Behavior Specialized Processors

by

Tony Nowatzki

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2016

Date of final oral examination: 12/07/2016

The dissertation is approved by the following members of the Final Oral Committee:
Gurindar Sohi, Professor, Computer Science
Mark Hill, Professor, Computer Science
Somesh Jha, Professor, Computer Science
Jeffrey Linderoth, Professor, Industrial Systems Engineering
Karthikeyan Sankaralingam, Associate Professor, Computer Science

For My Dad.

I remember when my dad brought me to Unisys on "take your kid to work day." At some point, while we were hanging around in his office, he drew me my first hardware block diagram – boxes representing cores, caches, and memory. I had no idea what he was talking about, but I always appreciated that he spoke to me as if I were smart enough to understand. Thank you, Dad, for instilling in me the love of understanding these crazy complex machines.

ACKNOWLEDGMENTS

This dissertation would not have been possible without the support, encouragement and critical advice from many people.

Chief among those is my advisor, Karu Sankaralingam. He taught me much over these last years – how to express arguments, how to look for topics with the most impact, and how to do rigorous research without getting lost in the details. He also inspired me to be passionate and speak passionately about my ideas. What I am most thankful for is how he saw capabilities in me that I didn't know I had or could ever have. Karu, I hope that I can inspire others in the same way you did for me, thank you.

My committee was incredibly helpful throughout my graduate studies, and I want to sincerely thank them for their feedback and advice. I consider Mark Hill to be my academic role model; I hope to be as wise as him someday, while remaining as personally balanced and strong. Guri challenged me intellectually in ways that no one else could, and his perspective and critical feedback has been invaluable. Thank you Somesh for encouraging me to be a more confident speaker, and many thanks to Jeff Linderoth for teaching me about optimization and modeling, which was foundational for much of my work. I would also like to thank the other computer science faculty at Wisconsin. This especially includes David Wood for his thoughts and insights (and as a never ending source of great stories).

Thanks to Google for the fellowship award – this allowed me to focus more on research, and it also incentivized Karu to keep me around longer, thank you *so* much.

I owe a special thanks to two faculty members from the University of Minnesota. The first is Professor Wei-Chung Hsu, whose passion for computer architecture and teaching capability was what got me initially excited about the field. Also, I am grateful to have worked for my undergrad advisor Paul Woodward, who first introduced me to the academic and publishing world, and the "coolness" of having a paper with your name on it.

I am grateful to have had such fantastic folks to work with in the Vertical Research Group. I learned so much from Venkatraman Govindaraju, who was my fearless leader on many research projects early in my academic career. He taught me that complicated problems could be solved easily with the right attitude: "it's nothing dude." Marc de Kruijf, you were like my second advisor in my first couple years. To Chen-han and the other `DySER BROS`, it was a blast. Special thanks to Zach Marzec and Chris Frericks for your bizarre sense of humor and making the office fun. Similarly, Jai and Raghu, thanks for your witty sarcasm and love of sparking arguments with Karu; it was sorely missed when you left. To my closest work buddies: Vijay, I've always enjoyed our conversations and your friendship. Vinay, I cannot thank you enough for all of the hard work you've put into our collaborations; I have your back if you need a favor. Finally, I only just met Mitchell, Preyas, William, Alex, and Junyong, but I can see your potential and know you'll go far.

I've been fortunate to have so many other great collaborators. To Michael Ferris and Cristian Estan, I really enjoyed our collaborative writing experience. It was equally a pleasure working with Amir Yazdanbakhsh, Lorenzo De Carli, and Michael Sartin-Tarm – I truly learned a lot from you.

I will sincerely miss the many other incredible architects that I have been lucky to share the same floor with: Hongil, Joel, Lena, Marc (while he stuck around), Swapnil, Srinath, Rathijit, Somayeh and Hamid. A special thanks to Jason, Emily, Shoaib, Jayneel, and Gagan for feedback in reviewing and critiquing my papers.

To my dear family, I owe you a debt that can never be repaid. Mom and Dad, you have been there for me without fail, through fear, uncertainty, and doubt. Steve and Paul, thank you for being such supporting big brothers, you have always been my role models since I was old enough to remember. Also Paul, I forgive you for trying to dissuade me from pursuing my PhD – turns out I *am* crazy enough for this.

Hamisha, thanks for putting up with me.

Newsha – this has been quite a ride, and I know this is the end of just the first adventure we are going to have in our life together. Thank you for supporting me when I needed you the most, and thank you for always believing in me.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Improvements in the performance and energy consumption of general purpose processors have slowed dramatically over the last decade. This is due to the combined effect of breakdowns in transistor scaling, causing severe chip-level power limitations, and monolithic and inefficient general purpose microarchitecture. Over the last decades, and especially in recent years, the community has turned towards domain specific processors, where general purpose programmability is jettisoned for efficiency. These trends threaten the future of general purpose architecture innovation.

This dissertation explores a promising alternative to the domain-specific approach: to specialize for broad properties of programs which span across domains, which we refer to as program *behaviors*. Programmable hardware engines which exploit these characteristics – *behavior specialized accelerators* – can be integrated into general purpose cores and used as offload engines to transparently improve their performance and energy efficiency during amenable program phases. This work addresses key challenges in accelerator modeling, microarchitecture, compilation and design-space exploration.

The evaluation and analysis herein suggests several key findings. First a small number of exploitable program behaviors can be used to characterize a majority of applications. Second, dataflow architectures become practical and useful in hybrid execution with a general purpose core. Third, synergistic behavior-specialized accelerators combined with simple general core pipelines can facilitate disruptive microprocessor tradeoffs, enabling mobile-class processor energy-efficiency with desktop-class performance. In addition to the architectural discoveries, this dissertation proposes a modeling methodology which enables rapid exploration of behavior specialized processors, as well as a mathematical formulation for declarative and optimal instruction scheduling on these architectures. Overall, the paradigm of behavior specialization demonstrates that the future for general purpose architecture innovation is bright.

1 INTRODUCTION

For many years, general-purpose processor architectures and microarchitectures took advantage of Dennard scaling and Moore's Law, exploiting increased circuit integration to deliver higher performance and lower energy computation. The benefits of microarchitecture-only modifications were clear; applications were transparently becoming faster and more energy efficient without any involvement from the programmer or compiler. The synergy between continuous device scaling and the leverage that it gave for augmenting general purpose microarchitecture has been a foundation of technological and human advancement over the last decades.

Unfortunately, and as has been lamented numerous times [1, 2, 3], those physical scaling trends are slowing, and the limited generational improvements to traditional general purpose architectures are now apparent. For instance, the latest high-end server processors from Intel (the largest vendor of general purpose processors) has achieved only 5% improvement in performance in its last generation (the average improvements over the last four generations were only slightly better at 9% [4]). In previous decades this was typically as high as 30% or more per processor generation.

In spite of the usefulness and value in high performance general purpose cores across server, desktop and mobile (as evidenced by the large research investments by top companies), the limitations in their improvements have caused a surge of interest in more narrowly-applicable architectures, which sacrifice generality in the hope of continuing improvements for some important tasks. By some measures, this has been a largely successful endeavor: often *many* factors or even orders-of-magnitude improvement are possible.

In fact, it is this stark disparity between the progress in general purpose architectures and domain-specific and application-specific architectures that leads to the fundamental question of this work: *Is it possible to achieve the benefits of specialization while retaining general purpose flexibility?*

This dissertation explores one possible route to addressing this challenge, and ultimately suggests that it is possible to have the best of both worlds, at least to a large extent. Its fundamental insight is that rather than specializing for applications, or even domains, we can instead specialize for more broad and generally applicable program behaviors. The principle being to find a small number of synergistic behaviors to specialize for, and create specialized hardware that is both integer factors more efficient than a general purpose counterpart, while still being broadly useful. The challenges in using such an approach simply becomes a matter of creating an effective architecture organization, a practical execution model, a compilation strategy that limits programmer burden, and modeling techniques that allow rapid exploration of the behaviors and specialized hardware.

The remainder of this chapter will focus on first defining what is meant by specialization, giving some definitions and historical context. Subsequently, we discuss the relevant form of specialization for this dissertation, *behavior specialization* and how that presents a unique opportunity. Then we will cover the main contributions of this work and the organization of the dissertation.

1.1 Hardware Specialization

Definitions *Hardware specialization* is the application of modifications to some reference design which improves its capabilities (defined by a metric, e.g. performance) for some set of workloads, while reducing its capabilities for some "larger" set of workloads. Specialization can be *strict*, meaning that the design is restricted in its applicability to perform certain applications. Specialization can also be *non-strict*, meaning that the design is simply tuned to favor certain applications.

A *behavior* is an aspect of a program that can be conceivably exploited with hardware specialization, *and* does not describe the entire computation being performed – in other words, a behavior is a general characteristic of a program on which hardware specialization relies. For example, the presence of highly-biased control, data parallelism or memory irregularity are all program behaviors. Non-behaviors in this context would be the prevalence of basic blocks with odd numbers of instructions (no conceivable way to exploit), or programs which compute a Fast Fourier Transform using the Cooley-Tukey algorithm (because this describes the entire computation, not a general aspect of it).

A related concept to specialization is *heterogeneity* – the inclusion of multiple components which serve the same purpose but are specialized differently. While heterogeneity generally implies specialization, it is possible to have specialization without heterogeneity (e.g. application-specific hardware), and heterogeneity without strict specialization (e.g. a system that contains two general purpose processors, one "big" and one "small").

Caveats The definitions above pose some interesting conundrums, and sometimes lead to unsatisfying discourse. We call these out here, and explain how we attempt to resolve them as best as possible.

First, defining what designs or design aspects are specialized is difficult because there is no definitive *standard* reference design with which to compare, no *standard* set of workloads with a particular composition, and no *standard* metric for every situation. For example, in a big-little system, which contains a big out-of-order processor and a little inorder processor, it would be fair to say that the OOO processor is specialized for performance on workloads with high instruction level parallelism. It would be equally fair to say that the inorder-core is specialized for energy-efficiency on memory-latency bound workloads. Future discussions address this issue by being clear about the reference design, intended workloads, and metrics.

Distinguishing strictness is also difficult. It would be tempting to use computability theory to make such distinctions, classifying architectures into combinational logic, finite state machine-capable, pushdown automata-capable, Turing machine-capable, and so on. By such definitions, most programmable architectures would be technically equivalent in terms of strictness, as they would be Turing complete. This would ignore the quite meaningful distinction of one architecture having vastly superior performance for some workloads. An example would be executing floating point instructions in hardware versus being emulated in software. To make future discussions intuitive and meaningful, we would consider the above type of order-of-magnitude improvements to be a form of strict specialization.

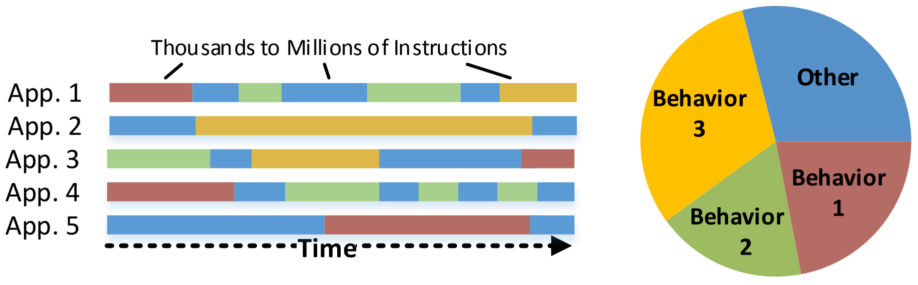
By its definition, what classifies as a behavior is subjective and changes over time as new hardware specialization mechanisms are invented. At one time the prevalence of highly biased control may not have been obviously an exploitable program behavior, but now it clearly is. The harder part of this definition is in drawing the line between a general behavior, and one that describes an "entire computation". For example, is a stencil computation ($B_{i,j} = \sum_{k=0}^{K} \sum_{l=0}^{L} A_{i+k,j+l}$) a complete computation, and therefore not a general behavior, even though it is a smaller part of many algorithms like a neural network algorithms and image filters? This dissertation will simply concede that it is a difficult question, and rely on the intuition and evidence for which features are general or application specific.

Historical Context The concept of specialization has been critical to design decisions for decades in the field of computer architecture, and we only scratch the surface of the historical context here. Even the very first electronic general purpose processor, the ENIAC, had a heterogeneous mix of multiplier, divider, and square-rooter units. Ever since, more specialized components have been making their way into general purpose processors' instruction sets.

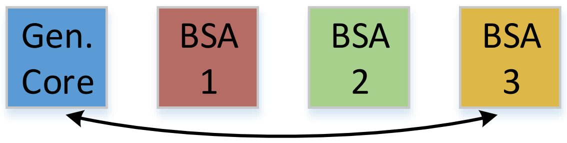
Beginning in the 1970s, the well-known debate between simple (aka reduced) instruction sets (RISC) and complex instruction sets (CISC), was largely an issue of specialization. Though not perhaps originally conceived of in terms of specialization, CISC can be viewed as a specialization of RISC for certain common instruction patterns, with the goal of reducing instruction size and frontend pipeline energy. A decade later, VLIW processors specialized for instruction level parallelism by making this parallelism explicit in the ISA. In the 1990s, ideas from large-scale parallel vector machines made their way into general purpose architectures in the form of short-vector SIMD instructions – a form of specialization for data parallelism.

In the same time period, there were many efforts to perform automatic customization to applications. A small subset of examples includes customizing VLIW processors [5, 6], or even creating custom ASICs for program phases [7].

Trends in specialization over the last decade have continued to coarsen the program granularity at which specialization is being performed. The now ubiquitous GPGPU (general purpose graphics processing unit) serve as specialized offload engines for program regions with massive parallelism.



(a) A small number of behaviors can characterize the execution of a majority of programs



(b) Execution is transferred between general core and behavior specialized accelerators(BSAs)

Figure 1.1: Behavior Specialization Paradigm

Most recently, domain specific architectures have begun to become popular for tasks that have typically been performed by general purpose processors, ranging from machine learning [8, 9], cryptography [10], XML processing [10], regular expression matching [11, 12], H.264 decoding [13], to databases [14, 15, 16] and many others.

To summarize, specialization has become both more prevalent and coarse-grained over time (from instructions to entire program regions). As will be described, the concept of behavior specialization keeps this trend of coarsening the targeted granularity of specialization, but backs away from application or domain specialization.

1.2 Behavior Specialization

The main principle of behavior specialization is that programs execute in phases, where each phase can be characterized by broad yet distinct characteristics or program behaviors (Figure 1.1(a) shows a cartoon example).

Once these behaviors are identified, specialized hardware engines are created to be highly effective at executing a subset of these behaviors – we call these behavior specialized accelerators (BSAs). The ultimate goal is to improve a majority of codes (thus retaining generality) by letting the most appropriate BSA execute each program phase. This is achieved by migrating the execution dynamically between the cores and accelerators at run-time (see Figure 1.1(b)).

Given the above approach, there are three hypotheses that must be true for behavior specialization to be effective:

1. In order to limit the amount of hardware and compiler overhead, there must be a *small* number of behaviors[1], that can be determined through either static or dynamic program analysis, which together can characterize a large class of general workloads.

2. To achieve enough overall gains, it must be possible to create a specialized architecture for each behavior, which can achieve significant (at least integer factor) improvements on the subset of workloads it was designed for.

3. The overheads for communicating to and from the core and memory system to the accelerator must be low.

This dissertation will attempt to show the above three hypotheses are true.

1.3 Contributions

This dissertation identifies a paradigm shift (due to device scaling and application trends) towards achieving general purpose efficiency by focusing specialization efforts on broad program behaviors, and it proposes an execution model and core organization that can exploit this paradigm. The potential impact of this is to enable continual microprocessor improvements for performance and energy-efficiency without the need to rely on device scaling. The specific contributions are in terms of the modeling methodology, identified behaviors for specialization, execution model/architecture organization for behavior specialization, and compiler techniques.

[1]Or a set of behaviors which can be co-optimzied for.

Modeling Methodology: Programmable acceleration techniques often must dispense with existing ISAs to achieve efficient execution, and require compiler support to employ code transformations to make them useful. Building compilers and simulators to evaluate each proposed architecture is impractical in terms of development time and evaluation-consistency. The first main contribution is a proposal for an alternate modeling methodology for modeling accelerators. We specifically propose a graph-based modeling framework which abstracts microarchitecture, application, and compiler interactions into graph transformations. Though it abstracts some aspects of microarchitectural execution, it retains high-accuracy, enables many accelerators to be evaluated in the same framework, and can produce models that can be described with either few lines of code or simple edge-based descriptions. This will serve as the modeling infrastructure for the remainder of this dissertation.

Demonstrating Synergy between VonNeumann & Explicit-Dataflow: Though there are some existing behavior-specialized accelerators for regular codes (SIMD for data-parallel code regions), we lack effective acceleration techniques for irregular codes. The next main contribution is to show the potential of specialization for irregular codes by exploiting the synergy between traditional VonNeumann out-of-order (OOO) machines and explicit-dataflow architectures. In particular, there are two main findings: First, that dataflow machines are more effective when control-flow decisions and value communication is not on the critical path, or when control decisions are unpredictable. Second, that it is possible to build an explicit-dataflow co-processor that has low power and area overheads compared to a conventional OOO processor. This suggests a viable answer to the long-standing debate in computer architecture over the effectiveness of VonNeumann versus Dataflow machines – each one is simply better at programs with different behaviors.

Design-Space Exploration of ExoCore Systems The basic architectural organization and execution model of the behavior specialization paradigm is called ExoCore. The primary challenge in adopting this organization is to determine the types of accelerator engines which can be combined profitably, without costing excess power and area. The third contribution is the description of a behavior space which corresponds to a plausible set of behavior-specialized accelerators, which in our evaluation achieves up to 2.0× average speedup and 1.7× average energy-efficiency, depending on the baseline general purpose core. In addition, we perform a design space exploration across cores, accelerators, and workloads, and demonstrate how various ExoCore instances can enable new design tradeoffs.

Compiler Techniques Finally, one of the primary challenges, from a compilation perspective, is mapping instructions onto the behavior specialized accelerators to maximize performance or energy-efficiency. Because behavior-specialized accelerators are explicitly targeted towards certain program behaviors, it is natural for them to expose more of their underlying hardware up to the compiler than a traditional architecture – these are termed *spatial architectures*. The standard approach is to use heuristic-based schedulers, which are difficult to implement and have solutions whose optimality is difficult to characterize. The fourth main contribution is a formulation of a mathematical model for spatial architecture scheduling using purely linear constraints with integer variables (as an integer linear program), which enables off-the-shelf solvers to efficiently and *exactly* solve the problem. Furthermore, this approach is shown to be general across a large variety of spatial architectures, meaning that scheduling support for architectural features can be trivially ported across architectures.

1.4 Organization

The organization of this dissertation outlined in Table 1.1, along with the relation to the author's prior work. We first discuss an execution model and architectural organization called ExoCore, and make a case for its essential design decisions (Chapter 2). We then discuss a modeling framework which can simultaneously model cross-level aspects of such an architecture, including the microarchitecture, compiler, and application interactions (Chapter 3). Then we focus on one of the largest problems

Chap.	Topic	Author's Related Prior Work
2	ExoCore Organization and Execution Model	ISCA 2015 [17], ASPLOS 2016 [18] (execution model), HPCA 2016 [19] (dynamic compilation)
3	Accelerator Modeling Technique	CAL 2015 [20],ASPLOS 2016 [18]
4	Dataflow Specialization	ISCA 2015 [17]
5	Multi-behavior ExoCore	ASPLOS 2016 [18]
6	Spatial Architecture Scheduling	PLDI 2013 [21], ISCA 2015 [17] (SEED Extensions)

Table 1.1: Dissertation Organization and Relation to Author's Prior Work

in behavior specialization – coming up with exploitable behaviors and associated hardware for programs with irregular control or memory access (Chapter 4). Afterwards, we generalize, and explore a design space of many accelerators and general purpose cores across a variety of workload domains (Chapter 5). Finally, we construct a mathematical scheduling framework for managing the resource-exposed architectures that are typically integrated as accelerators (Chapter 6).

2 THE CASE FOR AN EXOCORE PROCESSOR

This chapter develops the ExoCore concept, which is the architecture, execution model, and compilation model that serves as the basis for the remainder of the work in the dissertation. In short, ExoCore is a processor organization in which cores are composed of multiple programmable accelerators, each limited in scope but more efficient given certain program behaviors.

In this chapter, we make a case for studying an ExoCore design and execution model, starting by giving an overview of the hardware (Section 2.1) and compilation approach (Section 2.2). We then discuss the reasons behind focusing on programmer-transparent accelerators (Section 2.3), region-based execution (Section 2.4), and in-core integration (Section 2.5). Finally, we discuss what elements of the system are modeled, and why we made these decisions (Section 2.6).

2.1 ExoCore Organization Overview

ExoCore is a microprocessor core organization, consisting of a general core and several behavior specialized accelerators (BSAs), which are programmable to target many program phases. Accelerators "sit behind" the first level cache, and communicate with the core either via a dedicated data bus, or they have access to the cache hierarchy directly. Figure 2.1(a) shows an abstract system with 4 BSAs, and Figure 2.1(b) shows how these can be integrated into a multicore system using the standard approach where each core is identical.

Execution Model The essential execution model of an ExoCore is that an accelerator is handed execution on a phase-based program granularity corresponding to a static program region, and the accelerator hands back the execution to the general purpose processor when the region's phase is complete. More specifically, accelerators will execute for the duration of a fully-inlined trace, loop, nested loop, or function call (we discuss the reasons and implications later in Section 2.4). Since only one accelerator is active a time, there is no contention at the cache interface. Also, accelerators never

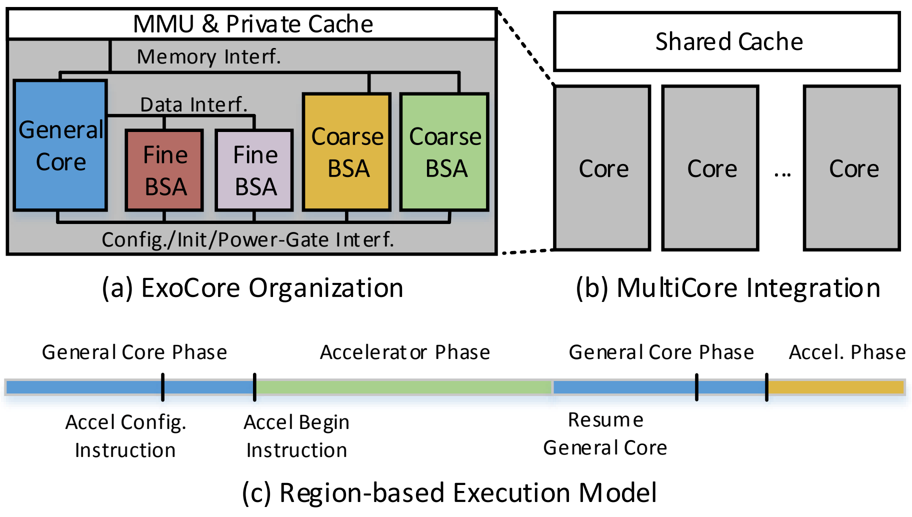


Figure 2.1: ExoCore Architecture and Execution Model Overview

communicate with each other; there is always a general purpose core phase between accelerator phases, which reduces the complexity of integration.

Configuration Stage In this work we consider reconfigurable accelerators which do not fetch their own instructions from memory – rather they are configured for each upcoming program region. Therefore, it is useful to begin streaming configuration data before the region starts. Figure 2.1(c) overviews this processes. The configuration stage for a BSA would start at the moment when region entry is known to be imminent. At this point, any instruction or configuration information for that region is streamed in to the accelerator, as well as the initialization of any region-invariant constants.

Region-Lifetime Prediction It may be unknown how long a program region lasts at compile time, and the overheads of switching to the BSA may be too high if the duration is too short. Therefore, for any given program region, both the compiled accelerator code and general purpose core code are available for execution. Also, if the region lasts long enough, it will be worthwhile to power-down the non-stateful components of the OOO core. To enable these decisions, we keep a simple direct-mapped table of the running average times of different BSA regions. If it is predicted to be short, we do not enter the BSA region, instead simply using the general purpose core version. If it is long enough, once the accelerator execution begins, the relevant components of the GPP core become either clock-gated or power-gated.

Host Instruction Extensions To enable the region-based execution model, two instructions are added to the host processor. The first, `ACCEL_CONFIG`, is inserted at the earliest dominating basic block in the host code, which signals the BSA unit to begin the configuration stage. This instruction contains the relevant memory addresses for configuration bits. The second added instruction, `ACCEL_BEGIN`, is a type of conditional branch, which transfers control to the BSA if it is predicted to run for long enough to mitigate overheads. If appropriate for the accelerator, this instruction signals the live value transfer from GPP registers.

Fine and Coarse Grain Integration Accelerators in ExoCore can be integrated either in fine or coarse grain fashion. Fine grain integration simply implies that the OOO core is active during the phase – usually for fetching data for the active BSA. The execution model is similar, but of course there are no benefits of power-gating the host core. Fine grain accelerators are typically integrated with a vector interface to send data (because otherwise the overhead of communication is far too high). Coarse grain accelerators have access to the same cache interface as the host general purpose core – sharing the memory management unit for supporting virtual memory, as well as any page-table walkers, which can fetch address translations concurrently with the accelerator.

Context Switching There are two basic strategies for handling context switching. The first is to checkpoint state at known intervals of the program, and throw away any uncommitted state at the time of a context switch. The alternative is to save live operands, and make these part of the architectural state. We choose the appropriate strategy for each BSA, basis based on the overheads of saving architectural state.

2.2 Compilation Overview

In an ExoCore approach, behavior specialized accelerators (BSAs) are programmable but not programmer-exposed. Therefore, it is the compiler's responsibility to transform the code to best employ the BSAs. This work considers a compilation environment consisting of profiling and "static" compilation – though we will discuss how to apply the same sort of architecture and execution model to other compilation environments later in this section.

There are four main responsibilities of a BSA compiler, corresponding to its required phases, as described next:

1. **Program Region Characterization** The first step is to characterize regions to determine whether they can be legally and likely profitably offloaded. For the BSAs developed in this work, path-profiling information was sufficient [22] from the perspective of dynamic information, which is gathered through offline profiling on training data.

2. **Per-region Accelerator Selection** The next step is to choose which accelerator is most appropriate for each region – as it is often the case that it is legal to apply more than one accelerator. The basic approach is to estimate the metric of interest relative to that of the baseline processor (performance improvement relative to the OOO core) using a combination of program IR analysis and profiling information. Since regions can be nested (ie. trace inside loop, loop inside nested loop, nested loop inside function), the estimations are made at multiple granularities, and a dynamic programming algorithm is used to select the best combination of sub-regions and accelerators. We use an algorithm that we go into much more detail on in Section 5.3.

3. **Accelerator Program Transformations** With the decision made of which accelerator to apply, the program is first transformed to take advantage of the accelerator (eg. Vectorization for data parallel accelerators). A common challenge across BSAs is that they put extra burden on the compiler for scheduling instructions onto their hardware substrate. This is because they necessarily expose the inner workings of their hardware in ways that are beyond that

of traditional ISAs, to get further exploitation of particular program behaviors. For example, an accelerator that targets the behavior of code having large-datapaths may expose the datapath configuration to the compiler. We address this problem by creating general and mathematically-based code schedulers which are useful across BSAs. The dissertation covers this in detail in Chapter 6.

4. **Accelerator Code Injection** Finally, the accelerator code is compiled into the binary. The host-processor instructions for configuring regions are inserted into the host code. For the fine-grain accelerators, there is an additional step of introducing vectorized-communication instructions into the core to transfer live values back and forth. Earlier work covers this process in detail [23].

Fragility of Compiled Binaries Since each BSA has its own ISA, and because the ISA is exposing the hardware and is fragile to hardware changes, the above compilation approach is non-ideal – it lacks binary compatibility. One solution is to check the version of the hardware at load time, and prevent invocation of accelerator regions which have different versions than the one it was compiled for. This is acceptable, but means recompilation is necessary to take advantage of changing hardware – losing some of the benefits of general purpose machines. There are at least two other compilation environments that are likely much easier to adopt, which we describe below:

Application to Software-Transparent BSA Compilation One possible approach is to use hardware-assisted dynamic binary translation, which we show one implementation of in Figure 2.2. There are three main elements: The first, region identification, monitors retiring instructions to construct candidate regions and selects the most opportune for translation and optimization. Selected regions are sent to the second element, which would analyze the region and perform the accelerator selection and program transformations required. This produces configuration information for the BSA and, if appropriate, modified software for the supporting processor. The third element, region injection,

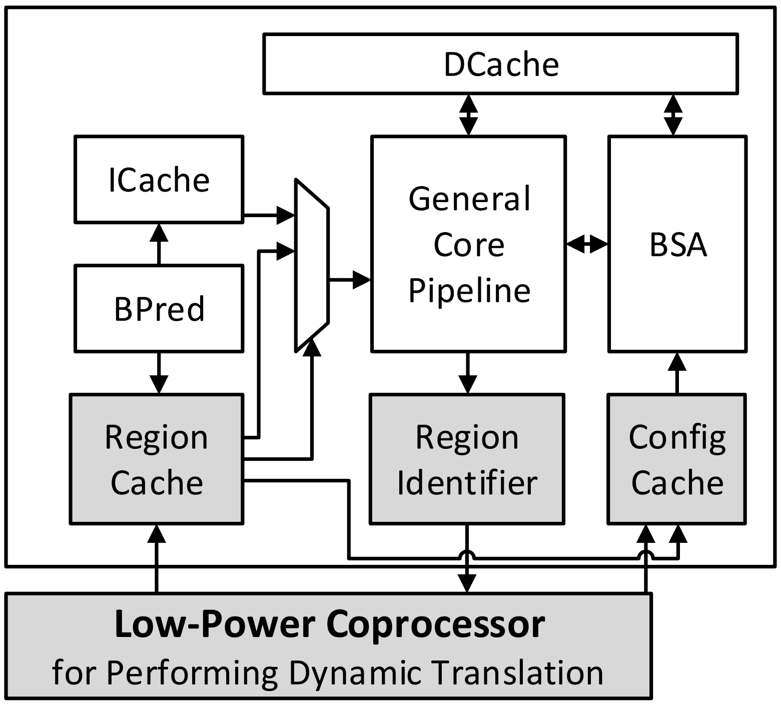


Figure 2.2: ExoCore Modified for Dynamic Binary Translation

stores the generated software in a special region cache. Future invocations of the region execute from this cache, and initiate the execution of the BSA.

The choice of using a low-power micro-processor integrated with the standard processor minimizes design effort and facilitates future translation algorithm changes. While adding a coprocessor does increase design and verification complexity, the overhead should be small since simple and open-source designs are available to leverage.

This paradigm has been studied in the context of a single BSA in work co-written by the author [19], and should be extensible to multiple accelerators.

Application to Virtual Machines Another plausible approach for using ExoCore would be inside existing virtual machine run-times like Java. A significant amount of software is written for Java, especially in the mobile environment, and already Java run-times use dynamic compilation as a way to speed up their standard interpreters. Only simple modifications seem necessary to use ExoCore

Programmer-Exposed Accelerators

Accel. (Domain)	Base	SP	EN
Convolution Eng. [24]	SIMD	6	9
DianNao [8] (Mach-Lrn.)	SIMD	118	21
HARP [15] (Database)	OOO4	8	8
H.264 [13] (Video)	OOO3	3	500
LINQits [29] (C# Queries)	OOO2	18	15
NPU [31] (Approx Comp)	OOO4	3	4
WIDX [14] (Database)	OOO4	3	7
Cambricon [34] (DNN)	OOO6	98	

Programmer-Transparent Accelerators

Accel.	Base	SP	EN
BERET [25]	IO1	1.1	1.5
CCA [26]	OOO4	1.3	
C-Cores [27]	IO2	1.0	1.5
DynaSpAM [28]	OOO8	1.4	1.2
Comp. Cores [30]	OOO3	1.0	1.2
DySER [32]	OOO2	3.2	1.7
Libra [33]	IO1	10.0	0.3
Chainsaw [35]	OOO4	1.2	1.5

Table 2.1: Examples of Opaque and Transparent Accelerators Base: Baseline; SP: Avg. Speedup; EN: Avg. Energy Reduction

in this setting, as region selection, program transformation and region injection can happen as before. The only change might be that after a region is identified as "hot" for dynamic compilation, it is instrumented and profiled to aid with accelerator selection. This phase may not even be necessary depending on the BSAs in question.

2.3 Why Programmer-Transparent Accelerators?

As mentioned, the following three sections discuss the major design choices of ExoCore – starting with how the accelerator is exposed to the programmer. This discussion is oriented around Table 2.1, which gives examples of accelerators that are either programmer-exposed and programmer-transparent, along with the evaluation baseline and average speedup and energy reduction.

Programmer-exposed techniques rely on the programmer's knowledge of the application in order to create specialized hardware which exploits the application's concurrency, computational needs, communication patterns, data-reuse needs, and coordination hardware [36]. Therefore, they able to attain significant potential performance and energy-efficiency advantages over a purely general approach.

With the considerable advantages of opaque specialization come several drawbacks. The most self-evident is the programmer burden and associated portability challenges. Also, these accelerators tend to be limited in their applicability, and are focused to either a particular workload, application, or domain. Lastly, we posit that the nearing end of Moore's law [1, 3] precludes a "sea" of non-general-purpose cores because of area constraints. Some amount of hardware generalization, for all but the most frequent tasks, is inevitable.

On the other hand, a programmer-transparent approach typically provides much more generality through flexible hardware (of course subject to being applicable for the program behaviors they were designed for). While transparency is desirable, it seems to come at a high price. According to the sample in Table 2.1, transparent accelerators seem to provide much less benefit over their respective baseline processors. This is somewhat intuitive, but also misleading. It is true that programmers can provide additional information and larger scale algorithm and data-layout transformations than what is typically possible in a modern compiler. However, these numbers could easily be inflated relative to transparent techniques because the ignore difficult-to-improve irregular workloads with less specialization potential.

Therefore, our goal is to explore how far we can push a general, programmer-transparent paradigm. While we know it is possible to build *highly* efficient programmer-exposed hardware for specific problems, we currently do not know how close we can come to bridging the gap with without the burdensome requirement of involving the programmer.

2.4 Why Region Based?

The accelerators that we target are what we call *region-based*, meaning that they are programmed to target fully inlined program regions of a fixed maximum size. The reason is simple: if we only are executing a finite number of instructions in a large enough period of time, we can simplify the hardware by eliminating the need to perform general purpose instruction fetch, and thereby reduce energy consumption significantly. This is a form of strict specialization for the behavior of region prevalence. The negative consequence, of course, is that there is additional configuration overhead for any entered-region, and the entire program cannot be legally run on the accelerator.

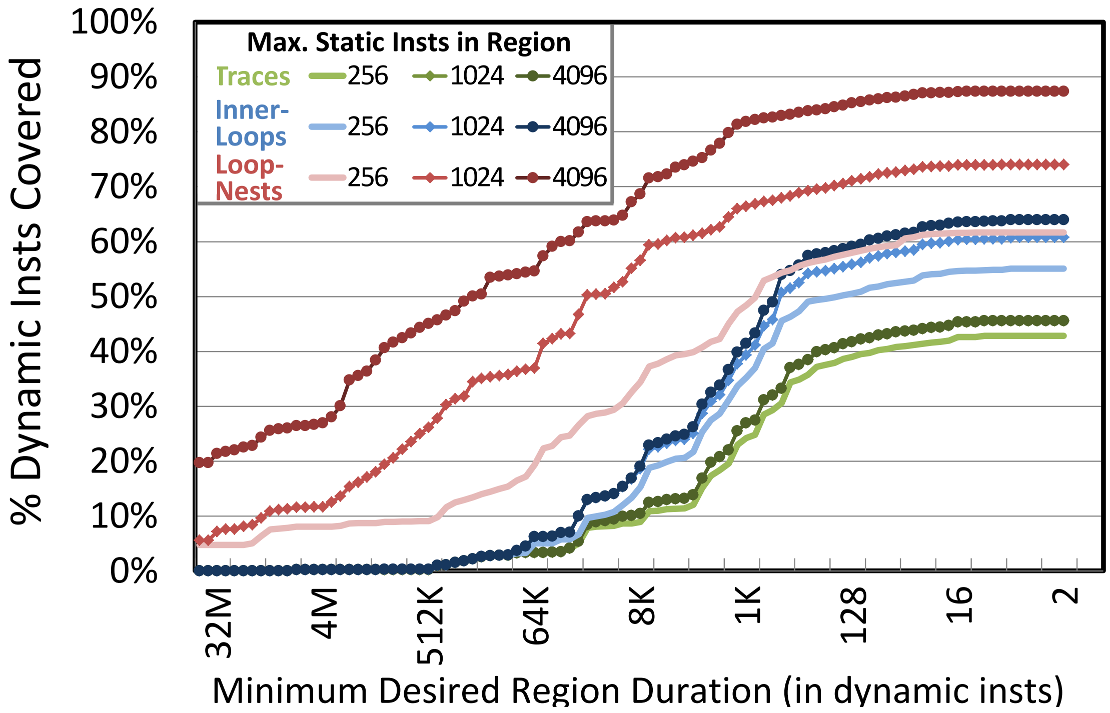


Figure 2.3: Cumulative % contribution for decreasing dynamic region lengths, shown for different static region sizes.

Example plausible regions include traces (series of repeating basic blocks), loops, and nested loops. We study their overheads by analyzing a set of irregular workloads (Described in detail in Chapter 4). Note that this study considers static binaries, and tradeoffs for dynamically-linked binaries will differ.

Figure 2.3 shows the cumulative distributions of dynamic instruction coverage by dynamic region granularity. For instance, considering regions with a duration of 8K dynamic instructions or longer (x-axis), nested loops can cover 70% of total instructions, while inner loops and traces can cover about 20% and 10% respectively. Considering *any* duration of region, the total instruction coverage is 88%, 64%, and 45% for nested loops, inner loops, and traces, respectively. Also, nested loops greatly increase the region duration (1K to 128K for 50% coverage).

For each region type, we also present different maximum region sizes: 256, 1024 and 4096 instructions. Targeting 1024 instruction regions presents a good tradeoff between total static instructions (more hardware cost) and dynamic region length.

The conclusion that we draw is that all region types can be useful for targeting a significant portion of programs, though if we want to target longer duration regions (more potential for power-gating) with fewer static instructions (more potential area benefits), then larger granularity regions like loops or nested loops are more effective, and should be preferred if compatible with the exploited program behavior.

2.5 Why In-Core?

Finally, the last major decision in the ExoCore model is the placement of the accelerator in relation to the general purpose core's cache hierarchy. Though we choose to attach the accelerators in-core, and thus behind the first-level cache, we could have attached them at a shared higher-level cache, or even close to memory. We present two arguments for why accelerators should indeed be in-core: 1. Regions can be short, meaning that switching is frequent, and overheads must be kept small, 2. Subsequent regions share a significant amount of data, which can be eliminated by sharing lower-level caches. We explain with data analysis as follows, using the same workloads as the previous section.

Dynamic Phase Length Figure 2.4 is a histogram of the percentage of total instructions spent in dynamic phases of different length, for inlined nested loops of with a static region size limit of 1000 instructions. Though around $1/3$ of dynamic phases are long (greater than 500K instructions in length), about as many of them are quite short (shorter than 5K instructions). If we are going to target these short regions, the overhead of offload must be much shorter than the region's execution time. Though putting the accelerator off-core could increase initialization time (for sending live values), the real overhead is in transferring re-used data already stored in caches, as we explain next.

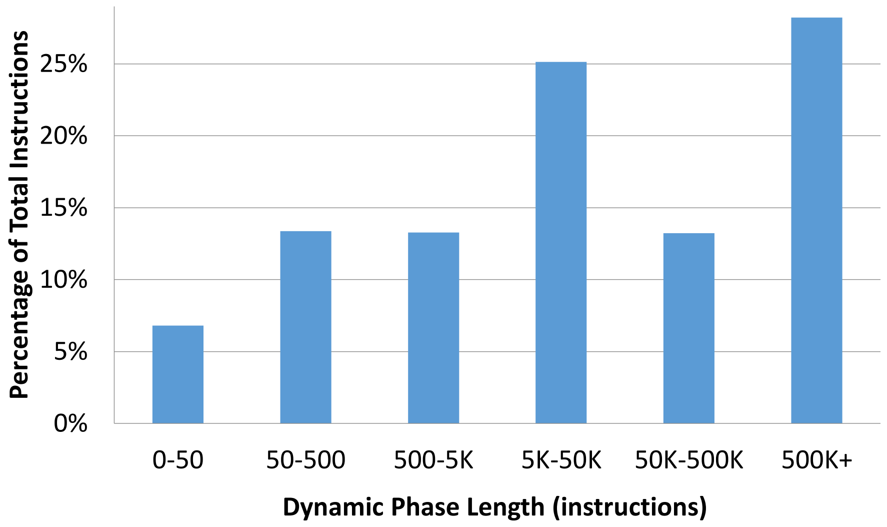


Figure 2.4: Dynamic Region Length (considering inlined nested loops < 1000 static instructions)

Dynamic Phase Data Sharing A potentially time-consuming component of accelerator switching is in transferring the data once the accelerator has begun execution. To measure this potential overhead, Figure 2.5 again bins regions by their dynamic length, but instead plots their average additional memory-system traffic per dynamic instruction, that would not have occurred had they shared a cache hierarchy. What becomes clear is that this is not a concern for longer phases, as data can be cached and re-used in accelerator caches. However, the same is not true for shorter phases, as there is not generally enough time to find reuse opportunities in short regions of the code. Regions that last less than 50K instructions on average must transfer about 1.5 bytes additional data per dynamic instruction executed. This translates into additional energy overheads, and perhaps even performance overheads. This overhead can be avoided by integrating accelerators in core, which is why we made this decision for this work.

2.6 Evaluated System and Modeling

To summarize, ExoCore is a core organization that includes multiple programmer-transparent accelerators, which are each activated in a phase-based manner with the core, based on the be-

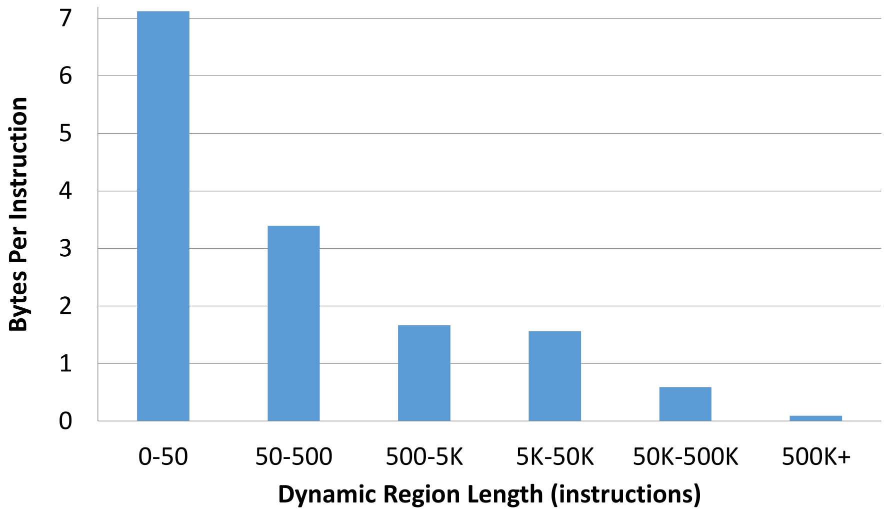


Figure 2.5: Data Transfer Overhead for Non-In-Core Accelerators (considering 2MB L2 Cache)

haviors of program regions. For evaluating these architectures, we need to be able to study the features of general purpose applications, compiler transformations targeted at accelerators, and their microarchitectural interactions.

For applications, we use unmodified single-threaded workloads, because the hardware interactions we care about are all in-core. Next, we use detailed modeling techniques for studying the compiler transformations and detailed microarchitecture. To be clear, the compiler *is* modeled in a low level way (modeling transformations on arbitrary code), but we have not implemented a compiler itself for these accelerators.

The compilation environment that we model is a static profile plus compilation approach, and we think the analysis and evaluation results will be similar for a dynamic binary translation environment, because of the relatively simple algorithms employed (one exception is the integer linear programming scheduler, which would not work well in a time-constrained dynamic environment).

We model microarchitecture at a core level, including the cache hierarchy, but we have not studied multi-core implications or built simulation infrastructure that supports a multi-threaded or multi-programmed environment. This is because the most relevant hardware interactions occur at the core level interacting with the general purpose core.

3 MODELING BEHAVIOR SPECIALIZATION

As argued earlier, behavior specialization has emerged as a promising paradigm for future microprocessors, which has been studied previously to some extent in one-off works. In such examples, it is natural to develop and evaluate such architectures within end-to-end vertical silos spanning application, language/compiler, hardware design and evaluation tools, leaving little opportunity for cross-architecture analysis and innovation. This chapter develops a novel program representation suitable for modeling these heterogeneous architectures with specialized hardware, called the transformable dependence graph (TDG), which combines semantic information about program properties and low-level hardware events in a single representation. We demonstrate, using four example architectures from the literature, that the TDG is a feasible, simple, and accurate modeling technique for transparent specialization architectures, enabling cross-domain comparison and design-space exploration.

In this chapter, we first discuss why existing modeling approaches are insufficient (Section 3.1). We then describe our approach for modeling behavior specialized accelerators – the Transformable Dependence Graph (TDG) – and go over a simple example (Section 3.2). To substantiate with more complex examples, we explain how to model four existing architectures from the literature (Section 3.3). We describe the implementation (Section 3.4) and perform validation (Section 3.5) and discuss limitations (Section 3.6). We finally cover related work (Section 3.7), and summarize (Section 3.8).

3.1 Limitations of Existing Evaluation Approaches

There are three known strategies for attaining insights into transparent accelerators. All are insufficient in different ways.

Analytical Modeling The general problem with analytical modeling is that it is too abstract from the low-level properties of the application upon which transparent accelerators rely. Often, because these models are detached from the details of the application, a model creator has to make a judgment call on whether to create a benchmark-specific model (i.e. analyze the benchmark first, then make the model), or make a benchmark-neutral model. Neither of these options will have the ability to capture the fine-grained interaction between the accelerators, applications, and baseline processors.

Simulation/Prototyping The second, arguably standard approach, is simulation (shown in Figure 3.1(a)). Due to the vertically-integrated nature of this approach, studying even one accelerator in this context requires extensive effort: accelerator ISA definition, host-ISA extensions, compiler analysis and transformations, code generation, and simulator implementation. This would be intractable for many accelerators due to development time. The tradeoffs for FPGA or full hardware prototypes are even more stark.

Studying Literature A final approach would be to rely on the combined insights of existing published works. However, having inconsistent GPPs or workloads across studies would lead to a poor understanding of the design space, and would not allow for the exploration of dividing work among multiple accelerators.

3.2 Transformable Dependence Graphs

Before discussing our transformable dependence graph modeling approach, we first touch on what are the requirements of an accelerator modeling technique. Subsequently, we describe an example model for a simple accelerator.

Requirements for modeling behavior specialization For modeling the workings of behavior specialized accelerators, a useful evaluation approach must capture the following aspects of execution:

1. **General Purpose Core Interaction** Since transparent accelerators selectively target certain code, the general purpose core must be modeled in detail (e.g. pipelining, structural hazards,

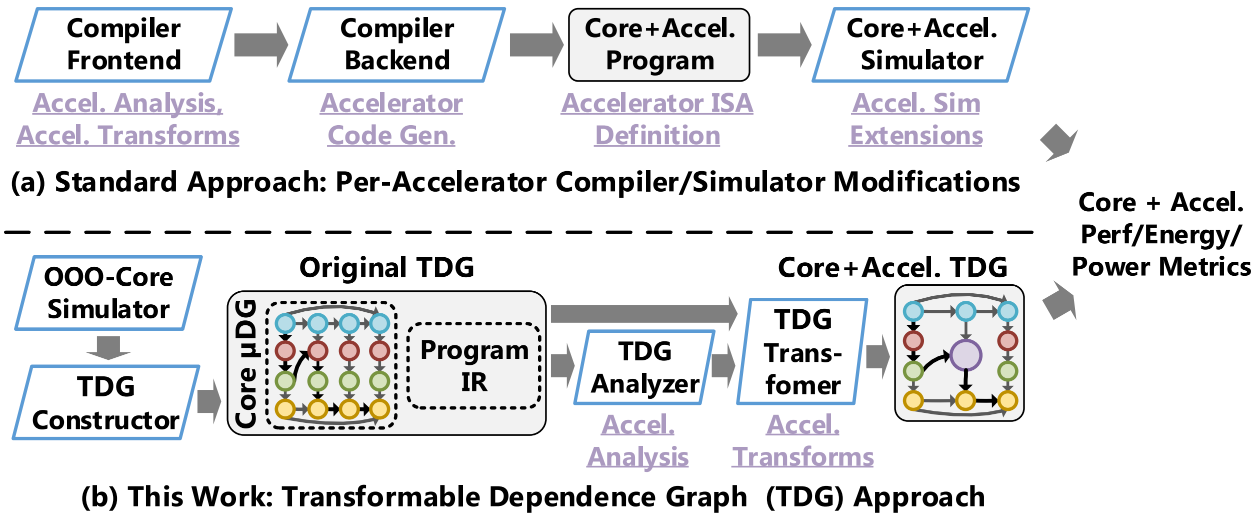


Figure 3.1: Approaches for modeling transparent specialization

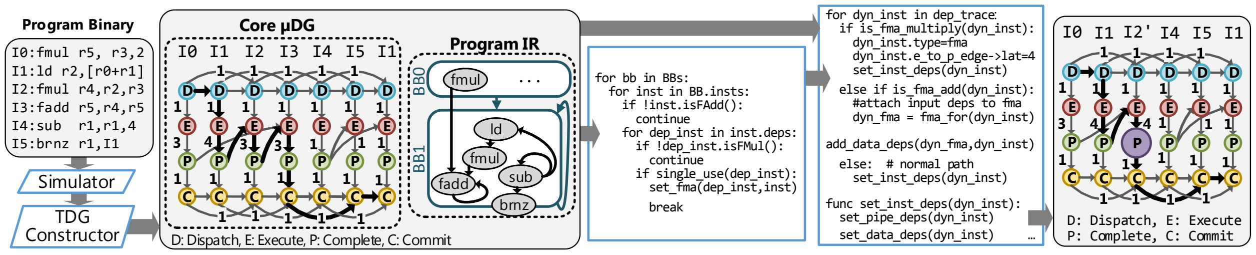
memory-system etc.). In addition, accelerators can interact with different parts of the general purpose core (e.g. sharing the LSQ), so this must be modeled as well.

2. **Application/Compiler Interaction** Much of a BSA's success depends on how well the compiler can extract information from the application, both for determining a valid accelerator configuration, and for deciding which accelerator is appropriate for a given region.

3. **Accelerator Behavior** Naturally, the low-level detailed workings of the accelerator must be taken into-account, including the microarchitecture and dynamic aspects of the execution like memory latency.

A traditional compiler and simulator approach can capture the above aspects very well, but the effort in building multi-BSA compilers and simulators is time consuming, if not intractable. What we require are higher-level abstractions.

Leveraging Microarchitectural Dependence Graphs (μDGs) For a higher-level model of microarchitectural execution, we turn to the μDG, a trace-based representation of a program. It is composed of nodes for microarchitectural events and edges for their dependences. It is constructed using dynamic information from a simulator, and as such, is input-dependent. The μDG has traditionally



```
I0:fmul r5, r3,2
I1:ld r2,[r0+r1]
I2:fmul r4,r2,r3
I3:fadd r5,r4,r5
I4:sub  r1,r1,4
I5:brnz r1,I1
```

```
for bb in BBs:
  for inst in BB.insts:
    if !inst.isFAdd():
      continue
    for dep_inst in inst.deps:
      if !dep_inst.isFMul():
        continue
      if single_use(dep_inst):
        set_fma(dep_inst,inst)
        break
```

```
for dyn_inst in dep_trace:
  if is_fma_multiply(dyn_inst):
    dyn_inst.type=fma
    dyn_inst.e_to_p_edge->lat=4
    set_inst_deps(dyn_inst)
  else if is_fma_add(dyn_inst):
    #attach input deps to fma
    dyn_fma = fma_for(dyn_inst)
add_data_deps(dyn_fma,dyn_inst)
  else:  # normal path
    set_inst_deps(dyn_inst)

func set_inst_deps(dyn_inst):
  set_pipe_deps(dyn_inst)
  set_data_deps(dyn_inst)     …
```

Figure 3.2: Example TDG-Based model for transparent fused multiply-add (`fma`) specialization.

been used for modeling out-of-order cores [37, 38]. Overall, it offers a detailed-enough abstraction for modeling microarchitecture, yet is still abstract and easy to model modifications/additions of effects.

What is missing from the above is the capability of capturing compiler/application interactions. Our insight is that *graph transformations* on the μDG can capture the effects of behavioral specialization – after all, a BSA simply relaxes certain microarchitectural dependences while adding others. To perform these transformations, we require knowing the correspondence between the program trace and the static program IR. This can be reconstructed from the binary.

The Transformable Dependence Graph

Approach Overview Putting the above together, the crux of our approach is to build the Transformable Dependence Graph (TDG), which is the combination of the μDG of the OOO core, and a Program IR (typically a standard DFG + CFG) which has a one-to-one mapping with μDG nodes.

As shown in Figure 3.1, a simulator produces dynamic instruction, dependence, and microarchitectural information, which is used by the constructor to build the TDG. The TDG is analyzed to find acceleratable regions and determine the strategy for acceleration. The TDG-transformer modifies the original TDG, according to a graph re-writing algorithm, to create the combined TDG for the general purpose core and accelerator. As part of the representation, the TDG carries information about overall execution time and energy. The next subsection describes the approach using an example.

Notation To aid exposition, the notation $\text{TDG}_{\text{GPP,ACCEL}}$ refers to a TDG representation of a particular general purpose processor (GPP) and accelerator. $\text{TDG}_{\text{OOO4,SIMD}}$, for example, represents a quad-issue OOO GPP with SIMD. As a special case, the original TDG prior to any transformations (not representing an accelerator) is $\text{TDG}_{\text{GPP},\emptyset}$.

Transformable Dependence Graph Example

Here we define the components of our approach using a running example in Figure 3.2, which is for transparently applying a simple fused multiply-accumulate (`fma`) instruction. We intentionally choose an extremely simple example for explanatory purposes, and note how a more complex accelerator would implement that component. The detailed modeling of such accelerators is in Section 3.3.

Constructing the TDG To construct $\text{TDG}_{\text{GPP},\emptyset}$, a conventional OOO GPP simulator (like gem5 [39]) executes an unmodified binary[1], and feeds dynamic information to the TDG constructor (Figure 3.2(a)). The first responsibility of the tool is to create the original μDG, which embeds dynamic microarchitectural information, including data and memory dependences, energy events, dynamic memory latencies, branch mispredicts and memory addresses. We note that this makes the TDG input dependent, which is similar to other trace-based modeling techniques.

To explain an example, Figure 3.2(b) shows the μDG for the original OOO core, which in this case was a dual issue OOO. Here, nodes represent pipeline stages, and edges represent dependencies to enforce architectural constraints. For example, edges between alternate dispatch and commit nodes model the width of the processor ($D_{i-2} \xrightarrow{1} D_i$, $C_{i-2} \xrightarrow{1} C_i$). The FU or memory latency is represented by edges from execute to complete ($E_i \rightarrow P_i$), and data dependencies by edges between complete to execute ($P_i \xrightarrow{0} E_j$).

The second responsibility of the constructor is to create a program IR (also in Figure 3.2(b)) where each node in the μDG has a direct mapping with its corresponding static instruction in the IR.

[1]Our implementation assumes that compiler auto-vectorization is off.

We analyze the stream of instructions from the simulator, using known techniques to reconstruct the CFG, DFG with phi-information, and loop nest structure using straightforward or known techniques [40]. Also, register spill and constant access is identified for later optimization. To account for not-taken control paths in the program, we augment the program IR with the CFG from binary analysis.

TDG Analyzer The next step is to analyze the TDG to determine which program regions can be the legally and profitably accelerated, as well as the "plan" for transformation. This "plan" represents the modifications a compiler would make to the original program. We explain with our example.

Figure 3.2(c) shows the algorithm (in pseudo-code) required for determining the `fma` instructions. To explain, the routine iterates over instructions inside a basic block, looking for a `fadd` instruction with a dependent `fmul`, where the `fmul` has a single use. The function `set_fma(inst1,inst2)` records which instructions are to be accelerated, and passes this "plan" to the TDG transformer. In concrete terms, a TDG-analysis routine is a C++ module that operates over the TDG's trace or IR, and the "plan" is any resulting information or data-structures that are stored alongside the TDG.

While the above is basic block analysis, more complex accelerators typically operate on the loop or function level. For example, determining vectorizability in SIMD would require analyzing the IR for inter-iteration data dependences.

TDG Transformer This component transforms the original TDG to model the behavior of the core and accelerator according to the plan produced in the previous step. It applies accelerator-specific graph transformations, which are algorithms for rearranging, removing, and reconstructing μDG nodes and dependence edges. In our notation, this is the transformation from $\text{TDG}_{\text{GPP-X},\emptyset}$ to $\text{TDG}_{\text{GPP-Y,ACCEL}}$.

Figure 3.2(d) outlines the algorithm for applying the `fma` instruction, which iterates over each dynamic instruction in the μDG. If it is an accelerated `fmul`, it changes its type to `fma` and updates its latency. If the original instruction is an accelerated `fadd`, it is elided, and the incoming data dependences are added to the associated `fma`.

This simple example operates at an instruction level, but of course more complex accelerators require modifications at a larger granularity. For instance, when vectorizing a loop, the μDG for multiple iterations of the loop can be collected and used to produce the vectorized μDG for a single new iteration.

Core+Accelerator TDG The core+accelerator TDG represents their combined execution, an example of which is in Figure 3.2(e), for $\text{TDG}_{\text{OOO2},\texttt{fma}}$. Here, `I2'` represents the specialized instruction, and `I3` has been eliminated from the graph. In practice, more complex accelerators require more substantial graph modifications. One common paradigm is to fully switch between a core and accelerator model of execution at loop entry points or function calls.

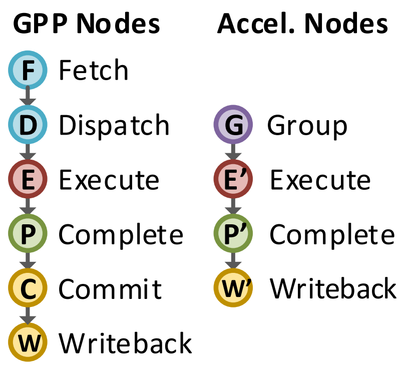
Finally, this TDG can be analyzed for performance and power/energy. The length of the critical path, shown in bold in the figure, determines the execution time in cycles. For energy, we associate events with nodes and edges, which can be accumulated and fed to standard energy-modeling tools.

3.3 TDG Transform Implementations

This section describes the TDG analysis and transformation algorithms for the GPPs and accelerators, and we begin by describing some preliminaries to ease explanations.

Subsequently, in describing transformations, we assume that $\text{TDG}_{\text{GPP-Orig},\emptyset}$ has already been constructed, and we are transforming to $\text{TDG}_{\text{GPP-New},\text{ACCEL}}$. Though we describe GPP and accelerator transformations separately, these would be applied *in-concert* to create a new TDG. Figure 3.3 gives example TDGs for all architectures on a simple program.

Also, in our presentation, we avoid formal algorithmic specification, and instead describe models



Edge	Constraint	Latency	New
$F \rightarrow D$	Frontend Pipeline	Fixed	✓
$D \rightarrow E$	Dispatch before Exec.	0	
$G \rightarrow E$	Group before Execute	0	✓
$E \rightarrow P$	Execution Latency	Fixed	
$E \rightarrow P$	Load Latency	Recorded	
$P \rightarrow C$	Commit Delay	Fixed	
$C \rightarrow W$	Store Latency	Recorded	✓

Table 3.1: Prism's Intra-Instruction Nodes and Edges

in informal terms, both because of space limits and to aid in exposition[2]. One of the contributions of this work is that these *are* straightforward transformations. Future work will explore formal graph-theory representations of transformations.

Preliminaries

In a TDG, the behavior of each dynamic instruction or operation is represented by a set of nodes for each of its stages of execution. Edges represent dependences between these stages, both inside and across instructions. Table 3.1 shows the nodes and inter-instruction edges and their latencies for all GPP and accelerator models discussed in this paper, and Table 3.2 describes inter-instruction edges.

Essential Components To give some intuition of the purpose of various nodes and edges, we describe common components here, specifically the Execute (E), Complete (P) and Writeback (W) nodes. The $E{\rightarrow}P$ edge captures FU and load latency, while the edge into W ($C{\rightarrow}W/P{\rightarrow}W$) captures store latency. The $P{\rightarrow}E$ edge captures data and memory dependences between instructions. The GPP model also includes Fetch (F) Dispatch (D) and Writeback (W) nodes for representing pipeline behavior. Other architecture-specific nodes and edges will be described as-needed.

[2]We also anticipate publicly releasing our entire framework.

Representing Resources with Dynamic Edges It is simple to represent microarchitectural resources which are acquired and released in-order, like the one which represents the ROB size ($C_{i-robsize} \rightarrow D_i$).

However, many resources in an architecture are *not* acquired and released in a pre-specified order (e.g. functional units).

Our approach for representing these resources is to keep a cycle-indexed data-structure that tracks which resources are occupied by which originating instruction and when they will be freed. If all of a certain resource are taken, a dependence is added between the first node which frees the given resource to the node which is requesting the resource. We refer to these edges as ***dynamic edges***, because their source and destination instructions change depending on the timing of other events. These edges are indicated in the "Dyn" column of Table 3.2.

GPP TDG Transformations

Here we describe GPP transformations, which construct a particular GPP's TDG by transforming another. We first describe how we elide (rip-up) particular edges, then how we insert edges for representing the GPP's execution. We first describe common edges across GPP processors, then edges which are specific to OOO and inorder GPPs.

Eliding Edges The first step of GPP transformation is to elide or rip-up edges which need to be modified. These include edges representing architectural features which may be changed, or they could be *dynamic edges* which do not have defined start and end nodes. There are certainly edges that not elided, including data and memory dependences, and edges enforcing pipeline serialization.

Common GPP Transforms The GPP representation is quite similar to [41], so we focus on newly proposed components, as indicated in the last column of Tables 3.1&3.2. Broadly speaking, their model contains edges for representing the issue-width, pipeline dependences, and execution and

	Edge	Constraint	Lat	Dyn	New
Common	$P_i \rightarrow E_j$	Data Dep.; Inst$_j$ depends on Inst$_i$	0		
	$P_i \rightarrow E_j$	Mem Dep.; Inst$_j$ mem depends on Inst$_i$	0		
	$E_i \rightarrow E_j$	B/W to L1; Mem$_i$ releases Mem-port to Mem$_j$	1	✓	✓
	$P_i/W_i \rightarrow$	B/W to L2; Load$_j$ proceeds after Load$_i$ releases	1	✓	✓
	E_j/P_j	L1 MSHR. Store source/dest nodes after slash.			✓
	$P_i \rightarrow P_j$	Load Inst$_i$ pulls cache line for Load Inst$_j$	0		
Common GPP	$F_{i-1} \rightarrow F_i$	Icache Latency; Inst$_i$ experienced cache miss	Rc		
	$F_{i-1} \rightarrow F_i$	Fetch In-order	0		✓
	$F_{i-fw} \rightarrow F_i$	$fw =$ Fetch Width	1		✓
	$D_{i-fp} \rightarrow F_i$	$fp =$ Frontend Pipeline Buffer Size	0		✓
	$C_{i-1} \rightarrow F_i$	Inst$_i$ fetch delayed b/c Ctrl$_i - 1$ mispredicted,	Cm		
	$C_i \rightarrow F_{i+1}$	Inst$_i$ is serializing	0		✓
	$W_i \rightarrow E_j$	Inst$_j$ is non-speculative, and Inst$_i$ is prev. store	0		✓
	$E_i \rightarrow E_j$	Execution Resource Conflict	1	✓	✓
OOO GPP	$D_{i-1} \rightarrow D_i$	Dispatch In-order	0		
	$C_{i-1} \rightarrow C_i$	Commit In-order i	0		
	$D_{i-dw} \rightarrow D_i$	$dw =$ Dispatch Width	1		
	$C_{i-cw} \rightarrow C_i$	$cw =$ Commit Width	1		
	$C_{i-rb} \rightarrow D_i$	$rb =$ Reorder Buffer Size	0		
	$P_i \rightarrow D_j$	LQ Size; Inst$_j$ is lq_size loads after Inst$_i$	0		✓
	$P_i \rightarrow D_j$	SQ Size; Inst$_j$ is sq_size stores after Inst$_i$	0		✓
	$P_i \rightarrow F_j$	Pipe flush; Load$_j$ blocks in the cache b/c of Inst$_i$	1		✓
	$E_i \rightarrow D_j$	IW Size, Inst$_i$ blocks Inst$_j$ from the IW	1	✓	✓
IO	$P_{i-1} \rightarrow P_i$	Complete In-order	0		✓
	$P_{i-1} \rightarrow E_i$	Long latency Inst$_{i-1}$ stalls Inst$_i$ by pipe length	Fx		✓
Co-proc	$G_i \rightarrow E_i$	Group for Accel Op$_i$ begins before execution	0		✓
	$P_i \rightarrow G_j$	Complete for Accel Op$_i$ before next Group	0		✓
	$P_i \rightarrow G_j$	Accel XFER: Complete GPP Inst$_i$ before Group$_j$	Cm		✓
	$P_i \rightarrow F_j$	GPP XFER: Accel Op$_i$ before GPP Fetch of Inst$_j$	Cm		✓
DySER	$E_i \rightarrow E_j$	Enforce issue late for Insts on same FU	Fx		✓
	$P_i \rightarrow P_j$	Instructions on same FU Complete in-order	1		✓
	$E_i \rightarrow P_i$	Data Dep.; Fixed latency models network delay.	Fx		✓

Table 3.2: Inter-Inst. Edges (Fx:Fixed, Cm:Computed, Rc:Recorded)

memory latency. Because we add support for dynamic resources, we can additionally represent functional unit and memory bandwidth resources using dynamic edges. The L1 bandwidth is represented by treating load/store ports as resources, and L2 bandwidth is approximated by considering a finite number of coalescing MSHRs as resources. Note that the memory interface transforms, including the insertion of memory bandwidth edges, are present in all models.

Out-of-Order (OOO) GPP ($TDG_{GPP\text{-}Orig,\emptyset}$ to $TDG_{OOO,\emptyset}$) Straightforward edges were added to model load/store queue size. Using a table of the last *queue size* loads or stores, we constrain the dispatch of the current memory operation to go after the complete of the oldest memory operation. We also model pipeline flushing for loads which block in the L1 cache, to model GPP pipeline replay. Finally, the instruction window is modeled with dynamic edges.

In-order (IO) GPP ($TDG_{GPP\text{-}Orig,\emptyset}$ to $TDG_{OOO,\emptyset}$) Only two extra edges are required here: the first enforces in-order complete, and the second delays the execution of instructions following long-latency operations. Misprediction delay is also reduced.

Accelerator Transformations

For each accelerator, we first give a brief background, then describe its TDG analysis and transformation algorithms.

Conservation Cores (C-Cores) TDG

C-Cores are automatically-synthesized circuits for application code. These are meant as simple, energy-efficient *coprocessors*, which serialize control and memory access. C-Cores target either fully-inlined loops or function calls.

TDG Analysis The analysis "plan" is a set of program regions to target using the TDG's profile data. We approximate the area using the total number of static operations as a proxy. The goal is to maximize the total dynamic instructions captured for some limit of static instructions.

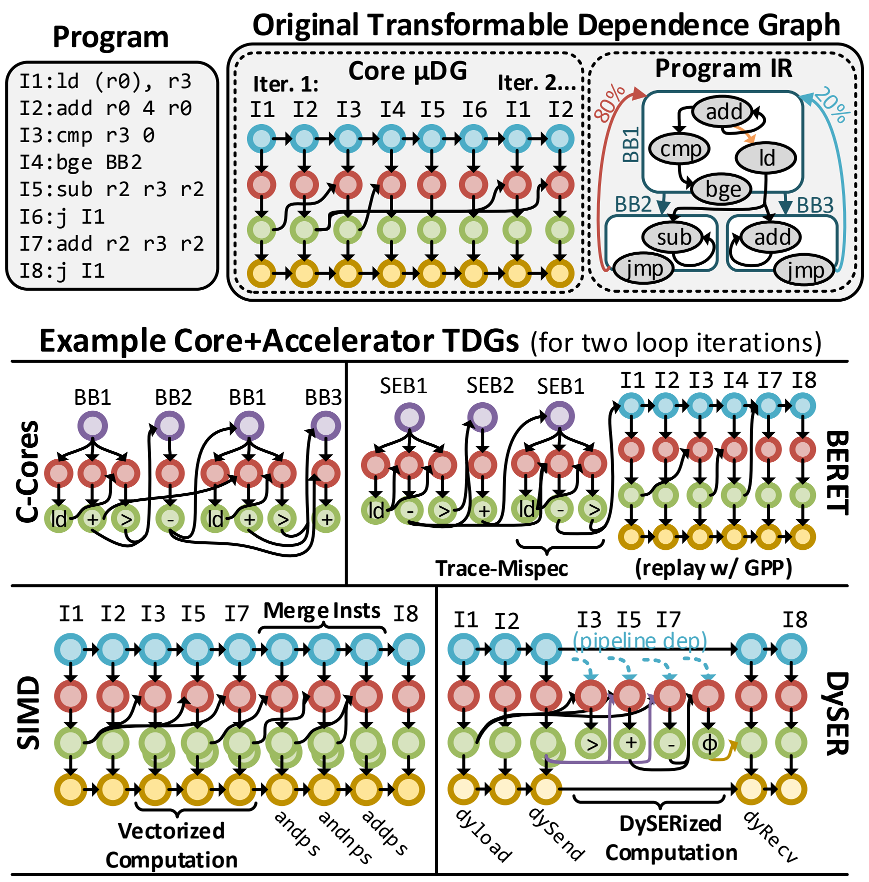


Figure 3.3: Example TDGs (Nodes/edges in Tables 3.1&3.2.)

The selection algorithm first builds a set of trees representing the hierarchy of inlineable loops and function calls. The region selection heuristic performs a bottom-up tree traversal, selecting code which has the highest dynamic to static instruction ratio.

TDG Transform (TDG$_{\text{GPP},\emptyset}$ to TDG$_{\text{GPP,SIMD}}$) Inside a C-Cores region of the μDG, according to the analysis plan, this transform elides all fetch, dispatch and commit nodes and edges. Then, since C-Cores only handle one control condition and memory request at a time, a Group (G) node is created for each "basic block," where they are split to ensure each only contains one memory operation. Edges are added from the Group node to/from the C-Cores instructions, which serialize basic blocks' execution (see Figure 3.3 and coproc rows in Table 3.2).

When the entering/exiting a C-Cores region, edges are inserted to model transfer time. This latency is the amount of live data across regions (computed by analyzing the DFG), divided by the bandwidth between core and accelerator.

BERET TDG

This *coprocessor* accelerates hot traces of inner loops. It targets energy efficiency by using compound functional units, called Serialized Execution Blocks (SEBs). Diverging from the hot loop trace entails re-execution on the GPP.

TDG Analysis The analysis "plan" is a set of eligible and profitable inner loops, and SEB instruction schedules for those loops. Eligible loops with hot traces are found using path profiling techniques [22]. Loops are selected if their loop back probability is higher than 80%, and their configuration size fits in the hardware limit. To eliminate over-specialization to the original target benchmarks, we consider fixed size SEBs, as opposed to specific compound functional units. For scheduling into SEB groups, we use an optimal integer linear program which minimizes the number of register file accesses.

TDG Transform ($\text{TDG}_{\text{GPP-Orig},\emptyset}$ to $\text{TDG}_{\text{OOO},\text{BERET}}$) This transform resembles that of C-Cores, where instead of basic blocks, instructions are grouped into SEBs, which are serialized to execute one at a time.

The GPP interaction is also similar to C-Cores, and uses the same edge insertion algorithm. In addition, if BERET mispeculates the trace path, instructions from that loop iteration are "replayed" on the host processor by reverting to the $\text{TDG}_{\text{GPP-Orig},\emptyset}$ to $\text{TDG}_{\text{GPP-New},\emptyset}$ transform (see Figure 3.3).

SIMD (Loop Auto-vectorization) TDG

For SIMD we focus on vectorizing independent loop iterations, as this is the most common form of auto-vectorization.

TDG Analysis The analysis "plan" is as set of legal and profitable loops for vectorization. First, a pass optimistically analyzes the TDG's memory and data dependences. Memory-dependences between loop iterations can be detected by tracking per-iteration memory addresses in consecutive iterations. Loops with non-vectorizable memory dependences are excluded, and considering loop-splitting and loop-reordering to break these dependences is future work. Similarly, loops with inter-iteration data dependences which are not reductions or inductions are excluded.

For vectorizing control flow, the TDG analysis considers an if-conversion transformation, where basic blocks in an inner-loop are arranged in reverse-post order, and conditional branches become predicate-setting instructions. This analysis also computes where masking instructions would need to be added along merging control paths. The TDG decides whether to vectorize a loop by computing the expected number of dynamic instructions per iteration by considering path profiling information. If it is more than twice the original, the loop is disregarded.

TDG Transform ($\mathbf{TDG_{GPP,\emptyset}}$ to $\mathbf{TDG_{GPP,SIMD}}$) When a vectorizable loop is encountered, μDG nodes from the loop are buffered until the vector-length number of iterations are accumulated. The first iteration of this group becomes the *vectorized* version, and not-taken control path instructions, as well as mask and predicate instructions, are inserted. Most instructions are converted to their vectorized version, except for non-contiguous loads/stores, for which additional scalar operations are added (as we target non-scatter/gather hardware). At this point, memory latency information is re-mapped onto the *vectorized* iteration, and the non-vector iterations are elided. If fewer than the minimum vector length iterations remain, the SIMD transform is not used.

DySER TDG

DySER is a reconfigurable circuit switched mesh of FUs, meant for exploiting instruction and data parallelism. It is tightly integrated to the GPP, using custom instructions for communication and a flexible vector interface.

TDG Analysis The analysis "plan" is a set of legal and profitable loops, potentially vectorized, where for each loop the plan contains the *computation subgraph* (offloaded instructions). Vectorization analysis is borrowed from SIMD. Since it only executes computation subgraphs, we use a known slicing algorithm [32] on the loop's PDG to partition the instructions between the GPP and accelerator. Control instructions which do not have forward memory dependences can be offloaded to DySER.

Similar to SIMD, a "DySER-ized" version of inner loops is considered, where the computation subgraph is removed from the loop, and communication instructions are inserted along the interface edges. If the loop is vectorizable, the computation can be "cloned" until its size fills the available DySER resources, or until the maximum vector length is reached, enabling more parallelism. The analysis algorithm disregards loops with more communication instructions than offloaded computation.

TDG Transform ($\text{TDG}_{\text{GPP},\emptyset}$ to $\text{TDG}_{\text{GPP},\text{DySER}}$) DySER keeps a small configuration cache, so if a configuration is not found when entering a DySERized loop, instructions for configuration are inserted into the TDG. Similar to SIMD, μDG nodes from several loop iterations are buffered until the vectorizable loop length is reached. At this point, if the loop is vectorizable, the first step is to apply the SIMD transformation as described earlier ($\text{TDG}_{\text{GPP},\emptyset}$ to $\text{TDG}_{\text{GPP},\text{SIMD}}$).

Then, DySER-ized instructions are first processed by removing their fetch, dispatch and commit nodes. Then two additional edges to enforce accelerator pipelining: one for the issue width between computation instances ($E{\rightarrow}E$), and one for in-order completion ($P{\rightarrow}P$). We model the scheduling and routing latency by adding a fixed cycle delay on the data dependence edges.

3.4 Implementation: Prism

Our framework's implementation, *Prism*, generates the original TDG using gem5 [39], enabling analysis of arbitrary programs. We implement custom libraries for TDG generation, analysis and transformation. Since transforming multi-million instruction traces can be inefficient, Prism uses a

	Common	GPP	C-Cores	BERET	SIMD	DySER
Analysis	6012	-	169	538	264	507
Transform	2206	2126	448	783	949	1215

Table 3.3: Source Lines of TDG Modeling Code

windowed approach. Windows are large enough to capture specialization granularity (max ∼10000 instructions). The final outputs include cycle count and average power.

Power and Area Estimation Prism accumulates energy event counts for both the GPP and accelerator from the TDG. It then uses McPAT [42] internally for computing power, calling McPAT routines at intervals over the program's execution. We use 22nm technology. The GPP core activity counts are fed to McPAT [42], a state-of-the-art GPP power model. For accelerators, a combination of McPAT (for FUs) and CACTI [43] is used, and for accelerator-specific hardware we use energy estimates from existing publications.

Using TDG Models in Practice The TDG can be used to study new BSAs, their compiler interactions and the effect of varying input sets. In practice, running TDG models first requires TDG-generation through a conventional simulator. The generated TDG can be used to explore various core and accelerator configurations. Since the TDG is input-dependent, studying different inputs requires the re-running the original simulation.

Implementing a TDG model is a process of writing IR analysis routines, graph-transformation algorithms and heuristics for scheduling, as outlined in Appendix A.

In general, this process is simple. To substantiate, Table 3.3 shows the lines of C++ code required to implement the analysis and the graph transformation for each accelerator (common code need not be modified to add an accelerator). Lines of code is roughly a measure of the "complexity" of a technique. But more importantly, the graph transformations are explicit about how the compiler and accelerator are specializing the processor, and hence are more insightful and easy for designers to use.

Accel.	Base	P Err.	P Range	E Err.	E Range
OOO8→1	–	3%	0.05→1.0 IPC	4%	0.75→2.75 IPE
OOO1→8	–	2%	0.02→5.5 IPC	3%	0.39→1.7 IPE
C-Cores	IO2	5%	0.84→1.2×	10%	0.5→0.9×
BERET	IO2	8%	0.82→1.17×	7%	0.46→0.99×
SIMD	OOO4	12%	1.0→3.6×	7%	0.30→1.3×
DySER	OOO4	15%	0.8→5.8×	15%	0.25→1.28×

Table 3.4: Validation Results (P: Perf, E: Energy)

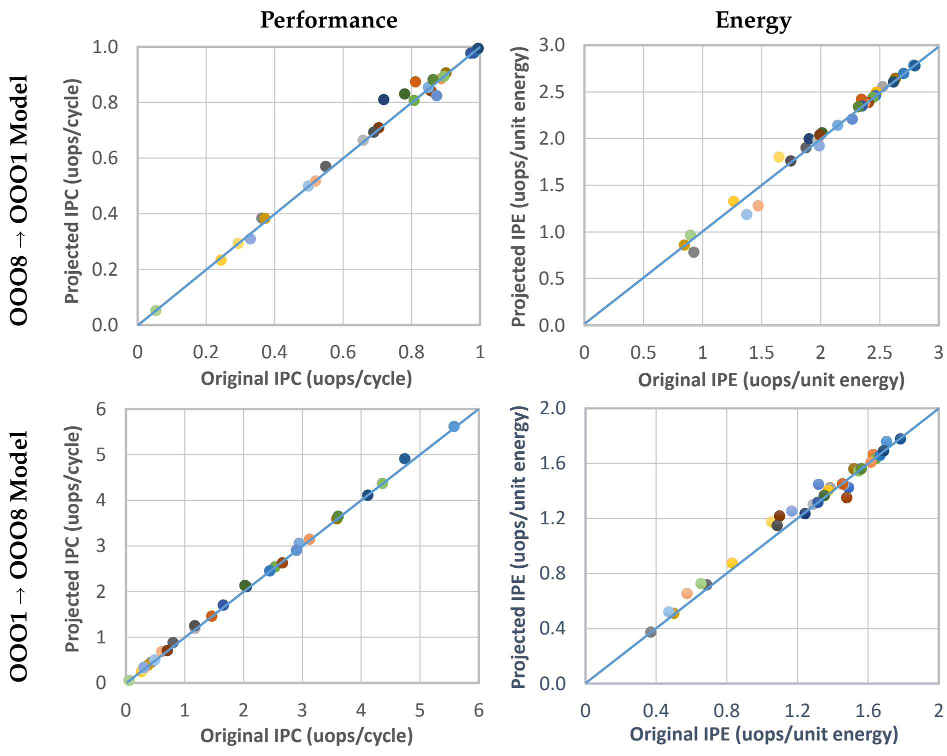


Figure 3.4: Prism Core Validation

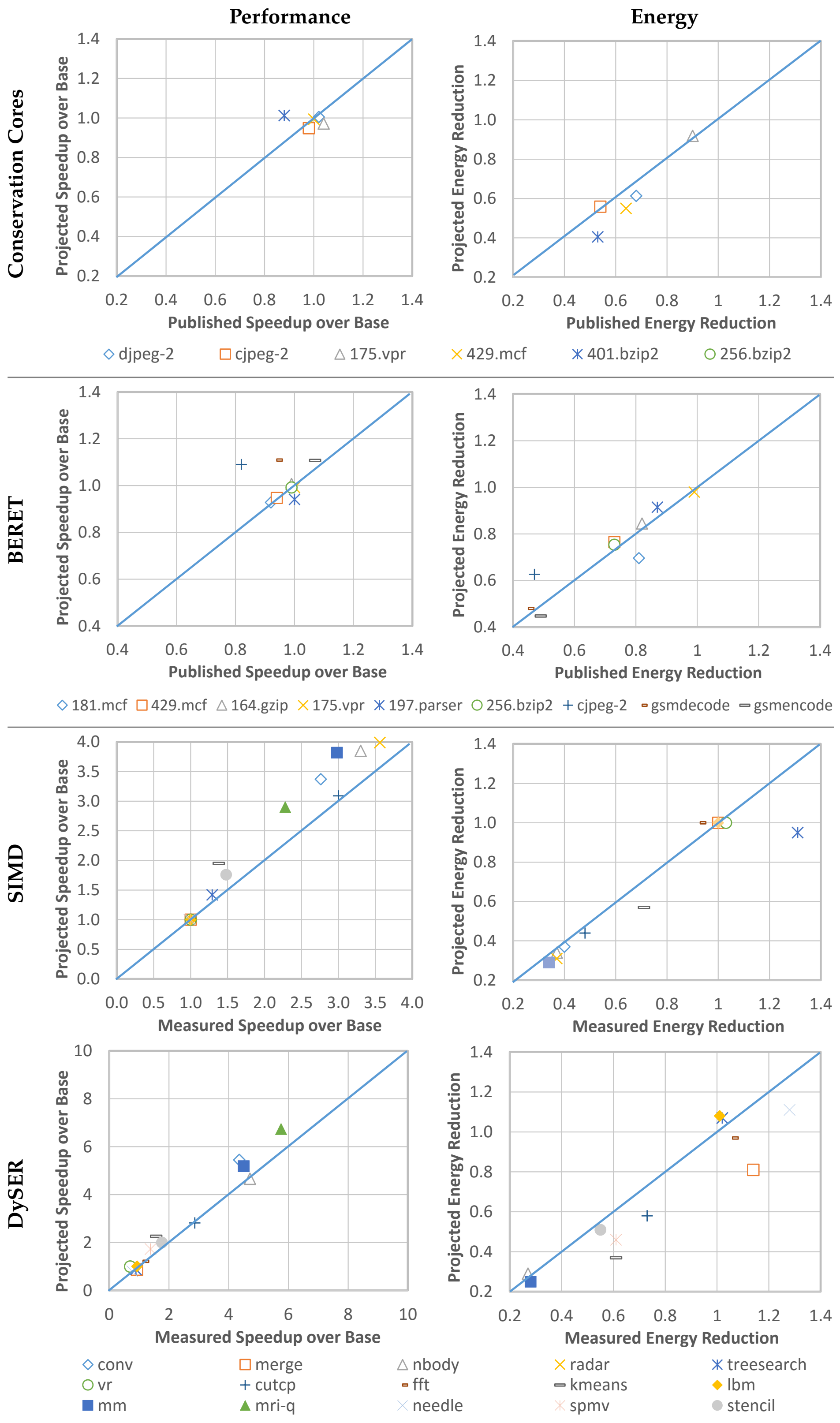


Figure 3.5: Prism Accelerator Validation

3.5 Core and Accelerator Validation

We perform validation by comparing the TDG against the published results of four accelerators, each of which use traditional compiler plus simulator evaluation. These results used benchmarks from the original publications [27, 25, 32], and Parboil [44] and Intel's microbenchmarks for SIMD. The original μDG is generated by fast-forwarding past initialization, then recording for 200 million instructions. Here, the error is calculated by comparing the relative speedup and energy benefits over a common baseline, where we configured our baseline GPP to be similar to the reported baseline GPP. Note that for SIMD and DySER, we were able to adjust the transformations to match the compiler output, as we had access to these compiler infrastructures.

Table 3.4 presents a validation summary for the OOO core and accelerators, where column "Base" is the baseline GPP, "Err." is the average error, and "Range" is the range of compared metrics. Figures 3.4 and 3.5 shows the validation graphs for each architecture, including performance (left) and energy (right). Each point represents the execution of one workload, and the distance to the unit line represents error. The X-Axis is the validation target's estimate (from published results or measured from simulation), and the Y-Axis is the TDG's estimate of that metric. We describe the validation of the core and accelerators below.

OOO-Core To determine if we are over-fitting the OOO core to our simulator, we perform a "cross validation" test: we generate a trace based on the 1-Wide OOO core, and use it to predict the performance/energy of an 8-Wide OOO core, and vice-versa. Figure 3.4 shows the results. The benchmarks in this study [45] are an extension of those used to validate the Alpha 21264 [46] (we omit names because there are too many). The high accuracy here ($< 4\%$ on average) demonstrates the flexibility of the model in either speeding up or slowing down the execution.

Conservation Cores are automatically generated, simple hardware implementations of application code [27], which are meant as offload engines for in-order cores. We validate the five benchmarks in the above paper, and achieve an average of 5% and 10% average error in terms of performance improvement and energy reduction. The worst case is for 401.bzip2, where we under-predict performance by 15%.

BERET is also an offload engine for hot loop traces, where only the most frequently executed control path is run on the accelerator, and diverging iterations are re-executed on the main processor [25]. Efficiency is attained through serialized execution of compound functional units. We achieve an average error of 8% and 7% in terms of performance improvement and energy reduction. We over-predict performance slightly on cjpeg and gsmdecode, likely because we approximate using size-based compound functional units, rather than more restrictive pattern based (to not over-conform to the workload set).

SIMD validation is performed using the Gem5 Simulator's implementation, configured as a 4-Wide OOO processor. Figure 3.5(e) shows how we attain an average error of 12% and 7% in terms of performance improvement and energy reduction. Our predictions for SIMD are intentionally optimistic, as there is evidence that compilers will continue to see improved SIMD performance as their analysis gets more sophisticated. We remark that our transformations only consider a straight-forward auto-vectorization, and will not be accurate if the compiler performs data-layout transformations or advanced transformations like loop-interchange.

DySER is a coarse grain reconfigurable accelerator (CGRA), operating in an access-execute fashion with the GPP through a vector interface [32]. On the Parboil and Intel microbenchmarks, we attain an average error of 15% for both speedup and energy reduction.

In summary, the TDG approach achieves an average error of less than 15% for estimating speedup and energy reduction, compared to simulator or published data.

3.6 Limitations of TDG Modeling

Lack of Compiler Information First, since the TDG starts from a binary-representation, it lacks native compiler information, which sometimes must be approximated in some way. An example is memory aliasing between loop instructions, useful for determining vectorization legality. In such cases, we use dynamic information from the trace to estimate these features, though of course this is optimistic.

Other Sources of Error Another consequence of beginning from a binary representation are ISA artifacts in the TDG. One important example is register spilling. In this case, the TDG includes a best-effort approach to identify loads and stores associated with register spills, which can potentially be bypassed in accelerator transformations.

The graph representation is itself constraining, in particular for modeling resource contention. To get around this, we keep a windowed cycle-indexed data structure to record which TDG node "holds" which resource. The consequence is that resources are preferentially given in instruction order, which may not always reflect the microarchitecture.

Another source of error is unimplemented or abstracted architectural/compiler features. An example from this work is the lack of a DySER spatial scheduler – the latency between FUs is estimated. Of course, TDG models can be made more detailed with more effort.

Flexibility The μDG itself embeds some information about the microarchitectural execution (eg. memory latency, branch prediction results), meaning that it is not possible to change parameters that affect this information without also re-running the original simulation. Understanding how those components interact with specialization would require recording multiple TDGs.

Transformation Expressiveness Some transformations are difficult to express in the TDG, limiting what types of architectures/compilation techniques can be modeled. In particular, non-local transforms are challenging, because of the fixed instruction-window that the TDG considers. One example of this would be an arbitrary loop-interchange.

3.7 Related Work

An alternative to modeling architectures with the TDG are analytical models. Relevant works include those which reason about the benefits of specialization and heterogeneity [47, 48, 49, 50], high-level technology trend projections [51, 52], or even general purpose processors [53, 54, 55]. There are also several analytical GPU models [56, 57, 58, 59, 60, 61]. The drawback of such models is that they are either specific to the modeling of *one* accelerator or general-purpose core, or they are too generic and do not allow design space explorations which capture detailed phenomenon.

Kismet [62, 63] is a model which uses profiles of serial programs to predict upper-bound speedups for parallelization. It uses a hierarchical critical path analysis to characterize the available and expressible parallelism.

Perhaps the most related technique is the Aladdin framework [64], a trace-based tool that uses a compiler IR interpreter, that enables design space exploration of *domain specific accelerators*. Using such an approach for behavioral specialized accelerators is possible, and should reduce errors from ISA artifacts. The drawback would be that the interaction with the general purpose core in that framework would be more difficult to capture. However, that style of approach may be an avenue for improving accuracy in the future.

3.8 Summary

Our work presents a novel program representation for transparent specialization called the TDG, which consists of a closely coupled μDG and Program IR for analysis. This representation allows for the study of the combined effects of compiler and hardware microarchitecture as graph transformations. We showed this representation is accurate, intuitive, and simple.

4 DATAFLOW SPECIALIZATION

Great strides have been made in the specialization of *regular* codes, through the development of SIMD, GPUs, and other designs [65, 32, 66, 67, 33]. These techniques can dramatically cut per-instruction overheads like the dynamic extraction of the data-dependence graph or in the maintenance of instruction-precise state. However, for irregular code, current approaches for specialization either heavily curtail performance or provide simply too little benefit. Interestingly, well known *explicit-dataflow* architectures eliminate these overheads by directly executing the data-dependence graph and eschewing instruction-precise recoverability. However, even after decades of research, dataflow architectures have yet to come into prominence as a solution. We attribute this to a lack of effective control speculation and the latency overhead of explicit communication, which is crippling for certain codes.

In this chapter, we first discuss the potential of dataflow specialization over existing techniques (Section 4.1). Then we explain the primitives required for fine-grain explicit-dataflow specialization (Section 4.2), and describe the architecture of a behavior specialized accelerator (SEED) which meets those primitives (Section 4.3). We discuss compiler responsibilities as well (Section 4.4). To understand the potential benefits, we present methodology (Section 4.5) and perform detailed evaluation and design space exploration (Section 4.6). We then discuss related work (Section 4.7), and summarize (Section 4.8).

4.1 Potential of Dataflow Specialization

In this section, we discuss existing and potential approaches for targeting codes that are irregular, either in terms of control or memory. Control irregularity includes divergent or unpredictable branches, and memory irregularity includes non-contiguous or indirect access[1].

[1]This section examines irregular workloads by restricting to SPECint/Mediabench. Observations here apply to irregular workloads, unless specified.

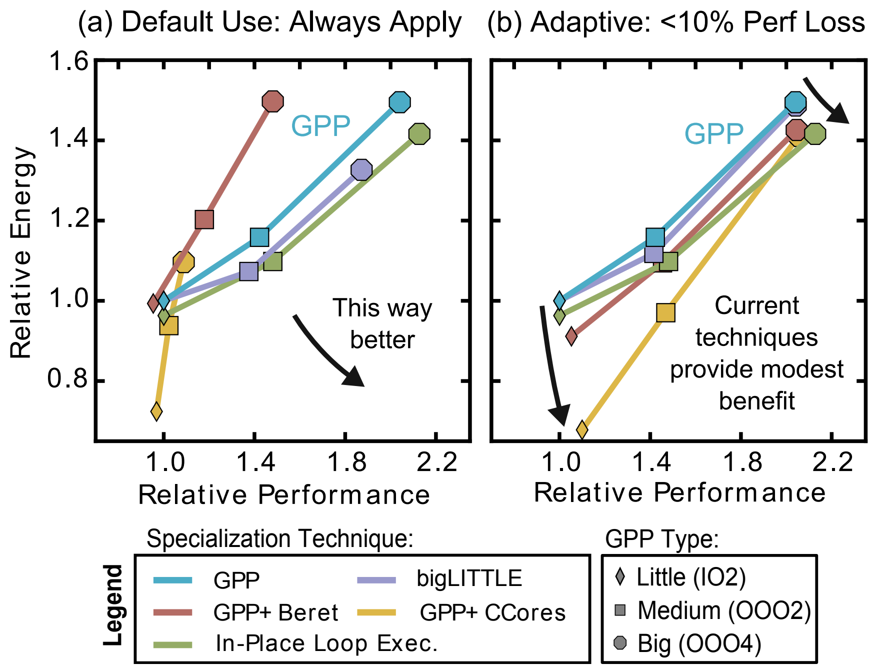
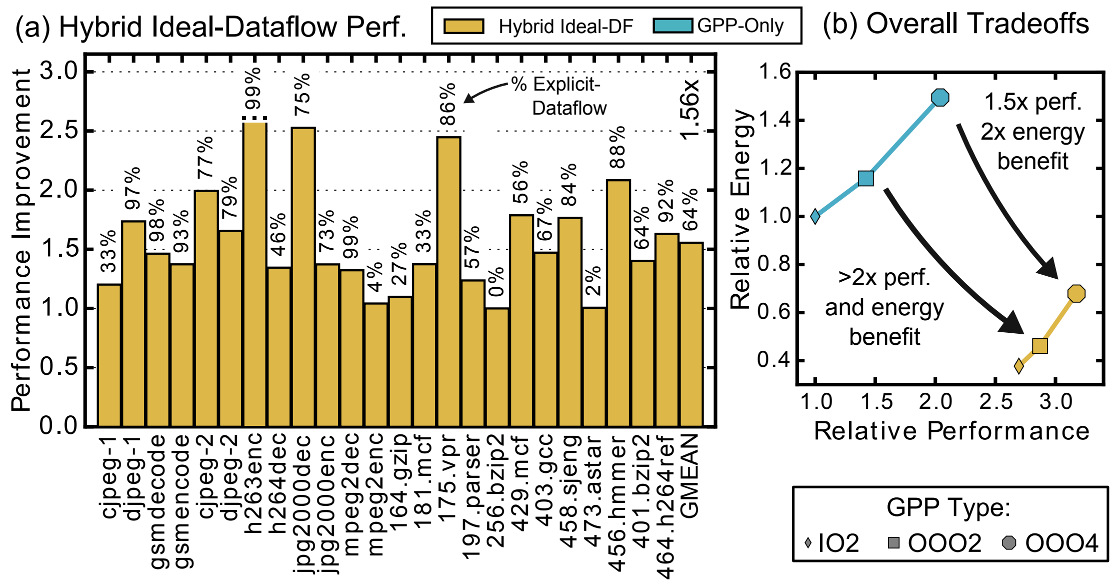


Figure 4.1: Energy/Perf. Tradeoffs of Related Techniques
(See Section 4.5 for methodology)

Primarily, irregular codes are executed on general purpose processors (GPPs), which incur considerable overheads in per-instruction processing, both in extracting instruction-level parallelism and for maintaining instruction-precise state. Two broad specialization approaches have arisen to address these challenges. The first is to use simplified and serialized low-power hardware in commonly used low-ILP code regions for better energy efficiency. Examples include architectures like bigLITTLE [68] and Composite Cores [30], which switch to an inorder core when ILP is unavailable, and "accelerators" like BERET [25], Conservation Cores [27] and QsCores [69]. The other approach is to enhance the GPP for energy-efficiency, like adding μop caches, loop caches, and in-place loop execution techniques like Revolver [70].

To highlight the benefits and limitations of existing approaches targeting irregular codes, Figure 4.1(a) shows their energy and performance advantages when integrated into several GPP cores[2]. Figure 4.1(b) is similar, but here an oracle scheduler only allows regions with slowdown of $< 10\%$.

[2]Note here that we allow switching arbitrarily at a fine-granularity and hence bigLITTLE subsumes Composite Cores.



An “ideal” dataflow processor is only constrained by the program’s control and data-dependencies, and not by any execution resources. It is also non-speculative, and incurs latency when transferring values between control regions. For its energy model, only functional units and caches are considered.

Figure 4.2: Potential of Ideal Explicit-Dataflow Specialization

These results show that low-power hardware approaches are effective when integrated to small inorder cores (1.5× energy-efficiency), but usually cost too much performance to be useful for OOO GPPs. Techniques like in-place loop execution are also beneficial, but can only improve performance/energy by a few percent, because they rely on expensive instruction window, reorder-buffer and large multi-ported register-file access, even during loop specialization mode. Overall, speedup and energy benefits are limited to less than 1.1× on large GPPs.

Dataflow The common feature of the above architectures is that they are fundamentally Von Neumann or “control flow” machines. However, there exist well-known architectures which eschew complex OOO hardware structures, yet can extract significant ILP, called *explicit-dataflow* architectures. These include early Tagged Token Dataflow [71], as well as the more recent TRIPS [72], WaveScalar [73] and Tartan [74]. But explicit-dataflow architectures show no signs of replacing conventional GPPs, for at least three reasons. First, control speculation is limited by the difficulty of implementing dataflow-based squashing. Second, the latency cost of explicit data communication can be prohibitive [75]. Third, compilation challenges have proven hard to surmount [76].

Overall, dataflow machines researched and implemented thus far have failed to provide higher instruction-level parallelism, and their theoretical promise of low power and yet high performance remains unrealized for irregular codes.

Unexplored Opportunity What has thus far remained unexplored is fine-grained interleaving of explicit-dataflow with Von Neumann execution – i.e. the theoretical and practical limits of being able to switch with low cost between an explicit-dataflow hardware/ISA and a Von Neumann ISA.

The potential benefits of an *ideal* hybrid architecture (ideal dataflow + four-wide OOO) are shown in Figure 4.2(a). Above each bar is the percentage of execution time in dataflow mode. Figure 4.2(b) shows the overall energy and performance trends for three different GPPs.

These results indicate that hybrid dataflow has significant potential, up to 1.5× performance for an OOO4 GPP (2× for OOO2), as well as over 2× average energy-efficiency improvement, significantly higher than previous specialization techniques. Furthermore, the preference for explicit-dataflow is frequent, covering around 65% of execution time, but also intermittent and application-phase dependent. The percentage of execution time in dataflow mode varies greatly, often between 20% to 80%, suggesting that phase types can exist at a fine grain inside an application.

When/Why Explicit-Dataflow? To understand when and why explicit-dataflow can provide benefits, we consider the program space along two dimensions: control regularity and memory regularity. Figure 4.3 shows our view on how different programs in this space can be executed by other architectures more efficiently than with an OOO core. Naturally, vector-architectures are the most effective when memory access and control is highly regular (see ❶). When memory latency bound (see ❷), little ILP will be available, and the *simplest possible* hardware will be the best (a low-power engine like BERET [25] or CCores [27]). An explicit-dataflow engine could also fill this role.

There are two remaining regions where explicit-dataflow has advantages over OOO. First,

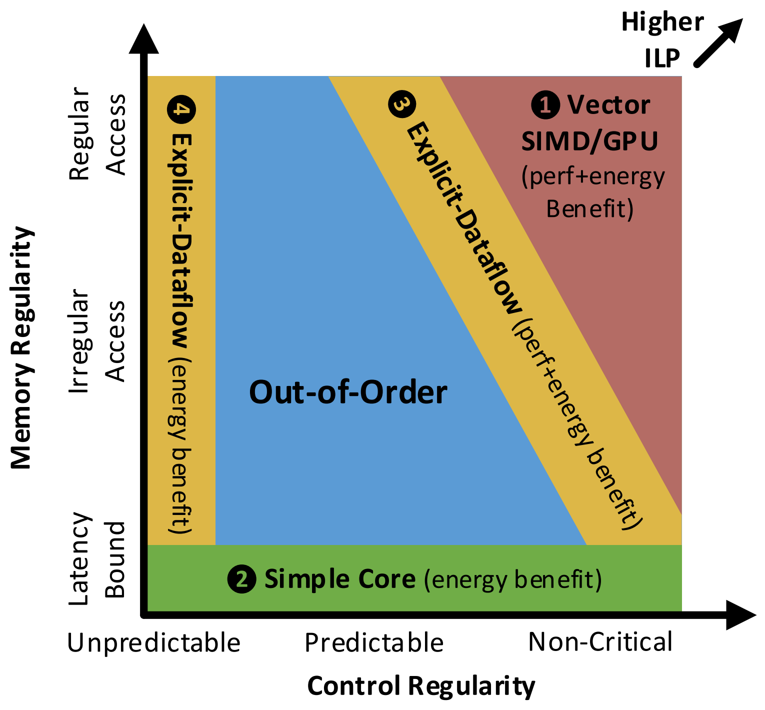

Higher ILP
Memory Regularity
Regular Access
Irregular Access
Latency Bound
❹ Explicit-Dataflow (energy benefit)
Out-of-Order
❸ Explicit-Dataflow (perf+energy benefit)
❶ Vector SIMD/GPU (perf+energy Benefit)
❷ Simple Core (energy benefit)
Unpredictable
Predictable
Non-Critical
Control Regularity

Figure 4.3: Arch. Effectiveness based on App. Characteristics

when the OOO processor's issue width and instruction window size limits the achievable ILP (see ❸), explicit-dataflow processors can exploit this through more efficient hardware mechanisms, achieving higher performance and energy efficiency. Second, when control is not predictable, which would serialize the execution of the OOO core [3] (see ❹), explicit-dataflow can execute the same code with higher energy efficiency by avoiding mispeculation overheads [4].

Overall, this suggests that a heterogeneous Von Neumann/explicit-dataflow architecture with fine-granularity switching can provide significant performance improvements along with power reduction, and thus lower energy.

Arch. Parameter	**TRIPS** [77]	**WaveScalar** [73]	**CCA** [26]	**DySER** [78]	**BERET** [25]	**SEED**
Execution Units	Homogeneous FUs	Heterog. FUs and tag matching ~~①~~	Triangular mesh + heterog. FUs ~~②~~	Grid of heterogeneous FUs	Serialized compound FUs ~~③~~	Compound Functional Units (CFUs)
Storage Structures	Multiple SRAMS per Grid ~~①~~	Banked queues	Pipelined FIFOs ~~②~~	Pipelined FIFOs ~~②~~	CRAM, Internal Reg. Files	Single ported SRAM structures
Interconnection Network	Large 2D mesh network ~~①~~	Large hierarchical interconnect ~~①~~	Multi-level bus	Switches	Bus-based	Bus-based + Arbiter
Communication	Dynamic routing ~~①~~	Results, tokens across clusters ~~③~~	Configuration, results over bus	Tightly coupled with GPP ~~③~~	Config. messages, results	Fixed sized packet based
Control Flow Strategy	Dataflow predication ~~③~~	ϕ and ϕ_{-1} instructions	Control flow assertion	Predication-only ~~②~~	Speculates hot traces only ~~②~~	Switch instructions

① : Low-area & low-power; ② : Application Generality; ③ : Low-overhead Dataflow;

Table 4.1: Suitability of Related Architectures for Explicit-Dataflow Specialization

4.2 SEED: An Architecture for Fine-Grain Dataflow Specialization

Our primary observation is the potential for exploiting the heterogeneity of execution models between Von Neumann and Dataflow at a fine grain. Attempting to exploit this raises this paper's main concern: *how can we exploit dataflow specialization with simple, efficient hardware?* We argue that any solution requires three properties: ① Has low-area and low-power, so that integration with the GPP is feasible. ② Is general enough to target a wide variety of workloads. ③ Achieves the benefits of dataflow execution with few overheads. Our codesign approach involves exploiting properties of frequently executed program regions, a combination of power-efficient hardware structures, and a set of compiler techniques.

First, we propose that requirement ①, low area and power, can be addressed by strictly specializing for a common, yet simplifying code type: *fully-inlined nested loops* with a limited total static instruction count. Limiting the number of per-region static instructions limits the size of the dataflow tags, and eliminates the need for an instruction cache; both of which reduce hardware

[3]Provided predication was not an option

[4]In addition, instructions after a control flow merge point can be executed before the dependent control decision is made, because of the dataflow ordering of control. This can help the dataflow processor keep pace with the OOO.

complexity. In addition, ignoring recursive regions and only allowing in-flight instructions from a single context eliminates the need for tag matching hardware – direct communication can be used instead. Targeting nested-loops also satisfies requirement ②: these regions can cover a majority of real applications' dynamic instructions.

To achieve low-overhead dataflow execution, requirement ③, the cost of communication must be lowered as much as possible. We achieve this through a judicious set of microarchitectural features. First, we use a *distributed-issue* architecture, which enables high instruction throughput with *low ported RAM* structures. Second, we use a *multi-bus network* for sustaining instruction communication throughput at low latency. Third, we use *compound instructions* to reduce the data communication overhead.

Using the above insights creates new compiler requirements: 1. Ability to create appropriately sized inlined nested loops matching the hardware constraints. 2. Algorithms for creating compound instruction groupings which minimize the communication overhead. For the first requirement, we can use aggressive inlining and loop nest analysis, and the second by employing integer linear programming models.

To address the architecture and compiler challenges, we propose SEED: Specialization Engine for Explicit-Dataflow, shown at a high level in Figure 4.4. To achieve this, it uses a distributed architecture with well-known dataflow principles to achieve high instruction parallelism, and mitigates the dataflow communication overheads by using compound instructions as the unit of computation. We explain in more detail in the next section.

4.3 SEED Architecture

We begin the description of SEED by giving insight into why some architectural innovation is required and how our solution borrows mechanisms from related techniques. We then give an example SEED program which elaborates the basic mechanics of SEED execution. Subsequently, we detail the SEED microarchitecture from the bottom-up by describing SEED's sub-modules, its

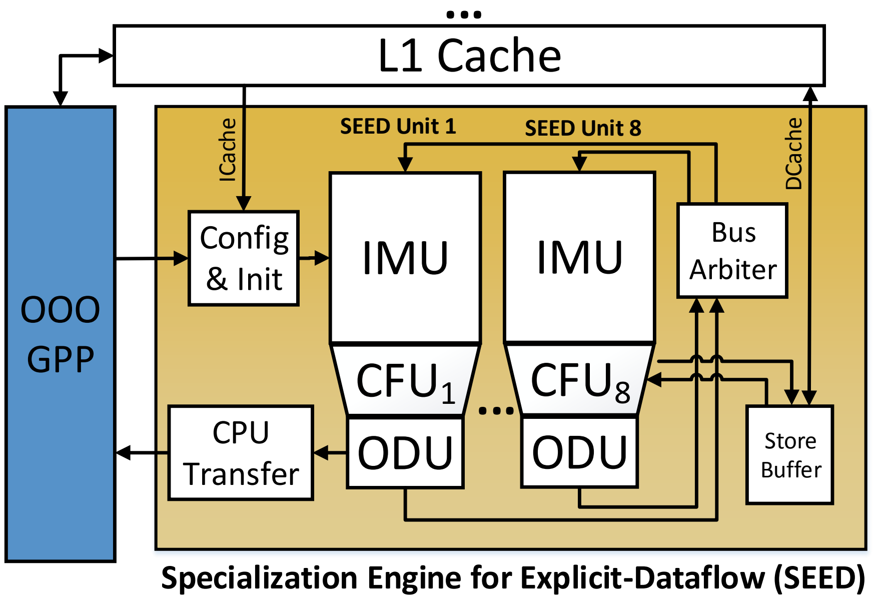


Figure 4.4: High-Level SEED Integration & Organization
(IMU: Instruction Management Unit; CFU: Compound Functional Unit; ODU: Output Distribution Unit)

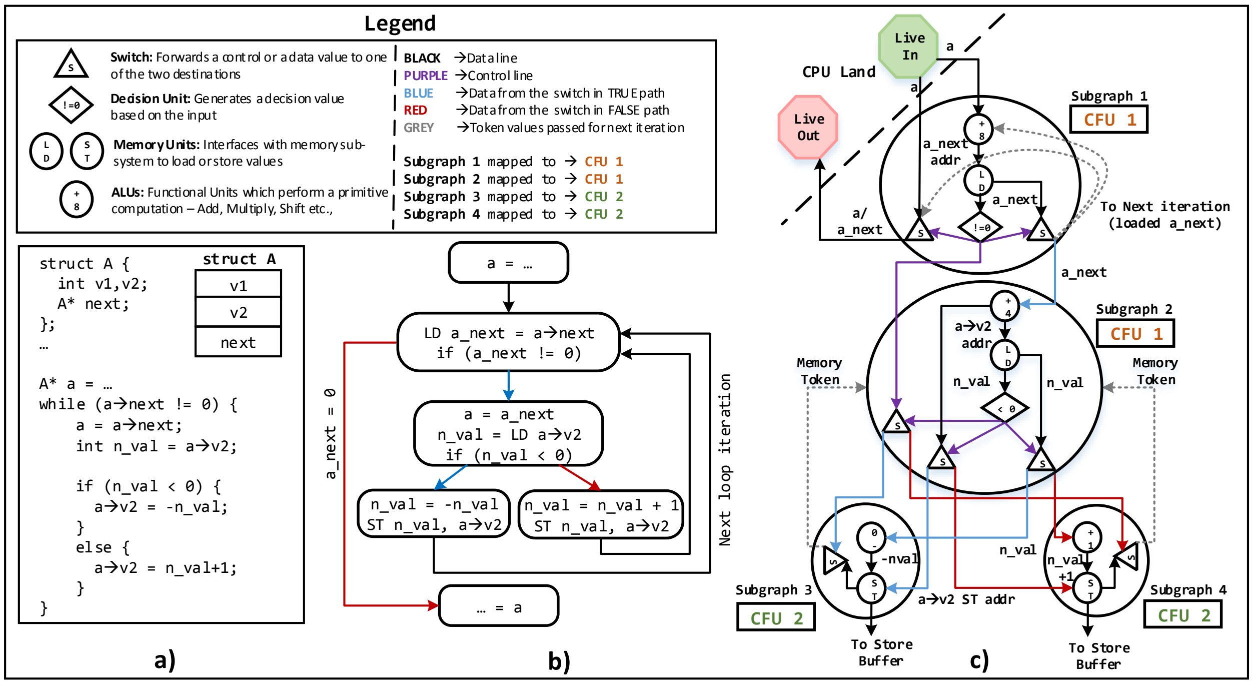
interconnection network and GPP integration.

We emphasize here that the organization of SEED is not the primary contribution of this chapter, rather, it is a tool for understanding the potential of dataflow specialization.

Architectural Innovation

In exploring the opportunity of fine-grain explicit-dataflow, it is important to consider whether existing architectures would be sufficient. We list five related architectures in Table 4.1, and describe their execution and storage units and their strategy for value-communication and control flow. Each cell also lists our opinion of whether the design choice would not meet the previously discussed requirements (low-area/power, generality, and low-overhead dataflow).

TRIPS and WaveScalar are designed for whole-program dataflow execution, and use higher-power, higher-area structures. TRIPS uses a large dynamically routed mesh network and WaveScalar uses complex tag-matching and a large hierarchical interconnect. The remaining architectures have much lower power and area, but are not general enough. None of them can offload entire loop



Legend
Switch: Forwards a control or a data value to one of the two destinations
Decision Unit: Generates a decision value based on the input
Memory Units: Interfaces with memory sub-system to load or store values
ALUs: Functional Units which perform a primitive computation – Add, Multiply, Shift etc.,
BLACK →Data line
PURPLE →Control line
BLUE →Data from the switch in TRUE path
RED →Data from the switch in FALSE path
GREY →Token values passed for next iteration
Subgraph 1 mapped to → CFU 1
Subgraph 2 mapped to → CFU 1
Subgraph 3 mapped to → CFU 2
Subgraph 4 mapped to → CFU 2
struct A { int v1,v2; A* next; }; …
A* a = … while (a→next != 0) { a = a→next; int n_val = a→v2; if (n_val < 0) { a→v2 = -n_val; } else { a→v2 = n_val+1; } }
struct A
v1
v2
next
a = …
LD a_next = a→next if (a_next != 0)
a = a_next n_val = LD a→v2 if (n_val < 0)
n_val = -n_val ST n_val, a→v2
n_val = n_val + 1 ST n_val, a→v2
… = a
a_next = 0
Next loop iteration
Live In
Live Out
CPU Land
a
a_next addr
a_next
a/ a_next
Subgraph 1
CFU 1
To Next iteration (loaded a_next)
a→v2 addr
n_val
Memory Token
Subgraph 2
CFU 1
-nval
n_val +1
a→v2 ST addr
Subgraph 3
CFU 2
Subgraph 4
CFU 2
To Store Buffer
a)
b)
c)

Figure 4.5: a) Example C loop; b) Control Flow Graph (CFG); c) SEED Program Representation;

regions in general – only the computation in CCA and DySER or hot loop-traces in BERET.

However, aspects of these architectures can be borrowed: the principle of offloading to a dataflow processor from DySER, the concept of efficient compound FUs from BERET and mechanisms for efficient and general dataflow-based control from WaveScalar. The next section describes how SEED combines these design aspects using an example program.

Example SEED Dataflow Program

Figure 4.5 shows an example loop for a simple linked-list traversal, where a conditional computation is performed at each node. This figure shows the original program, control flow graph (CFG), and the SEED program. The SEED representation strongly resembles those of previous dataflow architectures, where the primary difference is that instructions are grouped here into subgraphs. Familiar readers may skip ahead.

Data-Dependence Similar to other dataflow representations, SEED programs follow the dataflow firing rule: instructions execute when their operands are ready. To initiate computation, live-in values are sent. During dataflow execution, each instruction forwards its outputs to dependent instructions, either in the same iteration (solid line in Figure 4.5(c)), or in a subsequent iteration (dotted line). For example, the `a_next` value loaded from memory is passed on to the next iteration for address computation.

Control-Flow Strategy Control dependencies between instructions are converted into data dependencies. SEED uses a *switch* instruction, similar to other proposals, which forwards the control or data values to one of two possible destinations, depending on the input control signal. In the example, depending on the `n_val` comparison, `v2` is forwarded to either the `if` or `else` branch. This strategy enables control-equivalent regions to spawn simultaneously.

Enforcing Memory-Ordering SEED has a software mechanism to enforce correct memory-ordering semantics. When the compiler identifies dependent (or aliasing) instructions, the program must serialize these memory instructions through explicit tokens. In this example, the stores of `n_val` can conflict with the load from the next iteration (e.g. when the linked list contains a loop), and therefore, memory dependence edges are required between these instructions.

Executing Compound Instructions To mitigate communication overheads, the compiler groups primitive instructions (e.g. adds, shifts, switches, etc.) into subgraphs and executes them on compound functional units (CFUs). These are logically executed atomically. The example program contains four subgraphs, mapped to two CFUs.

SEED Microarchitecture

Our microarchitecture achieves high instruction parallelism and simplicity by using distributed computation units. The overall design is composed of 8 *SEED units*, where each SEED unit is

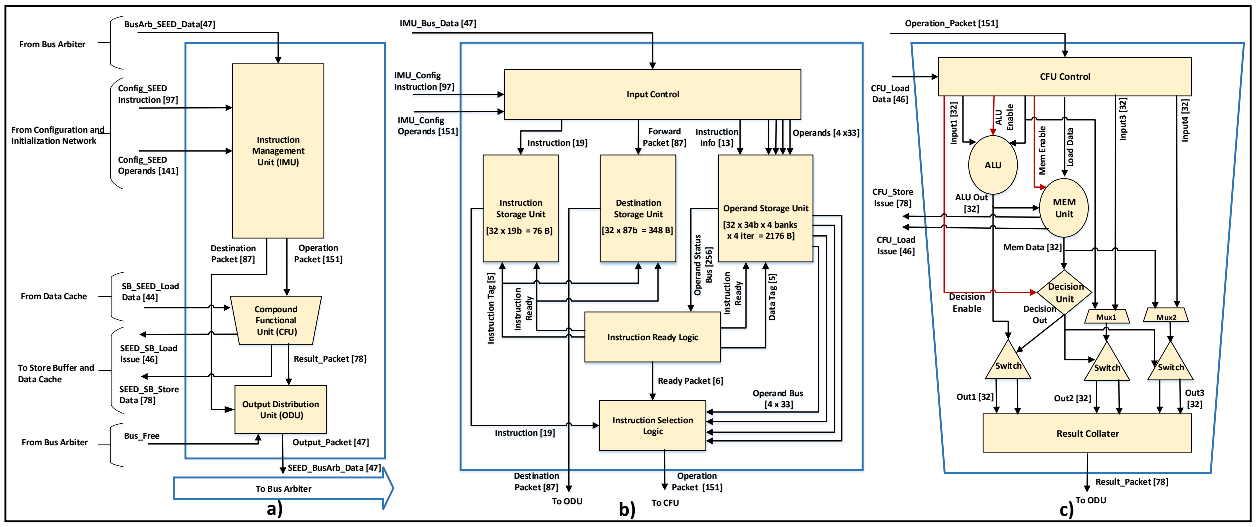


Figure 4.6: a) SEED Unit; b) IMU Microarchitecture; c) CFU Microarchitecture;

organized around one CFU. The SEED units communicate over a network, as shown in Figure 4.4. We describe the SEED unit internals and interconnect below (shown in Figure 6.8).

Compound Functional Unit (CFU) As mentioned previously, CFUs are composed of a fixed network of primitive FUs (adders, multipliers, logical units, switch units etc.), where unused portions of the CFU are bypassed when not in use. Long latency instructions (e.g. loads) can be buffered, and passed by subsequent instructions. An example CFU is shown in Figure 6.8 (c). Our design uses the CFU mix from existing work [25], where CFUs contain 2-5 operations. Our current design embeds integer hardware, but floating point (FP) units can be added either by instantiating new hardware or by adding bypass paths into the host processor's FP SIMD units.

CFUs which have memory units will issue load and store requests to the host's memory management unit, which is still active while using SEED. Load requests access a 32-entry store buffer for store-to-load forwarding.

Instruction Management Unit (IMU) The IMU, shown in Figure 6.8 (b), has three responsibilities:

1. **Storing instructions, operands & destinations:** The IMU has storage locations for 32 com-

pound instructions, each with a maximum of four operands each, and we keep operand storage space for four concurrent loop iterations. This results in storage of 2600 bytes of data. All IMUs in eight SEED units combined has $\sim$20KB of storage. The static instruction storage is roughly equivalent to a maximum of 1024 non-compound instructions.

2. **Firing instructions:** Ready logic monitors the operand storage unit, and picks a ready instruction (when all operands are available), with priority to the oldest instruction. Then the compound instruction and its operands and destinations are sent to the CFU.

3. **Directing incoming values:** The input control pulls values from the network to appropriate storage locations based on the incoming instruction tag.

The primary unique feature of the IMU is that it allows “unrolled” operand storage for four iterations of the loop. This allows instructions to directly communicate to dependent instructions without using power hungry tag-matching CAM structures at each execution node.

Output Distribution Unit (ODU) The ODU is responsible for distributing the output values and destination packets (SEED unit + instruction location + iteration offset), to the bus network, and buffer them during bus conflicts.

Bus Architecture and Arbiter SEED uses a bus interconnect for forwarding the output packets from the ODU to a data dependent compound instruction, present in either the same or another SEED unit. Note that this means dependent instructions communicating over the bus cannot execute in back-to-back cycles. To handle network congestion, the bus arbiter monitors the packet requests, and forwards up to three values on three parallel buses.

Integration with Core The integration with the host core follows the basic strategy outlined in Chapter 2. The host core communicates with SEED to initialize configuration (for instructions, destination storage, and loop invariant constants in the operand storage), and send and receive input/output live values.

To handle context switching, the current live operands must become part of the architectural state. To mitigate the overhead, we delay context switches until the current inner-loop iterations quiesce. In the worst case, we estimate needing to save 2KB of data, though typically the amount of live data is much less.

4.4 SEED Compiler Considerations

The two main responsibilities of the compiler are determining which regions to specialize and scheduling instructions into CFUs inside SEED regions. We discuss scheduling in detail in Chapter 6, and discuss region selection heuristics here.

As SEED may only execute fully-inlined nested-loops, the compiler must find or create such regions. There are two main goals: 1. Finding small enough regions to fit the hardware, and 2. Not hurting performance by aggressively applying SEED, when either the OOO core (through control speculation) or the SIMD units would have performed better.

For the first goal, a bottom up traversal of the loop-nest tree can be used to find appropriately sized regions. Enough space can be left for unrolling inner loops, which can increase inter-loop parallelism when it exists.

For the second goal, either static or dynamic options are possible. For the static approach, simple heuristics will likely suffice – i.e. do not perform explicit-dataflow when control is likely to be on the critical path. A dynamic approach can be more flexible; for example, training on-line predictors to give a runtime performance estimate based on per-region statistics. Other works have shown such mechanisms to be highly effective [30, 79], and we therefore do not evaluate or implement this aspect of the compiler/runtime. Instead, we use an oracle scheduler, as described in the evaluation. Chapter 5 discusses practical scheduling in the context of multiple accelerators.

Suite	Benchmarks
Mediabench	cjpeg, djpeg, gsmdecode, gsmencode cjpeg2, djpeg2, h263enc, h264dec, jpg2000dec, jpg2000enc, mpeg2dec, mpeg2enc
SPECint	164.gzip, 181.mcf, 175.vpr, 197.parser, 256.bzip2 429.mcf, 403.gcc, 458.sjeng, 473.astar, 456.hmmer

Table 4.2: Benchmarks

GPP	Characteristics
Little (IO2)	Dual Issue, 1 load/store port.
Medium (OOO2)	64 entry ROB, 32 entry IW, LSQ: 16 ld/20 st, 1 load/store ports, speculative scheduling.
Big (OOO4)	168 entry ROB, 48 entry IW, LSQ: 64 ld/36 st, 2 load/store ports, speculative scheduling.

Table 4.3: GPP Cores

4.5 Evaluation Methodology

Benchmark Selection The benchmarks we chose were from SPECint and Mediabench [80], representing a variety of control and memory irregularity, as well as some regular benchmarks (see Table 5.2).

GPP Characteristics All cores are x86, have 256-bit SIMD, and have a common cache hierarchy: a 2-way 32KB I$ and 64KB L1D$, both with 4 cycle latencies, and an 8-way 2MB L2$ with a 22 cycle hit latency. Also, to exclude the effects of frequency scaling, all cores run at 2Ghz. The differences between GPP configurations are highlighted in Table 4.3. The OOO4 has 3 ALUs, 2FPs, and 1 Mul/Div unit, which are scaled according to the GPP issue width.

Module	Area (mm^2)	Module	Area (mm^2)
IMU	0.034	Internal Network	0.058
CFU	0.011	Total SEED Unit	0.114
ODU	0.010	Bus Arbiter	0.016
Total (8 SEED Units + Bus Arbiter)			0.926

Table 4.4: SEED Area Breakdown

4.6 Evaluating Dataflow-Specialization Potential

To understand the potentials and tradeoffs of dataflow specialization while exploiting nested-loop regions, this section attempts to answer the following questions:

Q1. Can adequate instruction groupings be found?

Q2. Is the proposed design practical: what is the area cost?

Q3. How much performance can targeted regions provide?

Q4. What are the sources of performance differences?

Q5. Which GPP cores can we enhance with our technique?

Q6. Would it still be useful if GPPs were more efficient?

Q7. Besides performance/energy, are there other benefits?

Q8. How does SEED compare to related approaches?

Q1. Can adequate instruction groupings be found?

Figure 4.7 is a histogram of per-benchmark compound instruction sizes, showing on average 2-3 instructions. This is relatively high considering that compound instructions cannot cross control regions. Some singletons are necessary, however, either because control regions lack dependent computation, or because combining certain instructions would create additional critical-path dependencies.

Answer: *Yes, most dynamic instructions are grouped into a compound unit.*

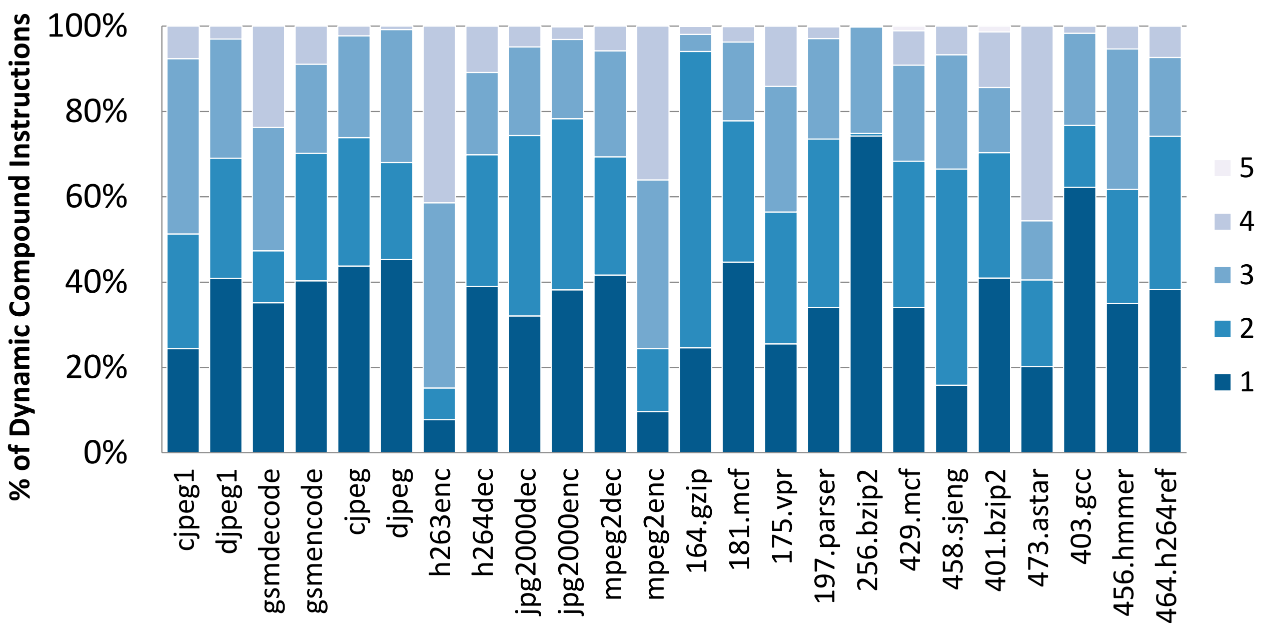


Figure 4.7: Compound Instruction Size Histogram

Q2. Is the proposed design practical?

To determine the area, we have implemented the SEED architecture in Verilog and synthesized the design using a 32nm standard cell library with the Synopsys Design Compiler. CACTI [43] was used for estimating SRAM area. Our results show that each SEED unit occupies reasonable area and all eight SEED units and bus arbiter together take up an area of 0.93 mm^2. Table 4.4 shows the area breakdown.

We also synthesized the design for 2 GHz, and the estimated power is 90mW based on its default activity factor assumptions for the datapath[5].

Answer: *The simple design and low area/power quantitative results show that the SEED unit is practical.*

[5]For fairness of comparing against McPAT-based GPP models, we have also used a McPAT-based model for SEED. For performance benefit regions (vs OOO4) the McPAT model reports an average power of 125mW, meaning this model should be conservative.

	Benchmark	Func. for SEED Region	% Exec. Insts	Vect- orized	OOO4 IPC	SEED IPC	Ideal-DF IPC	SEED En-Red.	BPKI	BMPKI	$MPKI	Explanation
Perf. > OOO4	jpg2000dec	jas_image_encode	50%		2.5	12.8	21.8	9.1	101	0	0	High Exploitable ILP
	429.mcf	primal_bea_mpp	37%		0.8	2.8	8.3	4.6	152	10	96	Higher Memory Parallelism
	cjpeg-1	encode_mcu_AC_refine	24%		2.5	5.9	6.2	4.2	48	0	2	Indirect Memory + High ILP
	181.mcf	primal_bea_mpp	31%		0.9	1.8	9.6	3.0	170	8	106	Higher Memory Parallelism
	djpeg-2	ycc_rgb_convert	33%		2.7	5.4	12.0	3.5	29	0	0	Indirect Memory + High ILP
	456.hmmer	Viterbi*	73%		2.9	5.4	7.3	4.5	32	0	4	High Exploitable ILP
	458.sjeng	std_eval	5%		2.4	3.7	4.1	3.8	126	5	0	High Exploitable ILP
	gsmdecode	Gsm_Short_Term_Syn...	61%		2.4	3.1	3.4	4.5	92	0	0	High Exploitable ILP
	cjpeg-2	compress_data	48%	✓	2.2	2.7	4.9	3.5	58	8	0	High Exploitable ILP
Perf. ≈ OOO4	gsmencode	Gsm_Long_Term_Pred...	49%		1.9	2.2	2.7	3.5	5	0	0	Modest ILP + Comm. Overheads
	djpeg-1	decompress_onepass	39%		2.6	2.7	3.6	3.6	18	1	0	Indirect Memory + Moderate ILP
	h263enc	MotionEstimation	98%		2.0	1.9	8.7	3.2	18	0	0	Comparable Performance
	164.gzip	inflate	23%		1.9	1.7	2.3	2.0	81	0	10	Modest ILP + Comm. Overheads
	473.astar	wayobj::fill	96%		1.1	1.0	1.1	3.3	114	31	2	Avoids Branch Misses, Modest ILP
	h264dec	decode_one_macroblock	21%		0.4	0.4	0.4	1.9	39	0	0	Comparable, Low ILP
	jpg2000enc	jpc_enc_encpkt	3%		2.1	1.8	2.0	1.3	135	6	2	Comparable Performance
Perf. < OOO4	403.gcc	ggc_mark_trees	4%		0.5	0.4	0.4	1.0	66	2	2	Comparable, Low ILP
	464.h264ref	SetupFastFullPelSearch	29%		1.5	1.3	1.7	2.7	40	0	0	Short Region (340 Dyn Insts)
	175.vpr	try_swap	49%		1.4	1.2	6.9	2.1	88	17	5	Avoids B-Misses, Comm. Overhead
	mpeg2enc	fullsearch.constprop.3	93%		1.9	1.5	2.9	3.9	17	0	0	Moderate ILP, Comm. Overhead
	mpeg2dec	conv422to444	31%		2.7	2.1	3.0	2.8	68	0	2	Moderate ILP, Comm. Overhead
	197.parser	restricted_expression	17%		3.3	1.6	3.7	1.4	108	0	0	Short Region (300 Dyn Insts)
	401.bzip2	BZ2_compressBlock	31%	✓	4.3	1.5	1.5	0.8	97	3	3	Region Vectorized
	256.bzip2	compressStream	99%	✓	13.5	2.0	2.0	0.4	83	0	0	Region Vectorized

Table 4.5: Region-Wise Comparison of OOO4 to SEED, Showing *only* top region per benchmark, Highest to Lowest Relative Perf.
(%Exec. Insts: % of original program executed by SEED; Vectorized: whether the GPP vectorized the region, SEED IPC: Effective IPC of SEED, Ideal-DF IPC: IPC of Ideal-dataflow, En-Red: SEED's Energy Reduction, BPKI: Branches per 1000 μops, BMPKI: Branch Mispred. per 1000 μops, $MPKI: Cache misses per 1000 μops)

Q3. How much performance benefit is possible?

To understand if there are potential performance benefits, we compare the speedups of SEED to our most aggressive design (OOO4) on the most frequent nested-loop regions of programs (each >1% total insts). The results, in Figure 4.8, show that different regions have vastly different performance characteristics, and some are favored heavily by one architecture.

Answer: *Nearly 3-5× speedup is possible, and many regions show significant speedup.*

Q4. What are the sources of performance differences?

Table 4.5 presents details on the highest contributing region from each benchmark. Note that the SEED IPC is an *effective IPC* which uses the GPP's instructions as total instructions. This allows

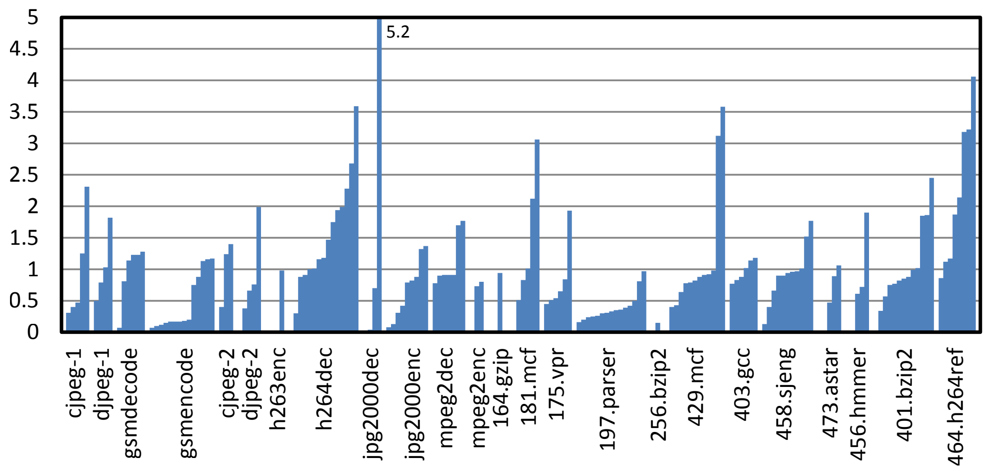


Figure 4.8: Per-Region SEED Speedups

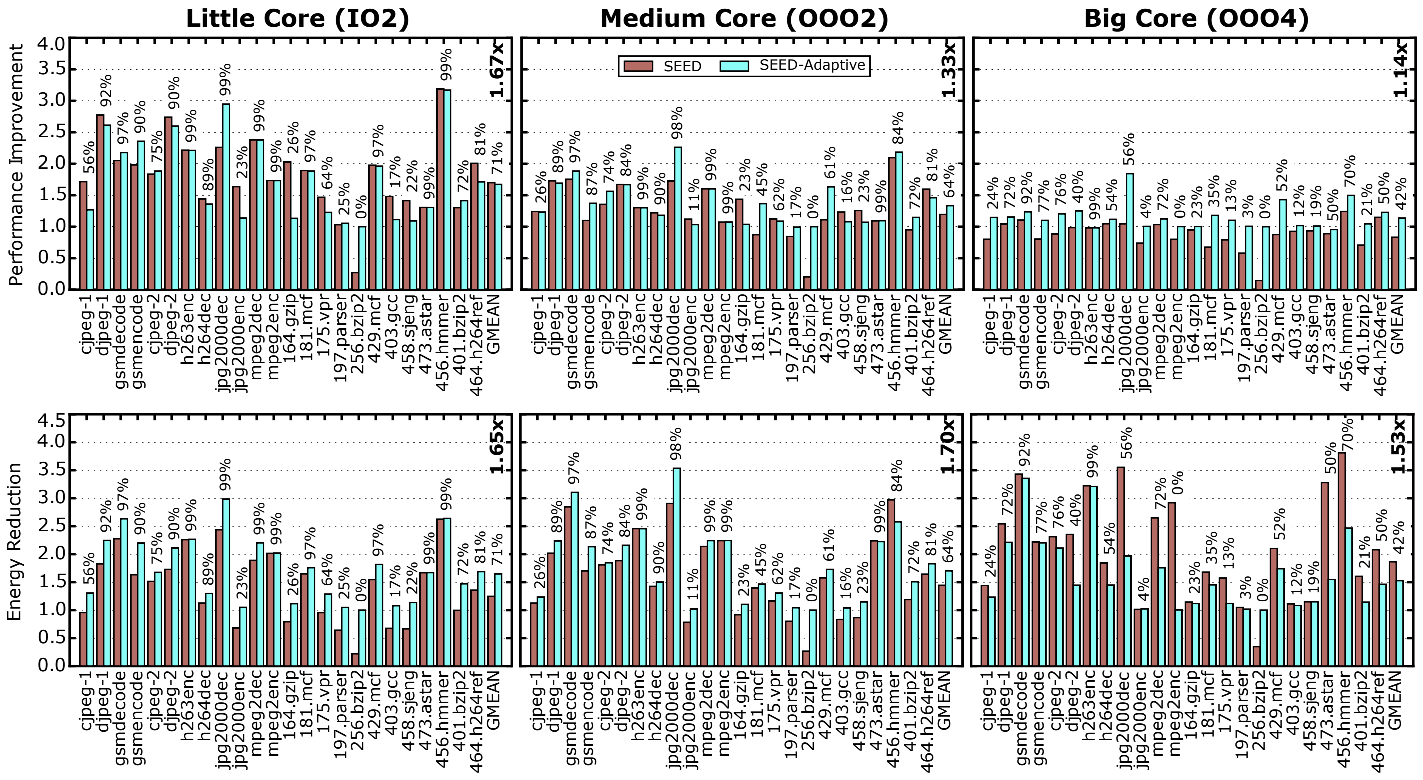


Figure 4.9: SEED Specialization for Little, Medium, and Big Cores

easier comparison, as many instructions for loading immediates and managing register spilling are not required in SEED. We discuss these in three categories:

Perf.&Energy Benefit Regions Compared to the OOO4-wide core, SEED can provide high speedups for certain applications, coming from the ability to exploit higher ILP in compute-intensive regions and from breaking the instruction window barrier for achieving higher memory parallelism.

In the first category are `jpg2000dec`, `cjpeg` and `djpeg`, which can exploit ILP past the issue width of the processor, while simultaneously saving energy by using less complex structures. Often, these regions have indirect memory access which precludes SIMD vectorization. In the second category are `181.mcf` and `429.mcf`, which experience very high cache miss rates, and clog the instruction window of the OOO processor. SEED is only limited by the store buffer size on these benchmarks.

Energy Benefit-Only Regions These regions have similar performance to the OOO4, but are more energy efficient by 2-3×. Overall, ILP tends to be lower, but control is mostly off the critical path, allowing dataflow to compete. This is the case for `djpeg-1` and `h264dec`. Benchmarks like `gsmencode` and `164.gzip` actually have some potential ILP advantages, but are burdened by communication overhead between SEED units. Benchmark `h263enc` actually has a very high potential ILP, but requires multiple *instances* of the inner loop (not just iterations) in parallel, which SEED does not support.

Contrastingly, benchmarks `473.astar` and `jpg2000enc` have significant control, but still perform close to the OOO core. These benchmarks make up for the lack of speculation by avoiding branch misses and relying on the control-equivalent spawning that dataflow provides.

Perf. Loss Regions Several SEED regions lose performance versus the OOO4 core, shown in the last set of rows in Table 4.5. The most common reason is additional communication latency on the critical path, affecting regions in `403.gcc`, `mpeg2dec` and `mpeg2enc`. Also, certain benchmarks have load-dependent control, like `401.bzip2`, causing a low potential performance for dataflow.

These are fundamental dataflow limitations. In two cases, configuration overhead hurt the benefit of a short-duration region(`464.h264ref` and `197.parser`). In practice, these regions would not be executed on SEED. Finally, some of these regions are vectorized on the GPP, and SEED is not optimized to exploit data-parallelism. This affects `401.bzip2` and `256.bzip2`.

Answer: *Speedups come from exploiting higher memory parallelism and instruction parallelism, and avoiding mispeculation on unpredictable branches. Slowdowns come from the extra latency cost on more serialized computations.*

Q5. Which GPP cores can we enhance with SEED?

Here we consider integrating with a little, medium, and big core, as outlined in Table 4.3. To eliminate compiler/runtime heuristics on when to accelerate, we consider using an oracle scheduler, which uses perfect information to decide when to use the OOO core, SEED, or SIMD. We report results for performance and energy reduction of all cores in Figure 6.11. The first bar in each graph shows the relative metric to the baseline, when *always* using SEED. The second bar, "adaptive," shows the result of the oracle scheduler, optimizing for Energy-Delay product, and not allowing any regions which degrade performance by more than 10%. We discuss the implications for each GPP type below.

Little GPP (IO2) For the little core, SEED provides a geometric mean performance and energy improvement of about 1.65×, and SEED runs for 71% of the execution time. For these benchmarks, SEED mainly loses performance on vectorized workloads like `256.bzip2`.

Medium GPP (OOO2) For the medium core, SEED is the chosen execution substrate for 64% of the execution time, providing energy reduction of 1.7×, and performance of 1.33×. Even if always chosen, in only four cases does it hurt performance, and in most cases energy-efficiency gain is significant.

Big GPP (OOO4) For the big core, for reasons described in the previous section, SEED is chosen less, around 42% of execution time. Overall though, it still provides 1.14× performance and 1.53× energy efficiency improvement.

Answer: *All cores can achieve significant energy benefits; little and medium cores can achieve significant speedup; and big cores receive modest performance improvement.*

Q6. Would this still be useful if GPPs were more efficient?

Figure 4.10(a) shows the performance, energy, and percentage of cycles that SEED is active across all workloads, while reducing the power of all GPP structures (not including the SEED unit). The x-axis is the factor by which the GPP is made more power efficient (1 means no change).

Naturally, the energy-reduction of all GPPs decrease as a direct effect of the changing parameters. More interestingly, the percentage of time SEED is chosen drops only by a few percent (and only for the little and medium cores), even if the GPP becomes 4× more energy efficient.

Answer: *Even future generations of power-efficient GPPs could take advantage of explicit-dataflow specialization.*

Q7. Beside energy/speedup, are there other benefits?

Figure 4.10(b) shows the effects of varying the region choice metric from energy-efficiency (E), to energy-delay (ED), and up to performance (D). Naturally, the little and medium cores are not particularly sensitive to the region choice metric, which is intuitive because SEED is a faster, yet still primarily lower-power design. The big core is quite sensitive; by optimizing for energy-efficiency (and using SEED more often) it can trade off 20% performance for over 40% energy efficiency on average.

Answer: *Explicit-dataflow specialization provides a microarchitectural mechanism for trading-off performance and energy-efficiency on large cores.*

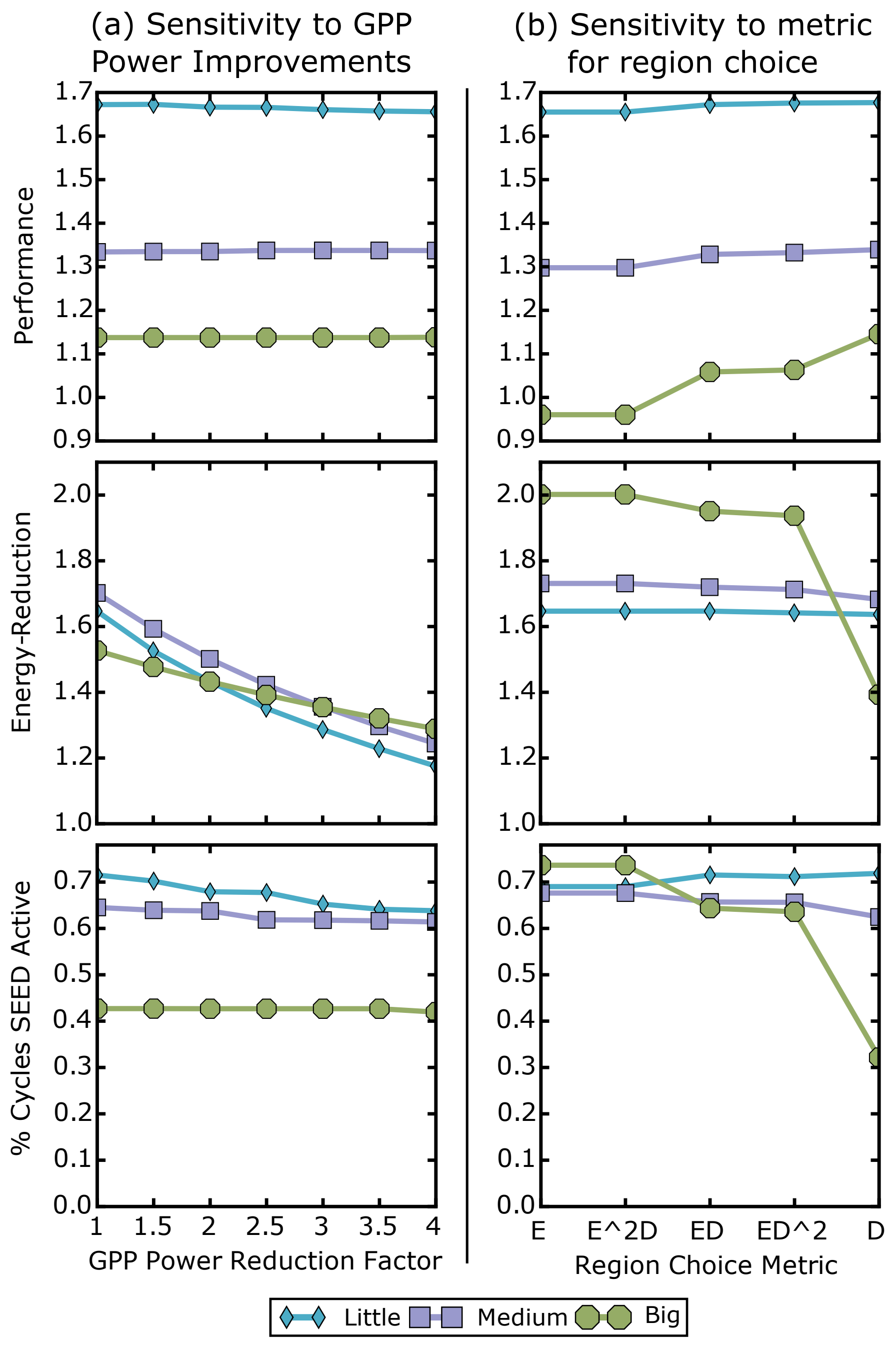


Figure 4.10: Sensitivity to GPP Improvements and Region Choice Metric

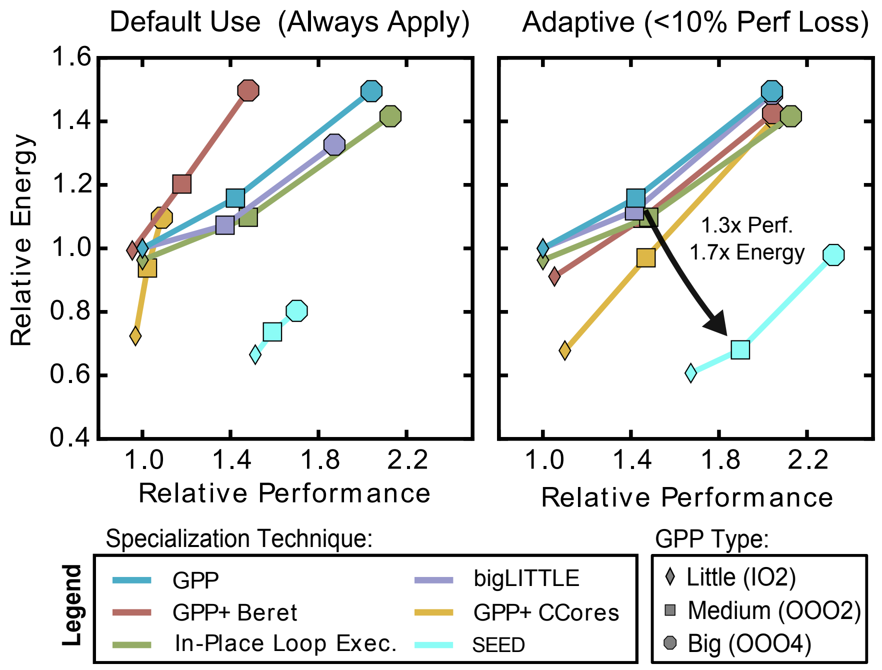


Figure 4.11: Comparison with Other Specialization Techniques

Q8. How does SEED compare with other specialization approaches?

An updated version of the first figure, Figure 4.11, compares SEED to existing techniques. In non-adaptive mode, SEED provides energy improvements for all cores, and performance enhancements for the inorder and medium cores. For adaptive mode, SEED improves both metrics across GPP cores types, significantly more than existing approaches. In terms of the overall design space, the OOO2+SEED cannot beat an OOO4 on average, but reaches within 15% performance (with much lower power). Also, though SEED improves OOO4 performance only modestly, the energy efficiency of OOO4+SEED is that of a simpler OOO2 GPP.

Answer: *Explicit-Dataflow specialization has significant potential beyond existing techniques across core types.*

4.7 Related Work

The notion of merging the benefits of Von Neumann with dataflow machines is far from new – we discuss the relationship to some of the original dataflow accelerators, some initial work in dataflow coprocessors, modern dataflow accelerators, and the relationship to similar techniques applied on a microarchitecture only level.

Early Dataflow Machines A large body of work in the dataflow paradigm is in executing explicitly parallel programs, like TTDA does with the Id programming language [71]. In that context, Iannucci proposes a hybrid architecture which adds a program counter to the TTDA to execute explicitly parallel programs more efficiently [81]. Along opposite lines, Buehrer and Ekanadham introduce another hybrid architecture which introduces mechanisms to support both sequential and explicitly parallel programming languages [82] for ease of transitioning. These works are orthogonal to ours, as they are attempting to target different programming models.

That said, SEED derives significant inspiration from the previous decades of dataflow research. One example is the Monsoon architecture [83], which improves the efficiency of matching operands by using an Explicit Token Store. This essentially eliminates complex matching hardware by allocating memory frames for instruction tokens and using offsets into the frame for token locations. Our strategy of using explicit offsets into the operand buffers provides similar benefits.

Dataflow Coprocessors Intellectually, a very similar work is Liu and Furber's Dataflow Coprocessor [84]. They also use an offload-based model of execution with a dataflow-based coprocessor. However, their microarchitecture only supports very simple code snippets typically found in the embedded domain, and supports limited ILP (instead focusing on power), while SEED is designed for high performance on potentially much larger and longer running regions. SEED could be viewed as an evolution of this work for modern OOO processors and challenging irregular workloads.

A concurrent work is the Memory Access Dataflow (MAD) architecture [85], which augments the GPP with a dataflow substrate for targeting memory access program phases. These phases occur either because the code is naturally memory intensive, or because an attached in-core accelerator offloads most of the computation. Conceptually, our work differs in that it explores the benefits of hybrid dataflow execution in both memory and computation intensive regions. In terms of the microarchitecture, they are both essentially speculation-free dataflow execution, strictly-specialize for nested-loops, and use queue-like structures for data storage. For computation, MAD uses a spatial grid of statically routed FUs, while SEED uses clustered-instruction execution.

Core Enhancements Revolver's [70] in-place loop execution somewhat resembles the in-place nested-loop acceleration of SEED, but uses higher-power structures. Another related OOO-enhancement is the ForwardFlow [86] architecture, which is also a CAM-free execution substrate using explicit pointer-based communication of values. Though it is more energy-efficient than a typical OOO design, it still suffers overheads of fetch and decode, centralized register-file access, and still must dynamically build the dependence graph.

4.8 Summary

This chapter demonstrated that there exists opportunity for fine-grain heterogeneity between an OOO core and explicit-dataflow processor, and quantified the potential benefits. It described the design of the Specialization Engine for Explicit-Dataflow (SEED) – a design which exploits strict specialization of nested loops, and yet is still widely applicable, and combines known dataflow-architecture techniques for high energy efficiency and performance.

In the design-space exploration, we explored the potential of dataflow heterogeneity by integrating SEED into little (Inorder), medium (OOO2) and big (OOO4) cores. We show all design points achieve $> 1.5\times$ energy benefit by power-gating the OOO core when SEED is active. For speedup, little, medium and big cores achieve 1.67×, 1.33× and 1.14× over the non-specialized design. Finally, our analysis connected workload behaviors to dataflow profitability, showing that code with high memory parallelism, instruction parallelism, and branch unpredictability is highly profitable for dataflow execution.

Overall, in the context of dataflow research, our work has shown how traditional Von Neumann OOO and explicit-dataflow architectures favor different workload properties, and that fine-grain interleaving can provide significant and achievable benefits over either execution model alone.

5 MULTI-BEHAVIOR SPECIALIZATION

This dissertation has discussed how Behavioral Specialized Accelerators (BSAs) can be designed to efficiently execute code with certain properties – even irregular code – and remain largely configurable or programmable. A natural question is how far can this paradigm be pushed: can multiple BSAs be composed synergistically inside an ExoCore architecture? The flexible nature of BSAs makes this question difficult, as different BSAs have overlapping strengths and weaknesses, and the differences affect the tradeoffs with the general purpose core type.

This chapter's goal is to elucidate the potentials of synergistic BSAs. We first discuss the intuition on why overlapping behaviors are both a problem and opportunity, and construct a synergistic combination of behaviors with high potential for specialization (Section 5.1). Next, we describe how to use those behaviors to create a synergistic set of accelerators (Section 5.2), and how to decide between these BSAs on a per-regions basis (Section 5.3). The following sections describe the evaluation methodology (Section 5.4), and the results and design space exploration (Section 5.5). Finally, we cover related work (Section 5.6) and summarize (Section 5.7).

5.1 Behavior Synergy

Recall from Chapter 1 that a behavior is a general characteristic of a program which hardware specialization can potentially exploit. Because of their generality, multiple behaviors can describe one particular code, and may even be highly correlated. This poses a problem and opportunity for any system with multiple general purpose accelerators: any set of targeted behaviors should be synergistic enough such that the marginal benefits of specializing for each behavior are worth the marginal costs of adding accelerator hardware.

For example, codes with vectorizable inner loops will tend to have a high degree of instruction-level parallelism. A behavior-specialized accelerator (BSA) which is targeted towards vectorizability (many parallel function units with consolidated control) will look quite different from one targeted

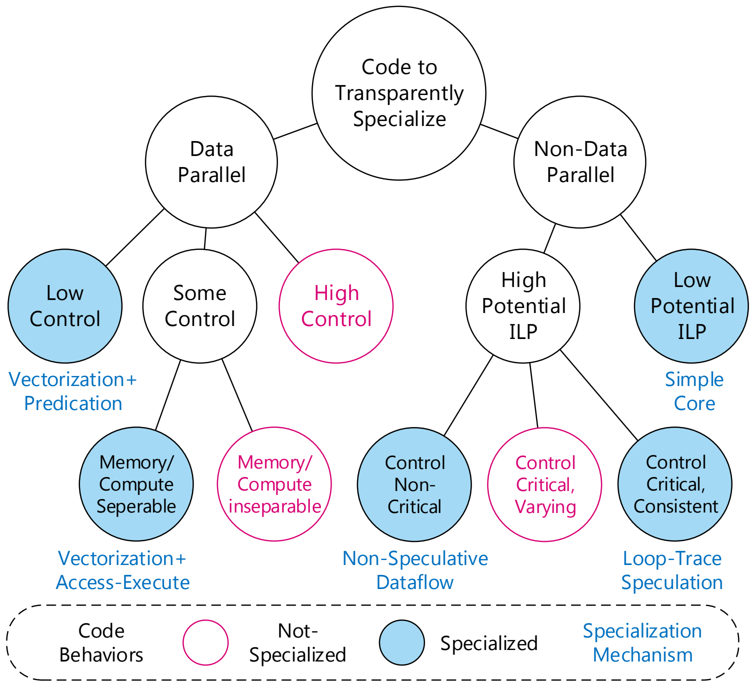


Figure 5.1: This work's behavior-space. (Note that non-specializable in this context means that we do not attempt specialization – not that it is known to be impossible to specialize for)

purely at high potential instruction-level parallelism (highly-parallel instruction dispatch, scheduling, and reorder logic). The correlation in behaviors, depending on the set of targeted workloads, might mean that it may not be useful to include both accelerators.

One way to help reason about program behaviors and their synergy in the abstract is to visualize them as a hierarchical *behavior space*. This is a tree-based classification of program behaviors where a path from root to leaf specifies the combination of behaviors that could potentially be specialized with custom hardware[1].

This Work's Behavior Space

Figure 5.1 shows the behavior space for our work. We explain the rational for this space as follows.

[1]We note that for any given set of specialized behaviors, there could be multiple "arrangements" of the tree. Here we attempt to make this classification intuitive by placing the behavior-distinctions with the most impact near the root of the tree.

At the top of the hierarchy, we separate codes into data-parallel (ultimately vectorizable) and non-data-parallel codes, as this will highly influence the specializable behaviors. The first behavior we specialize for are data-parallel codes with little to no control flow decisions. These represent a large and important class of workload with many well-known and effective hardware specialization techniques.

When the degree of control flow decisions becomes high, we look for another common property of data-parallel codes to exploit: separability of memory and computation instructions. This means that the addresses for the memory accesses are not dependent on the computations. The principles of decoupled access-execute can be used to specialize codes with this property.

Turning to non-data parallel codes, the first important distinction is whether there is any instruction-level parallelism to exploit. If not, we can consider specializing those codes with any sort of simple processor. If there is ILP, we can exploit other behaviors.

As discussed extensively in Chapter 4, we can exploit the prevalence of non-critical control flow, using mechanisms of non-speculative dataflow. Alternatively, another behavior we can exploit is the degree of control-flow bias (how consistent control flow decisions tend to be). If most of the control-flow decisions are consistent, then particular control-traces become common, and we can use specialization techniques that rely on executing long repeated traces.

To be clear, the behavior space we target is by no means exhaustive; there are many types of workloads that we would not specialize for, either because we use strict hardware specialization mechanisms or because the codes do not exhibit any targeted behaviors. Non targeted behaviors are indicated on our behavior-space diagram as red circles. The fact that we do not exhaustively search all known behaviors does not take away from our central goal in this chapter, which is to show that there is potential value in multi-behavior specialization. Furthermore, the unexplored behaviors can be the focus of future work in this area.

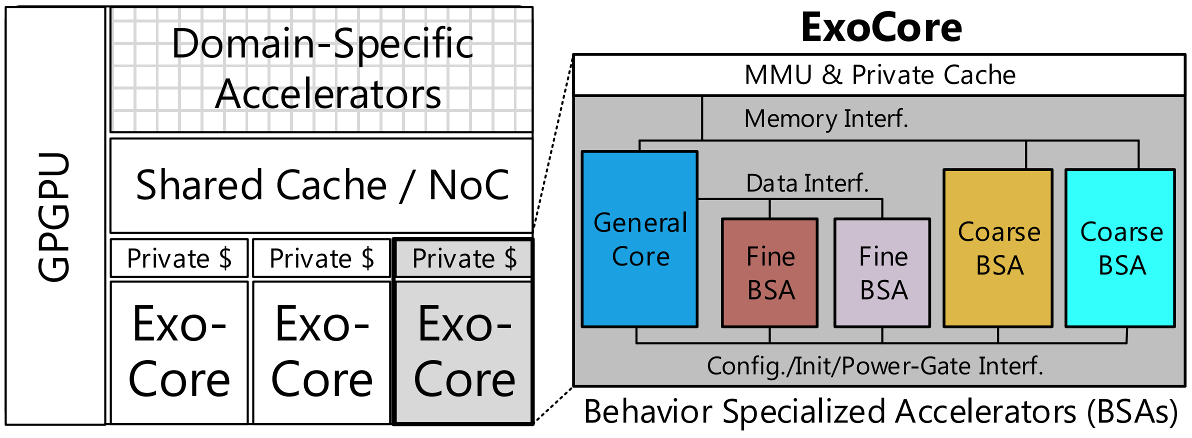


Figure 5.2: ExoCore-Enabled Heterogeneous System

5.2 Designing an ExoCore

As described earlier, Figure 5.2 shows how an ExoCore organization integrates a general purpose core with several other programmable or configurable BSAs targeted at different kinds of program behaviors. At run-time, depending on the affinity of the code, the execution migrates to a BSA which is most efficient, a process which is transparent to the programmer. An effective ExoCore design must incorporate accelerators which are *synergistic*, meaning they can effectively work together by specializing codes with different types of behaviors.

In this section, we first discuss how to use the behavior space we outlined in the previous section to create a composition of synergistic BSAs for an ExoCore. Then we give background on the individual BSAs that comprise it, and finally describe how they are modeled using the TDG.

Composing Synergistic Accelerators

The strategy we take in constructing a composition of BSAs is to draw from the existing accelerator literature, and select designs that specialize for the behaviors in the space we outlined earlier in Figure 5.1. Table 5.1 summarizes how different behaviors are exploited by the accelerators we incorporate, and gives insights into their benefits and drawbacks versus a general purpose core. We describe this composition of BSAs below; their high-level architecture is in Figure 5.3.

BSA **(Acronym)**	Exploited App. Behavior	Benefits vs General Core	Drawbacks vs General Core	Granularity	Inspired By
Short-Vector **SIMD**	Data-parallel loops with little control	Fewer instructions and less port contention	Masking/predicated inst penalty	Inner Loops	
Data Parallel CGRA **(DySER)**	Parallel loops w/ separable compute/memory	Vectorization + fewer insts. on general core	Extra comm. insts, predicated inst penalty	Inner Loops	DySER [32], Morphosys [87]
Non-speculative Dataflow **(SEED)**	Regions with non-critical control	Cheap issue width, larger instruction window	Lacks control speculation, requires instr. locality	Nested Loops	SEED [17], WaveScalar [73]
Trace-Speculative Proc. **(Trace-P)**	Loops w/ consistent control (hot traces)	Similar to above, but larger compound insts.	Trace mispeculation requires re-execution	Inner Loop Traces	BERET [25], CCA [26]

Table 5.1: Tradeoffs of Behavior Specialized Accelerators (BSAs) in this Work

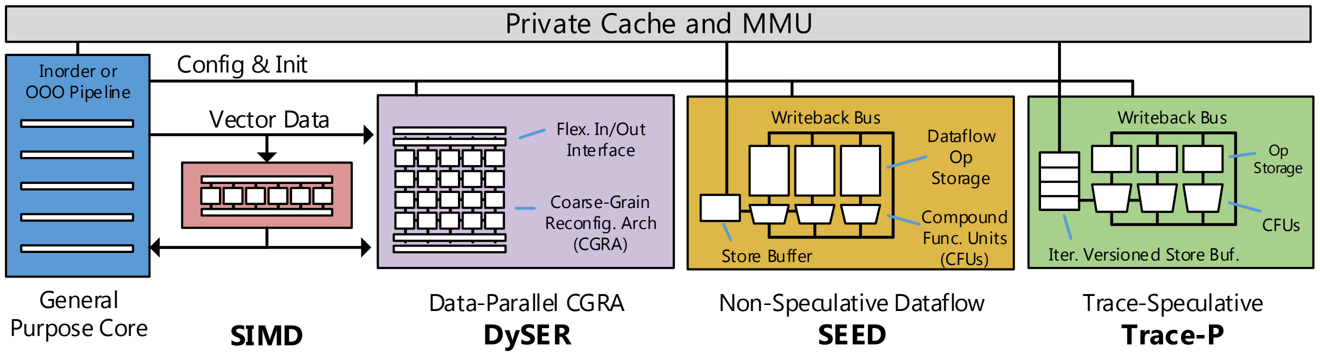


Figure 5.3: Example ExoCore Architecture Organization

SIMD For data-parallel regions with a limited amount of control and memory irregularity, we rely on Short vector **SIMD** extensions to a general purpose core's pipeline. We choose this because short vector extensions are very common, and are effective given the behaviors.

Data-Parallel Access-Execute Past a certain degree of control, SIMD is no longer effective, and instead we target the behavior of separable computation and memory.

The specialization technique is to offload the computation component of a loop to a CGRA which natively supports control flow, and pipeline the CGRA for executing parallel loops (typified by DySER [32], Morphosys [87]). The benefit of this is that there would be fewer total core instructions (which are expensive in terms of energy), and a higher degree of ILP can be exploited at lower power – all while tolerating some control flow inside the computation itself. Also, these accelerators can be beneficial for code with some memory irregularity, as they provide a shuffle network in their flexible input/output interfaces.

For this purpose, we use the DySER architecture proposed in work co-written and developed by this dissertation's author. Its design point has 64 functional units (FUs), and is configured similar to previous proposals [32].

Non-speculative Dataflow In code regions that are not data-parallel, but still have high potential ILP, non-speculative dataflow processors can be highly effective, especially when the control flow does not lie on the critical path.

For this purpose, we re-use the SEED [17] architecture, using distributed dataflow units communicating over a bus, and compound FUs (CFUs) for computation. (See Chapter 4 for a detailed description). This design targets inlined nested loops with 256 static compound instructions.

Trace-Speculative Core Often, control is on the critical path, meaning speculation is necessary for good performance, but it is *highly* biased – creating one hot path through a loop. Architectures like BERET [25] exploit this by sequencing through a speculative trace of instructions, using CFUs for energy efficiency. Instructions diverging from the hot loop trace require re-execution on the general core, making this accelerator strictly specialized for loops with hot traces.

We model a trace-speculative BSA similar to BERET, except that we add dataflow execution. This enables the design to be more competitive with an OOO core. We add a loop-iteration versioned store buffer to save speculative iterations. We refer to this design as a **Trace-P** for trace-processor[2]. Compared to SEED, Trace-P requires half as much operand storage, and can have larger CFUs, as compound instructions in Trace-P can cross control boundaries.

[2]Not to be confused with the Trace Processor from [88]

ExoCore Architecture Organization Putting the above together, Figure 5.3 shows how an ExoCore combines a general core with the four different BSAs. DySER and SIMD are integrated with vector datapaths from the general core. All designs besides SIMD are *configurable*, and require a configuration datapath. SEED and Trace-P have coarse grain integration, meaning they have their own interfaces to the cache hierarchy, and can power down parts of the core.

We emphasize that the detailed microarchitecture is not the emphasis or contribution of this chapter, rather the implications and opportunities of this design organization is the main contribution.

5.3 BSA Selection

One of the difficulties in using an ExoCore system with multiple accelerators is in the decision making processes for choosing between them, depending on the actual behaviors of program regions. While prior heterogeneous systems (mostly single-ISA works) have addressed the scheduling problem in their domain, the primary differences are 1. BSAs typically have restricted entry points (loop boundaries), and 2. BSAs targeting nested loops introduce a hierarchical scheduling problem (e.g. target an entire loop nest, or just the inner loop?).

This problem can be solved easily using the concept of what we call an Amdahl Tree, and a simple dynamic programming algorithm, as we explain next. As will become clear, the Amdahl Tree approach requires both information about region execution frequency and estimations for the expected benefit of an accelerator on a certain code region. Here, we first explain the concept of the Amdahl tree, then describe how to use it practically with limited profile information and performance estimates.

Definition An Amdahl Tree is first composed of a representation of the hierarchical region-structure of a non-recursive portion of the program. Nodes in the Amdahl Tree could represent arbitrary hierarchical regions, but typically will represent loops, functions, and traces to encode

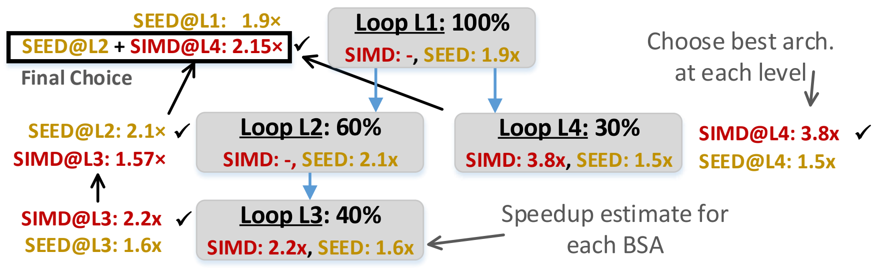


Figure 5.4: Amdahl Tree – Example of Triple Nested Loop

their nesting relationship. In addition to this structure, an Amdahl Tree's nodes are annotated with 1. That region's expected contribution to the execution time; and 2. The expected benefits of offloading each relevant accelerator given the entry point into the accelerated code is the beginning of that region.

Example Consider the example in Figure 5.4. Each node in the tree represents a candidate loop, nested loop or function call, and is labeled with the speedup of each BSA and the expected execution time. To explain the need for the following approach, looking at the innermost loops in the example (loop 3 and 4), the best option is SEED. However, by performing the bottom up traversal, using SEED at loop 2 and SIMD at loop 4 yields the best performance gain.

Algorithm The algorithm to solve this problem is straight forward. Perform a bottom-up traversal, and at each node consider two possibilities: 1. Use the best acceleration approach from each sub-granularity (each child node), and apply Amdahl's law to get the benefit for any individual accelerator. 2. Use the best acceleration approach at the current granularity (the current node). Note this can be extended to any metric (e.g. performance, energy), and would work as a solution for other specialization techniques that have different granularities (e.g. basic-blocks, traces, function-calls).

Practical Considerations In practice, employing the Amdahl tree requires profiling data and/or static analysis to estimate the region execution time and relative performance of each accelerator. We use a combination of both, along with simple heuristics for this task.

Suite	Benchmarks
TPT	conv, merge, nbody, radar, treesearch, vr
Parboil	cutcp, fft, kmeans, lbm, mm, needle, nnw, spmv, stencil, tpacf
SPECfp	433.milc 444.namd 450.soplex 453.povray 482.sphinx3
Mediabench	cjpeg, djpeg, gsmdecode, gsmencode cjpeg2, djpeg2, h263enc, h264dec, jpg2000dec, jpg2000enc, mpeg2dec, mpeg2enc
TPCH	Queries 1 and 2
SPECint	164.gzip, 181.mcf, 175.vpr, 197.parser, 256.bzip2 429.mcf, 403.gcc, 458.sjeng, 473.astar, 456.hmmer, 445.gobmk

Table 5.2: Benchmarks

5.4 ExoCore Exploration Methodology

The following methodology is used in the design-space exploration in the next section.

Benchmarks Selection Benchmarks were chosen from a wide range of suites (Table 5.2), and are meant to represent a wide range of workloads (they are a superset of the irregular workloads used in the previous chapters). These include highly regular codes from Intel TPT [32], scientific workloads from Parboil [44], image/video applications from Mediabench [80] and irregular workloads from SPECint. The diversity highlights ExoCore's ability to target a large variety of codes. Also, as much as possible, we picked benchmarks and suites from the respective accelerator publications.

General Core Configurations We considered four different cores of varying complexity, with parameters as outlined in Table 5.3. The common characteristics are a 2-way 32KB I$ and 64KB L1D$, both with 4 cycle latencies, and a 8-way 2MB L2$ with a 22 cycle hit latency. We model 256-bit SIMD.

Area Estimation We use McPAT for estimating area of general cores, and use area estimates from relevant publications [32, 25, 17].

Runtime Accelerator Selection Because we are exploring the potential of ExoCore systems, most of our results use an Oracle scheduler, which chooses the best accelerator for each static region, based

	IO2	OOO2	OOO4	OOO6
Fetch, Dispatch Issue, WB Width	2	2	4	6
ROB Size	-	64	168	192
Instr. Window	-	32	48	52
DCache Ports	1	1	2	3
FUs (ALU,Mul/Div,FP)	2,1,1	2,1,1,	3,2,2	4,2,3

Table 5.3: General Core Configurations

on past execution characteristics. The selection metric we use is energy-delay, where no individual region should reduce the performance by more than 10%. One later set of results compares the oracle and Amdahl Tree schedulers.

5.5 ExoCore Evaluation

In the evaluation of ExoCore, we seek to address three main questions: How much potential is there for an ExoCore System? Is having multiple accelerators useful within and across applications? Does an ExoCore system provide additional levels of flexibility in design choices?

We summarize the main results below. Note that a "full ExoCore" consists of an ExoCore with all four BSAs.

- **High Potential of BSAs** Across all workloads, a full OOO2-based ExoCore provides 2.0× performance and 1.6× energy benefits over an OOO2 core with SIMD (dual-issue, out-of-order). Compared to an OOO6 core with SIMD, an OOO6 ExoCore can achieve up to 1.4× performance and 1.7× energy benefits.

- **Broad Accelerator Affinity** ExoCores make use of multiple accelerators, both inside workload domains, and inside applications themselves. Across all benchmarks, considering a full OOO2 ExoCore, an average of only 20% of the original programs' execution cycles went un-accelerated.

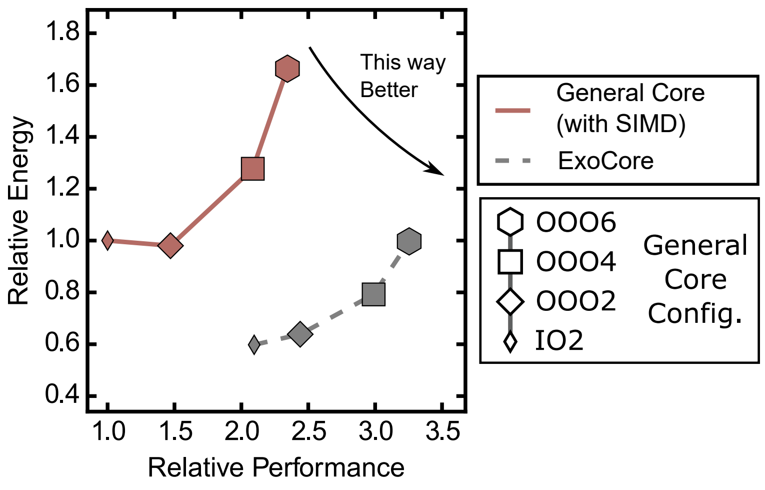


Figure 5.5: ExoCore Tradeoffs Across All Workloads

- **Increased Design Choice Opportunities** Our analysis suggests that there are several ways to break through current energy, area, and performance tradeoffs. For example, there are four OOO2-based ExoCores and nine OOO4-based ExoCores that achieve higher performance than an OOO6 with SIMD alone. Of these, the $\text{OOO2}_{\text{SIMD+DySER+SEED}}$ ExoCore achieves 2.6× better energy efficiency, while requiring 40% less area.

We discuss each of these results in the following subsections.

Overall Performance and Energy Benefits

Figure 5.5 shows geometric mean performance and energy benefits of the full ExoCore across all workloads. The solid line represents the general purpose core (with SIMD) only, while the dotted line represents the full ExoCore design. Each point on the curve represents a different general purpose core. The baseline is the dual-issue inorder core (IO2).

ExoCore designs improve all baseline cores significantly. A full OOO2-based ExoCore provides 2.0× performance and 1.6× energy benefits over an OOO2 core (dual-issue, out-of-order). Compared to an OOO6 core, an OOO6 ExoCore can achieve up to 1.4× performance and 1.7× energy benefits.

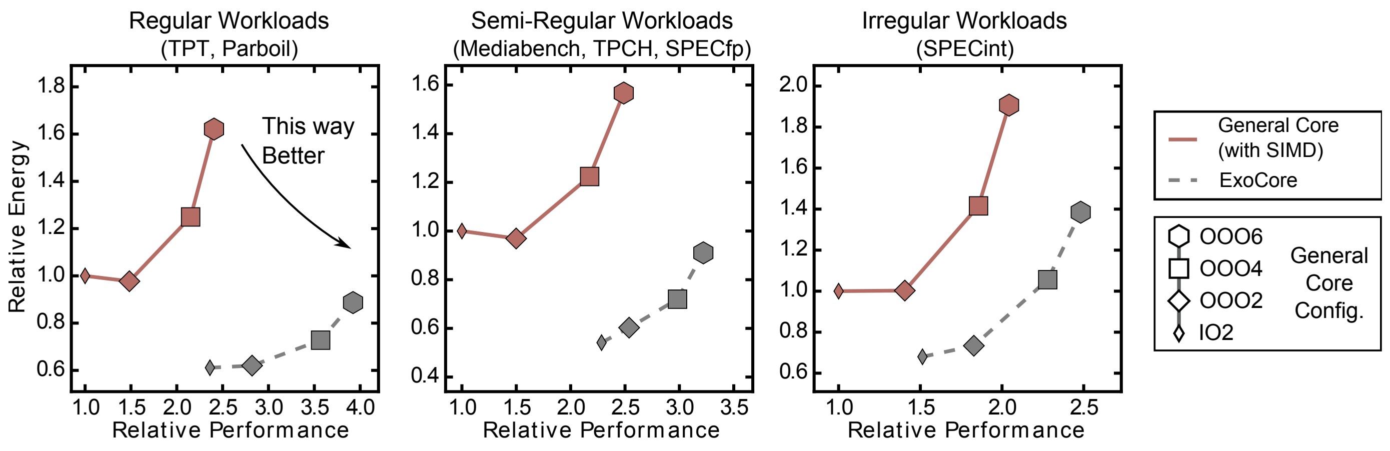


Figure 5.6: Interaction between Accelerator, General Core, and Workloads

Workload Interaction To observe workload-type specific behavior, Figure 5.6 divides the previous results into highly regular, semi-regular, and highly irregular workload categories. Categories are based on benchmark suites, listed in the figure above the graphs.

The major result here is that even on the most challenging irregular SPECint applications, ExoCores have significant potential. A full OOO2 ExoCore can achieve 1.3× performance and 1.4× energy benefits over OOO2 with SIMD. For OOO6, ExoCore achieves 1.2× performance and 1.4× energy efficiency improvement. Our findings also show that an ExoCore has a high potential on regular workloads, where ExoCore achieves 1.9× performance and 1.6× energy improvement for OOO2 core. This is surprising because SIMD itself works well on these workloads.

Inter-application BSA Utilization To understand where the benefits are coming from, we can break down performance and energy results for each application. Figure 5.7 considers an OOO2-based ExoCore, and shows each benchmark's energy and execution time, as spent in each accelerator. The baseline is the OOO2 alone with *no* SIMD. Most benchmarks, even irregular workloads, have some potential to effectively use BSAs, ranging from 100% of the program being offloaded, to around 20%. Secondly, more than half of the workloads benefit from multiple BSAs inside a single

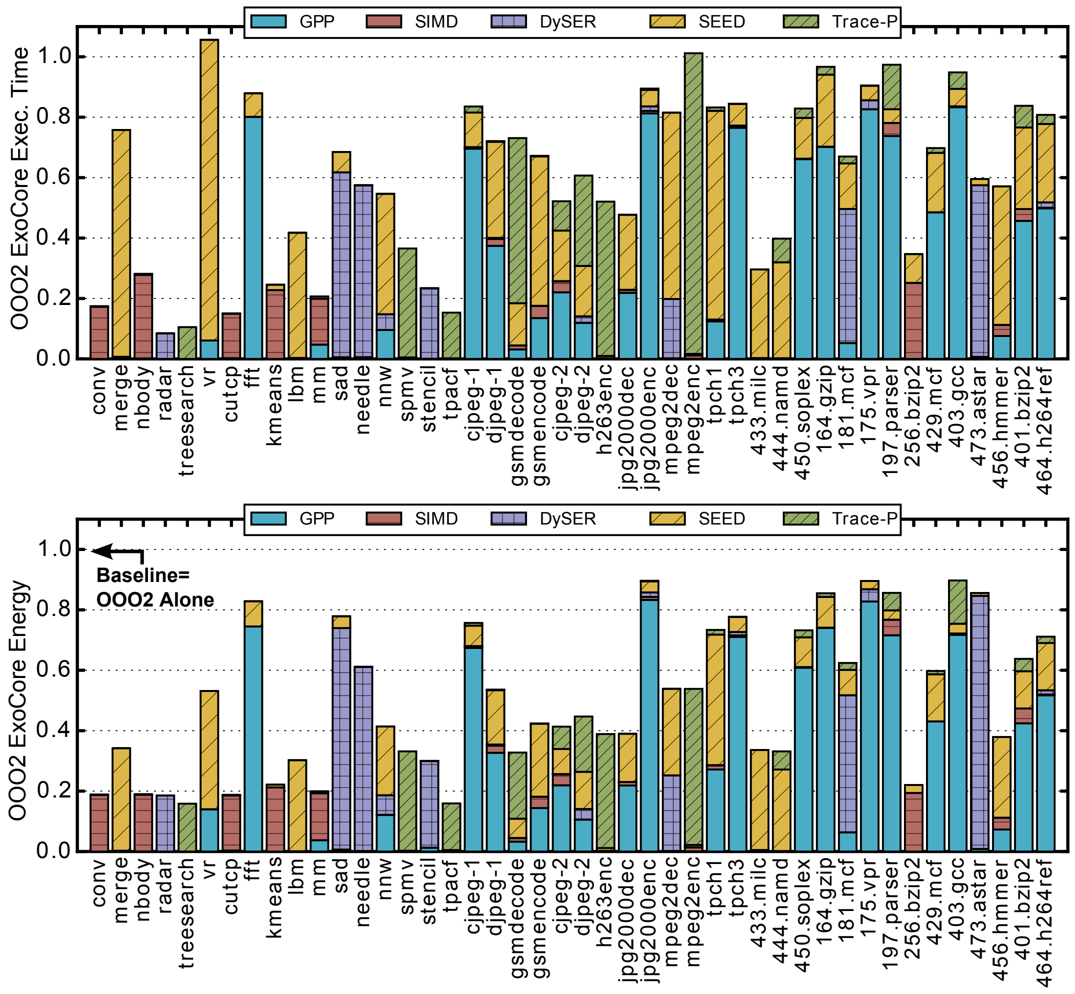


Figure 5.7: Per-Benchmark Behavior and BSA Execution Time and Energy Utilization

application[3]. For example, cjpeg-2 makes use of SIMD, SEED and Trace-P during its execution. Considering energy breakdowns in Figure 5.7, observe that for SIMD and DySER, the percentage of cycles offloaded is proportional to the energy contribution. For SEED, because it can power-gate portions of the general core when in use, the energy reduction is higher than the execution time reduction.

[3]This considers a 5% threshold in terms of instructions executed for an accelerator to be considered "used" for a given application.

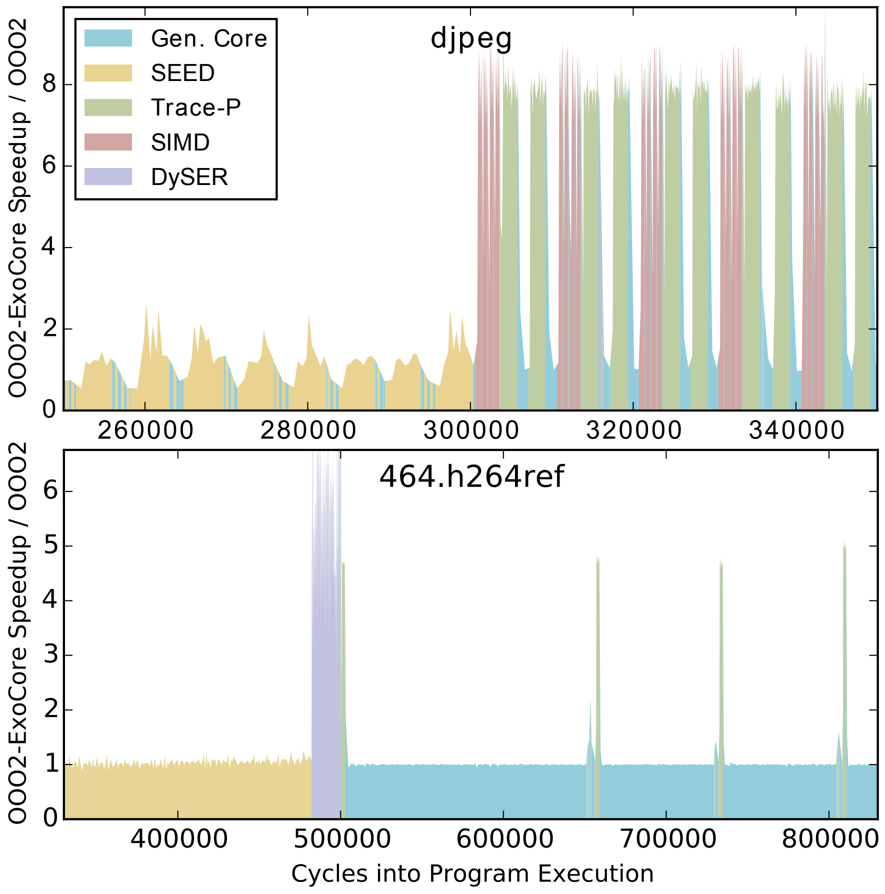


Figure 5.8: ExoCore's Dynamic Switching Behavior

Affinity Granularity To give insight into the dynamic behavior of an ExoCore, Figure 5.8 shows a trace of execution for two benchmarks, `djpeg` and `464.h264ref`. These graphs show the relative performance benefits of a full OOO2 ExoCore over the OOO2 core alone (no SIMD), over time. The performance is calculated by examining the cycle times of fixed-length (1000 instruction) windows of the original non-transformed TDG, hence some noise is introduced.

These graphs demonstrate that applications can have fine-grain affinity for different accelerators, while retaining large speedup benefits. SEED tends to give longer sustained benefits, while Trace-P, DySER, and SIMD give shorter but larger bursts of performance. These graphs also highlight the capabilities of the transformable dependence graph for capturing cycle level microarchitectural

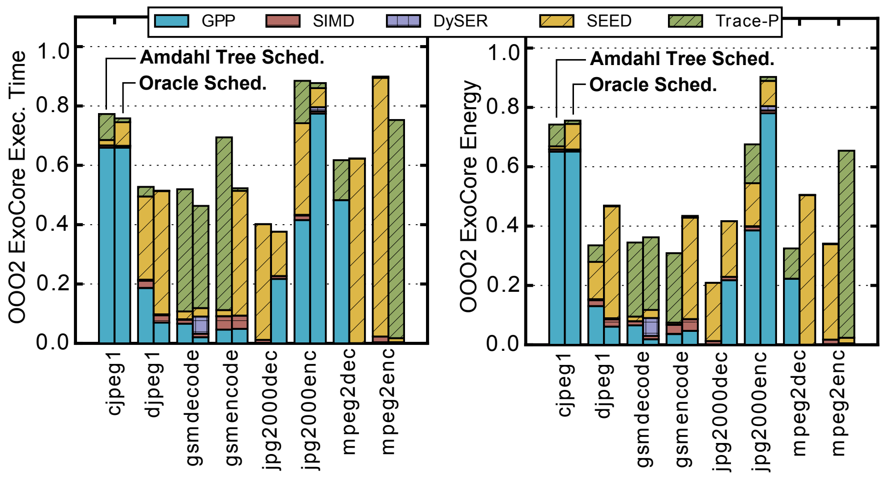


Figure 5.9: Oracle versus Amdahl Tree Scheduler

effects, explained in Chapter 3.

Practicality Though we have shown that ExoCores have significant potential, an important concern is whether they can practically provide benefits without oracle information. To study this, Figure 5.9 presents a comparison of the performance and energy of the Amdahl scheduler (left-bar) and the Oracle scheduler (right-bar). Here, the general core is the OOO2, and we are showing challenging benchmarks from Mediabench which require using multiple different accelerators in the same application to be effective.

Compared to the oracle scheduler, our scheduler is slightly over-calibrated towards using the BSAs rather than the general core, meaning it is biased towards energy efficiency – across all benchmarks (including those not shown) the scheduler provides 1.21× geomean energy efficiency improvement, while giving 0.89× performance of the Oracle scheduler. These heuristics can be tuned with more effort to favor different metrics, or to be useful for the OOO4 or OOO6 cores.

Design Space Exploration

An ExoCore organization opens design opportunities which can push traditional energy and performance tradeoffs forward, and this is true for both complex and simple general purpose cores.

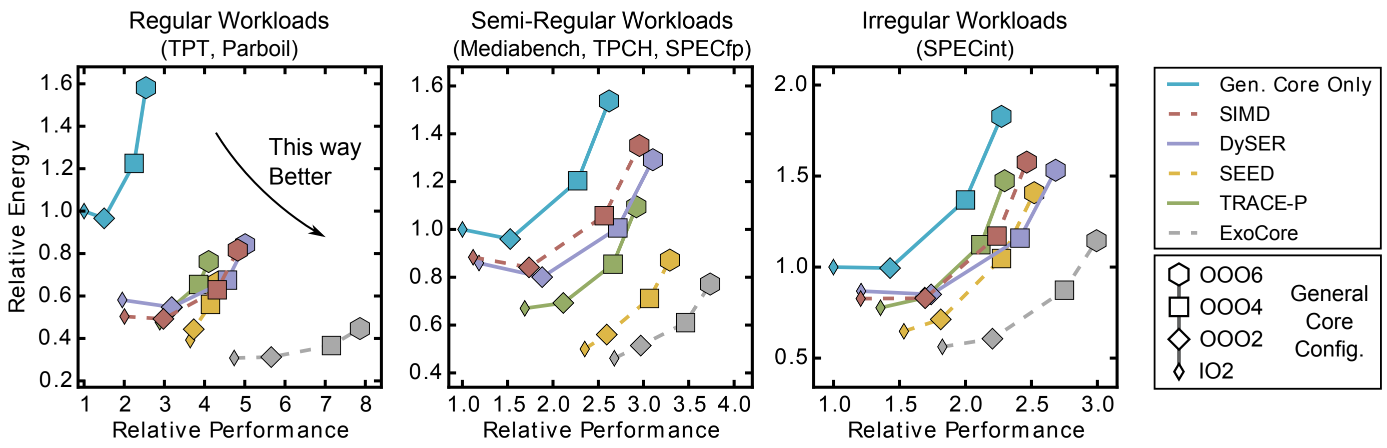


Figure 5.10: Interaction between Accelerator, General Core, and Workloads

To understand this flexibility, we explore a design space comprising the combinations of four general purpose cores and 16 choices for the subset of BSAs (from the set of four BSAs we study).

Single-BSA Designs and Workload Interactions We begin by considering Single-BSA designs and their contribution to different workload domains. Figure 5.10 shows similar results to Figure 5.6, but the single-BSA design points are included, as well as the general purpose core with no SIMD. Again, each line in this graph represents the set of designs with the same combination of accelerators (or no accelerators), and each point on the curve represents a different general purpose core. Note that the baseline here is the general purpose core alone (no SIMD).

Overall what this shows is that though each BSA alone has significant potential, their combination has even more potential. On the regular workloads, each BSA gives roughly the same benefits, with the coarse grain-integrated accelerators giving slightly more energy reduction, while the data-parallel accelerators give slightly more performance. However, having all of them does improve the performance and energy by about twice as much.

On the semi-regular workloads, SEED is by far the most effective, taking advantage of the high ILP that the vector accelerator could not because of memory irregularity. Still, a full ExoCore does provide 10-20% performance and energy benefits over SEED alone.

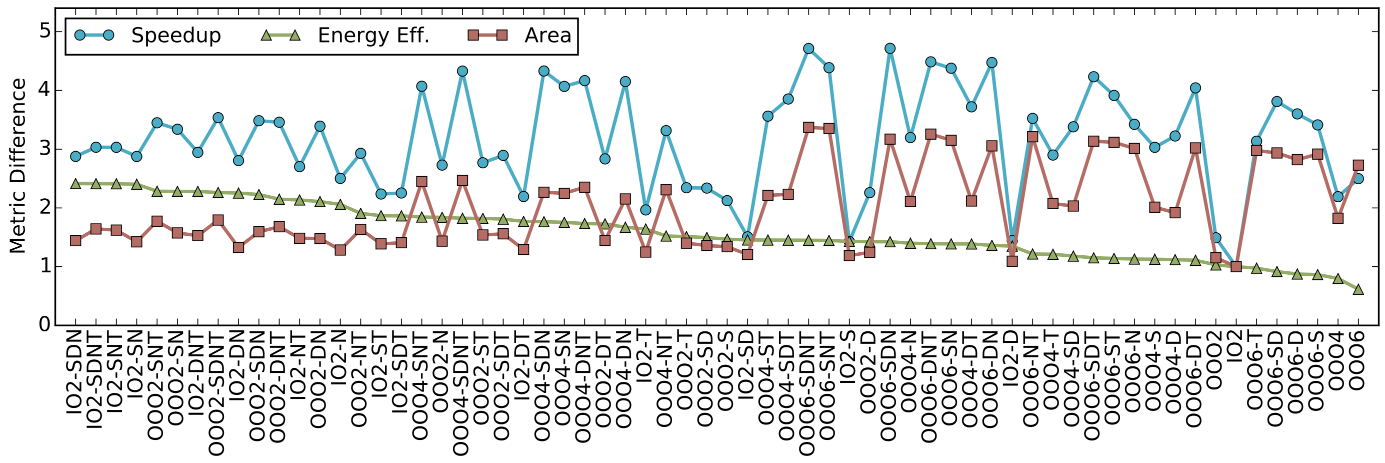


Figure 5.11: Design-Space Characterization. S: SIMD, D: Data-Parallel CGRA (DySER), N: Non-spec Dataflow (SEED), T: Trace-Processor

On the most irregular workloads (SPECint), the gap between single-BSA designs reduces, but both DySER and SEED lie on the Pareto optimal for a single-BSA. Here, A full ExoCore gives between 10-30% performance and energy benefits over the best single-BSA design.

The fact that the tradeoffs of the various metrics differ between single-BSAs designs opens the question of what tradeoffs become possible when different subsets of BSAs are allowed.

Full Design Space and Design Choice Opportunities To explore the potential design flexibility, we perform a design space exploration across all combinations of general purpose cores and BSAs across all workloads. The resulting performance, energy efficiency and area metrics are shown in Figure 5.11, where all points are relative to the dual-issue in-order (IO2) design. On the graph, higher is better for performance and energy efficiency, and lower is better for area. There are many valuable quantitative insights.

- **[Performance]** Compared to the OOO6 core with SIMD[4], four OOO2 and nine OOO4 ExoCore configurations match the performance with lower energy and area, by as much as 2×. This gives the designer freedom even if performance is a hard constraint.

[4]We use OOO6-SIMD as the baseline - since this resembles commercial processors with AVX2 etc. Note that if we had used OOO6 without SIMD as the baseline, the benefits of ExoCores would be even higher.

- **[Performance]** No in-order ExoCore configuration can match the OOO6 performance. The best achieved is 88% of the OOO6, with almost 1.7× lower area.
- **[Energy]** The OOO2 core with SIMD is the most energy efficient baseline core. Compared to this core, **twelve** in-order and **five** OOO4 ExoCores have higher energy efficiency - by as much as 65% and 25% respectively. In addition to being more energy efficient, these configurations are up to 1.35× (IO2-SDN) and 1.9× (OOO4-SNT) higher performance.
- **[Energy]** A full OOO6 ExoCore achieves the same energy efficiency (within 1%) of the OOO2-SIMD core and has 2.2× higher performance, but is nearly twice the area.
- **[Full ExoCores]** The full IO2 ExoCore is the most energy-efficient among all designs. The full OOO6 ExoCore has the best performance, the next best is the full OOO4 ExoCore, which has 10% lower performance, 1.25× lower energy and 1.36× lower area.

Overall these results suggest two trends. Behavior specialized cores are extremely effective at reducing energy regardless of core type. They provide high performance-improvements for in-order, small and medium-sized OOO cores - but not as much for big OOO cores. Therefore, the most high impact opportunity for future BSAs is to improve the performance of OOO cores, while further increasing their energy efficiency.

5.6 Related Work

A variety of heterogeneous cores have been explored in the past. We first discuss single-ISA heterogeneous systems, then multi-ISA.

Single-ISA Heterogeneity Single-ISA heterogeneous architectures have been extensively explored, starting with Kumar et al. [89]. Later work extended this by exploring a large design space of microarchitectural design points [90], and Lee et al. use regression analysis to broaden this design space even further. A related design approach is to use a shared pipeline frontend, but use a heterogeneous backend inside the microarchitecture, like Composite Cores [30].

Relatedly, many past works have explored processor steering methods on heterogeneous architectures [91, 92, 93, 94, 95]. The unique aspect of the scheduling problem in this work is that entry points to different accelerators are restricted by the program's loop structure.

An interesting single-ISA architecture is XLOOPs [96], which is a recent design targeting inner loops with specific loop-dependence patterns. It uses a small number of additional instructions to communicate loop patterns, which can be elided if run on the general purpose processor. Though its high-level adaptive specialization paradigm is similar to the strategies we present here, along with some targeted code properties, the microarchitecture is vastly different, and will favor different codes. One benefit of the architectures we target here is their ability to target coarser grain regions, enabling more effective power-gating of the OOO core.

Multi-ISA Heterogeneity To a lesser extent, multi-ISA heterogeneity (where all cores are general purpose) have been previously studied, including Venkat et al. [97], who show that fully general purpose cores with heterogeneous-ISAs alone can provide benefits. Our work considers the composition of accelerator-type architectures, which offer trade-offs beyond those of general purpose architectures.

A recent body of work by Chien et al., under the moniker 10x10 [98], has explored using multiple "micro-engines" as offload engines inside general purpose cores. Though the goal of improving general cores is shared, their approach has generally been in designing algorithm-class specific offload units, rather than behavior-specific. For example, they have developed micro-engines for pattern-matching [99], fast Fourier transform [100], bit-manipulation [101], finite automata [102], and data layout transformation [103]. They have also studied the workload clustering patterns of instruction data types (floating point, integer, etc) and operation types. This approach could be useful in a behavior-oriented context to find clusters of behaviors that frequently occur together. Overall, there could be significant opportunities to combine behavior-specific and algorithm-specific workloads in the future.

5.7 Summary

In this chapter, we proposed the ExoCore design, which incorporates multiple behavior-specialized accelerators inside a core. It is an appealing design, as it can simplify general purpose core design – behavior-specific microarchitecture blocks can be designed and integrated into the core in a modular fashion, without disruptive changes to the core's microarchitecture. It also provides a promising approach for exceeding the performance/energy frontier of conventional monolithic general purpose processors. A design space exploration further revealed that an ExoCore organization provides new design choice opportunities and flexibility. As an example, a 2-wide OOO processor with three BSAs matches the performance of a conventional 6-wide OOO core with SIMD, has 40% lower area and is more than $2\times$ energy efficient.

6 GENERAL MATHEMATICAL ACCELERATOR SCHEDULING

One of the primary advantages of behavior specialized accelerators is in how they expose elements of the hardware execution up through the ISA, enabling them to exploit program behaviors with more efficient hardware mechanisms. Of course, exposing elements of the hardware execution puts a larger burden on the compiler, which must now map computation on to these complex substrates. In fact, this is a problem not just for behavior specialized accelerators, but for a larger class of exposed hardware architectures that are termed *spatial* architectures. Generally, these problems (termed spatial scheduling) have been solved with architecture-specific heuristics, an approach which suffers from poor compiler/architect productivity, lack of insight on optimality, and inhibits migration of techniques between architectures.

The goal of this chapter is to develop a scheduling framework usable across behavior specialized accelerators. To this end, we express spatial scheduling as a constraint satisfaction problem using Integer Linear Programming (ILP)[1]. We observe that architecture primitives and scheduler responsibilities can be related through five abstractions: placement of computation, routing of data, managing event timing, managing resource utilization, and forming the optimization objectives. We encode these responsibilities as 20 general ILP constraints, and for each BSA we augment with a few additional architecture-specific constraints. Since these architectures do not necessarily have existing heuristic-based schedulers to compare against, we also examine some existing whole-program spatial architectures, and use our framework to build schedulers for them as well – these are the architectures which we evaluate in the conclusion. In all, we create schedulers for the BSAs we target (DySER, SEED, Trace-P), as well as the TRIPS and PLUG architectures. Our results show that a general declarative approach using ILP is implementable, practical, and typically matches or outperforms specialized schedulers.

In this chapter, we first present background on two additional spatial architectures outside

[1]For an introduction to integer linear programming and its applications to computer architecture, a work co-written by the author gives an overview, as well as some detailed examples [104].

the domain of behavior specialization (whole-program spatial architectures). We then present a brief overview of our approach (Section 6.2), and give our detailed general ILP formulation (Section 6.3). The subsequent section describes how to extend the general formulation to meet the architecture-specific constraints for the four spatial architectures we target (Section 6.4). We then present evaluation of the architectures that had previous spatial schedulers (Section 6.5). Finally, we describe the limitations of our modeling abstractions and the flexibility and generality of the approach (Section 6.6), cover the long history of related work in this area (Section 6.7) and summarize (Section 6.8).

6.1 Spatial Architectures and ILP Primer

Spatial Architectures

We use the term *spatial architecture* to refer to an architecture in which some subset of the hardware resources, namely functional units, interconnection network, or storage, are exposed to the compiler, whose job, as part of the scheduling phase, is to map computation and communication primitives in the instruction set architecture to these hardware resources. Behavior specialized architectures are often-times spatial, as it benefits hardware efficiency and gives the compiler opportunities to better exploit certain program behaviors. The accelerators that we use in this work, SEED, DySER, and Trace-P all fall into this category, as well as others like CCA, SoftHV, Veal, and many others. But of course, there are a large variety of whole-program spatial architectures as well, like VLIW architectures, dataflow machines like TRIPS and WaveScalar, and tiled architectures like RAW and PLUG.

DySER, SEED, and Trace-P have been described earlier (Chapters 4 and 5). Below, we give background on the two whole-program spatial architectures that we use in the evaluation, TRIPS and PLUG. A detailed diagram of the set of targeted architectures, along with their scheduling abstraction, is in Figure 6.8 (page 106).

The **TRIPS architecture** is whole-program general purpose dataflow machine. It is organized into 16 tiles, with each tile containing 64 slots, with these slots grouped into sets of eight. The slots from one group are available for mapping one block of code, with different groups used for concurrently executing blocks. The tiles are interconnected using a 2-D mesh network, which implements dimension-ordered routing and provides support for flow-control and network contention. The scheduler must perform computation mapping: it takes a block of instructions (which can be no more than 128 instructions long) and assigns each instruction to one of the 16 tiles and within them, to one of the 8 slots.

The **PLUG architecture** is designed to work as an accelerator for data-structure lookups in network processing. Each PLUG tile consists of a set of SRAM banks, a set of no-buffering routers, and an array of statically scheduled in-order cores. The only memory access allowed by a core is to its local SRAM, which makes all delays statically determinable. Applications are expressed as dataflow graphs with code-snippets (the PLUG literature refers to them as code-blocks) and memory associated with each node of the graph. Execution of programs is data-flow driven by messages sent from tile to tile - the ISA provides a *send* instruction. Using a principle of fixed delays in the execution model, the architecture is contention-free and completely statically scheduled. The scheduler must perform computation mapping and network mapping (dataflow edges $\rightarrow$ networks). It must ensure there is no contention for any network link, which it can do by scheduling when *send* instructions execute in a code-snippet or adjusting the mapping of graph nodes to tiles. It must also handle flow-control.

In all three architectures, multiple instances of a block, region, or dataflow graph are executing concurrently on the same hardware, resulting in additional contention and flow-control.

6.2 Overview

We present below the main insights of our approach in using constraint-solving for specifying the scheduling problem for spatial architectures. We distill the formulation into five responsibilities, each corresponding to one architectural primitive of the hardware. For a more general discussion of limitations and concerns related to our approach, see Section 6.6.

The scheduler for a spatial architecture works at the granularity of "blocks" of code, which could be basic-blocks, hyper-blocks, code-regions, or other more sophisticated partitions of program code. These blocks, which we represent as directed acyclic graphs (DAGs) consist of computation instructions, control-flow instructions, and memory access instructions that must be mapped to the hardware. We formulate the scheduling problem as spatially mapping a typed computation DAG G to a hardware graph H under certain constraints as shown by Figure 6.1 on page 96. For ease of explanation, we describe G as comprised of vertices and edges, while H is comprised of nodes, routers and links (formal definitions and details follow in Section 6.3).

#	**Architecture feature**	**Scheduler Responsibility**	**DySER**	**SEED**	**TRIPS**	**PLUG**
1	Compute hardware organization	Placement of computation	Heterogeneous compute units	Heterogeneous compound compute units	Homogeneous compute units	Homogeneous compute units
2	Network hardware organization	Routing of data	2D grid, unconstrained routing	Multi-bus network	2D grid, dimension order routing	2D multi-network grid, dim.-order routing
3	Timing and synchroni-zation	Manage timing of events	Data-flow firing & conflict-free network, flow control	Compound FU & Data-flow firing	Data-flow firing & dynamic network arb.	Hybrid dataflow/ in-order & static compute/net. timing
4	Concurrency within block	Manage resource utilization	None – dedicated compute units, switches, links	Parallel FUs inside CFU	8-slots per compute unit, reg-tile, data-tile	32 slots per compute-unit and multicast communication
	Concurrency across blocks		Pipelined execution	32 compound insts per CFU, Pipelined	Concurrent execution of different blocks	Pipelined execution across tiles
5	Performance Goal	MILP objective formulation	Latency & Lat. Mismatch	Latency	Throughput and Latency	Latency

Table 6.1: Relationship between architectural primitives and scheduler responsibilities.

To design and implement a general scheduler applicable to many spatial architectures, we observe that five fundamental architectural primitives, each with a corresponding scheduler responsibility, capture the problem as outlined in Table 6.1 (columns 2 and 3). Implementing these responsibilities mathematically is a matter of constraint and objective formulas involving integer variables, which form an ILP model, covered in depth in Section 6.3. Below we describe the insight connecting the primitives and responsibilities and highlight the mathematical approach. Table 6.1 summarizes this correspondence (in columns 2 and 3), and describes these primitives for three different architectures.

Computation HW organization $\rightarrow$ Placement of computation: The spatial organization of the computational resources, which could have a homogeneous or heterogeneous mix of computational units, requires the scheduler to provide an assignment of individual operations to hardware locations. As part of this responsibility, vertices in G are mapped to nodes in the H graph.

Network HW organization $\rightarrow$ Routing of data: The capabilities and organization of the network dictate how the scheduler must handle the mapping of communication between operations to the hardware substrate, i.e. the scheduler must create a mapping from edges in G to the links represented in H. As shown in the 2nd row of Table 6.1, the network organization consists of the spatial layout of the network, the number of networks, and the network routing algorithm. The flow of data required by the computation block and the placement of operations defines the required communication. Depending on the architecture, the scheduler may have to select a network for each message, or even select the exact path it takes.

Hardware timing/synchronization $\rightarrow$ Manage timing of events: The scheduler must take into consideration the timing properties of computation and network together with architectural restrictions, as shown in the 3rd row of table 6.1. In some architectures, the scheduler cannot determine the exact timing of events because it is affected by dynamic factors (e.g. memory latency through the caching hierarchy). For all architectures, the scheduler must have at least a partial view of timing of individual operations and individual messages to be able to minimize the latency of the computation block. In some architectures, the scheduler must exert extensive fine-grained control over timing to achieve static synchronization of certain events.

Concurrent hardware resource usage $\rightarrow$ Managing Utilization: Central to the difficulties of the scheduling problem is the concurrent usage of hardware resources by multiple vertices/edges in G of one node/link in H. We formalize this concurrent usage with a notion of *utilization*, which represents the amount of work a single hardware resource performs. Such concurrent usage (and hence > 1 *utilization*) can occur *within* a DAG and across concurrently executing DAGs. Overall, the scheduler must be aware of resource limits in H and which resources can be shared as shown in Table 6.1 row 4. For example, in TRIPS, within a single DAG, 8 instruction-slots share a single ALU (node in H), and across concurrent DAGs, 64 slots share a single ALU in TRIPS. In both cases, this node-sharing leads to contention on the links as well.

Performance goal $\rightarrow$ Formulate ILP objective: The scheduler generally has control over multiple quantities which can improve the performance. This often means deciding between the conflicting goals of minimizing the latency of individual blocks and managing the utilization among the available hardware resources to avoid creating bottlenecks, which it manages by prioritizing optimization quantities.

6.3 General ILP framework

This section presents our general ILP formulation in detail. Our formal notation closely follows our ILP formulation in GAMS instead of the more conventional notation often used for graphs in literature. We represent the computation graph as a set of vertices V, and a set of edges E. The computation DAG, represented by the adjacency matrix $G(V \cup E, V \cup E)$, explicitly represents edges as the connections between vertices. For example, for some $v \in V$ and $e \in E$, $G(v, e) = 1$ means that edge e is an output edge from vertex v. Likewise, $G(e, v) = 1$ signifies that e is an input to vertex v. For convenience, lowercase letters represents elements of the corresponding uppercase letters' set.

We similarly represent the hardware graph as a set of hardware computational resource nodes N, a set of routers R which serve as intermediate points in the routing network, and a set of L

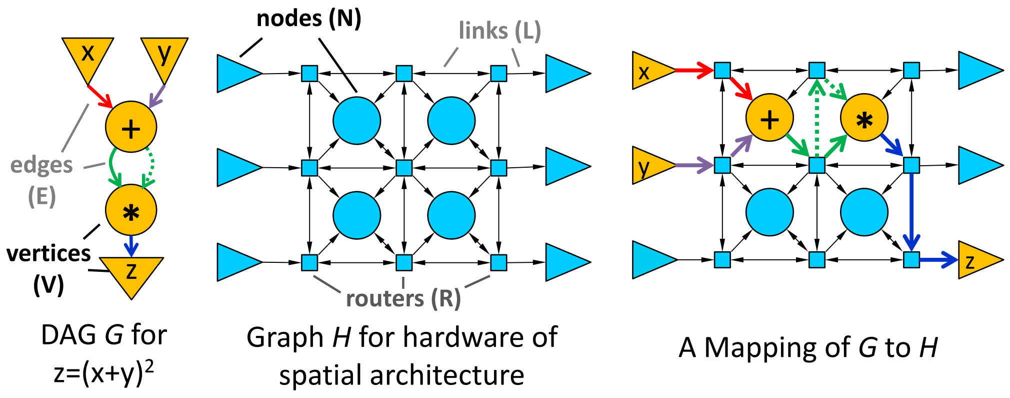


Figure 6.1: Example of computation G mapped to hardware H.

unidirectional links which connect the routers and resource nodes. The graph which describes the network organization is given by the adjacency matrix $H(N \cup R \cup L, N \cup R \cup L)$. To clarify, for some $l \in L$ and $n \in N$, if the parameter $H(l,n)$ was 0, link l would not be an input of node n. Hardware graphs are allowed to take any shape, and typically do contain cycles. Terms vertex/edge refer to members in G, and node/link to members in H.

Some of the vertices and nodes represent not only computation, but also inputs and outputs. To accommodate this, vertices and nodes are "typed" by the operations they can perform, which also enables the support of general heterogeneity in the architecture. For the treatment here, we abstract the details of the "types" into a compatibility matrix $C(V,N)$, indicating whether a particular vertex is compatible with a particular node. When equations depend on specific types of vertices, we will refer this set as V_{type}.

Figure 6.1 shows an example G graph, representing the computation $z = (x+y)^2$, and an H graph corresponding to a simplified version of the DySER architecture. Here, triangles represent input/output nodes and vertices, and circles represent computation nodes and vertices. Squares represent elements of R, which are routers composing the communication network. Elements of E are shown as unidirectional arrows in the computation DAG, and elements of L as bidirectional arrows in H representing two unidirectional links in either direction.

Inputs: Computation DAG Description (G)	
V	Set of computation vertices.
E	Set of Edges representing data flow of vertices
$G(V{\cup}E, V{\cup}E)$	The computation DAG
$\Delta(E)$	Delay between vertex activation and edge activation.
$\Delta(V)$	Duration of vertex.
$C(V,V)$ (SEED)	Describes whether vertices could be grouped in a CFU.
$\Gamma(E)$ (PLUG)	Delay between vertex activation and edge reception.
B_e	Set of bundles which can be overlapped in network.
B_v (PLUG)	Set of mutually exclusive vertex bundles.
$B(E{\cup}V, B_e{\cup}B_v)$	Parameter for edge/vertex bundle membership.
P (TRIPS)	Set of control flow paths the computation can take
$A_v(P,V)$, $A_e(P,E)$ (TRIPS)	Defining which vertices and edges get *activated* by given path
Inputs: Hardware Graph Description (H)	
N	Set of hardware resource Nodes.
R	Routers which form the network
L	Set of unidirectional point-to-point hardware Links
$H(N{\cup}R{\cup}L, N{\cup}R{\cup}L)$	Directed graph describing the Hardware
$I(L,L)$	Link pairs incompatible with Dim. Order Routing.
Inputs: Relationship between G/H	
$C(V,N)$	Vertex-Node Compatibility Matrix
MAX_N, MAX_L	Maximum degree of mapping for nodes and links.
Variables: Final Outputs	
$M_{vn}(V,N)$	Mapping of computation vertices to hardware nodes.
$M_{el}(E,L)$	Mapping of edges to paths of hardware links
$M_{bl}(B_e,L)$	Mapping of edge bundles to links
$\beta(V,V)$ (SEED)	Binary var. indicating CFUs mapped to one instance
$M_{bn}(B_v,N)$ (PLUG)	Mapping of vertex bundles to nodes
$\delta(E)$ (PLUG)	Padding cycles before message sent.
$\gamma(E)$ (PLUG)	Padding cycles before message received.
Variables: Intermediates	
$O(L)$	The order a link is traversed in.
$U(L{\cup}N)$	Utilization of links and nodes.
$U_p(P)$ (TRIPS)	Max Utilization for each path P.
$T(V)$	Time when a vertex is activated
$X(E)$	Extra cycles message is buffered.
$\lambda(b,e)$ (PLUG)	Cycle when e is activated for bundle b
LAT	Total latency for scheduled computation
SVC	Service interval for computation.
MIS	Largest Latency Mismatch.

Table 6.2: Summary of formal notation used.

The scheduler's job is to use the description of the typed computation DAG and hardware graph to find a mapping from computation vertices to computation resource nodes and determine the hardware paths along which individual edges flow. Figure 6.1 also shows a correct mapping of the computation graph to the hardware graph. This mapping is defined by a series of constraints and variables described in the remainder of this Section, and these variables and scheduler inputs are summarized in Table 6.2.

We now describe the ILP constraints which pertain to each scheduler responsibility, then show a diagram capturing this responsibility pictorially for our running example in Figure 6.1.

Responsibility 1: Placement of computation.

The first responsibility of the scheduler is to map vertices from the computation DAG to nodes from the hardware graph. Formally, the scheduler must compute a mapping from V to N, which we represent with the matrix of binary variables $M_{vn}(V, N)$. If $M_{vn}(v, n) = 1$, then vertex v is mapped to node n, while $M_{vn}(v, n) = 0$ means that v is not mapped to n. Each vertex $v \in V$ must be mapped to exactly one compatible hardware node $n \in N$ in accordance with $C(v, n)$. The mapping for incompatible nodes must also be disallowed. This gives us:

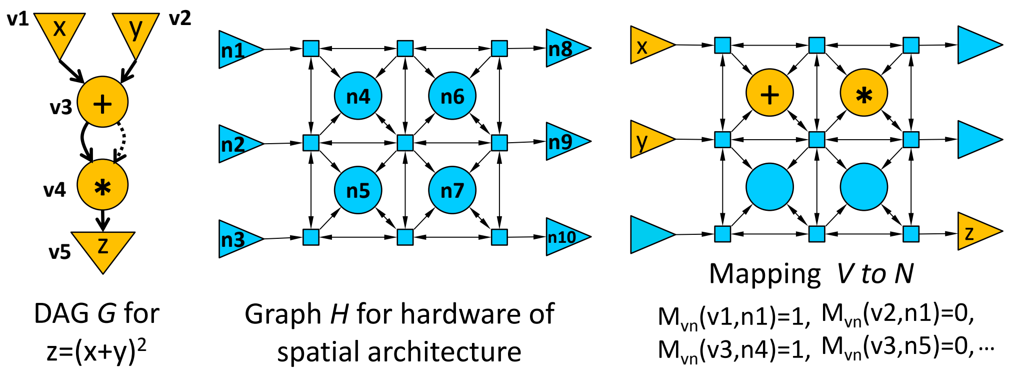
$$\forall v \;\; \Sigma_{n|C(v,n)=1} M_{vn}(v, n) = 1 \tag{6.1}$$

$$\forall v, n|C(v, n) = 0, \;\; M_{vn}(v, n) = 0 \tag{6.2}$$

An example mapping with corresponding assignments to M_{vn} is shown in Figure 6.2.

Responsibility 2: Routing of data

The second responsibility of the scheduler is to map the required flow of data to the communication paths in the hardware. We use a matrix of binary variables $M_{el}(E, L)$ to encode the mapping of edges to links. Each edge e must be mapped to a sequence of one or more links l. This sequence must start from and end at the correct hardware nodes. We constrain the mappings such that if a



v1
v2
x
y
v3
+
v4
*
v5
z
DAG G for z=(x+y)²
n1
n2
n3
n4
n5
n6
n7
n8
n9
n10
Graph H for hardware of spatial architecture
Mapping V to N
Mvn(v1,n1)=1, Mvn(v2,n1)=0,
Mvn(v3,n4)=1, Mvn(v3,n5)=0,...

Figure 6.2: Placement of computation

vertex v is mapped to a node n, every edge e leaving from v must be mapped to one link leaving from n. Similarly, every edge arriving to v must be mapped to a link arriving to n.

$$\forall v, e, n | G(v, e), \Sigma_{l|H(n,l)}, M_{el}(e, l) = M_{vn}(v, n) \tag{6.3}$$

$$\forall v, e, n | G(e, v), \Sigma_{l|H(l,n)}, M_{el}(e, l) = M_{vn}(v, n) \tag{6.4}$$

In addition, the scheduler must ensure that each edge is mapped to a *contiguous* path of links. We achieve this by enforcing that for each router, either we have no incoming or outgoing links mapped to a given edge, or we have exactly one incoming and exactly one outgoing link mapped to the edge.

$$\forall e \in E, r \in R \quad \Sigma_{l|H(l,r)}, M_{el}(e, l) = \Sigma_{l|H(r,l)} M_{el}(e, l) \tag{6.5}$$

$$\forall e \in E, r \in R \qquad \Sigma_{l|H(l,r)}, M_{el}(e, l) \leq 1 \tag{6.6}$$

Figure 6.3 shows these constraints applied to the example.

Some architectures require dimension order routing: a message propagating along the X direction may continue on a link along the Y direction, but a message propagating along the Y direction cannot continue on a link along the X direction. To enforce this restriction, we expand the description of the hardware with $I(L, L)$, the set of link pairs that cannot be mapped to the same edge (i.e. an edge cannot be assigned to a path containing any link pair in this set).

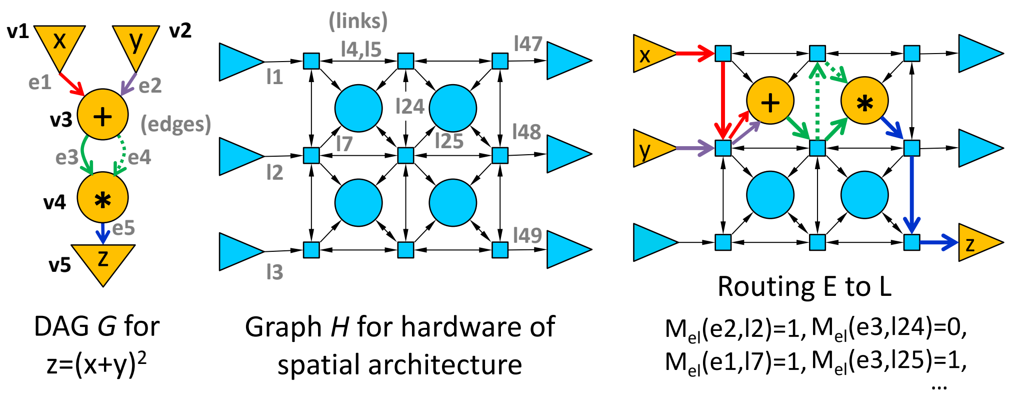

v1
x
y
v2
e1
e2
v3
+
(edges)
e3
e4
v4
*
e5
v5
z
DAG *G* for z=(x+y)^2
(links)
l4,l5
l47
l1
l24
l7
l25
l48
l2
l49
l3
Graph *H* for hardware of spatial architecture
x
y
z
Routing E to L
M_{el}(e2,l2)=1, M_{el}(e3,l24)=0,
M_{el}(e1,l7)=1, M_{el}(e3,l25)=1,
...

Figure 6.3: Routing of data.

$$\forall l, l' | I(l, l'), e \in E,\ M_{el}(e, l) + M_{el}(e, l') \leq 1 \tag{6.7}$$

Responsibility 3: Manage timing of events

We capture the timing through a set of variables $T(V)$ which represents the time at which a vertex $v \in V$ starts executing. For each edge connecting the vertices v_{src} and v_{dst}, we compute the $T(v_{dst})$ based on $T(v_{src})$. This time is affected by three components. First, we must take into account the $\Delta(E)$, which is the number of clock cycles between the start time of the vertex and when the data is ready. Next is the total routing delay, which is the sum of the number of mapped links between v_{src} and v_{dest}. Since the data carried by all input edges for a vertex might not all arrive at the same time, the variable $X(E)$ describes this mismatch.

$$\forall v_{src}, e, v_{dest} | G(v_{src}, e) \& G(e, v_{dest}),$$
$$T(v_{src}) + \Delta(e) + \Sigma_{l \in L} M_{el}(e, l) + X(e) = T(v_{dest}) \tag{6.8}$$

The equation above cannot fully capture dynamic events like cache misses. Rather than consider all possibilities, the scheduler simply assumes best-case values for unknown latencies (alternatively, these could be attained through profiling or similar means). Note that this is an issue for specialized

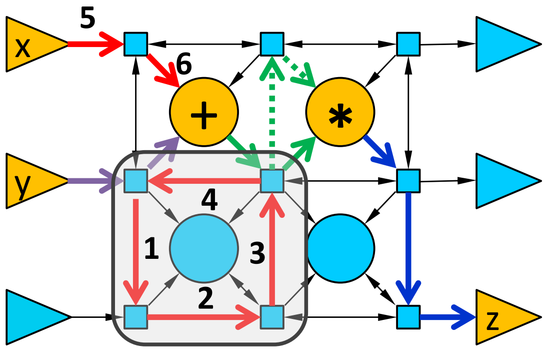


Figure 6.4: Example mapping with fictitious cycles.

schedulers as well.

With the constraints thus far, it is possible for the scheduler to overestimate edge latency because the link mapping allows fictitious cycles. As shown by the cycle in the bottom-left quadrant of Figure 6.4, the links in this cycle falsely contribute to the time between input "x" and vertex "+". This does not violate constraint 6.5 because each router involved contains the correct number of incoming / outgoing links.

In many architectures, routing constraints (see constraint 6.7) make such loops impossible, but when this is not the case we eliminate cycles through a new constraint. We add a new set of variables $O(L)$, indicating the partial order in which links activated. If an edge is mapped to two connected links, this constraint enforces that the second link must be of later order.

$$\forall l, l', e \in E | H(l, l'),\ O(l) + M_{el}(e, l) + M_{el}(e, l') - 1 \leq O(l') \tag{6.9}$$

Figure 6.5 shows the intermediate variable assignments that the constraints for timing provide.

Responsibility 4: Managing Utilization

The utilization of a hardware resource is simply the number of cycles for which it can not accept a new unit of work (computation or communication) because it is handling work corresponding

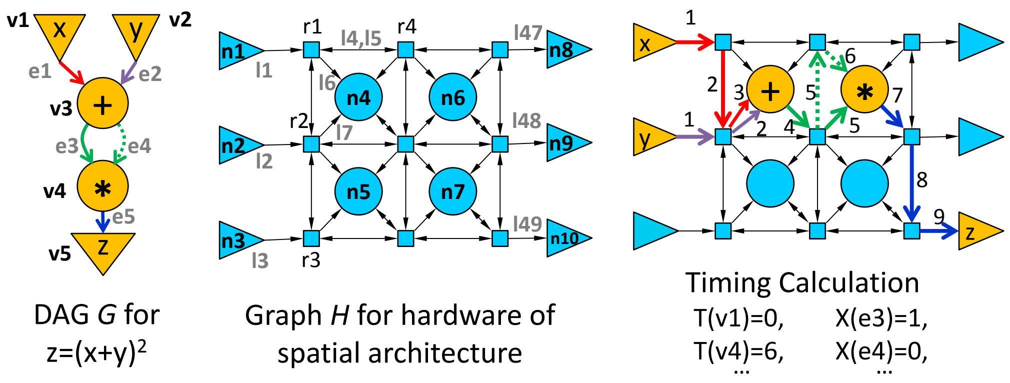


Figure 6.5: Timing of computation and communication.

to another computation. We first discuss the modeling of link utilization $U(L)$, then discuss node utilization $U(N)$.

$$\forall l \in L, \quad U(l) = \Sigma_{e \in E} M_{el}(e,l) \tag{6.10}$$

The equation above models a link's utilization as the sum of its mapped edges and is effective when each edge takes up a resource. On the other hand, some architectures allow for edges to be overlapped, as in the case of multicast, or if it is known that sets of messages are mutually exclusive (will never activate at the same time). This requires us to extend our notion of utilization with the concept of *edge-bundles*, which represent edges that can be mapped to the same link at no cost. The set B_e denotes edge-bundles, and $B(E, B_e)$ defines its relationship to edges. The following three constraints ensure the correct correspondence between the mapping of edges to links and bundles to links, and compute the link's utilization based on the edge-bundles.

$$\forall e, b_e | B(e, b_e), l \in L, \qquad M_{bl}(b_e, l) \geq M_{el}(e,l) \tag{6.11}$$

$$\forall b_e \in B_e, l \in L, \quad \Sigma_{e \in B(e,b_e)} M_{el}(e,l) \geq M_{bl}(b_e, l) \tag{6.12}$$

$$\forall l \in L, \qquad U(l) = \Sigma_{b_e \in B} M_{bl}(b,l) \tag{6.13}$$

To compute the vertices' utilization, we must additionally consider the amount of time that a vertex fully occupies a node. This time, $\Delta(V)$, is always 1 when the architecture is fully pipelined, but increases when the lack of pipelining limits the use of a node n in subsequent cycles. To compute utilization, we simply sum $\Delta(V)$ over vertices mapped to a node:

$$\forall n \in N \;\; U(n) = \Sigma_{v \in V} \Delta(v) M_{vn}(v,l) \tag{6.14}$$

For many spatial architectures we use utilization-limiting constraints such as those below. One application of these constraints are hardware limitations in the number of registers available, instruction slots, etc. Also, they ensure lack of contention with operations or messages from within the same block or other blocks executing concurrently.

$$\forall l \in L, \quad U(l) \leq MAX_L \tag{6.15}$$

$$\forall n \in N, \quad U(n) \leq MAX_N \tag{6.16}$$

As shown in the running DySER example below in Figure 6.6, we limit the utilization of each link $U(l)$ to $MAX_L = 1$. This ensures that only a single message per block traverses the link, allowing the DySER's arbitration-free routers to operate correctly.

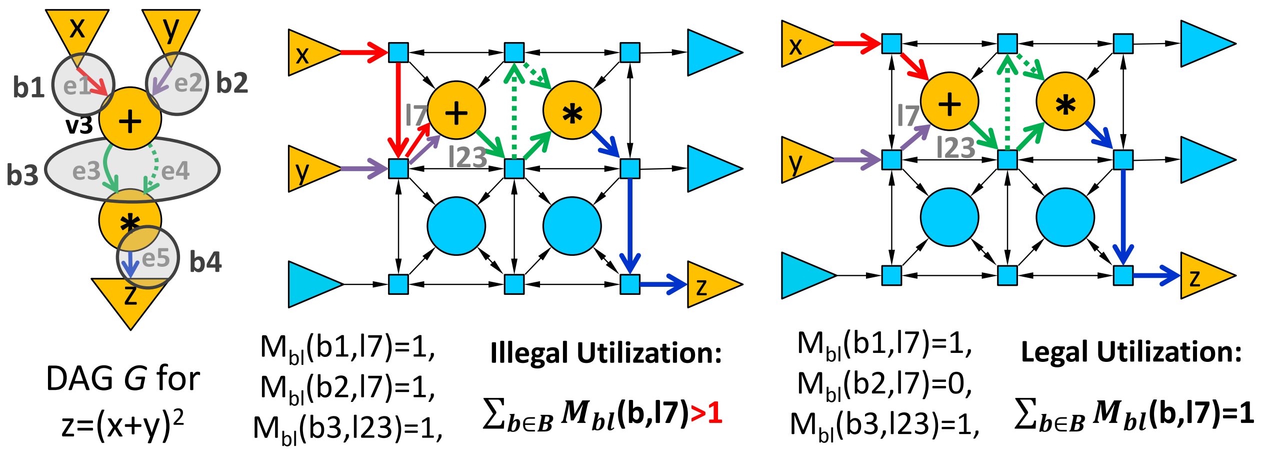


Figure 6.6: Utilization Management.

Responsibility 5: Optimizing performance

The constraints governing the previous sections model the quantities which capture only *individual* components for correctness and performance. However, the final responsibility of the scheduler is to manage the overall correctness while providing performance in the context of the overall system. In practice, this means that the scheduler must balance notions of latency and throughput. Having multiple conflicting targets requires strategic resolution, since there is not necessarily a single solution which optimizes both. The strategy we take is to supply to the scheduler a set of variables to optimize for with their associated priority.

To calculate the critical path latency, we first initialize the input vertices to zero (or some known value) then find the maximum latency of an output vertex LAT. This represents the scheduler's estimate of how long the block would take to complete.

$$\forall v \in V_{in}, \quad T(v) = 0 \tag{6.17}$$

$$\forall v \in V_{out}, \quad T(v) \leq LAT \tag{6.18}$$

To model the throughput aspects, we utilize the concept of the service interval SVC, which is defined as the minimum number of cycles between successive invocations when no data dependencies between invocations exists. We compute SVC by finding the maximum utilization on any resource.

$$\forall n \in N, \quad U(n) \leq SVC \tag{6.19}$$

$$\forall l \in L, \quad U(l) \leq SVC \tag{6.20}$$

For fully pipelined architectures, SVC is naturally forced to 1, so it is not an optimization target. Other notions of throughput are possible, as in the case of DySER, where minimizing the latency mismatch MIS is the throughput objective (see Section 6.4).

For our running example, the final solution is shown in Figure 6.7, where the critical path latency LAT and the latency mismatch MIS (mentioned above), are both optimized by the scheduler.

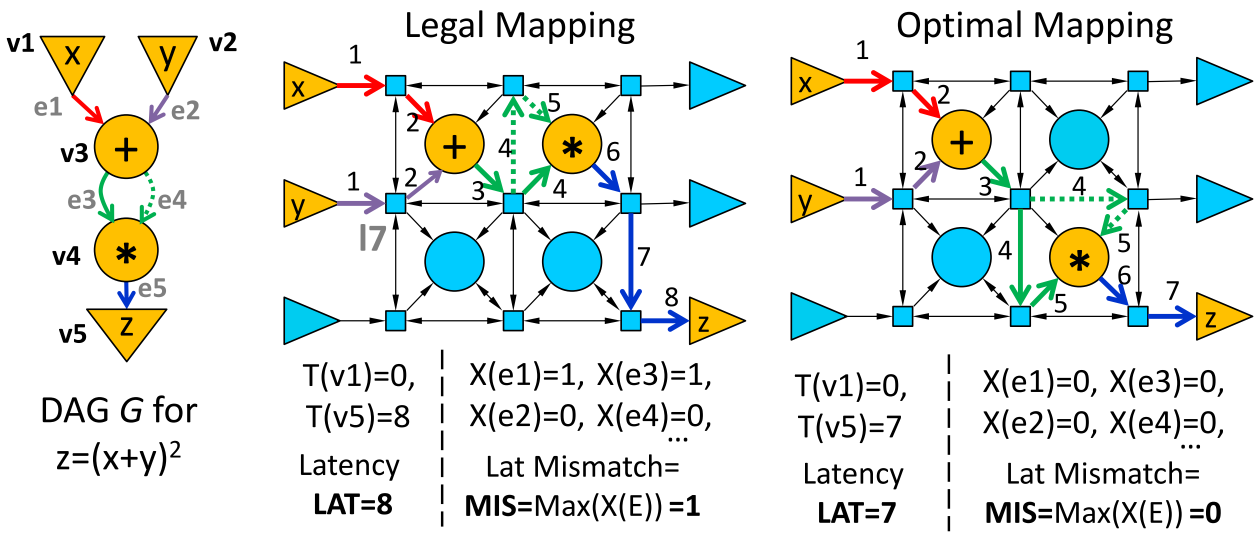

v1
x
y
v2
e1
e2
v3
+
e3
e4
v4
*
e5
v5
z
DAG G for z=(x+y)²
Legal Mapping
T(v1)=0, T(v5)=8
Latency LAT=8
X(e1)=1, X(e3)=1, X(e2)=0, X(e4)=0, ...
Lat Mismatch= MIS=Max(X(E)) =1
Optimal Mapping
T(v1)=0, T(v5)=7
Latency LAT=7
X(e1)=0, X(e3)=0, X(e2)=0, X(e4)=0, ...
Lat Mismatch= MIS=Max(X(E)) =0

Figure 6.7: Optimizing performance.

6.4 Architecture-specific modeling

In this section, we describe how the general formulation presented above is used by three diverse architectures. Figure 6.8 shows schematics and H graphs for the three architectures.

Architecture-specific details for DySER

Figure 6.8 shows the basic hardware diagram and abstract hardware graph H for each architecture or accelerator.

Computation organization $\rightarrow$ **Placement of computation:** We model DySER with the hardware graph H shown in Figure 6.8; heterogeneity is captured with the $C(V, N)$ compatibility matrix.

Network organization $\rightarrow$ **Routing data:** We use bundle-link mapping constraints to model multicast, and constraint 6.9 to prevent fictitious cycles. Since the network has the ability to perform multicast messages and can route multiple edges on the same link, we use the bundle-link mapping

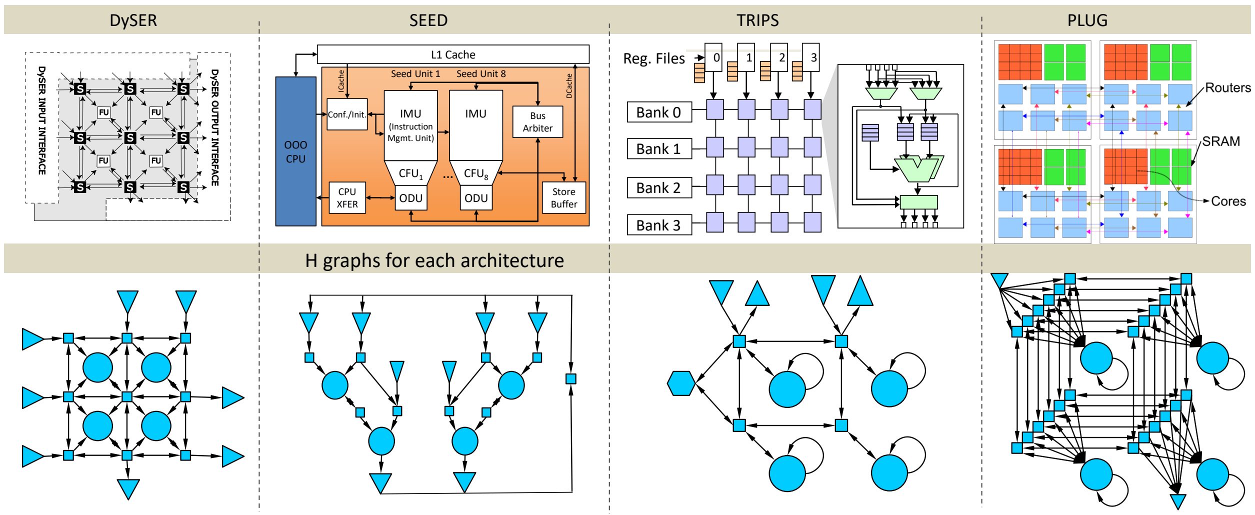


Figure 6.8: Three candidate architectures and corresponding H graphs, considering 4 execution resources (nodes) for each architecture.

constraints. Since there is no ordering constraint on the network, we need to prevent fictitious cycles.

HW timing $\rightarrow$ Managing timing of events: No additions to the general formulation are required.

Concurrent HW usage $\rightarrow$ Utilization: Since DySER can only route one message per link, and max one vertex to a node, both MAX_L and MAX_N are set to 1.

Objective Formulation: DySER throughput can be as much as one computation G per cycle, since the functional units themselves are pipelined. However, throughput degradation can occur because of the limited buffering available for messages. The utilization defined in the general framework does not capture this problem because it only measures the usage of functional units and links, not of buffers. Unlike TRIPS, where all operands are buffered as long as needed in named registers, DySER buffers messages at routers and at most one message per edge is buffered at each router. Thus, two paths that diverge and then converge, but have different lengths, will also have different amounts of buffering. Combined with back-pressure, this can reduce throughput.

Computing the exact throughput achievable by a DySER schedule is difficult, as multiple such pairs of converging paths may exist - even paths that converge after passing through functional units affect throughput. Instead we note that latency mismatches always manifest themselves as extra buffering delays $X(e)$ for some edges, so we model latency mismatch as MIS:

$$\forall e \in E, X(e) \leq MIS \tag{6.21}$$

Empirically, we found that external limitations on the throughput of inputs is greater than that of computation. For this reason, the DySER scheduler first optimizes for latency, adds the latency of the solution as a constraint, then optimizes for throughput by minimizing latency mismatch MIS, as below:

$$min\ LAT\ s.t.\ [6.1\text{–}6.11, 6.12\text{–}6.18, 6.21]$$

$$min\ MIS\ s.t.\ [6.1\text{–}6.11, 6.12\text{–}6.18, 6.21] \quad \text{and } LAT\ =\ LAT_{optimal}$$

Architecture-specific details for SEED and Trace-P

The formulation for SEED (and the Trace-Processor) differs in that it supports compound function unit execution – and hence must support ways of grouping instructions. Critically, compound instructions cannot fire until all operands are ready – and therefore it is possible to introduce extra latency with improper instruction grouping. Figure 6.9 shows the importance of an effective scheduler with two example schedules of the same region. This example shows that small differences in the schedule can affect the critical path by many cycles.

We note here that the scheduling formulation for SEED and Trace-P is nearly identical, as their architectures are very similar as far as the instruction scheduling abstraction is concerned. In terms of the compiler implementation, the main difference is that Trace-P schedules for an entire loop-trace, as it group instructions across basic blocks. SEED scheduling is only relevant within the basic block, as instructions cannot be re-grouped across control regions. Therefore, the scheduling

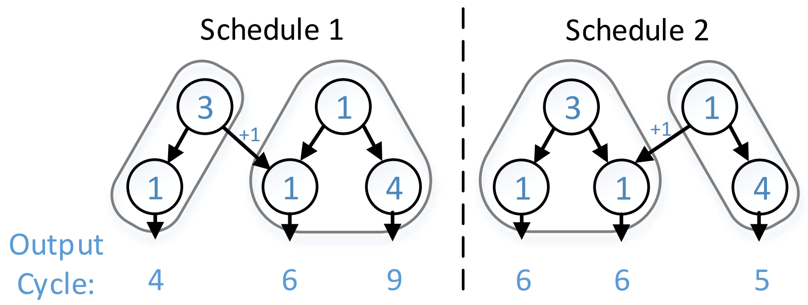


Figure 6.9: CFU Scheduling Example. Each operation is labeled with its latency, and compound instruction groups are circled.

extensions we describe here are relevant to both architectures.

In order to support compound instructions, we add a variable $\beta(v1, v2)(V, V)$. Its job is to indicate if there is a "boundary" between $v1$ and $v2$, which represents whether they are executing on different CFU instances. This will ultimately enable the modeling of instruction timing, as will become clear below.

Computation organization $\rightarrow$ **Placement of computation:** To model the mapping of vertices to different CFU instances, we require a few additional constraints. The one below enforces that either a vertex's inputs are either directly routed through a hardware input, or they are executed on separate instances of a CFU (indicated by $\beta(v1, v2) = 1$).

$$\forall_{\substack{v1,v2 \in G \\ n2 \in N}} \quad \beta(v1, v2) \geq M(v2, n2) - \sum_{\substack{v1,n1 \in C \\ n1,n2 \in H}} M(v1, n1) \tag{6.22}$$

Next, we need to model the fact that data cannot both leave and return to the same CFU instance. We do that by keeping consistent notions of the boundary, specifically that if there are no boundaries between $v1$ and $v2$, and $v2$ and $v3$, then there can not be a boundary between $v1$ and $v3$. We model this boundary transitivity property with constraints 6.23 and 6.24. Note that this is only enforced if all nodes can possibly map together, indicated by C.

$$\underset{v1,v2,v3 \in C(v1,v3) \cap C(v1,v) \cap C(v2,v3)}{\forall} \quad (1-\beta(v1,v2)) + (1-\beta(v2,v3)) - 1 \leq (1-\beta(v1,v3)) \tag{6.23}$$

$$\forall v1, v2 \in G, \quad \beta(v1,v2) = \beta(v2,v1) \tag{6.24}$$

We also need to enforce that for any two nodes that could possibly map to each other, they are only allowed to map to the same hardware node if they are on the same CFU instance. The following constraint enforces this.

$$\forall v1, v2, n | C(v1,n) \cap C(v2,n) \cap C(v1,v2), \quad M(v1,n) + M(v2,n) \leq \beta(v1,v2) + 1 \tag{6.25}$$

Network organization $\rightarrow$ Routing data:

No additions to the general formulation are required.

HW timing $\rightarrow$ Managing timing of events:

We add an additional constraint to enforce that all components of a CFU may only start after all inputs arrive. It uses a Big-M constraint formulation to enforce this property.

$$\forall v1, v2 \in C \cap vi, v1 \in G, \quad T(v2) \geq (\beta(vi,v1) - \beta(v1,v2) - 1) * M + \Delta(vi) + T(vi) \tag{6.26}$$

Concurrent HW usage $\rightarrow$ Utilization: SEED allows many instructions to be executing on the same functional units. Specifically the vertices per node is $MAX_N = 32$.

Objective formulation: Since SEED execution is generally latency constrained, (rather than constrained by the throughput of the individual compound functional units) the latency is used to formulate the objective function.

$$min\ LAT\ s.t.\ [6.1\text{–}6.6, 6.8, 6.10\text{–}6.18, 6.22\text{–}6.26]$$

Architecture-specific details for TRIPS

The trips formulation is a straightforward application of the general model, with a few additions to optimize for utilization variations in the presence of control flow.

Computation organization $\rightarrow$ Placement of computation: Figure 6.8 depicts the graph H we use to describe a 4-tile TRIPS architecture. A tile in TRIPS is comprised of nodes $n \in N$ denoting a functional unit in the tile and $r \in R$ representing its router - the two are connected with one link in either direction. The router also connects to the routers in the neighboring tiles. The functional unit has a self-loop that denotes the bypass of the tile's router to move results into input slots for operations scheduled on the same tile.

Network organization $\rightarrow$ Routing data: Since messages are dedicated and point-to-point (as opposed to multicast), we use constraints modeling each edge as consuming a resource and contributing to the total utilization. The TRIPS routers implement dimension-order routing, i.e. messages first travel along the X axis, then along the Y axis. TRIPS uses the $I(L, L)$ parameter, which disallows the mapping of certain link pairs, to invalidate any paths which are not compatible with dimension-order.

HW timing $\rightarrow$ Managing timing of events: We can calculate network timing without any additions to the general formulation.

Concurrent HW usage $\rightarrow$ Utilization: TRIPS allows significant level of concurrent hardware usage which affects both the latency and throughput of blocks. Specifically, the maximum number of vertices per node is $MAX_N = 8$. The utilization on links is used to finally formulate the objective function.

Extensions: For TRIPS, the scheduler must also account for control flow when computing the utilization and ultimately the service interval for throughput. Simple extensions, as explained below, can in general handle control flow for any architecture and could belong in the general ILP formulation as well. Let P be the set of control flow paths that the computation can take through G. Note that $p \in P$ is not actually a path through G, but the subset of its vertices and edges activated in a given execution. Let $A_v(P,V)$ and $A_e(P,E)$ be the activation matrices defining, for each vertex and edge of the computation, whether they are activated when a given path is taken or not. For each path we define the maximum utilization on this path $W_p(P)$. These constraints are similar to the original utilization constraints (6.10, 6.14), but also take control flow activation matrices into account.

$$\forall l \in L, p \in P, \quad \Sigma_{e \in E} M_{el}(e,l) A_e(p,e) \leq W_p(p) \tag{6.27}$$

$$\forall n \in N, p \in P, \quad \Sigma_{v \in V} M_{vn}(v,n) \Delta(v) A_v(p,v) \leq W_p(p) \tag{6.28}$$

And an additional constraint for calculating overall service interval:

$$SVC = \Sigma_{p \in P} W_p(p) \tag{6.29}$$

Note that this heuristic provides the same importance to all control-flow paths. With profiling or other analysis, differential weighting can be implemented.

Objective formulation: For the TRIPS architecture, we empirically found that optimizing for throughput is of higher importance, in most cases, then for latency. Therefore, our strategy is to first minimize the SVC, add the lowest value as a constraint, and then optimize for LAT. The following is our solution procedure, where numbers refer to constraints from the formulation:

Architecture	Description	MILP Modeling and scheduler responsibility	MILP Constraints
Compute HW Organization	16 tiles, 6 routers per tile	Each tile is 7 nodes in H, one in N, and six in R	Gen. framework
	4 mem-banks per tile	Handled with utilization	Gen. framework
	32 cores per tile	Handled with utilization	Gen. framework
Network HW Organization	2D nearest neighbor mesh	Node n connected to r; connected to 4 neighbors	Gen. framework
	Dimension order routing	$I(L, L)$ configure for dimension-order routing	Gen. framework
	Multicast messages deliver to every node on path	Map mutlicast message from link to node on path	Constraint 6.30
HW timing	Code-scheduling of network send instructions	Variables for send/receive time	$\Delta(E)$; $\Gamma(E)$
	Send instructions scheduled to avoid network conflicts	Variables for delaying send/receive timing with no-ops	Constraints 6.27a-c, 6.33
Concurrent HW usage	4 Mem-banks per time	Manage utilization (MAX_N = 4)	Gen. framework
	Dedicated network per message	Manage utilization (MAX_L = 1)	Gen. framework
	Mutually exclusive activation of nodes in G	Concept of vertex bundles and utilization refined	Constraints 6.34, 6.35, 6.36
	Code-length limitations (max 32) handled for all code on tile	Manage utilization and combine with vertex bundles	Constraints 6.37, 6.38, 6.39

Table 6.3: Description of MILP model implementation for PLUG

$$min\ SVC\ s.t.\ [6.1–6.8, 6.10, 6.14–6.18, 6.27–6.29]$$

$$min\ LAT\ s.t.\ [6.1–6.8, 6.10, 6.14–6.18, 6.27–6.29] \quad \text{and } SVC = SVC_{optimal}$$

Architecture-specific details for PLUG

The PLUG architecture is radically different from the previous two architectures since all decisions are static. Our formulation is general enough that it works for PLUG with only 10 predominantly simple additional constraints. In the interest of clarity, we summarize the key concepts of the PLUG architecture, corresponding ILP model, and additional equations in Table 6.3. The grayed rows summarize the extensions, and this section's text describes them.

Computation organization $\rightarrow$ Placement of computation: See Table 6.3, row 1. No additional constraints required.

Network organization $\rightarrow$ Routing data: See Table 6.3, row 2.
Additional constraints: Multicast handled with edge-bundles: Let $B_{multi} \subset B_e$ be the subset of edge-bundles that involve multicast edges. The following constraint, which considers links through a router to a node, then enforces that the bundle mapped to the router link must also be mapped to the node's incoming link.

$$\begin{gathered} \forall b \in B_{multi}, l, r, l', n | H(l, r) \& H(r, l') \& H(l', n), \\ M_{bl}(b, l) \leq M_{bl}(b, l') \end{gathered} \tag{6.30}$$

HW timing $\rightarrow$ Managing timing of events: See Table 6.3, row 3.
Additional constraints: We need to handle the timing of send instructions. We use $\Delta(E)$ and the newly-introduced $\Gamma(E)$ to respectively indicate the relative cycle number of the corresponding send instruction and use instruction.

Network contention is avoided by code-scheduling the send instructions with NOP padding to create appropriate delays and equalize all delay mismatch. $\delta(E)$ denotes sending delay added, and $\gamma(E)$ denotes receiving delay added. To model the timing for PLUG, we augment equation 6.8 as follows:

$$\begin{gathered} \forall v_{src}, e, v_{dst} | G(v_{src}, e) \& G(e, v_{dst}), \\ T(v_{src}) + \Sigma_{l \in L} M_{el}(e, l) + \Delta(e) + \delta(e) = T(v_{dst}) + \Gamma(e) + \gamma(e) \end{gathered} \tag{6.31}$$

Because the insertion of no-ops can only change timing in specific ways, we use two constraints to further link $\delta(E)$ and $\gamma(E)$ to $\Delta(E)$ and $\Gamma(E)$. The first ensures that the scheduler never attempts to pad a negative number of NOPs. The second ensures that sending delay $\delta(E)$ is the same for all multicast edges carrying the same message.

To implement these constraints we use the following 4 sets concerning distinct edges e, e': $SI(e, e')$ has the set of pairs of edges arriving to the same vertex such that $\Gamma(e) < \Gamma(e')$, $LIFO(e, e')$ has, for each vertex with both input and output edges, the last input edge e and the first output edge e', $SO(e, e')$ has the pairs of output edges with the same source vertex such that $\Delta(e) < \Delta(e')$, and $EQO(e, e')$ has the pairs of output edges leaving the same node concurrently.

$$\forall e, e' | SI(e, e'), \gamma(e) \leq \gamma(e') \tag{6.32a}$$

$$\forall e, e' | LIFO(e, e'), \gamma(e) \leq \delta(e') \tag{6.32b}$$

$$\forall e, e' | SO(e, e'), \delta(e) \leq \delta(e') \tag{6.32c}$$

$$\forall e, e' | EQO(e, e'), \delta(e) = \delta(e') \tag{6.33}$$

Concurrent HW usage $\rightarrow$ Utilization: See Table 6.3, row 4.

Additional constraints: PLUG groups nodes in G into "super-nodes" (logical-page), and programmatically only a single node executes in every super-node. This mutual exclusion behavior is modeled by partitioning V into a set of vertex bundles B_v with $B(V, B_v)$ indicating to which bundle a vertex $v \in V$ belongs. We introduce $M_{bn}(b, n)$ to model the mapping of bundles to nodes, enforced by the following constraints:

$$\forall v, b_v | B(v, b_v), n \in N, \qquad M_{bn}(b_v, n) \geq M_{vn}(v, n) \tag{6.34}$$

$$\forall b_v \in B_v, n \in N, \qquad \Sigma_{v \in B(v, b_v)} M_{vn}(v, n) \geq M_{bn}(b_v, n) \tag{6.35}$$

We then define the utilization based on the number of vertex bundles mapped to a node. We also instantiate edge bundles b_e for all the set of edges coming from the same vertex bundle and going to the same destination. Since all the edges in such a bundle are *logically* a single message source, the schedule must equalize the receiving times of the message they send. Let $B_{mutex} \subseteq B_e$ be the set of edge-bundles described above. Then we add the following timing constraint:

$$\forall e, e', b_x \in B_{mutex} | B(e, b_x) \& B(e', b_x), \quad \gamma(e) = \gamma(e') \tag{6.36}$$

Additionally, architectural constraints require the total length in instructions of the vertex bundles mapped to the same node to be ≤ 32. This requires defining, for each bundle, the maximum bundle length $\lambda(b_v)$ as a function of the last send message of the vertex. This length can then be constrained to be ≤ 32.

To achieve this, we first define the set $LAST(B_v, B_e)$, which pairs each vertex bundle with its last edge bundle, corresponding to the last send message of the vertex. This enables to define the maximum bundle length $\lambda(b_v)$ as:

$$\forall e, b_e, b_v | LAST(b_v, b_e) \& B(e, b_e), \quad \Delta(e) + \delta(e) \leq \lambda(b_v) \tag{6.37}$$

We finally define $Q(B_v, N)$ as the required number of instructions on node n from vertex bundle b_v and limit it to 32 (the code-snippet length).

$$\forall b_v, n \in N, \qquad Q(b_v, n) - 32 * M_{bn}(b_v, n) \geq \lambda(b_v) - 32 \tag{6.38}$$

$$\forall n, \qquad \Sigma_{b_v \in B_v} Q(b_v, n) \leq 32 \tag{6.39}$$

Objective Formulation: For PLUG, the smallest service interval is achieved and enforced for any legal schedule, and we optimize solely for latency LAT.

$$min\ LAT\ s.t.\ [6.1–6.7, 6.11–6.18, 6.30–6.39]$$

6.5 Implementation and Evaluation

In this section, we describe our implementation of the constraints in an off-the-shelf ILP solver and evaluate its performance compared to native specialized schedulers for the three architectures.

Implementation

We use the GAMS modeling language to specify our constraints as mixed integer linear programs, and we use the commercial CPLEX solver to obtain the schedules. Our implementation strategy for prioritizing multiple variables follows a standard approach: we define an allowable percentage optimization gap (of between 2% to 10%, depending on the architecture), and optimize for each variable in prioritized succession, finishing the solver when the percent gap is within the specified bounds. After finding the optimal value for each variable, we add a constraint which restricts that variable to be no worse in future iterations.

Figure 6.10 shows our implementation and how we integrated with the compiler/simulator toolchains [105, 32, 106]. For all three architectures, we use their intermediate output converted into our standard directed acyclic graph (DAG) for G and fed to our GAMS ILP program. We specified H for each architecture. To evaluate our approach, we compare the performance of the final binaries on the architectures varying only the scheduler. Table 6.4 summarizes the methodology and infrastructure used.

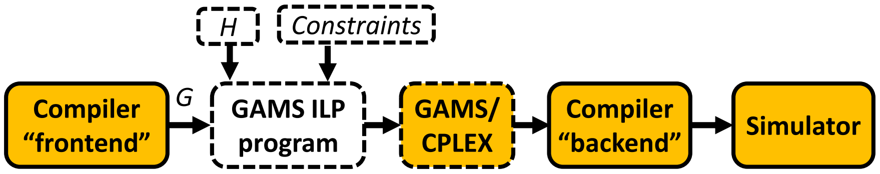
Results

Is this ILP-based approach implementable? Yes, it is possible to express the scheduling problem as an ILP problem and implement it for real architectures. Considering the ILP constraint formulation

	TRIPS	DySER	PLUG
Benchmarks	• Same as prior TRIPS scheduler papers [107]. SPEC microbenchmarks and EEBMC • Full SPEC benchmarks can not run to completion on simulator and do not stress scheduler (since blocks are small)	• DySER data-parallel workloads since they produce large blocks and complete code from compiler [32]. • Additional throughput microbenchmark [a]	• PLUG benchmarks from [106]
Native scheduler	• Optimized SPS scheduler [107]	• Specialized greedy algorithm in toolchain & hand scheduled [32]	• Hand scheduled [106]
Metric	• Total execution cycles for program	• Total execution cycles for program	• Total execution cycles for lookups

[a]DySER "throughput" microbenchmark: This performs the calculation $y = x - x^{2i}$ in the code-region. Paths diverge at the input node x, into one long path which computes x^{2i} with a series of i multiplies, and along a short path which routes x to the subtraction. This pattern tends to cause latency mismatch because one of these converging paths naturally takes less resources

Table 6.4: Tools and methodology for quantitative evaluation



"frontend": passes in the compiler that produce pre-scheduled code;
"backend": passes that convert scheduled code into binary.

Figure 6.10: Implementation of our ILP scheduler. Dotted boxes indicate the new components added.

for the general framework, our GAMS implementation is around 50 lines of code.

Result-1: A declarative and general approach to expressing and implementing spatial-architecture schedulers is possible.

Is the execution time of standard ILP-solvers fast enough to be practical? Table 6.5 (page 120) summarizes the mathematical characteristics of the workloads and corresponding scheduling behavior. The three right-hand columns respectively show the number of software nodes to schedule, the amount of single ILP equations created, and the solver time.[2] There is a rough correlation between the workload "size" and scheduling time, but it is still highly variable.

[2]For TRIPS, the per-benchmark number of DAGs can range from 50 to 5000, and the metrics provided are average per DAG. For DySER, #DAGs is 1 to 4 per benchmark, and PLUG is always 1.

The solver time of the specialized schedulers in comparison is typically on the order of seconds or less. Although some blocks may take minutes to solve, these times are still tractable, demonstrating the practicality of ILP as a scheduling technique.

Result-2: Our general ILP scheduler runs in tractable time.

Are the output solutions produced good? How do they compare against the output of specialized schedulers? Figure 6.11 (page 119) shows the performance of our ILP scheduler. It shows the cycle-count reduction for the executed programs as a normalized percentage of the program produced by the specialized compiler (higher is better, negative numbers mean execution time was increased). We discuss these results in terms of each architecture.

Compared to the TRIPS SPS specialized scheduler (a cumulated multi-year effort spanning several publications [72, 108, 107]), our ILP scheduler performs competitively as summarized below.[3]

Compared to SPS		
(a) Better on 22 of 43 benchmarks	up to 21%	GM +2.9%
(b) Worse on 18 of 43 benchmarks	within 4.9% **(typically 2%)**	GM -1.9%
(c) 5.4%, 6.04%, and 13.2% worse on ONLY 3 benchmarks		
Compared to GRST		
Consistently better, up to 59% better; GM +30%		

Groups (a) and (b) show the ILP scheduler is capturing the architecture/scheduler interactions well. The small slowdowns/speedups compared to SPS are due to dynamic events which disrupt the scheduler's view of event timing, making its node/link assignments sub-optimal, typically by only 2%. After detailed analysis, we discovered the reason for the performance gap of group (c) is the lack

[3]We did not run on SPEC benchmarks for three reasons: prior TRIPS scheduler work uses this set; TRIPS simulator does not have sim-point etc. to meaningfully simulate TRIPS benchmarks; TRIPS compiler does not produce good enough code on SPEC (10-15 inst blocks only) to make scheduler a factor [107, 76]. Using TRIPS hardware was impractical for us.

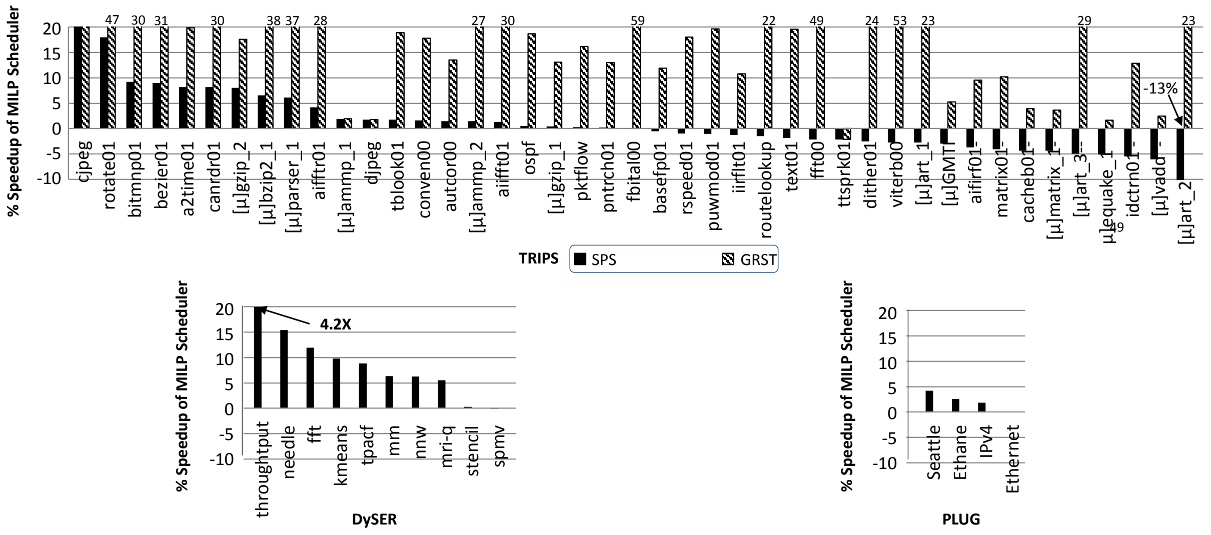


Figure 6.11: Normalized percentage improvement in execution cycles of ILP scheduler compared to specialized scheduler.

of information that could be easily integrated in our model. First, the SPS scheduler took advantage of information regarding the specific cache banks of loads and stores, which is not available in the modular scheduling interface exposed by the TRIPS compiler. This knowledge would improve the ILP scheduler's performance and would only require changes to the compatibility matrix $C(V, N)$. Second, knowledge of limited resources was available to SPS, allowing it to defer decisions and interact with code-generation to map movement-related instructions. What these results show overall is that our first-principles based approach is capturing all the architecture behavior in a general fashion and arguably aesthetically cleaner fashion than SPS's indirect heuristics. Our ILP scheduler consistently exceeds by appreciable amounts a previous generation TRIPS scheduler, GRST, that did not model contention [108], as shown by the hatched bars in the figure.

On DySER, the ILP scheduler outperforms the specialized scheduler on all benchmarks, as shown in Figure 6.11, for a 64-unit DySER. Across the benchmarks, the ILP scheduler reduces *individual* block latencies by 38% on average. When the latency of DySER execution is the bottleneck, especially when there are dependencies between instances of the computation (like the needle benchmark), this leads to significant speedup of up to 15%. We also implemented an extra DySER

Trips EEBMC	**# of nodes**	**# of eqns**	**Solve (sec)**
a2time01	11	1914	5
aifftr01	12	2173	25
aifirf01	11	2025	1
basefp01	10	1863	6
bitmnp01	9	1535	3
cacheb01	27	2745	76
candr01	10	1871	8
idctrn01	11	1947	3
iirflt01	11	2080	2
matrix01	11	1426	2
pntrch01	10	1819	8
puwmod01	10	1779	3
rspeed01	10	1816	7
tblook01	10	1818	4
ttsprk1	11	1993	8
cjpeg	12	2280	3
djpeg	12	2277	1
ospf	10	1778	3
pktflow	10	1774	3
routelookup	10	1747	3
bezier01	10	1788	2
dither01	10	3579	4
rotate01	10	1910	5
text01	10	1781	3
autocor00	10	1746	2
conven0	10	1758	4
fbital00	9	1699	3
viterb00	10	1870	5
TRIPS Avg.	**14**	**2832**	**31**

Trips μbench	**# of nodes**	**# of eqns**	**Solve (sec)**
ammp_1	17	3744	76
ammp_2	8	1593	11
art_1	22	4547	74
art_2	27	5506	76
art_3	33	7042	20
bzip_1	13	2655	10
equake_1	24	4455	3
gzip_1	23	4480	1
gzip_2	22	4506	111
matrix_1	19	3797	18
parser_1	33	7248	174
transp_GMTI	20	4159	115
vadd	30	7313	315

DySER Apps.	**# of nodes**	**# of eqns**	**Solve (sec)**
fft	20	120250	365
mm	32	159231	77
mri-q	19	98615	66
spmv	32	155068	72
stencil	30	153428	74
tpacf	40	211584	368
nnw	25	169197	102
kmeans	40	232399	218
needle	32	181686	183
throughput	9	45138	62
DySER Avg.	**28**	**152660**	**159**

PLUG Apps.	**# of nodes**	**# of eqns**	**Solve (sec)**
Ethernet	18	35603	57
Ethane	11	13905	14
IPv4	12	38741	384
Seattle	16	14531	26
PLUG Avg.	**14**	**23195**	**120**

Table 6.5: Benchmark characteristics and MILP scheduler behavior.

benchmark, which elucidates the importance of latency mismatch and is described in Table 6.4. The specialized scheduler tries to minimize the extra path length at each step, exacerbating the latency mismatch of the short and long paths in the program. The ILP scheduler, on the other hand, pads the length of the shorter path to reduce latency mismatch, increasing the potential throughput and achieving a 4.2× improvement over the specialized scheduler. Finally, we also compared to manually scheduled code on an *16-unit* DySER (since hand-scheduling for 64unit DySER is exceedingly tedious). The ILP scheduler always matched or out-performed it by a small ($< 2\%$) percentage.

The ILP scheduler matches or out-performs the PLUG hand-mapped schedules. It is able to both find schedules that force $SVC = 1$ and provide latency improvements of a few percent. Of particular note is solver time, because PLUG's DFGs are more complex. In fact, each DFG represents an *entire* application. The most complex benchmark, IPV4, contains 74 edges (24 more than any others) split between 30 mutually exclusive or multicast groups. Despite these difficulties, it completes in tractable time.

Result-3: Our ILP scheduler outperforms or matches the performance of specialized schedulers.

6.6 Discussion

In this section, we discuss some of the modeling limitations of our formulation, as well as implications for broader uses.

Modeling Limitations

Dynamic Events Modeling dynamic events, where uncertainty exists in certain problem quantities, is difficult to express in MILP. However, we found that approximating these dynamic events by common-case values was sufficient. We also note that we could extend our model with "stochastic programming" techniques, which solve the same problem for multiple input scenarios. We chose not to explore that for this problem because of the additional model complexity.

Cyclic Computations This chapter describes how to map computation *DAGs*, which are the typical unit of scheduling. Program loop structures will still occur, just around the unit of a DAG. Some spatial architectures require loops inside the unit of scheduling, but our framework's approach in the calculation of latency would lead to infeasible schedules when considering loops, because each node would have to "come after" the previous one. One simple solution to this problem is to ignore any loop backedges when considering timing, but we have not done a full investigation of an architecture that requires this feature.

Independence of Latency & Utilization One possible modeling of the spatial scheduling problem is to create binary decision variables both for "where" a computation should go, and "when" it should be activated, which we refer to as "space-time" scheduling. We have taken a slightly different approach in this formulation by assuming that the latency and utilization concerns are mostly independent, and only create decision variables for "where" a computation goes. We *rely* on the latency being calculable based on the mapping of computation and communication. This is not necessarily true, because with TRIPS, two computations which could both fire on the same tile at the same time will need to be arbitrated. For the purpose of the timing responsibility, the model optimistically assumes that both computations will fire at the same time. In general, our formulation does not take into account the fine-grained interaction of latency and utilization. That said, our approach use many fewer decision variables than a "space-time" approach, and we can more naturally model utilization constraints.

Flexibility and Broader Uses

Here we discuss some broader extensions and implications of our work. Specifically, we discuss the possibility and how our scheduler delivers on its promises of compiler-developer productivity/extensibility, cross-architecture applicability, and insights on optimality.

Formulation Extensibility: In our experience, our model formulation was easily adaptable and extensible for modeling various problem variations or optimizations. For example, we improved upon our TRIPS scheduler's performance by identifying blocks with carried-loop cache dependencies (commonly the most frequently executed), and extended our formulation to only optimize for relevant paths.

Application to Example Architectures: Table 6.6 shows how our framework could be applied to

Responsibility	RAW	WaveScalar	NPU
Placement	Homogeneous Cores (Tiles)	Homogeneous Processing Elements	8 Processing Elements, 1 Shared Bus
Routing	2D grid, unconstrained routing	Hierarchical Network. First two levels are fully connected, last level grid uses dynamic routing	Responsibility Not Applicable – Broadcast bus used for communication.
Timing	In-order execution inside tile, dataflow between tiles (Secondary list scheduler orders inter-stream events)	Data-flow execution, and dynamic network arbitration; network latency varies by hierarchy level	Fully Static execution. “No-ops” between bus events maintain synchronization.
Utilization	Many instructions per tile. Shared network links	64-Instructions/PE; Shared Network Links	Shared Processing Elements
Objective	Latency & Throughput	Contention & Latency	Latency

Table 6.6: Applicability to other Spatial Architectures

three other systems. For both WaveScalar and RAW, we can attain optimal solutions by refraining from making early decisions, essentially avoiding the drawbacks of multi-stage solutions. For WaveScalar, our scheduler would consider all levels of the network hierarchy at once, using different latencies for links in different networks. For RAW, our scheduler would consider both the partitioning of instructions into streams, and the spatial placement of these instructions simultaneously.

As a more recent example, NPU [31] is a statically-scheduled architecture like PLUG, but uses a broadcast network instead of a point-to-point, tiled network. Instead of using the routing equations for communication, the NPU bus is more aptly modeled as a computation type. Timing would be modeled similarly to PLUG, where “no-ops” prevent bus contention, allowing a fully static schedule.

Insights on Optimality: Since our approach provides insights on optimality, it has potentially broader uses as well. For instance, in the context of a dynamic compilation framework, even though the compilation time of seconds is impractical, the ILP scheduler still has significant practical value – it enables developers to easily formulate and evaluate objectives that can guide the implementation of specialized heuristic schedulers.

Revisiting NPU scheduling, we can observe another potential use of ILP models, specifically in designing the hardware itself. For the NPU, the fifo depth of each processing element is expensive in terms of hardware, so we could easily extend the model to calculate the fifo depth as a function of the schedule. One strategy would be to first optimize for performance, then fix the performance and optimize for lowest maximum fifo depth. Doing this across a set of benchmarks would give the best lower-bound fifo depth which does not sacrifice performance.

6.7 Related Work

Many others have used mathematical optimization to address scheduling problems in the past, and we summarize the most related in Table 6.7. In 1950, Wagner describes an ILP model for machine scheduling with dependent tasks [109]. Later, scheduling with MILP is brought to the field of computer architecture, including works like Feautrier's ILP model for modulo scheduling VLIW processors [110], and the multiprocessor scheduling work by Satish et. al. [111]. Perhaps the most related work, in terms of the domain, is that of Amarasinghe et. al, who formulated an ILP scheduling model for the RAW processor [112]. Though RAW is a spatial architecture, the model differs from ours in that it does not perform complex routing of communication, and does not model utilization.

Even though great strides have been made in mathematical models for scheduling, the state-of-the-art approaches for spatial scheduling are heuristic based. The five scheduling abstractions we described are not been modeled directly, and the typically NP-hard (depending on the hardware architecture) spatial architecture scheduling problem is side-stepped. Instead, the focus of architecture-specific schedulers has typically been on developing polynomial-time algorithms that approximate the optimal solution using knowledge about the architecture. Chronologically, this body of work includes the BUG scheduler for VLIW proposed in 1985 [117], UAS scheduler for clustered VLIW [118], synchronous data-flow graph scheduling [119], RAW scheduler [120], CARS VLIW code-generation and scheduler [121], TRIPS scheduler [107, 108], WaveScalar scheduler [122],

Yr	Technique	Comments or differences to our approach
1950	MILP machine sched. [109]	M-Job-DAG to N-resource scheduling. No job communication modeling, or network contention modeling. (missing ii,iv)
1992	MILP for VLIW [110]	Modulo scheduling. Cannot model an interconnection network, spatial resources, or network contention. (missing i,ii,iv)
1997	Inst scheduling [113]	Single-Processor Modulo Scheduling. (missing i,ii,iv)
2001	Process scheduling [114]	M-Job-DAG to N-resource scheduling using dynamic programming. Has no network routing or contention modeling, fixed job delays, and no flexible objective. (missing ii,iv,v)
2002	MILP for RAW [112]	M-Job-DAG to N-resource scheduling. Not generalizable as it does not model network routing or contention, just fixed network delays. (missing ii,iv)
2007	Multiproc. Sched. [111, 115]	M-Job-DAG to N-resource scheduling - No path assignment or contention modeling, just fixed delays. (missing ii,iv)
2008	SMT for PLA [116]	Strict communication and computation requirements: no network contention or path assignment modeling (missing ii,iv)

Table 6.7: Related work – Legend: i) computation placement ii) data routing iii) event timing iv) utilization management v) optimization objective

and CCA scheduler proposed in 2008 [123].

While heuristic-based approaches are popular and effective, they have three problems: i) *poor compiler developer/architect productivity* since new algorithms, heuristics, and implementations are required for each architecture, ii) *lack of insight* on optimality of solution, and iii) *sandboxing of heuristics* to specific architectures — understanding and using techniques developed for one spatial architecture in another very hard. Of course, the tradeoff in using MILP is that the solution time significantly longer. However, we were able to create a formulation that runs in a tractable amount of time to be useful for the architectures in question. Furthermore, the declarative approach that MILP enables takes much less development effort than designing an imperative algorithm to perform the same task.

6.8 Summary

Scheduling is a fundamental problem for spatial architectures, which are increasingly used to address energy efficiency, especially in the context of behavior specialized acceleration. Compared to architecture-specific spatial schedulers, which are the current state-of-the-art, this paper provides a general formulation of spatial scheduling as a constraint-solving problem. We applied this formulation to the behavior specialized accelerators used in this work, as well as two additional whole-program spatial architectures, ran these formulations on a standard ILP solver, and demonstrated such a general scheduler outperforms or matches the respective specialized schedulers, while providing solutions with guaranteed bounds on optimality.

7 CONCLUSION

This dissertation explores a promising alternative direction for continual improvements to general purpose processors: to create specialized hardware engines for broad program properties or behaviors. We have demonstrated that the ExoCore organization, which couples a general purpose processor with accelerators, enables disruptive new tradeoffs to general purpose cores without requiring programmer intervention. Specifically, it can provide up to 2$\times$ performance and 1.7$\times$ energy benefit, depending on the host core. The changes required are practical, requiring only simple integration and well known compilation (or dynamic compilation) techniques.

In addition, this work uncovers several key findings. First, that a small number of exploitable program behaviors can be used to characterize a majority of applications (up to 80% across our workloads). Second, that dataflow architectures become practical and useful in hybrid execution with a general purpose core. Third, that it is possible to attain the benefits of generality and specialization by composing accelerators for synergistic behaviors. Fourth, this paradigm enables disruptive microprocessor tradeoffs, enabling mobile-class processor energy-efficiency with desktop-class performance. Also, this dissertation proposed a novel modeling methodology for behavior-specialized accelerators which enables rapid exploration of the design space. It also developed a mathematical formulation for instruction scheduling on these architectures, which is declarative and simple to use while also providing exact (or bounded-error) solutions.

We conclude by discussing the implications of this work and future research directions.

Implications

Beyond the promising performance potential, the broad impact of this work will be along four dimensions: 1. A broad and natural product applicability, 2. A core paradigm shift making more radical architectures practical, 3. A framework which unifies application, compiler, and microarchitecture modeling, 4. Simplifying compiler design through mathematically based schedulers.

Industry Applicability and Impact A modular core can be used in a variety of settings, besides the simple static compilation plus profiling approach that we evaluated in this work. A natural setting for ExoCore would be to compile for BSAs dynamically from a virtual machine (eg. Java). Alternatively, a purely hardware approach can be taken through hardware-assisted dynamic translation. Products like NVIDIA's Denver already have proven this is practical. Either approach would avoid binary compatibility concerns.

Behavior-specific microarchitecture blocks can be designed and integrated into the core in a modular fashion, without disruptive changes to the core's microarchitecture. This enables designers to easily trade-off performance, area, energy, and design complexity in new ways. In addition, ExoCore can enable effective designs with simple components: an in-order core with four BSAs matches the performance of a quad-issue OOO core, with 15% less area and $2\times$ energy efficiency. The implication of this is that this paradigm provides a rapid path for a design team with a low-performance general core to easily augment it to outperform a much more well-established and complex general core – all without the many years of research and development that it would take to build a monolithic high performance core from scratch.

We note that while this dissertation's analysis and claims hinge upon the ability to add significant hardware and a new ISA and associated compiler, an interesting opportunity along another dimension is whether the same benefits can be achieved without (or with minimal) ISA modifications. The idea would be to apply modifications to a traditional OOO GPP, either at the microarchitecture level or through dynamic translation, to enable efficient execution of nested-loop regions through selectively eschewing instruction-precise recoverability and providing explicit-dataflow execution. This could be the fastest route to short-term industrial impact.

Paradigm Shift in Core Design When the restriction of creating a *monolithic* microprocessor is relaxed, a huge unexplored design space is created in terms of the general core, the set of accelerators, and their integration. Additionally, it grants architects and researchers the freedom to explore radically new designs. To explain, mechanisms that have huge promise for certain workloads, but are impractical in a general context because of power or performance overheads in other workloads, would become the most valuable and sought-after when applied in a behavior specialized microprocessor. This also implies that many of the already-existing techniques in the literature that were overlooked, become exciting potential candidates for inclusion as behavior specialized accelerators.

Methodological Advances Studying programmable accelerators requires cross-layer modifications (compilers, ISAs, microarchitecture), and the established simulation-based approach requires time-consuming development for many pieces of infrastructure. In contrast, the TDG provides a unified space and high-productivity environment for studying the interaction between accelerators, cores, compilers and applications.

Our work has also proposed methods that greatly simplify the development of compilers for behavior specialized architectures – particularly those that expose the routing and placement of their computations. The abstractions we created should easily lend to the automatic exploration of co-designed hardware and compilers inside this paradigm.

Open Questions and Future Directions

This work on behavior specialization opens far more fundamental and practical questions than it answers. We describe a few of the most important ones here.

Untapped Specialization Benefits Though ExoCore showed impressive improvements in performance and energy efficiency, we are far from achieving the benefits of ideal specialization, either in terms of application coverage or total speedup and energy gains. The success of programmer-exposed and domain-specific accelerators should provide an inspiration for where to look for future general purpose gains. At the same time, we should not forget about more irregular, hard-to-specialize programs, where we still require the identification of new exploitable behaviors, and to move away from relying on the strict-specialization of inlined loop nests.

Codesign of Accelerators and General Cores Of course, as we change the accelerators, this affects the choice of the optimal general purpose host core; so just how simple can the general core be? There are also many unanswered questions on what is the most efficient interface between the accelerator and core, especially when accelerator count becomes high, or when accelerators have vastly different memory bandwidth and latency requirements than the core.

Combining Accelerator Mechanisms Beyond including combinations of accelerators, another question arises of how to combine their mechanisms for both hardware efficiency and to reduce complexity. For instance, could a set of data-parallel mechanisms be used to augment SEED to target more regular workloads, completely obviating the need for short-vector SIMD extensions? Or could Trace-P and SEED be merged together to create a unified dataflow accelerator which adapts naturally to the branching behavior by partially speculating different traces?

Improving Design-Space Exploration The above discussion demonstrates how truly vast is the accelerator design space. This means that we require better tools and practices for modeling, compiling, and automating the design-space exploration for acceleration techniques.

In terms of modeling, the TDG is a good start towards simplifying accelerator exploration, but still requires manual implementations of graph transformations. Developing declarative primitives to simplify the TDG-modeling process would be of huge value, but determining a flexible, simple, and powerful abstraction is challenging.

In terms of compilation, our ILP approach has been shown to be effective across a handful of architectures. An open question remains on "universality": what architecture primitives could render our framework ineffective? We suspect that new scheduling abstractions and modifications to the formulation will need to be devised as the number and types of accelerator primitives continues to broaden.

Finally, there is a question of just how automatic can we make this design space exploration. Can we identify new behaviors automatically in programs? Are there correlations between behaviors that suggest pairing microarchitectural features in opportunistic ways? Can we automatically derive the right set of accelerators for given workloads? Answering these questions could prove incredibly useful to mitigate the design cost of future architectures as we require increasing levels of specialization.

Concluding Thoughts

This work demonstrates that a core organized around behavior specialization provides unprecedented improvement, and there is no reason to believe we have reached the limit of what can be accomplished. The benefits obtained in this work were through accelerator designs that were largely well-understood – successive generations of accelerator innovation and refinement can provide factors more improvement. The immediate implication of the modular core design and behavior specialization principles is that the there is *still* a promising path forward for advancing general purpose microprocessors.

A STEPS IN TDG MODEL CONSTRUCTION

Here we discuss the practical aspects and steps in constructing a TDG model.

Analysis The first step is identifying the required compiler analysis or profiling information, and implementing a pass to compute it, operating on the IR or trace respectively. Often, the required information (eg. path profiling) already exists because it is common among BSAs. When this information cannot be computed, approximation may be appropriate.

Transformations The next step is to write an algorithm (a "transform") which reads the incoming μDG trace, and modifies dependences to model the behavior of the BSA in question. Depending on the granularity of acceleration, a transform may operate on a single instruction at a time, or it might collect a basic block, loop iteration, or several iterations before it can decide what the final μDG should be. The modified μDG is transient, and is discarded after any analysis (eg. critical path analysis), once it is no longer inside the current instruction window. A transformation should also include a model for how it interacts with the core when it enters and exists a region.

Scheduling Finally, the model must decide *when* to apply the BSA transform (ie. at what point in the code). In a single-BSA system, the BSA's transform can be used at any legal entry point. For multi-BSA systems, the model should provide per-region performance (or other metric) estimates relative to the general purpose core for the BSA based on either the IR or profiling information. This is used with the Amdahl tree to decide which BSA to use in each region.

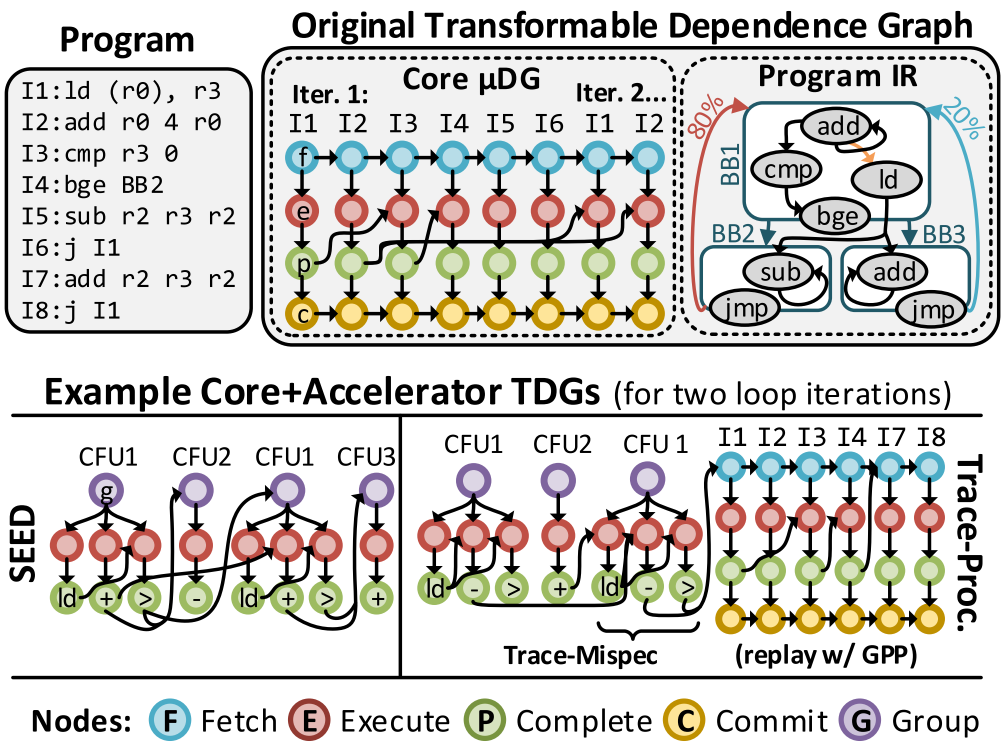
Validating new BSAs Validating a TDG model of newly-proposed BSAs is similar to validating a cycle-level simulator. Writing microbenchmarks and sanity checking is recommended (eg. examining which edges are on the critical path for some code region).

B TDG-MODELING OF SEED AND TRACE-P

Here we briefly discuss the TDG-modeling of SEED and Trace-P. For each accelerator model, we discuss the analysis plan and transforms, and Figure B.1 shows an example code and transform for both BSAs.

Non-speculative Dataflow (SEED) TDG *TDG Analysis:* The primary analysis here is to find fully inlinable loops or nested loops that fit within the hardware budget. Once a loop nest selected, the control is converted into data-dependences by computing the program dependence graph. This determines where "switch" instructions should be inserted to handle control. Instructions are then scheduled onto CFUs using known mathematical optimization [17] techniques.

TDG Transform ($TDG_{GPP,\emptyset}$ to $TDG_{GPP,SEED}$): This transform operates at basic-block granularity, by inserting edges to enforce control dependences and to force each compound instruction to



```
I1:ld (r0), r3
I2:add r0 4 r0
I3:cmp r3 0
I4:bge BB2
I5:sub r2 r3 r2
I6:j I1
I7:add r2 r3 r2
I8:j I1
```

Figure B.1: Example SEED and Trace-P TDGs

wait for all operands before beginning execution. It also adds edges to enforce writeback network capacity. Additionally, edges between the general core and SEED regions are inserted to model live value transfer time.

Trace-Processor (Trace-P) TDG *TDG Analysis:* The analysis plan is a set of eligible and profitable inner loops and compound instruction schedules. Eligible loops with hot traces are found using path profiling techniques [22]. Loops are considered if their loop back probability is higher than 80%, and their configuration size fits in the hardware limit. A similar CFU scheduling technique is used, but compound instructions are allowed to cross control boundaries.

TDG Transform ($TDG_{GPP\text{-}Orig,\emptyset}$ to $TDG_{OOO,Trace\text{-}P}$): This transform is similar to SEED, except that the control dependences are not enforced, and if Trace-P mispeculates the path, instructions are replayed on the host processor by reverting to the $\mathrm{TDG}_{\mathrm{GPP\text{-}Orig},\emptyset}$ to $\mathrm{TDG}_{\mathrm{GPP\text{-}New},\emptyset}$ transform.

BIBLIOGRAPHY

[1] R. Colwell, "The chip design game at the end of moore's law," Hot Chips 2013, 2013.

[2] B. Sutherland, "No moore? a golden rule of microchips appears to be coming to an end," The Economist, 2013.

[3] R. Courtland, "The end of the shrink," *Spectrum, IEEE*, vol. 50, pp. 26–29, November 2013.

[4] I. Cutress in *AnandTech*, 2015. http://www.anandtech.com/show/9483/intel-skylake-review-6700k-6600k-ddr4-ddr3-ipc-6th-generation/4.

[5] J. A. Fisher, P. Faraboschi, and G. Desoli, "Custom-fit processors: Letting applications define architectures," in *Proc. of the 29th Ann. Int'l Symp. on Microarchitecture*, pp. 324–335, December 1996.

[6] D. Jain, A. Kumar, L. Pozzi, and P. Ienne, "Automatically customising vliw architectures with coarse grained application-specific functional units," in *International Workshop on Software and Compilers for Embedded Systems*, pp. 17–32, Springer, 2004.

[7] V. Kathail, S. Aditya, R. Schreiber, B. R. Rau, D. C. Cronquist, and M. Sivaraman, "Pico: automatically designing custom computers," *Computer*, vol. 35, pp. 39–47, Sep 2002.

[8] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning," in *ASPLOS*, 2014.

[9] D. Liu, T. Chen, S. Liu, J. Zhou, S. Zhou, O. Teman, X. Feng, X. Zhou, and Y. Chen, "PuDianNao: a polyvalent machine learning accelerator," in *ASPLOS*, 2015.

[10] J. Brown, S. Woodward, B. Bass, and C. Johnson, "IBM Power Edge of Network Processor: A wire-speed system on a chip," *IEEE Micro*, 2011.

[11] B. C. Brodie, D. E. Taylor, and R. K. Cytron, "A scalable architecture for high-throughput regular-expression pattern matching," in *ISCA*, 2006.

[12] J. V. Lunteren, C. Hagleitner, T. Heil, G. Biran, U. Shvadron, and K. Atasu, “Designing a programmable wire-speed regular-expression matching accelerator,” in *MICRO*, 2012.

[13] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, “Understanding sources of inefficiency in general-purpose chips,” in *ISCA*, 2010.

[14] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan, “Meet the walkers: Accelerating index traversals for in-memory databases,” in *MICRO*, 2013.

[15] 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*, ISCA ’13, (New York, NY, USA), pp. 249–260, ACM, 2013.

[16] 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 *ASPLOS*, 2014.

[17] T. Nowatzki, V. Gangadhar, and K. Sankaralingam, “Exploring the potential of heterogeneous Von Neumann/Dataflow execution models,” in *ISCA*, 2015.

[18] T. Nowatzki and K. Sankaralingam, “Analyzing behavior specialized acceleration,” in *ASPLOS*, 2016.

[19] M. Watkins, T. Nowatzki, and A. Carno, “Software transparent dynamic binary translation for coarse-grain reconfigurable architectures,” in *HPCA*, 2016.

[20] T. Nowatzki, V. Govindaraju, and K. Sankaralingam, “A graph-based program representation for analyzing hardware specialization approaches,” *Computer Architecture Letters*, 2015.

[21] 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 *PLDI*, 2013.

[22] T. Ball and J. R. Larus, “Efficient path profiling,” in *MICRO*, 1996.

[23] V. Govindaraju, T. Nowatzki, and K. Sankaralingam, “Breaking simd shackles with an exposed flexible microarchitecture and the access execute pdg,” in *PACT*, pp. 341–351, 2013.

[24] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz, “Convolution engine: Balancing efficiency and flexibility in specialized computing,” in *ISCA*, 2013.

[25] S. Gupta, S. Feng, A. Ansari, S. Mahlke, and D. August, “Bundled execution of recurring traces for energy-efficient general purpose processing,” in *MICRO*, 2011.

[26] N. Clark, M. Kudlur, H. Park, S. Mahlke, and K. Flautner, “Application-specific processing on a general-purpose core via transparent instruction set customization,” in *MICRO*, 2004.

[27] 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*.

[28] F. Liu, H. Ahn, S. R. Beard, T. Oh, and D. I. August, “Dynaspam: Dynamic spatial architecture mapping using out of order instruction schedules,” in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA ’15, (New York, NY, USA), pp. 541–553, ACM, 2015.

[29] E. S. Chung, J. D. Davis, and J. Lee, “Linqits: Big data on little clients,” in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA ’13, (New York, NY, USA), pp. 261–272, ACM, 2013.

[30] 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 *MICRO*, 2012.

[31] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, “Neural acceleration for general-purpose approximate programs,” in *MICRO*, 2012.

[32] 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*, 2012.

[33] Y. Park, J. J. K. Park, H. Park, and S. Mahlke, "Libra: Tailoring simd execution using heterogeneous hardware and dynamic configurability," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, (Washington, DC, USA), pp. 84–95, IEEE Computer Society, 2012.

[34] S. Liu, Z. Du, J. Tao, D. Han, T. Luo, Y. Xie, Y. Chen, and T. Chen, "Cambricon: An instruction set architecture for neural networks," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[35] A. Sharifian, S. Kumar, A. Guha, and A. Shriraman, "Chainsaw: Von-neumann accelerators to leverage fused instruction chains," in *Proceedings of the 49th International Symposium on Microarchitecture*, 2016.

[36] T. Nowatzki, V. Gangadhan, K. Sankaralingam, and G. Wright, "Pushing the limits of accelerator efficiency while retaining programmability," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[37] B. Fields, S. Rubin, and R. Bodik, "Focusing processor policies via critical-path prediction," in *ISCA*, 2001.

[38] J. Lee, H. Jang, and J. Kim, "Rpstacks: Fast and accurate processor design space exploration using representative stall-event stacks," in *MICRO*, 2014.

[39] 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*, 2011.

[40] T. Moseley, D. Grunwald, D. A. Connors, R. Ramanujam, V. Tovinkere, and R. Peri, “Loopprof: Dynamic techniques for loop detection and profiling,” in *Proceedings of the 2006 Workshop on Binary Instrumentation and Applications (WBIA)*, 2006.

[41] B. Fields, R. Bodik, M. Hill, and C. Newburn, “Using interaction costs for microarchitectural bottleneck analysis,” in *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pp. 228–239, 2003.

[42] 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 *MICRO*, 2009.

[43] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, “Cacti 6.0: A tool to model large caches,” *HP Laboratories*, 2009.

[44] *Parboil Benchmark Suite*. impact.crhc.illinois.edu/parboil/ parboil.aspx.

[45] *Vertical Microbenchmarks*. http://cs.wisc.edu/vertical/microbench.

[46] R. Desikan, D. Burger, and S. W. Keckler, “Measuring experimental error in microprocessor simulation,” in *ISCA*, 2001.

[47] P. A. M. Eric S. Chung, J. C. Hoe, and K. Mai, “Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPUs?,” in *MICRO ’10*.

[48] T. Zidenberg, I. Keslassy, and U. Weiser, “Optimal resource allocation with multiamdahl,” *Computer*, 2013.

[49] M. Hempstead, G.-Y. Wei, and D. Brooks, “Navigo: An early-stage model to study power-contrained architectures and specialization,” in *Proceedings of Workshop on Modeling, Benchmarking, and Simulations (MoBS)*, 2009.

[50] M. Shoaib Bin Altaf and D. Wood, "LogCA: a performance model for hardware accelerators," *Computer Architecture Letters*, 2015.

[51] S. Borkar and A. A. Chien, "The future of microprocessors," *Commun. ACM*, vol. 54, no. 5, pp. 67–77, 2011.

[52] R. Iyer, "Accelerator-rich architectures: Implications, opportunities and challenges," in *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*, 2012.

[53] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," *SIGARCH Comput. Archit. News*, 2011.

[54] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A mechanistic performance model for superscalar out-of-order processors," *ACM Trans. Comput. Syst.*, 2009.

[55] L. Eeckhout, "Computer architecture performance evaluation methods," *Synthesis Lectures on Computer Architecture*, 2010.

[56] C. Nugteren and H. Corporaal, "The boat hull model: adapting the roofline model to enable performance prediction for parallel computing," in *PPOPP*, 2012.

[57] C. Nugteren and H. Corporaal, "A modular and parameterisable classification of algorithms," Tech. Rep. ESR-2011-02, Eindhoven University of Technology, 2011.

[58] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Commun. ACM*, 2009.

[59] S. Hong and H. Kim, "An integrated GPU power and performance model," in *ISCA '10*.

[60] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc, "A performance analysis framework for identifying potential benefits in GPGPU applications," in *PPoPP*, 2012.

[61] J. Meng, V. Morozov, K. Kumaran, V. Vishwanath, and T. Uram, "GROPHECY: GPU performance projection from CPU code skeletons," in *SC'11*, ACM, 2011.

[62] D. Jeon, S. Garcia, C. Louie, and M. B. Taylor, “Kismet: Parallel Speedup Estimates for Serial Programs,” in *OOPSLA*, 2011.

[63] S. Garcia, D. Jeon, C. Louie, and M. B. Taylor, “Kremlin: Rethinking and rebooting gprof for the multicore age,” in *PLDI*, 2011.

[64] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks, “Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures,” in *ISCA*, 2014.

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

[66] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic, “The vector-thread architecture.,” in *Proceedings of the 31st International Symposium on Computer Architecture*, pp. 52–63, June 2004.

[67] 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 *ACM SIGARCH Computer Architecture News*, 2011.

[68] P. Greenhalgh, “Big. little processing with arm cortex-a15 & cortex-a7,” *ARM White Paper*, 2011.

[69] G. Venkatesh, J. Sampson, N. Goulding-hotta, S. K. Venkata, M. B. Taylor, and S. Swanson, “Qscores: Trading dark silicon for scalable energy efficiency with quasi-specific cores,” in *MICRO*, 2011.

[70] M. Hayenga, V. Naresh, and M. Lipasti, “Revolver: Processor architecture for power efficient loop execution,” in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pp. 591–602, Feb 2014.

[71] K. Arvind and R. S. Nikhil, “Executing a program on the mit tagged-token dataflow architecture,” *IEEE Trans. Comput.*, 1990.

[72] 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.

[73] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin, “Wavescalar,” in *MICRO*, pp. 291–, 2003.

[74] M. Mishra, T. J. Callahan, T. Chelcea, G. Venkataramani, S. C. Goldstein, and M. Budiu, “Tartan: evaluating spatial computation for whole program execution,” in *ASPLOS*, 2006.

[75] M. Budiu, P. V. Artigas, and S. C. Goldstein, “Dataflow: A complement to superscalar,” in *ISPASS*, 2005.

[76] 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*.

[77] K. Sankaralingam, R. Nagarajan, R. McDonald, R. Desikan, S. Drolia, M. Govindan, P. Gratz, D. Gulati, H. Hanson, C. Kim, H. Liu, N. Ranganathan, S. Sethumadhavan, S. Sharif, P. Shivakumar, S. W. Keckler, and D. Burger, “Distributed Microarchitectural Protocols in the TRIPS Prototype Processor,” in *MICRO ’06: Proceedings of the 39th Annual International Symposium on Microarchitecture*, December 2006.

[78] 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*, pp. 503–514, 2011.

[79] S. Padmanabha, A. Lukefahr, R. Das, and S. A. Mahlke, “Trace based phase prediction for tightly-coupled heterogeneous cores,” in *MICRO*, 2013.

[80] C. Lee, M. Potkonjak, and W. Mangione-Smith, "MediaBench: a tool for evaluating and synthesizing multimedia and communications systems," in *MICRO*, 1997.

[81] R. A. Iannucci, "Toward a dataflow/von neumann hybrid architecture," in *ISCA*, 1988.

[82] R. Buehrer and K. Ekanadham, "Incorporating data flow ideas into von neumann processors for parallel execution," *Computers, IEEE Transactions on*, 1987.

[83] G. M. Papadopoulos, "Monsoon: an explicit token-store architecture," in *ISCA*, 1990.

[84] Y. Liu and S. Furber, "A low power embedded dataflow coprocessor," in *IEEE Computer Society Annual Symposium on VLSI: New Frontiers in VLSI Design (ISVLSI'05)*, pp. 246–247, May 2005.

[85] C.-H. Ho, S. J. Kim, and K. Sankaralingam, "Efficient execution of memory access phases using dataflow specialization," in *ISCA*, 2015.

[86] D. Gibson and D. A. Wood, "Forwardflow: A scalable core for power-constrained cmps," in *ISCA*, 2010.

[87] H. Singh, M.-H. Lee, G. Lu, F. Kurdahi, N. Bagherzadeh, and E. Chaves Filho, "Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications," *Computers, IEEE Transactions on*, 2000.

[88] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith, "Trace processors," in *MICRO*, 1997.

[89] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, "Single-isa heterogeneous multi-core architectures: The potential for processor power reduction," in *MICRO*, 2003.

[90] R. Kumar, D. M. Tullsen, and N. P. Jouppi, "Core architecture optimization for heterogeneous chip multiprocessors," in *PACT*, 2006.

[91] J. A. Winter, D. H. Albonesi, and C. A. Shoemaker, "Scalable thread scheduling and global power management for heterogeneous many-core architectures," in *PACT*, 2010.

[92] T. Sondag and H. Rajan, “Phase-based tuning for better utilization of performance-asymmetric multicore processors,” in *CGO*, 2011.

[93] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, “Scheduling heterogeneous multi-cores through performance impact estimation (pie),” in *ISCA*, 2012.

[94] S. Padmanabha, A. Lukefahr, R. Das, and S. Mahlke, “Trace based phase prediction for tightly-coupled heterogeneous cores,” in *MICRO*, 2013.

[95] S. Navada, N. K. Choudhary, S. V. Wadhavkar, and E. Rotenberg, “A unified view of non-monotonic core selection and application steering in heterogeneous chip multiprocessors,” in *PACT*, 2013.

[96] S. Srinath, B. Ilbeyi, M. Tan, G. Liu, Z. Zhang, and C. Batten, “Architectural specialization for inter-iteration loop dependence patterns,” in *MICRO*, 2014.

[97] A. Venkat and D. M. Tullsen, “Harnessing isa diversity: Design of a heterogeneous-isa chip multiprocessor,” in *ISCA*, 2014.

[98] A. A. Chien, A. Snavely, and M. Gahagan, “10x10: A general-purpose architectural approach to heterogeneity and energy efficiency,” *Procedia Computer Science*, vol. 4, pp. 1987 – 1996, 2011.

[99] Y. Fang, R. Rasool, D. Vasudevan, and A. A. Chien, “Generalized pattern matching micro-engine,” in *Fourth Workshop on Architectures and Systems for Big Data, Held in conjunction with The 41st International Symposium on Computer Architecture (ISCA 2014)*, IEEE & ACM, 2014.

[100] T. Thanh-Hoang, A. Shambayati, C. Deutschbein, H. Hoffmann, and A. A. Chien, “Performance and energy limits of a processor-integrated fft accelerator,” in *High Performance Extreme Computing Conference (HPEC), 2014 IEEE*, pp. 1–6, Sept 2014.

[101] D. Vasudevan and A. A. Chien, “The bit-nibble-byte microengine (bnb) for efficient computing on short data,” in *Proceedings of the 25th Edition on Great Lakes Symposium on VLSI*, GLSVLSI ’15, (New York, NY, USA), pp. 103–106, ACM, 2015.

[102] Y. Fang, T. T. Hoang, M. Becchi, and A. A. Chien, “Fast support for unstructured data processing: The unified automata processor,” in *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, (New York, NY, USA), pp. 533–545, ACM, 2015.

[103] T. Thanh-Hoang, A. Shambayati, and A. A. Chien, “A data layout transformation (dlt) accelerator: Architectural support for data movement optimization in accelerated-centric heterogeneous systems,” in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1489–1492, March 2016.

[104] T. Nowatzki, M. Ferris, K. Sankaralingam, C. Estan, N. Vaish, and D. Wood, “Optimization and mathematical modeling in computer architecture,” *Synthesis Lectures on Computer Architecture*, vol. 8, no. 4, pp. 1–144, 2013.

[105] “Trips toolchain, http://www.cs.utexas.edu/ trips/dist/,” 2009.

[106] 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*, SIGCOMM ’09, pp. 207–218, 2009.

[107] K. E. Coons, X. Chen, D. Burger, K. S. McKinley, and S. K. Kushwaha, “A spatial path scheduling algorithm for edge architectures,” *SIGARCH Comput. Archit. News*, vol. 34, pp. 129–140, Oct. 2006.

[108] R. Nagarajan, S. K. Kushwaha, D. Burger, K. S. McKinley, C. Lin, and S. W. Keckler, “Static placement, dynamic issue (spdi) scheduling for edge architectures,” in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT ’04, pp. 74–84, 2004.

[109] H. M. Wagner, "An integer linear-programming model for machine scheduling," *Naval Research Logistics Quarterly*, vol. 6, no. 2, pp. 131–140, 1959.

[110] P. Feautrier, "Some efficient solutions to the affine scheduling problem.," *International Journal of Parallel Programming*, vol. 21, pp. 313–347, 1992.

[111] N. Satish, K. Ravindran, and K. Keutzer, "A decomposition-based constraint optimization approach for statically scheduling task graphs with communication delays to multiprocessors," in *DATE '07*, 2007.

[112] S. Amarasinghe, D. R. Karger, W. Lee, and V. S. Mirrokni, "A theoretical and practical approach to instruction scheduling on spatial architectures," tech. rep., MIT, 2002.

[113] A. E. Eichenberger and E. S. Davidson, "Efficient formulation for optimal modulo schedulers," in *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, PLDI '97, pp. 194–205, 1997.

[114] D. W. Engels, J. Feldman, D. R. Karger, and M. Ruhl, "Parallel processor scheduling with delay constraints," in *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, SODA '01, pp. 577–585, 2001.

[115] M. Kudlur and S. Mahlke, "Orchestrating the execution of stream programs on multicore platforms," in *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pp. 114–124, 2008.

[116] K. Fan, H. h. Park, M. Kudlur, and S. o. Mahlke, "Modulo scheduling for highly customized datapaths to increase hardware reusability," in *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '08, (New York, NY, USA), pp. 124–133, ACM, 2008.

[117] J. R. Ellis, *Bulldog: a compiler for vliw architectures*. PhD thesis, 1985.

[118] E. Özer, S. Banerjia, and T. M. Conte, "Unified assign and schedule: a new approach to scheduling for clustered register file microarchitectures," in *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, MICRO 31, pp. 308–315, 1998.

[119] S. S. Battacharyya, E. A. Lee, and P. K. Murthy, *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, 1996.

[120] 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 *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, ASPLOS VIII, pp. 46–57, 1998.

[121] K. Kailas, A. Agrawala, and K. Ebcioglu, "Cars: A new code generation framework for clustered ilp processors," in *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, HPCA '01, pp. 133–, 2001.

[122] M. Mercaldi, S. Swanson, A. Petersen, A. Putnam, A. Schwerin, M. Oskin, and S. J. Eggers, "Instruction scheduling for a tiled dataflow architecture," in *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS XII, pp. 141–150, 2006.

[123] H. Park, K. Fan, S. A. Mahlke, T. Oh, H. Kim, and H.-s. Kim, "Edge-centric modulo scheduling for coarse-grained reconfigurable architectures," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pp. 166–176, 2008.