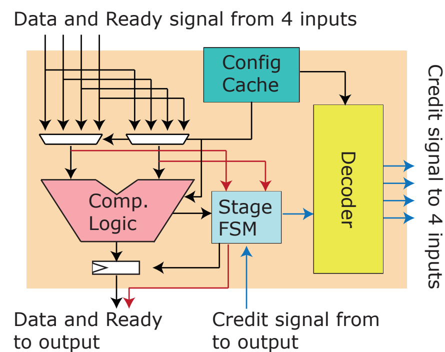
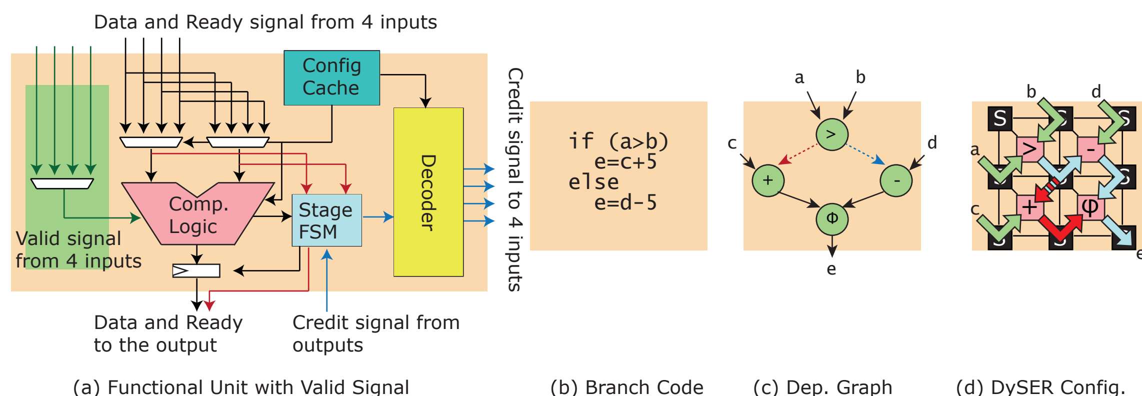


Figure 3.4: DySER microarchitecture: functional unit

(red arrows) and a backward *credit* signal (blue arrows). After configuration, each stage on the configured route is initialized with a credit. Whenever a stage sees a ready data from the previous stage, it gives its credit to the previous stage and consumes the ready data. The stage is then turned into a zero credit state, until the subsequent stage passes its credit for the ready data. The credit signals, which come from the output (subsequent stages), are decoded and forwarded to the corresponding stage FSM. This handshake protocol enables pipelining and stalling in the network.

3.2.2 DySER Functional Unit

Figure 3.4 presents the microarchitecture of a functional unit; the logics for supporting control flow is omitted in this figure and deferred to the next subsection. In a functional unit, there is a computation logic selecting input operands via multiplexers from 4 input directions (northeast, northwest, southeast and southwest), and output to the southwest neighbor switch. The stage FSM and flip-flop are also used in functional units; it in addition synchronizes the input data from different directions. The stage FSM in a functional unit controls the computation logic such that it only computes when both two of its operands are ready. When the computation finishes, the stage FSM then gives credit to both of the inputs and latches the result in the flip-flop stage. In the decoupled access/execute model, the inputs from the access component are not always synchronized (i.e., the input of a functional unit may come in at different cycles) and the synchronization inside DySER is necessary. This microarchitecture design has two advantages:



(a) Functional Unit with Valid Signal (b) Branch Code (c) Dep. Graph (d) DySER Config.

Figure 3.5: DySER microarchitecture: Phi-function

(1) It eliminates the need for a packet-switching router to synchronize the data; and (2) Stage FSM synchronization works with pipelining and stalling, thus the operand scheduling is agnostic to software (i.e., no operand scheduling has to be done by the programmer or compiler).

3.2.3 Control-flow in DySER

DySER functional units play an important role in the second design goal–supporting control-flow. Figure 3.5 shows the microarchitecture and an example from source code to DySER configuration. First, besides the $ready$ and $credit$ meta signal, all functional units in addition have a $valid$ signal along the configured path to indicate the validness (i.e., whether the values should be used in the following stages) of the execution. This valid signal works as a predicate to a functional unit such that the functional unit produces a valid/invalid output, and can be merged by a select operator, ϕ. Figure 3.5b presents a simple example which has a branch and two computations. In the program dependence graph of this code snippet, the value of e is determined by either of the branches and has multiple producers in the graph. In Single Static Assignment (SSA) form [39], the results of each branch will be assigned as "$e1$" and "$e2$" as shown in Figure 3.5c, which are then merged and selected via a ϕ-function. Based on the validness of $e1$ and $e2$, the ϕ-function generates the final output "e". Last, in Figure 3.5d, the program dependence graph is mapped onto DySER as configuration, and the functional units of operator " $+$ " and " $-$ " takes the valid signal output from the companion operator. Based on the configuration and valid input, these two functional units output data with valid/invalid

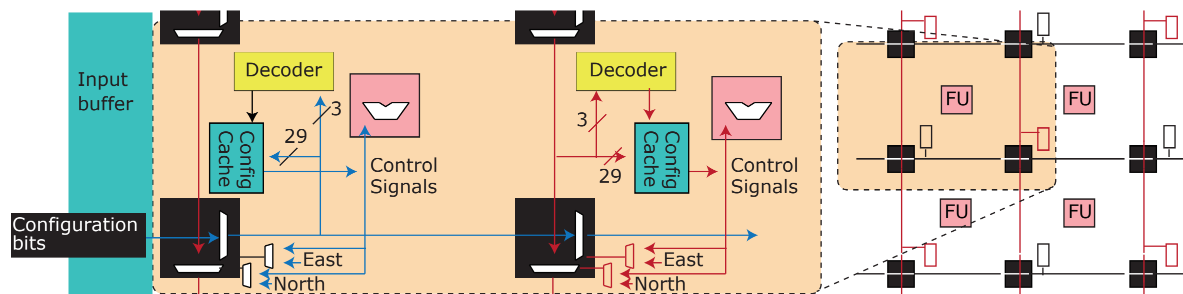


Figure 3.6: DySER microarchitecture: configuration mode

state to the ϕ-function, which selects the correct value of e.

3.3 Configuring DySER

To program DySER, the host processor sends configuration bits into the configuration caches at each functional unit and switch. DySER reuses its network to transmit the configuration bits as shown in Figure 3.6. When the host processor sets DySER into configuration mode using $dyserinit$, the switches construct parallel lanes from input to each configuration cache; in this mode, the switches are controlled by hard-coded bits and forward data from north to south and from east to west. These parallel lanes deliver configuration messages, each with a 3-bit index tag and 29 bit payload data in a 32-bit datapath. When a message reaches a switch, a comparator examines the index tag to check if this message is meant for the current node. In such case, the value is written into the configuration cache. This design configures DySER efficiently without dedicated configuration wires; further, the configuration lanes are parallel and pipelined to reduce the configuration cost. Compared to repeatedly fetching and executing, DySER is configured once and re-used many times to save dynamic power.

3.3.1 Fast Configuration Switching

It is often observed that the application phase, identified by the programmer or the compiler, may encompass many more or fewer operators than what are in the DySER network. In the case of a small phase, many software techniques (e.g., combining the neighboring phase, unrolling loops for more parallel operators [59]) can be applied to create a larger execute component with

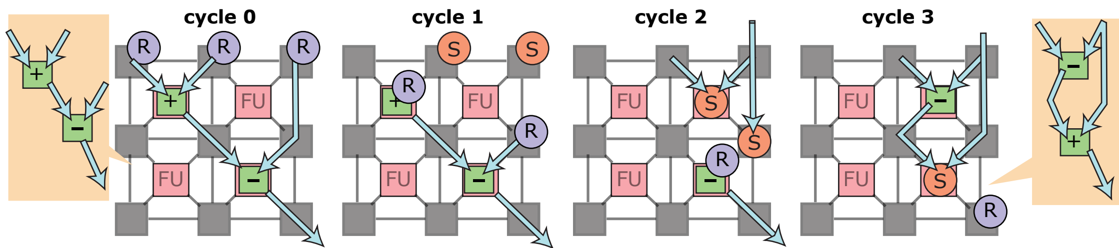


Figure 3.7: Fast config switching example

an appropriate number of operators; hardware power gating can also be used on the idling functional units to save power. In the case of an overly large phase, the execute component could be divided into regions that are proportional to the size of DySER. DySER can then be switched between different configurations of subregions of the same phase. However, this incurs configuration overhead that can be reduced through several hardware enhancements.

Figure 3.7 illustrates the basic concept of the fast configuration switching mechanism– a "Set/Reset" protocol. This protocol explicitly reuses the dataflow in the two regions to synchronize the set and $reset$ signals. These two signals are also exposed to the software such that the access component can explicitly trigger the switching of configurations. The protocol works as follows:

- The access component fires the reset signals to the DySER ports that are used in the *current* subregion of a divided phase;
- The reset signal follows the configured route of the current sub-region in the phase and turns off the functional units or switches along the route;
- During the "reseting" of the current sub-region, the access component can fire the set signal to the DySER ports that are used in the *next* sub-region; and
- If a functional unit or switch receives a set signal, and *it and all of its neighboring functional unit and switches are "freed" by the reset signal*, it can switch its configuration to the next sub-region.

In the example, the set and reset signals switch the network from configuration $(a + b) - c$ to configuration $(a - b) + c$.

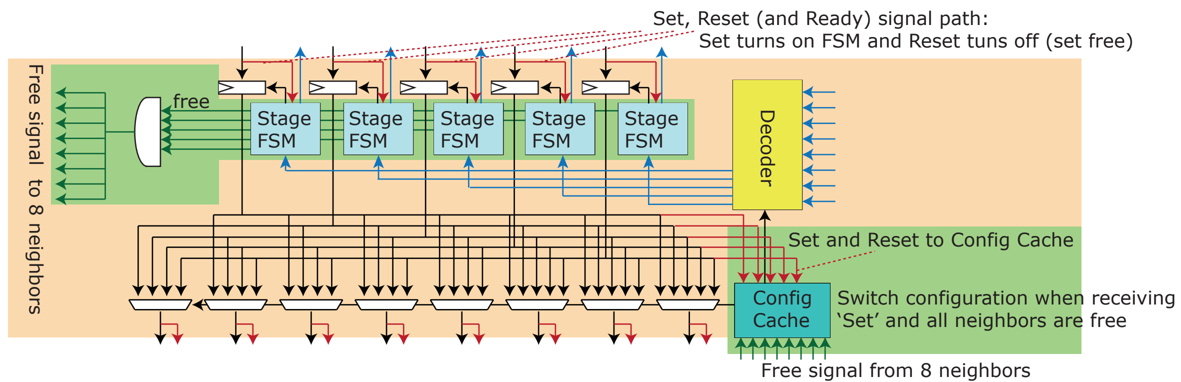


Figure 3.8: Fast config switching in a DySER switch

Figure 3.8 presents the hardware microarchitecture that enables fast configuration switching in a DySER switch. In each functional unit or switch, the configuration cache could store multiple configurations for different regions. In additional to the extra configuration storage, a 1-bit $free$ signal network is introduced to connect configuration caches; this free signal indicates that if the functional unit or switch is "freed" and can accept new operations or routing assignments. The reset and set signals, on the other hand, are augmented with the existing datapath and deactivate/activate the stage FSMs along the route. A reset-free-set hardware cycle proceeds as: (1) a reset signal deactivates the stage FSM; (2) all stage FSMs in current switch/functional units are turned off, creating a free signal that broadcast to neighbors; and (3) after all neighbors are freed, a switch/functional unit can accept set signals from its inputs (if any) and switch its configuration. The free signal in this cycle guards between reset and set signals and avoids overlapping. A first-order proof of the Set/Reset protocol can be found in Appendix B

3.4 Integrating DySER

To summarize, from the above discussion there are several communication primitives from access component or host to DySER:

- `dyserload/dyserstore, dysersend/dyserrecv`: Send/receive (register) and load/store

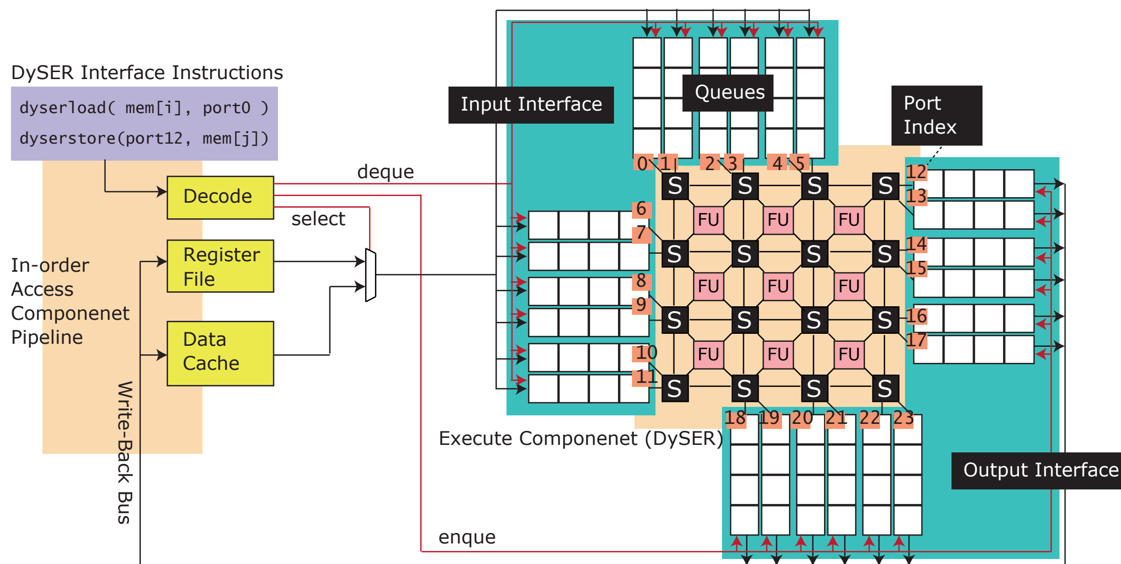


Figure 3.9: Simple queue-based DySER interface

(memory) primitives are the fundamental interface that delivers data to DySER ports in a first-in, first-out fashion;

- `dyserinit`: This primitive configures DySER by triggering the configuration mode of DySER and a series of configuration sends to DySER ports in parallel; and
- `dyserset/dyserrst`: Set/reset signals also targets DySER ports to wipe out current configuration and switch to the next one.

A DySER integrated core should provide instruction extensions for these interfacing primitives. Integrating DySER is flexible and non-intrusive; based on the execution model of the access component hardware, the DySER integration interface can be (1) a simple queue-based interface, or (2) a speculation capable queue-based interface. These two interfaces can in addition be enhanced with a parallel datapath from the memory or the vector register to the input ports and are discussed in the reminder of this section.

3.4.1 Simple Queue-based Interface

As previously mentioned, the execution component in a specialized phase is transformed into Execute-PDG, which is mapped onto DySER. During the execution, this graph could be invoked

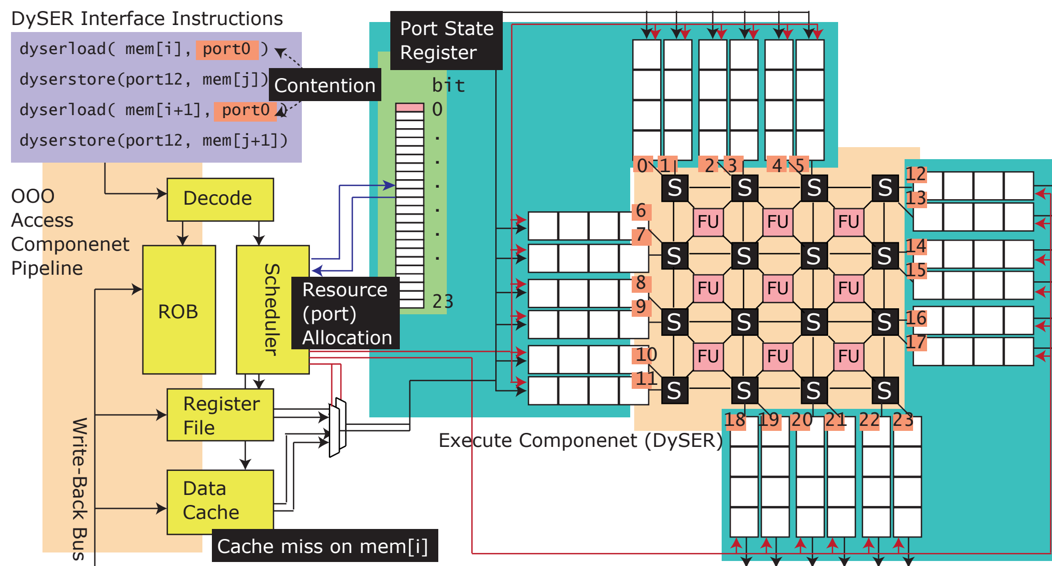


Figure 3.10: OOO DySER interface

multiple times before switching to the next phase. DySER executes the program dependence statically, following the invocation ordering; for a DySER port, the data values belong to a previous invocation are sent/received first to/from DySER, and then the data of succeeding invocations can be sent/received. As a result, DySER can be easily integrated into an in-order machine via queues (FIFOs) for buffering. Figure 3.9 presents the design of a simple queue-based interface. The access pipeline reads instruction extension for the port index and sends data to the corresponding queue, where the port index is analogous to named registers, and a decoder/multiplexer can be used to drive the input/output of queues.

3.4.2 Out-of-Order Execution and Speculation

DySER can also be flexibly integrated into an out-of-order machine, where the data values may be sent/received to queues out-of-order. Few restrictions apply:

- Within an invocation of the program dependence graph, the data values can be sent to DySER in arbitrary order;
- Between invocations, the data values that are sent/received to different DySER ports can also be done in arbitrary order; and

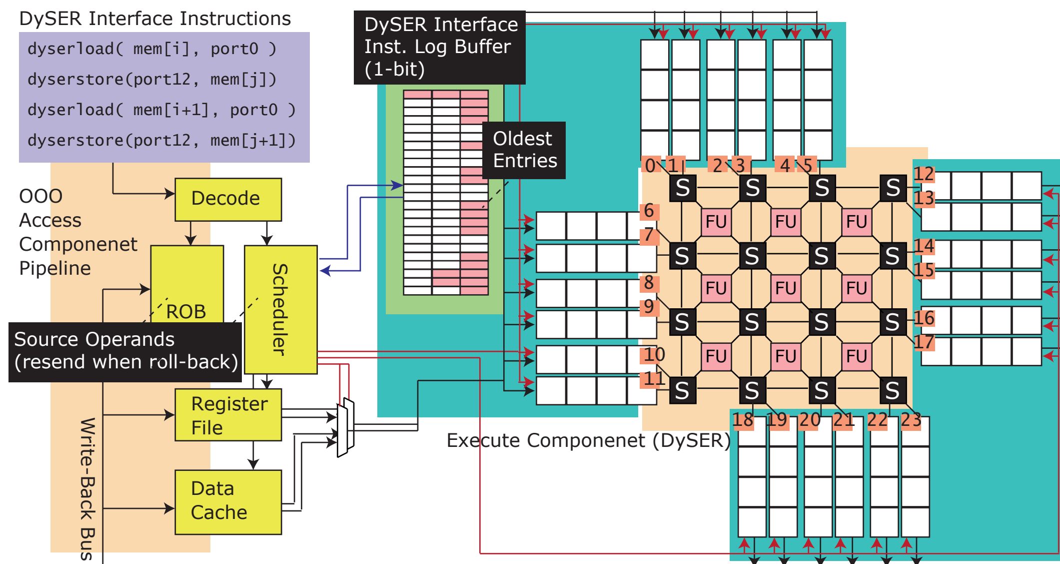


Figure 3.11: OOO DySER interface with speculative execution

- Between invocations, the data values that are sent/received to the same DySER port has to follow the invocation order.

Figure 3.10 explains the last rule with an example. The access pipeline experienced a data cache miss in the current invocation, and port 0's data is delayed due to the miss. While the next DySER instruction to port 0 may be dispatched during the delay, the data of this next instruction *cannot* be pushed into DySER because it breaks the invocation order. To avoid such situation, a port state register is used to indicate if the port is currently occupied by a delayed long latency DySER interface instruction; if so, the next DySER interface instruction is stalled from issuing to the same port. This design is analogous to the out-of-order issuing, where each port represents separate resources, and two instructions targeting the same port contend for the resource.

Speculative Execution, Exception and Recovery The access component may be executed speculatively and encounter a mis-speculation or exception. The DySER interface, as a result, has to support recovery based on the latest executed invocation of the execute program dependence graph in the application phase. In the event of an exception or mis-speculation, specifically:

- All instructions belonging to previous invocations are committed first;

- The instructions after the mis-speculated/exception instruction in the current invocation and the front-end (fetch, decode and dispatch) pipeline are flushed;
- For instructions that belong to the current invocation and are issued before the faulty instruction, a buffer is required to record the DySER interface instructions for later recovery;
- Before the recovery, DySER and the DySER interface are completely flushed;
- The DySER interface is now in a clean state; and
- The DySER interface recovers its state by re-issuing the instructions before mis-speculated/exception instruction.

Figure 3.11 presents the microarchitecture that supports recovery. The recovery of the DySER interface is analogous to the recovery mechanism in a merged register file based architecture [55], where an "invocation-wise" instruction log buffer checkpoints the latest invocation of the program dependence graph. After all previous invocations are committed and all following invocations are flushed, the host processor travels through this log buffer and recovers to a correct state by resending the source operands of corresponding DySER interface instructions via the ROB and the scheduler. The ROB, in such a case, has to work with the instruction log buffer, and prevent the speculated DySER outputs from committing.

An alternative design can be used to reduce the recovery overhead in traversing the instruction log in the current invocation and leave the ROB and the scheduler untouched. Figure 3.12 shows this approach: all input buffers are augmented with state bits to indicate the invocation state, and the recovery can be done by flushing the mis-speculated invocation and re-injecting the buffered values that were prior to the mis-speculated instruction. With the help of distributed state bits, flushing can be pipelined with re-injecting values. Based on the buffer state, DySER interface first injects dummy values to fill all ports that belong to the current invocation and create dummy outputs; DySER outputs then drop these dummy values; and then the DySER input interface re-injects the correct values based on buffer states. An element in the buffer may have the following state: $ready$ to be consumed, $valid$ to be retired, and $invlaid$ to be flushed or dropped. Compared to the first approach, this design improves the recovery speed by not

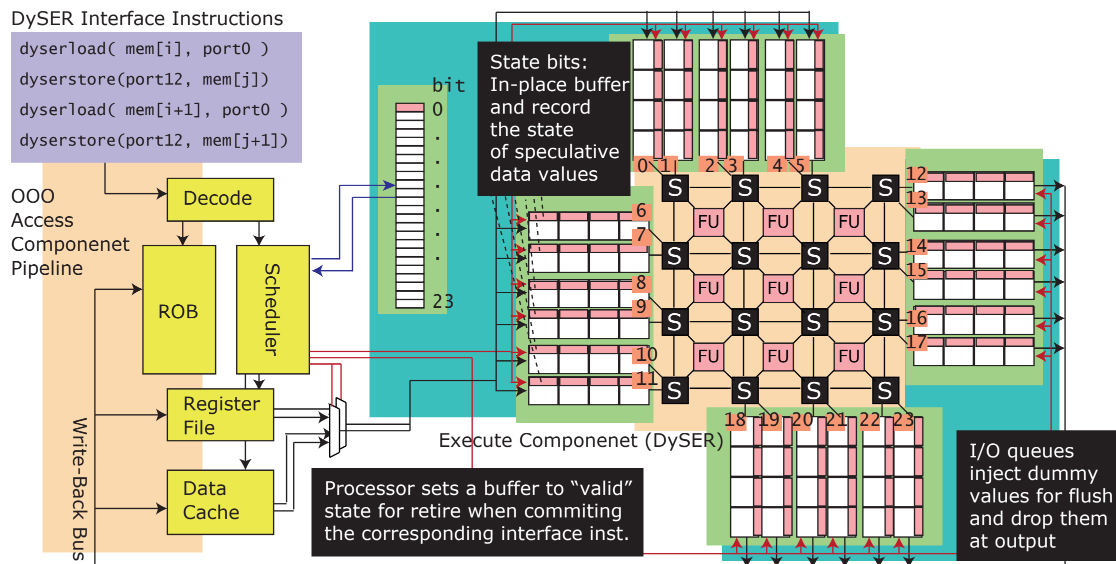


Figure 3.12: OOO DySER interface: An alternative solution

always restarting invocations from sequential traversals of the ROB and the log buffer, but comes with more hardware cost.

Page Faults, Context-Switches, etc. Pages faults may arise when the access component performs a memory access. DySER, during a page fault, can leverage the recovery mechanism discussed to restore to a given memory access instruction after a page fault is served. 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; thus no mix of different phases/program dependence graphs is allowed in DySER. 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. In all, restarting DySER after a context switch is the same as restarting DySER after a pipeline squash.

3.4.3 Vectorizing DySER Interface

The profitable application phase often has abundant data level parallelism. In order to fully exploit the parallelism in the application phase, the DySER interface buffers are enhanced to cover

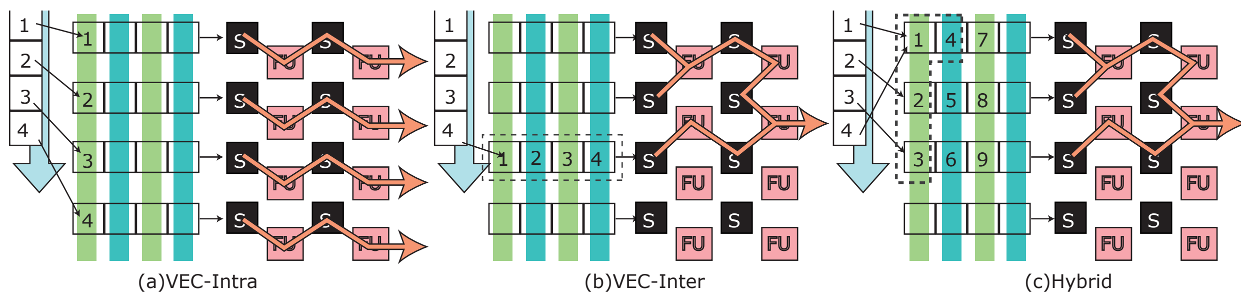


Figure 3.13: Vector port mapping

vectorized memory reads and distribute multiple data words from contiguous memory locations at once. As mentioned, a vector `dyserloadvec` is used in this case; this vectorized DySER interface instruction moreover reduces the pressure of processing instructions in the access component. This section discusses the microarchitectural support for data-level parallelism from memory to DySER.

Vectorizing Memory Access Patterns State-of-the-art superscalar processors also have SIMD/vector instruction extensions that load contiguous sections of memory to its vector interface, often registers. DySER utilizes the same concept, but provides more flexibility in mapping the data words in a loaded vector to the DySER input ports. In particular:

- **Intra-invocation Communication (VEC-Intra):** Figure 3.13(a) shows a simplified execute program dependence graph from the convolution (CONV) benchmark; each contiguous memory word is mapped to a different input port of DySER (all belong to the same invocation). This pattern is referred to as Intra-invocation (VEC-Intra) communication. Figure 3.14(a) shows a simple way of supporting VEC-Intra communication. The data loaded from memory is held in a special vector buffer, which is pre-configured to distribute data to different DySER input queues. The configuration information includes the mapping from each vector buffer entry to destination input port of DySER. Additional datapaths and multiplexers are used prior to the original DySER interface buffers in order to consume the data from the vector buffer.

- **Inter-invocation Communication (VEC-Inter):** Figure 3.13(b) shows the computation subregion from the stencil(STNCL) benchmark. Each contiguous memory word is mapped

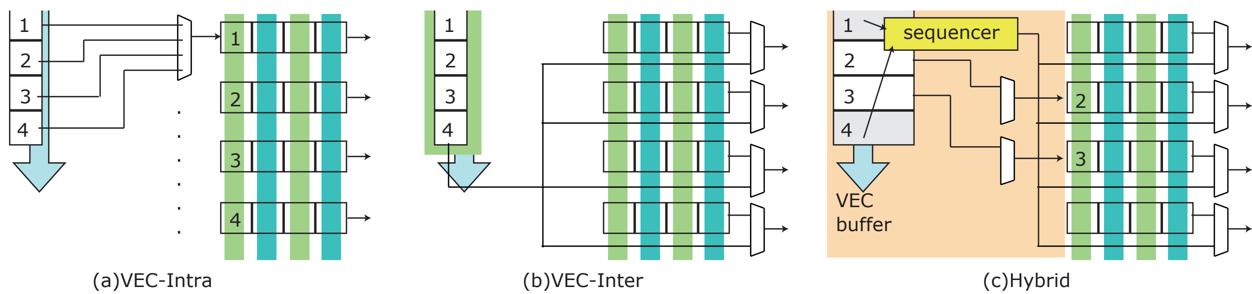


Figure 3.14: Vector port mapping: microarchitecture

to the same port since subsequent invocations use contiguous memory addresses, thus allowing lower latency between multiple pipelined invocations. This pattern is referred to as Inter-invocation (VEC-Inter) communication. The implementation of VEC-Inter communication is straight forward; as shown in Figure 3.14(b), multiplexers are used before the DySER input ports to drain the data in the vector buffer in-place. An alternative implementation is to directly dump all data values into the target queue; the tradeoff here is that it may increase the overhead in examining the available spaces in a DySER input buffer for vectors of various lengths.

- **Hybrid Communication:** Figure 3.13(c) shows a computation sub-region from the TPACF benchmark. The vector buffer has sub-words 1 to 9, which is fed to three different buffers. The pattern is that sub-words 1, 4, 7 have to be fed to the first input port, 2, 5, 8 and 3, 6, 9 to the second and third port. Neither inter-invocation nor intra-invocation is sufficient to perform such a vector communication. As a result, a hybrid method is needed to forward the data. Figure 3.14(c) shows the microarchitecture of this Hybrid communication. In the figure, an example vector insertion with 4 sub-words is shown, where sub-words 1 and 4 have to be fed into the first port, and sub-words 2 and 3 have to be fed into the second and third port, respectively. A sequencer is introduced in the Hybrid communication; first, a sequencer sends data 1, 2, 3 to their destinations via the same logic in used VEC-Intra. Next, this sequencer reads the pre-configured control microcode, moves to the remaining data and distributes them.

3.5 Chapter Summary

Under the DySE model, an application phase is transformed into execute and access components, and the execute component, represented as a program dependence graph, is mapped and executed on DySER. This Chapter details the hardware design goals of DySER, an energy-efficient reconfigurable architecture. It details the microarchitecture, configuration, integration interface and the supporting mechanism for data level parallelism. While first-order analyses of DySER can be found in [57, 59], in the next chapter a full-system prototype of DySER, integrated with OpenSPARC, is discussed to prove the concept of the DySER architecture.

4 SPARC-DYSER PROTOTYPE

Although early stage results from simulation and modeling provide reasonable estimates, performance prototyping on a physical implementation uncovers the fundamental sources of improvement and bottlenecks in a concrete manner. In this chapter we walk through SPARC-DySER, the OpenSPARC T1 integrated with DySER, in terms of the ISA extensions, microarchitectural integration details, and the lessons learned from a practical DySER implementation. This integration by construction proves that DySER can be non-intrusively integrated into a commercial processor with some effort.

4.1 SPARC-DySER Integration

OpenSPARCT1 is an open-source release of Sun's UltraSPARCT1, a highly integrated processor that implements the 64-bit SPARC V9 instruction set. The release contains the full implementation of OpenSPARC T1 (an 8-core chip multiprocessor), as well as a Xilinx FPGA evaluation kit (system and electronic design automation tools, scripts, ready-to-use Xilinx Embedded Development Kit project, and so forth). It is the best off-the-shelf open source processor with a FPGA evaluation kit; this significantly reduces the prototyping time of DySER while providing a practical integration experience.

OpenSPARC T1's pipeline consists of six stages: fetch, thread select, decode, execute, memory and writeback; the major blocks are `ifu` (instruction fetch unit, including decode and thread selection stages), `exu` (execution unit), `lsu` (load-store unit), `ffu` (floating-point unit), `tlu` (trap logic unit), `spu` (special function unit) and few peripheral blocks. We implement DySER as a sub-block in the OpenSPARC T1 Verilog RTL and reverse-engineer the code to integrate DySER into the pipeline. Table 4.1 details the new logic and signals introduced in the number of new lines. In the decode, execute and load-store unit, a total of 432 lines are added into original RTL files. Except rare cases such as replacing the implementation-dependent instruction in OpenSPARC for DySER interface instruction, most of these lines are non-intrusively augmented on top of OpenSPARC's logics. The wrapper module, `sparc_dyser`, is the only new RTL file which we developed from scratch.

File	Description	#New Lines
	Decode RTL files	
sparc_ifu_dec	Instruction decode, added decode logics for DySER interface instructions	72
sparc_ifu_fcl	Fetch control, added DySER stall and roll-back logics	61
sparc_ifu_swl	Thread switch, added DySER stall and stall completion support	38
sparc_ifu_swpla	Long latency instruction control, changing the control of DySER interface instrucitons	6
sparc_ifu_thrcmpl	Thread completion control, added DySER stall and stall completion support	31
sparc_ifu_ifu	Fetch, thread Select, and decode unit's top level	50
	Execute RTL files	
sparc_exu_byp	Bypass muxes, added datapath to DySER	14
sparc_exu_ecl	Execute control, added write and stall request signals	36
sparc_exu_ecl_wb	Bypass Control, added DySER write signals	38
sparc_exu	Top level	38
	Load Store Unit RTL files	
lsu_dctl	Datapath control, added controls for DySER load/store miss	12
lsu	Top level, added datapath between DySER and LSU	36
	SPARC-DySER interface	
sparc_dyser	Retire buffer, flush signals, thread stall/wakeup signals	458

Table 4.1: OpenSPARC RTL modifications (comments included)

The above modifications or additions mainly serve two purposes: (1) to interface DySER with manually optimized or compiled DySE codes; and (2) to incorporate DySER into OpenSPARC. The two following subsections describe them respectively in depth.

4.1.1 Interfacing DySER

Table 4.2 provides a stylized listing of the assembly instructions added for OpenSPARC. Table 4.3 shows the exact instruction encodings. *dyser_init*, *dyser_send*, and *dyser_recv* are encoded via the *impdep2* instruction in the SPARC ISA with bits [7:5] as the DySER instruction opcode. *dyser_load* and *dyser_store* are encoded as SPARC load/store type instructions. The instructions provide a means for the processor and compiler to configure and interact with the DySER block.

***dyser_init*:** Once a *dyser_init* is decoded, DySER is signaled to take the 21 configuration bits from the instruction in the execute stage.

Instruction	Description
dyser_init [config data]	DySER block placed in config mode, and the config data shifted in.
dyser_send RS1 → DI1 *dyser_send* RS1 → DI1, RS2 → DI2	Reads data from the register file and sends the data to a DySER input port. 1 or 2 source registers are sent to the specified DySER input ports.
dyser_recv DO → RD	Write data from DySER output port DO to register RD.
dyser_load [RS] → DI	Read from memory address in register RS and send result to DySER input port DI.
dyser_store DO → [RS]	Writes data from DySER output port DO to memory using the address in register RS.

Table 4.2: A stylized listing of the DySER instructions

Instruction	Instruction encoding							
dyser_init	10	config	110111	config			000	config
dyser_send	10	DI1[4:0]	110111	RS1[4:0]	DI2[4:0]	V	001	RS2[4:0]
dyser_recv	10	DO1[4:0]	110111	RD[4:0]	unused		010	unused
dyser_load	10	DI1[4:0]	000000	RS1[4:0]	0	1000000	000	RS2[4:0]
dyser_store	10	DO1[4:0]	000100	RS1[4:0]	0	1000000	000	RS2[4:0]

Table 4.3: Design of the DySER instruction extensions for OpenSPARC

dyser_send: A *dyser_send* reads up to two values from the register file, specified by RS1 and RS2, and pipelines the data to the DySER input ports denoted by DI1 and DI2 respectively. Setting the 'V' bit to zero sends only the first value.

dyser_recv: A *dyser_recv* sends the results from a DySER output port, specified by DO1, to a register file destination, denoted by RD. It behaves no differently than a normal SPARC arithmetic instruction by pipelining the data and register destination from the execute stage to writeback.

dyser_load: *dyser_load* loads data from memory directly into DySER, using the bits normally restricted for the register destination as the DySER port destination. The source address is specified by r[RS1] + r[RS2]. The *dyser_load* is identical in every way to a normal SPARC load instruction except for bits [10:5], which are reserved for loading from alternate space. If no alternate memory is specified (as is the default case for OpenSPARC), then anything non-zero would throw an illegal instruction exception. We take advantage of this behavior by reserving the most significant alternate space bit for *dyser_load*s and *dyser_store*s and remove them from

being trapped by the illegal instruction logic. The difference between a load and a *dyser_load* is that when the data comes back, it is forwarded to DySER instead of the register file. It is important to note that in neither the *dyser_load* nor *dyser_store* are we allowed to use immediate bits instead of a register operand. Loads with immediate values do not have the reserved bits for alternate space, making it very difficult to decode in our scheme.

***dyser_store*:** As with *dyser_load*s, *dyser_store*s also take advantage of the alternate space bits for decoding. *dyser_store*s behave similar to *dyser_recv*s in that the data coming out of DySER is stored, but in memory. It is also similar to SPARC stores in that the ALU calculates the memory address destination.

4.2 Incorporating DySER into OpenSPARC

It is often straightforward to add an arbitrary accelerator into a processor pipeline and create instruction extensions and a datapath for it; to make the accelerator function correctly with the pipeline, however, is a more delicate task. This task can be extremely difficult when the accelerator design is intrusive, and the processor pipeline is complex. Fortunately, DySER and OpenSPARC can be integrated without prohibitive effort because of the simplicity in the DySER interface.

As previously described, DySER interfaces with processor pipeline via queues. Ideally when executing a specialized program phase, the data can be sent to DySER via these queues with interface instructions. Nevertheless, two situations may break this ideal assumption: (1) a DySER send/receive instruction is issued when the queue is full or the data has not been produced at DySER's output ports yet; and (2) a trap or an exception occurs in the OpenSPARC T1 pipeline. While the former can be naively solved by padding `NOP` (bubble) instructions, it requires the programmer to count DySER's worst case latency manually. Because DySER's internal latency is static for a specific configuration, padding `NOP` instructions is possible in hand-optimized DySE programs. However, requiring the compiler to reason about the hardware pipeline design is over-burdensome. Regarding traps and exceptions, unfortunately, there is no simple solution because the architectural state of OpenSPARC has to be maintained to ensure a correct execution

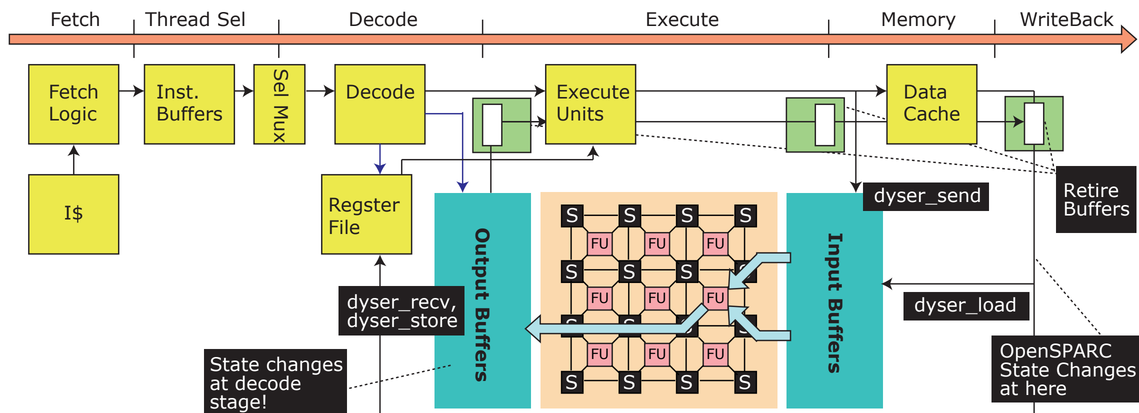


Figure 4.1: SPARC-DySER integration

in any case.

To overcome this issue, we first modify the existing OpenSPARC rollback mechanisms to take DySER into account. When a *dsend* sends a value to DySER, it follows the same execution model of a potentially long latency instruction inside OpenSPARC–it switch out its thread, informing the pipeline to monitor the completion of itself so that the pipeline can switch the thread back in and fetch new instructions. If the queue is full or a trap/exception occurs, the value is flushed, and the pipeline is rolled back.

Next, while the OpenSPARC T1's roll-back mechanism resolves the state change at writeback stage, DySER does not; in addition to the above change in DySER interface instructions, modifications have to be made to the processor pipeline to avoid incorrect state changes. Figure 4.1 details this issue. In the prototype, stateful DySER interface queues are integrated at the decode stage, mimicking the behavior of the register file (values are ready to use in execute stage). This design, however, changes the microarchitectural state of the pipeline because the dequeued values leave storage structures permanently before the instruction commits in the writeback stage; in the case of a flush, the data from DySER is lost. Therefore, we introduce DySER retire buffers at execute, memory, and writeback stages, each 64 bit wide. The retire buffer discards DySER outputs only after all exceptions are resolved. This microarchitectural support not only correctly resolves the traps and exceptions, but also enables compiler efficiency because it does not have to pad `NOP`s to account for DySER internal latency.

Work	Quantitative results	Demonstrated insights
DySER	*Early-stage*: 2.1 × AVG on workloads [57] *Prototype*: improvement on irregular workloads requires further compiler work, 3× compiler, 6 .2× hand,on data-parallel workloads	Dynamic pecialization
TRIPS	∘ 1 IPC in most SPEC benchmarks ∘ best case 6.51 [52]	Dataflow efficiency
RAW	∘ up to 10× on ILP workloads ∘ up to 100× on stream workloads [119]	Tiled architecture
Wave Scalar	∘ 0.8 to 1.6 AIPC on SPEC ∘ 10 to 120 AIPC with mutli-threading [102]	Dataflow efficiency
Imagine	IPC from 17 to 40, GFLOPS from 1.3 to 7.3 [8]	Streaming

Table 4.4: Summary of the performance evaluation works

4.3 Summary and Lessons Learned

The findings of some other prototype evaluations are summarized in Table 4.4. Although quantitative results have sometimes been lower in early stage results because of features eliminated from the prototype compared to the design proposals, the studies have a lasting impact by establishing the fundamental merit of their underlying principles.

In a separate work on the performance evaluation of SPARC-DySER, we also found the same discrepancy; this projected performance gap between simulator model and prototype are mainly because the low-performance of OpenSPARC T1. However, the prototyping tasks, including RTL implementation, verification, FPGA mapping, and manually re-writing/optimizing the code for DySE are proved manageable, and the prototyping of DySER proves that DySER can be non-intrusively integrated to a processor in practice.

A questions left for us to answer is: Can DySER be non-intrusively integrated into a high-performance processor with a more sophisticated processor-DySER interface? Reflecting on the experiences, it would be advantageous to have open-source implementations of high-performance baseline processors reflecting state-of-art designs. Among what is available, OpenRISC [96] and Fabscalar [29] have low performance (OpenRISC's average IPC is 0.2) — and this could impede the prototyping of new accelerator proposals.

5 MEMORY ACCESS DATAFLOW (MAD)

According to the Dynamically Specialized Execution model, specialized phases are divided into access and execute components. Chapter 3 discusses the specialized hardware that supports the execute component, which leverages a reconfigurable network to perform computations. While this execute accelerator can attach to a general purpose pipeline and leverage it for data delivery, the access component of the phase can be performed on a specialized hardware which is optimized for memory access. This chapter presents the Memory Access Dataflow (MAD) architecture for this purpose. The MAD architecture applies the concept of Event-Condition-Action and dataflow execution, and natively supports the dataflow delivery from the cache to the execute accelerator for computation.

This chapter first discusses the design goals and the motivations for an access component specialization. It then describes the MAD ISA, the microarchitecture, and complex scenarios with examples. Finally, this chapter also demonstrates how the MAD architecture can be integrated with different execute component accelerators to show MAD's generality.

5.1 MAD Design Goals

General purpose processors have long thrived in the market, delivering sustainable performance to various applications. Although now many specialization approaches prosper, general purpose processors are still the heart of a microprocessor. In practice, the processor industry has strived to build high-performance out-of-order general purpose processors to respond to various demands in applications, even on power/energy critical platforms [10].

The DySE model assumes a platform that requires (1) a powerful general purpose processor to executes arbitrary applications; and (2) a specialized hardware that optimizes application phases when "profitable"– the profitable phases have recurring and parallel patterns that may exists in any applications. Under such a dynamic framework, DySER is first proposed to specialize the execute component dynamically in a decoupled access/execute fashion, and it leverages the recurring and parallel patterns in the application phase. The access component, which may be executed on host high-performance general purpose hardware, also retains the recurring and

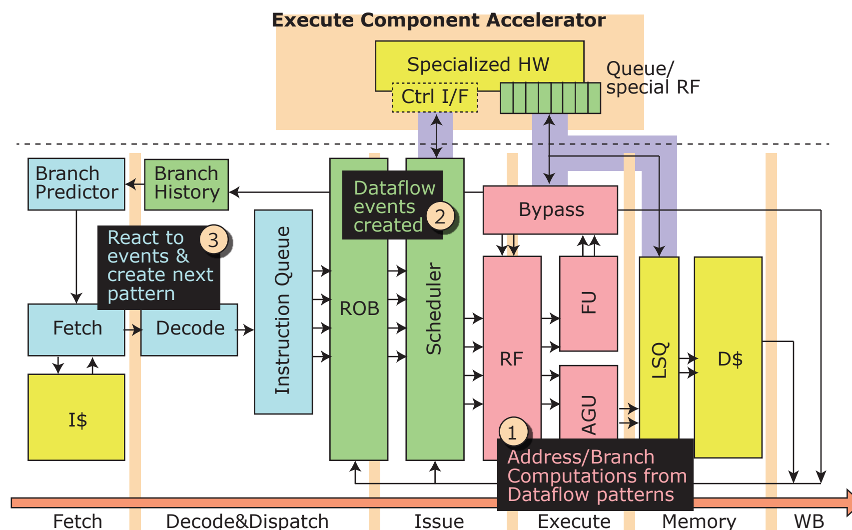


Figure 5.1: The block diagram and the execution of an out-of-order host processor with an accelerator

parallel patterns and can be specialized and optimized with a different hardware. Specifically, there are several design goals to specialize the access component in a profitable application phase:

- The access specialization should provide justifiable performance to deliver data from memory to execute accelerator, in a profitable application phase in an arbitrary application;
- Similar to DySER, the access specialization in the DySE model should minimize the dynamic power overheads in its microarchitecture, leveraging reconfigurable hardware for efficiency.

These two goals suggest that *the access specialization should provide similar, or better, performance to a general purpose processor with lower power*. To enable such an optimization in hardware, we first investigate the general purpose processor that is accessing memory for an execute accelerator under the decoupled access/execute model, as shown in Figure 5.1. This out-of-order general purpose processor is the same as the hypothetical OOO processor described in Section 2.1.2, and is integrated with an execute component accelerator. In a specialized application phase, the processor first follows few dataflow patterns to generate memory addresses. These dataflow patterns are encoded in Von Neumann instruction steam, and the *computations* within are

dynamically captured in the register file, the functional units and address generation units of the OOO pipeline (step 1). Second, the front-end of the processor pipeline and the reorder buffer track the dataflow *events*, which are the arrival of the computation results in OOO pipeline, to drive the program counter and the issue logic (step 2). Last, the scheduler issues the instructions, which can be viewed as the event-driven *actions*, to move the data to/from data cache (ld/st instructions) thus to trigger the computation in the accelerator or the next computation pattern in the host processor (step 3).

5.1.1 Access Component Specialization

To sum up, the basic building blocks of the dataflow patterns in the access component are: address/branch computation, event matching, and action triggering. An access component specialization should be optimized for the above building blocks, in particular:

- **Address and Branch Computation:** The access component specialization should optimize for a few dataflow patterns to compute memory addresses and control branches within an application phase;
- **Dataflow Event:** The access component specialization should provide hardware to store the outcome of the addess and branch computations; and
- **Data-Movement Action:** Based on the stored events, the access component specialization should trigger data movement actions to move data between the accelerator and memory, often a data cache.

Under the DySE model, the above building blocks of the access component in addition follows two principles:

- **Recurrence:** A few dataflow patterns (consist of computations, events and actions) repeat during a specialized phase.
- **Concurrency:** Based on the dataflow dependencies, independent computations, events and actions can be performed in parallel.

The first principle allows *area efficiency*; a reconfigurable architecture can be used to map the computations, events and actions without prohibitive configuration overhead. The second principle, moreover, implies that an access specialization architecture developed with the basic building blocks may achieve similar performance compared to an out-of-order processor if providing *the same amount of parallelism to process computations, events, and actions as an out-of-order processor*. Given these two principles, it is evident that a specialized hardware, which is natively optimized for address/branch computations, dataflow events and data-movement actions, can be built with reconfigurability and parallelism such that it can provide similar performance compared to an out-of-order processor with lower power.

5.2 Memory Access Dataflow Overview

Thus far, we have described the goals of building an access component specialization architecture and identified three basic building blocks in an access component of a specialized application phase. The Memory Access Dataflow (MAD) architecture is one such access component specialization architecture that supports the Dynamic Specialized Execution model. The key insight of the MAD architecture is to expose the three building blocks explicitly in *an ISA that is only used in accelerated phases.* This MAD ISA, consequently, enables the native support for the address/branch computation, dataflow events and data-movement actions in the microarchitecture. Because of the recurrence of the dataflow patterns, and since the total number of the patterns are small, the above tasks can be encoded in the MAD ISA with little complexity. The MAD hardware, which executes the MAD ISA, eliminates most of the dynamism in computations, events and actions and hence is more efficient than an out-of-order processor's power-hungry "general" structures. This section describes the MAD architecture with a simple example, which walks through the ISA transformation, basic building blocks of the microarchitecture, and the integration interface.

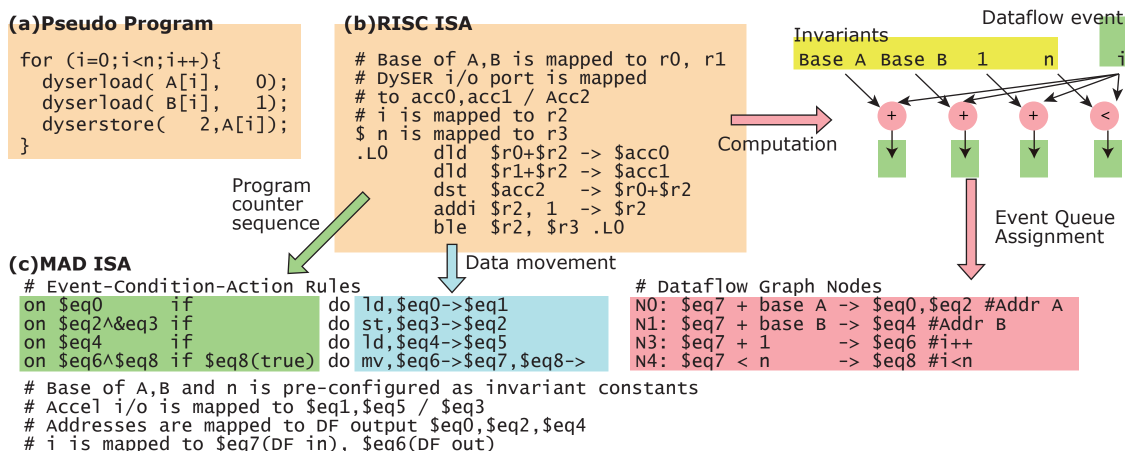


Figure 5.2: The MAD ISA

5.2.1 Encoding the Computations

In the decoupled access/execute model, the access component contains a few computations (sometimes with branches) in order to generate memory addresses. To support the computations and branches inside an access component, MAD leverages dataflow analysis and a reconfigurable hardware. Figure 5.2a shows a pseudo access component program, which has a loop that sends the data in array A and B to DySER. This pseudo program is re-written into a stylized hypothetical RISC ISA as shown in 5.2b. In this RISC ISA, the base address of array A and B is mapped to register `$r0` and `$r1`; the DySER interface port is mapped to `acc0`, `acc1`, and `acc2`; and the induction variable `i` and iteration number `n` is mapped to register `$r2` and `$r3`. The RISC program contains five instructions: two loads to the accelerator, one store to the cache, a loop counter increment instruction and a branch.

By observation, computations in this access component code can be found in the two load instructions (computing address), `addi` (induction variable), store (address) and a branch comparison. The MAD architecture encodes these computations in dataflow graphs, as shown in Figure 5.2c. The input of dataflow graph nodes can be (1) phase invariants, which does not change during an application phase and can be pre-configured to hardware as constants at runtime; and (2) Dataflow events, which are dynamically generated as events during the execution of MAD. In the MAD ISA, the former is encoded as invariant constants (similar to immediate values in a RISC ISA), and the latter is assigned to named *event queues*. These event

bench- mark	Insts. (#)	Ld/St (%)	Acc. I/F (%)	Exec. Comp. (%)	SP/Off. (%)	Base/Off. (%)	Base/Index/. Off. (%)	Base/Index/ Width/Off. (%)
radar	33	15.2	51.5	33.3	40	20	40	0
nbody	73	24.7	42.5	32.9	27.8	50	22.2	0
stencil	555	33.7	47.8	18.6	73.8	5.9	15.0	5.4
fft	162	37.7	44.4	17.9	77.1	16.4	0	6.6
cutcp	207	38.7	53.6	7.7	87.5	7.5	1.3	3.8
kmeans	253	20.6	71.5	7.9	38.5	38.5	13.5	9.6
Avg.(All)	254.4	24.5	40.1	35.4	57.8	31.3	6.4	4.5

Table 5.1: Dataflow patterns: Exec Comp.: Execute component, Acc. I/F: Accelerator interface instructions in the access component code, SP: Stack pointer, Off.: Offset, Base: Base address

queues serve as an interface and the storage between computation microarchitecture in MAD, the accelerator, and the cache; they are also used in the Event-Condition-Action (ECA) rules which are explained later.

The few recurring computation patterns in the access component of a specialized phase are, in fact, small in size. Table 5.1 shows quantitative characteristics of the access component code for representative benchmarks and average for all. The second column shows the total instructions in the phase (e.g., multiple basic blocks that may include a nested loop) in the original code; the access component is shown in column 3 and 4, in terms of load/store instructions and accelerator interface instructions; the execute components are shown in the fifth column in percentage of the original code. The patterns, classified in the combination of memory address primitives (stack pointer, base address, offset, index, width), are listed in column 6 to 9. Here, the main point is that a few patterns dominate, but are not trivial either because of the computations of these primitives may vary. Thus, a coarse-grain reconfigurable/switched fabric can be used to map these computations. The MAD ISA describes the nodes in the graph of Dataflow-Computation in 3-tuples: source nodes, the operation, and destination nodes. This node description is analogous to the computations and registers in a von Neumann ISA. For example, the `addi` instruction in RISC ISA becomes node `N3: $eq7 + 1 -> $eq6`.

5.2.2 Event-Condition-Action in MAD

The concept of event-driven action has been developed in many settings. For example, *Event-Action-Condition (ECA)* is used in active databases, network and workflow management [101, 89, 26, 93]. The Memory Access Dataflow architecture leverages the same concept to encode the dataflow events and data-movement actions in the MAD ISA. This section describes the syntax of events and actions in the MAD ISA in terms of Event-Action-Condition.

An ECA rule in the MAD ISA has the following syntax:

on *event* **if** *conditions* **do** *actions*

The *event* defines when a rule has to be triggered; a triggered rule examines the current state and matches it with the *condition*; and *actions* are fired if the condition holds. Executing a rule may fire the action that in turn triggers another rule. In the MAD ISA, there is an end of program rule that triggers an action that finishes the specialized phase and gives the control back to the host processor to continue the application.

Events The event in a MAD ECA rule is a combination of primitive dataflow events, which is *the arrival of data*; the MAD architecture use *queues* to hold such a primitive event. Each of these queues, called *event queue*, indicates a primitive "data arrived" event when it is not empty. Primitive events can be combined with event algebra [53, 16]. While there are many common operators in a traditional event algebra, the MAD ISA only adopts conjunction operator ($\wedge$), which can be naturally implemented with and gates in hardware.

Conditions The conditions in a MAD ECA rule specifies the additional states with the triggering events that have to be satisfied to drive an action. In the MAD architecture, these states are coupled to a primitive event, inferring the result of a conditional branch—true or false. A MAD ECA rule may not need to examine the state of the event; the primitive event itself fulfills the driving condition. In such a case, the condition part of the rule is empty.

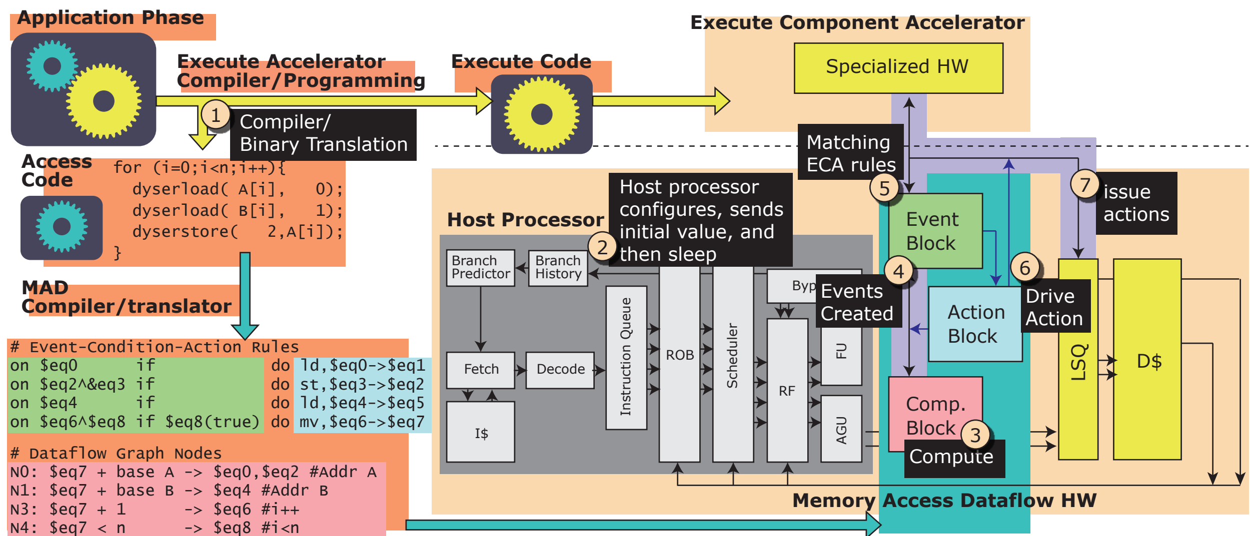


Figure 5.3: An example execution of the MAD hardware

Actions The action part in a MAD ECA rule specifies data-movement actions, which includes loads, stores, or moves between event queues. These actions pop or push data from event queues and hence create new “data arrived“ primitive events if there is new data in the queue output.

MAD ECA Rules Example As shown in Figure 5.2c, parts of the RISC instructions are translated into ECA rules. The Event-Condition and Action are separately color coded in green and blue, respectively. Events are described with named event queues, and the condition is the states with event queues. The ECA rules of load and store actions are often triggered by the dataflow primitive events on memory addresses (and data in cases of stores). The branch instruction (`ble $r2, $r3, .L0`) in the program, however, includes a non-empty condition part; this condition part examines the state of the 8th event queue (`$eq8`), and drives the action when the state is true. The action moves the induction variable for the next loop condition check and pops the 8th event queue (and discard it) since the value is no longer needed. A detailed description of the encoding of the ECA rules can be found in Appendix A.

5.2.3 Generating the MAD Configuration

Generating the MAD configuration bits in the MAD ISA can be done in a co-designed compiler or a binary translator. After manual or compiler-assisted DySE (described in Chpater 2), the

compiler can produce the MAD configuration in the machine code generation pass using its internal representation of the access component. In our work, the MAD configuration generation is done in a binary translator, which follows the steps below:

- first, it scans the instructions and constructs the program dependence graph from the instructions, which are the nodes in the graph;
- second, it breaks the graph into sub-graphs and map computations into the computation block;
- third, it assigns the leaf nodes in the subgraphs may either be a load/store action, a move action (moving data between event queues), or a branch output that triggers different actions based on the outcome of the branch; and
- last, it prioritizes the above actions based on the program order, creating ECA rules for each action.

While compiling from source code may enable more opportunities for optimization, we use a binary translator to evaluate the first-order characteristics of the MAD architecture.

5.2.4 Executing Memory Access Dataflow

The MAD hardware natively supports the three basic building blocks (computations, events and actions) in the MAD ISA. Figure 5.3 shows a sketch of the MAD architecture with an execution example labeled with sequence numbers. In this hypothetical integration, an execute component accelerator and a MAD implementation are coupled to a general purpose host processor. The application, divided into phases, is regionally specialized (e.g., only profitable phases are specialized) via the compiler/programmer of the execute accelerator, and then a MAD compiler/binary translator. This MAD compiler/translator only works after all the access binaries or internal representations are generated, thus it is decoupled from a specific accelerator and can support different execute accelerators (described in Section 5.5).

Following the DySE model, the host processor first configures the execute accelerator and the MAD hardware before entering a specialized application phase. Second, in addition to

the configuration microcode that sets up the events, actions and computations, the host also sends initial events to start MAD execution. Third, the processor goes to sleep mode and turns on the MAD hardware, which has three basic building blocks: a computation block, an event block, and an action block; each of the building blocks corresponds to a part of the MAD ISA. The computation block has a reconfigurable network with functional units that performs the computations; the event block contains event queues that serves as the I/O interface of the computation block and the execute accelerator; and the action blocks prioritizes the triggered actions and drives them when there is available bus bandwidth.

Fourth, the initial events trigger the computations in the computation block or accelerator; the execution model of event block to computation block/execute accelerator triggering is pure static dataflow, which means that whenever a ready data appears at the input queues of computation block/execute accelerator, the computation is triggered. Based on the outcome of the computation, a new set of events are created in parallel at the event block. Fifth, the event block matches these primitive events and conditions with pre-configured Event-Condition-Action (ECA) rules, and then generates the actions indices that should be driven to the action block. In step 6 and 7, the action block takes the indices, selects the ones that could be issued in the current cycle, and controls the data bus to move the data values to accelerator or between event queues. This whole cycle then repeats itself and stops when the ”end of program” action is triggered, which is the branch evaluation in this case. Finally, the MAD hardware wakes up the host processor to continue the application, and through memory passes all the architectural state changes during acceleration to the host processor.

5.3 MAD Microarchitecture

Figure 5.4 presents the detailed microarchitecture of the MAD hardware, which consists of three blocks.

- **Computation Block:** The computation block executes the dataflow graph to generate the memory addresses and branches with a fine grain (tens of RISC instructions) reconfigurable fabric.

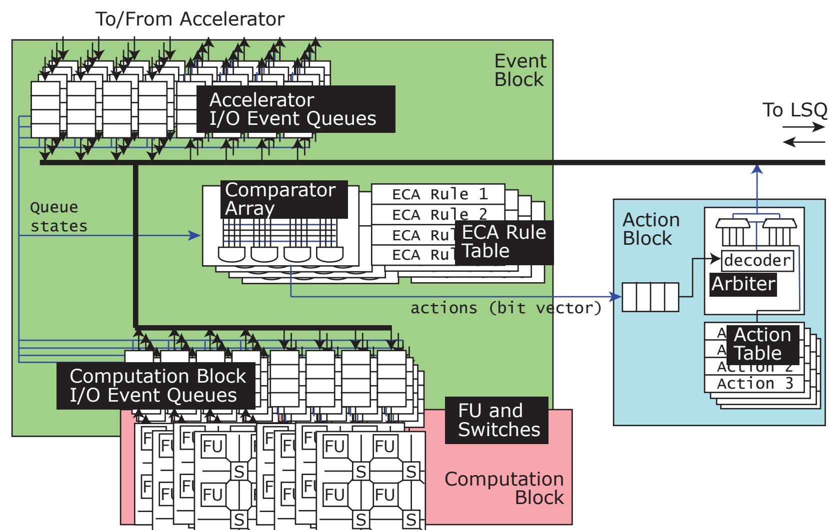


Figure 5.4: MAD microarchitecture

- **Event Block:** The event block includes event queues to hold dataflow events. It also stores ECA rules and uses comparators to check if a rule can be triggered and if the conditions are satisfied. It produces results in the form of action indices.

- **Action Block:** The action block receives and grants the action indices from the event block based on priority; it then drives the interconnect bus to move data between event queues and cache.

MAD execution is similar to a dataflow machine: it assumes the execute accelerator automatically executes when the required data arrives and produces data to pre-designated event queues. The execution of address computations and accelerated code can be seen as static dataflow; the triggering of ECA rules, however, is dynamic dataflow but follows the "dataflow order" in the program (enforced by ordered queues). This dataflow execution results in a weak memory consistency model. It may reorder the reads and writes to different memory address locations and thus break the memory consistency model of the host processor. For example, using MAD in SPARC architecture breaks SPARC's TSO model.

5.3.1 Computation Block

The computations block computes the pre-configured dataflow graph statically with functional units (responsible for compute operators) and switches (responsible for forming the edges in a dataflow graph). Figure 5.4 outlines a 16-functional unit computation block. In this design, four function units and four switches constructs a cluster, and the clusters are interfaced with event queues to talk to the cache, accelerator, or between clusters themselves. Since it performs static dataflow execution, no dynamic popping (pushing) of the input (output) event queues is necessary; all I/O input event queues are used immediately when available.

The switches and functional units are the same as in DySER. They implement the same flow-control and synchronization stages such that the input dataflow events can occur at any time (e.g., it is not required that all inputs are prepared before invoking a cluster; the fabric will automatically buffer and synchronize the data when ready)

An alternative implementation for the computation block is the BERET-like Subgraph Execution Block [63], which leverages conventional centralized register file and write-back bus with the reconfigurable functional unit clusters.

5.3.2 Event Block

The event block connects the datapath between the cache, the computation block and the execute accelerator. As shown in Figure 5.4, it has the following sub-blocks:

- **Accelerator I/O event queues:** It is required that the execute component accelerator is interfaced with event queues in an Accelerator/MAD integration. MAD assumes a data-driven execute component accelerator, which consumes data as soon as they appear at the input event queue, and produces results to the output event queues. The integration of execute accelerators is described later in Section 5.5.

- **Computation I/O event queues:** As mentioned, the clusters in the computation block are interfaced with event queues. Same as the execute accelerator I/O queues, the computation block consumes the input event queues when there exists available data and produces results to output event queues.

- **ECA rule table:** The ECA rules in the translated program are stored in the ECA rule table, and are compared to the state of event queues to trigger actions.

- **Comparator array:** The event block employs a comparator array to match the dataflow events and states from event queues to pre-configured ECA rules. The comparator array takes 3 types of inputs: (1) the pre-configured ECA rules from the ECA rule table; (2) the not-empty state of the event queues from the controller of the queues; (3) the state of the first output data in an event queue. The last input, representing the state (true or false) of the corresponding dataflow event, is compared with the condition part in an ECA rule. In our design, a non-zero value in the event queue represents the true state. The comparator compares the (1) ECA rules to the (2) event and (3) condition states and outputs a bit-vector, which represent the indices of triggered actions.

The triggering of the event-condition and the execution of the action is decoupled; the triggering event queues in a satisfied ECA rule turn into an inactive state before the values are popped by actions. This strategy of deferring the driving of the actions reduces the timing and bandwidth requirement over an approach that always matches events and drives actions in the same cycle.

5.3.3 Action Block

The last building block in Figure 5.4 is the action block; it is responsible for controlling the data movement between event queues and the load-store queue of the host processor. The actions in MAD ECA rules are stored in an action table; they are indexed by the action index bit vector (received from the event block). When the triggered actions arrive as a bit vector, the action block buffers them and decodes them with a priority decoder, which arbitrates the buffered actions. If there are more triggered actions than the bandwidth of the data bus, actions with higher priority are issued first.

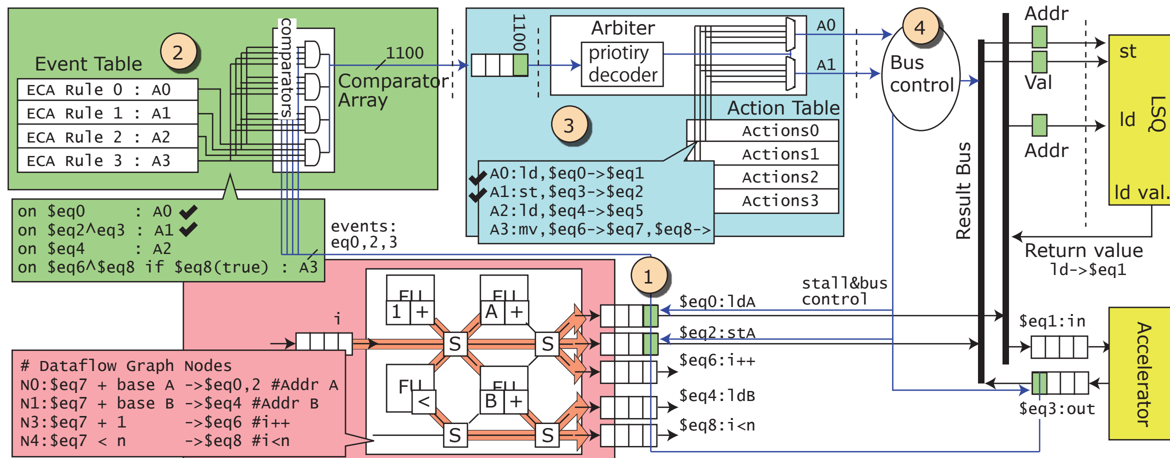


Figure 5.5: Detailed MAD execution with a simple code

5.3.4 Microarchitectural Execution Flow

Carrying the same pseudo program, Figure 5.5 shows the MAD execution of the occurrence of two events, the matching two ECA rules, and the driving of two actions. Before the execution, the processor configures the 3 blocks of the MAD hardware and fills configurations as shown in the colored dialogue boxes. For illustration purposes, we simplify the drawing to present only the activated microarchitecture logics. The example execution flow begins with three primitive dataflow events: there exists data in event queue 0, 2, and 3. Since there no conditions need to be evaluated in these two rules, these events directly triggers two actions, action 0 and 1, as in the action index vector (step 2). Reaching the action index buffer, the priority decoder decodes the bit-vector, and outputs the two actions, `ld,$eq0->$eq1` and `st,$eq3->$eq2`, to the bus controller (step 3). Finally, in step 4, the bus controller pops the event queues and moves the data values to D$.

5.3.5 Implementation and Design Decisions

We implemented the MAD hardware in Verilog RTL and synthesized it with a TSMC 55nm library. This section discusses several design decisions. First, one can intuitively implement the interconnections between event queues and comparator array as a fully connected network; this implementation, however, may cause timing problems and is over-designed. We observed that, in a typical program, the ECA rules can be divided into disjoint sets, and the hardware (event

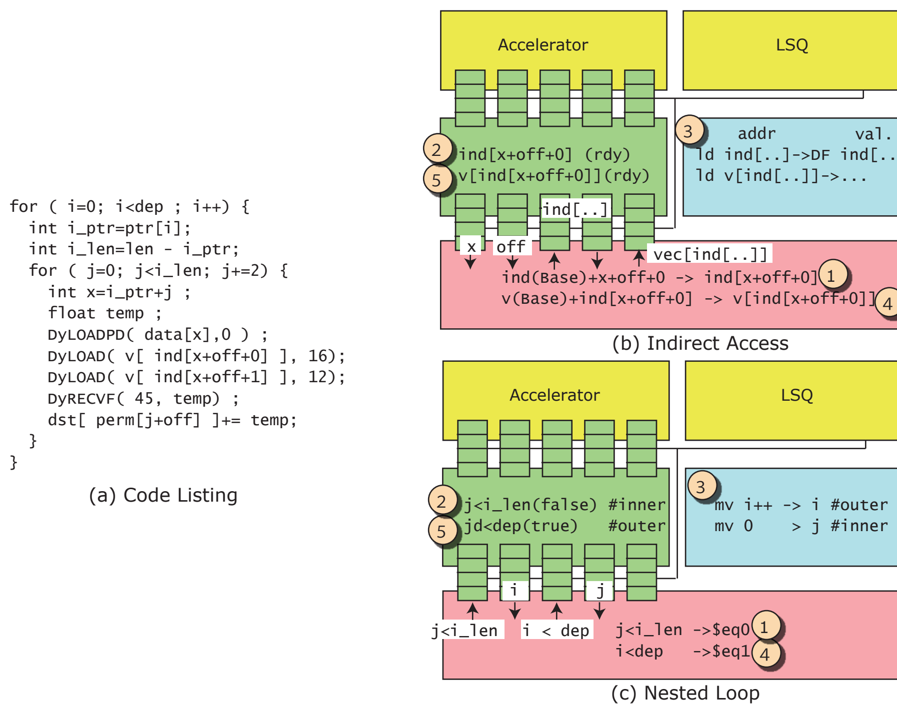


Figure 5.6: MAD execution with a complex example

queues and comparators) can be clustered.

Second, the number of functional units in the computation block determines the maximum available computational parallelism in the MAD hardware. The implementation details of the three blocks are:

- The computation block has 32 integer ALU and 4 integer multipliers, supports up to 256 dataflow graph nodes; each cluster (4 ALU, 1 or no multiplier) can switch between different configurations stored in a config cache.

- The event block has 32+32 event queues (computation block + execute accelerator); they clustered in to 8 groups and support up to 256 ECA rules. An ECA rule can use at most 4 primitive events and 4 condition states.

- The action block has four 8-wide 32 bit action index buffers, each decodes 2 actions in one cycle; however, the load and store action bandwidth are still limited by the number of data cache ports in the host processor.

The runtime dynamic power of our implementation is around 1 Watts @ 1GHz with the selected benchmarks, which is equivalent to one-third of the power of a 2-issue out-of-order processor (a highly optimistic OOO setting). Last, the total configuration bits in the implementation is close to 1.5KB.

5.4 Complex Scenarios

To demonstrate MAD's generality, several complex scenarios are discussed here: (1) memory disambiguation; (2) indirect memory access, which creates irregular offset addresses for a base; and (3) nested loops. Both are ubiquitous in data structure traversal. Sparse Matrix-Dense Vector Multiplication (`spmv`) from the Parboil suite [6] is used as the example.

Figure 5.6a shows the code of matrix-multiply accelerated with DySER accelerator (variable names modified for readability). In the example, DySER is performing blocked matrix multiply based on `i` and `j` indices. The `DyLOADPD` and `DyRECVF` are the DySER instructions (ISA extensions produced by DySER compiler for host) which load from the data cache to DySER ports and receive from DySER ports to the MAD hardware, respectively. The MAD compiler compiles these instructions as ECA rules where the actions load and receive data.

5.4.1 Memory Disambiguation

In an out-of-order processor, the pipeline allows out-of-order execution and thus requires memory disambiguation and in-order retirement to preserve the sequential semantics. Since the MAD architecture executes in dataflow order, it could allow independent instructions from a sequential program to execute "out-of-order" in terms of its sequential semantics. In the example code snippet, the two loads are independent and can be executed in any order. Also, the computation of the induction variable `j` in any order does not depend on the outcome of the load-accelerator-store dataflow chain; they can run ahead to trigger a new set of load-accelerator-store computation when the previous iteration is not finished because of a store miss. While the dataflow order may increase the performance, may-alias memory access between independent dataflow may cause conflicts. In particular:

- The MAD hardware needs memory disambiguation for may-alias memory accesses, which is provided by the load-store queue. The index (time sequence) for disambiguation is the action index with the iteration number from the event queues.
- When the memory dependence predictor predicts wrong (the may-alias memory accesses alias and conflict), the MAD hardware has to roll-back to the correct state before mis-speculation.

The latter case implies that the MAD hardware has to support checkpointing; this is done with additional state bits in each event queue and ECA rule table. If the MAD execution reaches a load that may-alias with a previous store, it sets the destination event queue entry into speculative state. The speculative state propagates to triggered ECA rules and computation block. If mis-speculated, the MAD hardware flushes all the incorrect values in the event queues (and also the computation block that consumes this mis-speculated value) by walking through the event block. If a speculative action targets an event queue that already contains speculative data, the execution stalls; this simplifies the hardware design by preventing overlapped speculation. Merging the LSQ and MAD, or more aggressive speculative execution in MAD could potentially increase the energy efficiency and are future work.

5.4.2 Indirect Access

The indirect access that occurs in the following code of `spmv`: `DyLOAD(v[ind[x+off+0]],16);` is shown in Figure 5.6b. The ECA rule table, the action table, and the dataflow graph nodes in the computation block are labeled with sequence numbers and the same color code as previous figures. For illustration purpose, stylized ECA rules and computations (as dataflow graph nodes) are used. First, the computation block computes the inner offset (step 1: `ind(Base)+x+off+0`, and creates a dataflow event (step 2) that triggers the first load action (step 3). Next, the action block loads the data of `ind[..]` from D$ back to the computation block for the second base and offset computation (step 4). Finally, the address of `v[ind[..]]` is ready (step 5) and triggers the indirect load.

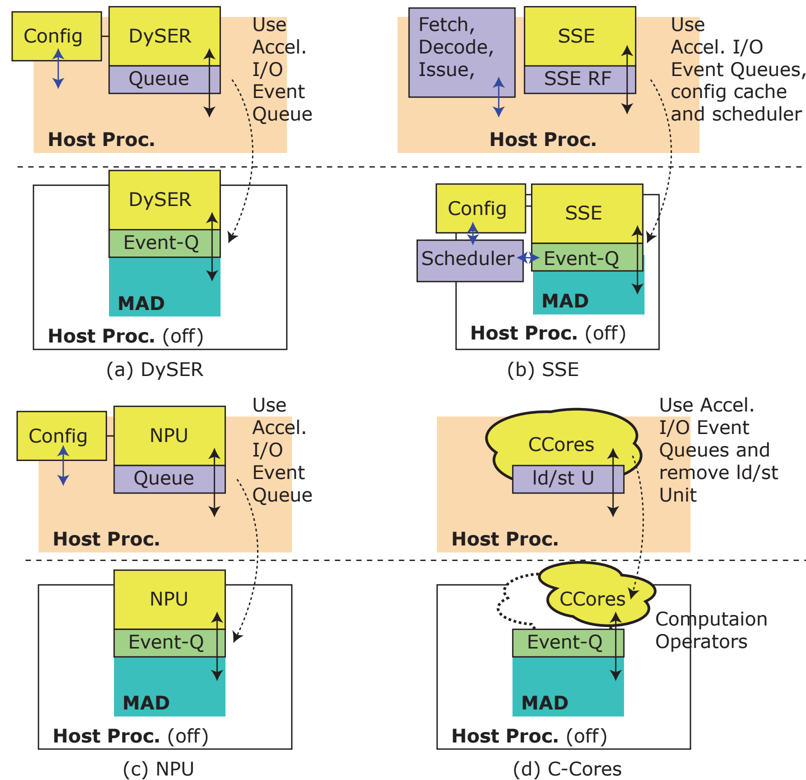


Figure 5.7: The integration of accelerators and the MAD hardware

5.4.3 Nested Loops

The two-level loop in spmv iterates on two loop indices, `i` and `j`. Starting from the end of the inner loop, Figure 5.6c shows the execution of the nested loops. First, the computation of inner loop condition is completed (step 1) and creates a "false" state on the event queue (step 2), which drives the action of moving the incremented outer loop index `i++` for next computation (step 3). This action pops the data value from the output of the computation block back to its input event queue. Next, the computation block computes the outer loop condition `i<dep` (step 4) and creates a "true" state on the event (step 5). Finally, this event and the “true“ state matches the ECA rule and triggers the action of sending an initial value 0 for the next inner loop iteration, such that the new inner loop condition computation can be executed.

5.5 Integration

The flexibility of the DySE model allows different hardware to assume the role of the execute component, where MAD can be used to drive these different hardwares. We discuss the integration between the MAD architecture, DySER and other accelerators using three representative cases to demonstrate MAD's generality/diversity. In particular:

- **Intel SSE:** The SSE instruction set extension represents the performance-oriented vector accelerator and demonstrates "practical/industry" relevance; it is widely used in various applications.
- **Neural Processing Unit (NPU):** Esmaeilzadeh et al. [45] built a multi-layer perceptron artificial neural network accelerator to accelerate approximated application regions generated from NPU compiler; it is a representative domain-specialized accelerator that synergistically combine novel algorithmic and micro-architecture ideas to deliver acceleration.
- **Conversation Cores (C-Cores):** Venkatesh et al. [122] developed a compiler to ASIC approach towards an extreme power efficiency; Conversation Cores hardware are directly generated from compiler and used with in-order processors for low power platforms. It is representative of hard-ASIC acceleration.

In the description of the MAD architecture, we include one part of the execute hardware–the accelerator I/O event queues. This fundamental assumption of MAD implies two integration details: (1) the supported execute accelerators have to use the accelerator I/O event queues (described in Section 5.3) as their interfaces to the memory; and (2) the supported execute accelerators may have to be modified such that they are data driven, where the data from these I/O event queues trigger accelerators' execution. If the above integration implications can be satisfied, MAD can be used to drive any given accelerators that follw the DySE or DAE model. Below, we describe how DySER, SSE, NPU and C-Cores can be integrated into the DySE model with the MAD hardware. Most of the integration is fairly straight-forward as shown in Figure 5.7 — top half is original design from literature and bottom is the MAD integration.

DySER (Figure 5.7a) Since DySER is naturally interfaced with queues, the accelerator I/O events queues in MAD can natively support DySER to drive its execution.

SSE (Figure 5.7b) In order to drive a SIMD accelerator such as SSE unit, two modifications have to be made: (1) the input operands of the SIMD unit have to be changed from specialized vector register to the accelerator I/O event queues in MAD; and (2) the front-end processor pipeline that decodes SSE instructions for driving the SSE unit is now replaced with a small configuration cache and a reconfigurable scheduler. The configuration cache in the latter modification stores all the instructions inside an application phase, and the scheduler checks the operand readiness from event queues to issue SSE instructions to the SSE unit. Note that with this change, in the software side the programmer/SSE compiler has to identify the application phase and provide a configuration mechanism to set up the configuration cache and scheduler.

NPU (Figure 5.7c) Similar to DySER, NPU is interfaced with queues and can be naturally driven with the accelerator I/O event queues in MAD.

C-Cores (Figure 5.7d) Conservation Cores, focusing on power efficiency, encompass non-reconfigurable hard datapath to issue memory requests in-order for its computation operators (i.e., functional units). To apply the MAD architecture, we remove all memory related datapath (including address computation) and replace them with the accelerator I/O queues; this replacement creates many small C-Core computation operator clusters, where each of them is attached to some subset of accelerator I/O event queues for I/O operands. The benefit of MAD integration is that it allows higher throughput delivery of memory to C-Cores compared to the original "power-efficient extremely" design.

5.5.1 Stand-Alone MAD

Often, in some applications there exist phases that are only responsible for accessing and preparing data. In such a case, the computation to memory operation ratio is considerably low, and the phase may have no execute component (all computation is for memory accesses) or the execute component may only have a few computation operators. If the phase is frequently

used, MAD can be applied with no execute accelerator. This stand-alone MAD configuration is similar to the original design, although the event block has no accelerator I/O interface queues. We evaluate this configuration in the end of Chapter 7 with several memory intensive microbenchmarks.

5.6 Chapter Summary

This chapter presented the MAD architecture that natively supports the three basic building blocks in the access component of a specialized application phase. We discussed the motivation, design goals, ISA and microarchitecture of MAD. In summary, MAD exposes the computation patterns, the triggering events, and the data-movement actions within a specialized application phase in a low-level ISA, which enables a native support in hardware that does not require power-hungry structures in a general purpose out-of-order pipeline.

This chapter demonstrates MAD's feasibility and generality via execution examples and provides integration details for four different execute component accelerators. Specifically, we discussed memory disambiguation, nested loop and indirect access for feasibility; and presented the integration of DySER, SSE, NPU and C-Cores for MAD's generality.

6 EVALUATION METHODOLOGY

This chapter discusses the evaluation methodology that evaluates the Dynamically Specialized Execution Model and its supporting architectures, DySER and MAD. Since these two architectures supports the two decoupled components in an application phase, the evaluation is done in an incremental approach to detail the contribution of each supporting architecture. To this end, four architectural configurations are used in the DySE model. In particular:

- **Host(Access)/DySER(Execute):** In this configuration, the DySER is used to perform the execute component, and is integrated into a general purpose out-of-order processor pipeline that is responsible for executing the access component of a specialized phase.
- **MAD(Access)/DySER(Execute):** This configuration exploits the full potential of the DySE model with DySER and MAD.
- **MAD(Access)/SSE,NPU or C-Cores(Execute):** The DySE model can be flexibly applied to many execute accelerators other than DySER; while the efficiency varies, the MAD architecture can be non-intrusively integrated with different execute accelerators.
- **MAD(Access-only):** Sometimes, an application may include phases that merely accesses data and do very few computations. In such a case, the MAD architecture can be solely used for power efficiency.

This Chapter describes the architecture models (Section 6.1), the benchmarks (Section 6.2), and the evaluation methodology to evaluate the above configurations (Section 6.3).

6.1 Architectural Models

The DySE model assumes a general purpose host processor. We modeled our x86 out-of-order architectures as the host processor in gem5 [19], a cycle-accurate simulator that is parameterized and offers processor models at various level and also with different ISA. The x86 out-of-order processor model in gem5 has fetch, decode, rename, issue, execute, writeback and commit stages; it models a tournament branch predictor with 4K branch target buffer and a return

Parameters	Dual-issue OoO	4-issue OoO
Fetch, Decode, Issue, and Writeback Width	2	4
ROB Size	40	168
Scheduler (issue queue)	32	54
Register File (int/fp)	56/56	160/144
LSQ (ld/st)	10/16	64/36
DCache Ports	1(r/w)	2(r/w)
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	

Table 6.1: General purpose host processor models

Architectures	Configuration	Models	
GP Host	x86 out-of-order	gem5	Used in DySER+host configuration, also as baseline
DySER	64 functional units	gem5, RTL	Used in DySER+host and DySER+MAD
SSE	SSE, SSE2, SSE3	gem5	Used in SSE+host and SSE+MAD
NPU	8 Processing Elements [45]	gem5	used in NPU+host and NPU+MAD
C-Cores	ASIC	gem5, RTL	Used in C-Cores+host and C-Cores+MAD
MAD	256 DFG Nodes, 256 ECA rules	gem5, RTL	Used in MAD+DySER, SSE, NPU and C-Cores, also in MAD-only configuration

Table 6.2: Architectural models

address stack, a reorder buffer, an issue queue (scheduler), a load-store queue with memory dependence prediction using store sets. Two different host design points are modeled: a low power dual-issue out-of-order processor and a 4-issue high-performance out-of-order processor. The former is similar to the state-of-the-art low power OoOs such as Intel Silvermont or AMD Bobcat [7, 65], and the latter is similar to desktop processors such as Intel Sandy Bridge [5]. Table 6.1 details the microarchitectural parameters.

We implemented DySER, MAD and few other accelerators (SSE, NPU, C-Cores) in our general purpose code model in simulator, Table 6.2 summarizes the architectural models; each of them is explained below.

6.1.1 DySER

The DySER architecture model is developed and integrated into the gem5-based host general purpose processor model. It models the DySER network, switches and functional units and integration details. Two major changes are made in the host processor model to integrate DySER: (1) the decode stage, which is extended to support DySER interface instructions; and (2) the issue stage, which needs to be modified to correctly schedule DySER interface instructions, especially in the case of mis-predictions and exceptions. The scheduling policy of DySER interface instructions in our model is implemented as described in Section 3.4. The DySER ports are exposed and are viewed as separate hardware resources; the status of ports are examined before issuing the interface instructions. When there is a `dysersend` to full port or a `dyserrecv` on empty port, the instruction is stalled due to a structural hazard (the instructions to other ports can still be issued freely). A separate log buffer (as mentioned in Section 3.4.2)is used to record DySER interface instructions such that when using the out-of-order host, DySER can be flushed and rolled-back to a correct state if mis-prediction occurs.

Two variants of DySER is modeled for floating-point and integer workloads: (1) a 64-functional unit DySER with 16 INT ALUs, 16 FP ALUs, 12 INT multipliers, 12 FP multipliers, 4 FP dividers, and 4 FP square root functions; and (2) a 64-functional unit DySER with 40 INT adders, 16 INT comparator and shifters, and 8 INT multipliers. While they have different power and area, these two configuration of DySER are comparable to an Intel AVX unit [58].

RTL Model Part of the DySER RTL implementation mentioned in Chapter 4 is also used in the evaluations of this dissertation. Unlike the prototyping and proof-of-concept paper [18], this dissertation focus on the analysis on energy efficiency and architectural trade-offs; hence, the DySER RTL model here is not integrated on SPARC but serves as a stand-alone model which supports the gem5 based simulator model and provides power estimates. We create testbenches from the trace of gem5 simulations and run these testbenches with Synopsys VCS simulator for SAIF (Switching Activity Interchange Format) files. These SAIF files are then fed into Synopsys Design Compiler along with DySER Verilog RTL to collect power numbers for the TSMC 55 nm standard cell library.

While the gem5 trace generation is based on the DySER model in the simulator, we have verified our DySER simulator model with the RTL model regarding performance, architectural events, and power. This calibration compared the execution traces of both models, and helped us to correct any discrepancies in interface mechanisms, latencies, protocols and so forth.

6.1.2 Other Execute Accelerators

As described in Section 5.5, many accelerators naturally lend them self to the decoupled access/execute model or the DySE model. Three execute accelerator models are built in gem5, which are the SSE unit [4], NPU [45] and C-Cores [122]. Intel SSE instruction set extensions, including SSE, SSE2, and SSE3 are built-in instruction in gem5; the neural processing unit (NPU) and Conservation Cores are the simulator models we built in-house. We have tested them to verify the correctness and accuracy when comparing to the published results. For C-Cores, RTL models are built to acquire a better understanding of C-Cores ASIC's (Application-Specific Integrated Circuit) power behavior; similar to DySER, this RTL model takes a gem5 trace as input and produce power/energy estimates.

6.1.3 MAD

The Memory Access Dataflow architecture model was built in the gem5 simulator, which uses a binary translator to translate the x86 binary from the host processor to the MAD hardware model. RTL implementation of the computation block, the event block and the action block are also built for energy estimates. The details of our implementation and the simulator model are previously described in Section 5.3.5 and are also summarized in Table 6.2.

6.2 Benchmarks

An ideal application phase for the DySE model has sufficient computation that can be executed in accelerators; this dissertation selects benchmarks from Parboil [6], Intel Throughput Kernels [110] and Rodinia [25] benchmarks. The reason behind the selection of benchmarks are: (1) they have to be complex enough and representative of emerging or real workloads; (2) they have to be

reasonably large, such that re-programming and targeting different accelerators are feasible. Efforts have been made to identify meaningful application phases in these benchmarks and re-program them into accelerator configuration and access component codes. These benchmarks are referred as "base benchmarks" as shown in Table 6.3. The configurations of Host/DySER, MAD/DySER, MAD/SSE and MAD/C-Cores uses these benchmarks. In our implementation, the specialized phases sometimes involve changes in the algorithm. For example, software pipelining may be used to optimize the access component code such that DySER can be utilized better. These optimizations, however, can only be used on specific accelerators and hence are considered as specialization features instead of additional improvements over the original codes.

In the NPU/MAD configuration, we use NPU specific benchmarks [45] instead of implementing neural network approximation in our base benchmarks. These benchmarks are reported to be efficient when applying neural approximation, sufficiently utilizing the NPU hardware. The NPU benchmarks are shown in the second part of Table 6.3.

Last, in order to evaluate the MAD only configuration, a few benchmarks are selected for their access-only phases. Three micro-benchmarks are used to represent common data structure traversal phases in real workloads, and two SPECINT [115] benchmarks are selected as examples of access-only phases in legacy applications. These benchmarks are referred as "Access-only Benchmarks."

All the benchmarks are compiled with GNU GCC with full optimizations (option -O3). We annotate the benchmarks such that all simulation statistics are collected from the phase of interest. The benchmarks that have the same name but belong to different sets (e.g., `kmeans` and `fft`) either use a different algorithm or are simulated in different phases.

6.3 Measurements and Metrics

Across different architectural configurations, performance, power and energy efficiency are measured and compared to the same dual-issue out-of-order x86 baseline for speedup and energy reduction. In addition, microarchitectural studies are conducted to reveal the reasons behind the efficiency of DySER and MAD in the hardware. The metrics are elaborated below:

Benchmark Suite	Benchmark	Description
Base Benchmarks		
Parboil	`cutcp`	Distance-Cutoff Coulombic Potential
Parboil	`fft`	Fast Fourier Transform
Parboil	`lbm`	Lattice-Boltzmann Method Fluid Dynamics
Parboil	`mm`	Dense Matrix-Matrix Multiply
Parboil	`mriq`	Magnetic Resonance Imaging - Q
Parboil	`sad`	Sum of Absolute Differences
Parboil	`spmv`	Sparse-Matrix Dense-Vector Multiplication
Parboil	`stencil`	3-D Stencil Operation
Parboil	`tpacf`	Two Point Angular Correlation Function
GPGPU-Sim	`nnw`	Neural Network
Rodinia	`kmeans(kmns)`	Dense Linear Algebra
Rodinia	`needle`	Needleman-Wunsch Method
Throughput Kernel	`conv`	2-D Convolution
Throughput Kernel	`merge`	Merge Kernel
Throughput Kernel	`nbody`	Simulation of System of Particles
Throughput Kernel	`radar`	1-D Convolution
Throughput Kernel	`treesearch(tsrch)`	Linear Search on Queries
Throughput Kernel	`vr`	Volume Rendering
Access-only Benchmarks		
Micro-benchmark	`agg`	Array Aggregation
Micro-benchmark	`fs`	File Scan
SPECINT	`libquantum`	Physics / Quantum Computing
SPECINT	`mcf`	Combinatorial Optimization
Micro-benchmark	`sortmerge`	Join Kernel
NPU Benchmarks [45]		
NPU	`fft`	Fast Fourier Transform
NPU	`invk2j`	Inverse Kinematics for 2-joint Arm
NPU	`jmeint`	Triangle Intersection Detection
NPU	`jpeg`	JPEG Encoding
NPU	`kmeans`	Dense Linear Algebra
NPU	`sobel`	Sobel Edge Detector

Table 6.3: Benchmarks in evaluation

Performance The overall performance improvement over non-specialized baseline is measured as an improvement in execution time in cycle counts from our gem5 based model:

$$\frac{Baseline\ Execution\ Time(cycles)}{Specialized\ Execution\ Time(cycles)}$$

Here the baseline execution time is the time of the general purpose host processor executing original (non-specialized) code. Although the evaluated architectures may have different implications in timing, we assume that they can all be implemented in a state-of-the-art 2GHz host processor and functions at the same core clock.

Power and Energy-Efficiency The power consumption of the baseline host processor, including the SSE unit, is modeled in McPAT [85]. McPAT is a parameterized power estimation tool that takes microarchitectural parameters and events as input (e.g., reads and writes to cache, functional unit accesses) and provides dynamic, peak and leakage power. For our purpose, only the runtime dynamic power is used. As previously mentioned, we built our power model for DySER, MAD, NPU and C-Cores based on RTL implementations.

This dissertation uses the power delay-product ($Power \times Execution\ Time$) as the energy measurement. The other common metric, energy-delay product, is not used because it may over-emphasize the performance on battery-critical platforms like mobile devices. The overall energy reduction is calculated with the following:

$$\frac{Baseline\ Execution\ Energy}{Specialized\ Execution\ Energy}$$

This metric guarantees that the reduction is always a positive value; compared to the baseline, an inefficient specialization approach results in a value that is lower than 1.

Microarchitectural Studies This dissertation focuses on two supporting architectures in the DySE model, DySER and MAD; detailed microarchitectural studies are performed to understand the sources of speedup, potential bottlenecks, and their sensitivity to hardware resources. These studies include the microarchitectural events like instructions or operations per cycle, the

instruction window size; we also break the power consumption at microarchitecture level to examine the tradeoffs with our specialized hardware. Regarding the sensitivity study, we alter the cache bandwidth and LSQ size, as well as the overall execution parallelism in our models. The former (events and power breakdown) are collected through instrumenting our architecture models in the gem5 simulator, and the latter are simply changes in microarchitectural parameters.

7 EVALUATION

The Evaluation of Dynamically Specialized Execution and its supporting architectures are performed in an incremental basis. DySE, as a flexible model, is evaluated in four different architectural configurations, where each of them leverages one or both of the supporting architectures (DySER and MAD) developed in this dissertation. We evaluate the overall performance, power and energy efficiency compared to a non-specialized platform, and investigates the microarchitectural trade-offs and potential bottlenecks. This chapter is organized as follows: Section 7.1 presents the results of using DySER with a general purpose host processor as access component hardware; Section 7.2 evaluates the full-potential of the DySE model, using DySER with MAD to perform execute and access component, respectively; Section 7.3 examines MAD's generality with the integration of MAD and other accelerators; and last, Section 7.4 discuss MAD as a single accelerator that performs access-only phases.

7.1 DySER with Host General Purpose Processor as The Access Engine

The first step of applying the Dynamically Specialized Execution model is to integrate DySER into a general purpose host processor. In this configuration, this general purpose processor is responsible for both hosting DySER (configuration and executing non-specialized phase) and executing the access component of the phase. These phases, in our hand-optimized benchmarks, are identified and decoupled into access and execute components by programmers. Such transformation process improves the overall performance and energy efficiency as the following: (1) it reduces the total number of instructions by offloading operations to DySER as well as leveraging the vector interface of DySER. (2) it uses DySER to execute offloaded operations, which is the execute component in the DAE model.

Reducing the Dynamic Instructions By vectorization and offloading instructions to DySER, the number of instructions is significantly reduced; the processor pipeline is now only responsible for a few packed memory accesses. Compared to conventional SIMD or vectorization, DySER's

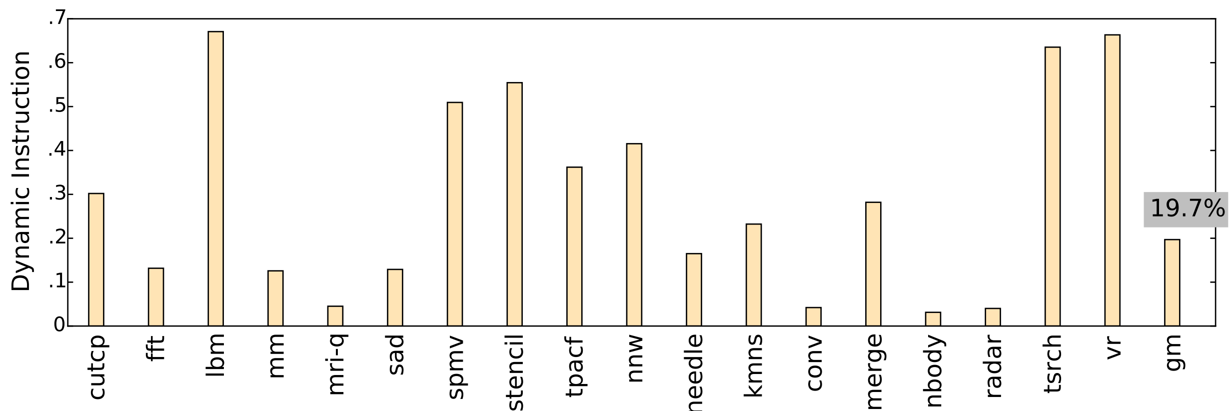


Figure 7.1: The reduced dynamic instructions in percentages

flexibility in its vector interface allows more aggressive vectorization and hence reduce more instructions [58]. Figure 7.1 presents the total percentage of dynamic instruction that is reduced, with the execution of the same phases. In data-parallel kernels such as `fft, mm, conv`, close to 90% of the dynamic instructions can be eliminated; some of these instructions are turned into compact vector instructions, and others are offloaded to DySER. Overall, when the host processor and DySER execute programs that are written in the DySE model, 60% of the instruction, on average, can be eliminated and thus reliefs the host processor pipeline. We refer to these instruction reductions again when explaining speedup results.

It is observed that in these data-parallel benchmarks, the interface instructions are not an overhead when re-programming; for some general purpose workloads the DySER interface instruction may results in a higher overall instruction counts [58].

Size of the Execute Component Figure 7.2 summarizes the number of operations (each is mapped onto DySER's functional units) in the execute component of dominated phases in our benchmarks. While these application phases may be very different, our manually optimized benchmarks can exploit from roughly 10 to 60 functional units on DySER. Note that these operations may not be viewed as "instructions" because it may require many more Von Neumann instructions that the number of nodes (operations) to construct a dataflow graph. In a first-order observation, the above results suggest that the Host/DySER configuration increases the overall energy efficiency by removing the load on the general purpose pipeline, in both software

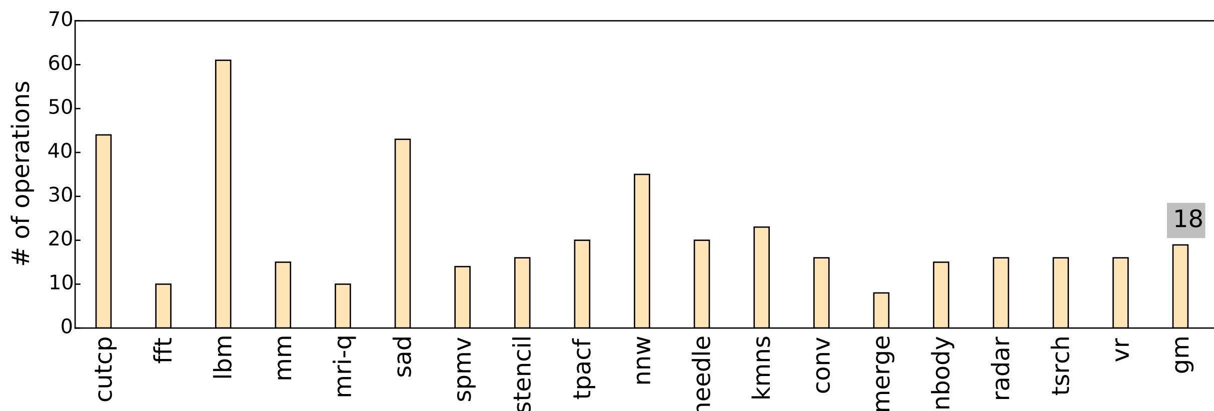


Figure 7.2: Execute component size in operations

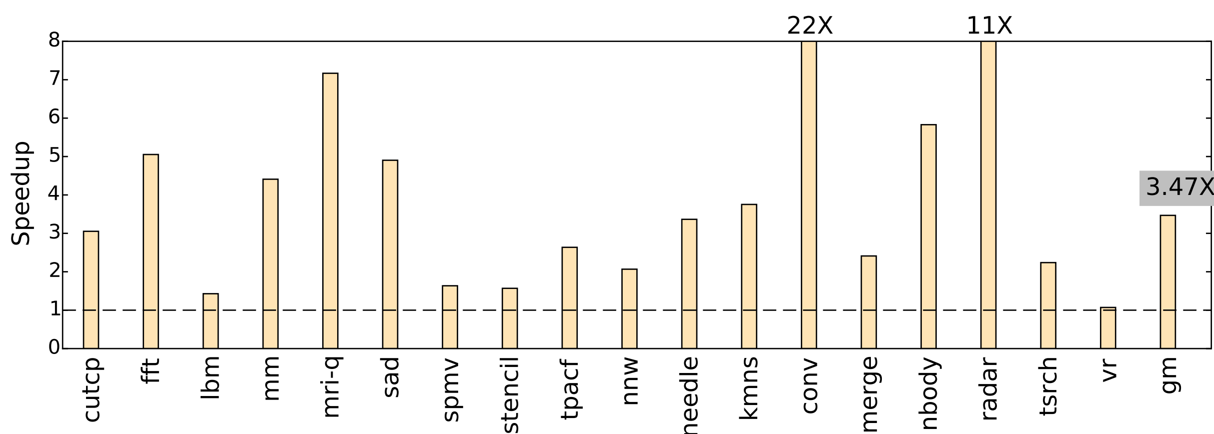


Figure 7.3: Speedup over 2-OOO baseline

(re-programming and the reduction of dynamic instructions) and hardware (operations are executed on DySER).

7.1.1 Speedup and Energy Reduction

Figure 7.3 quantifies the overall performance improvement over the baseline 2-issue out-of-order processor when applying DySER; this Host/DySER configuration speedups the benchmarks from roughly 1.1× to 22×. The speedup in general follows the trend observed in the reduction of instructions; the benchmarks that use fewer instructions (compared to baseline) in consequence have higher speedup. For example, in highly regular benchmarks such as `conv` and `radar`, the application phases can be programmed with the least instructions and DySER can effectively

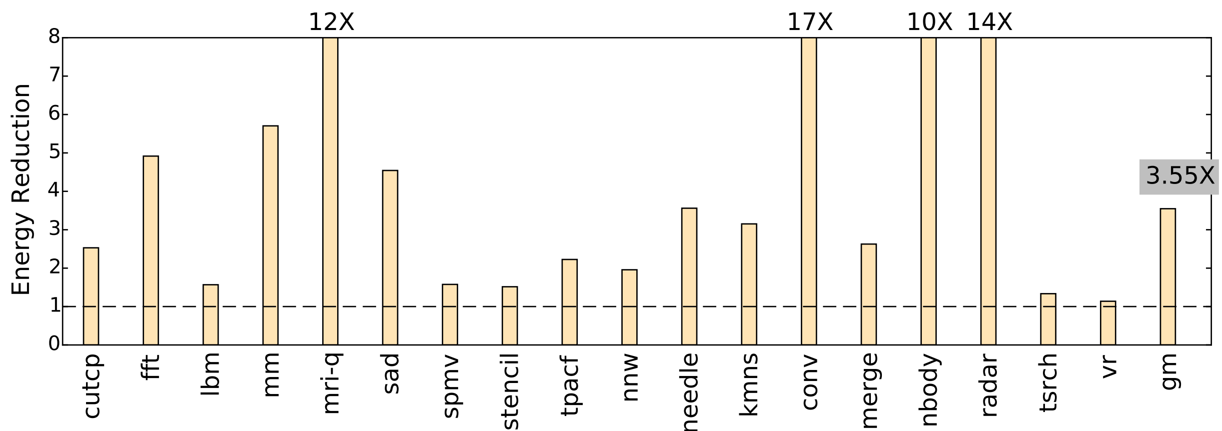


Figure 7.4: Energy reduction over 2-OOO baseline

exploit its vector interface to perform the computations in the execute component. For the benchmarks that the DySE version did not reduce many instructions, in contrast, the performance gain from DySER is minor. In particular:

- `lbm` has a very large phase, and the execute component we extracted from it is not proportional to the size of the phase.
- `spmv` and `nnw`'s access components have many indirect memory accesses. While we tried to move some address calculations in the access components to the execute ones, the true data dependency between the two components limits the overall performance.
- `treesearch` has a long dependence chain which can only be executed serially.
- `vr` cannot be efficiently vectorized because of the irregular control-flow and control-dependent memory accesses.

The above performance improvements, moreover, are the major contributor of the energy reduction. Figure 7.4 plots the energy reduction of Host/DySER compared to non-specialized baseline. On average, utilizing DySER reduces the overall energy by 3.55×, which is roughly 70% of the energy; we also observed that the overall power is reduced while we in fact are using more hardware (DySER) for execute component. For example, in the benchmark that has no

speedup such as `vr`, the overall energy is still reduced because of the lower power. The next subsection elaborates more on the details.

7.1.2 Microarchitectural Analysis

The decoupling of access and execute component in the DySE model enables two microarchitectural benefits. First, it allows the use of the hardware resources in the execute component accelerator and improves parallelism. Second, it effectively creates a larger instruction window to saturate the execute component accelerator without increasing the size of general purpose structures such as reorder buffer. Below, we discuss the instruction and operation level parallelism in the Host/DySER configuration, and also quantifies the effective instruction window size. At the end of this section, we examine the power breakdown of Host/DySER to conclude on DySER's efficiency.

Instruction and Operation Level Parallelism While we have seen that the performance improvements are proportional to the use of vectorized instructions, these instructions, in the end, are fetching data from memory to DySER for parallel computation. To evaluate the parallelism in both host processor and DySER, we use IPC (for host) and DySER operations per cycle (DySER OPC) as the metrics. As shown in figure 7.5, Host/DySER configuration has a lower IPC in the processor pipeline, but it increases the parallelism, on average, by two DySER operations per cycle with the DySER functional units. In our power efficient out-of-order baseline configuration (where the non-specialized execution has an average IPC of 0.9), roughly the DySER OPC increases the overall parallelism by 3×. The lower parallelism in host pipeline in the Host/DySER configuration, however, is because that the host often waits for values from memory access or DySER computation to proceed its execution.

DAE and Instruction Window In an out-of-order superscalar processor, the parallelism in the pipeline is supported by a wide instruction window that buffers in-flight instructions and saturates the execution stage. The Host/DySER configuration, similarly, holds a large instruction window for both the instructions in host and the functional units in DySER. The

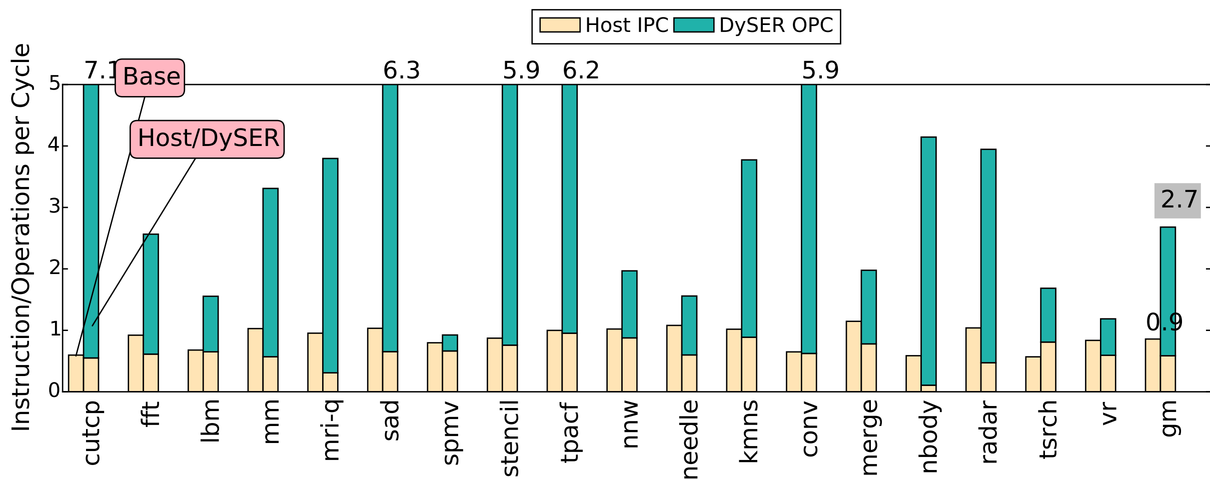


Figure 7.5: Instruction and operation level parallelism

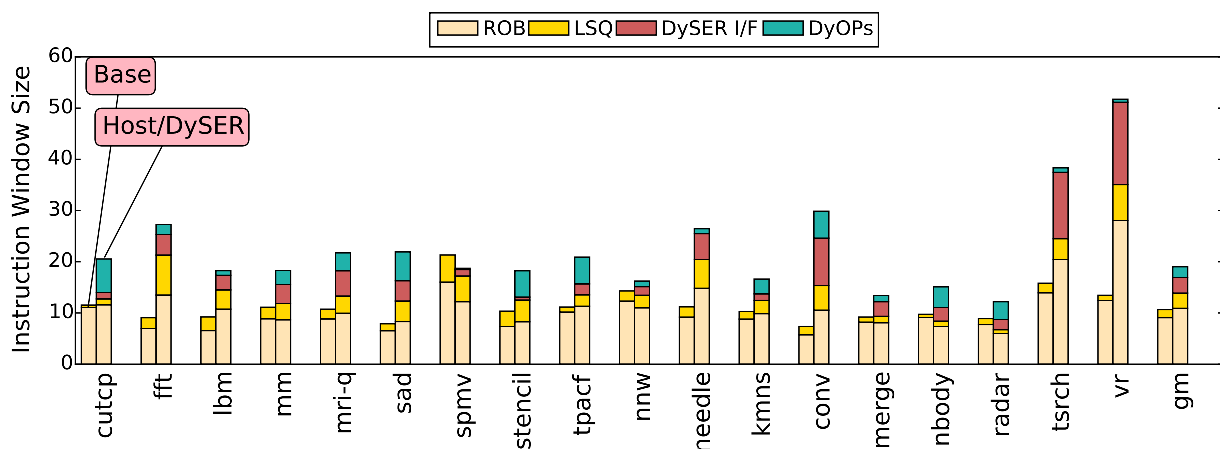


Figure 7.6: Instruction window size

size of this window impacts the "extractable" parallelism in the application phase, since a small instruction window may be filled with dependent instructions and incapable of executing new independent instructions. As shown in Figure 7.6, we present this effective instruction window in the baseline processor and Host/DySER as the sum of the following: (1) mean reorder buffer entries (ROB); (2) mean load store queue entries (LSQ); (3) mean DySER interface instructions in the DySER interface buffers (DySER I/F); and (4) mean DySER operations (DyOps). In the results, Host/DySER on average has twice the window size than non-specialized baseline. A few observations are:

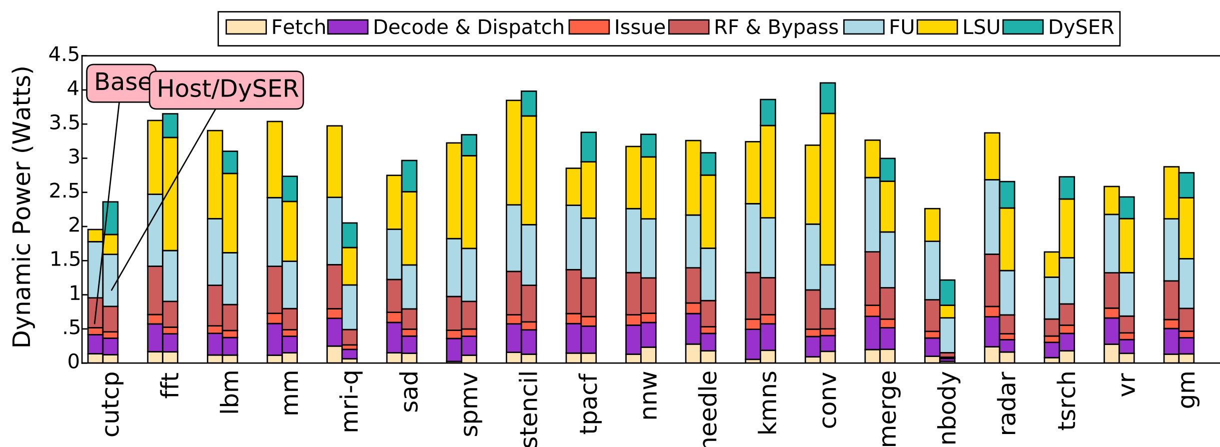


Figure 7.7: Dynamic power

- The memory access burst length (the number of memory access in a given short period) is larger in Host/DySER than non-specialized baseline, which results in larger LSQ bar in the figure. This phenomenon is because that the decoupled access/execute model tend to pipeline many memory accesses for loading values to DySER, and then issue many stores for the outputs from DySER. Fortunately, our 1 data cache port and 26-entry LSQ host is still capable of tolerating this burst with its LSQ.

- The reorder buffer in Host/DySER holds more instructions, because some instructions may depend on DySER output.

- In the benchmarks that have long dependence chain (`treesearch`), or many interface instructions (`vr`), or employ software pipelining (`conv`), the DySER instructions may have to stay in the buffer for some time before they can commit.

The above results show that while the Host/DySER configuration has a wider instruction window, it may stress the ROB or LSQ. In the next analysis, we inspect the power breakdown of Host/DySER to understand its power-efficiency.

Power Breakdown Figure 7.7 plots the power breakdown in the following categories: (1) fetch, (2) decode and dispatch, (3) issue, (4) register file and bypass logic (including the result bus), (5) functional units in the pipeline, (6) load store unit. In particular:

Sub-Module	Functional Unit (Int)	Functional Unit (FP)	Switch	I/O Queue
Power	6.7mW	3mW	5.2mW	1.9mW

Table 7.1: Power breakdown of DySER's sub-modules

- DySER is not a major contributor to the overall power; it consumes roughly 400 mW, which is less than 15% of the total power.
- The increased instruction window does not result in visible power increase in ROB or LSQ.
- However, the data cache is stressed in a shorter execution time and hence uses more power.
- In every benchmark, the power of register file is reduced; on average using DySER results more than 2$\times$ of saving in RF power.

We observed that half (9/18) of the benchmarks use more power than the non-specialized baseline. In `fft`, `tpacf`, `kmeans`, `convolution`, and `treesearch`, the data cache accesses are the major contributor because of more frequent accesses in the re-programmed code. In `cutcp`, `stencil`, `nnw`, and `sad`, the power consumption of host and non-specialized baseline are similar, and DySER becomes the additional cost. Although not shown in graph, the Host/DySER configuration will be more power-efficient if considering a more aggressive baseline (such as 4-issue out-of-order host); the reason behind is that the savings from the general purpose structures increases with a power-hunger host.

Table 7.1 details the power breakdown of functional units, switch, and I/O queues in DySER, where the power is the average power across all benchmarks in this section. In the model, the floating-point functional unit [1] uses the most power; the switch power is similar to the floating-point unit power. Compared to the above two modules, the functional unit with integer ALU and the I/O queues use less power, roughly 0.5$\times$.

Sensitivity on Hardware Resources In the DySE model, the data delivery rate of the access component impacts the overall performance. The design decisions of the out-of-order host under the DySE model, moreover, may be different from a general purpose machine. Figure 7.8

[1] we use an open-sourced float-point unit from open-cores for our purpose [2].

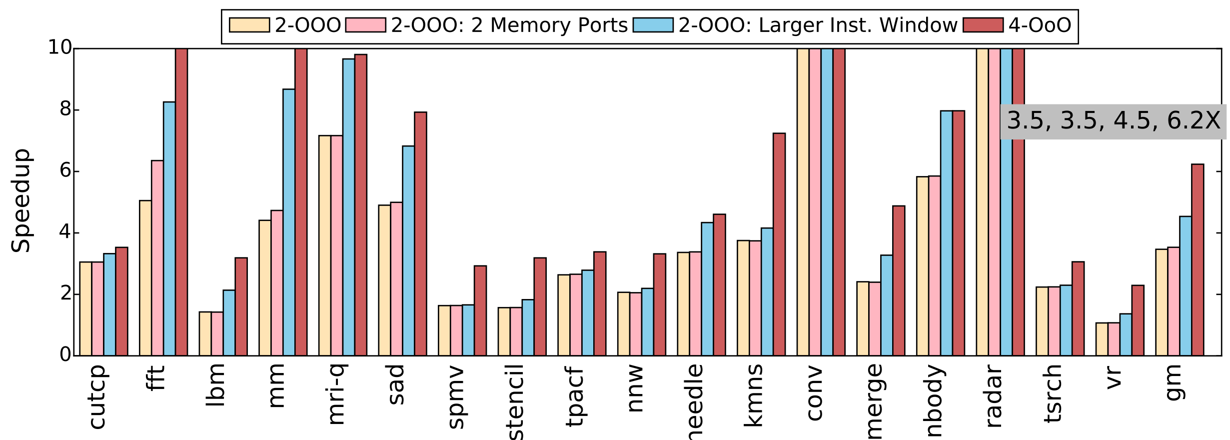


Figure 7.8: DySER's sensitivity on different access hardware

presents our finding of DySER when integrating to our baseline 2-issue out-of-order processor, a 2-issue out-of-order with the same parameters as baseline but doubling the resources in LSU (the cache port doubled from 1 to 2, doubled LSQ entries), a 2-OOO with the same cache ports but a larger instruction window (the LSQ, the ROB and the scheduler size are equal to the sizes in 4-OOO), and a 4-issue out-of-order processor modeling the high-performance Intel Sandy Bridge architecture. From the results, the 2-OOO/DySER with 2× memory port configuration shows speedup only on benchmarks that have a higher LSQ usage, such as `fft` and `mm`; these benchmarks have more memory access bursts and benefits from the doubled LSQ and cache ports. Other benchmarks, however, are still limited by the front-end of the pipeline, including the issue width and the instruction window size. In the case of a larger ($\approx 4\times$) instruction window, the performance is increased by roughly 30%. If DySER is driven by 4-OOO, both the limitations in the execution width and the instruction window are removed, and we observe prominent speedup (77% on average) across all benchmarks. Overall, the performance is highly correlated to the data delivery speed, which involves both the parallelism, the instruction window and the cache bandwidth.

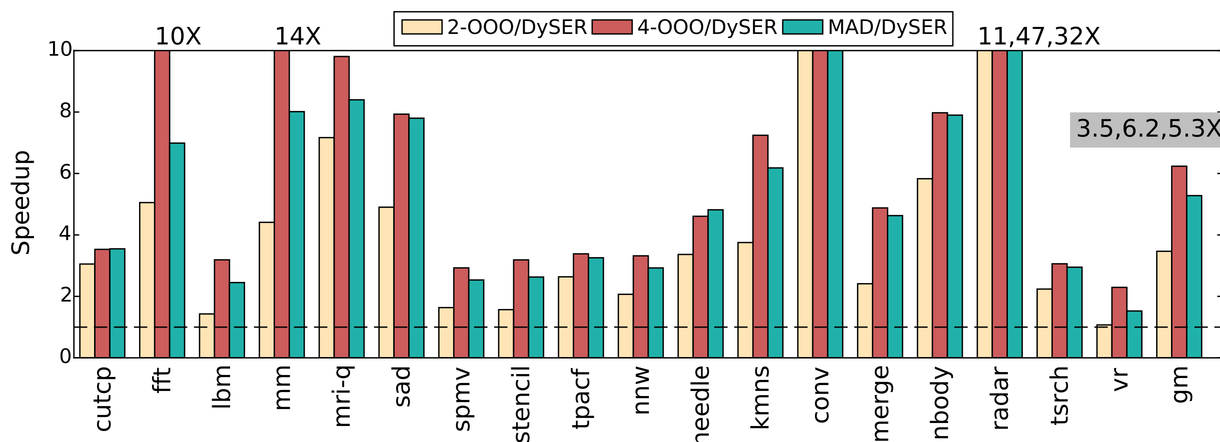


Figure 7.9: Speedup over 2-OOO baseline

7.2 DySER with MAD

In the analysis on Host/DySER configuration, we have observed that DySER is not a major contributor to the overall power, and the host processor spends half of the power on the front-end, issue, RF, bypass and functional units of the pipeline. For a more aggressive pipeline (wider than our baseline low-power 2-OOO configuration), the power consumption in these superscalar structures may also increase. The MAD architecture, on the other hand, provides more parallelism than a 2-OOO processor from its dataflow scheduling and parallel microarchitecture with lower power. In this section, we summarize the overall speedup and energy reduction of MAD/DySER configuration and compare it with two reference design points: 2-OOO/DySER and 4-OOO/DySER.

7.2.1 Speedup and Energy Reduction

Figure 7.9 shows the performance when using DySER with different access component hardwares. Here, 2-OOO/DySER represents a basic configuration that specializes only the execute component of the program phase, and 4-OOO/DySER offers a better performance than the basic configuration via faster data delivery. From the results, we can observe that with a wider issuing width, larger instruction window and more cache ports, 4-OOO/DySER achieves roughly 2× more performance than 2-OOO/DySER. In particular:

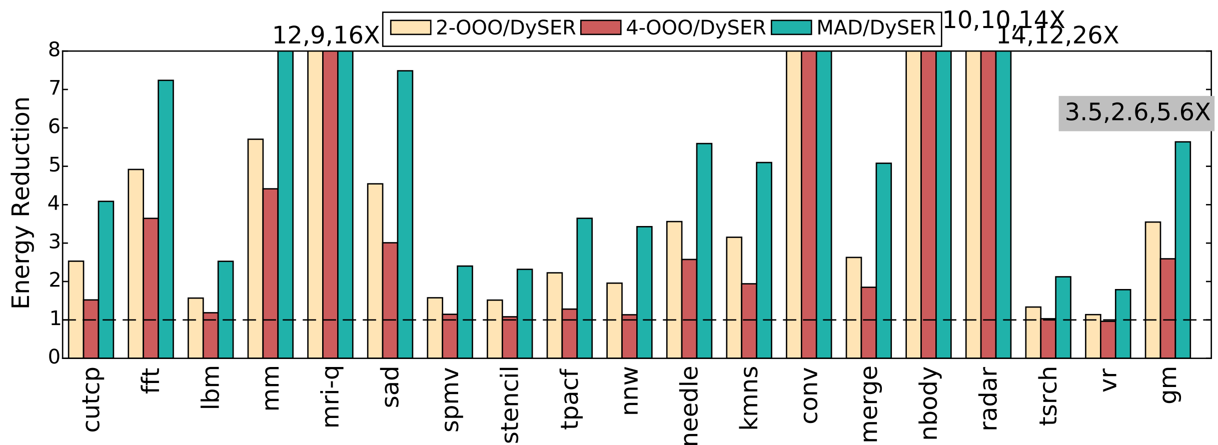


Figure 7.10: Energy reduction over 2-OOO baseline

- In benchmarks that have abundant parallelism (e.g., `mm` and `fft`), the 2-wide pipeline width of 2-OOO is not able to extract all the parallelism in the access component code; this results in significant speedup from 2-OOO/DySER to 4-OOO/DySER.

- In `vr` and `lbm`, the 4-OOO pipeline can tolerant the cache misses (roughly 20%) and buffer the load store instructions with its wider pipeline and larger instruction window; thus 4-OOO/DySER has a 2× performance boost.

The MAD/DySER configuration has more functional units (in the computation block) compared to 2-OOO/DySER, but has the same load-store unit and same number of cache ports; it can benefit from the parallelism but cannot buffering loads and stores as aggressive as 4-OOO host. Therefore, in the benchmarks that have little stress on the load store queue (e.g., `cutcp`, `sad`, `needle`, and `nbody`), MAD/DySER can perform as 4-OOO/DySER. In `needle`, we specifically re-arrange the loads such that the required values can be pre-load from memory; this results in slightly better speedup in MAD/DySER than 4-OOO/DySER. Overall, with MAD's dataflow architecture, the performance is increased by roughly 50% than 2-OOO/DySER but off by 16% when compared with 4-OOO/DySER.

Unlike the performance, 4-OOO/DySER is not the best choice among three design points regarding the overall energy. Figure 7.10 presents the energy of 2-OOO/DySER, 4-OOO/DySER, and MAD/DySER. First, in our power model the 4-OOO processor (modeling Intel Sandy Bridge

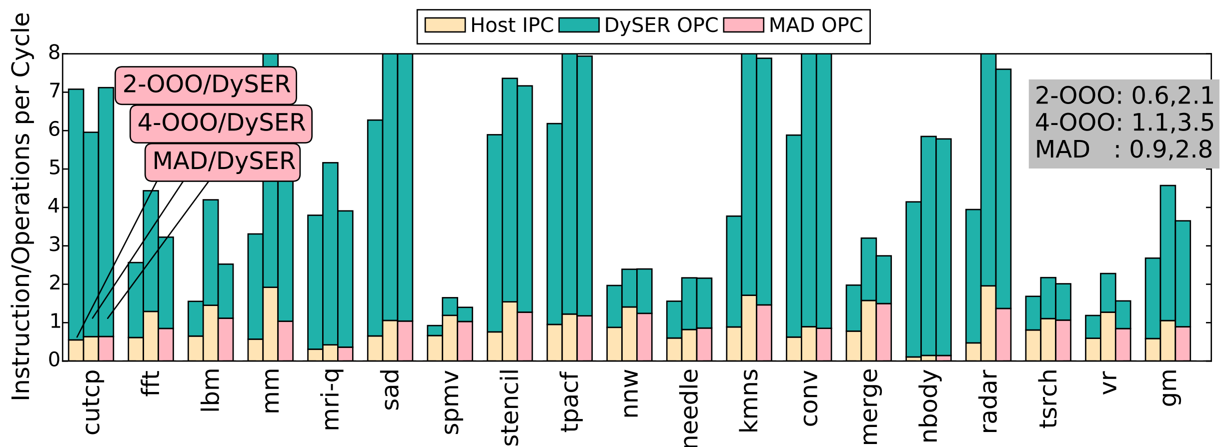


Figure 7.11: Parallelism in MAD/DySER

architecture) on average consumes 2× power compared to 2-OOO. This gap in the model results in higher energy consumption of 4-OOO/DySER than 2-OOO/DySER (and hence little or no energy reduction in many of the benchmarks) event though its performance is better. Second, the results show that MAD/DySER configuration offers better energy reduction; compared to 2-OOO/DySER and non-specialized baseline, it is 1.6× and 2.1× better respectively. Thus, using the MAD architecture can be more energy-efficient than increasing the hardware budget for a faster and wider out-of-order engine.

7.2.2 Microarchitectural Analysis

The MAD architecture performs dataflow execution for performance and power efficiency. For the former, it breaks a sequential phase into dataflow graph and extracts parallelism from concurrent events, actions and computations; this section examines the effectiveness of our MAD phase transformation. For the power efficiency, the MAD architecture leverages pre-configured computations and ECA rules to avoid the excess dynamism inside a processor pipeline; the section presents the power breakdown of the MAD/DySER configuration to quantifies this power-efficiency. In the end, a sensitivity study is performed to validate the design point of our implementation of the MAD hardware.

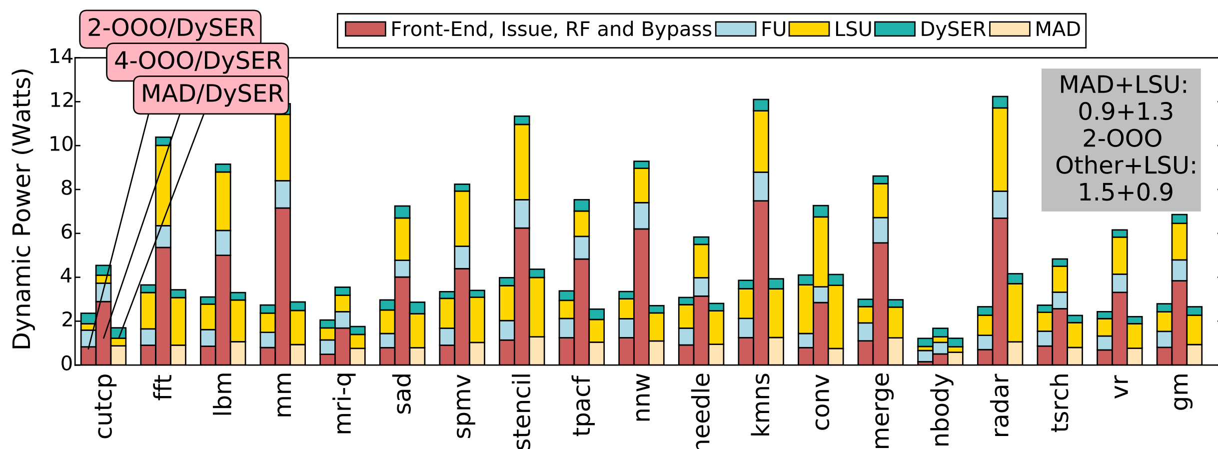


Figure 7.12: Dynamic power

Dataflow Parallelism in MAD Figure 7.11 plots the instruction/operation level parallelism. In host processor (2-OOO and 4-OOO), the parallelism is evaluated with instructions per cycle. In MAD, we calculate the operations per cycle as below:

$$MAD_{OPC} = \frac{\#\ of\ actions\ +\ \#\ of\ compute\ operations}{cycles}$$

where the compute operations are the activated functional units inside the computation block of the MAD hardware. In all, the MAD can trigger around one action or activate one computation per cycle and deliver the data faster (hence better DySER OPC) compared to our 2-OOO host, under the same number of cache port. We also observed that, during the MAD execution, the execution pattern is more like computation or action bursts followed by waiting for DySER outputs. Similar to other statically scheduled spatial architecture, MAD's simple hardware may have a lower utilization rate but does not need many auxiliary structures to saturate a few functional units.

Power Breakdown In a specialized phase, the MAD hardware replaces processor's role on the front-end, issue, execute and write back stages. We summarize the power of these stages and compare the host pipeline with the MAD hardware. As shown in Figure 7.12, the host pipe is break into functional units (which performs the necessary address computation), the load-store

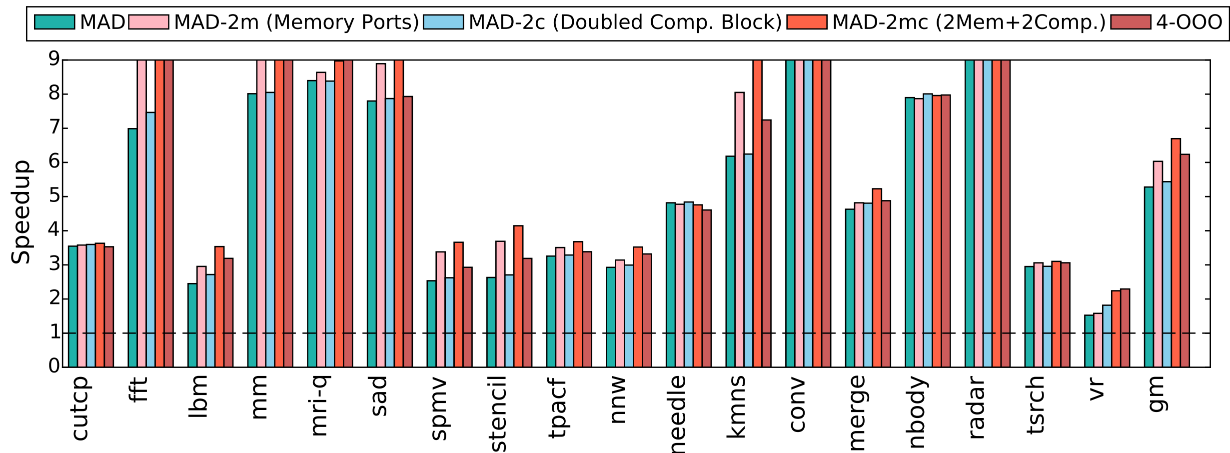


Figure 7.13: MAD's sensitivity on hardware resources

unit and a composition of other stages. Few observations are:

- The MAD hardware consumes less power than the sum of Fetch, Decode, Dispatch, Issue, Execute, and Write-back stages while they are performing the same work;
- MAD/DySER stress LSQ more than 2-OOO/DySER because more frequent accesses (less total execution time); and
- Compared to 4-OOO/DySER, MAD/DySER has similar power consumption on LSQ but significantly better in other parts.

In our design, the integration of MAD bottlenecks in load-store queue when considering power; one of our future works is to merge load-store queue into event queues to save power.

Sensitivity on Hardware Resources Thus far, we have studied our MAD hardware implementation with two canonical design points of commercial out-of-order processor. To understand that if our MAD hardware design point is a balanced implementation as our 2-OOO and 4-OOO model, here we discuss the MAD architecture's sensitivity on its hardware resources (the number of functional units in the computation block) and interface (the number of cache ports). Figure 7.13 considers three additional design points: (1) **MAD-2m**, where we use two cache ports (and a wider action block) instead of the 1 cache ports to increase the bandwidth;

(2) **MAD-2c**, where we use doubled functional units in the computation block with doubled computation block I/O event queues; (3) **MAD-2mc**, where the computation block and cache ports are both doubled. The 4-OOO/DySER configuration is also plotted for reference. Two observations are:

- MAD is more sensitive on the memory bandwidth; our 2-OOO baseline's one cache port and small LSQ may be overly conservative for a MAD integration. Simply increase the computation block size does not increase parallelism or performance.
- However, with both higher bandwidth and more functional units, MAD increases the performance by roughly 25%. Although not shown here, the overall energy reduction is not as good as performance increase because the additional consumption on hardware.

The finding of this sensitivity study is different from the study on 2-OOO, where we increased the cache ports and LSQ but observed little speedup. This observation suggests that, effectively, MAD provides more parallelism and a larger instruction window with its pre-configured dataflow execution compared to our power-efficient 2-OOO model.

7.3 MAD Driving Other Execute Accelerators

Section 5.5 discusses three additional integrations, MAD/SSE, MAD/NPU and MAD/C-Cores, and shows the flexibility of the MAD architecture. In this section, we quantitatively analyze the performance and energy reduction when using MAD to diving these accelerators. A specific in-order core design point is used in this section for C-Cores; this is because the Conservation Cores proposal [122], it was integrated into an in-order machine for maximum power efficiency. To control the experimental variables, we use the same issue width, branch predictor, cache configurations as our 2-OOO host (it does not have ROB or other out-of-order specific structures).

7.3.1 Speedup of MAD/Accelerators

Figure 7.14 presents the performance of the three accelerators when driving by MAD, 2-OOO or 4-OOO. First in Figure 7.14a, 4-OOO/SSE and MAD/SSE have prominent improvement over

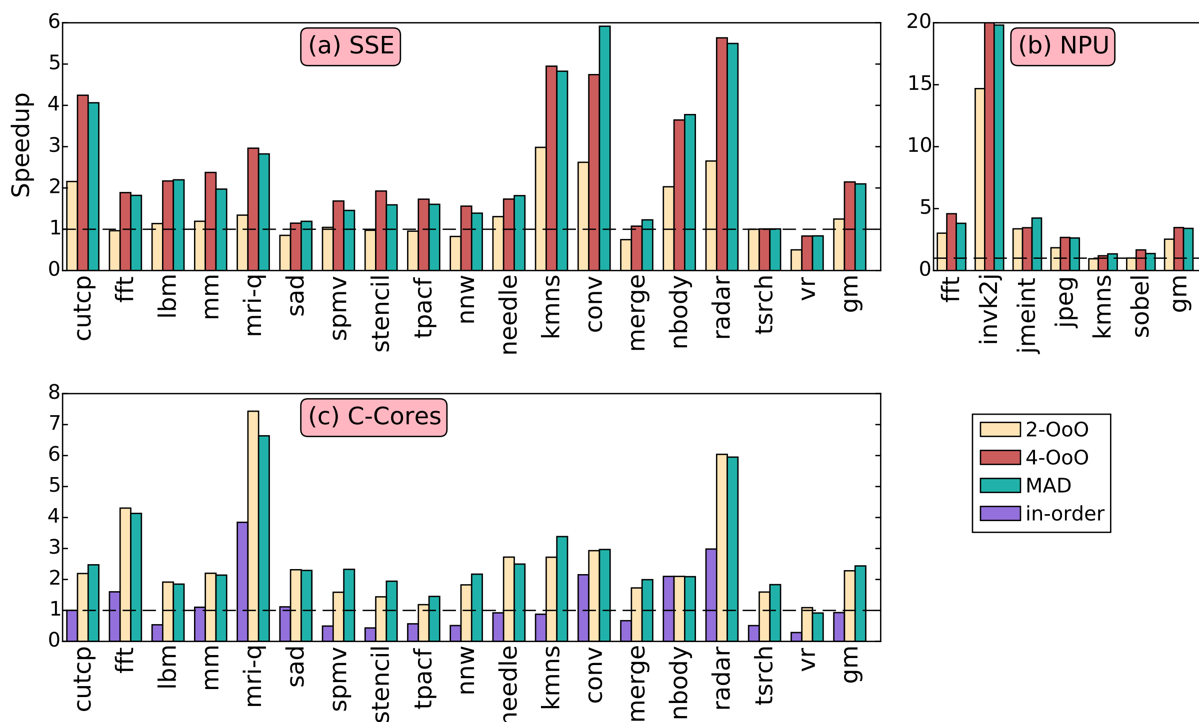


Figure 7.14: Performance of MAD/Accelerators

2-OOO/SSE; this is because that, when compared to DySER other accelerator, SSE use more instructions and requires a wider front-end to issue them. These fine-grained instructions are often independent or with some ILP in our benchmarks, and 4-OOO and MAD can effectively perform them. In benchmarks like `conv`, our software-pipelined phase contains many more parallel SSE instructions such that even 4-issuing is not enough; MAD/SSE performs better because the MAD hardware can be configured with more parallel lanes. `needle` and `merge` and `nbody` have similar characteristic.

Figure 7.14b shows that MAD/NPU has similar performance as 4-OOO/NPU, and both of them are better than 2-OOO/NPU. The speedup increase is not as large as what in the SSE analysis because of two reasons: (1) in NPU benchmarks, there are dependencies between invocations of a neural network in NPU; and (2) NPU does not have a vector interface. Overall, MAD/NPU offers roughly 20% more performance over 2-OOO/NPU.

As previously mentioned in MAD/C-Cores analysis, we compare MAD against in-order and 2-OOO hosts. As shown in Figure 7.14c, the performance of in-order C-Cores is limited

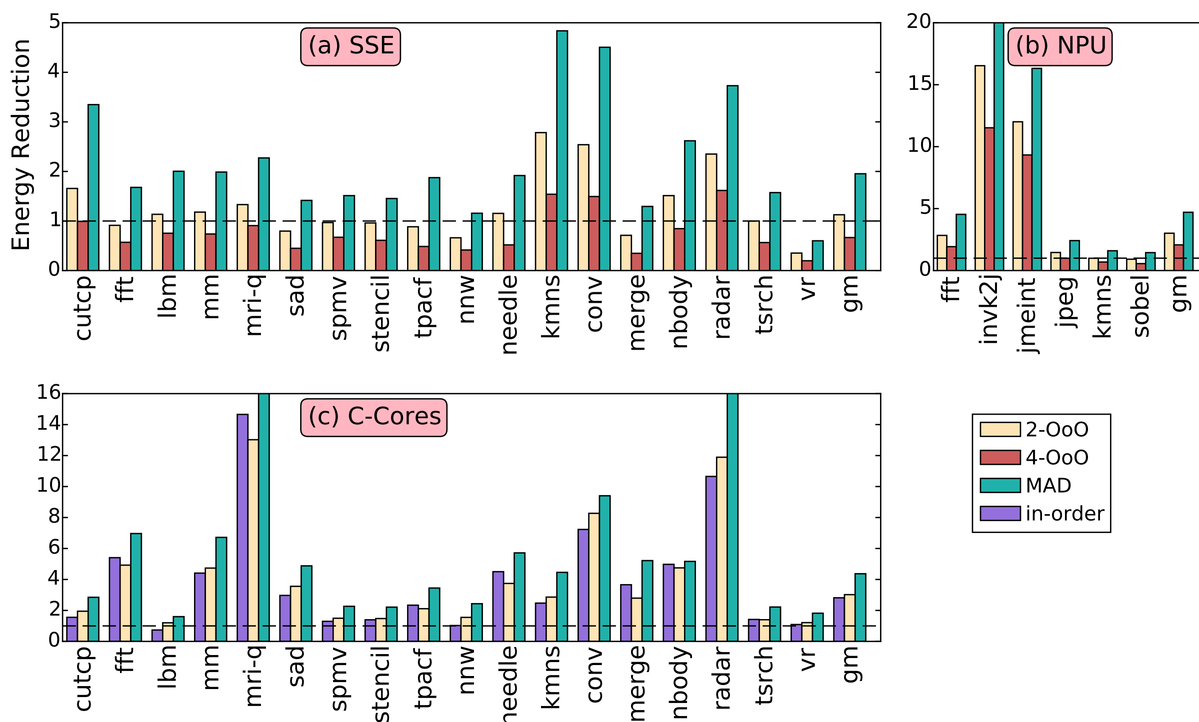


Figure 7.15: Energy reduction of MAD/Accelerator

because of the sequential execution, and is always less than MAD/C-Cores and 2-OOO/C-Cores; MAD and 2-OOO is a better access component architecture when it can exploit the parallelism in memory access dataflow and feed multiple C-Cores (with only computation operations) concurrently. In addition, we observe that 2-OOO/C-Cores surpasses MAD/C-Cores in a few benchmarks. Because C-Cores is built with an inherently sequential execution model for low power (no vector interface as well), our C-Cores program cannot fully exploit the parallelism in the MAD hardware.

7.3.2 Energy Reduction of MAD/Accelerators

The basis of energy improvement is straightforward: the MAD hardware consumes 50% less power compared to the sum of Front-end, issue, RF and functional units of a 2-OOO host. In Figure 7.15a, we observed that MAD/SSE outperforms 2-OOO/SSE in energy; the SSE instructions create a heavy load on the 2-OOO pipeline, but not in the spatially configured MAD hardware. This observation suggests that the design of our MAD hardware integration can

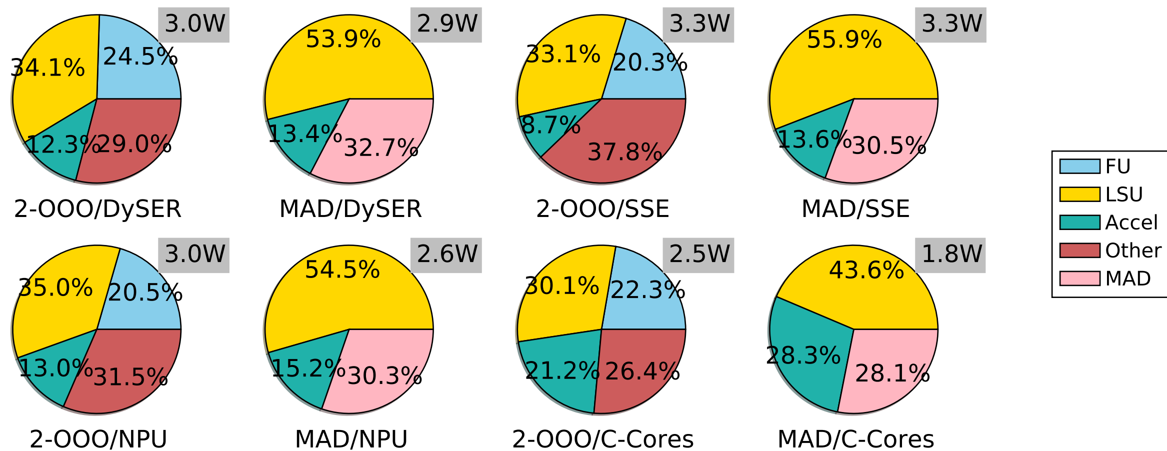


Figure 7.16: Energy reduction of MAD/Accelerator

effectively reduce the dynamism in the pipeline (in fact our power model reports a higher overall power on 2-OOO/SSE and 4-OOO/SSE compared to non-specialized baseline). On average, MAD/SSE offers roughly 2× energy reduction, which is 50% less energy.

Figure 7.15b presents the overall energy reduction of MAD/NPU, 2-OOO/NPU and 4-OOO/NPU. It is observed that NPU energy reduction is dominated by the energy consumption of the access component hardware; our power model shows that the NPU hardware consumes a lower power than all the other accelerators used. In such a case, MAD/NPU offers the best energy reduction (more than 4×) over the baseline.

In Chapter 2 we have discussed that in-order processors are not comparable to out-of-order hosts in the two following cases: executing general purpose phases of an application and the overall energy envelope. Figure 7.15c validates this statement with C-Cores as an example; in many cases, in-order/C-Cores consumes more energy during the specialized phase because of a longer execution time. [2] From the results, using MAD is slightly better than 2-OOO on average, and our 2-OOO/C-Cores and MAD/C-Cores consumes less energy because of better speedup.

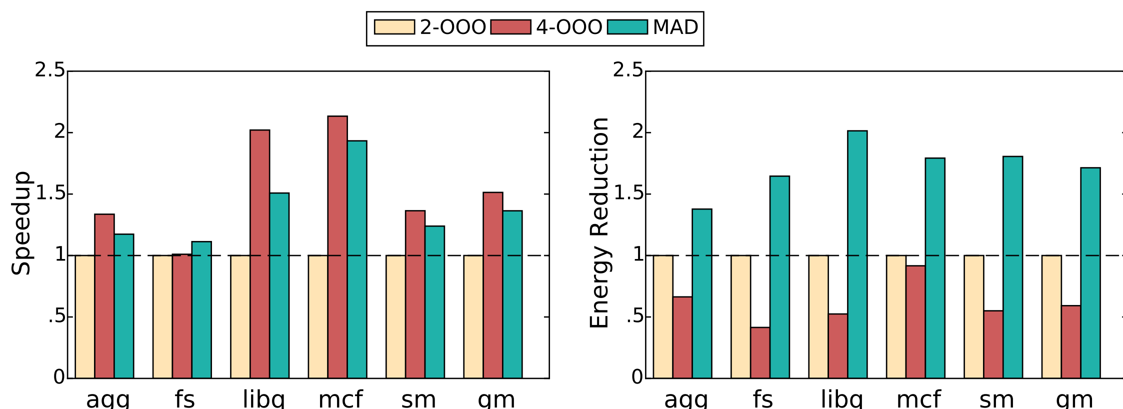


Figure 7.17: MAD (Access-only) performance and energy

7.3.3 Overall Power Breakdown of MAD/Accelerators

Figure 7.16 summarizes the power breakdown of MAD/Accelerators. For simplicity, we classify the power in to the five categories, functional unit, load-store-unit, accelerator, other, and MAD as described in Section 7.2.2 and only shows the result of 2-OOO. The power consumption numbers are the average across all the benchmarks in the corresponding set (e.g., MAD/NPU uses NPU benchmarks). By observation, in the 2-OOO/Accelerator configurations, the one-third of “other“ (Fetch, Decode and Dispatch, Issue, RF and Bypass) and the power of functional unit are merged into the MAD hardware in the MAD/Accelerator configuration. While in general the power of load-store unit increases because of the better performance and utilization, MAD/Accelerator is a more balanced design compared to 2-OOO/Accelerator.

7.4 MAD Executing Access-Only Benchmarks

We evaluate five memory intensive kernels to understand the potential of the MAD hardware as a pure memory access engine without any accelerators attached. The five kernels are: (1) array aggregation (`agg`), (2) filescan (`fs`), (3) two node array traversal and conditional update kernels in libquantum (`libq`), (4) `mcf` from SPECINT [115], and (5) a join kernel in sortmerge

[2]Compared to 2-OOO without acceleration, the energy reduction of our in-order/C-Cores configuration is less than the energy reduction reported in [122]. This is because of the performance of the OOO processors are much better than in-order with our data-parallel workloads.

(`sm`). These kernels traverse through data structures and have very little computation, which are often not specializable for compute accelerators.

7.4.1 Speedup and Energy Reduction

Figure 7.17 shows the speedup and energy reduction when using the MAD hardware to execute these access-only application phases. On average, MAD(Access-only) offers roughly 40% speedup over the non-specialized baseline, with 1.7× energy reduction. In a specialized phase, the results give the evidence that the MAD architecture can improve the overall performance without the cost of power-hunger out-of-order structures. In benchmarks that have many misprediction from memory-dependent branches like `fs`, MAD can offer a better performance than 4-OOO from less overhead in rollbacks.

7.5 Chapter Summary

This chapter explores four different configurations under the DySE model, and deeply investigate the efficiency of the two supporting architecture– DySER and MAD.

DySER Conventional out-of-order processor design uses many auxiliary structures to guarantee a sustainable performance when executing applications for different purpose; the DySE model helps relief the load of this general purpose pipeline by decoupling and specializing application phases. Our analysis shows that we can effectively create a more efficient code (with fewer instructions) and use DySER to perform the execute component in a specialized phase. With a general purpose host executing the access component code, DySER improves the overall performance and energy efficiency and is not a major source of power consumption. Regarding the performance sources and bottlenecks, DySER is offering much more parallelism through its hardware, vector integration interface, and the re-programming of the phase; it is bottlenecked by its host in both performance and energy.

MAD To exploit more benefit from the DySE model, the MAD architecture can be used to perform the access component code in a specialized phase. MAD/Accelerator configuration

replaces the power in-efficient general purpose host and performs memory access for execute accelerators. Our analysis on the MAD hardware reports better performance and energy reduction compared to a 2-OOO host, diving four different accelerators. We also observed that, while providing more performance with lower energy, our MAD hardware is bottlenecked on the load-store queue and the memory bandwidth.

The MAD/DySER Integration Although not emphasized in the evaluation, MAD/DySER exploits the full potential of the DySE model and achieves the best speedup and energy reduction over our 2-issue out-of-order baseline processor. It performs better than MAD/SSE or MAD/C-Cores in the same benchmarks, which follows the finding prior works [60, 58]. The flexibility in the interface design allows DySER and MAD to specialize the application phase efficiently with more parallelism, and the microarchitecture of DySER and MAD eliminates the use of power hunger out-of-order structures.

8 RELATED WORK

Numerous hardware specialization approaches have been proposed to address power and energy efficiency. As previously mentioned, we only consider architectures that target specified tasks, regions, or phases as a specialization approach–not a different architecture that executes an arbitrary program. In particular, this section compares the two supporting architectures in our DySE model to related architectures. Section 8.1 discusses DySER and other execute architectures and Section 8.2 compares MAD to other access architectures.

8.1 DySER and Execute Architectures

The closest work to DySER is the Burroughs Scientific Processor (BSP) [81]. BSP uses arithmetic elements implemented with pipelining to accelerate vectorized FORTRAN code. The evolution of two important insights from BSP lead to the DySER architecture. First, compilers are used achieve generality in both BSP and DySER; they enable automatic identification and specialization of application phases. DySER further expands the flexibility by using a circuit-switch network and a vector interface. Second, both BSP and DySER identify the critical role of intermediate value storage. The arithmetic elements in the BSP have dedicated register files that are not part of the architecture state. Unlike this "centralized" design, DySER provides distributed storage in its network using flow-control pipelined registers. The final difference is in the implementation. While the BSP spends much effort on building a fast storage system (register, I/O, special memory), DySER can leverage the flexibility in the DySE model and use conventional core or MAD to achieve the same goal.

In addition to BSP, we briefly discuss a few execute specialization architectures, in the order of increasing granularity (as described in Chapter 2 Figure 2.1).

Fine-Grain Execute Specialization Fine-grain execute specialization approaches often involves a smaller number of functional units and a simpler specialized control logic or front-end; these advantages allow a lower deploy cost. Intel's MicroOp-fusion is one successful exam-

ple, where a simple specialized front-end keeps examining the instructions stream and uses specialized datapath for fused micro-ops.

Another commercialized example is the SIMD accelerator, which extends a single instruction into multiple functional units and process parallel data streams at the same time. These architectures specialize the parallel patterns in an application and can be extremely efficient when executing regular data-parallel patterns. However, using SIMD accelerators usually requires some amount of programming effort; despite continuous effort on compiler development over decades, automatic code generation for SIMD accelerators such as Intel SSE [4] is still difficult and sub-optimal because of the underlying restricted hardware structure.

Moreover, even with programmers' involvement, they cannot execute irregular workloads efficiently. It is often observed that some programs can be ported to SIMD accelerators with very limited gain or worse performance. Various techniques have been proposed to address the above issue. In addition, there are efforts to compile and generate SSE code automatically. Lark et al. proposed the idea of Liquid SIMD [35] to vary the SIMD width to address data-parallel inefficiency.

Coarse-Grain Execute Specialization Coarse-grain execute specialization usually implies more changes in the hardware or integration. From the recent literature, the CCA and VEAL Loop accelerator architectures [32, 91, 30, 33] are related. These implementations target loops and hence are designed with limited scalability, supporting only a few functional units. Specifically, VEAL is limited to inner-most loops that must be modulo-schedulable, and CCA has limited branching support. They do not support a diverse applications like DySE, and have memory access limitations (CCA does not allow a code-region to span across load/stores). VEAL exploits the loop's modulo-schedulability for a novel design which is limited to 2 INT ALU, 2 FP and one compound unit; CCA uses a feed-forward cross-bar network connecting consecutive rows which can limit scalability and thus generality to many domains. Compared to them, the circuit-switched interconnection network and pipelining are the profoundly powerful features in DySER

Other reconfigurable architectures include the following: Garp uses an FPGA-like substrate

for tight-inner loops but suffers when loop iterations are small [68]; the hardware also lacks the capability of mapping control-flow, which limits the patterns it can specialize. Chimaera maps instruction sequences to a reconfigurable, FPGA-like substrate [126]. Compared to DySER, it specializes execution at the circuit level using look-up tables, registers and FPGA routing; DySER, on the other hand, intuitively specializes at operation level and natively supports compiler generated graphs.

At coarse granularity, there are also non-reconfigurable accelerators. These application-specific integrated circuits (ASICs) provides maximum hardware efficiency, in terms of area and power for a specific program. For example, Conservation Cores (C-Cores) is similar to DySER in that the functionality of applications is extracted at compiler IR level, by which a synthesis and mapping process statically specializes the functionality into fixed hardware substrates. To add flexibility of patching the target application, C-cores offer exception-handling capabilities. From above, C-cores maximize the energy efficiency with limited flexibility.

Compared to DySER, Conservation Cores can save more power but sacrifice a significant amount of performance since no parallelism is extracted. In addition, the generality is limited since only little reconfigurability is provided. DySER, as a more balanced approach, may provide energy efficiency over conservation cores because of a higher improvement in performance.

Works related to DySER Microarchtiecture The DySER microarchitecture design is similar to tiled architectures like RAW [82], Wavescalar [117], and TRIPS [22]. DySER is a pure computation fabric and thus has no buffering, instruction storage, or data-memory storage in the tiles. Second, DySER implements circuit-switched static routing of values, thus making the network far more energy efficient than the dynamically arbitrated networks. While these architectures distribute design complexity to different tiles, DySER does not add complexity. The DySE model and the decoupling of access and execute components enable several simplifications and DySER only needs to perform computation. In all, they are a different architecture aiming for general purpose, but DySER is a specialization approach for specific application phases.

Recent Proposals The design philosophy of DySER can be seen in several recent proposals in their principles; here we classify recent proposals that related to DySER in two categories:

algorithmic specific acceleration and dynamic composition of primitive functions.

- **Algorithmic specific acceleration:** Convolution Engine [103], Widx accelerator in the Meet the Walkers [78], HARP [124], and NPU [45] accelerates a specific algorithm in hardware, which enables efficient hardware optimization and often involves a specialized data acquisition mechanism. Convolution Engine customizes load, store, register file and the datapath (map and reduce logic) to deliver data to its functional units; the Widx accelerator accesses hash buckets in parallel for advanced hash computation, but with a conventional RISC processor; HARP accelerates range partitioning, where it applies a decoupled execute model that interfaces its comparator to memory with queues (stream buffer); and NPU, as previously mentioned, is developed for computing multi-layer perceptron neural networks. Among these proposals, NPU and HARP are more related to DySER because of their decoupled execution model.

- **Dynamic Composition of primitive functions:** BERET [63], Q100 [125] both identify the primitive functions in phases. BERET constructs such phases as subgraph execution graphs and developed functional units and interconnection to map the graph; Q100 focuses on database primitives and links these primitives together for a larger phase. They both remove the overheads of issuing instructions to perform primitive functions in a phase.

8.2 MAD and Access Architectures

In terms of overall goals, MAD is most closely related to the position paper by Hou et al. [70], which describes "the common characteristics of the data access patterns of the accelerators; and points out both opportunities and practical challenges in addressing deficiencies in existing designs." As depicted in Table 8.1, MAD is a novel confluence of concepts in three disparate categories - accelerators/heterogeneity, the decoupled access/execute model, and dataflow. Pre-computation and loop accelerators share MAD's end goal of improving the memory behavior. Prior works are either too narrowly tied to a "native" platform (Loop accelerators), or too general that they become unusable in concert with accelerators (Dataflow), or do not target memory access, power, and performance goals (DAE and Pre-computation)

Topic	Closest related	Similarity	Conceptual difference
DAE	Outrider [38]	Goal is same - DAE to improve memory performance	Outrider creates many threads, MAD does exposed fine-grained dataflow for energy efficiency
Dataflow	Triggered Execution (TE) [99]	Events & Actions exposed in ISA	MAD μarchitecture targeted for integration with accelerator, lower-level μarchitectural events, & MAD does dataflow computation
Pre-computation	SDDM [107, 106]	Data-driven "strands" of address compute	SDDM is von-Neumann and "strands" run as SMT threads; MAD's native dataflow achieves energy efficiency
Loop accelerators	RSVP [30]	"Configurable" address generation patterns for DMA-like engine	RSVP supports only vector model (memory access mechanism tied to RSVP compute fabric), no branching allowed and host processor must remain on.

Table 8.1: Comparison to related work

Access Component in Decoupled Access/Execute The classic DAE model was introduced in the early 1980s [112, 56], and is incarnated in classic vector processors, conditional vectors, decoupled vector clustered microarchitectures [76, 62], VT and Maven [80, 84, 17, 83], coarse-grain software decoupling [79, 71] in very recent work, in a fragment processor design [12] allowing/exploiting extremely regular addresses in graphics workloads, and hosted accelerators [103, 45, 122, 57]. In a sense, accelerators can be viewed as vector cores with arbitrary vector chaining, and the FIFOs provide are akin to a highly "irregular" vector register file. In that regard, dynamic vectorization conceptually seeks to achieve the same as specializing the access-code [97, 120].

Dataflow MAD's dataflow execution resembles the classic dataflow machines from 70s to 90s [42, 64, 72, 13, 98], and more recent incarnations [109, 117, 21]: we also specify explicit targets and dataflow based computation of work. As shown in the Table 8.1, Triggered Instructions(TI) [99] which combined action/events with dataflow is most similar to MAD. Scheduled Dataflow [75] is also quite similar to TI and has inspired many other DAE/dataflow hybrids. None of them

use low-level dataflow for memory access.

Pre-computation, pre-computation with reintegration In pre-computation, the idea is to execute a "reduced" version of a program in the hope of triggering cache accesses early for costly cache misses [128, 127, 9, 94] or for branch mispredictions [46]. Assisted execution [43] and SSMT [24] are general paradigms for such pre-computation, and lookahead combines pre-computation and DAE [51, 104, 50]. Pre-computation with register integration, as developed in the SDDM work when viewed in the accelerator context, creates a data-driven thread for each load and store. This approach is conceptually similar to MAD's execution model, but with a very different implementation because of different end goals.

8.3 Chapter Summary

In this chapter, we discussed the related work to the two supporting architectures, DySER and MAD. Prior works related to DySER differ in the granularity and the microarchitecture capability (e.g., control-flow, reconfigurability, and the vector interface). MAD, moreover, is a very different approach that efficiently realizes the access component; it differs from the related work in its execution model and the event-driven dataflow microarchitecture.

9 Conclusions and Future Work

This dissertation describes an integrated approach of hardware specialization by computation and data acquisition optimization under the DySE model. In this final Chapter, we summarize the contributions and discuss the possible areas of future work.

9.1 Contributions and Conclusions

Architectural hardware specialization have thrived over the past 20 years, and this work begins with a summary of the underlying principles of hardware specialization in Chapter 1. It then details the Dynamically Specialized Execution model and its two supporting architectures:

The Execution Model This dissertation develops an alternative execution model, DySE, which replaces the original Von Neumann superscalar execution in profitable application phases. It shows how decoupled access/execute and reconfigurable hardware can be effectively utilized to enable new optimizations.

An Execute Specialization Architecture We detailed the design, implementation and evaluation of DySER, the execute component hardware and accelerator under the DySE model. DySER uses a light-weight and reconfigurable circuit-switched network to construct the computation datapath dynamically. With the reconfigurability, the flow-control protocol design, the capability of supporting control flow and data-parallel integration interface, DySER can eliminate the overheads in conventional instruction processing and provide energy reduction. We specifically discuss a successful integration, SPARC-DySER, as a proof of the concept for DySER.

An Access Specialization Architecture We examine the out-of-order core architecture and discover the fundamental tasks when driving an execute accelerator in a specialized application phase. We then propose the MAD architecture, which replaces the role of the host processor and drives a family of execute accelerators that fits in the decoupled access/execute model. We developed the ISA, microarchitecture, and integration of MAD; then evaluate it with DySER and other accelerators, and explore its potential with access-only application phases.

9.2 Future Work

The DySE model and its two supporting architectures are empirically evaluated, with analysis of many aspects. Through the design decision and evaluations, we identified several optimization opportunities for future works, in both microarchitecture and system level design. This section discusses the directions of future work, enlightened by our findings.

System Level Integration DySE like other specialization approaches can be applied out of the processor core; following the configure-execute-reuse philosophy, DySER and MAD can be integrated onto a system-wide interconnection and work as co-processors for specific code regions. This type of integration implies the some changes to the current design, in particular:

- **Memory:** Leaving the processor core, DySER and MAD lacks a well-defined memory subsystem that previously was provided by the data cache inside the core. A specialized memory subsystem (or even scratchpad memory) can be used and may increase efficiency; a cache compliant memory subsystem is also applicable and has advantages in sharing data between general purpose cores or other accelerators on chip. However, the latter may require modifications in MAD to work with a coherent cache.

- **Configuration and Interface:** The configuration and DySER/MAD interface to cores of a system-level integration may be different from the instructions extensions discussed in this dissertation. The processor cores have to send remote message packets over the system to configure DySER and MAD, which may not be very efficient. One intuitive solution is to use an additional DMA to configure both DySER and MAD by reading from a given memory block.

- **Resource Sharing and Scheduling:** The use of DySER and MAD under system level enable opportunities in sharing DySER and/or MAD; consequently this integration requires scheduling of cores and application phases that are executed on DySER/MAD.

Software or Hardware Specialized Front-End Section 2.2.4 discussed two cases that the current DySE model cannot efficiently support with the architectures we designed: the flat profile

and control intensive phases. Besides re-writing the application manually, compiler or programming language techniques may be used to organize the generated configurations differently. For example, the SNACK middleware [61] programming language, although in a different domain, allows the compiler to re-organize the control flow of the final program. Control-flow optimization [87, 37] is another long-developed software approach to address such issues. Section 2.3.2 also approached the solution; if a hardware front-end can be built, it can dynamically profile and cache the frequent phases instead of creating it at compile time. While a pure hardware approach may result in prohibitive power consumption or limit the phase size, a hardware/software co-designed runtime profiling and phase creation platform could be a future direction.

Merging LSQ into MAD In the evaluation of MAD, we observed that the load-store queue, the data TLB and data cache becomes a major source of power consumption. While the data cache and data TLB are necessary, the load-store queue can be merged into MAD since the MAD hardware already has storage structures (event queues) that temporally holds data values. To utilize the existing distributed event queues, we can borrow the wisdom from prior efforts on removing the CAM structure of the load-store queue [116]. The basic idea is to use dependence predictors to predict if the loads are dependent on a prior store, and to push the values from the event queue that contains the store data to the target event queue that is supposed to hold the future loaded data. In such a case, roll-back and flush mechanisms have to be modified in the event block to manage mis-predictions.

9.3 Reflections

This final section provides few lessons learned from the work with a closing remark. First, it discusses possible alternatives regarding the microarchitecture design; with even simpler and straight-forward hardware, DySER or MAD can be an alluring approach for industrial architects. Second, this section revisits the findings of the microarchitectural analysis, which provides some insights for future research.

9.3.1 Even Simpler Microarchitecture Designs

The microarchitectural design complexity can be critical in the industry; in such a circumstance, simpler microarchitecture help to justify the deployment of a new unit such as DySER or MAD. To reduce the complexity of MAD or DySER, we consider the following designs:

Trading Generality for Complexity

In the DySER network and the MAD computation block, we can reduce the complexity by supporting fewer functions or lower connectivity between functions. This approach, however, scarifies the generality and may not be able to execute some dataflow graphs with the simpler network.

Homogeneous Functional Units and Alternatives to the Interconnection Network

In the DySER network and the MAD computation block, reconfigurable fabrics are used to perform static dataflow computations. Several approaches can be used to reduce the complexity. First, homogeneous functional units can lower the scheduling complexity; they come with a higher overhead in power because more logics may be unused in a configuration. Second, with the homogeneous functional units, we can use grouped FUs (like compound functional units) instead of a dataflow network. Unlike the circuit switched dataflow network, which is the original design, using the grouped FUs makes the execution similar to the execution stage in a superscalar pipeline; grouped FUs switch between pre-configured instructions to execute a dataflow graph. Buffers (temporal registers) can be used to chain the computation within grouped FUs in different cycles. While this design may use more energy per computation operation, it follows conventional wisdom and can be simple in implementation.

Event Triggering with CAM and Temporal Registers

In the MAD event block, we use fine-grained ECA rules to check the dataflow events and states in the event queues. Instead of spatially clustered queues and comparator arrays, A CAM and temporal registers can be used to hold the values and check the readiness the values. This approach resembles the instruction wakeup logic in a superscalar pipeline, where the scheduler sends register ID, value and the state bit to the CAM for waking up the consumer instructions.

With the previously mentioned grouped FUs, the event triggering can be done at a coarser granularity with a packet of temporal register IDs, values and state bits; the CAM then takes this packet from grouped FUs and wakes up the pre-configured instructions to the grouped FUs. The action block, however, have to be changed to schedule pre-configured instructions besides scheduling the loads and stores.

Overall, the ideas of DySER and MAD can be implemented differently with tradeoffs in complexity, performance and power.

9.3.2 Revisiting the Observations

This dissertation developed several observations in microarchitecture design. Although it may take more efforts to generalize these observations, they are intuitive and can be hints to future research:

- Parallel execution units or stage is energy-efficient, and the harder problem is to create such parallelism in hardware. The performance from pipeline width have to be considered along with the instruction window size. The hardware units that create the parallelism, like the scheduler, register file and reorder buffer in a superscalar pipeline, are not only cycle-critical but also power-critical.

- Software can help hardware to create the parallelism for the execution units; pre-configuration of a hot code-region is one of the approaches that relief the burden of the hardware. Conventional instruction stream does not help much since the hardware units have to dynamically analyze the relationships between each instruction.

- Cache bandwidth and the capability to process cache access requests in the same cycle are not only critical to the throughput, but also to the execution time or latency. In the case of decoupled access/execute, having memory values faster results in a lower execution time of the access and the execute components and more overlapping between the two components. It changes the critical path.

9.3.3 Closing Remark

Today, hardware specialization and accelerators are everywhere, from high-performance computing and servers to mobile devices. Every new spark, inevitably, face the same question on novelty and has to distinguish itself from a vast sea of literature. In the development of DySER and MAD, we went through the same question and developed a complete statement that covers the software stack, the execution model, and the support architecture and microarchitecture. Hardware specialization is not merely about parallelism for performance or producing microarchitectures that "harden" tasks from software for power efficiency; it should has design principles and mechanisms, different than the ones in general purpose machines, that enables more parallelism or lower power on specific tasks. Besides offering a new hardware specialization approach, we hope this work also improves the development process of future proposals, facilitating the realization of better hardware specializations in commodity processors or systems.

A THE ENCODING OF MAD ISA

This chapter discusses the MAD ISA and the encoding of the configuration bits. Specifically, Section A.1 discuss the encoding of the dataflow graph nodes, which use the same encoding of the functional unit and switches in DySER. Section A.2 details the encoding of the ECA Rules (described in Chapter 5) inside the event and action table.

A.1 Dataflow Graph Node

In the MAD ISA, the address and branch computation are represented as dataflow graph nodes; Section 5.2.1 described the dataflow graph nodes in 3-tuples (source, operation, destination). This description abstracts the underlying hardware and thus could support different implementations. Here, we discuss a specific implementation of the MAD computation block; it leverages the idea of DySER and constructs the dataflow graph with functional units and switches. In this implementation, the graph is mapped onto the hardware (which is exposed into the MAD binary translator) and encoded in terms of functional units and switches. The functional units are responsible for the operations in dataflow graph nodes, and the switches construct the edges in-between.

Table A.1 presents the encoding of the functional unit and switch; each of them is 24-bit in length. We describe the two encoding below.

Switch The first row of Table A.1 shows the encoding of a switch. It has 8 output multiplexers (`W`, `SW`, `S`, `SE`, `E`, `NE`, `N`, and `NW`); each of them selects from 5 input direction and forwards to the destination. The encoding of a switch is straightforward: Each of the multiplexer uses 3 bits to select the input.

Functional Unit The functional units has the following fields: (1) Data input mux selection from `DI0` to `DI2`, where each of them uses 3 bits; (2) 4-bit control flow code (`CF`), which specifies if this functional unit uses predicate, phi function, or selection; and (3) 10-bit opcode (`Opcode`). The second row of Table A.1 lists the encoding.

<table>
<tr><th>μarch</th><th colspan="9">Encoding</th></tr>
<tr><td>Switch</td><td>W[2:0]</td><td>SW[2:0]</td><td>S[2:0]</td><td>SE[2:0]</td><td>E[2:0]</td><td>NE[2:0]</td><td>N[2:0]</td><td>NW[2:0]</td></tr>
<tr><td>FU</td><td>DI0[2:0]</td><td>DI1[2:0]</td><td>DI2[2:0]</td><td colspan="2">CF[3:0]</td><td colspan="3">Opcode[9:0]</td></tr>
</table>

Table A.1: The encoding of the dataflow graph

<table>
<tr><th>Rule</th><th colspan="5">Encoding</th></tr>
<tr><td>Event-Conditions</td><td colspan="2">EQ0-4[15:0]</td><td colspan="3">Condition0-4[7:0]</td></tr>
<tr><td>Actions</td><td>Src-Dst0[7:0]</td><td>Src-Dst1[7:0]</td><td>Op0[1:0]</td><td>Op1[1:0]</td><td>reserved</td></tr>
</table>

Table A.2: The encoding of the ECA rules

A.2 ECA Rules

In the MAD hardware, an ECA rule is decoupled into Event-Conditions and Actions. The former is stored in the event table of the event block, and the latter is stored in the action table of the action block. Similar to the encoding of the functional units and the switches, each of the encoding of Event-Conditions or Actions is 24 bits.

Event-Conditions As described in Section 5.3.5, an ECA rule in our implementation supports four triggering event queues. For each of the event queue, a condition field can be set to examine the condition of the corresponding event queue. The condition of the event queue can be `true`, `false`, or empty, which indicates that the examination of the state is not required for this event queue. The first row of Table A.2 shows the encoding of the Event-Conditions.

Action The second row of the Table A.2 presents the encoding of the actions. Each action supports two source-destination event queue pairs (`Src0, Dst0, Src1, Dst1`); each of the source-destination event queue pair has a `Op` field, which indicates if this movement is a load, a store, or a simple move. In the case of loads and stores, the data in the source or destination event queue is used as memory addresses.

B THE SET/RESET PROTOCOL

This chapter sets out to prove the correctness of the Set/Reset protocol (described in Section 3.3.1). In particular we prove two properties: i) The reset phase sets all nodes in a configuration into free state; and ii) the set phase that follows sets them all correctly into the new configuration state.

To begin, we can model the DySER network as a connected graph G that has some number of nodes. A configuration of DySER uses a subset of the nodes to construct a directed subgraph G'.

Assumption B.1. *The DySER network G is a bi-directional connected grid graph (i.e., a partially connected mesh topology).*

This assumption simplifies the topology and connection of the DySER network; this proof can be extended to the real DySER network with some effort. We define the configuration subgraph G' as the following:

Definition B.2. *In an configuration, a subset of nodes and edges of G constructs subgraph G′, which is an acyclic graph; each of the node ∈ G′. has some producers and some consumers. In G′, there exists some source nodes and sink nodes that has no producer or consumer.*

Using the set and reset protocol, the DySER network is able to switch between different configurations, where each of the configuration may use different nodes and edges and thus has a different subgraph than others. For a node $n \in G$, it can be in one of the following state:

- $Used$: the node is currently in use. This state represents $n \in G \wedge n \in G'$.
- $Free$: the node is free to be set to a new configuration; it may still in the free state (not used) after it is set to a new configuration.
- Set : the node received a set signal and about to be used (will be turned into the used state in future); similar to the used state, it represents $n \in G \wedge n \in G'$.
- $Reset$: the node received at least one reset signal and about to be freed.

The data (as well as the set and reset signals) are sent from the edge nodes of the current configuration subgraph G'. These data are processed in-order in the $used$ nodes in G'. Two additional variables, P_n and C_n, are used to describe the set of producer nodes and the set of consumer nodes of node n in G'. We assume that a correct hand-shake flow-control protocol is responsible for the data and the signals between the nodes.

Assumption B.3. *For the data or signal that is send from node n to m, a hand-shake protocol guarantees the receipt of the data or signal; if it has not been received, the data or signal is hold (asserted) in node n until node m receives it. Node n does not receive any new data or signals if it holds any data or signal, or if it have not received all data from P_n, which is required for processing the data.*

The set and reset signal may modify the state, the sources, and the destinations of a node; they are represented in two functions $set(n, m)$ and $reset(n, m)$. neighboring nodes is in the $free$ state. With the above states and functions, we define the state machine of the set and reset protocol below:

- A node m in the $used$ state turns into the $reset$ state if all nodes $n \in P_m$ have sent $reset(n, m)$ and the data processing is done;
- A node n in the $reset$ state sends $reset(n, m)$ to all nodes $m \in C_n$ and turns into the $free$ state after all m have received the reset signal;
- A node m in the $free$ state turns into the set state if any of the neighboring node n sent a $set(n, m)$ signal, and other neighboring nodes are in $free$ state; and
- A node n in the set state sends $set(n, m)$ to all nodes $m \in C_n$ and turns into the $used$ state, after all m have received the set signal.

Note that the $set(m, n)$ and $reset(m, n)$ follows Assumption B.3; this implies that $reset(n, m)$ only occurs after all data from n have been processed at m. Initially, the nodes in graph G' is in the $used$ state, and the nodes in $G - G'$ is in the $free$ state. The first step of the proof is the correctness of the reset protocol:

Lemma B.1. *Sending reset signals to the source nodes in the subgraph G' of the current configuration eventually turns all nodes in G' into the $free$ state.*

Proof. First, the processor sends the reset signal to all source nodes n in G'. These nodes turn into the $reset$ state and send $reset(n, m)$ for all $m \in C_n$; after confirming the receipt, they can turn into the free state. Since G' is a connected directed acyclic graph, the nodes in G' turn into free state in-order and all nodes can be freed. □

The second step of the proof is the correctness of the set protocol:

Lemma B.2. *After the sending reset signals, sending the set signals to the source nodes (in the free state) of the new acyclic subgraph G'' (new configuration) can correctly set all nodes in G'' to the used state.*

Proof. Sending the set signal to all source nodes n in G'' turns them into the set state. These source nodes are then turned into the $used$ state after sending $set(n, m)$ to all $m \in C_n$. A node m may be in one of the following two conditions:

- **Condition 1:** $m \notin G'$.
- **Condition 1:** $m \in G'$.

If condition 1 is true, then the set signal can be propagated. If the condition 2 is true, then the node m will eventually be reseted into the $free$ state (Lemma B.1) and the set signal can be propagated. □

[1] "ARMv8 Instruction Set Overview," http://www.element14.com/community/servlet/JiveServlet/previewBody/41836-102-1-229511/ARM.Reference_Manual.pdf, accessed: 2014-08-14.

[2] "Floating Point Unit :: Overview," http://opencores.org/project,fpu, accessed: 2014-08-14.

[3] "Intel Advanced Encryption Standard (AES) Instructions Set," https://software.intel.com/en-us/articles/intel-advanced-encryption-standard-aes-instructions-set, accessed: 2014-08-14.

[4] "Intel Streaming SIMD Extensions 4 (SSE4)," http://http://www.intel.com/technology/architecture-silicon/sse4-instructions/index.html, accessed: 2014-08-14.

[5] "Intel's Sandy Bridge Microarchitecture," http://www.realworldtech.com/sandy-bridge/, accessed: 2014-08-14.

[6] *Parboil Benchmark Suite*. http://impact.crhc.illinois.edu/parboil.php.

[7] "Silvermont, Intel's Low Power Architecture," http://www.realworldtech.com/silvermont/, accessed: 2014-08-14.

[8] J. H. Ahn, W. J. Dally, B. Khailany, U. J. Kapasi, and A. Das, "Evaluating the imagine stream architecture."

[9] M. Annavaram, J. M. Patel, and E. S. Davidson, "Data prefetching by dependence graph precomputation," in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, ser. ISCA '01. New York, NY, USA: ACM, 2001, pp. 52–61. [Online]. Available: http://doi.acm.org/10.1145/379240.379251

[10] ARM, *Cortex-A15 Processor*. http://www.arm.com/products/processors/cortex-a/cortex-a15.php.

[11] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, “Boosting mobile gpu performance with a decoupled access/execute fragment processor,” in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ser. ISCA ’12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 84–93. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337159.2337169

[12] ——, “Boosting mobile gpu performance with a decoupled access/execute fragment processor,” in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ser. ISCA ’12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 84–93. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337159.2337169

[13] K. Arvind and R. S. Nikhil, “Executing a program on the mit tagged-token dataflow architecture,” *IEEE Trans. Comput.*, vol. 39, no. 3, pp. 300–318, Mar. 1990. [Online]. Available: http://dx.doi.org/10.1109/12.48862

[14] O. Azizi, A. Mahesri, B. C. Lee, S. J. Patel, and M. Horowitz, “Energy-performance tradeoffs in processor architecture and circuit design: a marginal cost analysis,” in *Proceedings of the 37th annual international symposium on Computer architecture*, ser. ISCA ’10. ACM, 2010, pp. 26–36. [Online]. Available: http://doi.acm.org/10.1145/1815961.1815967

[15] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, “Models and issues in data stream systems,” in *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, ser. PODS ’02. New York, NY, USA: ACM, 2002, pp. 1–16. [Online]. Available: http://doi.acm.org/10.1145/543613.543615

[16] E. Bach, “The algebra of events,” *Linguistics and Philosophy*, vol. 9, no. 1, pp. 5–16, 1986. [Online]. Available: http://dx.doi.org/10.1007/BF00627432

[17] C. F. Batten, “Simplified vector-thread architectures for flexible and efficient data-parallel accelerators,” Ph.D. dissertation, Cambridge, MA, USA, 2010, aAI0822514.

[18] 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 *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, Feb 2012, pp. 1–12.

[19] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718

[20] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for gpus: Stream computing on graphics hardware," in *ACM SIGGRAPH 2004 Papers*, ser. SIGGRAPH '04. New York, NY, USA: ACM, 2004, pp. 777–786. [Online]. Available: http://doi.acm.org/10.1145/1186562.1015800

[21] M. Budiu, G. Venkataramani, T. Chelcea, and S. C. Goldstein, "Spatial Computation," in *ASPLOS XI*.

[22] 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 the TRIPS Team, "Scaling to the end of silicon with EDGE architectures," *IEEE Computer*, vol. 37, no. 7, pp. 44–55, 2004.

[23] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "Legup: High-level synthesis for fpga-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '11. New York, NY, USA: ACM, 2011, pp. 33–36. [Online]. Available: http://doi.acm.org/10.1145/1950413.1950423

[24] R. S. Chappell, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt, "Simultaneous subordinate microthreading (ssmt)," in *Proceedings of the 26th Annual International Symposium on Computer Architecture*, ser. ISCA '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 186–195. [Online]. Available: http://dx.doi.org/10.1145/300979.300995

[25] S. Che, M. Boyer, M. anoyer, J. Meng, D. Tarjan, J. Sheaffer, S. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing."

[26] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang, "Niagaracq: A scalable continuous query system for internet databases," in *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '00. New York, NY, USA: ACM, 2000, pp. 379–390. [Online]. Available: http://doi.acm.org/10.1145/342009.335432

[27] T.-F. Chen and J.-L. Baer, "Effective hardware-based data prefetching for high-performance processors," *Computers, IEEE Transactions on*, vol. 44, no. 5, pp. 609–623, May 1995.

[28] K. S. Chen-Han Ho, Sung Jin Kim, "Memory access dataflow," University of Wisconsin Computer Sciences Technical Report CS-TR-2014-1802, Mar 2007.

[29] Choudhary et al., "Fabscalar: Composing synthesizable rtl designs of arbitrary cores within a canonical superscalar template," in *ISCA '11*.

[30] S. Ciricescu, R. Essick, B. Lucas, P. May, K. Moat, J. Norris, M. Schuette, and A. Saidi, "The reconfigurable streaming vector processor (rsvp trade;)," in *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, 2003, pp. 141–150.

[31] N. Clark, A. Hormati, and S. Mahlke, "Veal: Virtualized execution accelerator for loops," in *ISCA '08*, pp. 389 –400.

[32] ——, "Veal: Virtualized execution accelerator for loops," in *Computer Architecture, 2008. ISCA '08. 35th International Symposium on*, June 2008, pp. 389–400.

[33] 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 *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on*, Dec 2004, pp. 30–40.

[34] N. Clark, A. Hormati, and S. Mahlke, "Veal: Virtualized execution accelerator for loops," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ser. ISCA '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 389–400. [Online]. Available: http://dx.doi.org/10.1109/ISCA.2008.33

[35] N. Clark, A. Hormati, S. Yehia, S. Mahlke, and K. Flautner, "Liquid simd: Abstracting simd hardware using lightweight dynamic mapping," in *HPCA '07*.

[36] 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 *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 37, 2004, pp. 30–40. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2004.5

[37] J. D. Collins, D. M. Tullsen, and H. Wang, "Control flow optimization via dynamic reconvergence prediction," in *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 37. Washington, DC, USA: IEEE Computer Society, 2004, pp. 129–140. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2004.13

[38] N. C. Crago and S. J. Patel, "Outrider: Efficient memory latency tolerance with decoupled strands," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA '11. New York, NY, USA: ACM, 2011, pp. 117–128. [Online]. Available: http://doi.acm.org/10.1145/2000064.2000079

[39] 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 TOPLAS*, vol. 13, no. 4, pp. 451–490, Oct 1991.

[40] L. De Carli, Y. Pan, A. Kumar, C. Estan, and K. Sankaralingam, "Plug: flexible lookup modules for rapid deployment of new protocols in high-speed routers," in *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, ser. SIGCOMM '09, 2009, pp. 207–218. [Online]. Available: http://doi.acm.org/10.1145/1592568.1592593

[41] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted mosfet's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, pp. 256–268, October 1974.

[42] J. B. Dennis and D. P. Misunas, "A preliminary architecture for a basic data-flow processor," in *Proceedings of the 2Nd Annual Symposium on Computer Architecture*,

ser. ISCA '75. New York, NY, USA: ACM, 1975, pp. 126–132. [Online]. Available: http://doi.acm.org/10.1145/642089.642111

[43] M. Dubois and Y. H. Song, "Assisted execution," Department of EE-Systems, University of Southern California, Tech. Rep. #CENG 98-25, 1998.

[44] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," *SIGARCH Comput. Archit. News*, vol. 39, no. 3, pp. 365–376, Jun. 2011. [Online]. Available: http://doi.acm.org/10.1145/2024723.2000108

[45] 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*, ser. MICRO '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 449–460. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2012.48

[46] A. Farcy, O. Temam, R. Espasa, and T. Juan, "Dataflow analysis of branch mispredictions and its application to early resolution of branch outcomes," in *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 31. Los Alamitos, CA, USA: IEEE Computer Society Press, 1998, pp. 59–68. [Online]. Available: http://dl.acm.org/citation.cfm?id=290940.290960

[47] E. Fernandez, W. Najjar, S. Lonardi, and J. Villarreal, "Multithreaded fpga acceleration of dna sequence mapping," in *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on*, Sept 2012, pp. 1–6.

[48] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, Jul. 1987. [Online]. Available: http://doi.acm.org/10.1145/24039.24041

[49] D. H. Friendly, S. J. Patel, and Y. N. Patt, "Putting the fill unit to work: Dynamic optimizations for trace cache microprocessors," in *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society Press, 1998, pp. 173–181.

[50] A. Garg and M. C. Huang, “A performance-correctness explicitly-decoupled architecture,” in *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 41. Washington, DC, USA: IEEE Computer Society, 2008, pp. 306–317. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2008.4771800

[51] A. Garg, R. Parihar, and M. C. Huang, “Speculative parallelization in decoupled look-ahead,” in *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT ’11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 413–423. [Online]. Available: http://dx.doi.org/10.1109/PACT.2011.72

[52] M. Gebhart, B. A. Maher, K. E. Coons, J. Diamond, P. Gratz, M. Marino, N. Ranganathan, B. Robatmili, A. Smith, J. Burrill, S. W. Keckler, D. Burger, and K. S. McKinley, “An evaluation of the trips computer system,” in *ASPLOS ’09*.

[53] N. H. Gehani, H. V. Jagadish, and O. Shmueli, “Composite event specification in active databases: Model & implementation,” in *Proceedings of the 18th International Conference on Very Large Data Bases*, ser. VLDB ’92. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992, pp. 327–338. [Online]. Available: http://dl.acm.org/citation.cfm?id=645918.672484

[54] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. Taylor, “PipeRench: A Reconfigurable Architecture and Compiler,” *IEEE Computer*, vol. 33, no. 4, pp. 70–77, April 2000.

[55] A. GonzÃ¡lez, F. Latorre, and G. Magklis, “Processor microarchitecture: An implementation perspective,” *Synthesis Lectures on Computer Architecture*, vol. 5, no. 1, pp. 1–116, 2010. [Online]. Available: http://dx.doi.org/10.2200/S00309ED1V01Y201011CAC012

[56] J. R. Goodman, J.-t. Hsieh, K. Liou, A. R. Pleszkun, P. B. Schechter, and H. C. Young, “Pipe: A vlsi decoupled architecture,” in *Proceedings of the 12th Annual International Symposium on Computer Architecture*, ser. ISCA ’85. Los Alamitos, CA, USA: IEEE Computer Society Press, 1985, pp. 20–27. [Online]. Available: http://dl.acm.org/citation.cfm?id=327010.327117

[57] V. Govindaraju, C.-H. Ho, and K. Sankaralingam, "Dynamically specialized datapaths for energy efficient computing," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, 2011, pp. 503–514.

[58] V. Govindaraju, "Energy Efficient Computing Through Compiler Assisted Dynamic Specialization," PhD Dissertation, Unversity of Wisconsin-Madison, 2014.

[59] 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*, vol. 33, no. 5, 2012.

[60] V. Govindaraju, T. Nowatzki, and K. Sankaralingam, "Breaking simd shackles: Liberating accelerators by exposing flexible microarchitectural mechanisms," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques "'(PACT)"'*, 2013.

[61] B. Greenstein, E. Kohler, and D. Estrin, "A sensor network application construction kit (snack)," in *Proceedings of the 2Nd International Conference on Embedded Networked Sensor Systems*, ser. SenSys '04. New York, NY, USA: ACM, 2004, pp. 69–80. [Online]. Available: http://doi.acm.org/10.1145/1031495.1031505

[62] W. Gruenewald and T. Ungerer, "A multithreaded processor designed for distributed shared memory systems," in *Proceedings of the 1997 Advances in Parallel and Distributed Computing Conference (APDC '97)*, ser. APDC '97. Washington, DC, USA: IEEE Computer Society, 1997, pp. 206–. [Online]. Available: http://dl.acm.org/citation.cfm?id=523660.786383

[63] 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*, ser. MICRO-44 '11, 2011, pp. 12–23. [Online]. Available: http://doi.acm.org/10.1145/2155620.2155623

[64] J. R. Gurd, C. C. Kirkham, and I. Watson, "The manchester prototype dataflow computer," *Commun. ACM*, vol. 28, no. 1, pp. 34–52, Jan. 1985. [Online]. Available: http://doi.acm.org/10.1145/2465.2468

[65] T. R. Halfill, "AMD Bobcat snarls at Atom," *Microprocessor Report*, August 2010.

[66] P. Hammarlund, A. Martinez, A. Bajwa, D. Hill, E. Hallnor, H. Jiang, M. Dixon, M. Derr, M. Hunsaker, R. Kumar, R. Osborne, R. Rajwar, R. Singhal, R. D'Sa, R. Chappell, S. Kaushik, S. Chennupaty, S. Jourdan, S. Gunther, T. Piazza, and T. Burton, "Haswell: The fourth-generation intel core processor," *Micro, IEEE*, vol. 34, no. 2, pp. 6–20, Mar 2014.

[67] E. N. Harris, S. L. Wasmundt, L. De Carli, K. Sankaralingam, and C. Estan, "Leap: Latency- energy- and area-optimized lookup pipeline," in *Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ser. ANCS '12. New York, NY, USA: ACM, 2012, pp. 175–186. [Online]. Available: http://doi.acm.org/10.1145/2396556.2396595

[68] J. R. Hauser and J. Wawrzynek, "Garp: A MIPS Processor with a Reconfigurable Coprocessor," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1997, pp. 16–18.

[69] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002.

[70] R. Hou, L. Zhang, M. Huang, K. Wang, H. Franke, Y. Ge, and X. Chang, "Efficient data streaming with on-chip accelerators: Opportunities and challenges," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, 2011, pp. 312–320.

[71] L. W. Howes, A. Lokhmotov, A. F. Donaldson, and P. H. Kelly, "Deriving efficient data movement from decoupled access/execute specifications," in *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, ser. HiPEAC '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 168–182. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-92990-1_14

[72] R. A. Iannucci, "Toward a dataflow/von neumann hybrid architecture," in *Proceedings of the 15th Annual International Symposium on Computer Architecture*, ser. ISCA '88. Los Alamitos, CA, USA: IEEE Computer Society Press, 1988, pp. 131–140. [Online]. Available: http://dl.acm.org/citation.cfm?id=52400.52416

[73] G. P. Jones and N. P. Topham, "A comparison of data prefetching on an access decoupled and superscalar machine," in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 30. Washington, DC, USA: IEEE Computer Society, 1997, pp. 65–70. [Online]. Available: http://dl.acm.org/citation.cfm?id=266800.266807

[74] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ser. ISCA '90. New York, NY, USA: ACM, 1990, pp. 364–373. [Online]. Available: http://doi.acm.org/10.1145/325164.325162

[75] K. M. Kavi, R. Giorgi, and J. Arul, "Scheduled dataflow: Execution paradigm, architecture, and performance evaluation," *IEEE Trans. Comput.*, vol. 50, no. 8, pp. 834–846, Aug. 2001. [Online]. Available: http://dx.doi.org/10.1109/12.947003

[76] K. M. Kavi, H. Y. Youn, and A. R. Hurson, "Pl/ps: A non-blocking multithreaded architecture with decoupled memory and pipelines," in *Proc. of the Fifth International Conference on Advanced Computing (ADCOMP '97)*, 1997.

[77] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel, "Rigel: An architecture and scalable programming interface for a 1000-core accelerator," in *ISCA '09*, pp. 140–151.

[78] 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*, ser. MICRO-46. New York, NY, USA: ACM, 2013, pp. 468–479. [Online]. Available: http://doi.acm.org/10.1145/2540708.2540748

[79] K. Koukos, D. Black-Schaffer, V. Spiliopoulos, and S. Kaxiras, "Towards more efficient execution: A decoupled access-execute approach," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ser. ICS '13. New York, NY, USA: ACM, 2013, pp. 253–262. [Online]. Available: http://doi.acm.org/10.1145/2464996.2465012

[80] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic, "The vector-thread architecture," *Micro, IEEE*, vol. 24, no. 6, pp. 84–90, 2004.

[81] D. J. Kuck and R. A. Stokes, "The burroughs scientific processor (bsp)," *IEEE Trans. Comput.*, vol. 31, no. 5, pp. 363–376, May 1982. [Online]. Available: http://dx.doi.org/10.1109/TC.1982.1676014

[82] 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," in *ASPLOS VIII*.

[83] Y. Lee, "Efficient VLSI Implementations of Vector-Thread Architectures," MS Thesis, UC Berkeley, 2011.

[84] 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," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA '11. New York, NY, USA: ACM, 2011, pp. 129–140. [Online]. Available: http://doi.acm.org/10.1145/2000064.2000080

[85] S. Li, J.-H. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, 2009, pp. 469–480.

[86] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch, "Thin servers with smart pipes: Designing soc accelerators for memcached," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013, pp. 36–47. [Online]. Available: http://doi.acm.org/10.1145/2485922.2485926

[87] S. H. Low and D. E. Lapsley, “Optimization flow control—i: Basic algorithm and convergence,” *IEEE/ACM Trans. Netw.*, vol. 7, no. 6, pp. 861–874, Dec. 1999. [Online]. Available: http://dx.doi.org/10.1109/90.811451

[88] A. Lukefahr, S. Padmanabha, R. Das, F. M. Sleiman, R. Dreslinski, T. F. Wenisch, and S. Mahlke, “Composite cores: Pushing heterogeneity into a core,” in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO ’12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 317–328. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2012.37

[89] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, “Tinydb: An acquisitional query processing system for sensor networks,” *ACM Trans. Database Syst.*, vol. 30, no. 1, pp. 122–173, Mar. 2005. [Online]. Available: http://doi.acm.org/10.1145/1061318.1061322

[90] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz, “Smart Memories: A modular reconfigurable architecture.” in *ISCA ’00: Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000, pp. 161–171.

[91] B. Mathew and A. Davis, “A loop accelerator for low power embedded vliw processors,” in *Hardware/Software Codesign and System Synthesis, 2004. CODES + ISSS 2004. International Conference on*, 2004, pp. 6–11.

[92] B. Mathew, A. Davis, and Z. Fang, “A low-power accelerator for the sphinx 3 speech recognition system,” in *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, ser. CASES ’03. New York, NY, USA: ACM, 2003, pp. 210–219. [Online]. Available: http://doi.acm.org/10.1145/951710.951739

[93] D. McCarthy and U. Dayal, “The architecture of an active database management system,” in *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’89. New York, NY, USA: ACM, 1989, pp. 215–224. [Online]. Available: http://doi.acm.org/10.1145/67544.66946

[94] A. Moshovos, D. N. Pnevmatikatos, and A. Baniasadi, “Slice-processors: An implementation of operation-based prediction,” in *Proceedings of the 15th International*

Conference on Supercomputing, ser. ICS '01. New York, NY, USA: ACM, 2001, pp. 321–334. [Online]. Available: http://doi.acm.org/10.1145/377792.377856

[95] 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*, ser. PLDI '13. New York, NY, USA: ACM, 2013, pp. 495–506. [Online]. Available: http://doi.acm.org/10.1145/2491956.2462163

[96] "Openrisc project, http://opencores.org/project,or1k."

[97] A. Pajuelo, A. González, and M. Valero, "Speculative dynamic vectorization," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ser. ISCA '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 271–280. [Online]. Available: http://dl.acm.org/citation.cfm?id=545215.545246

[98] G. Papadopoulos and D. Culler, "Monsoon: an explicit token-store architecture," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, May 1990, pp. 28–31.

[99] 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," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013, pp. 142–153. [Online]. Available: http://doi.acm.org/10.1145/2485922.2485935

[100] M. S. Park, S. Kestur, J. Sabarad, V. Narayanan, and M. Irwin, "An fpga-based accelerator for cortical object classification," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, March 2012, pp. 691–696.

[101] A. Poulovassilis, G. Papamarkos, and P. T. Wood, "Event-condition-action rule languages for the semantic web," in *Proceedings of the 2006 International Conference on Current Trends*

in Database Technology, ser. EDBT'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 855–864. [Online]. Available: http://dx.doi.org/10.1007/11896548_64

[102] A. Putnam, S. Swanson, K. Michelson, M. Mercaldi, A. Petersen, A. Schwerin, M. Oskin, and S. J. Eggers, "The Microarchitecture of a Pipelined WaveScalar Processor: An RTL-based study," Tech. Rep. TR-2005-11-02, 2005.

[103] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz, "Convolution engine: Balancing efficiency & flexibility in specialized computing," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013, pp. 24–35. [Online]. Available: http://doi.acm.org/10.1145/2485922.2485925

[104] W. Ro, S. Crago, A. Despain, and J.-L. Gaudiot, "Design and evaluation of a hierarchical decoupled architecture," *The Journal of Supercomputing*, vol. 38, no. 3, pp. 237–259, 2006. [Online]. Available: http://dx.doi.org/10.1007/s11227-006-8321-2

[105] E. Rotenberg, S. Bennett, and J. E. Smith, "Trace cache: a low latency approach to high bandwidth instruction fetching," in *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society, 1996, pp. 24–35.

[106] A. Roth and G. Sohi, "Register integration: a simple and efficient implementation of squash reuse," in *MICRO '33*, 2000.

[107] A. Roth and G. S. Sohi, "Speculative data-driven multithreading," in *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, ser. HPCA '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 37–. [Online]. Available: http://dl.acm.org/citation.cfm?id=580550.876429

[108] R. M. Russell, "The CRAY-1 Computer System," *Communications of the ACM*, vol. 22, no. 1, pp. 64–72, January 1978.

[109] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, S. W. Keckler, D. Burger, and C. R. Moore, "Exploiting ILP, TLP and DLP with the Polymorphous TRIPS Architecture," in

ISCA '03: Proceedings of the 30th Annual International Symposium on Computer Architecture, June 2003, pp. 422–433.

[110] 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*, ser. ISCA '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 440–451. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337159.2337210

[111] 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," *Micro, IEEE*, vol. 32, no. 2, pp. 8–19, March 2012.

[112] J. E. Smith, "Decoupled access/execute computer architectures," in *Proceedings of the 9th Annual Symposium on Computer Architecture*, ser. ISCA '82. Los Alamitos, CA, USA: IEEE Computer Society Press, 1982, pp. 112–119. [Online]. Available: http://dl.acm.org/citation.cfm?id=800048.801719

[113] J. E. Smith and R. Nair, "The architecture of virtual machines," *Computer*, vol. 38, no. 5, pp. 32–38, 2005.

[114] A. Sodani and G. S. Sohi, *Dynamic instruction reuse*. ACM, 1997, vol. 25, no. 2.

[115] *SPEC CPU2006*. Standard Performance Evaluation Corporation, 2006.

[116] S. Subramaniam and G. H. Loh, "Fire-and-forget: Load/store scheduling with no store queue at all," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 39. Washington, DC, USA: IEEE Computer Society, 2006, pp. 273–284. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2006.26

[117] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin, "Wavescalar," in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 36, 2003, pp. 291–. [Online]. Available: http://dl.acm.org/citation.cfm?id=956417.956546

[118] D. Talla and L. K. John, "Mediabreeze: A decoupled architecture for accelerating multimedia applications," *SIGARCH Comput. Archit. News*, vol. 29, no. 5, pp. 62–67, Dec. 2001. [Online]. Available: http://doi.acm.org/10.1145/563647.563659

[119] M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. P. Amarasinghe, and A. Agarwal, "Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams," in *Proceedings of the 31st Annual International Symposium on Computer Architecture*, 2004, pp. 2–13.

[120] S. Vajapeyam, P. J. Joseph, and T. Mitra, "Dynamic vectorization: A mechanism for exploiting far-flung ilp in ordinary programs," in *Proceedings of the 26th Annual International Symposium on Computer Architecture*, ser. ISCA '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 16–27. [Online]. Available: http://dx.doi.org/10.1145/300979.300981

[121] S. P. Vanderwiel and D. J. Lilja, "Data prefetch mechanisms," *ACM Comput. Surv.*, vol. 32, no. 2, pp. 174–199, Jun. 2000. [Online]. Available: http://doi.acm.org/10.1145/358923.358939

[122] 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 *ASPLOS '10*.

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

[124] L. Wu, R. J. Barker, M. A. Kim, and K. A. Ross, "Navigating big data with high-throughput, energy-efficient data partitioning," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*. ACM, 2013, pp. 249–260.

[125] 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*. ACM, 2014, pp. 255–268.

[126] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee, “Chimaera: a high-performance architecture with a tightly-coupled reconfigurable functional unit,” in *ISCA '00*.

[127] C. Zilles and G. Sohi, “Execution-based prediction using speculative slices,” in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, ser. ISCA '01. New York, NY, USA: ACM, 2001, pp. 2–13. [Online]. Available: http://doi.acm.org/10.1145/379240.379246

[128] C. B. Zilles and G. S. Sohi, “Understanding the backward slices of performance degrading instructions,” in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ser. ISCA '00. New York, NY, USA: ACM, 2000, pp. 172–181. [Online]. Available: http://doi.acm.org/10.1145/339647.339676