Technical Report

UCAM-CL-TR-870
ISSN 1476-2986

Number 870

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Accelerating control-flow intensive code in spatial hardware

Ali Mustafa Zaidi

May 2015

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

© 2015 Ali Mustafa Zaidi

This technical report is based on a dissertation submitted February 2014 by the author for the degree of Doctor of Philosophy to the University of Cambridge, St. Edmund's College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Abstract

Designers are increasingly utilizing spatial (e.g. custom and reconfigurable) architectures to improve both efficiency and performance in increasingly heterogeneous systems-on-chip. Unfortunately, while such architectures can provide orders of magnitude better efficiency and performance on numeric applications, they exhibit poor performance when implementing sequential, control-flow intensive code. This thesis studies the problem of improving sequential code performance in spatial hardware without sacrificing its inherent efficiency advantage.

I propose (a) switching from a statically scheduled to a dynamically scheduled, dataflow execution model, and (b) utilizing a newly developed compiler intermediate representation (IR) designed to expose ILP in spatial hardware, even in the presence of complex control flow. I describe this new IR – the Value State Flow Graph (VSFG) – and how it statically exposes ILP from control-flow intensive code by enabling control-dependence analysis, execution along multiple flows of control, as well as aggressive control-flow speculation. I also present a High-Level Synthesis (HLS) toolchain, that compiles unmodified high-level language code to dataflow custom hardware, via the LLVM compiler infrastructure.

I show that for control-flow intensive code, VSFG-based custom hardware performance approaches, or even exceeds the performance of a complex superscalar processor, while consuming only $1/4\times$ the energy of an efficient in-order processor, and $1/8\times$ that of a complex out-of-order processor. I also present a discussion of compile-time optimizations that may be attempted to further improve both efficiency and performance for VSFG-based hardware, including using alias analysis to statically partition and parallelize memory operations.

This work demonstrates that it is possible to use custom and/or reconfigurable hardware in heterogeneous systems to improve the efficiency of frequently executed sequential code, without compromising performance relative to an energy inefficient out-of-order superscalar processor.

Acknowledgements

First and foremost, my sincerest thanks to my supervisor, David Greaves, for his careful and invaluable guidance during my PhD, especially for challenging me to explore new and unfamiliar topics and broaden my perspective. My heartfelt thanks also to Professor Alan Mycroft for his advice and encouragement as my second advisor, as well as for his enthusiasm for my work. I would also like to acknowledge Robert Mullins for his encouragement, many insightful conversations, and especially for running the CompArch reading group. Sincere thanks also to Professor Simon Moore, for seeding the novel and exciting Communication-centric Computer Design project, and for making me a part of his amazing team.

I would also like to acknowledge my colleagues for making the Computer Laboratory a fun and engaging workplace, and especially for all of the illuminating discussions on diverse topics, ranging from computer architecture, to the Fermi paradox, to the tractability of magic roundabouts. In particular, I would like to thank Daniel Bates, Alex Bradbury, Andreas Koltes, Alan Mujumdar, Matt Naylor, Robert Norton, Milos Puzovic, Charlie Reams, and Jonathan Woodruff.

My gratitude is also due to my friends at St. Edmund's College, as well as around Cambridge, for helping me to occasionally escape from work to try something less stressful, like running a political discussion forum! In particular, my heartfelt thanks go the co-founders of the St. Edmund's Political Forum, as well as to my friends Parul Bhandari, Taylor Burns, Ali Khan, Tim Rademacher, and Mohammad Razai.

Last but not least, I am most grateful to my wife Zenab for her boundless patience and support during my PhD, to my son Mahdi for bringing both meaning and joy to the life of a humble student, and to my parents for their unwavering faith, encouragement and support.

Contents

1 Introduction **11**
1.1 Thesis Statement 13
1.2 Contributions 14
1.3 Publications and Awards 15

2 Technical Background **17**
2.1 The Uniprocessor Era 17
2.2 The Multicore Era 20
2.3 The Dark Silicon Problem 21
2.3.1 Insufficient Explicit Parallelism 21
2.3.2 The Utilization Wall 23
2.3.3 Implications of Dark Silicon 24
2.4 The *Spatial* Computation Model 26
2.4.1 Advantages of Spatial Computation 26
2.4.2 Issues with Spatial Computation 28
2.4.3 A Brief Survey of Spatial Architecture Research 31
2.5 Summary 36

3 Statically Exposing ILP from Sequential Code **39**
3.1 The Nature of Imperative Code 39
3.2 Exposing ILP from Imperative Code 42
3.2.1 False or Name dependencies 42
3.2.2 Overcoming Control Flow 44
3.2.3 Pointer Arithmetic and Memory Disambiguation 46
3.3 The Superscalar Performance Advantage 47
3.3.1 Case Study 1: Outer-loop Pipelining 49
3.4 Limitations of Superscalar Performance 52
3.4.1 Case Study 2: Multiple Flows of Control 52
3.5 Improving Sequential Performance forSpatial Hardware 53
3.5.1 Why the *Static* Dataflow Execution Model? 54
3.5.2 Why a VSDG-based compiler IR? 55
3.6 Overcoming Control-flow with the VSDG 55
3.6.1 Defining the VSDG 55
3.6.2 Revisiting Case Studies 1 and 2 64
3.7 Related Work on Compiler IRs 70
3.8 Summary 72

4 Definition and Semantics of the VSFG **75**
4.1 The VSFG as Custom Hardware . . . 75
4.2 Modeling Execution with Petri-Nets . . . 77
4.2.1 Well-behavendess in Dataflow Graphs . . . 79
4.3 Operational Semantics for the *VSFG-S* . . . 82
4.3.1 Semantics for Basic Operations . . . 84
4.3.2 Compound Operations: Nested Acyclic Subgraphs . . . 89
4.3.3 Compound Operations: Nested Loop Subgraphs . . . 92
4.4 Comparison with Existing Dataflow Models . . . 100
4.4.1 Comparison with Pegasus . . . 100
4.4.2 Relation to Original Work on Dataflow Computing . . . 101
4.5 Limitations of Static Dataflow Execution . . . 103
4.6 Summary . . . 107

5 A VSFG-Based High-Level Synthesis Toolchain **109**
5.1 The Toolchain . . . 109
5.2 Conversion from LLVM to VSFG-S . . . 111
5.2.1 Convert Loops to Tail-Recursive Functions . . . 111
5.2.2 Implement State-edges between State Operations . . . 116
5.2.3 Generate Block Predicate Expressions . . . 117
5.2.4 Replace each ϕ-node with a MUX . . . 119
5.2.5 Construct the VSFG-S . . . 119
5.3 Conversion from VSFG-S to Bluespec . . . 120
5.4 Current Limitations . . . 122
5.5 Summary . . . 124

6 Evaluation Methodology and Results **125**
6.1 Evaluation Methodology . . . 125
6.1.1 Comparison with an Existing HLS Tool . . . 126
6.1.2 Comparison with Pegasus/CASH . . . 126
6.1.3 Comparison with Conventional Processors . . . 127
6.1.4 Selected benchmarks: . . . 128
6.2 Results . . . 132
6.2.1 Cycle Counts . . . 132
6.2.2 Frequency and Delay . . . 139
6.2.3 Resource Requirements . . . 141
6.2.4 Power and Energy . . . 144
6.3 Estimating ILP . . . 150
6.4 Summary . . . 151

7 Conclusions and Future Work **153**
7.1 Future Work . . . 154
7.1.1 Incremental Enhancements . . . 154
7.1.2 Mitigating the Effects of Dark Silicon . . . 154

Bibliography **170**

CHAPTER 1

Introduction

Over the past two decades, pervasive, always-on computing services have become an integral part of our lives. Not only are we using increasingly portable devices like tablets and smartphones, there is also an increasing reliance on *cloud* computing: server-side computation and services, like web search, mail, and social media. On the client side, there is an ever growing demand for increased functionality and diversity of applications, as well as an expectation of continued performance scaling with every new technology generation.

Designers incorporate increasingly powerful processors and systems-on-chip into such devices to meet this demand. However, the key trade-off in employing high-performance processors is the high energy cost they incur [GA06]: for increasingly portable devices, in addition to the demand for ever higher performance, users have an expectation of a minimum time that their battery should last under normal use.

On the server-side, power dissipation and cooling infrastructure costs are growing, currently accounting for more than 40% of the running costs for datacenters [Ham08], and around 10% of the total lifetime cost [KBPS09]. To meet the growing demand for computational capacity in datacenters, computer architects are striving to develop processors capable of not only providing higher throughput and performance, but also achieving high energy efficiency.

Unfortunately, for the past decade, architects have had to struggle with several key issues that hinder their ability to continue scaling performance with Moore's Law, while also improving the energy efficiency of computation. Poor wire-scaling, together with the need to limit power dissipation and improve energy efficiency, have driven a push towards ever more decentralized, modular, multicore processors that rely on explicit parallelism for performance instead of frequency scaling and increasingly complex uniprocessor microarchitectures.

Instead of the dynamic, run-time effort of exposing and exploiting parallelism in increasingly complex processors, in the multicore era the responsibility of exposing further parallelism to scale performance rests primarily with the programmer. Nevertheless, despite the increased programming costs and complexity, performance has continued to scale for application domains that have abundant, easy-to-express parallelism, in particular for server-side applications such as web and database servers, scientific and high-performance computing, etc.

The Dark Silicon Problem: On the other hand, for client-side, general-purpose applications, performance scaling on explicitly parallel architectures has been severely limited due to Amdahl's Law, as such applications exhibit limited coarse-grained (data or task-level) parallelism that could be cost-effectively exposed by a programmer [BDMF10].

Furthermore, more recently, a new issue has been identified that limits performance scaling on multicore architectures, even for applications with abundant parallelism: due to the end of Dennard scaling [DGnY+74], on-chip power dissipation is growing in proportion to the number of on-chip transistors, meaning that for a fixed power budget, the proportion of on-chip resources that can be actively *utilized* at any given time decreases with each technology generation. This problem is known as the *Utilization Wall* [VSG+10].

Together, the utilization wall and Amdahl's law problems lead to the issue of *Dark Silicon*, where a growing fraction of on chip resources will have to remain switched off, either due to power dissipation constraints, or simply because of insufficient parallelism in the application itself. A recent study has shown that with future process generations, even as Moore's Law provides a 32× increase in on-chip resources, dark silicon will limit effective performance scaling to only about $3-8\times$ [EBSA+11].

The Potential of *Spatial* Computation: To mitigate the effects of the utilization wall, it is essential to make the most efficient use of the fraction of transistors that can be active at any given time. Architects are doing exactly this as they build increasingly heterogeneous systems incorporating *spatial* computation hardware such as custom or reconfigurable logic[1]. Unlike conventional processors, spatial hardware relegates much of the effort of exposing and exploiting concurrency to the compiler or programmer. Spatial hardware is also highly specialized, tailored to the specific application being implemented, thereby providing orders-of-magnitude improvements in energy efficiency and performance [HQW+10].

Examples of such hardware include video codecs and image processing datapaths implemented as part of heterogeneous systems-on-chip commonly used in modern smartphones and tablets. By implementing specialized hardware designed for a small subset of tasks, architects essentially trade relatively inexpensive and abundant transistor resources for essential improvements in energy-efficiency.

Current Limitations of *Spatial* Computation: To mitigate the effects of Amdahl's Law and continue scaling performance with Moore's law, it is essential to also aggressively exploit implicit fine-grained parallelism from otherwise sequential code, and to do so with high energy efficiency to avoid running into the utilization wall. Recent work has attempted to implement sequential, general-purpose code using spatial hardware, in order to improve energy efficiency [VSG+10, BVCG04]. Unfortunately, sequential code exhibits poor performance in custom hardware, meaning that for performance scaling under Amdahl's Law, architects must employ conventional, complex, and energy-inefficient out-of-order processors [BAG05].

[1] Unlike the *temporal* execution model of conventional processors, wherein intermediate operands are communicated between operations through a centralized memory abstraction such as a register file, *spatial* computation utilizes a point-to-point interconnect to communicate intermediate operands directly between producing and consuming processing elements. Consequently, unlike with conventional processors, *placement/mapping* of operations to processing elements must be determined before program execution. Spatial Computation is described in greater detail in Section 2.4.

Not only does this affect usability by reducing the battery life of portable devices, it also means that overall performance scaling would be further limited due to the utilization wall limiting the amount of parallel processing resources that can be activated within the remaining power budget. To overcome this Catch-22 situation, it is essential that new approaches be found to implement such sequential code with high performance, without incurring the energy costs of conventional processors.

This dissertation focuses on combining the high energy efficiency of spatial computation, with the high sequential-code performance of conventional superscalar processors. Success in this endeavour should have a significant positive impact on a diverse range of computational domains in different ways.

For instance, embedded systems would be able to sustain higher performance within a given power budget, potentially also reducing effort required to optimize code. For example, the primary energy consumption in a smartphone is typically not due to the application processor. Instead subsystems like high-resolution displays, or radio signalling and processing consume a majority of the power budget. As a result, even an order of magnitude improvement in computational efficiency would not significantly affect how frequently a user is expected to charge their phone. However, the increased efficiency could instead be utilized to undertake more complex computation within the same power budget, perhaps to provide a better user experience.

Conversely, cloud and datacenter infrastructure could directly take advantage of the increased efficiency to reduce energy costs. As the key reasons for the high energy cost in server-side systems are (a) power consumed by processors, and (b) the cooling infrastructure needed to dissipate this power, more efficient processing elements would simultaneously reduce the operating costs due to both of these factors without compromising computational capacity.

1.1 Thesis Statement

My main thesis is that by statically overcoming the limitations on fine-grained parallelism due to control-flow, the sequential code performance of energy-efficient spatial architectures can be improved to match or even exceed the performance of dynamic, out-of-order superscalar processors, without incurring the latters' energy cost.

To achieve this, this dissertation focuses on the development of a new compiler intermediate representation that accelerates control-intensive sequential code by enabling aggressive speculative execution, control-dependence analysis, and exploitation of multiple flows of control in spatial hardware. In order to demonstrate my thesis, this dissertation is structured as follows:

- **Chapter 2:** A brief overview of the energy and performance issues faced by computer architects is presented, followed by an introduction to *spatial* computation, along with a brief survey of existing spatial architectures, demonstrating the current issues with sequential code performance.

- **Chapter 3:** I study the key underlying reasons for the performance advantage of complex, out-of-order superscalar processors over spatial hardware when implementing general-purpose sequential code. The goal being to understand how to

overcome these limitations without compromising the inherent energy-efficiency of spatial hardware.

- **Chapter 4:** I then develop a new compiler intermediate representation called the Value State Flow Graph that simplifies the static exposition of fine-grained instruction level parallelism from control-flow intensive sequential code. The VSFG is designed so that it can be used as an intermediate representation for compiling to a wide variety of spatial architectures and substrates, including a direct implementation as application-specific custom hardware.

- **Chapter 5:** A high-level synthesis toolchain using the VSFG representation is developed that allows the compilation of high-level language code to high performance custom hardware.

- **Chapter 6:** Finally, results from benchmarks compiled using this toolchain demonstrate that in most cases, the performance of the generated custom hardware matches, or even exceeds the performance of a complex superscalar processor, while incurring a fraction of its energy cost. I highlight the fact that performing compile-time optimizations on the VSFG can easily improve both performance and energy-efficiency even further.

Chapter 7 concludes the dissertation, and highlights some areas for future research in the area of spatial architectures and compilers.

1.2 Contributions

This thesis makes the following contributions:

- A new low level compiler intermediate representation (IR), called the Value State Flow Graph (VSFG) is presented, that exposes ILP from sequential code even in the presence of complex control flow. It achieves this by enabling aggressive control-flow speculation, control dependence analysis, as well as execution along multiple flows of control. As conventional processors are typically unable to take advantage of the last two features, the VSFG can potentially expose far greater ILP from sequential code [LW92].

- The VSFG representation is also designed to be directly implementable as custom hardware, replacing the traditionally used CDFG (Control-Data Flow Graph) [NRE04]. The VSFG is defined formally, including the development of eager (dataflow) operational semantics. A discussion of how the VSFG compares to existing representations of dataflow computation is also presented.

- To test this new IR, a new high-level synthesis (HLS) tool-chain has been implemented, that compiles from the LLVM IR to the VSFG, then implements the latter as a hardware description in Bluespec SystemVerilog [Nik04]. Unlike the statically-scheduled execution model of traditional custom hardware [CM08], I employ a dynamically-scheduled static-dataflow execution model for our implementation [Bud03, BVCG04], allowing for better tolerance of variable latencies and statically unpredictable behaviour.

- Custom hardware generated by this new tool-chain is shown to achieve an average speedup of 1.55× (max 4.05×) over equivalent hardware generated by LegUp, an established CDFG-based high-level synthesis tool [CCA+11]. Furthermore, VSFG-based hardware is able to approach (in some cases even improve upon) the cycle-counts of an Intel Nehalem Core i7 processor, on control-flow intensive benchmarks. While this performance incurs an average 3× higher energy cost than LegUp, the VSFG-based hardware's energy dissipation is still only 1/4× that of a highly optimized in-order Altera Nios II/f processor (and 1/8× that of a Core i7-like out-of-order processor).

- I provide recommendations for how both the energy efficiency and performance of our hardware may be further improved by implementing simple compiler optimizations, such as performing alias-analysis to partition and parallelize memory accesses, as well as how to reduce the energy overheads of speculation.

1.3 Publications and Awards

- **Paper (to appear):** Ali Mustafa Zaidi, David Greaves, "A New Dataflow Compiler IR for Accelerating Control-Intensive Code in Spatial Hardware", 21st Reconfigurable Architectures Workshop (RAW 2014), associated with the 28th Annual International Parallel and Distributed Processing Symposium (IPDPS 2014), May 2014, Phoenix, Arizona, USA.

- **Poster:** Ali Mustafa Zaidi, David Greaves, "Exposing ILP in Custom Hardware with a Dataflow Compiler IR", The 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT 2013), September, 2013, Edinburgh, UK.
  - **Award:** Awarded Gold Medal at the PACT 2013 ACM Student Research Competition.

- **Paper:** Ali Mustafa Zaidi, David Greaves, "Achieving Superscalar Performance without Superscalar Overheads – A Dataflow Compiler IR for Custom Computing", The 2013 Imperial College Computing Students Workshop (ICCSW'13), Septermber 2013, London, UK.

- **Award:** Qualcomm Innovation Fellowship 2012, Cambridge, UK. Awarded for research proposal titled: "Mitigating the Effects of Dark Silicon".

CHAPTER 2

Technical Background

This chapter presents a brief history of computer architecture, highlighting the technical and design challenges architects have faced previously, as well as those that must be addressed today, such as dark silicon. I establish the need for achieving both high *sequential* performance, as well as much higher energy efficiency, in order to mitigate the effects of dark silicon. This chapter also presents a survey of prior work on *spatial computation*, establishing its scalability, efficiency and performance advantages for the numeric application domain, as well as its shortcomings with respect to implementing and accelerating sequential code. This dissertation attempts to overcome these shortcomings with the development of a new dataflow compiler intermediate representation which will be discussed in Chapter 3, and described formally in Chapter 4.

2.1 The Uniprocessor Era

For over two decades, Moore's Law enabled exponential scaling of uniprocessor performance. Computer architects used the ever growing abundance of on-chip resources to build increasingly sophisticated uniprocessors that operated at very high frequencies. Starting in the mid 1980s, uniprocessor performance improved by three orders of magnitude, at approximately 52% per year (Figure 2.1, taken from [HP06]), until around 2004. Of this, two orders of magnitude can be attributed to improvements in fabrication technology leading to higher operating frequencies, while the remaining $10\times$ improvement is attributed to microarchitectural enhancements for dynamically exposing and exploiting fine-grained instruction level parallelism (ILP), enabled by an abundant transistor budget [BC11].

Programmers would code using largely sequential programming models, while architects utilized ever more complex techniques to maximize ILP: incorporating deeper pipelining, superscalar as well as out-of-order execution to accelerate true dependences, register renaming to overcome false dependences, as well as aggressive branch prediction and misspeculation recovery mechanisms to overcome control dependences.

While the benefits of explicit parallel programming were known due to extensive work done in the high-performance and supercomputing domains [TDTD90, DD10], there was little incentive for programmers to utilize explicit parallelism in the general-purpose computing domain, since uniprocessor performance scaling effectively provided a 'free lunch': doubling observed performance every 18 months, with no effort required on the part of the

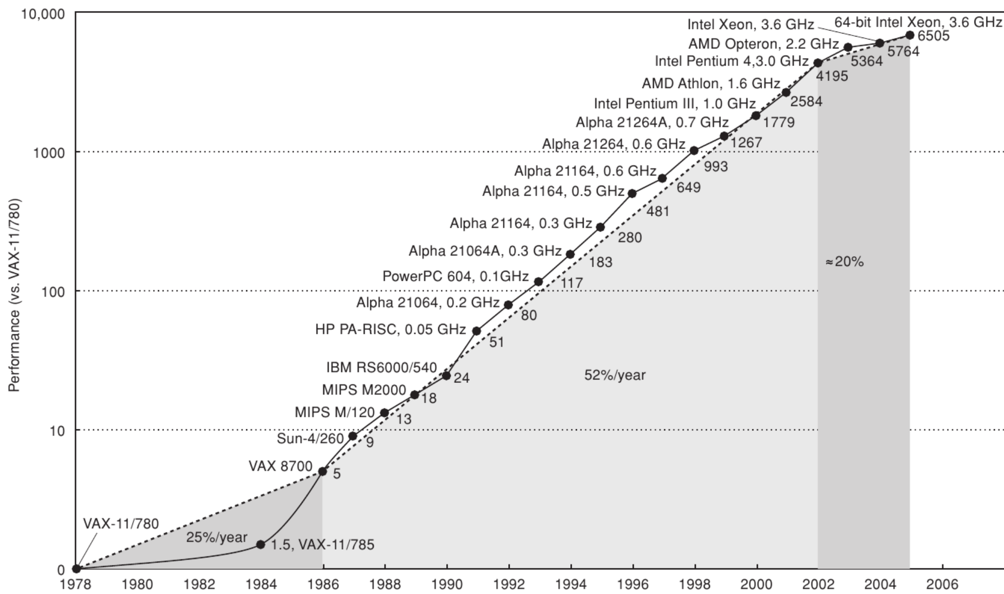


Figure 2.1: Uniprocessor Performance Scaling from 1978 to 2006. Figure taken from [HP06]

programmer [Sut05]. Thus all the 'heavy lifting' of exposing and exploiting concurrency was left to the microarchitecture level of abstraction, leading to increasingly complex mechanisms for instruction stream management at the microarchitecure level.

Utilizing the exponentially growing on-chip resources to develop evermore complicated uniprocessors ultimately proved to be unsustainable. Around 2004, this trend came to an end due to the confluence of several issues, commonly known as the ILP, Power, Memory, and Complexity Walls [OH05].

1. **The ILP Wall:** In the early 1990s, limit studies carried out by David Wall [Wal91] and Monica Lam [LW92] determined that, with the exception of *numeric* applications[1], the amount of ILP that can be dynamically extracted from a sequential instruction stream by a uniprocessor is fundamentally limited to about 4-8 instructions per cycle (IPC).

   Lam noted that this ILP Wall is not due to reaching the limit of available ILP in the code at runtime, but rather because control-flow remains a key performance bottleneck despite aggressive branch prediction, particularly as uniprocessors are limited to exploiting ILP by speculatively executing independent instructions from a *single flow of control*. By enabling the identification of multiple independent regions of code through control dependence analysis, and then allowing their concurrent execution (i.e. exploiting *multiple flows of control*), Lam observed that ILP could again be increased by as much as an order of magnitude in the limit [LW92].

2. **The Memory Wall:** As transistor dimensions shrank, both processor and DRAM clock rates improved exponentially, but the rate of improvement for processors far outpaced that for main memory. This meant that the cycle latency for accessing main memory grew exponentially [WM95]. This issue was mitigated to an extent through the use of larger last-level caches and deeper cache hierarchies, but at a

[1]Applications with abundant data level parallelism, and often regular, predictable control-flow. Examples include signal processing, compression, and multimedia.

significant area cost: the fastest processors dedicated as much as half of total die area to caches. Despite this, it was expected that DRAM access latency would ultimately become the primary performance bottleneck to performance, given the historic reliance on ever higher clock rates for uniprocessor performance improvement.

3. **The Complexity Wall:** While transistor dimensions and performance have scaled with Moore's Law, the performance of wires has diminished as feature sizes shrink. This is partly due to the expectation that each process generation will enable higher frequency operation, so the distance that signals can propagate in a single clock cycle is reduced [AHKB00]. Furthermore, narrower, thinner and more tightly packed wires exhibit higher resistance (R) and capacitance (C) per unit length, and thus increased signaling delay [HMMH01].

   Uniprocessors have heavily relied on monolithic, broadcast resource abstractions such as centralized register files, and broadcast buses, in their designs, primarily in order to maintain a unified program state and support precise exceptions. However, such resources scale poorly when increased performance is required [ZK98, TA03].

   With poor wire scaling limiting clock rate improvements, together with the ILP wall limiting improvements in IPC, designers observed severely diminishing returns in overall performance, even as design complexity, costs and effort continued to grow [BMMR05, PJS97].

4. **The Power Wall:** With each process generation, Moore's law enables a quadratic growth in the number of transistors per unit area, as well as allowing these transistors to operate at higher clock rates. For a given die size, this represents a significant increase in the number of circuit elements that can switch per unit time, potentially increasing power dissipation proportionally. Thankfully, total power dissipation per unit area could be kept constant thanks to Dennard scaling [DGnY+74], which posits that both per-transistor load capacitance and supply voltage can be lowered each generation. (This is described in more detail in Section 2.3.2).

   However, Dennard scaling did not take into account the increased complexity of newer processor designs, as well as the poor scaling of wires, both of which contributed to an overall increase in power dissipation. Furthermore, as transistor dimensions shrink and supply voltage (and therefore threshold voltage) are reduced, there is an exponential increase in leakage current, and therefore relative static power dissipation increases as well [BS00]. Until about 1999, static power remained a small fraction of total power dissipation, but was becoming increasingly more severe with each process generation [TPB98].

   A combination of these factors has meant that the total power dissipation of uniprocessor designs continued to grow to such an extent that chip power densities began to approach those of nuclear reactor cores [Pol99].

Due to the ILP, Memory, Complexity, and Power Walls, further scaling of performance could no longer be achieved by simply relying on faster clock rates and increasingly complex uniprocessors. Ending frequency scaling became necessary to keep memory access latency and architectural complexity from worsening, as well as to compensate for power increases due to leakage, poor wire scaling, and growing microarchitectural complexity

with successive process generations. This meant that further performance scaling would have to rely solely on the increased exploitation of concurrency.

In order to overcome the ILP Wall and improve IPC, processors would need to exploit parallelism from multiple, independent regions of code. Exploitation of more coarse-grained and/or more explicit concurrency became essential, but must be achieved without further increasing design complexity. To address poor wire scaling and the complexity wall, decentralized, modular, highly scalable architectures must be devised, so that the worst-case wire lengths do not have to scale with the amount of resources available. Instead of relying on the simple unified abstraction provided by non-scalable centralized memory or broadcast interconnect structures, cross-chip communication must now be explicitly managed between modular components.

Since 2004, computer architecture has developed into two distinct directions. Multicore architectures are primarily utilized for the *general-purpose* computing domain, that includes desktop and server applications. Alternatively, *spatial* architectures, discussed in Section 2.4 are increasingly being utilized to accelerate numeric, data-intensive applications, particularly in situations where both high performance and/or high energy efficiency are required.

2.2 The Multicore Era

The need for modularity and explicit concurrency was answered with an evolutionary switch to multicore architectures. Instead of having increasingly complex uniprocessors, designers chose to implement multiple copies of conventional processors on the same die. In most cases, architects relied on the shared-memory programming model to enable programmers to write parallel code, as this model was seen as an extension of the Von-Neumann architecture that programmers were already familiar with.

Wire scaling and complexity issues were mitigated thanks to the modular nature of multicore design, while the memory wall was addressed by ending frequency scaling. The ILP Wall would be avoided by relying on explicitly parallel programming to identify and execute 'multiple flows of control' organised into threads, communicating and synchronizing via shared memory. Ideally, the Moore's Law effect of exponential growth in transistors would then be extended into an exponential growth in number of cores on chip.

Multicore processors are able to provide high performance scaling for *embarrasingly parallel* application domains that have abundant data or task level parallelism. This is often facilitated with the help of domain-specific programming models like MapReduce [RRP+07], that are used for web and database servers and other datacenter applications, or OpenCL [LPN+13], which is useful for accelerating highly numeric applications such as games, or multimedia and signal processing[2].

However, for many non-numeric, *consumer-side* applications, performance scaling on multicore architectures has proven far more difficult. Such applications are characterized by low data or task parallelism, complex and often data-dependent control-flow, and irregular memory access patterns[3]. Previously, programmers had relied on the fine-grained

[2]Graphics Processors or GPUs can be considered a highly specialized form of multicore processor, designed to accelerate such data-parallel applications.

[3]In this thesis, I refer to such code as belonging to the *client-side*, *consumer*, or *general-purpose* application domains, or simply as *sequential* code.

ILP exploitation capabilities of out-of-order superscalar processors to achieve high performance on such code [SL05].

The shared memory programming model has proven to be very difficult for programmers to utilize in this domain, particularly when constrained to exploiting such fine-grained parallelism [HRU+07]. Programmers are required to not only explicitly expose concurrency in their code by partitioning it into threads, but also to manually manage communication and synchronization between threads at run-time. The shared-memory threaded programming model is also highly non-deterministic, since the programmer is largely unaware of the order in which concurrent threads will be scheduled, and hence alter shared state, at runtime. This non-determinism further increases the complexity of debugging such applications, as observed behavior may change with each run of the application [Lee06].

Thus, despite the decade-old push towards multicore architectures, the degree of threaded parallelism in consumer workloads remains very low. Blake et al. observed that over a period of 10 years, the number of concurrent threads in non-numeric applications has been limited to about two [BDMF10]. As a result, performance scaling for such applications remains far below what users have come to expect over the past decades. In part due to this insufficient explicit parallelism in applications, a new threat to continued performance scaling with Moore's Law has recently been identified, called Dark Silicon.

2.3 The Dark Silicon Problem

Despite ongoing exponential growth of on-chip resources with Moore's Law, the performance scalability of future designs will be increasingly restricted. This is because the total *usable* on-chip resources will be growing at a much slower rate. This problem is known as 'Dark Silicon', and is caused by two factors [EBSA+11]:

1. Amdahl's Law and performance saturation due to insufficient explicit parallelism, and

2. the end of Dennard Scaling, together with limited power budgets leading to the *Utilization Wall*.

This section describes these issues in more detail, and discusses current strategies for addressing them.

2.3.1 Insufficient Explicit Parallelism

Amdahl's Law: As mentioned in the last section, the general-purpose, or consumer application domain exhibits low degrees of parallelism. Amdahl's Law [Amd67] governs the performance scaling of parallel applications by considering them composed of a parallel and a sequential fraction, and states that for applications with insufficient parallelism, achievable speedup will be strictly constrained by the performance of the sequential fraction.

$$Perf(f, n, S_{seq}, S_{par}) = \frac{1}{\frac{(1-f)}{S_{seq}} + \frac{(f)}{n.S_{par}}} \tag{2.1}$$

A generalized form of Amdahl's Law for multicore processors is shown in equation 2.1 (adapted from [HM08]), where f is the fraction of code that is perfectly parallel, thus $(1-f)$ is the fraction of sequential code. S_{seq} is the speedup that a particular architecture provides for the sequential portion of code, n is the number of parallel processors, and S_{par} is the speedup each parallel processor provides when executing a thread from the parallel region of code.

Figure 2.2 shows a plot of the relative speedup of a machine with high S_{seq}, versus a machine with low S_{seq}, as f and n are varied (assume $S_{par} = 1$ for both machines). The vertical axis is the ratio of speedup of a machine with $S_{seq} = 4$ to a machine with $S_{seq} = 1$. Figure 2.2 shows that even with moderate amounts of parallelism ($0.7 \leq f \leq 0.9$), overall speedup is highly dependent on the speedup of the sequential fraction of code even as the number of parallel threads is increased. Thus achieving high sequential performance through dynamic exploitation of implicit, instruction-level parallelism remains important for scaling performance with Moore's Law. However, this must be achieved without again running into the ILP, Complexity, Power and Memory Walls.

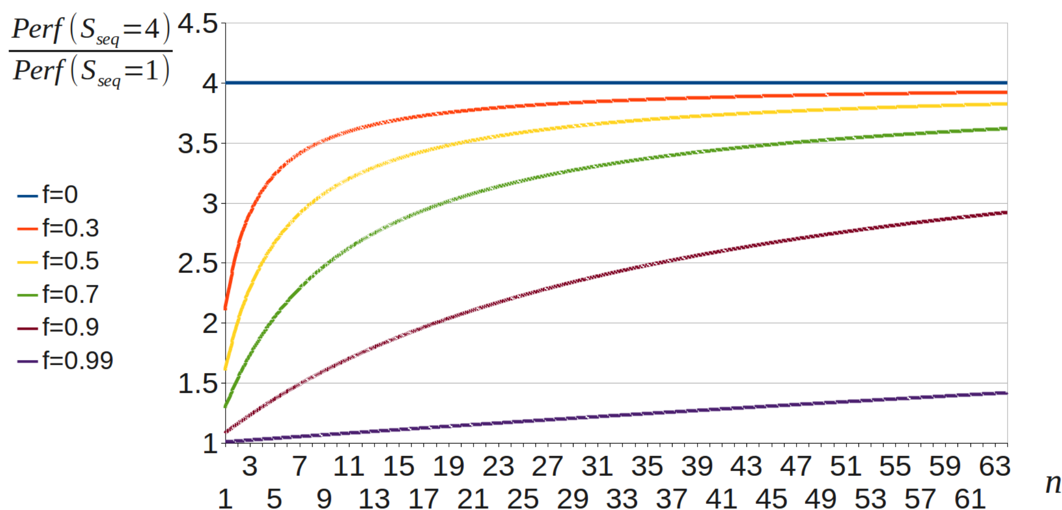


Figure 2.2: Plot showing the importance of sequential performance to overall speedup. The y-axis measures the *ratio* of performance between two machines, one with high sequential performance ($S_{seq} = 4$), vs. one with low sequential performance ($S_{seq} = 1$), with all other factors being identical.

A more comprehensive analysis of multicore speedups under Amdahl's Law is presented by Hill and Marty [HM08]. They consider various configurations of multicore processors given a fixed resource constraint: fewer coarse-grained cores, many fine-grained cores, as well as asymmetric and *dynamic* multicore processors. They find that while performance scaling is still limited by the sequential region, the best potential from speed-up arises from the dynamic multicore configuration, where many smaller cores may be combined into a larger core for accelerating sequential code, assuming minimal overheads for such reconfiguration. It is important to note that theirs is a highly optimistic analysis, as it assumes that sequential performance can be scaled indefinitely (proportional to $\sqrt{n}$, where n is the number of execution resources per complex processor), whereas Wall [Wal91] notes a limit to ILP scaling for conventional processors.

Esmaelzadeh et al. identifed insufficient parallelism in applications as the primary

source of dark silicon [EBSA+11]: with the sequential fraction of code limiting overall performance, most of the exponentially growing cores will remain unused unless the degree of parallelism can be dramatically increased.

Brawny **Cores vs.** ***Wimpy*** **Cores:** Even for server and datacenter applications that exhibit very high parallelism, and thus are less susceptible to being constrained by sequential performance, per-thread sequential performance remains essential [H10]. This is because of a variety of practical concerns not considered under Amdahl's Law – the explicit parallelization, communication, synchronization and runtime scheduling overheads of many fine-grained threads can often negate the area and efficiency advantages of *wimpy*, or energy-efficient cores. Consequently, it is often better for overall cost and performance to have fewer threads running on fewer *brawny* cores than to have a fine-grained manycore in most cases [LNC13].

Add to this the fact that a vast amount of legacy code remains largely sequential, we find that achieving high sequential performance will remain critical for performance scaling for the forseeable future. Unfortunately, currently the only means of achieving high performance on general-purpose sequential code is through the use of complex, energy inefficient out-of-order superscalar processors.

2.3.2 The Utilization Wall

The average power dissipation of CMOS circuits is given by equation 2.2, where n is the total number of transistors, α is the average activity ratio for each transistor, C is the average load capacitance, V_{DD} is the supply voltage, and f is the operating frequency. P_{static} represents static, or leakage power dissipation that occurs independently of any switching activity, while $I_{leakage}$ is the leakage current, and k_{design} is a constant factor. This model for P_{static} is taken from [BS00].

$$P_{total} = P_{dynamic} + P_{static} = n.\alpha.C.V_{DD}^2.f + n.V_{DD}.I_{leakage}.k_{design} \tag{2.2}$$

The effect of Moore's Law and Dennard Scaling on power dissipation is described as a first-order approximation in [Ven11], and is adapted and briefly summarized here:

> If a new process generation allows transistor dimensions to be reduced by a scaling factor of S (i.e. transistor width and length are both reduced by $1/S$, where $S > 1$), then the number of transistors on chip (n) grows by S^2, while operating frequency (f) also improves by S. This implies that the total switching activity per unit time should increase by S^3, for a fixed die size. However, chip power dissipation would also increase by S^3.
>
> In 1974, Robert Dennard observed that not only does scaling transistor dimensions also scale its capacitance (C) by $1/S$, but that it is also possible to scale V_{DD} by the same factor [DGnY+74]. This meant that $P_{dynamic}$ could be kept largely constant, even as circuit performance per unit area improved by S^3!
>
> However, as V_{DD} is lowered, the threshold voltage of transistors (V_{th}) must be lowered as well, and this leads to an exponential increase in leakage current ($I_{leakage}$) [BS00]. Although $I_{leakage}$ was rising exponentially, P_{static} accounted

for a very small fraction of total power until about 1999, but has been increasingly significant since then [TPB98].

This has meant that Dennard scaling effectively ended with the 90nm process technology in about 2004, because if V_{DD} was lowered further, P_{static} would be a significant and exponentially increasing fraction of total power dissipation [TPB98]. Consequently, with only transistor capacitance scaling, chip power dissipation would increase by S^2 each process generation if operated at full frequency.

This issue resulted in the Power Wall described in section 2.1. Switching to multicore and ending frequency scaling meant that power would now only scale with S. In addition, enhancements in fabrication technology such as FinFET/Tri-gate transistors and use of high-k dielectrics allowed designers to avoid this power wall at least temporarily [RM11, AAB+12, HLK+99].

Unfortunately, the end of Dennard scaling has another implication: for a fixed power budget, this means that with each process generation, only an ever decreasing fraction of on-chip resources may be active at any time, even if frequency scaling is ended. This problem is known as the *Utilization Wall*, and is exacerbated even further with the growing performance demands of increasingly portable yet functional devices like tablets, smartphones and smart-watches, that have evermore limited power budgets.

2.3.3 Implications of Dark Silicon

The issue of insufficient parallelism, together with the Utilization Wall means that despite ongoing exponential growth in transistors or cores on-chip with Moore's Law, a growing proportion of these resources must frequently remain un-utilized, or 'dark'. Firstly, performance scaling will primarily be limited due to poor sequential performance scaling in the consumer domain. Secondly, even for applications with abundant data or task level parallelism, such as in the server/datacenter or multimedia domains, the Utilization Wall limits the amount of parallelism that can be exploited in a given power budget.

A comprehensive analysis of these two factors by Esmaelzadeh et al. found that in 6 fabrication process generations, from 45nm to 8nm, while available on-chip resources grow by 32×, dark silicon will limit the ideal case performance scaling to only about 7.9× for highly parallel workloads, with a more realistic estimate being about 3.7×, or only 14% per year – well below the 52% we have been used to for most of the past three decades [EBSA+11]. This analysis assumed ideal per-benchmark multicore configurations from among those described by Hill and Marty [HM08], so actual performance scaling on a fixed, realistic architecture can be expected to be even lower.

Overcoming the effects of dark silicon would require addressing each of the constituent issues. Breakthroughs in auto-parallelisation or an industry-wide switch to novel programming models that can effectively expose fine-grained parallelism from sequential code would be required to effectively exploit available parallel resources. Overcoming the Utilization Wall, returning to Dennardian scaling, and re-enabling the full use of all on-chip resources would likely require a switch to a new post-CMOS fabrication technology that either avoids the leakage current issue, or is just inherently far more efficient overall [Tay12]. Barring such breakthroughs however, the best that architects can attempt is to *mitigate* the effects of each of these factors.

The Need to Accelerate Sequential Code

To accelerate sequential code, architects are currently developing heterogeneous multicore architectures, composed of different types of processor cores. One approach is to implement an *Asymmetric Multicore* processor, that combines a few complex out-of-order superscalar cores for accelerating sequential code, with many simpler in-order cores for running parallel code more efficiently [JSMP13]. Another is the *Single-ISA Heterogeneous Multicore*[4] approach, where cores of different performance and efficiency characteristics but implementing the same ISA, cooperate in the execution of a single thread – performance critical code can be run on the complex out-of-order processors, but during phases that do not require as much processing power (e.g. I/O intensive code), execution seamlessly switches over to the simpler core for efficiency [KFJ+03, KTR+04].

An example of such a design is ARM's big.LITTLE which is composed of two different processor types: large out-of-order superscalar Cortex-A15 cores, as well as a small and efficient Cortex-A7 cores [Gre11]. big.LITTLE can be operated either as an asymmetric multicore processor, with all cores active, or as a single-ISA heterogeneous multicore, where each Cortex-A15 core is paired with a Cortex-A7 in such a way that only one of them is active at a time, depending on the needs of the scheduled threads.

However, running sequential code faster on complex cores also inevitably means a decrease in energy efficiency, thus such architectures essentially trade-off between performance and energy by running non-critical regions of code at lower performance. Grochowski and Annavaram observed that after abstracting away implementation technology differences for Intel microprocessors, a linear increase in sequential performance leads to a power-law increase in power dissipation, given by the following equation [GA06]:

$$Pwr = Perf^{\alpha} \quad where \;\; 1.75 \leq \alpha \leq 2.25. \tag{2.3}$$

This puts architects between a rock and a hard place – without utilizing complex processors, performance scaling is limited by Amdahl's Law, and practical concerns, but with such processors, their high power dissipation means that the Utilization Wall limits speedup by limiting the number of active parallel resources at one time. Esmaelzadeh et al note that in order to truly mitigate the effects of dark silicon: "Clearly, architectures that move well past the Pareto-optimal frontier of energy/performance of today's designs will be necessary" [EBSA+11].

The Need for High Energy Efficiency

To mitigate the effects of the Utilization Wall, it is essential to make the most efficient use possible of the fraction of on-chip resources that can be activated at any given time. Recently, architects have been increasingly relying on custom hardware and/or reconfigurable architectures, incorporated as part of heterogeneous systems-on-chip, in order to achieve

[4]Although both involve combining fast cores with simple cores on the same multicore system-on-chip, a subtle distinction is made between *asymmetric* and *heterogeneous* multicores, primarily due to the different use cases these terms are associated with in the cited literature. The former expects sequential/low-parallelism fraction of an application to run on fewer large cores, with the scalable parallel code running on many smaller cores, as a direct response to Amdahl's Law, whereas the latter involves switching a single sequential thread from a small core to a large core in order to trade-off energy with sequential performance, as needed. Asymmetric multicores view all cores as available to a single parallel application, whereas the heterogeneous multicores approach typically makes different kinds of cores seamlessly available to a single sequential thread of execution.

high performance and high energy efficiency on computationally intensive operations such as video codecs or image processing. This trend has largely been driven by growing demand for highly portable (thus power limited) computing devices like smartphones and tablets, with strong multimedia capabilities. For such applications, custom hardware is able to provide as much as three orders of magnitude improvements in performance and energy efficiency [HQW+10] .

Custom and reconfigurable hardware are types of *spatial* computing architectures. The next section describes spatial computation, and considers how its advantages may be utilized to mitigate the effects of dark silicon. I also describe the current limitations of spatial computation that need to be overcome in order to be truly useful for addressing the dark silicon problem, and how several research projects have attempted to do so.

2.4 The *Spatial* Computation Model

Conventional processors rely on an imperative programming and execution model, where communication of intermediate operands between instructions occurs via a centralized memory abstraction, such as a register file or addressed memory location. For such processors, the *spatial* locality between dependent instructions – i.e. *where* they execute in hardware relative to each other – is largely irrelevant. Instead, what matters is their correct *temporal* sequencing – *when* an instruction executes such that correct state can be maintained in the shared memory abstraction.

Custom and reconfigurable hardware on the other hand utilize a more dataflow-oriented, *spatial* execution model. Dataflow graphs of applications are mapped onto a collection of processing resources laid out in space, with intermediate operands directly communicated between producers and consumers using point-to-point wires, instead of through a centralized memory abstraction. As a result, *where* in space an operation is *placed* is crucial for achieving high efficiency and performance – dependent instructions are frequently placed close to each other in hardware in order to minimize wiring lengths in the spatial circuit.

2.4.1 Advantages of Spatial Computation

Scalability

As the number of operations that can execute in parallel is increased, the complexity of memory elements such as register files in conventional architectures grows quadratically [ZK98, TA03]. Instead of such structures, spatial architectures (a) rely on programmer- or compiler-directed placement of operations, making use of abundant locality information from the input program description to minimize spatial distances between communicating operations, and then (b) implement communication of operands between producers and consumers through short, point-to-point, possibly programmable wires.

While broadcast structures like register-files and crossbars are capable of supporting non-local, random-access, any-to-any communication patterns, recent work by Greenfield and Moore indicates that maintaining this level of flexibility is unnecessary. By analysing the dynamic-data-dependence graphs of many benchmark applications, they observe that the communication patterns of many applications demonstrate *Rentian* scaling in both temporal and spatial communication between dependent operations [GM08a, GM08b].

By making use of short, point-to-point wiring for communication, spatial computation is able to take advantage of the high locality implied by Rent's rule: instead of having the worst-case, quadratic complexity growth of a multi-ported register-file, the communication complexity of spatial architectures would be governed by the complexity of the communication graph for the algorithm/program it is implementing.

The communication-centric nature of spatial architectures is also beneficial in addressing the general issue of poor wire scaling. Ron Ho et al. observed that: "increased delays for global communication will drive architectures towards modular designs with explicit global latency mechanisms" [HMMH01]. The highly modular nature of spatial architectures, together with their exposure of communication resources and their management to higher levels of abstraction (i.e. the programmer and/or compiler) means that they are inherently more scalable than traditional uniprocessor architectures.

Computational Density

In complex processor cores, only a small fraction of the die area is dedicated to execution resources that perform actual computation. Complex processors are designed to maximize the utilization of a small set of execution resources by overcoming false and control dependencies, and accelerating true dependence in an instruction stream. Consequently, the majority of core resources are utilized in structures for dynamically exposing concurrency from a sequential instruction stream: large instruction windows, register renaming logic, branch prediction, re-order buffers, multi-ported register files, etc.

Spatial architectures instead dedicate a much larger fraction of area to processing elements. Per unit area, this allows spatial architectures to achieve much higher computational densities than conventional superscalar processors [DeH96, DeH00]. Provided that applications can be mapped efficiently to spatial hardware such that the abundant computational elements can be effectively utilized, spatial architectures can achieve much greater performance per unit area. Given the fact that the proportion of usable on-chip resources is shrinking due to the Utilization Wall, the higher computational density offered by spatial architectures is an effective way of continuing to scale performance by making more efficient use of available transistors.

Energy Efficiency

Due to poor wire scaling, the energy cost of communication now far exceeds the energy cost of performing computation. Dally observes that transferring 32-bits of data across chip consumed the energy equivalent of 20 ALU operations in the 130nm CMOS process, which increased to about 57 ALU operations in the 45nm process, and is only expected to get worse [Dal02, MG08].

The communication-centric, modular, scalable, and decentralized nature of spatial architectures makes them well-suited to also addressing the energy efficiency challenges posed by poor wire scaling. Exploitation of spatial locality reduces the distances signals must travel, while reliance on decentralized point-to-point interconnect instead of multi-ported RAM and CAM (content-addressable memory) structures reduces the complexity of communication structures.

Programmable spatial architectures are able to reduce the energy cost of programmability in two more ways. First, instead of being broadcast from a central instruction store to execution units each cycle (such as the L1 instruction cache), 'instructions' are *config-*

ured locally near each processing element, thereby reducing the cost of instruction stream distribution. Secondly, spatial architectures are able to amortize the cost of instruction fetch by fixing the functionality of processing elements for long durations – when executing loops, instructions describing the loop datapath can be fetched and spatially configured once, then reused as many times as the loop iterates. DeHon demonstrates in [DeH13], that due to these factors, programmable spatial architectures exhibit an asymptotic energy advantage over conventional temporal architectures.

A further energy efficiency advantage can be realised by removing programmability from spatial computation structures altogether. Instead of utilizing fine-grained programmable hardware like FPGAs, computation can be implemented as fixed-function custom hardware, eliminating the area, energy, and performance overheads associated with bit-level logic and interconnect programmability. For certain applications, this approach has been shown to provide as much as three orders of magnitude improvements in energy efficiency over conventional processors [HQW+10], and almost 40× better efficiency than a conventional FPGA, with its very fine-grained, bit-level programmable architecture [KR06]. While this improved efficiency (and often performance) comes at the cost of flexibility, given the ever-diminishing cost per transistor thanks to Moore's Law, incorporating fixed-function custom hardware is an increasingly attractive option for addressing the utilization wall problem by actively making use of dark silicon through hardware specialization.

Of course, a middle-ground does exist between highly-programmable but relatively inefficient FPGAs and inflexible but efficient custom hardware: researchers are increasingly developing *coarse-grained* reconfigurable arrays (CGRAs), that are optimized for specific application domains by limiting the degree and granularity of programmability in their designs. This is done for instance by having n-bit ALUs and buses, instead of bitwise programmable LUTs and wires. Examples of such architectures are discussed in Section 2.4.3.

However, there remain several issues with spatial architectures that must be addressed before they can be more pervasively utilized to mitigate the effects of dark silicon, particulary for the general-purpose computing domain.

2.4.2 Issues with Spatial Computation

Despite its considerable advantages, spatial computation has not found ubiquitous utilization in mainstream architectures, due to a variety of issues. Many of these issues are often specific to particular types of spatial architecture, and could be addressed by switching to a different type. For instance, FPGAs provide a high degree of flexibility, but incur high device costs due to their specialized, niche-market nature, as well as exorbitant compilation times due to their fine-grained nature. This makes it difficult to incorporate them into existing software engineering practices that rely on rapid recompilation and testing. FPGAs also incur considerable cost, area, performance, and efficiency penalties over custom hardware, limiting the scope of their applicability to application areas where their advantages outweigh their drawbacks [Sti11].

Some of the efficiency, performance and cost issues can be mitigated by utilizing fixed-function custom hardware, or domain-specific CGRAs, as suggested by several academic research projects [KNM+08, MCC+06, VSG+10]. However, there are two fundamental issues for spatial computation that must be addressed before such architectures can be

easily and pervasively used.

Programmability

Implementing computation on spatial hardware is considerably more difficult than writing code in a high level language. This is a direct consequence of the spatial nature of computation. Whereas conventional processors employ microarchitecture-level *dynamic* placement/allocation of operations to execution units, and rely on broadcast structures for routing of operands, spatial architectures instead relegate the responsibility of operation placement and operand routing to the higher levels of abstraction. The programmer and/or the compiler (and in some cases even the system runtime) are now responsible for explicitly orchestrating placement, routing, and execution/communication scheduling of individual operations and operands. This is analogous to the way that the shift to multicore meant that the the programmer was responsible for exposing concurrency, except that spatial computation makes this far more complex, as much more low-level hardware details must now be managed explicitly.

A related issue is that of hardware *virtualization*: for conventional processors, programmers remain unaware of the resource constraints of the underlying execution environment – for a given ISA, the hardware may implement a simple processor with fewer execution units, or a complex processor with more. The programmer need not be aware of this difference when writing code. On the other hand, the spatial nature of computation also exposes the hardware *capacity* constraints to higher levels of abstraction. The programmer must in most cases ensure that this developed spatial description satisfies all cost, resource or circuit-size constraints.

Historically, programmers relied on low-level hardware description languages (HDLs) such as Verilog and VHDL to precisely specify the hardware for the functionality that they wished to implement. More recently, design portability and programmer productivity have been improved thanks to sophisticated high-level synthesis (HLS) tools, that allow the programmer to define hardware functionality using a familiar high-level language (HLL). Many such tools support a subset of existing HLLs like C, C++, or Java [CLN+11, CCA+11], while some augment these languages with specialized extensions to simplify concurrent hardware specification in a sequential language [Pan01, PT05]. At the same time, common coding constructs that enhance productivity, such as recursion, dynamic-memory allocation and (until recently) object-oriented code are usually not permitted.

Furthermore, the quality of output from such tools is highly sensitive to the coding style used [Sti11, SSV08], thus requiring familiarity with low-level digital logic and design optimization in order to optimize the hardware for best results. Recent HLS tools manage to provide much better support for high-level languages [CLN+11, CCA+11], but nevertheless, the spatial hardware programming task remains one of describing the hardware implementation in a higher level language, instead of merely coding the algorithm to be implemented.

Due to the difficulty and cost of effectively programming spatial architectures, their use has largely been relegated to numeric application domains where the abundant parallelism is easier to express, and the order-of-magnitude performance, density and efficiency advantages of spatial computation far outweigh the costs of their programming and implementation.

Amenability

Conventional processors dedicate considerable core resources to managing the instruction stream, and identifying and overcoming name and control dependences between individual instructions to increase parallelism. On the other hand, spatial architectures dedicate very few resources to such dynamic discovery of concurrency, opting instead for high computational density by dedicating far more area to execution units and interconnect. In order to make full use of the available execution resources, discovering and overcoming dependences then becomes the responsibility of higher levels of abstraction.

Overcoming many statically (compile-time) known name dependences becomes trivial through use of point-to-point communication of intermediate values, since there is no centralized register file with a finite number of registers that must be reused for multiple values. Also, unlike the total order on instructions imposed by a sequential instruction stream, spatial architectures directly implement the data-flow graph of an application, which only specifies a partial order on instructions considering only memory and true dependences.

However, overcoming dependences that require dynamic (runtime) information becomes more difficult, as the statically-defined structure of a spatial implementation cannot be easily modified at run-time. Code with complex, data-dependent branching requires aggressive control-flow speculation to expose more concurrency. Based on code profiling, the compiler may be made aware of branch *bias* – i.e. which side of a branch is more likely to execute in general – but it cannot easily exploit knowledge of a branch's dynamic behavior to perform effective branch *prediction*, which tends to be significantly more accurate [MTZ13]. Similarly, code with pointers and irregular memory accesses introduces further name dependences that cannot be comprehensively overcome with only compile-time information (e.g. through alias-analysis).

Due to these issues, spatial architectures have thus far largely been utilized for application domains that have regular, predictable control-flow, and abundant data or task-level parallelism that can be easily discovered at compile-time. Conversely, for general-purpose code that contains complex, data-dependent control-flow, spatial architectures consistently exhibit poor performance.

Implications

For these reasons, spatial architecture utilization is restricted largely to numeric application domains such as multimedia, signal-processing, cryptography, and high-performance computing, where there is an abundance of data-parallelism, and often regular, predictable control-flow and memory access patterns. Due to an increasing demand for highly portable yet functional, multimedia oriented devices, custom hardware components are commonly included in many smartphone and tablet SOCs, particularly to accelerate video, radio-modem, and image processing codecs. Such custom hardware presents an effective utilization of dark silicon, since these components are only activated when implementing very specific tasks, and remain dark for the remainder of the time.

While FPGAs are unsuitable for the portable computing domain due to their high area and relatively higher energy cost, they are increasingly being utilized in high performance computing systems, again to accelerate data intensive tasks such as signal-processing for oil and gas exploration, financial analytics, and scientific computing [Tec12].

However, with growing expectations of improved performance scaling with future tech-

nology generations, as well as critical energy-efficiency concerns due to the utilization wall, it is becoming increasingly important to broaden the applicability, scope and flexibility of spatial architectures so they may be utilized to address these issues. Section 2.4.3 highlights several recent research projects that attempt to address the programmability and/or amenability issues with spatial computation. This brief survey shows that while considerable success has been achieved in addressing the programmability issue, addressing amenability (by improving sequential code performance) has proven more difficult, especially without compromising on energy efficiency.

2.4.3 A Brief Survey of Spatial Architecture Research

While there are many examples of research projects developing spatial architectures targeted at numeric application domains like multimedia, signal processing etc. [HW97, PPM09, PNBK02], this brief survey focuses on selected projects that attempt to address at least one, if not both of the key limitations of spatial architectures, namely programmability and amenability.

RICA: The Reconfigurable Instruction Cell Array [KNM+08]

RICA is a coarse-grained reconfigurable architecture designed with the goal of achieving high energy efficiency and performance on digital signal processing applications. RICA is programmable using high-level languages like C, and executes such sequential code one basic-block at a time. To conserve energy, basic-blocks are 'depipelined', meaning that intermediate operands are only latched at basic-block boundaries, reducing the number of registers required in the design, but resulting in each basic block executing with a variable latency. To address this, execution of instructions in a block are scheduled statically, so that the total latency of each block is known at compile-time. This known clock latency is then used to *enable* the output latches from each basic block after the specified number of cycles.

RICA does not attempt to overcome control-flow dependences in any significant way. No support is provided for control-flow speculation, though the compiler does implement some optimizations such as loop unrolling and loop fusion that reduce some of the control-flow overheads. Though not mentioned, RICA might be able to implement some speculative execution through the use of compile-time techniques such as if-conversion [AKPW83] and hyperblock formation [MLC+92] to combine multiple basic blocks into larger blocks and expose more ILP across control-flow boundaries – such approaches are already used for generating statically scheduled VLIW code [ACPP05].

While RICA is able to address the issue of programmability to some degree, it still suffers from poor amenability, and as such is limited to accelerating DSP code with simple control-flow. RICA provides 3× higher throughput than a low power Texas Instruments TI C55x DSP processor, but with 2-6× lower power consumption. Compared to an 8-way VLIW processor (the TI 64X processor), RICA achieves similar performance on applications with simple control-flow, again with a 6× power advantage. However, for DSP applications with complex control-flow, RICA performs as much as 50% worse than the VLIW processor despite the numeric nature of the applications.

The MIT RAW Architecture [TLM+04]

The RAW architecture was developed to address the problem of developing high throughput architectures that are also highly scalable. Unlike RICA's Coarse-Grained Reconfigurable Array, RAW is classified as a massively parallel processor array (MPPA), since each of its processing elements is not simply an ALU, but a full single-issue, in-order MIPS core, with its own program counter. Each such core executes its own thread of code, and also has an associated programmable router. The ISA of the cores is extended with instructions for explicit communication with neighbouring cores.

The RAW architecture supports the compilation of general-purpose applications through the use of the RAWCC compiler, which is responsible for partitioning code and data across the cores, as well as statically orchestrating communication between cores. Much like a VLIW architecture, the responsibility for exposing and exploiting ILP rests with the compiler.

Compared to an equivalent Pentium III processor, a 16-tile RAW processor is able to provide as much as 6× performance speedups on RAWCC compiled numeric applications. Unfortunately, performance on non-numeric sequential applications is as much as 50% worse. While energy and power results are not provided, the 16-tile RAW architecture requires 3× the die area of the Pentium III processor at 180nm.

However, when utilizing a streaming programming model that enables the programmer to explicitly specify coarse-grained data-parallelism in numeric applications [GTK+02], RAW is able to provide as much as 10× speedups on streaming and data-parallel applications over the Pentium III.

DySER [GHN+12]

The DySER architecture is a spatial datapath integrated into a conventional processor pipeline. DySER aims to simultaneously address two different issues: *functionality specialization*, where a frequently executed region of sequential code is implemented as spatial hardware in order to mitigate the cost of instruction fetch and improve energy-efficiency, and *data-level parallelism*, where the spatial fabric is utilized to accelerate numeric code regions. Unlike many previous CGRAs, DySER relies on dynamically-scheduled, static-dataflow style execution of operations: instead of the execution schedule being determined at compile-time and encoded as a centralized finite-state machine, each processing element in the DySER fabric is able to execute as soon as its input operands are available.

DySER relies on a compiler to identify and extract frequently executed code regions and execute them on the spatial fabric. The DySER fabric does not support backwards (loop) branches or memory access operations: code regions selected for acceleration must therefore be partitioned into a computation subregion and a memory subregion, with the former mapped to the DySER fabric. All backwards branches and memory access operations are executed in parallel on the main processor pipeline. From the perspective of the processor, a configured DySER fabric essentially looks like a long-latency, pipelined execution-unit.

As the DySER fabric need not worry about managing control-flow or memory accesses, this approach greatly simplifies its design. However, being tightly coupled with a conventional processor considerably limits the advantages of spatial computation for the system as a whole. Without support for backwards branching, DySER is limited to accelerating code from the inner-most loops of applications. Furthermore, the efficiency

and performance advantages of the spatial fabric can be overshadowed by the energy cost of the conventional processor that cannot be deactivated while the fabric is active.

A 2-way out-of-order superscalar processor extended with a DySER fabric is able to achieve a speedup of 39% when accelerating sequential code, over the same processor without DySER. However, only a 9% energy efficiency improvement is observed. On data-parallel applications, DySER achieves a 3.2× speedup, with a more respectable 60% energy saving over a conventional CPU.

Wavescalar [SSM+07]

The Wavescalar architecture was designed with two objectives in mind: (1) develop a highly scalable, decentralized processor architecture, (2) that matches or exceeds the performance of existing superscalar processors. The compiler for Wavescalar compiles HLL code into a dataflow intermediate representation: the Wavescalar ISA. Unlike conventional processor ISAs, the Wavescalar ISA does not have the notion of a program counter or a *flow* of control. Instead, execution of each operation is dynamically scheduled in dataflow order. Abandoning the notion of a flow of control between blocks of instructions allows Wavescalar to concurrently execute instructions from multiple control-independent regions of code, effectively executing along *multiple-flows of control*, as described by Lam [LW92] (this is discussed in greater detail in Chapter 3).

Wavescalar employs the dynamic dataflow execution model [AN90], meaning that multiple copies of each instruction in the Wavescalar ISA may be active at any time, and may even execute out of order, depending on the availability of each copy's input operands. This is similar to an out-of-order superscalar processor, which implements a restricted version of the dynamic-dataflow execution model for instructions in its issue window [PHS85].

The original implementations of the dataflow execution model in the 1970's and 1980's did not support the notion of mutable state, and hence were unable to support compilation of imperative code to dataflow architectures, instead relying on functional and dataflow programming languages [WP94, Tra86]. To support mutable state, Wavescalar introduces a method of handling memory-ordering called *wave-ordered memory*, that associates sequencing information with each memory instruction in the Wavescalar ISA. Wave-ordered memory, enables out-of-order issue of memory requests, but allows the memory system to correctly sequence the out-of-order requests and execute them in program order. Thus a total load-store sequencing of side-effects in program-order can be imposed.

Wavescalar currently does not support control speculation, or dynamic memory disambiguation. Nevertheless, thanks to its ability to execute along multiple flows of control, it performs comparably to an out-of-order Alpha EV7 processor on average – outperforming the latter on scientific SpecFP benchmarks, while performing 10-30% worse than the Alpha on the more sequential, control-flow intensive SpecINT benchmarks. While not implemented, Swanson et al. estimate that with perfect control-flow prediction and memory disambiguation, average performance could be improved by 170% in the limit.

As energy-efficiency was not a stated goal for the Wavescalar architecture, no energy results are presented. However, efficiency improvements from the Wavescalar architecture can be expected to be limited, since unlike statically scheduled architectures (PPA [PPM09], RICA [KNM+08], C-Cores [VSG+10]) or static-dataflow architectures (DySER [GHS11], Phoenix/CASH [BVCG04]), Wavescalar incurs considerable area and energy overheads in the form of tag-matching hardware to implement the dynamic-

dataflow execution model. Only a small fraction of each Wavescalar processing element (PE) is dedicated to the execution unit, reducing its computational density advantage – the majority of PE resources are used to hold up to 64 in-flight instructions, along with tag-matching, instruction wake-up and dispatch logic.

TRIPS [SNL+04] and TFlex [KSG+07]

The TRIPS architecture had similar design goals to the Wavescalar project: develop a scalable spatial architecture that can overcome poor wire-scaling, while continuing to improve performance for multiple application types, including sequential code. TRIPS aimed to be *polymorphic*, i.e. capable of effectively accelerating applications that exhibited instruction, data, as well as thread-level parallelism.

Unlike the purely dataflow nature of the Wavescalar ISA, TRIPS retained the notion of a flow-of-control, utilizing a hybrid approach that attempted to combine the familiarity of the Von-Neumann computation model with the spatial, fine-grained concurrency of the static dataflow execution model. A program for TRIPS was compiled to a Control-Data Flow Graph (CDFG) representation. Unlike the pure dataflow Wavescalar ISA, TRIPS retained the notion of a sequential order between basic-blocks in a CFG, but like RICA, only a dataflow order is expressed within each such acyclic block.

As it is important to overcome control-flow dependences in order to achieve high performance on sequential code, TRIPS utilizes a combination of static and dynamic techniques to perform control speculation. The TRIPS compiler relies on aggressive loop-unrolling and *flattening* (i.e. transforming nested loops into a single-level loop) to reduce backwards branches, and then reduces forward branches by combining multiple basic-blocks from acyclic regions of code into hyperblocks [MLC+92, SGM+06].

The TRIPS architecture incorporates a 4×4 spatial grid of ALUs, onto which the acyclic dataflow graph of each hyperblock is statically placed and routed by the compiler. To accelerate control-flow between hyperblocks, TRIPS utilizes aggressive branch prediction hardware to fetch and execute upto 8 hyperblocks speculatively, by mapping each into its own separate context on the 4x4 grid. This approach has its own trade-offs: correct prediction of hyperblocks can potentially enable much greater performance, but incorrect prediction incurs a considerable energy and performance cost, as the misspeculated hyperblock and all its predicted successors must be squashed and a new set of blocks fetched.

TRIPS incurs considerable hardware overheads due to managing hyperblocks in separate contexts, dynamically issuing instructions from across multiple hyperblocks, and effectively *stitching* hyperblocks together to communicate operands between hyperblock contexts [Gov10]. As a result of this complexity, despite the advantages of largely decentralized spatial execution, TRIPS exhibits only a 9% improvement in energy efficiency compared to an IBM Power4 superscalar processor, at roughly the same performance, when executing sequential code.

More recent work shows that TRIPS performance on sequential SpecINT benchmarks is 57% worse on average cycle-counts than an Intel Core2 Duo processor (with only one core utilized) [KSG+07]. Kim et al. attribute this poor performance to the limitations of their academic research compiler, as code must be carefully tuned to produce high performance on TRIPS. A newer version of TRIPS, called TFlex, manages to improve average sequential performance by 42% over TRIPS, while dissipating the same power. This is achieved primarily by allowing the compiler to adapt the hardware to the requirements

of the application more accurately – TFlex does not constrain execution to a fixed 4×4 grid of PEs, instead allowing the compiler to select the appropriate grid configuration for each application.

The CMU Phoenix/CASH [BVCG04]

The CMU Phoenix project developed the *Compiler for Application Specific Hardware* (CASH), the goals of which were to maximize the programmability, scalability, performance and energy efficiency potential of custom-computing by compiling unrestricted high-level language code to asynchronous static-dataflow custom hardware. Instead of targeting a specific spatial architecture like TRIPS, Wavescalar or DySER, the CASH compiler transformed largely unrestricted[5] ANSI C code into a custom hardware description in Verilog.

The CASH compiler converted C code into the 'Pegasus' IR – a CDFG-derived dataflow graph which could be directly implemented in hardware. Like the TRIPS compiler, Pegasus relied upon aggressive loop unrolling and hyperblock formation to mitigate control-flow dependences and expose ILP. However, unlike TRIPS, the generated application-specific hardware (ASH) did not incorporate any branch prediction to overcome inter-block control-flow dependences at run-time. As a result, while instructions from across multiple hyperblocks (or even multiple instances of the same hyperblock) could execute in parallel, they would only do so when the control-flow predicate for each hyperblock had been computed.

Being full-custom hardware, ASH avoided the overheads of dynamic instruction-stream management and dataflow stitching incurred by TRIPS. By employing static-dataflow instead of dynamic-dataflow, ASH avoided the costs of dynamic tag-matching and instruction triggering logic incurred by Wavescalar. A further efficiency improvement results from the fact that applications are implemented as fully *asynchronous* hardware, thus the dynamic power overhead of driving a clock-tree, which forms a significant proportion of dynamic power dissipation in synchronous circuits, is avoided. In combination, these factors allow the generated hardware to achieve three orders of magnitude greater energy efficiency than a conventional processor. One drawback of the CASH approach is that the size of an ASH circuit generated is proportional to the number of instructions in the code being implemented.

In order to overcome this issue, as well as target a more flexible, reconfigurable hardware platform, Mishra et al. developed Tartan, a hybrid spatial architecture that loosely coupled a conventional processor with an asynchronous CGRA. The conventional processor implements all the code that could not be compiled for the CGRA, in particular handling library and system calls. Unlike DySER, the CGRA is largely independent of the conventional processor and able to access the memory hierarchy independently. Thus while the CGRA is active, the coprocessor may be deactivated to conserve power.

The Tartan CGRA provided a 100× energy-efficiency advantage over the same code implemented on a simulated out-of-order processor [MCC+06]. Tartan also alleviated the circuit size issue by supporting hardware virtualization in the CGRA [MG07]: during program execution, the ASH implementations of required program functions would be fetched and configured into the CGRA, and evicted once more space was needed for

[5] Even recursive functions are supported. Only missing was support for library and system calls, run-time exception handling (using *signal()* and *wait()*, *setjmp()* and *longjmp()* functions). Code containing these types of operations is offloaded to a loosely-coupled conventional processor.

newer functions. This allowed a fixed-size CGRA to implement an arbitrary sized ASH circuit, so long as the size of an individual function did not exceed the capacity of the CGRA.

The performance characteristics of ASH follows a similar pattern observed in earlier projects: for embedded, numeric applications with simple control-flow, ASH achieves speedups ranging from 1.5-12×, but for sequential code, ASH can be as much as 30% slower than a simulated 4-way out-of-order superscalar processor [BAG05]. Budiu et al. identify two key reasons for this limitation: the lack of control-flow speculation between hyperblocks, and the fact that the dynamic-dataflow model approximated by conventional out-of-order processors is fundamentally more powerful than the static-dataflow model.

Conservation Cores [VSG+10]

The GreenDroid project developed the Conservation Cores approach, designed specifically to mitigate the effects of the utilization wall. Their objective was to identify frequently executing regions of code in a system, and implement these as *patchable*[6] custom hardware cores. Unlike conventional systems-on-chip that incorporate custom hardware for accelerating numeric applications, conservation-cores focus on energy-efficiency instead of performance.

Conservation cores is an attempt to make effective use of dark silicon by specializing available hardware resources, and utilizing a fraction of them as needed. Energy-intensive code regions are selected for implementation as conservation-cores. These are compiled to a CDFG style intermediate representation, and then implemented as synchronous, statically-scheduled custom hardware (with some modifications to incorporate patching support). A number of conservation cores are loosely-coupled (like Tartan/ASH, and unlike DySER) with a simple in-order MIPS 24K processor core that executes the remaining code. Patchable conservation cores are able to provide 10× higher energy efficiency than the same functionality implemented on the in-order MIPS core, but with a 5-10% performance loss compared to the in-order processor.

More recent revisions of this work apply ‘selective depipelining’ to reduce the number of datapath registers, and incorporate ‘cachelets’ to accelerate memory access to the cores [SVGH+11]. This has improved efficiency over conservation-cores by 2× on average, for a total efficiency advantage over the MIPS core of 20×, and improved performance to about 1.2× the in-order MIPS baseline. Nevertheless, with statically scheduled execution and limited control-flow speculation, sequential performance of conservation-cores is far below that of an out-of-order superscalar processor.

2.5 Summary

Computer Architects face several critical challenges to the continued scaling of performance with Moore's Law:

1. Poor wire scaling, together with growing design cost and complexity issues are driving the development of highly modular, scalable, *communication-centric* archi-

[6]Conservation cores are *slightly* programmable, in that some of their behavior can be modified at runtime. This puts them conceptually between conventional fixed-function custom hardware, and highly programmable reconfigurable architectures.

	RICA [KNM+08]	Cons-Cores [VSG+10]	CASH [BVCG04]	TRIPS [Gov10]	DySER [GHS11]	Wavescalar [SSM+07]	Superscalar
Execution Scheduling	static	static	SDF	SDF	SDF	DDF	DDF
Control-flow Acceleration	static	static	static	BP	BP	MFC	BP
Sequential Performance	Low	Low	Medium	Medium	High	High	High
Numeric Performance	High	High	High	High	High	High	Medium
Energy Efficiency	High	High	High	Medium	Low	Low	Low

Table 2.1: A summary of features and comparison of energy/performance characteristics of spatial architectures surveyed in Section 2.4.3. SDF = Static Dataflow; DDF = Dynamic Dataflow; BP = Branch Prediction; MFC = Multiple-flows of Control.

tectures, that require explicit management of communication at higher levels of abstraction (e.g. by the programmer or compiler).

2. The end of Dennard scaling, and the resultant Utilization Wall, necessitate the development of highly energy efficient architectures with high computational density, to provide continued performance scaling with Moore's Law.

3. The continuing importance of achieving high performance on sequential code, necessitated by Amdahl's Law for improving performance on applications with insufficient explicit parallelism [EBSA+11]. Even for domains with abundant thread-level parallelism, per-thread sequential performance remains important [H10].

Spatial Architectures are well suited to addressing two of these three issues. They are modular and decentralized, mitigating the effects of poor wire-scaling, while also providing high application specific adaptivity and computational density, leading to orders-of-magnitude improvements in energy efficiency that help mitigate the effects of the utilization wall [BVCG04, HQW+10]. In addition, they have also demonstrated high potential for accelerating applications with abundant data-level parallelism.

However, the crucial problem of *amenability* still needs to be properly addressed: while embedded applications with simple, regular control-flow and abundant data-parallelism can be conveniently compiled to spatial architectures and provide high performance and efficiency [GHS11, Bud03, KNM+08, PPM09], sequential code with limited data parallelism, complex control-flow, and often irregular memory access patterns runs poorly when compiled to spatial architectures. Either performance [VSG+10, BVCG04], or energy-efficiency [SNL+04, GHS11, SSM+07] must be compromised when compiling such code to spatial architectures. Typically, researchers suggest utilizing complex and energy-inefficient out-of-order superscalar processors to accelerate such code [BAG05, H10].

Table 2.1 highlights the relative energy/performance trade-offs of the various architectures surveyed in Section 2.4.3. Several spatial architectures manage to approach or exceed the high sequential performance of superscalar processors when running sequential code, but lose much of their efficiency advantage due to significant microarchitectural overheads incurred by dynamic management of instructions, data-forwarding, or management of speculative execution contexts [SSM+07, Gov10, GHS11]. High energy-efficiency is achieved instead by architectures that generate highly application-specific hardware,

targeting fixed-function hardware or CGRAs with minimal dynamic instruction or data-stream management overheads [KNM+08, BVCG04, VSG+10], but these architectures exhibit poor sequential performance in comparison to a superscalar processor.

It is also important to note that architectures that implement dataflow-style dynamic execution scheduling of instructions, as well as some mechanism for overcoming control-flow dependences (either via branch prediction or exploiting multiple-flows of control) [BVCG04, MCC+06, SNL+04, SSM+07, GHS11] achieve higher sequential performance than architectures that rely on static execution scheduling and/or have limited control-flow speculation support [VSG+10, TLM+04].

In order to improve sequential performance in spatial architectures it is therefore necessary to expose ILP by overcoming control-dependences in sequential code, as well as to exploit this ILP by utilizing dataflow style dynamic execution scheduling of instructions. However, in order to do so without compromising the energy efficiency advantages of spatial hardware, I argue that much of this work ought to be done at the compiler level instead of the hardware microarchitecture level, in order to minimize runtime energy overheads. To this end, this dissertation develops a new dataflow compiler IR, designed to statically expose ILP from control-flow intensive code, and suitable for implementation as application specific static-dataflow hardware [BVCG04].

Chapter 3, describes the key limitations to achieving high sequential performance in application specific hardware, and then discusses how this new IR is able to expose ILP from control-flow intensive code.

CHAPTER 3

Statically Exposing ILP from Sequential Code

In Chapter 2, we observed the need to achieve very high energy efficiency, as well as high sequential performance, in order to mitigate the effects of dark silicon. Different spatial computation approaches provide *either* efficiency [KNM+08, BVCG04, VSG+10], *or* sequential performance [SSM+07, Gov10, GHS11], but not both. This chapter starts by discussing the nature of imperative, sequential code, and how concurrency may be exposed and exploited from it. We then discuss how spatial architectures (custom/reconfigurable hardware in particular) achieve high energy efficiency by relying primarily on static exposition of ILP by the compiler or programmer, at the cost of sequential performance [CCA+11, BVCG04, MCC+06, VSG+10]. This is contrasted with the primarily dynamic exposition of ILP in out-of-order superscalar processors, incurring a high energy cost for improving sequential performance.

Finally, we discuss how the performance limitations of spatial architectures may be overcome, with the help of the Value State Dependence Graph [Law07], without incurring the high energy cost of dynamic ILP techniques.

3.1 The Nature of Imperative Code

A simple grammar to represent an assembly-like imperative language is presented in Figure 3.1. Each operation 'o' updates the computational *state* by reading values from its input locations, performing the appropriate operation, and assigning a value to its output location(s). As with modern processor assembly-languages, I assume *strict* execution semantics for each operation, i.e. all inputs must be available before an operation can execute. Storage locations serve as a means of communicating values between producer and consumer operations. The set of all program locations L constitutes the state of the computation, and may be considered composed of two sets: $L = Reg \cup Mem$, with $r \in Reg$, where Reg is the set of temporary registers, and $m \in Mem$, where Mem is the set of memory locations[1].

[1]For simplicity, only a few addressing modes are shown, such as register ($binary(r_{src1}, r_{src2})$), immediate ($binaryI(r_{src1}, imm \in \mathbb{Z})$), direct ($loadDirect(m_{src})$), and register-indirect ($load(r_{addr})$, or $store(r_{addr}, r_{data})$). I also assume that I/O operations are memory-mapped, and volatility issues will be taken into account by the programmer/compiler that produces such code. Finally, I assume that no

Given $r \in Reg, m \in Mem$, and $L = Reg \cup Mem$, where L is the set of 'locations' constituting program (and system) state:

$$
\begin{array}{lrl}
\text{Operations} & o ::= & r_{dest} := binary(r_{src1}, r_{src2}) \quad | \\
 & & r_{dest} := binaryI(r_{src1}, imm \in \mathbb{Z}) \quad | \\
 & & r_{dest} := unary(r_{src}) \quad | \\
 & & r_{dest} := select(r_{pred}, r_{src1}, r_{src2}) \quad | \\
 & & r_{dest} := loadDirect(m_{src}) \quad | \\
 & & r_{dest} := load(r_{addr}) \quad | \\
 & & m_{dest} := store(r_{addr}, r_{data}) \\
 & & \\
\text{Control-Flow} & c ::= & skip \mid o_1; c_N \mid \textbf{if } r_P \textbf{ then } c_T \textbf{ else } c_F \mid \textbf{while } r_P \textbf{ do } c_L \mid \\
 & & call(c_{function}) \mid return
\end{array}
$$

Figure 3.1: BNF Grammar for low-level imperative code. I assume that I/O is memory-mapped and handled through the load/store instructions, and that volatility aspects are taken into account. Also note that for the *store* operation, the contents of r_{addr} specify the memory location m_{dest}.

Multiple operations can be composed into an application control-flow graph (CFG), by making use of the control-flow commands 'c'. Sequential composition is used to construct basic blocks ($o_1; c_N$), while the '*if*' and '*while*' statements implement control-flow between basic-blocks.

This typical representation of imperative code makes the data dependencies between operations implicit – i.e. values are passed between producer and consumer operations indirectly, through storage locations – whereas the control-flow ordering between operations is made explicit. It is expected that the order of execution of operations and control-flow commands (and hence the order of updates to the set of all locations L) will occur strictly in the control-flow order specified by the program (i.e. *sequentially*) in order to ensure correctness of program execution and predictability of state updates throughout execution. In conventional processors, this is implemented by utilizing a *program-counter*, that specifies the unique next operation that will execute in control-flow order. However, this strictly sequentializing nature of control-flow can be relaxed in various ways in order improve performance by allowing some concurrent updates to state, without affecting the correctness of program execution.

Observable vs Trace Semantics: For instance a distinction can be made between the *observable* semantics of a program and its *trace* semantics. As defined by Alan Lawrence [Law07]: "the observable semantics of a program describe the actions of the program in terms of a function from its inputs to its outputs (including side-effects[2])".

operation has any side-effects other than updates to the set of register and memory locations L.

[2]A function is said to have a side-effect if, in addition to returning a value, it also has an observable interaction with its calling function, the remainder of the program, or the outside world through modi-

Unlike observable semantics, the trace semantics of a program instead deal with the precise sequence of steps with which a program will compute its results.

In imperative languages like C, a program or function may receive explicit arguments as well as the current computational state (analogous to the set of locations L from Figure 3.1), as input. The function may produce an explicit output value, and/or modify state before returning: between its invocation and return, the function may allocate memory for, and update local data-structures and/or temporary variables on the call-stack, as well as access and update global heap memory. In addition the function may also execute system calls to perform I/O, dynamically allocate or free heap memory, raise exceptions, or request other system services.

Any operations in the function that modify state are considered side-effects if these modifications or their effects remain *live* – i.e. can affect the observable behavior of the system – after the function exits. In addition to the function's output return value, these side-effects are also part of the observable semantics. For instance, I/O operations or any changes to heap-allocated memory would be considered side-effects, as such changes often are visible after the function returns. Conversely, temporary variables allocated on the call-stack are not typically preserved on function exit, and thus updates to these would not be considered side-effects.

In order for correctness of the program representation and implementation, only the observable semantics need to be preserved exactly by any compiler transformation. Thus, so long as the function outputs, including all of its side-effects, remain indistinguishable from a strictly sequential, control-flow ordered execution, the execution order for individual operations may be relaxed. For non side-effecting operations, the execution order of operations need only (and must necessarily) be constrained by the *data*-flow dependencies between operations, as opposed to their control-flow ordering. This allows for the possibility of accelerating computation by concurrently executing operations that are data-independent of each other.

With reference to a function '$c_{function}$' written in our imperative language from Figure 3.1, the observable semantics are concerned with the correct values assigned (and order of updates) to a subset S of state L, i.e. $S \subseteq (Reg \cup Mem)$, that comprises the output of $c_{function}$, as well as its side-effects. The set S would typically exclude most temporary registers as well as memory locations in the program call-stack, save for those used to pass input and output arguments. It would of course include all locations in heap memory that are updated and affect observable program behavior (i.e. remain live) after $c_{function}$ returns, as well as memory-mapped locations utilized to implement I/O operations or system-calls.

In order to accelerate the execution of sequential imperative code, the goal is therefore to maximize concurrent execution of operations (i.e. exploit *instruction-level parallelism*). To ensure correctness, only the true data dependencies need to be preserved for non side-effecting operations, while both data dependencies and control-flow ordering must be preserved for operations with side-effects[3]. However, several issues arise when attempting to

---

fications to the computational state, given by the set of locations L in Figure 3.1. L includes not only programmer visible or programmer managed locations, but also system locations such as those affected by I/O operations, system calls, and exceptions.

[3] In actuality, only the *appearance* of correct control-flow ordering need be preserved for side-effecting operations – the execution of such operations may occur out of control-flow order, so long as their effects become visible to the system in the correct order. This type of decoupling of operation execution from state-update is extensively utilized in aggressive superscalar processors that often execute memory

relax the control-flow ordering, and exploit data dependencies between non side-effecting operations from an imperative program representation like the one from Figure 3.1:

1. Name dependencies (known at compile-time),
2. Overcoming control-flow between basic-blocks, and
3. Memory disambiguation (dependencies only resolved at runtime).

The manner in which different architectures address each of these issues when implementing (C-like) imperative languages affects how much ILP can be exploited, hence performance improved, for a given section of imperative code.

3.2 Exposing ILP from Imperative Code

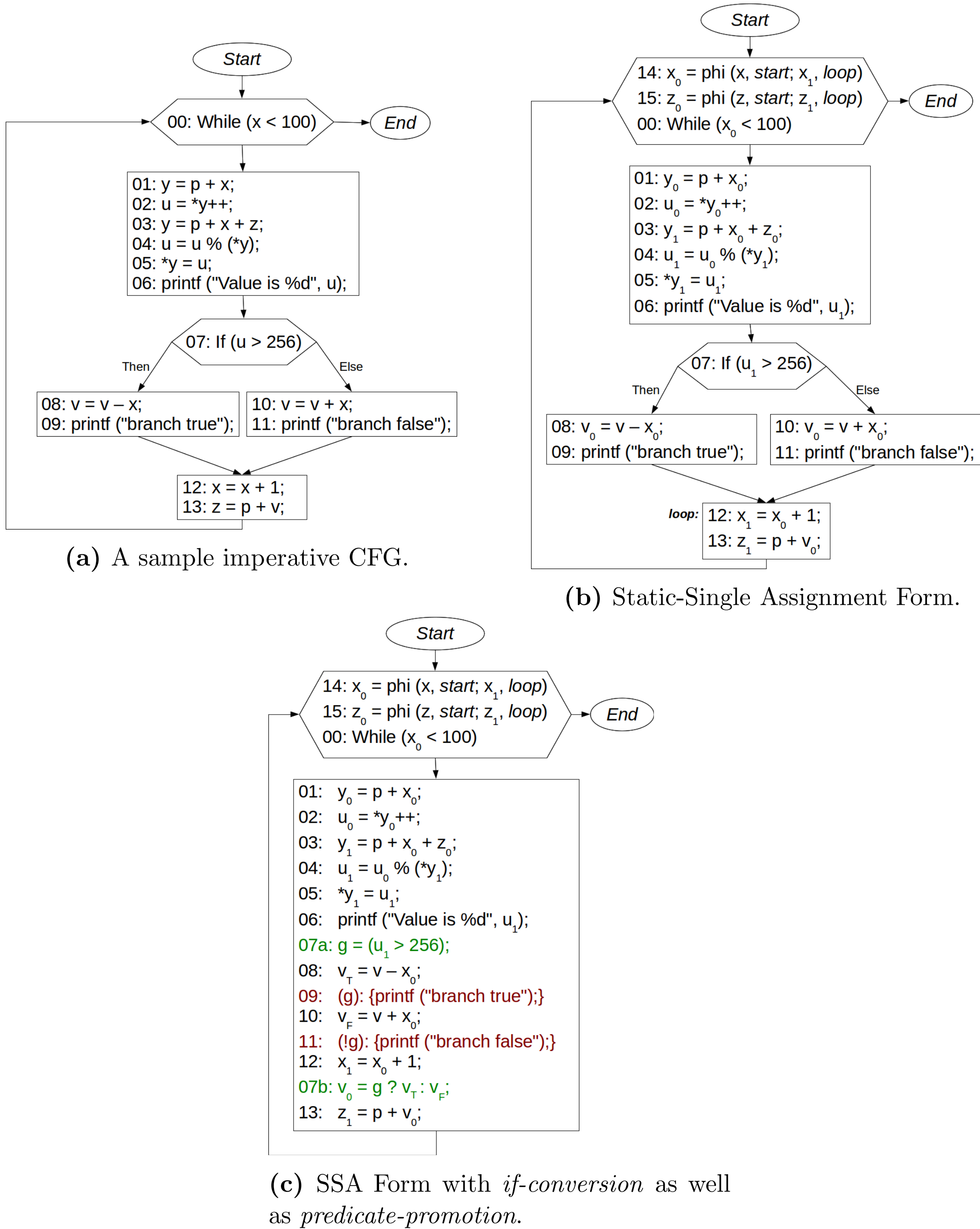
This section discusses how both modern out-of-order superscalar processors, as well as custom hardware (and by extension, energy-efficient spatial architectures) attempt to overcome these issues, and demonstrates the reasons for the performance advantage of the former on irregular, sequential, control-intensive code. The sample control-flow code shown in Figure 3.2a is used to illustrate each of the three issues mentioned above.

3.2.1 False or Name dependencies

Name dependencies, such as write-after-write (WAW) and write-after-read (WAR), occur due to the need to *reuse* a finite set of locations during program implementation for storing the results from an arbitrarily larger (potentially infinite) number of dynamically executed operations at run-time, and unnecessarily constrain operation execution ordering. In Figure 3.2a, the pointer variable y is written to by operations 01 and 03, while it is read by operations 02, 04 and 05. Although there is no dataflow between operations 01 and 03, there is a write-after-write (WAW) hazard. Similarly, there is both a write-after-read (WAR) and a WAW hazard between operations 02 and 03, but again no dataflow. The only true data dependencies through the pointer variable y in this basic block are: $01 \rightarrow 02$, $03 \rightarrow 04$, and $03 \rightarrow 05$. A similar set of name dependencies exist for the variable u. Overcoming name dependencies often involves increasing the number of available temporary locations.

For instance, a compiler-based approach to reducing name-dependencies in imperative code is to convert it to static-single-assignment (SSA) representation, where each variable is written to only once in the code [CFR+91]. As shown in Figure 3.2b, SSA form makes the dataflow between operations more explicit in the code by avoiding reuse of the same variable for different purposes. Note however that there still exist name dependencies between multiple iterations of the same loop – each of the variables $(u_i, v_i, x_i, y_i, z_i)$ is reused each time the loop body is re-entered. To reduce run-time name dependencies due to loops, one solution is to unroll each loop a number of times, introducing new SSA variable names for each unrolled iteration [Fin10, CM08]. Loop unrolling can help reduce the effect of loop-carried name dependencies, but often at the cost of increased code-size.

operations speculatively and out-of-order, yet only *commit* them to the program state in-order. Work done by Lokhmotov et al on 'Sieve C++' also allows the programmer to expose more parallelism by manually introducing this decoupling in code by *delaying* side-effects [LMR07].



(a) A sample imperative CFG.

(b) Static-Single Assignment Form.

(c) SSA Form with *if-conversion* as well as *predicate-promotion*.

Figure 3.2: A sample imperative code CFG (with various optimized versions) to illustrate the issues that must be overcome in order to expose true data dependencies and exploit concurrency between operations.

High-level synthesis tools rely extensively on such static approaches to resolve name dependencies. However, increased code-size due to loop unrolling often translates into increased area and resource utilization in the generated custom hardware. Custom hardware also usually employs static execution scheduling, where the execution schedule for operations is determined at compile-time, and implemented at run-time by a centralized finite-state machine [CCA+11, VSG+10]. In the presence of complex control-flow, unrolling loops when implementing statically-scheduled hardware also affects hardware complexity and performance, as discussed in Section 3.2.2.

Instead of relying on the compiler, out-of-order superscalar processors employ 'register-renaming' logic to overcome name dependencies dynamically. The small set of ISA registers used in the input binary are dynamically mapped to a much larger set of renaming registers. On advantage of this approach is that processors can unroll loops *dynamically* instead of at compile-time – with register renaming, multiple instances of the same instruction, say from a tightly wound loop, may be issued and executed independently, without causing any WAW or WAR hazards, and without the increase in code size that is caused by static unrolling. Register-renaming, combined with dynamic, out-of-order execution scheduling, allows superscalar processors to approximate the dynamic-dataflow model of execution [PHS85].

3.2.2 Overcoming Control Flow

In Figure 3.2b, operation 13 is data-dependent upon operations 08 or 10, which only exhibit a loop-carried data dependency on themselves (and each other). Operations 09 and 11 are side-effects, but also have no data dependencies on preceding instructions within the while-loop. However, the execution of operations 13, 08 and 10 is constrained not only by their respective dataflow dependencies, but by their control-flow dependence on operation 07. The evaluation of 07 must be preceded by several long latency operations: modulus, and two memory loads (operations 04 and 02).

If we constrain all operations to execute in control-flow order, despite having very simple data-dependencies, 08, 10, and 13 cannot execute until the control-dependency from 07 is resolved. A similar situation arises when executing operations from across multiple iterations of the while loop (e.g. through loop unrolling): the control-flow condition for the next iteration of the while loop (operation 00) is data dependent on operation 12 only, which again only has a loop carried dependence on itself. However, since 12 is again control-dependent on 07, the start of the next iteration cannot be determined until all the long-latency operations that 07 requires are completed.

As discussed earlier, in addition to the true data dependencies, correct program execution requires that control-dependencies must also be respected, but only for side-effects (and operations with name data dependencies through memory). Several compile-time and runtime techniques are available for overcoming control-flow dependencies for non-side-effecting operations.

Modern superscalar processors utilize aggressive branch prediction at runtime to relax the constraints imposed by control flow on ILP: branch predictors with very high accuracy ($\geq 95\%$) enable effective speculation across multiple branches, providing a much larger region of code from which multiple independent instructions may be discovered and executed out of order. To handle cases of branch mis-speculation, processors make use of an in-order commit buffer (or re-order buffer) to selectively commit executed instruc-

tions in program order. When a misprediction is detected, executed instructions from the mispredicted paths can simply be discarded from this buffer, preserving correct program state. This decoupling of *execution* of operations from the *commitment* of their effects to program state allows superscalar processors to improve ILP by speculatively executing both side-effecting and non-side effecting operations.

Static approaches to overcoming control-flow rely on enabling speculative and *predicated* execution from forward branches through if-conversion and/or hyperblock formation [MLC+92]. These types of approaches are typically utilized by simpler, energy efficient processors such as those from ARM, or VLIW architectures. Compilers for such architectures often apply *if-conversion* together with *predicate-promotion* in order to convert forward (if-else) control-flow into dataflow. If-conversion involves replacing forward branches with a computed boolean guard value, that is then used to *predicate* the execution of control-dependent operations. Then, control-dependent operations that do not cause side-effects may be predicate-promoted, i.e. allowed to execute speculatively, without becoming data dependent on the guard condition.

Figure 3.2c shows the CFG from Figure 3.2b after if-conversion and predicate-promotion. The four basic blocks forming the body of the loop have now been combined into a single large block. The *if* control-flow command in 07 has been replaced by the computation of a guard g (07a). The control-dependent side-effect operations (09 and 11) have their execution predicated by this guard condition such that both may execute, but only one will have its effects committed to program state, depending on the value of g. While operations 08, 10 and 12 were also control-dependent on 07, as they do not have side effects (nor any name dependencies), their execution does not need to be predicated by g, i.e. they have been predicate-promoted, and thus may execute *speculatively*. Finally, a *select* operation is needed to choose the correct new value of v_0 (07b), essentially converting control-flow in this instance to dataflow.

Predicate promotion of 08, 10, and 12 allows their execution in dataflow order without having to wait for g. As a result the next loop condition (00) can also be computed much earlier. As if-conversion is only applicable to acyclic code, conventional (or VLIW) processors that support predicated instructions often augment if-conversion with either static loop unrolling or dynamic branch prediction to speculatively execute across loop iterations as well.

Unfortunately, standard approaches to generating custom hardware are unable to take full advantage of if-conversion – the static execution scheduling typical for most conventional custom hardware implies that such hardware can only be conservatively scheduled for the multiple possible control-flow paths through the code, leaving it unable to adapt to runtime variability that may occur due to data-dependent control-flow, predicated side-effects and memory operations, variable-latency operations, or unpredictable events such as cache misses.

Furthermore, while speculative execution of non-side-effecting operations is possible through if-conversion, predication of side-effects is indistinguishable in a static schedule from execution in control-flow order – in both cases, operations must wait for the computation of the guard/control-flow condition. Also, while if-conversion allows speculative execution of some forward branches (without side-effects) in custom hardware, currently no mechanisms exist for safely speculating on backwards branches (loops).

Superscalar processors are able to both speculate across backwards branches, as well as speculatively execute side-effecting operations. This is due to the fact that such processors

decouple the execution of an operation from any updates it can make to the program state L. This is done through the use of some form of 'commit-buffer', where the results of all operations that have been issued and executed are stored temporarily, without becoming part of the program state. Superscalar processors only *commit* the results of operations from this buffer to program state in the correct control-flow order. This decoupling also provides a convenient mis-speculation roll-back and recovery mechanism – operations, side-effecting or otherwise, that have been mis-speculated can simply have their results discarded from the commit buffer.

Custom hardware, with its specialized and minimalist nature, does not have any such decoupling of operation execution from state update via a commit-buffer. Speculation via if-conversion on forward branches means that the results of speculatively executed operations will be discarded at some later synchronization step, such as the *select* operation 07b from Figure 3.2c. However, without a commit-buffer-like mis-speculation roll-back and recovery mechanism, speculation is not permitted for side-effecting operations, nor across backwards branches.

High-level synthesis tools employ static unrolling, flattening and pipelining of loops in order to decrease the number of backwards branches that would be dynamically executed (i.e. by converting them into forward branches) [CM08]. But this can significantly increase the complexity of the control and multiplexing logic that implements the static schedule for the hardware datapath, resulting in very long combinational paths that can overwhelm any gains in IPC [GDGN04, KSP07]. As observed by Gupta et al in [GDGN04]: "Although loop unrolling and pipelining have been proposed previously for high-level synthesis, we found that when the resource utilization is already high – because of either high instruction level parallelism in the design or as a result of loop unrolling – the control and interconnect (multiplexing) costs of further loop unrolling or loop pipelining outweigh the gains achieved in performance. This is because frequently the longest combinational path in the circuit (the critical path length) increases so much that the input to output circuit delay becomes worse".

Overcoming the performance limitations due to explicit control-flow is the key issue that needs to be addressed for custom hardware to become performance-competitive with conventional processors on sequential code [BAG05].

3.2.3 Pointer Arithmetic and Memory Disambiguation

Pointer arithmetic is an established feature of many imperative languages (especially C and C++). Together with the use of data-dependent control-flow, the implication of such pointer arithmetic is that the precise sequence of locations that a pointer points-to at runtime cannot be determined at compile-time, except in the most trivial cases (i.e. static points-to analysis is *undecidable* [Lan92]). Pointer arithmetic can therefore often obfuscate the data dependence between producers and consumers, leaving the potential for a true or name dependence that can only be resolved at runtime, when the values of pointer variables are known. This is the 'unknown address problem' [PMHS85].

This effect puts an additional constraint on the ordering of memory operations, even those accessing locations that are not a part of the set S. All accesses to memory locations that (a) have their 'address-taken', or are accessed after pointer arithmetic, and (b) whose points-to set membership cannot be determined with sufficient precision, must also be constrained by the program control-flow in a similar manner to side-effecting operations,

until the precise address locations pointed-to can be discovered at runtime. This constraint applies even for temporary storage locations on the call-stack that are not typically part of the set S, and is necessary to preserve data dependencies through such locations.

In Figure 3.2b, while SSA form helps distinguish between pointer locations y_0 and y_1, it may not be possible through static analysis to determine whether $*y_0$ and $*y_1$ refer to the same value. Thus even though there may not be a data dependence through the location pointed to by y_0, as long as independence cannot be determined with certainty, operation 02 must precede 03 ($02 \rightarrow 03$). Similarly, if it cannot be determined that y_i from successive iterations of the loop will always point to distinct, independent locations, all accesses through pointer y_i across iterations must occur in the control-flow order specified by the CFG. Thus, although the set of locations pointed-to by y_i may be temporary stack variables and thus not a part of S, any operations that access memory through y_i must also be treated like side-effects (i.e. constrained by control-flow as well as dataflow).

Superscalar processors rely on out-of-order execution and dynamic memory disambiguation [SLH90] logic to expose and exploit concurrency between memory operations. Memory operations are issued to a Load/Store Queue (LSQ) within the processor, where the processor can dynamically determine execution order *after* the target addresses for each such operation are known: execute operations in-order of there is a name dependency between them, or allow out-of-order execution otherwise.

High-level Synthesis tools for generating custom hardware instead typically rely on static alias-analysis to disambiguate memory addresses at compile-time and allow concurrent execution of memory access operations. The inability to take advantage of precise run-time information limits the degree of concurrency that can be exposed and exploited compared to the dynamic disambiguation mechanisms in superscalar processors. To address this, work by Budiu improved upon existing high-level synthesis tools by incorporating a LSQ into their generated custom dataflow hardware, enabling limited dynamic memory disambiguation and out-of-order memory access execution in addition to the static memory disambiguation already implemented by their compiler [Bud03].

Their work however demonstrated that despite incorporating support for dynamic memory disambiguation, control-flow (Section 3.2.2) and loop-carried name dependencies (Section 3.2.1) remain the primary issues hindering the performance of spatial hardware on irregular sequential code [BAG05]. Given the existing work on incorporating dynamic memory disambiguation into spatial hardware, this thesis leaves the study of efficient dynamic memory disambiguation for future work, and instead focuses on overcoming control-flow and loop-carried name dependences in spatial hardware.

3.3 The Superscalar Performance Advantage

When it comes to control-intensive code, custom hardware is often limited by its total reliance on programmer and compiler effort for exposing and exploiting concurrency:

- Static execution scheduling reduces the control-flow and dataflow acceleration achievable via if-conversion on forward branches.

- Custom hardware does not support dynamic branch prediction, and must instead rely on loop unrolling to expose ILP from loops.

- Custom hardware must often rely on static alias-analysis for memory disambiguation.

Conversely, modern out-of-order superscalar processors rely extensively on dynamic effort in order to resolve name dependencies, overcome control dependencies, and disambiguate memory addresses:

- Branch prediction, together with misprediction recovery through the in-order commit buffer (re-order buffer) enables aggressive control-flow speculation across both forwards and backward branches.

- Dynamic, out-of-order execution scheduling, combined with register-renaming, allows superscalar processors to approximate the dynamic-dataflow model of execution [PHS85].

- Memory operations can frequently be executed concurrently (from within a finite instruction window), thanks to dynamic memory disambiguation.

Dynamic execution scheduling helps in dealing with unpredictable behavior at runtime such as variable-latency operations, cache-misses, or branch mispredicts. Instructions are allowed to execute as soon as their input operands, as well as the appropriate execution resources, become available. For instance, in the event of a cache miss, only instructions that are dependent on the stalled instruction would be delayed, while independent instructions can continue to execute.

The cost of all of this dynamic effort in silicon is incurred in both area and power dissipation. Figure 3.3 shows the power dissipation figures for a simulated Intel Nehalem Core i7 processor core running various integer benchmarks. As can be seen, a large proportion of power is dissipated in the register-renaming logic (OOO Logic), and dynamic memory disambiguation at the LSQ (OOO Mem). Next, instruction fetch and instruction caches (IF and Icache) dissipate the most power, followed by the operand forwarding and bypass network (Bypass) that is used to accelerate true dependencies by forwarding results between pipeline stages. In fact, only a small fraction of the total power is dissipated by the integer execution units (Exec Int): about 0.3W, which forms only 3-8% of the total power dissipated per benchmark.

As mentioned in Section 3.2.3, this thesis will not focus on addressing the memory disambiguation problem here, as there already exists abundant literature on (a) improving static pointer analysis and auto-parallelization, (b) or agumenting static analysis with profiling information [TWFO09], as well as (c) incorporating dynamic disambiguation logic into spatial hardware [SSM+07, Bud03]. Instead, I focus on the remaining two advantages of out-of-order superscalar processors identified above: support for aggressive control-flow speculation, as well as the ability to dynamically overcome loop-carried name dependences with register-renaming (thereby approximating restricted dynamic dataflow execution). The objective is to find ways of matching these advantages in spatial hardware, without incurring the high energy costs associated with the dynamic logic utilized by conventional processors.

The following case study illustrates the performance advantage of superscalar processors due to these two features on a region of code the performs poorly when implemented in conventional, statically scheduled custom hardware.

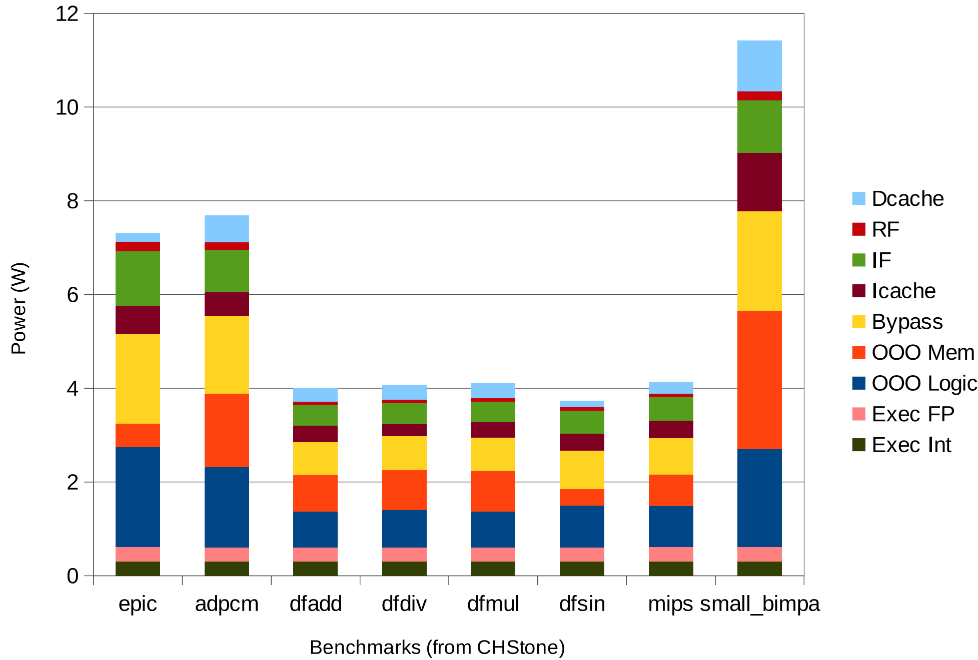


Figure 3.3: Power dissipation breakdown for a single core of the Intel Nehalem Core i7 architecture. Generated using Sniper interval simulator and the McPAT power estimation tool (The following settings were used for McPAT: $V_{DD} = 1.2V$; DVFS enabled; 45nm process).

3.3.1 Case Study 1: Outer-loop Pipelining

Consider for instance, the *internal_int_transpose* function from the *epic* Mediabench benchmark given in Figure 3.4. Budiu et al identified this as a region of code that performs poorly when implemented as custom hardware [BAG05].

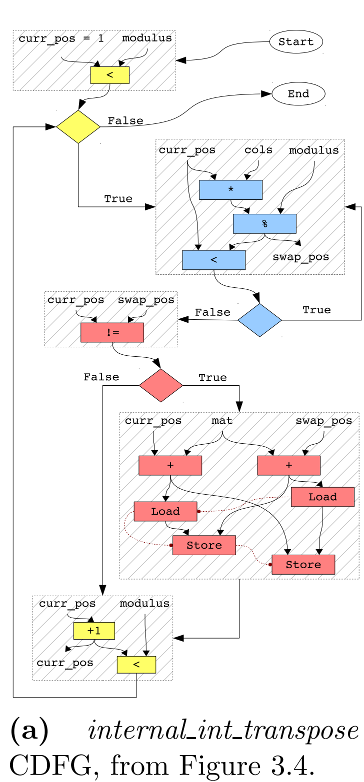
At runtime, the inner do-while loop rarely executes more than once and never more than twice each time, while the outer-loop executes for a large number of iterations. The branch prediction logic in conventional processors adapts to this execution pattern and is effectively able to execute multiple copies of the outer loop, i.e. performing *outer-loop pipelining*, thereby effectively hiding much of the latency of the % and * operations in the inner loop. Conventional high-level synthesis tools would implement the control-data flow graph (CDFG) of this code, shown in Figure 3.5a, as statically-scheduled custom hardware. Without branch prediction, each basic block (grey boxes) will be executed one at a time, in sequence. The exit predicates for the active block must be computed before control can *flow* to the next block.

One may attempt to alleviate this strict control-flow ordering somewhat by statically unrolling the loops. Unrolling the innermost loop will not yield significant benefit as it rarely executes more than once. Unrolling the outer-loop would replicate the blocks that comprise it, including the inner-loop and the if-block. Figure 3.5b shows the CDFG from Figure 3.5a unrolled twice. As can be seen, the control-flow within the outer-loop body is replicated as-is, and must still be accelerated either dynamically through branch prediction, or statically through if-conversion and/or hyperblock formation. Due to presence of

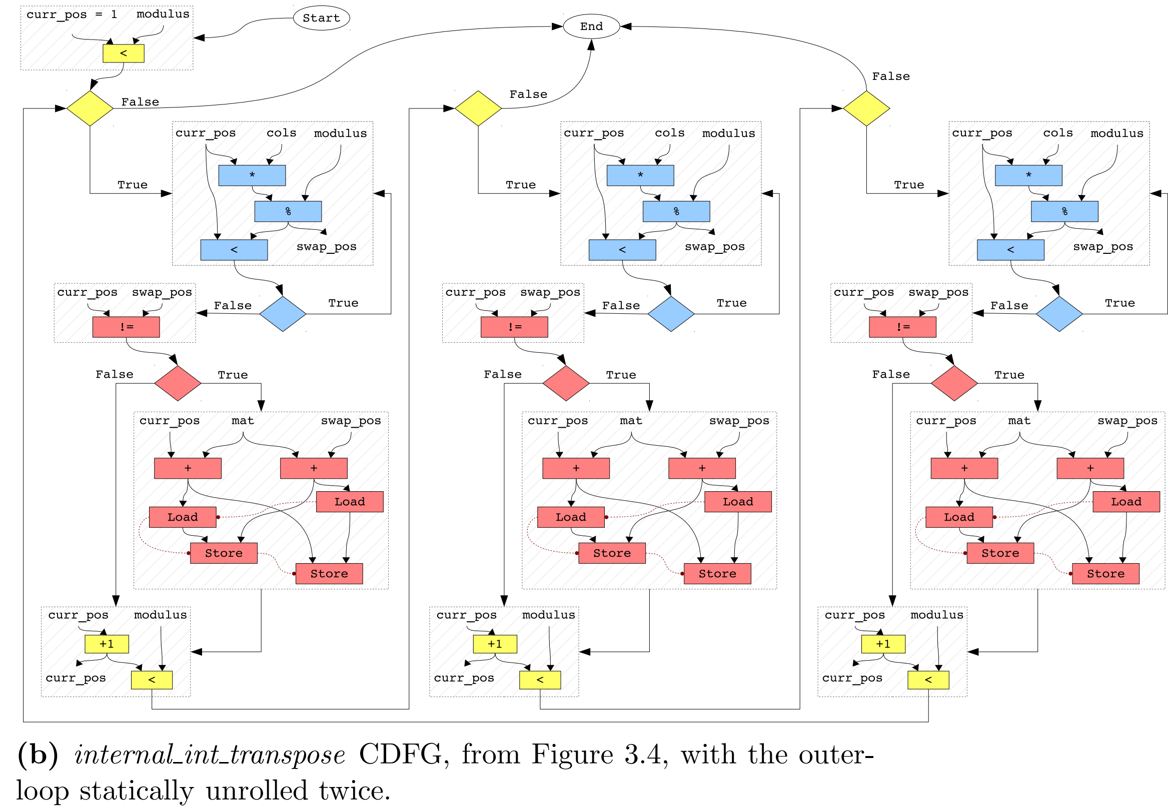
```
/*=============================================                    1
In-place (integer) matrix tranpose algorithm.                      2
Handles non-square matrices, too!                                  3
=============================================*/                    4
void internal_int_transpose(int* mat, int rows, int cols, int modulus  5
    ){
  int swap_pos, curr_pos, swap_val;                                6
                                                                   7
  for (curr_pos=1; curr_pos<modulus; curr_pos++) {                 8
    swap_pos = curr_pos;                                           9
    do {                                                           10
      swap_pos = (swap_pos * cols) % modulus;                      11
    }                                                              12
    while (swap_pos < curr_pos);                                   13
                                                                   14
    if (curr_pos != swap_pos) {                                    15
      swap_val      =   mat[swap_pos];                             16
      mat[swap_pos] =   mat[curr_pos];                             17
      mat[curr_pos] =   swap_val;                                  18
    }                                                              19
  }                                                                20
}                                                                  21
```

Figure 3.4: The *'internal_int_transpose'* function from the *'epic'* Mediabench benchmark

the inner-loop cycle, the latter solution is not possible. Due to the lack of mis-speculation recovery mechanisms for backwards branches and memory operations, neither the inner-loops nor the memory operations in the if-block can be executed speculatively. Thus, unrolling the outer-loop provides no benefit in this case, since the predicate computations of each basic block in the new sequence would still be on the critical path.



(a) *internal_int_transpose* CDFG, from Figure 3.4.



(b) *internal_int_transpose* CDFG, from Figure 3.4, with the outer-loop statically unrolled twice.

Figure 3.5: The CDFGs for *'internal_int_transpose'*, from Figure 3.4. The blue operations belong to the inner do-while loop, the red operations belong to the subsequent if-block, while the yellow operations are the outer-loop iterator increment and condition-checking operations.

```
for (i = 0; i < 100; i++)          1
  if (A[i] > 0) foo();             2
bar();                             3
```

Figure 3.6: Example C Code for illustration purposes

3.4 Limitations of Superscalar Performance

Despite these key advantages of out-of-order superscalar processors, there exists a fundamental limit to how much ILP can be dynamically extracted from a sequential instruction stream by such a processor. A limit study by Wall [Wal91] considered a hypothetical 64-issue out-of-order superscalar processor with a 2000 instruction issue window, and found that even assuming perfect memory disambiguation and register renaming, IPC was limited to between 4-8 for non-numeric sequential code, due to imperfect control-flow prediction.

A subsequent study by Lam and Wilson confirmed control-flow as the primary cause of the ILP Wall limiting sequential code performance [LW92]. They also noted that uniprocessor performance is limited not just due to imperfect branch prediction, but also because uniprocessors are inherently limited by having to maintain a single *program counter*, effectively exploiting ILP from only a *single flow of control*. By enabling the identification of multiple independent regions of code through control dependence analysis, and then allowing such regions to execute independently, (i.e. exploiting *multiple flows of control*), they observed that ILP could again be increased by as much as an order of magnitude. This was corroborated by a more recent limit study by Mak and Mycroft [MM09].

3.4.1 Case Study 2: Multiple Flows of Control

Consider the sample code in Figure 3.6. A conventional processor would be capable of dynamically unrolling the *for* loop and executing instructions from multiple iterations of the loop. Aggressive branch prediction will also accelerate the transfer of control from the loop into the *foo()* function, whenever the *if* condition is predicted-taken. Once control-flow switches to *foo()*, the processor will issue instructions from within *foo()* until control returns back to the context of the loop. Assuming *foo()* contains non-trivial code, the processor will not be able to simultaneously issue independent instructions from subsequent iterations of the *for*-loop, as it is constrained to executing from a single flow-of-control (i.e. maintaining a single program-counter). Similarly, even though the contents of the *for*-loop and the *bar()* function are control-independent, instructions from *bar()* can only be issued once control exits the loop.

Lam and Wilson note that as long as the data-dependencies and side-effect ordering between different calls to *foo()* are respected, instructions from multiple copies of *foo()* could be issued in parallel. Similarly, instructions from *bar()* could also be issued concurrently with the execution of instructions from multiple copies of *foo()* and the dynamically unrolled loop iterations.

Exploiting multiple-flows of control by extending a conventional out-of-order processor would be impractical, as a conventional processor cannot keep track of multiple-flows of control and their interdependencies. In order to automatically exploit multiple flows of control while maintaining a single program-counter, an impractically large instruction

issue window would be needed that could contain all instructions from multiple copies of *foo()*. Furthermore, perfect branch and jump prediction (as well as possibly if-conversion and inlining of *foo()*) would be necessary to decide precisely how much dynamic unrolling of the *for*-loop is needed, so as to determine when to start fetching instructions from *bar()*. Without perfect prediction, any misprediction would lead to all non-committed operations being discarded from the commit-buffer, unless some form of control-dependence analysis can be incorporated in the processor to determine the control-independent instructions that need not be discarded [LW92].

Various approaches have been explored to overcome the ILP Wall by exploiting multiple-flows of control. These can be classified according to the level of abstraction they focus on:

- **Programming Model:** Conventional multicore systems rely on programmer specified concurrency – often a shared-memory threaded parallel programming model is utilized by the programmer to explicitly partition code into multiple threads (hence multiple flows of control). However, as noted in Section 2.2, due to the difficulties with this programming model, performance gains on non-numeric client-side applications remain low [BDMF10].

- **Compiler Level:** Auto-parallelization tools are primarily focused on extracting data-level parallelism. However, more recent work has studied parallelizing more irregular code [CJH+12], achieving speedups of 2.25× on a 6-core Intel Core i7-980X processor. The Wavescalar project is another example where the compiler IR exposes dataflow parallelism from multiple-flows of control by abandoning the notion of a program counter (section 2.4.3 provides more details).

- **Language/system Runtime:** Considerable research has been undertaken into the area of Thread-Level Speculation (TLS), or Speculative Multithreading (SpMT), where sequential code is speculatively partitioned into concurrent threads by the system runtime [YRHBL13]. Recently, such systems have also demonstrated modest speedups on regular sequential applications with easily discovered parallelism.

- **Architecture Level:** The Multiscalar architecture was proposed to expose and exploit multiple-flows of control from a sequential instruction stream by partitioning the stream into *tasks*, then executing each task on a distinct processing-element. Architectural features were utilized to manage data and control-dependencies between tasks executing on different PEs. Multiscalar exhibited modest speedups over a conventional architecture on sequential code [SBV95].

These approaches have largely focused on improving performance, and have not considered energy-efficiency as an objective, or even a constraint.

3.5 Improving Sequential Performance for Spatial Hardware

Various research projects have attempted to overcome these sequential performance limitations by devising spatial architectures that perform aggressive control flow speculation,

dynamic execution scheduling, or both. TRIPS employs both dynamic and static techniques to expose ILP from control-flow intensive code [SNL+04], while DySER offloads all control-flow speculation duties to its host processor [GHS11]. Both projects utilize the static-dataflow model for dynamic execution scheduling. On the other hand, Wavescalar discards the notion of the program-counter, exploits multiple flows of control, and utilizes the dynamic-dataflow model for execution scheduling. Unfortunately, all three approaches also significantly compromise the orders-of-magnitude energy-efficiency advantage the custom/reconfigurable spatial hardware can provide.

In order to overcome the sequential performance limitations of conventional custom hardware, without compromising the energy efficiency potential of spatial harwdare, I propose two key changes during high-level synthesis:

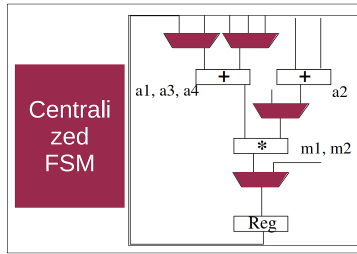
- Instead of using a statically scheduled instruction execution model for custom hardware, the dynamically scheduled static-dataflow execution model should be used, similar to the Phoenix/CASH approach mentioned in Section 2.4.3, and described in [BVCG04, Bud03]. This approach enables dynamic scheduling without compromising energy efficiency in spatial hardware [MCC+06].
- A new compiler IR is needed to replace the CDFG-based IRs that are traditionally used for hardware synthesis. This new IR should be based on the Value State Dependence Graph (VSDG) [Law07, JM03], as it has no explicit notion of a sequential flow-of-control between basic-blocks. Instead the VSDG only represents the necessary value and state dependencies in the program. Section 3.6 defines the VSDG, and its advantages.

3.5.1 Why the *Static* Dataflow Execution Model?

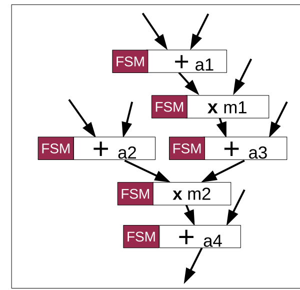
Unlike conventional statically-scheduled custom hardware, the static-dataflow execution model for spatial computation proposed by Budiu et al [BVCG04] enables the execution of each individual operation to be scheduled dynamically, based on the status of its input and output edges. This allows the spatial hardware to be more tolerant of variable or unpredictable latencies at runtime.

Static-dataflow provides an additional advantage over statically-scheduled spatial hardware: since each operation is triggered based only on the state of its input and output edges, the control-path in hardware can be fully decentralised – there is no need for a centralized finite-state machine (FSM) to implement a compiler-defined schedule (Figure 3.7a). Instead, each processing element has its own simple, local FSM (Figure 3.7b). Due to this decentralization, control and multiplexing costs do not scale as the size of the machine being implemented grows, thus improving hardware scalability [Bud03]. This becomes particularly useful when applying loop optimizations like unrolling, flattening and software pipelining, as centralized control can often increase the longest combinational path in the circuit and limit performance gains [GDGN04, KSP07].

Unlike the dynamic dataflow model of the Wavescalar architecture (also approximated by out-of-order processors), each edge in a static-dataflow graph may hold only one value at any time, thereby avoiding the need for complex operand-tag-matching, instruction wake-up, and issue logic [SSM+07]. Nevertheless, the dynamic dataflow model is fundamentally more powerful than static dataflow, thanks to the former's ability to eliminate false dependencies in tightly wound loops by supporting multiple values per dataflow



(a) Statically-scheduled custom hardware with the schedule implemented by a centralized FSM.



(b) Dataflow spatial hardware supporting dynamic scheduling. Each hardware operation has its own FSM.

Figure 3.7: Static execution scheduling vs dynamic execution scheduling in hardware.

edge [BAG05]. Ideally, to match the performance of conventional superscalar processors, custom-hardware would need to be able to approximate the dynamic-dataflow model, but must do so without overly compromising its energy-efficiency.

Switching to static-dataflow execution for custom hardware alone goes some way towards improving sequential code performance beyond statically-scheduled hardware, as demonstrated by the Phoenix/CASH work [BVCG04]. However, performance on sequential code is still poor compared to an out-of-order processor, primarily due to the control-flow limitations imposed by their CDFG-based compiler IR [BAG05].

3.5.2 Why a VSDG-based compiler IR?

In order to overcome the limitations of sequential control-flow on ILP in spatial architectures, I propose the utilization of a new compiler IR that is based on the Value State Dependence Graph, instead of the traditionally used Control Flow Graph. The main reason for emphasizing a change at the compiler IR level instead of relying on dynamic, microarchitectural control-flow acceleration like TRIPS, DySER and Multiscalar is to maximize energy-efficiency – as much of the effort in exposing ILP as possible ought to be done at compile time, in order to minimize the effort and energy expended at runtime.

The Value State Dependence Graph is a compiler IR, originally introduced by Johnson and Mycroft [JM03, Joh04], and then updated by Lawrence [Law07]. Unlike the Control-Flow Graph, which represents flow of control explicitly and dataflow implicitly, the VSDG represents the data and side-effect ordering dependencies between operations explicitly, without any explicit representation of a flow of control. Section 3.6 describes the VSDG in more detail, and discusses how its structure can expose more ILP from control-flow intensive code when implemented as static-dataflow spatial hardware.

3.6 Overcoming Control-flow with the VSDG

3.6.1 Defining the VSDG

A detailed definition of the VSDG has been developed by Johnson [Joh04], and revised by Lawrence [Law07]. This section briefly summarizes the definition developed by Lawrence, albeit slightly modified.

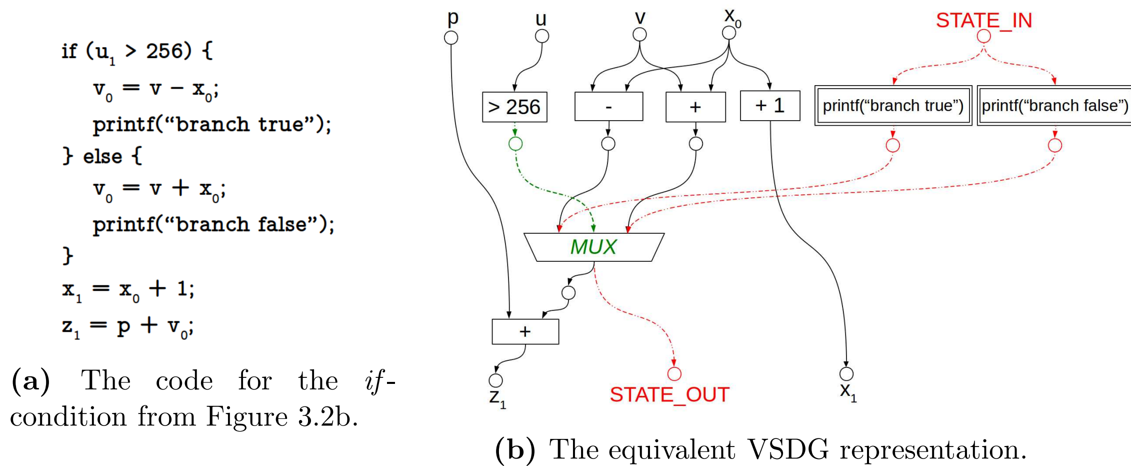
Definition 3.1. Value State Dependence Graph (VSDG): A VSDG is a labelled, directed, acyclic, hierarchical Petri-net[4] $G = (P, T, E)$, where:

- **Transitions** T represent operations. Operations may be of three types:
  - **Value Transitions** represent the basic, non side-effecting unary and binary arithmetic and logic operations.
  - ***MUX* Transitions** are used to implement speculative execution by applying if-conversion – a MUX transition will select and output one of two input values based on a third, boolean predicate input.
  - **State Transitions** represent those operations that either access or modify system state and thus must be constrained to execute in control-flow order to preserve the observable semantics of the program. State transitions are subdivided into a further two types: **Memory-access transitions** (i.e. load and store), and **Compound transitions**, representing nested VSDG subgraphs.
- **Places** P represent the results of operations. Each place may be of either *Value* type, or *State* type. Value places hold data results produced by a Value operation, while state places hold tokens produced by State operations representing an access to, or update of system state. Aside from the places representing intermediate values within the graph, each VSDG would have a set of input argument places, and a set of output result places:
  - **Arguments** $P_{in} \subset P$ are places holding values (or state) being passed into a VSDG.
  - **Results** $P_{out} \subset P$ are the places holding the output values (or state) generated by a VSDG.
- **Edges** $E \subseteq P \times T \cup T \times P$ represent dependencies on, and production of results by operations.

Unlike the explicitly sequential control-flow ordering specified by imperative languages (and their Control Flow Graphs), the VSDG represents only the true data-dependencies between operations/transitions, as well as the memory-access and side-effect ordering dependencies that must be imposed to match the observable semantics of the code being represented. There is no explicit *flow* of control from one operation to the next, or between basic-blocks. Instead of organizing code into basic blocks, forward branches are represented using MUX-nodes, that implement a selection between two sets of incoming *value* and/or *state* edges based on a third *predicate* value.

The VSDG is a hierarchical, acyclic data-dependence graph: it represents each function call (and loop) in the program as a compound operation containing a nested subgraph. Figure 3.8b shows the *if* condition from the CFG in Figure 3.2b, expressed as a VSDG. All of the operations in the VSDG retain their necessary dataflow or *value* edges from the CDFG (solid black arrows). State operations such as loads, stores, function call and loops subgraphs that may access or modify state upon execution, have an additional type of

[4] For now, the use of a Petri-net is strictly a matter of representation, and does not imply Petri-net style execution semantics. Lawrence develops graph-reduction based lazy execution semantics for this representation in [Law07], while this dissertation develops eager, static-dataflow execution semantics for a modified variant of the VSDG in Chapter 4.



(a) The code for the *if*-condition from Figure 3.2b.

(b) The equivalent VSDG representation.

Figure 3.8: An example VSDG representation of a forward branch. Solid black lines indicate data-flow dependencies, red dotted lines indicate state/side-effect ordering dependencies, and the green dotted lines indicate predicate/guard computation.

edge: an incoming and outgoing *state* edge (red dashed arrows) that pass a token between state-sensitive operations in order to enforce sequentialization of side effects in program order. As can be seen, the explicit data (and state ordering) dependencies represented by the VSDG allow for an implicit if-conversion of forward branches (described previously in Figure 3.2c).

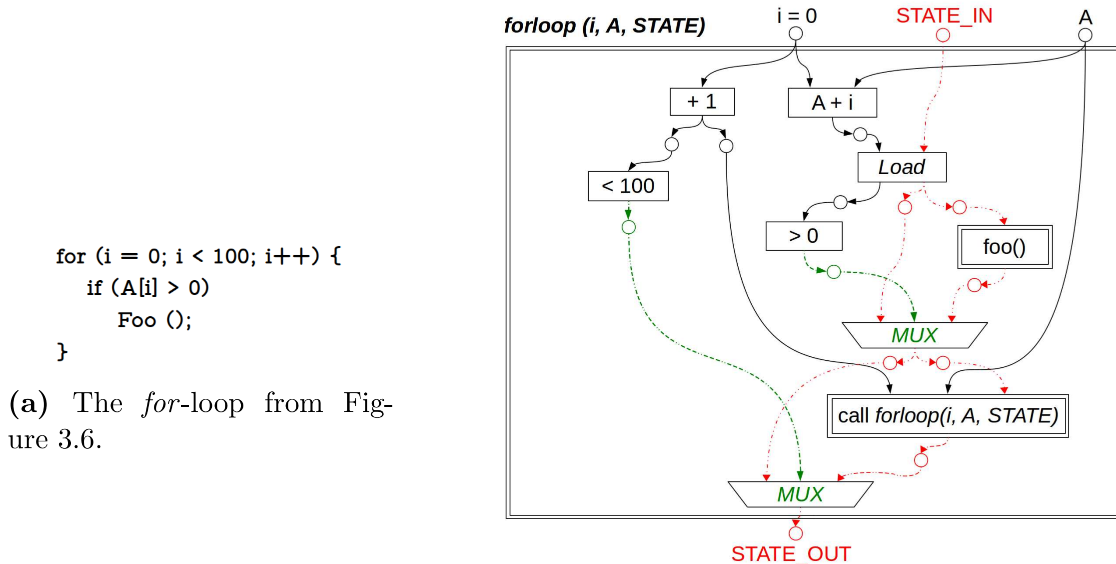
Double-borders are used to represent compound operations – transitions containing their own nested VSDG subgraphs. From the perspective of its parent graph, each compound operation appears like any of the other Petri-net transitions: it consumes tokens from its input places (P_{in}), and produces output tokens (P_{out}) *atomically* – i.e. each compound transition appears to its parent graph to occur instantaneously[5]. The *isolation* implied by atomicity is guaranteed by the fact that all side-effects and state accesses are explicitly ordered by the state-edge throughout all levels of the VSDG graph hierarchy.

The VSDG is also a directed acyclic graph, as it contains no backwards edges. The support for hierarchy enables loops to be represented as acyclic, infinite nested graphs using tail-recursion [Law07]. Figure 3.9b shows the representation of the *for*-loop shown in Figure 3.9a. Similarly, general recursive functions may also be represented using the hierarchical nature of the VSDG. Figure 3.10b shows the VSFG representation for the recursive Fibonacci function given in Figure 3.10a.

Lazy vs. Eager Execution Semantics

Lawrence and Johnson both recommended lazy execution (*pull*) semantics for the VSDG representation [Law07, Joh04]. Under lazy execution, only the transitions that are required to produce a result would be allowed to fire. For instance, for the VSDG in Figure 3.8b, the external environment would *pull* on the output places of this subgraph ($P_{out} = \{z_1, STATE_OUT, x_1\}$). This would trigger the computation of the x_0 increment, the $z_1 = p + v_0$, and the MUX transitions. At the MUX transition, a pull request would trigger the computation of the predicate ($u > 256$), after which, only the appropriate set

[5]Though in reality, any transition, compound or otherwise, may take a finite, possibly even variable amount of time to execute.



(a) The *for*-loop from Figure 3.6.

(b) The equivalent VSDG representation. Note that the loop only updates state, and does not produce a value output.

Figure 3.9: An example VSDG representation of a loop as a tail-recursive function. The outer double-border bounding box represents the definition of the *forloop(i, A, STATE)* function that is tail-recursively called within itself. Note the introduction of the *STATE* argument to the function definition.

of remaining predecessors would be triggered: if the predicate holds, only the $v_0 = v - x_0$ and the $printf(\text{“}branch\ true\text{''})$ transitions would be *pulled* and hence executed, while their false counterparts would not. Lazy evaluation constrains the execution of control-flow dependent side-effects and state-updates in this manner, ensuring that the observable semantics of the original program are preserved, even in the absence of an explicit flow of control in this program representation.

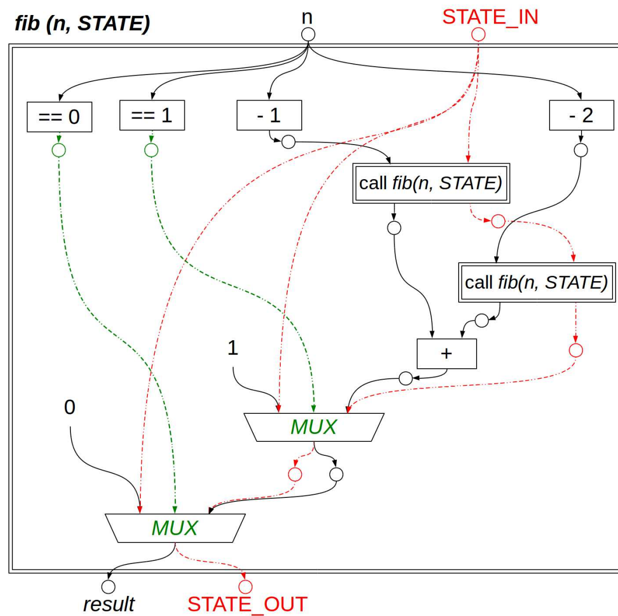
However, as the intent of this work is to improve performance, in particular by overcoming the constraints of control-flow by enabling speculative execution, we must modify the VSDG structure slightly to make it suitable for eager/dataflow evaluation semantics, while allowing for predication of State operations, and predicate promotion of Value operations. An additional reason for preferring eager evaluation is the relative simplicity of implementing dataflow graphs in hardware, compared to the complexity and overheads of implementing a graph reduction machine to support lazy evaluation [PJ87].

In order to support eager/dataflow execution semantics, while also permitting speculative execution of control-dependent operations, we must modify the VSDG slightly: in addition to the state input and output edges associated with each state operation, a *predicate* input edge must also be incorporated. Each basic block in the original program CFG will have an equivalent predicate expression implemented as additional boolean operations in the VSDG. Correct control over the execution of state operations is thus implemented by predicating their execution, as described previously in Section 3.2.2.

Figure 3.11 shows the VSDG from Figure 3.8b with the additional predicate edges incorporated. Note that with eager evaluation, all operations would be ready to execute as soon as their inputs are available. However, the state operations $printf(\text{“}branch\ true\text{''})$, and $printf(\text{“}branch\ false\text{''})$ have an additional predicate input, which ensures that only

```
int fib (int n) {
    If (n == 0)
        return 0;
    else if (n == 1)
        Return 1;
    else
        return (fib(n-1) + fib(n-2));
}
```

(a) The code for a recursive Fibonacci function.



(b) The equivalent VSDG representation. Note that this function produces both a state and a value output.

Figure 3.10: The VSDG representation of the recursive fibonacci function.

one of them will access/update state, depending on the value of the guard condition ($u > 256$). As in Figure 3.2c, the value operations 08 ($v_T = v - x_0$), and 10 ($v_T = v + x_0$) have been predicate-promoted, as they do not have any side-effects. To distinguish between the VSDG as described by Lawrence and Johnson, this predicated, eager-evaluation version of the VSDG will from now on be referred to as a Value State *Flow* Graph, or VSFG.

Similarly, Figures 3.12 and 3.13 show the VSFG equivalents of the VSDGs shown in Figures 3.9b and 3.10b, respectively. Again, thanks to predication of state operations, note that the next iteration of the for-loop (i.e. the recursive call to *forloop(i, A, STATE, pred)*), will only be triggered if the next loop predicate ($i < 100$) is true. The same is true for the nested subgraph implementing the function *foo(STATE, pred)*.

Generating Predicate Expressions for Eager Evaluation

The process of converting imperative code into an equivalent VSFG will be described in greater detail in Chapter 5. Here, I briefly describe how predicate expressions are generated for each basic block from the CFG of the input code. Part of the process for converting to the VSFG involves extracting all loops in the code into their own functions, and then transforming these into tail-recursive functions. This transformation means that each of the constituent functions in the application code will have acyclic control-flow graphs. Figures 3.14b and 3.14a show the acyclic control-flow graphs for a loop and a non-loop function, respectively, from the AES benchmark that forms part of the CHStone suite [HTH+08]. For the function CFG in Figure 3.14b, the basic block labelled *tailRecurseBlock* contains the tail-recursive call of the function to itself.

Given an acyclic control-flow graph, predicate expressions for each of the basic blocks in the graph may be computed relative to the first, or *entry* block. Recall that all State

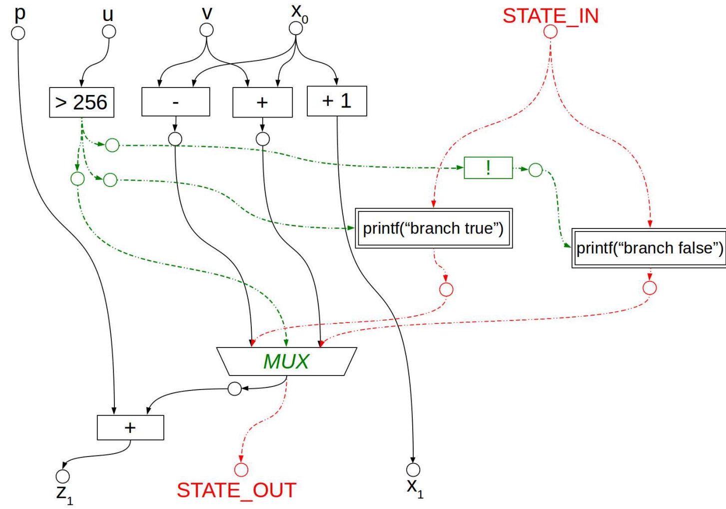


Figure 3.11: The equivalent VSFG representation to the VSDG from Figure 3.8b. Note that all State operations are now *predicated*, while Value operations are *predicate-promoted*.

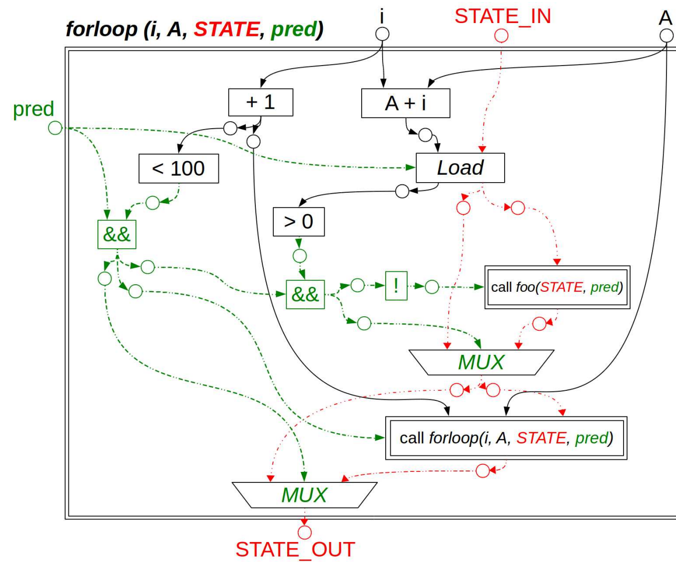


Figure 3.12: The equivalent VSFG representation to the VSDG from Figure 3.9b.

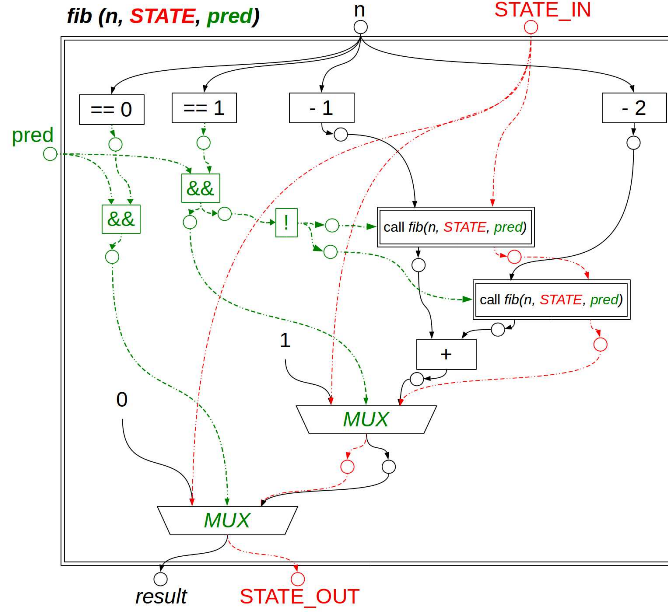


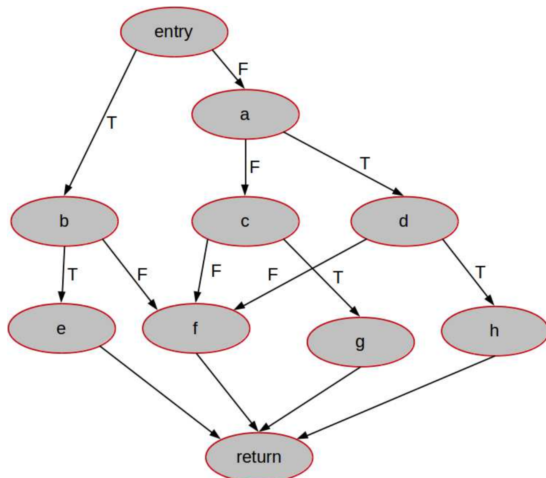
Figure 3.13: The equivalent VSFG representation to the VSDG from Figure 3.10b.

operations are to be controlled by predicates, including nested subgraphs representing function calls. Thus the predicate of each entry block itself would be the input predicate from the calling function, written as $inPred$. Let us denote the predicate associated with a block b as $p(b)$. Thus, $p(b)$ would drive the predicate inputs of all state operations in b – if $p(b)$ holds, then all state operations within b will execute non-speculatively, as soon as their remaining value and state-edge inputs are available.

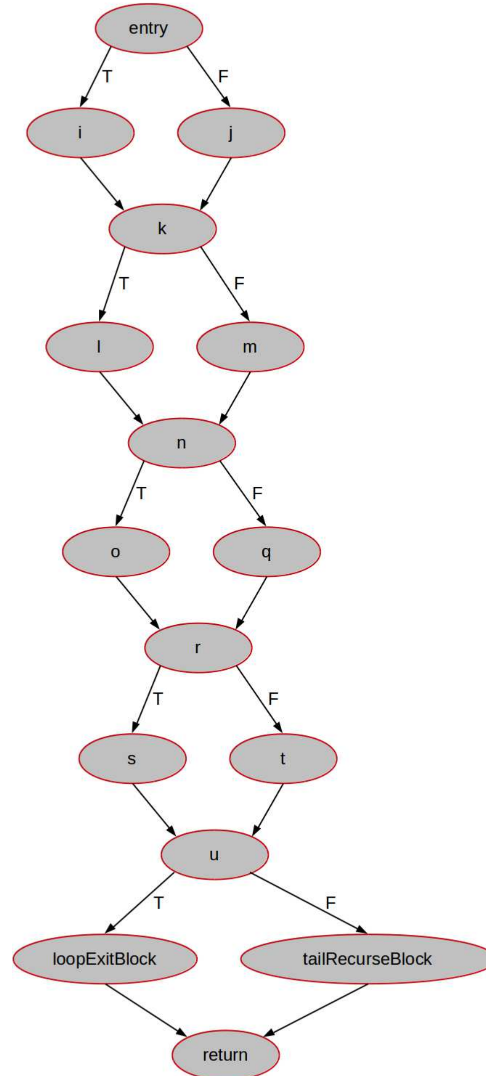
Similarly, a control-flow edge between two blocks b_1 and b_2 will also have an associated predicate, denoted by $p(b_1, b_2)$, essentially representing the condition evaluation for a conditional branch at the end of block b_1. If $p(b_1, b_2)$ holds, this implies that control will pass from block b_1 to block b_2, assuming control-flow has reached block b_1 (i.e. if $p(b_1)$ also holds). As such condition computations are typically value operations, and may execute speculatively in the VSFG, it is essential that the predicate for block b_2 be generated using both the block predicate for b_1, and the edge predicate for the edge from b_1 to b_2, as follows: $p(b_2) = p(b_1)\ .\ p(b_1, b_2)$. Using this notation, we can compute the predicate expressions for each of the basic blocks in an acyclic CFG. For the CFG from Figure 3.14a, the predicate expressions for each of the basic blocks can be computed as shown in Equations 3.1 and 3.2. Each basic block in the original CFG will therefore have an equivalent predicate expression implemented as additional boolean operations in the VSFG.

Since we have:

$$\begin{aligned} p(entry, b) &= \overline{p(entry, a)} \\ p(a, d) &= \overline{p(a, c)} \\ p(b, f) &= \overline{p(b, e)} \\ p(c, g) &= \overline{p(c, f)} \\ p(d, h) &= \overline{p(d, f)} \end{aligned} \tag{3.1}$$



(a) Acyclic CFG for a non-loop function.



(b) Acyclic CFG for a loop, extracted into its own, tail-recursive function.

Figure 3.14: Sample loop and non-loop CFGs from the AES benchmark in the CHStone Benchmark Suite [HTH+08]. These CFGs were generated using an LLVM pass, *after* preprocessing the benchmark code to extract loops into tail-recursive functions.

Therefore:

$$
\begin{aligned}
p(entry) &= inPred \\
p(a) &= p(entry)\ .\ p(entry, a) \\
p(b) &= p(entry)\ .\ \overline{p(entry, a)} \\
p(c) &= p(a)\ .\ p(a, c) \\
p(d) &= p(a)\ .\ \overline{p(a, c)} \\
p(e) &= p(b)\ .\ p(b, e) \\
p(f) &= p(b)\ .\ \overline{p(b, e)}\ + p(c)\ .\ p(c, f)\ + p(d)\ .\ p(d, f) \\
p(g) &= p(c)\ .\ \overline{p(c, f)} \\
p(h) &= p(d)\ .\ \overline{p(d, f)} \\
p(return) &= p(e)\ +\ p(f)\ +\ p(g)\ +\ p(h)
\end{aligned}
\tag{3.2}
$$

Enabling Control-Dependence Analysis

As boolean expressions, predicates for each basic block may be minimized. Consider, for instance the predicate $p(return)$ for the *return* block in Equations 3.2. Expanding the constituent terms, we get:

$$
\begin{aligned}
p(return) &= p(e)\ +\ p(f)\ +\ p(g)\ +\ p(h) \\
&= \begin{array}{r} p(b)\ .\ p(b, e)\ +\ p(b)\ .\ \overline{p(b, e)}\ + p(c)\ .\ p(c, f)\ + p(c)\ .\ \overline{p(c, f)}\ + \\ p(d)\ .\ p(d, f)\ + p(d)\ .\ \overline{p(d, f)} \end{array} \\
&= p(b)\ +\ p(c)\ +\ p(d) \\
&= p(entry)\ .\ \overline{p(entry, a)}\ + p(a)\ .\ p(a, c)\ +\ p(a)\ .\ \overline{p(a, c)} \\
&= p(entry)\ .\ \overline{p(entry, a)}\ +\ p(entry)\ .\ p(entry, a) \\
&= p(entry) = inPred
\end{aligned}
\tag{3.3}
$$

This implies that the state operations in the *return* block from Figure 3.14a may execute non-speculatively as soon as the input predicate to the function subgraph, $inPred$, evaluates as 'true'. Applying such logic minimization to the predicate expressions for each basic block enables us to incorporate *control-dependence analysis* into the VSFG. All predicate expressions within a function may be evaluated in dataflow order, and be available concurrently in the eagerly-evaluated VSFG. This allows even the predicated state-operations to non-speculatively execute earlier than in a CFG-based spatial implementation, where control must flow through the CFG, enabling the non-speculative execution of state operations one basic block at a time. The minimized boolean predicate

expressions for the loop CFG of Figure 3.14b are given by Equations 3.4.

$$\begin{aligned}
p(entry) &= inPred \\
p(i) &= inPred \,.\, p(entry, i) \\
p(j) &= inPred \,.\, \overline{p(entry, i)} \\
p(k) &= inPred \\
p(l) &= inPred \,.\, p(k, l) \\
p(m) &= inPred \,.\, \overline{p(k, l)} \\
p(n) &= inPred \\
p(o) &= inPred \,.\, p(n, o) \\
p(q) &= inPred \,.\, \overline{p(n, o)} \\
p(r) &= inPred \\
p(s) &= inPred \,.\, p(r, s) \\
p(t) &= inPred \,.\, \overline{p(r, s)} \\
p(u) &= inPred \\
p(loopExitBlock) &= inPred \,.\, p(u, loopExitBlock) \\
p(tailRecurseBlock) &= inPred \,.\, \overline{p(u, loopExitBlock)} \\
p(return) &= inPred
\end{aligned} \tag{3.4}$$

From Equations 3.4, we see that state operations in blocks $entry$, k, n, r, u, and $return$ may execute non-speculatively, as soon as the $inPred$ predicate input is available. Execution of these state-operations would thus be constrained primarily by the sequential state-edge traversing each such operation in the VSFG. Furthermore, the next loop iteration (i.e. tail-recursive loop call in the *tailRecurseBlock*) may be initiated as soon as the expression $\overline{p(u, loopExitBlock)}$ can be computed.

Such control-dependence analysis can also be applied when converting acyclic control-flow into dataflow within a hyperblock, via if-conversion [MLC+92], as has been applied by Budiu in his implementation of spatial computation [Bud03]. However, the key advantage of switching away from the CFG towards a VSDG-based intermediate representation is that the latter is fully acyclic, thus this form of control-dependence analysis is not restricted to within acyclic regions of an existing control-flow graph, but can be applied across the entire code, and within each level of the VSDG/VSFG graph hierarchy.

These predicate expressions, in conjunction with the MUX operators, serve to effectively convert all control-flow in the program into dataflow, while the state-edge does the same for side-effect ordering. Applying minimization on these expressions can further expose concurrency by enabling aggressive control-dependence analysis, as described by Lam [LW92].

3.6.2 Revisiting Case Studies 1 and 2

The Value State Flow Graph represents a good starting point for overcoming the performance limitations faced due to control flow in spatial hardware for three reasons:

- The VSFG represents data-dependencies explicitly, but control-flow only implicitly. There is no notion of a program counter, or a selection between executing basic

blocks. Instead, control is implemented using a form of *if*-conversion (selection between speculatively computed values based on predicates) and predicate-promotion (predication of state operations only). Since control-flow is the primary bottleneck to sequential code performance [LW92, Wal91, MM09], the VSFG would provide a better option for describing spatial dataflow hardware than the CFG, particularly when attempting to accelerate control-intensive sequential code.

- The VSDG/VSFG is acyclic, representing loops as an infinite sequence of forward branches. This can be beneficial in addressing the limitations of custom hardware with respect to addressing speculation across backwards branches[6].

- The VSDG/VSFG is a gated-data dependence representation containing operations that have convenient equivalent representations in hardware. For instance, the MUX-node utilized in the VSFG to represent value selection based on control-flow is equivalent to a multiplexer in hardware. This is in contrast to its counterpart, the ϕ-node that is utilized in static single assignment CFG IRs for the same purpose, but has no direct hardware analogue (operations 14 and 15 in Figures 3.2b, and 3.2c are examples of the usage of ϕ-nodes in SSA form).

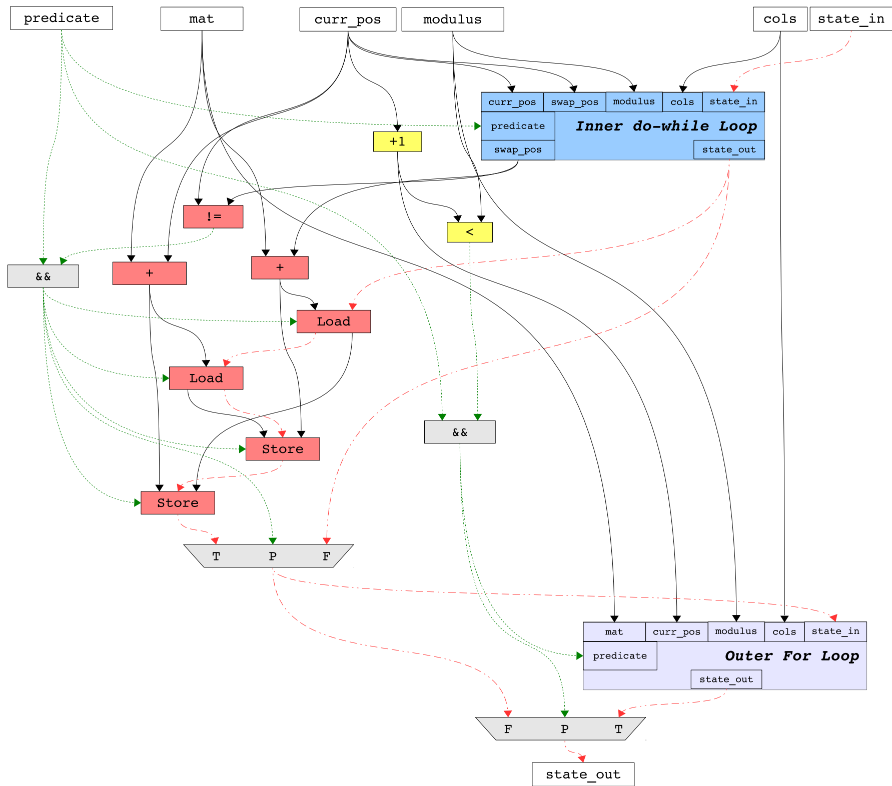
Figure 3.15 presents the VSFG equivalent to the CDFG from Figure 3.5a (and the code in Figure 3.4). In Figure 3.15a, the outer loop contains the inner-loop in a nested-subgraph represented by the block labeled *'Inner do-while Loop'*, the contents which are shown in Figure 3.15b. Similarly, the next iteration of the outer-loop itself is represented as a tail-recursive call to the nested-subgraph marked *'Outer For Loop'*.

Outer-loop Pipelining *without* Dynamic-dataflow Overheads: One key advantage of this lack of explicit control flow, combined with nesting of subgraphs is the ability to perform loop unrolling and pipelining at multiple levels of a loop nest. Any of the nested subgraphs in a VSFG can be *flattened* into the body of the parent graph [Law07]. In the case of loops, flattening the subgraph representing the tail-recursive loop call is essentially equivalent to unrolling the loop. Figure 3.16 shows the *'Outer For Loop'* subgraph in Figure 3.15 flattened/unrolled once.

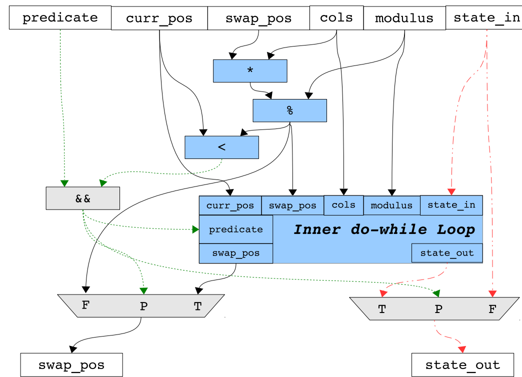
Furthermore, as each loop is implemented within its own subgraph, this type of unrolling may be implemented within different subgraphs independently of others. Therefore, it is possible in the VSFG to exploit ILP within a loop nest by unrolling the inner loops independently of the outer loops. Thus, for the example code in Figure 3.4 that performs poorly when implemented as custom hardware, or even CDFG based spatial computation [BAG05], by utilizing the VSFG as our dataflow IR, we are able to exploit greater ILP through *outer-loop pipelining* in the same manner as the superscalar processor, simply by flattening the nested-subgraph representing the outer-loop tail-recursive function call any number of times.

The VSFG can therefore more aggressively overcome the effects of loop-carried name dependencies by enabling outer-loop parallelism in energy-efficient static-dataflow hardware, without incurring the exorbitant energy cost of complex register-renaming logic (as in superscalar processors) or instruction-operand tag matching, wake-up and select logic

[6]Note however that in order to implement the VSFG representation with a finite amount of custom hardware, dataflow cycles will need to be carefully reintroduced to implement loops using finite resources. This is described in greater detail in Chapter 4.



(a) VSFG for *'internal_int_transpose'* Outer Loop.



(b) VSFG for *'internal_int_transpose'* Inner Loop.

Figure 3.15: VSFG for the outer for-loop of *'internal_int_transpose'*, showing the inner loop as a nested subgraph. The next iteration of both outer and inner loops is also represented as nested subgraphs, essentially implementing loops as tail-recursive functions.

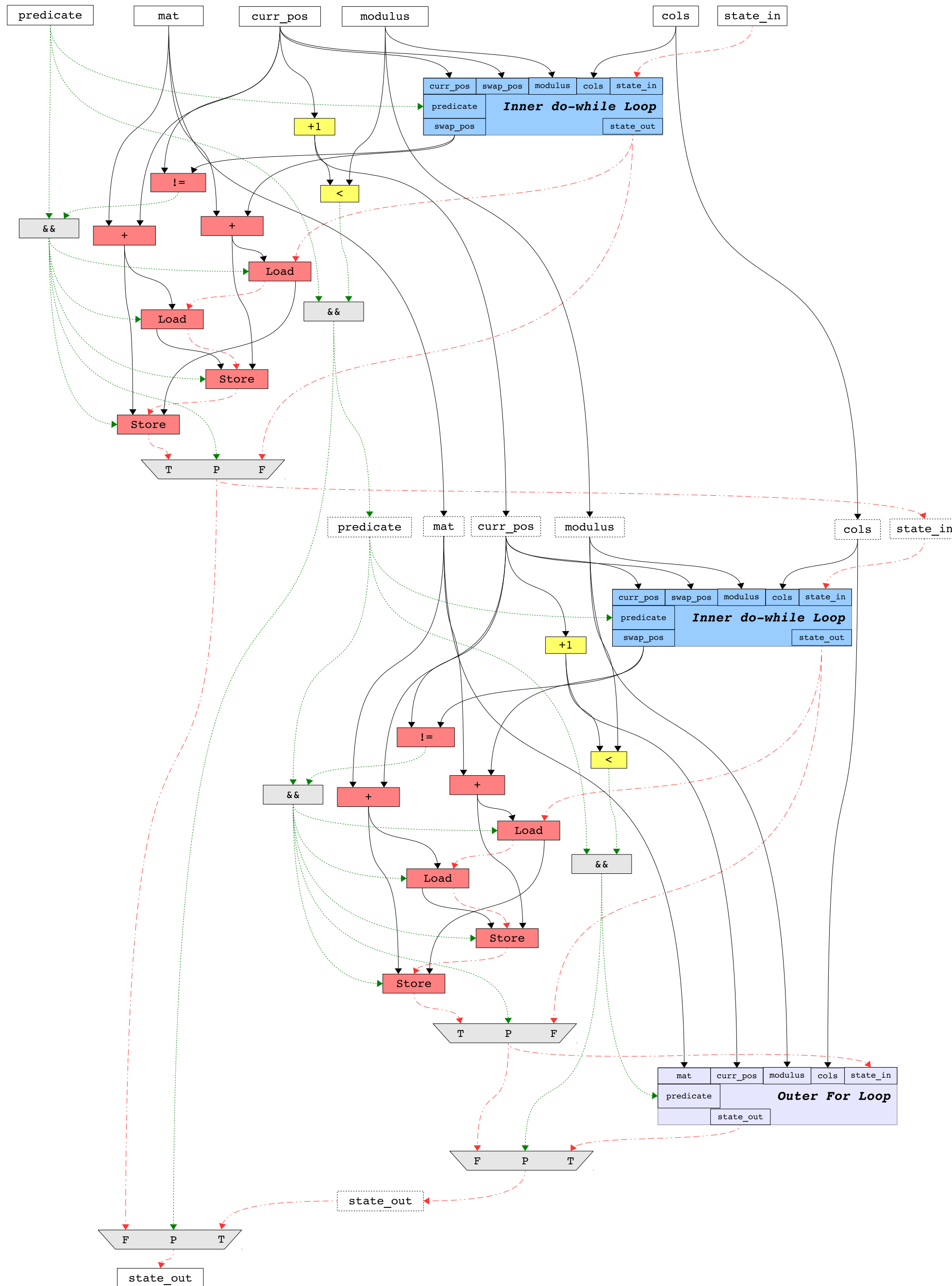


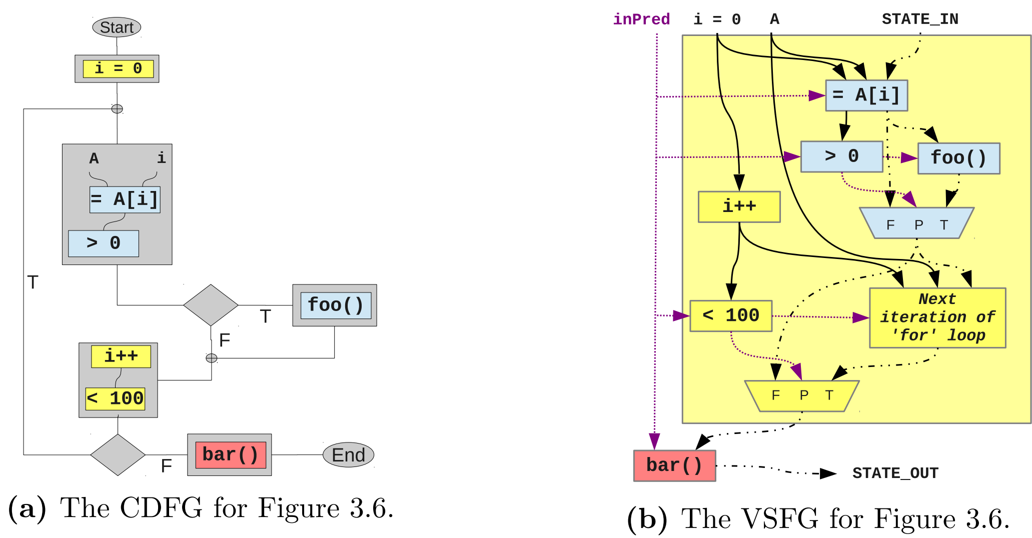
Figure 3.16: Loop *unrolling* is implemented by *flattening* the loop's tail recursive call subgraph a number of times. Here, the VSFG of the *Outer For Loop*, is unrolled once, implementing two copies of the inner loop. Each loop in a loop nest may similarly be flattened/unrolled independently of the others to expose loop parallelism.

(used by both superscalar processors and Wavescalar), etc. that is needed to approximate the dynamic-dataflow execution model.

Note that in the actual hardware implementation, cycles must be reintroduced to convert from the infinite, acyclic representation of loops into finite hardware. However, this can be done after the appropriate degree of unrolling/flattening has been achieved for each loop-level, and is described in greater detail in Chapter 4.

Aggressive Static Control Speculation: All forward branches support speculative execution of operations through if-conversion and predicate promotion. Given that loops are represented as infinite, hierarchical sequence of forward-branches (at least until cycles are reintroduced for hardware implementation), the VSFG representation is able to support speculative execution on all branches.

Such a high degree of speculation can potentially incur a high energy cost, as only a few of the speculatively executed operations in a VSFG will produce results that are not discarded at the multiplexers. Thanks to the hierarchical nature of the VSFG, it is possible to control the degree of speculation by selectively controlling the execution of nested-subgraphs. For instance, for the *'Inner do-while Loop'* subgraph, we may choose whether the contents of this subgraph execute speculatively or not: the predicate input to it may be used to only allow its execution when the predicate is true, thereby providing no speculation. Alternatively, the subgraph may start executing irrespective of the predicate value, in which case, the predicate value will be passed into the subgraph, where side-effect free operations may execute speculatively even before the predicate value becomes available. All side-effecting operations will however always be predicated.



(a) The CDFG for Figure 3.6.

(b) The VSFG for Figure 3.6.

Figure 3.17: The equivalent CDFG and VSFG for the code given in Figure 3.6.

Exploiting Multiple Flows of Control: Another advantage of having control flow converted in to boolean predicate expressions is the ability to perform *control dependence analysis* to identify regions of code that are control-flow equivalent and may therefore execute in parallel – provided all state and dataflow dependencies are satisfied. Consider the *bar()* function in the code given in Figure 3.6 (the equivalent CDFG is given in Figure 3.17a). Despite aggressive branch prediction, a superscalar processor will not

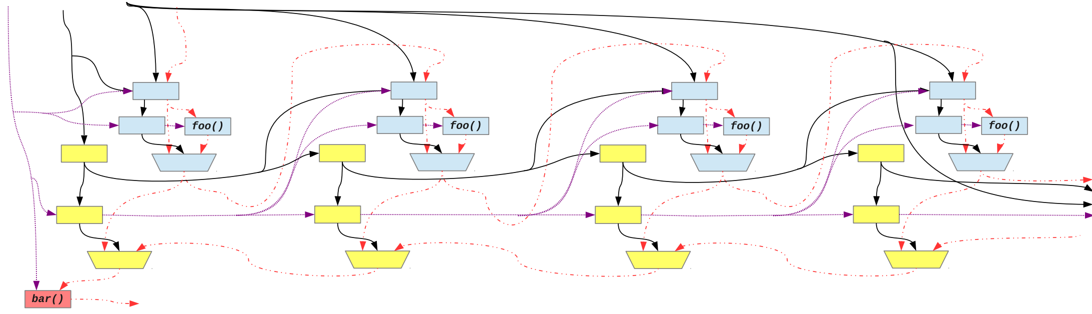


Figure 3.18: The VSFG from Figure 3.17b with the loop unrolled 4 times.

be able to start executing *bar()* until it has exited the loop. Similarly, when the *if* branch is predicted-taken, the superscalar processor must switch from executing multiple dynamically unrolled copies of the for-loop and instead focus on executing the control-flow within *foo()*. This is because a conventional processor can only execute along a single flow of control [LW92].

The VSFG on the other hand can apply boolean expression optimization on the control-predicte expressions of each basic block to identify the control-flow equivalence between the contents of the for-loop and the *bar()* function. Thanks to this implicit *control dependence analysis* [LW92], independent instructions from within the *bar()* function may start executing in parallel (speculatively or otherwise) with the *for*-loop.

Furthermore, so long as their dataflow and state-ordering dependencies are resolved, the contents of the *foo()* subgraph can also execute in parallel with its parent graph. If we combine this concurrency with loop unrolling as shown in Figure 3.18, it becomes possible to execute multiple copies of the loop and *foo()*, in parallel with the execution of *bar()*. This ability to execute multiple regions of code concurrently is equivalent to enabling execution along multiple flows of control, and has the potential to not only expose greater ILP than even a superscalar processor, but possibly also to break through the ILP Wall as well [LW92, MM09].

Potential performance limitations: In the simplest form of the VSFG or the VSDG, there is a single valid state-token in the graph at any time, i.e. there is a total-order imposed on the execution of all state access and side-effect sensitive operations throughout all levels of the graph hierarchy. This implies that after a certain amount of unrolling, the sequentializing state-edge will often be on the dynamic critical path[7] through the VSFG, limiting the achievable ILP.

Further improvements to ILP will require *partitioning* this state-edge, so that multiple memory operations to disjoint locations may occur concurrently, and possibly out of order. As mentioned earlier, conventional superscalar processors often have support for aggressive memory operation reordering at their load-store queues to enhance performance, without visibly violating the sequential order on side-effects. However, this approach requires complex hardware structures and incurs a significant energy cost. In order to avoid this, in the first instance, the compiler should rely on alias and memory dependence analysis to perform static memory disambiguation, enabling partitioning of the sequential state-edge chain. This partitioning would convert the total order on State operations into a

[7]i.e. the longest dependency chain of operations in a pipelined implementation of the graph, as opposed to the longest combinational path between registers when implementing custom hardware.

partial order, permitting concurrent updates to disjoint memory locations to improve performance as much as possible without incurring an additional energy cost.

Such static partitioning is employed by many existing high-level synthesis tools, like LegUp [CCA+11] and Kiwi [SG08]. Budiu et al have even augmented this static partitioning by incorporating dynamic memory disambiguation hardware in their generated hardware [Bud03, BVCG04]. In this thesis, static memory disambiguation through alias analysis has not been incorporated in the results, partly due to constraints of time, and partly because the effects of such optimizations for high-level synthesis have already been studied extensively.

3.7 Related Work on Compiler IRs

As discussed in Section 2.4.3, The most common compiler IRs used for compiling to spatial architectures are closely related to the CDFG: RICA DySER, TRIPS, CASH/Phoenix, and Conservation Cores all utilize a version of the CDFG, often with optimizations like loop unrolling, if-conversion and hyperblock formation applied to reduce the effects of control-flow on performance. RICA and DySER only exploit spatial computation for loop-free regions of code. However, recent academic research describes several innovative projects developing compiler IRs for spatial computation.

Program Dependence Graph (PDG)

The PDG represents a program as a graph incorporating two types of edges: one set to represent data dependences and another to represent control [FOW87]. Unlike the CDFG that expresses control *flow* between basic blocks, a PDG control edge expresses explicit fine-grained control *dependence* between individual operations. The PDG also allows grouping control-equivalent operations into a single *region*, but unlike the CFG, instructions within such regions may be selected from multiple basic-blocks. This balanced treatment of both control and data dependences allows the PDG to simplify several important code transformations, particularly those that require the interaction of both control and data dependences [FOW87].

Variants of the PDG that only represent true data dependences (i.e. the PDG+SSA: a combination of the PDG with the static-single assignment representation) have repeatedly proved useful for partitioning single threaded code into multiple communicating threads: Li et al [LPC12] use an 'SSA-PDG' to identify strongly connected components (SCC) in a program graph, considering both control and data dependence edges. Each such SCC may then be assigned for execution in a different thread on a multithreaded processor as a means of converting imperative code for execution as coarse-grained hybrid dataflow on a multicore architecture. A similar technique is used by [Ott08].

The PDG has also been utilized for the generation of higher quality custom hardware [GWK04]. A PDG+SSA IR is generated from the input CFG of an application. Like in the VSFG, the control-dependence edges in the PDG+SSA are implemented as dataflow predicate edges, and SSA ϕ-nodes are converted into *MUX* elements. Unlike the boolean minimization that may be applied to simplify control-dependence in the VSFG, the PDG-SSA instead uses control-flow dominance and post-dominance relationships (similar to what is described in Section 5.2.3). Gong et al demonstrate that due to such optimizations, their PDG+SSA based, statically-scheduled custom hardware ex-

hibits a 7% performance improvement over their equivalent CDFG based hardware on various multimedia kernels from the Mediabench benchmark suite [GWK04].

Unlike the VSFG, the PDG does not exploit hierarchy to represent loops, and therefore can contain explicit control and dataflow cycles[8]. This, combined with the statically scheduled execution model employed by Gong et al makes this approach unable to exploit outer-loop pipelining as demonstrated for the VSFG.

Hierarchical Control Dependence Graph (HCDG)

Like the VSFG, the HCDG IR was developed to address the implementation of applications that exhibit complex control-flow as well as dataflow [KW02]. Unlike the VSFG, the HCDG emphasizes the traditional, statically scheduled execution model, and is designed to efficiently implement scheduling, allocation, and binding during high-level synthesis. Like the PDG+SSA, the HCDG also incorporates both control and true dataflow edges. But instead of representing control dependences as in the PDG, control is represented using boolean expressions just like in the VSFG[9], and therefore is subject to boolean minimization as well.

However, unlike the VSFG, the HCDG does not usually apply predicate promotion to non-side-effecting operations. Instead, this *hierarchical* predicate information for each operation in the IR is used to perform efficient static-scheduling, allocation, mapping during HLS, for instance, by establishing the mutual exclusivity of various operations due to diverging control-flow in the original program graph. Predicate promotion is only applied when speculative execution is needed.

Unlike Pegasus, but like the VSFG and the PDG+SSA [GWK04], the HCDG does not incorporate the notion of control-directed dataflow. Although, the HCDG does not currently support the representation of loops, Kountouris and Wolinski recommend introducing the notion of a *composite* node representing the loop body for handling loops in the future [KW02]. This would be similar to how the VSFG and VSDG represent loops as nested subgraphs.

Given their commitment to statically scheduled execution, they recommend the generation of a hierarchical FSM[10], with each composite loop node having its own FSM, communicating with a higher level FSM that manages execution in the parent graph. Code with multiple levels of nested loops would therefore be managed by an FSM with corresponding levels in its hierarchy. This approach would allow for independent composite nodes to be executing concurrently, but it remains unclear if this approach would be suitable for exploiting inter-iteration parallelism in loops without unrolling, or outer-loop pipelining.

Given this suggestion for hierarchical construction as a means of handling loops, the HCDG becomes very similar to the VSFG, except that due to its commitment to static execution scheduling, it does not need to incorporate state-edges to preserve the order of side-effecting operations. Given an application with complex control flow, the HCDG is able to better exploit 'conditional resource sharing' to improve the resource requirements of the generated hardware over existing list-scheduling based approaches [KW02]. The

[8] thus, well-behavedness issues must be considered for static-dataflow implementations.

[9] The *'Hierarchical'* prefix in the name of this IR refers to this construction of complex tree of boolean expressions based on the structure of the CFG, and not to the type of hierarchy that has been discussed thus far in reference to the VSDG and VSFG.

[10] here the word *hierarchy* is used in a similar fashion as the VSDG/VSFG.

present work on the HCDG IR may prove a useful starting point if the development of a statically-scheduled VSFG implementation is ever desired.

Control-Memory Dataflow Graph (CMDFG)

The CMDFG IR is utilized in the COMRADE compiler for 'adaptive' computing systems [GK07], that generates static-dataflow custom hardware from hot regions of imperative code. The CMDFG can be seen as an amalgam of the features of the PDG+SSA, the HCDG and the Pegasus IR.

Like the HCDG, the CMDFG has no control-directed dataflow within acyclic regions, and instead relies on boolean predicate expressions instead. Like the Pegasus and VSFG IRs, it targets the static-dataflow execution model, and thus must incorporate a sequentializing state-edge between side-effecting operations. Like the PDG+SSA, the CMDFG replaces SSA ϕ-nodes with *MUX* nodes, and it too allows cycles in the graph to represent loops[11], instead of the hierarchical approach favoured by the VSFG and the HCDG. Lacking this notion of hierarchy, the CMDFG thus cannot exploit multiple flows of control or outer-loop pipelining as proposed for the VSFG.

However, the key contribution made by this work was to improve the performance of speculative execution through the use of '*cancel tokens*' [GK08]. Speculation is implemented in a static-dataflow graph via if-conversion: all speculative datapaths are allowed to execute concurrently, with only the results from the true path being selected at a *MUX*. However, if such a speculative datapath is enclosed within a loop body, the throughput of the loop will be constrained by the longest of the speculative paths, even if it is not the most frequently used path[12]. In order to avoid this unnecessary latency along falsely speculated paths, *MUX* elements in the CMDFG are able to produce *cancel tokens* that traverse the static-dataflow graph *backwards* along all false paths, once the correct *MUX* predicate value is known. These cancel tokens are used to kill computations that may be in progress in the false paths, thereby reducing the unnecessary latency incurred, and improving loop throughput.

Gadke and Koch are able to demonstrate a 3× improvement in performance for an example code kernel compared to the baseline, and a 1.8× improvement over *lenient MUX* execution[13] [GK08]. This novel approach would also be useful for improving performance of such loops in VSFG based implementations, as discussed later in Section 6.2.1. For now, the inclusion of cancel tokens into the VSFG is left for future work.

3.8 Summary

This chapter started by studying the nature of sequential, imperative code, and the issues that arise in exposing and exploiting fine-grained concurrency from it. I highlighted

[11]Note that as the CMDFG targets static-dataflow execution, well-behavedness issues must be taken into account when implementing code with such loops. Section 4.2.1 discusses well-behavedness in dataflow computation graphs.

[12]In cases where there are no loop-carried dependences, techniques like *pipeline balancing* may be utilized [Gao91] to maximize throughput at the expense of increased latency along the shorter speculative paths. However this may not help for loops that have loop-carried dependences. A lengthier discussion on this issue is presented in Section 6.2.1, and Figure 6.4.

[13]Lenient execution is also used by the VSFG, and is described in greater detail in Sections 4.3.1 and 4.3.2.

the fact that superscalar processors achieve high sequential performance by performing aggressive branch prediction to overcome control-flow, and approximate the dynamic-dataflow execution model to accelerate true dependencies and mitigate the impact of unpredictable latencies at runtime. I also noted that while such processors are constrained by the ILP Wall, it may be possible to exploit more ILP from code by exploiting multiple-flows of control.

To match, or even exceed superscalar performance in hardware, I proposed (a) switching to the static-dataflow execution model for spatial hardware, in order to retain high energy-efficiency while providing dynamic execution scheduling, and (b) switching from CFG-based compiler IRs to a VSDG-based IR – the Value State Flow Graph – that represents data and side-effect ordering dependencies explicitly, enables aggressive control speculation, control-dependence analysis, and exploitation of multiple-flows of control. By enabling the independent unrolling of each loop in a loop-nest, the VSFG could even enable exploiting concurrency in nested loops via outer-loop pipelining, without needing to incur the area and energy overheads associated with the dynamic-dataflow execution model.

It is important to note that unlike a conventional processor, or even high-performance spatial architectures like TRIPS and DySER, the proposed approach relies solely on static, compile-time manipulation of the program IR to expose greater ILP from control-flow intensive code. Chapter 4 formally describes the operational semantics of the *VSFG-S*: a version of the VSFG IR that is suitable for implementation as static-dataflow custom hardware. Chapter 5 then describes the implementation of a toolchain that compiles high-level language code to the VSFG-S IR, and then implements it as dataflow custom hardware.

CHAPTER 4

Definition and Semantics of the VSFG

Chapter 3 described the Value State Dependence Graph (VSDG) and how its structure could facilitate aggressive static exposition of ILP from control-intensive sequential code, by (a) enabling aggressive speculation through if-conversion, (b) employing control-dependence analysis to accelerate predicated operations, and (c) exposing ILP from across multiple flows-of-control. I highlighted the need to develop an eager-evaluation, dataflow version of the VSDG, called the Value State Flow Graph (VSFG). The objective of developing the VSFG program representation is to facilitate the implementation of conventional imperative languages on spatial architectures with minimal programmer intervention, while matching the performance of conventional superscalar processors.

This chapter describes in greater detail the structure and semantics of a version of the VSFG that may be implemented directly as custom or reconfigurable hardware. This constrained version of the VSFG may be referred to as the VSFG-S, where the appended 'S' specifies the static-dataflow nature of the execution model being targeted, as opposed to the pure dataflow VSFG representation from Chapter 3[1].

4.1 The VSFG as Custom Hardware

In order to evaluate the performance and energy characteristics of spatial implementations using the VSFG-S, a prototype high-level synthesis toolchain is developed that compiles imperative code into the VSFG representation, and then implements it as static-dataflow custom hardware described using standard Verilog HDL. This approach was selected for the following reasons:

- Custom hardware is considered an important target for implementing computation in the Dark Silicon era [VSG+10]. Our compiler IR must be able to efficiently and effectively compile to custom hardware while exhibiting better performance than hardware generated using existing HLS tools, and retaining the inherent efficiency advantage of custom hardware.

[1] However, from Chapter 5 onwards, 'VSFG' and 'VSFG-S' are used interchangeably, unless explicitly stated otherwise.

- There is no need to design and optimize a target spatial architecture (e.g. a CGRA) for the VSFG, nor to build specialized compilation (i.e. place and route) tools to map the VSFG to such an architecture. Generating custom hardware representation in a standard HDL from the VSFG IR allows us to utilize existing prototyping substrates and tools, like FPGAs.

- High-level synthesis to application-specific hardware allows us to evaluate the performance and efficiency potential of the IR itself, without the evaluation being complicated by the characteristics and design trade-offs inherent in any particular spatial architecture. In fact, the development of a spatial architecture is best treated as a separate (albeit related) research project, wherein a thorough design space exploration must be performed to develop a well tuned architecture with the appropriate performance, energy and area characteristics for its target application domain(s). Such an approach is being pursued by the Loki project [Bat14].

In addition to the incorporation of predicate expressions to control the execution of side-effects and compound operations in the eagerly-evaluated VSFG (Section 3.6), the following issues must also be considered when trying to use the VSFG to directly represent a static-dataflow machine in custom hardware:

1. **Developing dataflow semantics for VSFG Operations**: Unlike the ϕ-nodes from SSA form, we must ensure that all of the constituent operations in a VSFG can be implemented easily and efficiently in hardware. This includes not only the value, memory-access, and *MUX* operations, but also determining how compound VSFG operations representing nested-subgraphs may be implemented in hardware. Section 4.3.1 describes push semantics for all basic operations, while Section 4.3.2 describes the representation of compound operations in custom hardware.

2. **Implementing loop VSFGs in finite hardware**: The VSFG, like the VSDG, thus far represents each loop as an infinite directed-acyclic graph. However, cycles must be reintroduced into the VSFG in order to implement it with finite custom hardware. Section 4.3.3 describes how dataflow back-edges and limited control-dependent dataflow are reintroduced into the VSFG in order to implement loops, without negating its advantages described in Chapter 3.

3. **Maintaining *well-behavedness* of dataflow graphs**: With the reintroduction of back-edges into the VSFG, finite hardware resources such as hardware operations, registers and wires will be 'reused' at runtime. In this case, maintaining the *well-behavedness* of the VSFG IR across all subgraphs is essential for correct execution of dataflow graphs[2]. Section 4.2.1 discusses well-behavedness and how the VSFG must be constrained to be well-behaved.

Petri-nets are well suited to representing dataflow execution. Continuing from the definition of the structure of the VSFG as a Petri-net (Definition 3.1), the next few sections describe the structure and operational semantics of the VSFG-S, as well as discussing how each of the above issues is addressed in this representation.

[2]Especially in the case of static-dataflow, where each output place only holds one value at a time.

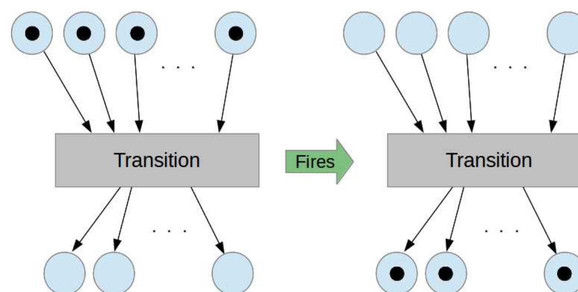
4.2 Modeling Execution with Petri-Nets

Consider a generalized dataflow operation with n inputs and m outputs, as shown in Figure 4.1a. In the abstract or *pure* dataflow execution model, operation execution is only constrained by the availability of values on the input edges [AC86]. Once one input value is available on each of the n input edges, the the dataflow operation may execute, consuming one value from each of its input edges and simultaneously producing one value on each of m output edges.

The pure/abstract dataflow execution model holds the presumption of *unbounded FIFO buffers* – meaning that each edge in a dataflow graph may hold an arbitrary number of values, that are consumed by a consumer in the order they were produced [AC86]. A static dataflow machine, however, allows only one value to be buffered on each edge at a time [AC86]. In this case, the dataflow operation from Figure 4.1a can only execute when one input value is available on each of its n input edges, *and* the buffers on each of the m output edges is empty.

(a) Generalized Dataflow operation with n inputs and m outputs

(b) Unbounded Petri-net representation with n input and m output places



(c) Petri-net from Figure 4.1b firing.

Figure 4.1: Representations of a generalized Dataflow operation using Petri-nets.

Dataflow execution can be modelled effectively using Petri-nets [Buc93]. Similar to the dataflow operation from Figure 4.1a, a Petri-Net transition is ready to *fire* when each of its input places have at least one token. Figure 4.1b shows the equivalent Petri-net representation of the dataflow operation from Figure 4.1a. A Petri-net used to model dataflow execution semantics is represented as a four-tuple $G = (P, T, E, M_0)$, where P, T, and E are the same as in Definition 3.1, while M_0 is the *initial marking*[3].

[3] Wang [Wan07] defines a marking as follows: "A marking in a petri net is an assignment of tokens

Given the shown initial marking in Figure 4.1c, the equivalent transition (operation) *fires*, removing tokens (values) from its input places and placing tokens into its output places. Petri-nets of this type are useful for modelling abstract or pure dataflow, as there is no restriction on the number of tokens that any given place may hold (i.e. Petri-net places are *unbounded*), and hence useful for modelling unbounded buffers on each dataflow arc. In order to model static-dataflow execution, we constrain our Petri-net to be a *homogeneous*, *marked-graph*, that is *1-bounded* and *safe* [Buc93]:

- *Homogeneous Petri-nets:* For a Petri-net, homogeneity means that a firing transition removes only one token from each input place, and adds only one token to each output place.

- *Marked-graph Petri-nets:* A Marked graph is a Petri-net in which every place has exactly one input transition and one output transition. Marked-graphs are useful in modelling deterministic Petri-net execution. The use of homogeneous, marked graphs is well-established for representing dataflow computation [Buc93].

- *1-bounded Petri-nets:* From Wang [Wan07]: "A place p is said to be *k-bounded* if the number of tokens in p is always less than or equal to k (k is a nonnegative integer number) for every marking M reachable from the initial marking M_0". For a Petri-net to model static-dataflow, each place in the Petri-net must be constrained to hold only a single value at a time, i.e. each place in the Petri-net must be *1-bounded.*

- *Safe Petri-nets:* A Petri-net $G = (P, T, E, M_0)$ is *safe* if each place $p \in P$ is 1-bounded.

Safeness can be enforced by adding a set of *acknowledgement places* to the existing set of places in the Petri-net:

$$P' = P_A \cup P, \text{ where } P_A = \{p'_k \mid k \in \mathbb{N} \wedge p_k \in P\} \tag{4.1}$$

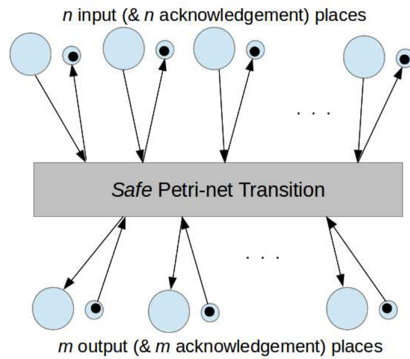
The new set of places P' thus contains an additional acknowledgement place $p'_k \in P_A$ for each of the places $p_k \in P$, with the property that p'_k would have a token if and only if p_k does not have a token. Given that each transition $t \in T$ in the Petri-net will have a set of input places $I(t) \subset P$, and a set of output places $O(t) \subset P$, this is implemented by:

- adding p'_k to the output set $O(t)$ of transition t, if p_k is in the input set $I(t)$ of transition t, or

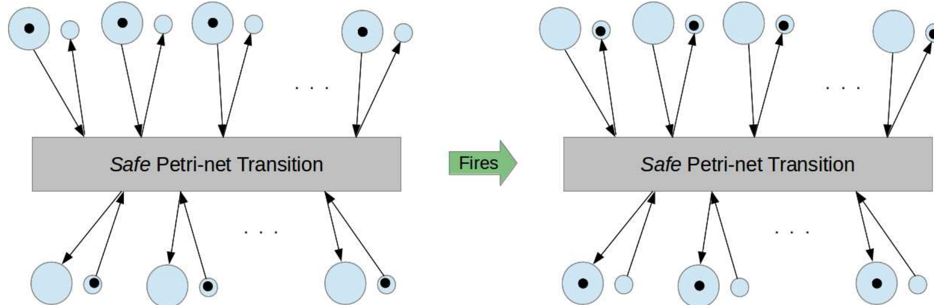
- adding p'_k to the input set $I(t)$ of transition t, if p_k is in the output set $O(t)$ of transition t.

A safe Petri-net used to model static dataflow execution will have an initial marking where all the acknowledgement places have a token, and all the value places will be empty, as shown in Figure 4.2a. Triggering the execution of such a Petri-net involves placing

to the places of a Petri-Net. Tokens reside in the places of a Petri-net. The number and positions of tokens may change during the execution of a Petri-net. The tokens are used to define the execution of a Petri-Net."



(a) *Safe* Petri-net representation with n input and m output places, together with corresponding acknowledgment places, also showing an initial marking.



(b) Petri-net from Figure 4.2a firing after the input places have valid tokens and the input acknowledgement places have been cleared.

Figure 4.2: Representations of a generalized Dataflow operation using Petri-nets.

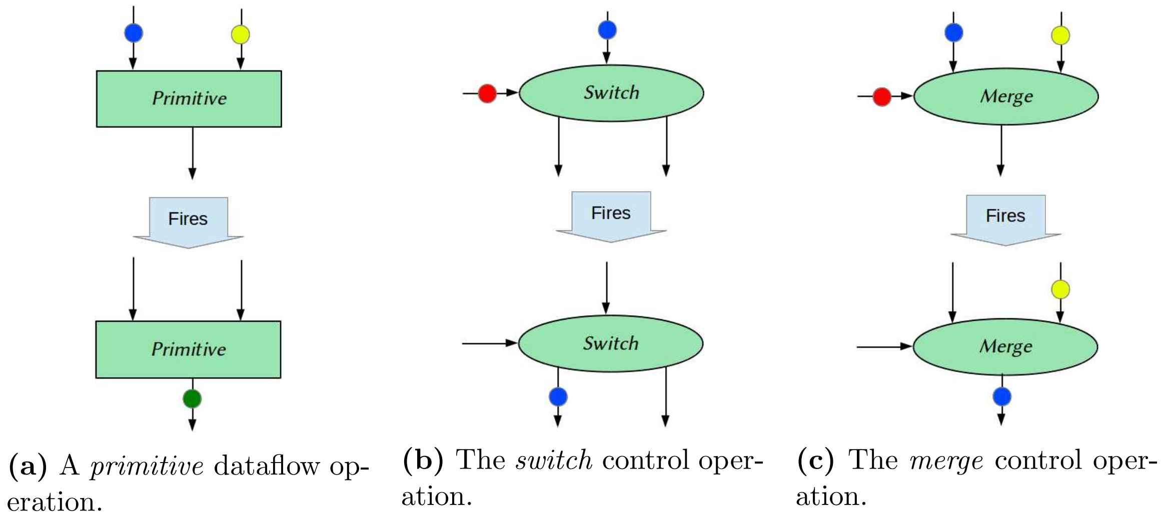
tokens in all $p_i \in P_{in}$[4], while simultaneously removing tokens from all the corresponding acknowledgement places p'_i. The transition corresponding to the dataflow operation from Figure 4.1a will then fire as shown in Figure 4.2b.

4.2.1 Well-behavendess in Dataflow Graphs

Before discussing the operational semantics of the individual operations in the VSFG-S, it is important to consider the issue of *well-behavedness* of dataflow graphs. The original dataflow computational model was composed of various types of operations: primitive operations representing arithmetic/logic functionality as shown in Figure 4.3a, and control operations, that implemented control-directed flow of data, as shown in Figure 4.3b and Figure 4.3c [AC86]. Primitive operations consume all input tokens and produce tokens on all outputs, just like the generalized dataflow operation considered in Figure 4.1a, but the *switch* and *merge* control operations produce or consume tokens selectively, based on a predicate input.

Implementation issues arise when composing dataflow graphs out of these operations, particularly when implementing control dependent dataflow. For instance, Figure 4.4a describes an attempt at speculative execution by using the *merge* operation, while Figure 4.4b describes an attempt at predicated execution using the switch operation. Since *merge* does not consume tokens from its false input edge, two problems arise when the

[4]Recall that $P_{in} \subset P$ is the set of input places of a VSFG graph or subgraph, as defined in Definition 3.1.

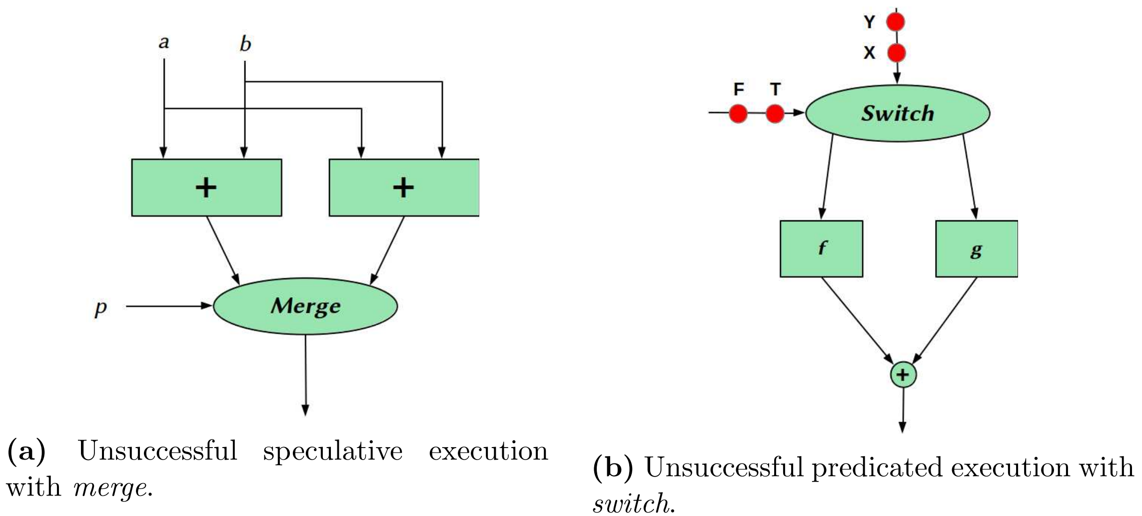


(a) A *primitive* dataflow operation.

(b) The *switch* control operation.

(c) The *merge* control operation.

Figure 4.3: Dataflow operations as defined by Arvind and Culler [AC86].



(a) Unsuccessful speculative execution with *merge*.

(b) Unsuccessful predicated execution with *switch*.

Figure 4.4: Unsuccessful attempts at control-dependent dataflow, taken from [AC86, Tra86, AN90].

schema in Figure 4.4a is reused (i.e. resides inside a loop): (1) if the predicate p holds for many iterations of the loop, the false edge could accumulate an unknown number of tokens, and (2) if the predicate changes value at each iteration, the output token selected would always be one generated during the previous iteration instead of the current one. For the schema in Figure 4.4b, outputs may emerge in the wrong order, for instance, if primitive operation g computes much faster than f, the value $g(Y)$ may emerge at the output before $f(X)$. In order to address these issues, Arvind et al assert that all dataflow graphs be *well-behaved*.

Definition 4.1. Paraphrasing from Gao et al [GGP92]: "A dataflow graph is said to be *well-behaved* if there exists an infinite, fair firing sequence of the operations of the graph that requires only finite memory on every arc". According to Arvind and Culler, all of the following conditions must be met for a graph to be well-behaved [AN90]:

1. It has at least one input *and* one output edge;

2. Assuming that initially there are no tokens in the graph, given exactly one token on every input, ultimately exactly one token is produced on every output;

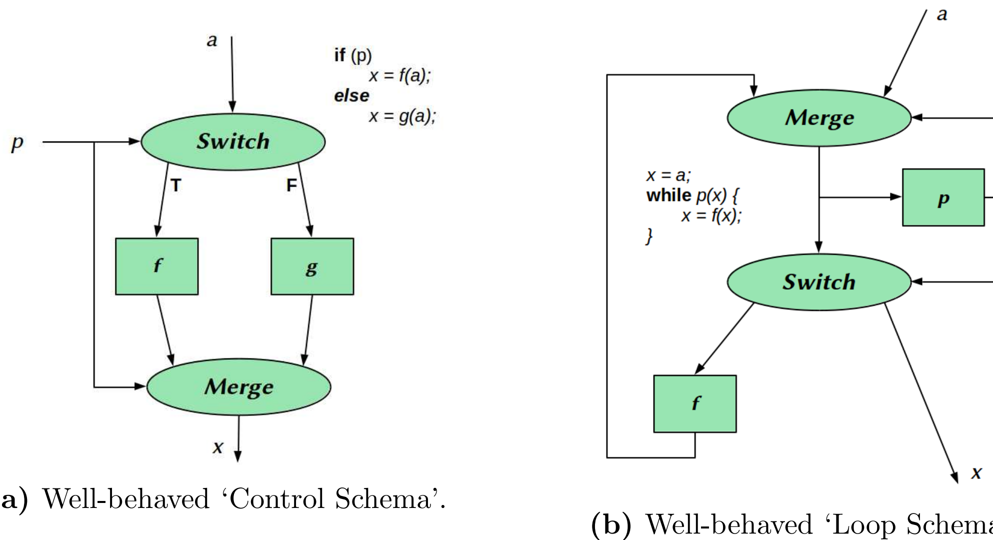
3. when all output tokens have been produced, there are no tokens left in the graph, i.e. the graph is *self-cleaning*.

Per Definition 4.1, the primitive dataflow operation from Figure 4.3a is well behaved, but the *switch* (Figure 4.3b) and *merge* (Figure 4.3c) operations are not. However, well-behaved graphs can still be constructed out of non well-behaved operations: Figures 4.5a and 4.5b show the well-behaved conditional and loop schemas, respectively, developed by Arvind et al [AC86, Tra86, AN90], who also listed the following rules for the construction of well-behaved graphs using these schemas:

1. An acyclic interconnection of graphs is a well-behaved graph if all of its component graphs is a well-behaved graph;

2. The conditional schema is a well-behaved graph if the graphs for the true side and false side are well-behaved graphs;

3. The loop schema is a well-behaved graph if the graphs for the predicate and body are well-behaved graphs.



(a) Well-behaved 'Control Schema'.

(b) Well-behaved 'Loop Schema'.

Figure 4.5: Constructing well-behaved dataflow graphs from non-well-behaved components. Taken from [AC86, Tra86, AN90].

Well-behavedness is therefore an important concern for the VSFG IR, and must be considered when dealing with control-directed dataflow operations such as loops and forward branches. For instance, I employ the well-behaved MUX operator instead of the *merge* in order to implement speculative execution (Section 4.3.1). Similarly, care must be taken when implementing loops as nested-subgraphs (Section 4.3.3).

4.3 Operational Semantics for the *VSFG-S*

Given the context of a homogeneous, 1-bounded, safe, marked-graph Petri-net to represent the structure of the VSFG-S, we can define the semantics of the various transitions. Figure 4.6 lists the various types of transitions that form part of the VSFG-S, as well as the types (Γ) associated with each place, and the various values (V) that each place may hold, constrained by its type.

$$\begin{array}{rcl} \text{Values } V & ::= & n \in \mathbb{N} \mid z \in \mathbb{Z} \mid b \in \mathbb{B} \mid \Delta \mid \tau \mid \bot \\ \text{Types } \Gamma & ::= & \mathit{int} \mid \mathit{float} \mid \mathit{bool} \mid \mathit{token} \end{array}$$

Transitions $t \in T$ ::=

$$\begin{aligned} O(t) &= \mathit{binary}\ (i_1, i_2) \quad \mid \\ O(t) &= \mathit{unary}\ (i_1) \quad \mid \\ O_{data}(t), O_{STATE}(t) &= \mathit{load}\ (i_{addr}, i_{STATE}, i_{pred}) \quad \mid \\ O_{STATE}(t) &= \mathit{store}\ (i_{addr}, i_{data}, i_{STATE}, i_{pred}) \quad \mid \\ O(t) &= \mathit{mux}\ (i_{pred1}, i_{val1}, i_{pred2}, i_{val2}, ..., i_{predN}, i_{valN}) \quad \mid \\ O_{fused}(t) &= \mathit{signal}\ (i_{val}, i_{pred}) \quad \mid \\ O(t) &= \mathit{wait}\ (i_{fused}, i_{pred}) \quad \mid \\ O(t) &= \mathit{eta}\ (i_{pred}, i_{val}) \quad \mid \\ O(t) &= \mathit{mu}\ (i_1, i_2, ..., i_N) \quad \mid \\ O_{fused}(t) &= \mathit{inGate}\ (p_1, v_1, ..., p_N, v_N) \quad \mid \\ O(t) &= \mathit{outGate}\ (p_1, f'_1, ..., p_M, f'_M) \end{aligned}$$

Figure 4.6: List of Values, Types and Transitions supported in the VSFG-S

The following subsections describe all of the transitions mentioned in Figure 4.6 in two formats. First, an equivalent, safe Petri-net representation is provided, and second, a Plotkin-style operational semantics [Plo81] is presented. The latter formalizes the semantics of each operation, while the Petri-net representations are useful in converting the VSFG to custom hardware via the Bluespec SystemVerilog HDL, as described in Chapter 5.

Section 4.3.1 describes the *binary*, *unary*, *load*, *store*, and *mux* operations. These basic operations have been studied extensively in prior literature – the Plotkin-style semantics presented in Equations 4.2 through 4.10 were originally developed by Budiu [Bud03], while their equivalent Petri-net representations were developed for this work.

Sections 4.3.2 and 4.3.3 introduce new operations specifically tailored to deal with the hierarchical nature of the VSFG, namely the *signal*, *wait*, *inGate*, and *outGate* operations. Both their Petri-net representations, as well as their operational semantics (Equations 4.11 through 4.14, and 4.18 through 4.21) were developed specifically for this dissertation.

The notation used to present the operational semantics of the VSFG-S is described as follows:

- The list of values includes Δ to denote don't care values, and a $\perp$ value to indicate an empty, or undefined value.
- In addition to the basic types, Γ also includes a valueless *token* type. Places of token type can only either be empty, signified by the value $\perp$, or indicate the presence of a token with the symbol τ.
- Each place in a VSFG-S can hold a specific *type* of value. I borrow terminology from work on Coloured Petri-nets by Jensen [Jen91] to describe the type of value each place may hold: each place $p \in P$ is described as having a *colour*, given by the function $C : P \to \Gamma$. Hence the expression $C(p) = bool$ indicates that place p holds values of type *bool* (i.e. place p is of colour *bool*).
- The *state* of a Petri-net σ is a function mapping all places in the net to values: $\sigma : P \to V$. Thus we are able to denote the fact that place p holds value v with the expression $\sigma(p) = v$. The expression $\sigma(p) = \perp$ indicates that place p is empty, or undefined (and thus its corresponding acknowledgement place p' currently holds a token). Note that the STATE edge that constrains the ordering of side-effects in the VSFG-S should not be confused with the state σ of the Petri-net.
- A firing of any transition t is defined by a change in the state of the Petri-net from σ to σ'. We write $\sigma' = \sigma[p \mapsto v]$ to indicate that state σ' can be reached from σ by assigning a token with value v to place p (i.e. mapping p to v). By extension, the expression $\sigma' = \sigma[p_1 \mapsto v_1][p_2 \mapsto v_2]$ would indicate that two places must map to new values in order to reach state σ' from σ.
- The expression $\sigma(p) \neq \perp$ can be abbreviated with $def(p)$, signifying that place p has a token. Conversely, $!def(p)$ is the same as $\sigma(p) = \perp$, to signify that place p has no token. The expression can be extended to describe the status of multiple places as follows: $def(p_1, p_2)$.
- The clearing of a place, given by the expression $[p \mapsto \perp]$, can be abbreviated as $erase(p)$. The expression can be extended to describe the clearing of multiple places as follows: $erase(p_1, p_2)$.
- Each Transition $t \in T$ in the VSFG has a set of input places $I(t)$, and a set of output places $O(t)$. Each transition acts upon values from all of its input places, and updates all of its output places. $O(t)$ may contain multiple output places if there are multiple consumers for an output value (i.e. $fanout > 1$), or if a transition has multiple distinct output types. An example of the latter case is the $load(i_{addr}, i_{STATE}, i_{pred})$ transition, which returns both a data value from memory, as well as producing an output state token. Here, $O_{data}(load)$ represents the set of fanout places of the memory value being retrieved, while $O_{STATE}(load)$ represents the set of fanout places for the state-token produced by the *load* transition: $O(load) = O_{data}(load) \cup O_{STATE}(load)$.
- If $addr \in V$ is used to indicate a value representing a memory address, we write $update(addr, v)$ to indicate that value $v \in V$ is being stored at memory address $addr$. Similarly, $lookup(addr)$ is used to denote a value retrieved from the location specified by $addr$.

- The semantics for each transition $op \in T$ are specified by (1) a *precondition* indicating when a transition may be triggered, as well as (2) the resultant *change* affected on the Petri-net state σ when the precondition holds and the transition fires. This is denoted as:

$$op \frac{precondition}{change}$$

4.3.1 Semantics for Basic Operations

Value Operations

As introduced by Definition 3.1, Value operations represent the basic, non side-effecting unary and binary ALU operations, and are analogous to the well-behaved primitive dataflow operation from Figure 4.3a. The list of such operations currently supported is largely similar to the binary and unary aritmetic, logic and conversion operations listed in the LLVM Language Reference Manual [LLV], since the HLS toolchain implemented for this dissertation uses the LLVM IR as the input language. Further details about this toolchain are provided in Chapter 5.

Figures 4.7a and 4.7c show the dataflow graph representation of such operations, while Figures 4.7b and 4.7d show their 1-bounded, safe Petri-net equivalents. As these operations have are not side-effect sensitive, the do not require a state-edge input, nor a predicate input (i.e. they are *predicate promoted*). The operational semantics of these types of operations, as described by Budiu [Bud03], are described by Equations 4.2 and 4.3.

$$o_1 = unary\ (i_1)\ \frac{\sigma(i_1) \neq \perp, \sigma(o_1) = \perp}{\sigma' = \sigma[o_1 \mapsto unary(\sigma(i_1))][i_1 \mapsto \perp]} \tag{4.2}$$

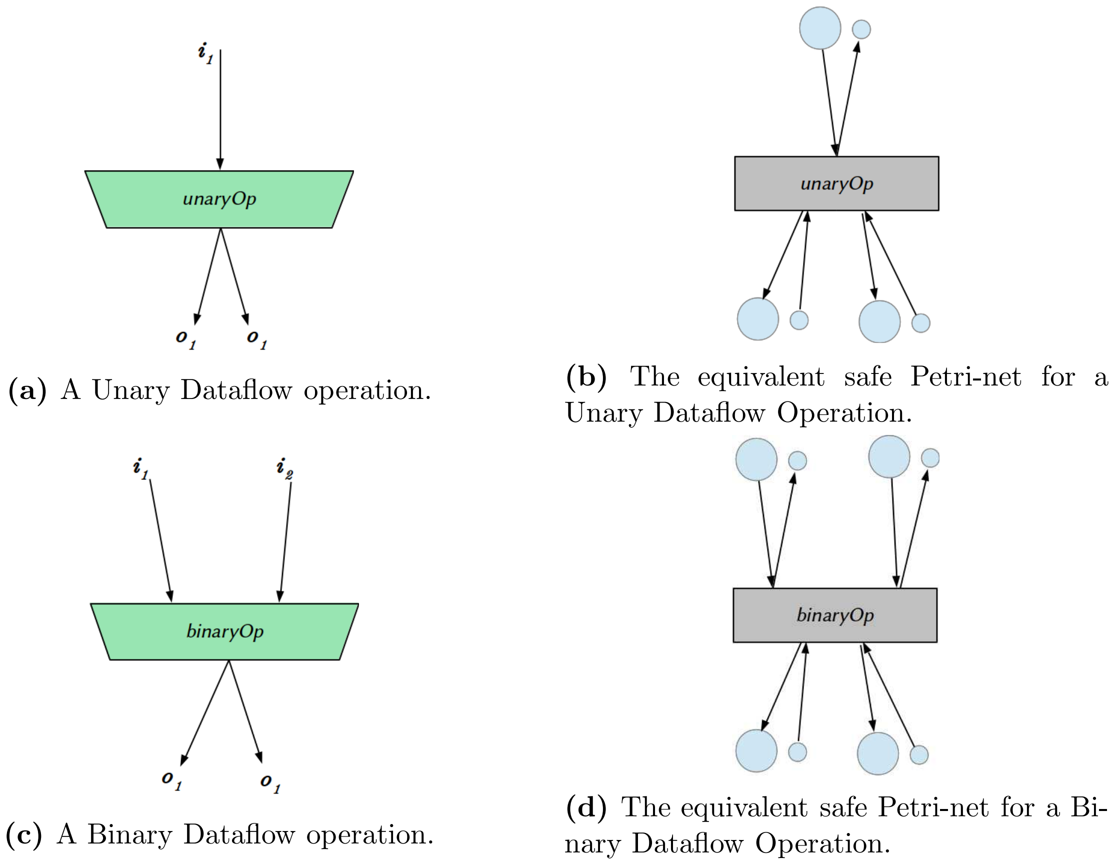
$$o_1 = binary\ (i_1, i_2)\ \frac{\sigma(i_1) \neq \perp, \sigma(i_2) \neq \perp, \sigma(o_1) = \perp}{\sigma' = \sigma[o_1 \mapsto binary(\sigma(i_1), \sigma(i_2))][i_1 \mapsto \perp][i_2 \mapsto \perp]} \tag{4.3}$$

For simplicity, I assume that each operation whose semantics are described in this way has a fanout of 1, meaning only one output place, $o_1 \in O(t)$, needs to be updated. As an example, consider Equation 4.3. The precondition for this transition indicates that it will fire only when the input places i_1 and i_2 are defined (not empty), *and* the output place o_1 is empty. Upon firing, o_1 is updated with the computed result of the *binary* operation performed on the two input values ($[o_1 \mapsto binary(\sigma(i_1), \sigma(i_2))]$), while simultaneously, the input places are *acknowledged*, or *cleared* ($[i_1 \mapsto \perp][i_2 \mapsto \perp]$). This clearing action could also have been written using the defined abbreviation: $erase(i_1, i_2)$.

Load and Store Operations

The dataflow graph component for the *load* and *store* operations are shown in Figures 4.8a and 4.8b respectively. As mentioned in Section 3.6, as State operations, both *load* and *store* have a state and a predicate input (i_S and i_p, respectively), as well as a state output (o_S), in addition to their usual address and data inputs and outputs (i_a, i_d and o_d). Both operations are well-behaved, since even *store* has an output state-edge.

The equivalent safe Petri-net transitions for these operations are similar to those for the *binary* and *unary* operations, and are shown in Figures 4.8c and 4.8d. As described by Budiu [Bud03], and adapted for use with our Petri-net framework, the operational



(a) A Unary Dataflow operation.

(b) The equivalent safe Petri-net for a Unary Dataflow Operation.

(c) A Binary Dataflow operation.

(d) The equivalent safe Petri-net for a Binary Dataflow Operation.

Figure 4.7: Basic Value Operations in the VSFG-S: Unary and Binary.

semantics for the *load* operation are given by Equations 4.4 and 4.5, while the semantics for the *store* operation are given by Equations 4.6 and 4.7. Again, for simplicity, I assume that the output set for each kind of output ($O_{data}(t), O_{STATE}(t) \subseteq O(t)$) for each of the transitions only contains one place (o_d and o_S, respectively).

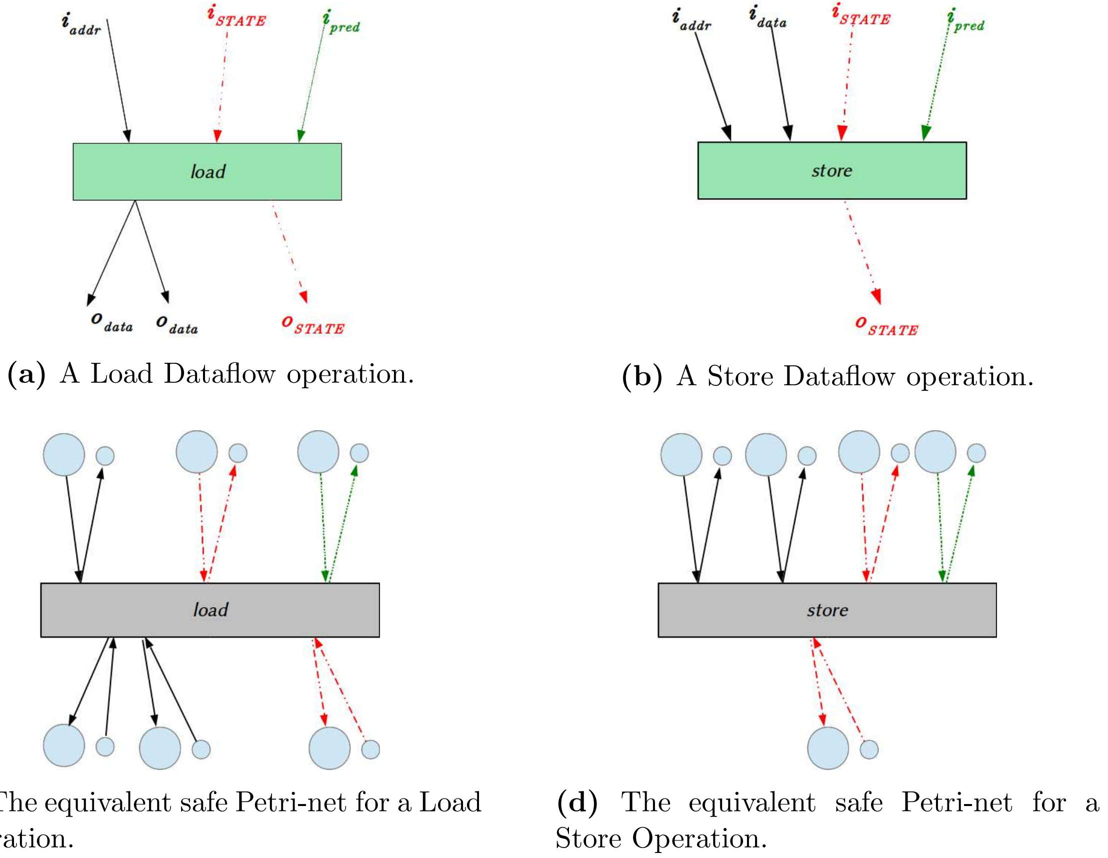
$$o_d, o_S = load\ (i_a, i_S, i_p)\ \frac{\sigma(i_p) = \mathbf{True}, def(i_a, i_{STATE}), !def(o_d, o_S)}{\sigma' = \sigma[o_d \mapsto lookup(\sigma(i_a))][o_S \mapsto \tau] \circ erase(i_a, i_S, i_p)} \tag{4.4}$$

$$o_d, o_S = load\ (i_a, i_S, i_p)\ \frac{\sigma(i_p) = \mathbf{False}, def(i_a, i_S), !def(o_d, o_S)}{\sigma' = \sigma[o_d \mapsto \Delta][o_S \mapsto \tau] \circ erase(i_a, i_S, i_p)} \tag{4.5}$$

$$o_S = store\ (i_a, i_d, i_S, i_p)\ \frac{\sigma(i_p) = \mathbf{True}, def(i_a, i_d, i_S), !def(o_S)}{\sigma' = \sigma[o_S \mapsto \tau] \circ update(\sigma(i_a), \sigma(i_d)), erase(i_a, i_d, i_S, i_p)} \tag{4.6}$$

$$o_S = store\ (i_a, i_d, i_S, i_p)\ \frac{\sigma(i_p) = \mathbf{False}, def(i_a, i_d, i_S), !def(o_S)}{\sigma' = \sigma[o_S \mapsto \tau] \circ erase(i_a, i_d, i_S, i_p)} \tag{4.7}$$

If the input predicate (value in place i_p) holds, the *load* operation accesses the memory location specified by the value in the i_a input place, through the $lookup(addr)$ function, and places the result atomically in its output place o_d, along with a state token τ in its state output o_S (Equation 4.4), while simultaneously clearing its inputs (i.e. removing tokens from the input places, and placing tokens in their corresponding acknowledgement places).



(a) A Load Dataflow operation.

(b) A Store Dataflow operation.

(c) The equivalent safe Petri-net for a Load Operation.

(d) The equivalent safe Petri-net for a Store Operation.

Figure 4.8: Basic State operations in the VSFG-S: Load and Store.

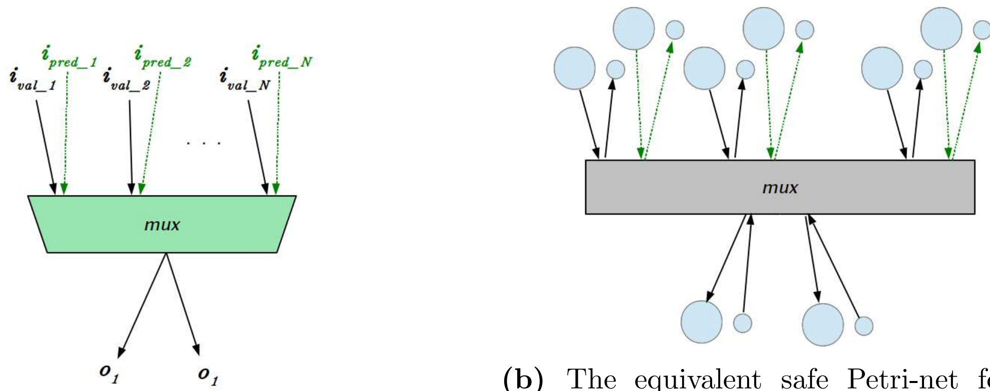
Similarly, if the predicate holds for a *store* operation, the $update(addr, data)$ function is utilized to place the value in i_{data} into the memory location specified by the value in i_a (Equation 4.6). Note that as the memory address space is not incorporated as part of the VSFG-S semantics, the $update(addr, data)$ function does not update any places in the Petri-net state σ.

If the input predicate is false, then neither operation accesses memory, as described by the rules given in Equations 4.5 and 4.7. However, due to the need to enforce well-behavedness in the graph, both operations must always place output values into their output places: the *load* operation places a don't care value Δ in o_d, while both *load* and *store* place a state token τ immediately in their o_S output places, while simultaneously clearing all inputs. Well-behavedness was discussed in greater detail in Section 4.2.1.

The MUX Operation

The MUX operation defined as part of the VSDG (and the VSFG) in Definition 3.1 selects from one of two input values based on a third, input predicate value. However, my implementation of the MUX is slightly different: I implement a *decoded* multiplexer that selects one from N input values, based on another N input predicate values. The dataflow graph representation of this decoded MUX operation is shown in Figure 4.9a.

For such a decoded multiplexer, it must be guaranteed that at most only one of the N input predicates will hold, while the rest must be false. Recall from Section 3.6 that during the conversion from a CFG to the VSFG, predicate expressions are generated for each



(a) The MUX operation as Dataflow.

(b) The equivalent safe Petri-net for a *strict* implementation of the MUX Operation.

Figure 4.9: The MUX operation in the VSFG-S, and its *strict* Petri-net implementation.

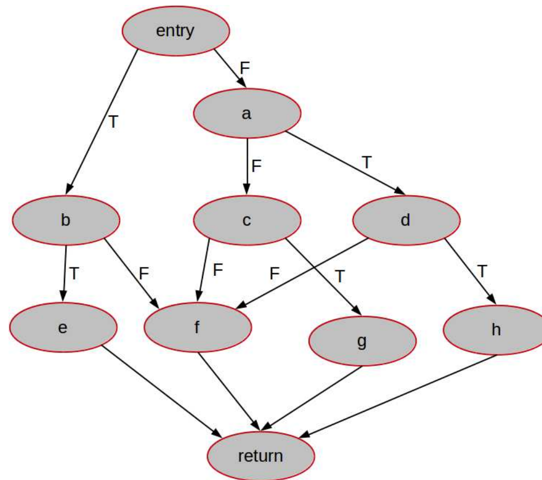


Figure 4.10: Acyclic CFG for a non-loop function. Copy of Figure 3.14a.

basic-block in the CFG. MUX operations replace the ϕ-nodes in SSA form, that occur at 'value-join points' in the code: i.e. at basic-blocks that have multiple predecessors. For instance, consider the basic block f from Figure 4.10, as it has multiple potential predecessor basic blocks, and assume that it contains a ϕ-node selecting from one of three potential values (since there are three predecessors to f, namely b, c, and d):

$$v_{out} = \phi\Big((v_b, b), (v_c, c), (v_d, d)\Big)$$

Each of the tuples of the form (v_x, x) specify the value v_x to be selected if control-flow reaches the ϕ-node from predecessor block x. Whereas the ϕ-node relies on the flow of control from one of multiple predecessors to the value-join basic block in order to select the appropriate output value, the MUX node instead relies on the computed predicate expressions generated for each of the the predecessor blocks and control-flow edges (as described in Section 3.6):

$$v_{out} = mux(v_b, p_b, v_c, p_c, v_d, p_d)$$

where:

$$p_b = p(b) \,.\, \overline{p(b,e)}, \qquad p_c = p(c) \,.\, \overline{p(c,g)}, \qquad p_d = p(d) \,.\, \overline{p(d,h)}$$

For instance, the MUX would select the value v_b, if the predicate $p(b)$ for block b holds, *and* the control-flow condition evaluation $p(b, e)$ at the end of block b evaluates as false, in which case, after reaching b, control would flow from b to f. For our decoded MUX operation, the constraint that *at most* only one predicate input be true is guaranteed by the fact that in the original CFG, control would flow to f from only one of its predecessors, i.e. the predicates p_b, p_c, and p_d are mutually exclusive.

Note that the predicate $p(f)$ of the host block f for the MUX operation is not used, as MUX is a value operation, and can be predicate-promoted. Note also, that this implies that in a VSFG supporting aggressive speculative execution, it is possible for all of the predicate inputs to a MUX to be false. This would happen for instance if control-flow never passes to f, instead going through e, g, or h. In this case, the MUX must still produce a don't care value in order to meet well-behavedness requirements (Section 4.2.1). Another example MUX operation in the *return* block would have simpler predicate expressions, since all the branches to *return* from its predecessors are unconditional:

$$v_{out} = mux(v_e, p_e, v_f, p_f, v_g, p_g, v_h, p_h)$$

where:

$$p_e = p(e), \qquad p_f = p(f), \qquad p_g = p(g), \qquad p_h = p(h),$$

The MUX operation semantics may be defined in a similar fashion to the *binary* and *unary* operations, where the MUX operation waits for all inputs to become available before selecting the one whose predicate holds and returning that as its output, while simultaneously clearing all of its inputs, as shown in Figure 4.2b. Note however, that this introduces unnecessary synchronization into the dataflow execution: a value with a true predicate would have to wait for all other values and their predicates to arrive at MUX before it can be forwarded across MUX to its dataflow consumers. The Petri-net representation of this *strict* implementation of the MUX operation is shown in Figure 4.9b.

Unlike the operations described thus far, MUX does not need all of its inputs to be available before it can generate an output. A *non-strict*, or *lenient* implementation is possible, where as soon as a value with a corresponding *true* predicate is available, the MUX operation updates its output places. Again, to preserve well-behavedness, the acknowledgement of inputs would only occur once all inputs have arrived. The use of lenient MUX operations has been shown to improve performance in prior work on generating static-dataflow custom hardware [BAG05]. The operational semantics for the lenient MUX operation as described by Budiu [Bud03] are given by Equations 4.8, 4.9, and 4.10.

$$o = mux\ (v_1, p_1, ..., v_N, p_N)\ \frac{!def(o), !def(\mathbf{sent}), \exists k.(def(v_k) \wedge \sigma(p_k) = \mathbf{True})}{\sigma' = \sigma[o \mapsto \sigma(v_k)][\mathbf{sent} \mapsto \mathbf{True}]} \tag{4.8}$$

$$o = mux\ (v_1, p_1, ..., v_N, p_N)\ \frac{\forall k.\sigma(p_k) = \mathbf{False},}{\sigma' = \sigma[o \mapsto \Delta][\mathbf{sent} \mapsto \mathbf{True}]} \tag{4.9}$$

$$o = mux\ (v_1, p_1, ..., v_N, p_N)\ \frac{\forall k.def(v_k, p_k), \sigma(\mathbf{sent}) = \mathbf{True},}{\sigma' = \sigma[\mathbf{sent} \mapsto \perp] \circ \forall k.erase(v_k, p_k)} \tag{4.10}$$

In addition to the input and output places, an internal boolean value **sent** is used to track whether a value has been produced at the outputs. The rule in Equation 4.8 handles the case when one of the input predicate values (p_k) holds – the **sent** variable is

set **True**, and the corresponding input value (v_k) is forwarded to the output o. The rule in Equation 4.9 handles the case when all input predicates are false – in which case, a don't care value is forwarded to the output, and **sent** is again set to **True**. The rule in Equation 4.10 is used to acknowledge all inputs once they're ready, and a value has been sent (i.e. **sent** $\mapsto$ **True**) by either of the preceding rules.

4.3.2 Compound Operations: Nested Acyclic Subgraphs

The VSFG as defined in Definition 3.1 is a hierarchical graph, with compound operations representing the nested subgraphs of called functions and tail-recursive loops. In order to present the operational semantics for such compound operations, I again borrow from work done on Hierarchical Coloured Petri-nets [Jen91]:

- **Substitution Transitions:** Within a Hierarchical Petri-net, a compound operation can be represented using a *substitution transition*, which is a transition that, along with its surrounding edges, may be replaced by a more complex Petri-net. The replacing Petri-net is referred to as a subpage.

- **Fused Places:** Hierarchical Petri-nets introduce the notion of *fusion* of places. Specifying that a set of places in a Petri-net are *fused*, means that they all represent a single place, even though they are drawn as individual places.

In our VSFG-S Petri-net, compound operations are therefore equivalent to substitution transitions, with each such transition representing the nested subgraph of the compound operation. Fused places are used to connect the set of input and output places (P_{in} and P_{out}, respectively) of the nested subgraph with the function call argument and result places in the parent graph.

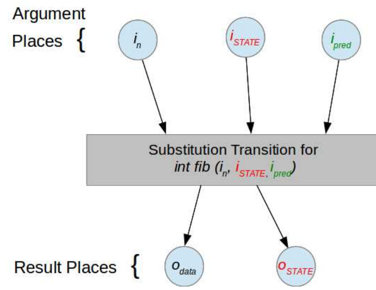
As an example, let us assume that the fibonacci function from Figure 3.10a (VSFG shown in Figure 3.13) is the nested subgraph being called from a parent graph. Figure 4.11a shows the substitution transition, while Figure 4.11b shows the subpage being substituted[5]. The fused places between both figures are identified with the same label: the i_n, i_{STATE}, and i_{pred} argument places from Figure 4.11a are each fused with their corresponding, similarly named input places ($\in P_{in}$) in the subpage VSFG from Figure 4.11b. Similar fusion of places applies between the result places from Figure 4.11a and the output places ($\in P_{out}$) from Figure 4.11b.

From the perspective of the parent graph, a substitution transition appears like any other transition, and thus may operate in the same fashion: atomically consuming all inputs and producing outputs as described in Figure 4.2b. However, for the nested subgraph being activated, this approach would imply two constraints on performance:

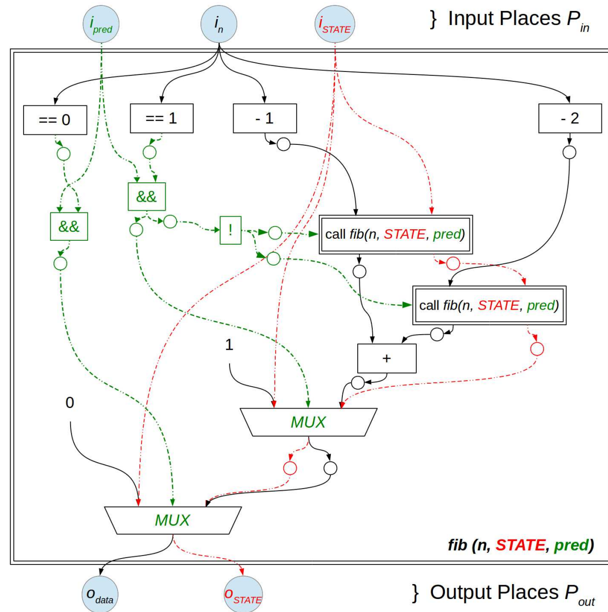
- **Call Strictness:** All input arguments must be available before computation within the subgraph starts.

- **No Loop Parallelism:** Given the static-dataflow execution semantics, new inputs may not be injected into the nested subgraph until all of the outputs have been produced from the previous set of inputs. Thus if the subgraph is nested within a loop/tail-recursive function, it can only execute operations from one iteration at a time, restricting concurrency.

[5]For simplicity, none of the acknowledgement places have been shown.



(a) Substitution transition for the $fib(i_n, i_{STATE}, i_{pred})$ nested VSFG



(b) Petri-net subpage showing the $fib(i_n, i_{STATE}, i_{pred})$ VSFG.

Figure 4.11: The Substitution transition for the nested VSFG subgraph $fib(i_n, i_{STATE}, i_{pred})$ in its parent graph, and its replacement Petri-net subpage.

Call strictness was identified as one of the issues limiting achievable performance for static-dataflow custom hardware by Budiu et al [BAG05]. To address this, *leniency* may be introduced in a similar fashion to the MUX operation, without violating well-behavedness, by allowing lenient insertion of arguments into the subgraph, but delaying acknowledgement of the input arguments until all of the corresponding outputs have been retrieved from the subgraph. This approach is discussed in Section 4.3.3, where the nested subgraphs implement loops, and thus are not guaranteed to be well-behaved. However, so long as the nested subgraph is known to be well-behaved, loop parallelism may be exploited by allowing repeated insertion of input arguments into the subgraph, irrespective of whether a result has been obtained from the previous set of arguments. This is possible since flattening a well-behaved nested subgraph into its well-behaved parent graph produces a well-behaved combined graph[6].

As mentioned in Section 3.6.2, a means of controlling the degree of speculative execution of nested subgraphs is needed in an eager-evaluated VSFG. To implement such

[6]Recall from Section 4.2.1: "An acyclic interconnection of graphs is a well-behaved graph if all of its component graphs is a well-behaved graph" [Tra86, AN90].

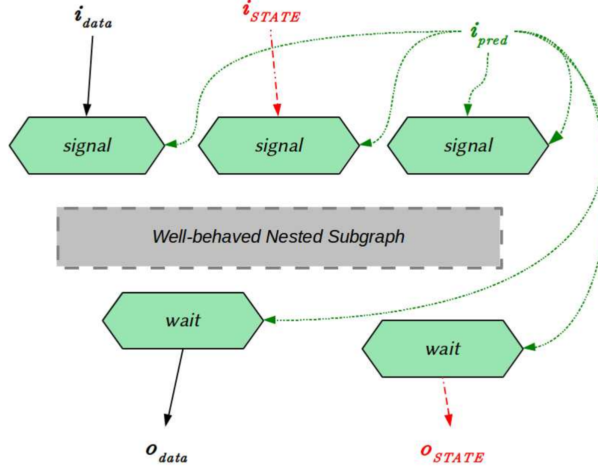


Figure 4.12: The *signal* and *wait* operations for communicating with well-behaved nested subgraphs.

unrestricted communication with well-behaved nested subgraphs, I introduce two new dataflow operations: *signal* and *wait*. The use of *signal* and *wait* in a parent graph to communicate with a well-behaved nested subgraph is shown in Figure 4.12. The predicate inputs to the *signal* operations are used to enforce predicated execution of subgraphs. If speculative execution is desired for a subgraph, the predicate input may be set *true*, or removed altogether.

The *signal* operation takes a value input that is to be communicated to the subgraph, as well as a predicate input, but produces no output in the parent graph. If the predicate input holds, only then is the value inserted into the subgraph. The *wait* operation only takes a predicate input from the parent graph, and produces a value output that has either been fetched from the subgraph (if the predicate holds), or is a don't care value (if the predicate is false). The necessary values inserted into the nested subgraph are the input *state*-edge token, as well as an input predicate to the subgraph (i.e. the *inPred* value for the subgraph, mentioned in Section 3.6), while the necessary value returned from the subgraph is the output *state*-edge token[7].

The equivalent Petri-net representation for the dataflow operations in Figure 4.12 is shown in Figure 4.13. Each *signal* and *wait* operation is represented using two transitions: one that fires only when the input predicate holds ($signal_T$, $wait_T$), and one that fires otherwise ($signal_F$, $wait_F$)[8]. The $signal_T$ transition has a set of output places $O_{fused}(t)$ that are fused with some of the input places P_{in} of the nested VSFG subgraph (i.e. $O_{fused}(t) \subseteq P_{in}$), while similarly, each $wait_T$ transition has an input place i_{fused} that is fused with one of the output places of the nested VSFG subgraph (i.e. $i_{fused} \in P_{out}$). Equations 4.11 and 4.12 give the operational semantics of the *signal* transitions, while Equations 4.13 and 4.14 give the semantics for the *wait* transitions.

A $signal_T$ transition inserts values into its output place fused with the nested subgraph ($o_{fused} \in P_{in}$ of the subgraph) only if the predicate value $\sigma(i_p)$ is true (Equation 4.11),

[7]A possible optimization would be to remove the state-edge and predicate inputs in the case that there are no state-operations in the nested subgraph

[8]Note that with the inclusion of the *signal* and *wait* transition pairs, the VSFG-S would no longer be a marked graph. However this does not introduce non-determinism into the execution of the Petri-net, as the firing of the appropriate transition is determined by the value within the predicate place.

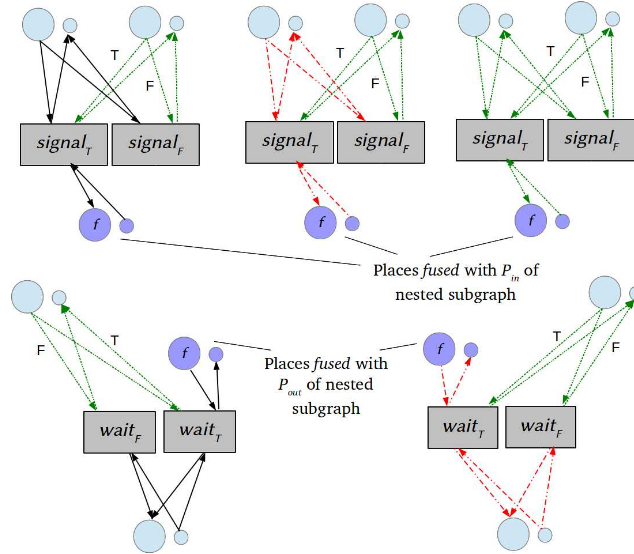


Figure 4.13: The equivalent *signal* and *wait* Petri-net transitions, showing the fused places used to communicate with a nested subgraph.

otherwise $signal_F$ fires instead, simply clearing its inputs, and preventing the activation of any transitions in the nested subgraph (Equation 4.12). Similarly, the $wait_T$ transition only seeks to retrieve values from its input place fused with the nested subgraph ($i_{fused} \in P_{out}$ of the subgraph) if its input predicate holds (Equation 4.13). If the predicate is false, *wait* assumes that the corresponding *signal* operation would not have inserted tokens into the subgraph, and thus it expects no input from the subgraph. In this case, $wait_F$ fires, clearing its predicate input, and producing a don't care value at its output places in the parent graph (Equation 4.14).

$$o_{fused} = signal\ (i_v, i_p)\ \frac{\sigma(i_p) = \textbf{True}, def(i_v), !def(o_{fused})}{\sigma' = \sigma[o_{fused} \mapsto \sigma(i_v)] \circ erase(i_v, i_p)} \tag{4.11}$$

$$o_{fused} = signal\ (i_v, i_p)\ \frac{\sigma(i_p) = \textbf{False}, def(i_v)}{\sigma' = \sigma \circ erase(i_v, i_p)} \tag{4.12}$$

$$o_v = wait\ (i_{fused}, i_p)\ \frac{\sigma(i_p) = \textbf{True}, def(i_{fused}), !def(o_v)}{\sigma' = \sigma[o_v \mapsto \sigma(i_{fused})] \circ erase(i_{fused}, i_p)} \tag{4.13}$$

$$o_v = wait\ (i_{fused}, i_p)\ \frac{\sigma(i_p) = \textbf{False}, !def(o_v)}{\sigma' = \sigma[o_v \mapsto \Delta] \circ erase(i_p)} \tag{4.14}$$

4.3.3 Compound Operations: Nested Loop Subgraphs

Loops in the VSFG are represented using tail-recursion as infinite, nested, acyclic subgraphs. However, for implementation as static-dataflow hardware using finite resources, cycles must be re-introduced into the VSFG, by incorporating back edges that allow loop variant values to be reinserted into the loop-body for each iteration of a loop. The VSFG-S replaces the VSFG's compound transitions representing the tail-recursive calls with back-edges.

There are two aspects to accomplishing this safely: (1) how to modify the structure of a loop-subgraph in order to introduce cycles in the static-dataflow, and (b) how to maintain well-behavedness in both the loop subgraph and the parent graph. During both of these stages, it is important to ensure that the inherent advantage of the VSFG in dealing with loops – i.e. facilitating outer-loop parallelism through independent unrolling of loops in a loop nest (Section 3.6.2) – are retained.

Reintroducing back-edges in the loop subgraph

Consider a loop VSFG, such as the one defined by the tail recursive function from Figure 3.12: $forloop(i, a, STATE, pred)$. Loops in the VSFG representation may be considered to have three distinct parts, as shown in Figure 4.14:

1. **The loop-body:** A well-behaved acyclic graph representing the loop body, and excluding both the tail-recursive subgraph and the exit MUX operation. The loop-body, like any other VSFG, has input places and output places. However, we distinguish between two different types of output places, as marked in Figure 4.14:

   (a) The loop ***exit*** outputs ($P_{exit} \subset P_{out}$): these are the value, state and predicate outputs from the loop-body that would be used if the exit predicate holds during the current iteration.

   (b) The loop ***iteration*** outputs ($P_{iteration} \subset P_{out}$): these are the value state and predicate outputs that serve as input arguments to the tail-recursive call – essentially the input values for the next iteration of the loop. Thus this set of outputs exactly matches the set of input places of the loop-body

   For each loop-body, $P_{out} = P_{iteration} \cup P_{exit}$, and $P_{iteration} \cap P_{exit} = \emptyset$. In addition, the predicates for the loop-exit and iteration outputs will be mutually exclusive, since a loop will always either exit or iterate, but not both.

2. **The tail-recursive call** that occurs if the exit predicate p generated by the loop-body is false.

3. **The exit MUX operation** that selects the appropriate set of loop-exit outputs to return to the loop's parent graph.

Our objective is to replace the tail-recursive call with control-directed dataflow that allows the tail-call function arguments to be reinserted back into the loop-body. To this end, I utilize two new operations, *eta* and *mu*, that were defined by Budiu in his implementation of static-dataflow hardware [Bud03]:

- **A Conditional Gate Operation, *Eta***: The *eta* operation receives a predicate input and a value input. If the predicate input holds, the value input is forwarded to the *eta* output, and both the value and predicate inputs are cleared/acknowledged. Otherwise, if the predicate input is false, the inputs are acknowledged, but no output is produced. Figure 4.15a shows the dataflow representation of the Gate operation, while Figure 4.15b shows the equivalent Petri-net representation.

  The operational semantics for *eta*, as described by Budiu [Bud03], are given in Equations 4.15 and 4.16. Note that *eta* is not a well-behaved operation, as it is not

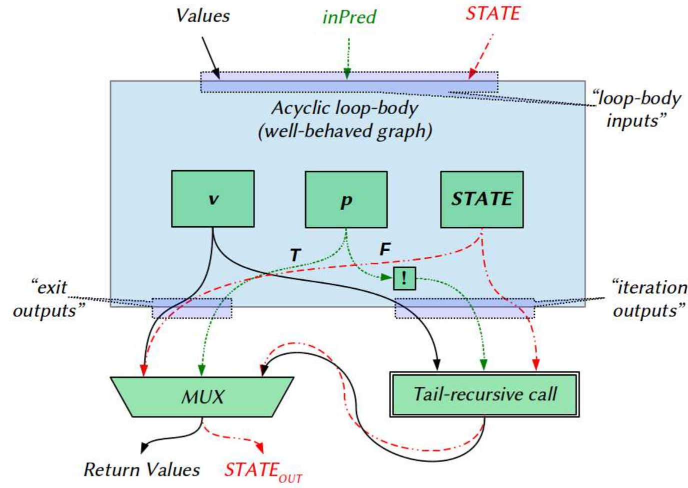
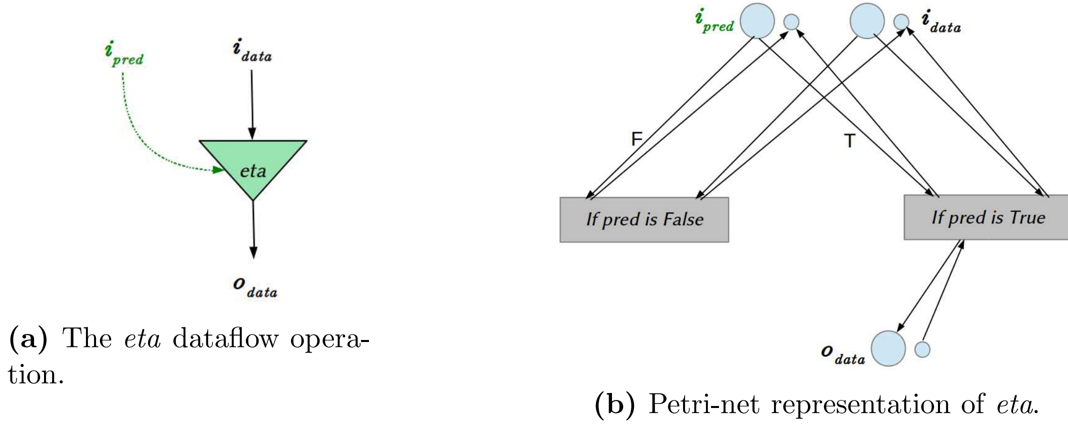


Figure 4.14: The three components of a loop represented using a VSFG: (1) a well-behaved, acyclic loop body, (2) the tail recursive call, and (3) the exit MUX.



(a) The *eta* dataflow operation.

(b) Petri-net representation of *eta*.

Figure 4.15: The *eta* operation, for implementing control-directed dataflow in the VSFG-S.

guaranteed to produce an output each time it consumes its inputs.

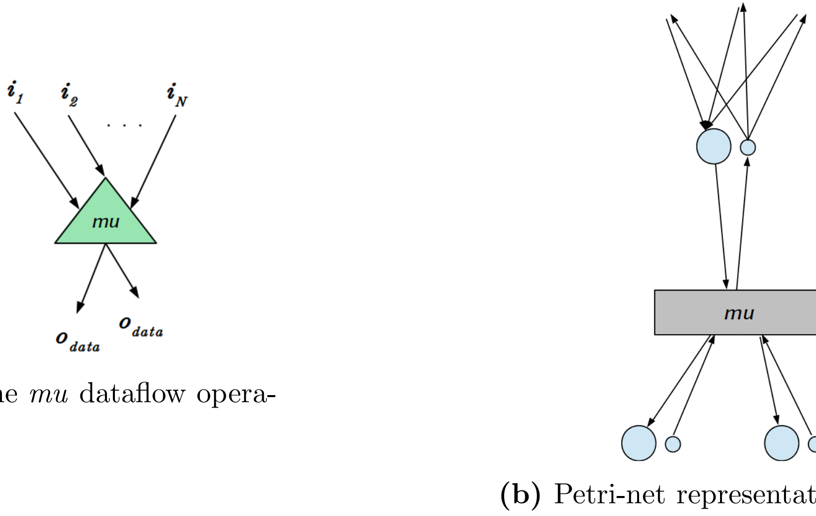
$$o = eta\ (i_v, i_p)\ \frac{\sigma(i_p) = \textbf{True}, def(i_v), !def(o)}{\sigma' = \sigma[o \mapsto \sigma(i_v)] \circ erase(i_v, i_p)} \tag{4.15}$$

$$o = eta\ (i_v, i_p)\ \frac{\sigma(i_p) = \textbf{False}, def(i_v)}{\sigma' = \sigma \circ erase(i_v, i_p)} \tag{4.16}$$

- **A Non-deterministic merge operation, *Mu***: The *mu* operation is essentially a non-deterministic merge – whereas for the *merge* operation discussed in Section 4.2.1, a predicate input was used to control which of its inputs a token is selected, there is no predicate input for the *mu* operation: an input token is selected and placed at the outputs immediately upon arrival. Figure 4.16a shows the dataflow representation of *mu*, and Figure 4.16b shows the equivalent Petri-net.

  The Petri-net representation of *mu* essentially has an input place with multiple incident edges[9]. When the *mu* transition fires, it simply copies a value from its

[9] As with the *signal* and *wait* operations, note that with the inclusion of either an *eta* or a *mu*



(a) The *mu* dataflow operation.

(b) Petri-net representation of *mu*.

Figure 4.16: The *mu* operation, for implementing non-deterministic merge in the VSFG-S.

input place to each of its output places. The operational semantics for *mu*, adapted from Budiu [Bud03], are given in Equation 4.17. Note that *mu* is also not well-behaved, as it produces an output by consuming inputs from only one of its input edges.

$$o = mu\ (i_1, i_2, ..., i_N)\ \frac{\exists k.def(i_k), !def(o)}{\sigma' = \sigma[o \mapsto \sigma(i_k)] \circ erase(i_k)} \tag{4.17}$$

The *eta* and *mu* operations are utilized to replace the tail-recursive call in the VSFG with loop-back edges in the VSFG-S, as shown in Figure 4.17. The well-behaved, acyclic loop-body remains unchanged from the VSFG. However, both the tail-recursive call and the exit MUX are replaced with *eta* operations – one *eta* for the "exit outputs" and another for the "iteration outputs" of the loop-body. The predicates driving both these *eta* operations are complements of each other; thus together both the *eta* operations implement similar functionality to the *switch* operation from Figure 4.3b (described in Section 4.2.1): The 'exit-outputs' *eta* will only produce an output when the loop-exit predicate holds, otherwise it will produce nothing; conversely, the 'iteration-outputs' *eta* will drive the loop-back edges only when the complement of the loop-exit predicate holds.

There will be a loop-back edge for each of the input places of the loop-body VSFG, including the *inPred* predicate input. If the loop-exit predicate is false, the 'iteration-outputs' *eta* will place the next-iteration input values (including $inPred =$ **True**) on the loop-back edges. A *mu* operation is used to forward input values entering the loop from either the loop's parent graph (marked 'parent-graph inputs' in Figure 4.17), or from the loop-back edges. Given the non-deterministic nature of *mu*, it is essential to impose a constraint within the parent graph that prevents repeated insertion of values into the loop subgraph until the loop has terminated. This will be discussed in the following subsection.

In order to support loop unrolling, the tail-recursive VSFG may have its recursive subgraph *flattened* into the loop body any number of times, before loop-back edges need

operation, the VSFG-S will no longer be a *Marked Graph*. The *eta* Petri-net still exhibits deterministic execution based on the input predicate value, while the *mu* can exhibit non-deterministic behavior. Non-determinism in dataflow execution due to the *mu* operator is prevented through the use of the *inGate* and *outGate* operations described shortly.

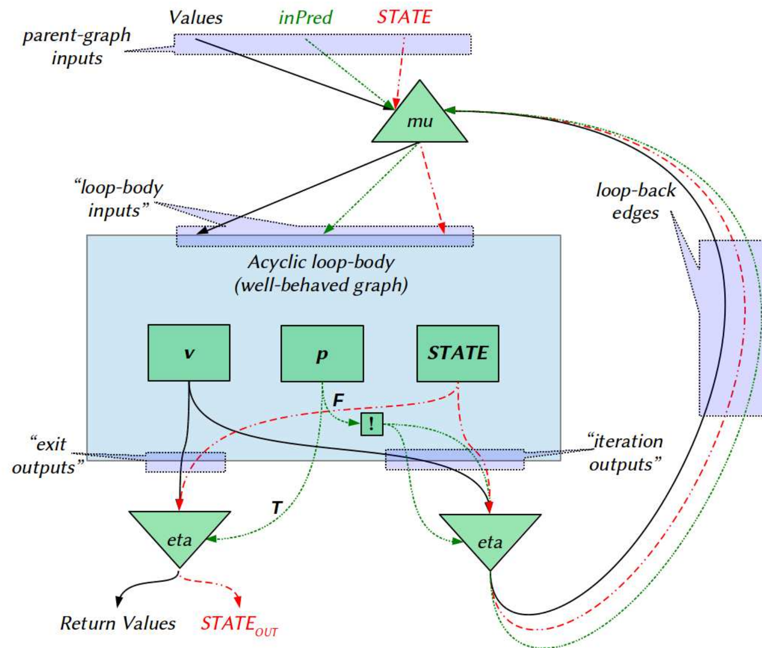


Figure 4.17: Reintroducing back-edges into the VSFG-S by removing the MUX and Tail-call subgraph with *eta* and *mu* operations.

to be introduced. Figure 4.18 shows the VSFG from Figure 4.14 with the tail-recursive loop call flattened twice into the main loop body. Once the desired degree of unrolling has been achieved, loop-back edges may be introduced in a similar fashion to Figure 4.17.

Figure 4.19 shows the thrice unrolled version of the loop from Figure 4.17. As before, the 'exit-outputs' of each copy of the loop-body will have an *eta* controlled by its respective exit predicate. The 'iteration-outputs' of the last copy of the loop body will similary have an *eta* controlled by the complement of its exit predicate, with the output values of the eta connected to the input *mu* operation. As there are multiple 'exit-output' *eta* operations in the flattened/unrolled VSFG-S, another *mu* operation is needed at the outputs to the loop in order to non-deterministically select the correct loop-exit outputs from one of the exit *etas*.

The three 'exit-output' *eta* operations will be mutually exclusive, since their predicate inputs are mutually exclusive. The output *etas*, together with the output *mu*, essentially implement a decoded multiplexer. The key difference between this form of decoded multiplexer and the MUX described in Section 4.3.1 is that MUX synchronizes all of its inputs before clearing/acknowledging them, whereas here, each individual *eta* will acknowledge its own inputs.

Enforcing well-behavedness in the parent graph

As observed earlier, a mechanism is needed to prevent injection of additional values from a parent graph into this form of loop subgraph, due to the presence of a non-deterministic *mu* operation at the loop input.

As mentioned in Section 4.3.2, a simple means of achieving this would be to treat any compound transition representing a loop subgraph as an atomic transition within the parent graph, as described in Figure 4.2b. Recall from Section 4.3.2, that this would imply two constraints on performance: (1) call strictness, and (2) no re-injection of values until

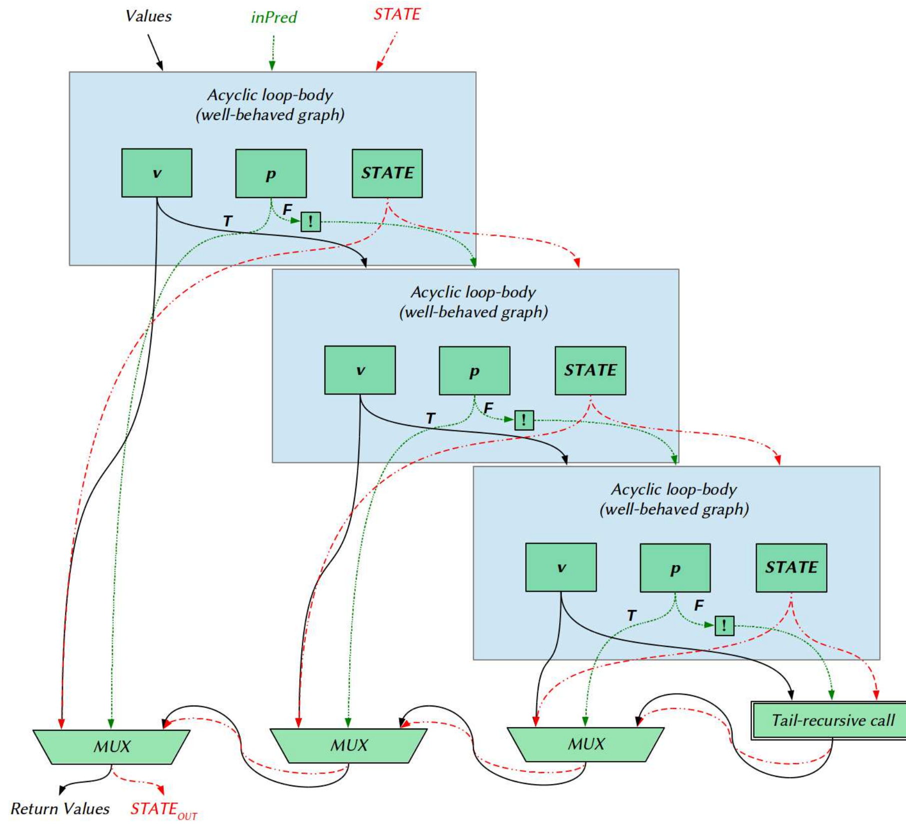


Figure 4.18: Loop structure from Figure 4.14, with the tail-call *flattened* twice.

previous inputs are acknowledged. For well-behaved subgraphs, both of these constraints were relaxed through the use of our *signal* and *wait* operations.

Due to the presence of the *mu* non-deterministic merge operation, VSFG-S loop subgraphs are not well-behaved. However, so long as re-insertion of values into a running loop subgraph is prevented, *and* the loop terminates, a loop subgraph is guaranteed to be *self-cleaning*, thanks to the *eta* operations at all of the outputs of the well-behaved acyclic loop-body VSFGs. Thus loop subgraphs need not be treated as strictly atomic operations – lenient insertion of operands is permissible, so long as subsequent operands are only passed into the subgraph *after* all previous outputs from it have been retrieved, thereby guaranteeing an empty loop subgraph.

To achieve this, variants of *signal* and *wait* from Section 4.3.2 have been developed, called *inGate* and *outGate*. Figure 4.20 shows the Petri-net transitions for *inGate* and *outGate* operations used within a parent graph to insert operands into a loop subgraph. The Figure assumes a structure analogous to Figure 4.13, where the loop subgraph has one value input in addition to the mandatory state-edge and *inPred* predicate inputs, and produces one value and one state output.

The *inGate* operation is analogous to *signal*, in that if its input predicate holds, it places a value token from the parent graph into a nested subgraph via a fused place. Unlike *signal*, however, *inGate* does not acknowledge its predicate or value inputs in the parent graph. Similarly, *outGate* is analogous to the *wait* operation, as it retrieves a value from a nested subgraph via a fused place if its input predicate holds. Unlike *wait*, *outGate* does not acknowledge its predicate input from the parent graph, though like *wait*, it does acknowledge values received from the nested-subgraph fused place. Furthermore,

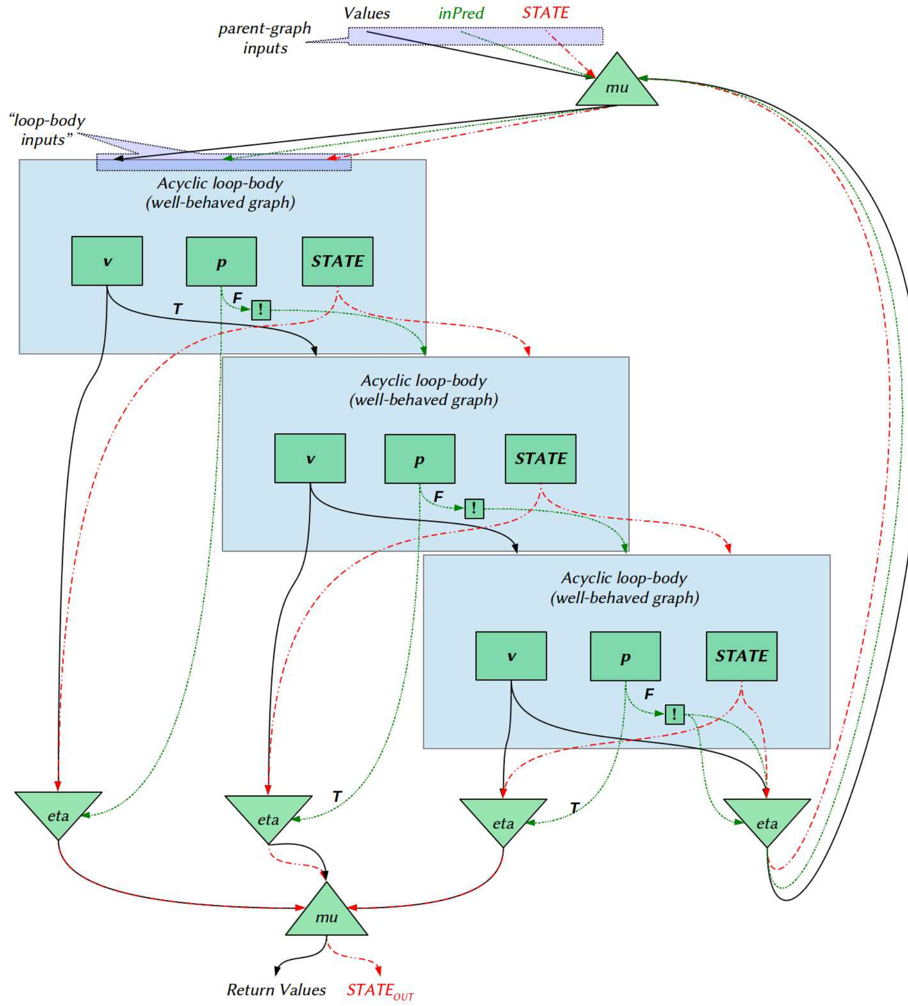


Figure 4.19: The equivalent VSFG-S loop with back-edges, for the VSFG shown in Figure 4.18.

each *outGate* also sets a local variable (**rcvd**) each time it fires, irrespective of the input predicate value.

Once all the *outGate* **rcvd** variables have been set, it means either that all output values have been received from the nested subgraph (if the parent predicate evaluated true), or that no output values were expected from the subgraph (if the parent predicate evaluated to false). In either case the *acknowledge_inputs* transition may fire, finally acknowledging all of the the input values and predicates from the parent graph to the nested subgraph. In this way, the combination of *inGate* and *outGate* operations allows for lenient activation of loop subgraphs, while maintaining well-behavedness by preventing re-insertion of value tokens until the loop subgraph has terminated.

$$f_1, f_2, ...f_N, = inGate\ (p_1, v_1, ..., p_N, v_N)\ \frac{\exists n.\big(\sigma(p_n) = \textbf{True} \land def(v_n) \land !def(f_n)\big)}{\sigma' = \sigma[f_n \mapsto \sigma(v_n)]} \tag{4.18}$$

$$f_1, f_2, ...f_N, = inGate\ (p_1, v_1, ..., p_N, v_N)\ \frac{\forall n.def(p_n, v_n), \forall m.(\sigma(\textbf{rcvd}_m) = \textbf{True})}{\sigma' = \sigma[\forall m.\textbf{rcvd}_m \mapsto \perp] \circ \forall n.erase(v_n, p_n)} \tag{4.19}$$

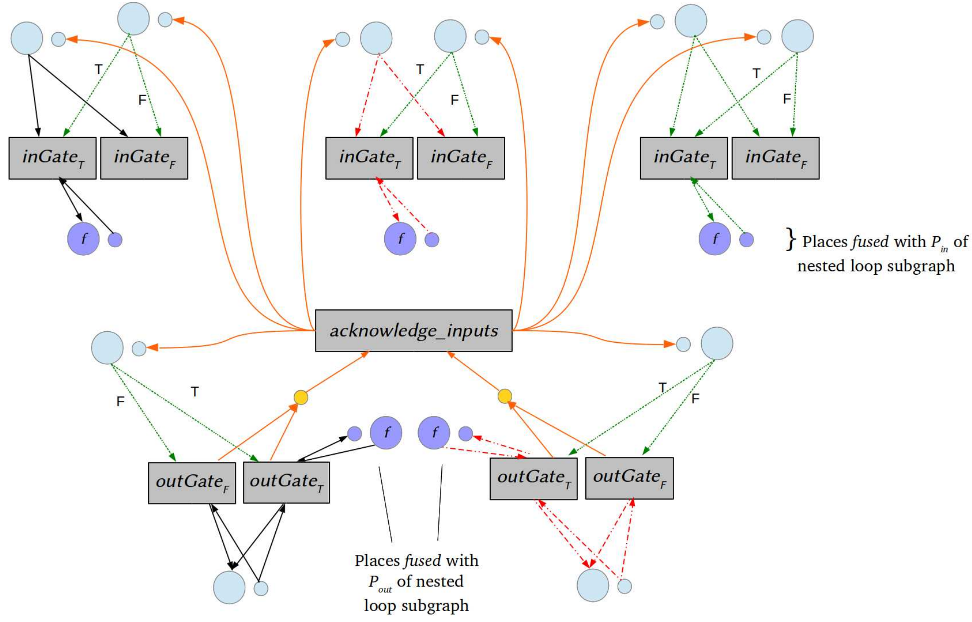


Figure 4.20: The use of *inGate* and *outGate* Petri-net transitions, along with an additional transition that acknolwedges all input places in the parent graph, but only after all outputs have been retrieved from the subgraph by the *outGate* transitions.

$$o_1, ..., o_M = outGate\,(p_1, f'_1, ..., p_M, f'_M)\ \frac{\exists m.\big(\sigma(p_m) = \textbf{True} \wedge def(f'_m) \wedge !def(o_m)\big)}{\sigma' = \sigma[o_m \mapsto \sigma(f'_m)][\textbf{rcvd}_m \mapsto \textbf{True}] \atop \circ\, erase(f'_m)} \tag{4.20}$$

$$o_1, ..., o_M = outGate\,(p_1, f'_1, ..., p_M, f'_M)\ \frac{\exists m.\big(\sigma(p_m) = \textbf{False} \wedge !def(o_m)\big)}{\sigma' = \sigma[o_m \mapsto \Delta][\textbf{rcvd}_m \mapsto \textbf{True}]} \tag{4.21}$$

Equations 4.18 through 4.21 describe the operational semantics for the *inGate* and *outGate* operations, under the assumption that there are N input arguments (including state and predicate inputs) to, and M output values (including state) from the nested-subgraph. Equation 4.18 describes the operational semantics for a set of *inGate* operations when the predicate inputs p_i evaluate true. Unlike *signal*, none of the inputs are cleared upon sucessful insertion of a value into a fused input place in the nested subgraph ($f_i \in P_{in}$ of nested subgraph). Equation 4.20 describes the semantics for the corresponding *outGate* operations if the predicate inputs p_i evaluate true. Subgraph output values are retrieved from the fused output places ($f'_i \in P_{out}$ of nested subgraph) if the predicates p_i hold. Note that the fused places f'_i are acknowledged by *outGate*, but the predicate places p_i are not, as they reside in the parent graph. Note also that each *outGate* operation sets its local variable **rcvd**, even if the input predicate is false (Equation 4.21). Equation 4.19 ultimately acknowledges all inputs once all have arrived, *and* all *outGate* operations have set their respective **rcvd** flags.

The combination of *inGate* and *outGate* with the *eta* operations within the loop behave very similarly to the well-behaved loop schema discussed previously in Section 4.2.1 (Figure 4.5b). Thanks to the isolation provided by *inGate* and *outGate* operations when communicating with loop subgraphs, the parent graph appears acyclic at its level in the hierarchy. This allows us to preserve the key advantage of the VSFG, i.e. the ability to perform loop unrolling within a level of hierarchy, independently of other levels.

For instance, the well-behaved loop-body of the unrolled loop shown in Figure 4.17

may itself contain a nested loop. So long as this inner loop is isolated from its parent graph using *inGate* and *outGate* operations, unrolling the parent loop, as in Figure 4.19 may be carried out independently of the inner loop.

4.4 Comparison with Existing Dataflow Models

4.4.1 Comparison with Pegasus

The VSFG-S IR, and the HLS toolchain for its evaluation described in Chapter 5 share a lot of similarities with previous work undertaken by Budiu et al on the Pegasus IR and the CASH compiler toolchain developed at Carnegie Mellon University [BVCG04, Bud03, MCC^{+}06]. We both compile general-purpose imperative code to a dataflow intermediate representation capable of being implemented directly as static-dataflow spatial computation. As a result, both the Pegasus and the VSFG-S IR utilize many of the same dataflow operations: value operations, *load*, *store*, MUX, *eta*, and *mu*.

The work presented in this dissertation is intended to improve upon the Pegasus IR and compiler work specifically by overcoming complex control-flow in order to expose and exploit greater ILP in spatial hardware from sequential code. Whereas Budiu's Pegasus IR is a variant of the Control Flow Graph, composed of acyclic hyperblocks, the VSFG-S is based upon the Value State Dependence Graph [Law07], that elides all explicit control flow, representing only true value and state dependences. This choice is expected to provide a significant performance advantage over the Pegasus IR, due to the ability of the VSFG-S to perform much broader control-dependence analysis, independent unrolling/pipelining of loops in a loop-nest, and exploit multiple flows of control, as described in Chapter 3. In addition, the following key advantages of the VSFG-S IR over the Pegasus IR have also been identified:

- **No Call Strictness:** Whereas Pegasus implements a strict *Call* operation that collects all arguments before initiating called function execution, the VSFG-S enables lenient communication of operands between parent graphs and their nested subgraphs through the use of *signal*, *wait*, *inGate*, and *outGate* operations.

  There are two interrelated reasons for call-strictness in the Pegasus IR. Firstly, instead of *inlining/flattening* all function calls as the VSFG-S currently does, Pegasus incorporates a call-stack mechanism to support the implementation of *true* function calls and general recursion. Secondly, Pegasus is based on the CDFG, and must therefore maintain the abstract notion of a single-flow of control, even though its implemented dataflow graph may be executing operations out-of-order, from across multiple hyperblocks. Given the last-in-first-out nature of the call-stack, call-strictness ensures that even in the presence of out-of-order dataflow execution, values are *pushed* to and *popped* from the call-stack in the correct order. Further discussion of this is presented in [Bud03].

  On the other hand, the key strength of the VSFG-S is that it abandons the single-flow of control constraint. Although the VSFG-S doesn't currently support true function calls or general recursion, a means of supporting both features in the future, *without* requiring a single-flow of control contraint or a sequentializing call-stack, is discussed in Section 4.5.

- **Reduced Control Overhead:** Typically, the dataflow execution model incurs a higher resource overhead than the conventional Von-Neumann model due to the need to incorporate a large number of *control-instructions* (like *switch*, *merge*, *eta*, and *mu*). One such operation is required for each operand that is subject to control-directed dataflow, whereas conventional code simply uses a single branch instruction to explicitly alter control-flow, independent of the number of live variables.

  Previous work compiling the Id Nouveau dataflow language found a $2-3\times$ instruction count overhead due to conditional instructions, compared to equivalent code written in C [HCAA93]. Wavescalar IR reveals a similar $2-4\times$ overhead in instruction counts [PPM+06], while Budiu et al report a $20-80\%$ performance penalty due to these instructions in the Pegasus IR [BAG05]. Both Wavescalar and Pegasus compile to their respective dataflow representations from imperative code, and must introduce a control-instruction for each live value at the entry and exit of each basic block (or hyperblock) in the original code CFG.

  The VSFG on the other hand elides most of the original control-flow. Consequently, the VSFG-S utilizes its own control-directed dataflow operations (i.e. *signal* and *wait*, *inGate* and *outGate*, *eta* and *mu*) only at the entry to and exit from subgraphs, instead of at each basic-block boundary. Within each subgraph, the VSFG-S only employs MUX operations at value join points to replace ϕ-nodes. As a result, VSFG-S based hardware should exhibit a much smaller instruction-count and performance overhead due to control-instructions. An indirect measurement of this is provided by the fact that VSFG-S based dataflow hardware exhibits only an average of 15% resource requirement overhead compared to statically-scheduled CFG-based custom hardware[10] when synthesized to an FPGA (please see results from Section 6.2.3).

Nevertheless, the work on Pegasus and the CASH compiler by Budiu et al does retain certain advantages, such as the support for general recursion and true function calls, as mentioned above. Furthermore, the CASH toolchain and generated harwdare incorporates both static and dynamic memory disambiguation to exploit memory level parallelism, whereas the VSFG toolchain currently does not. Finally, the HLS toolchain developed for this dissertation to evalue the VSFG-S IR does not currently provide full support for all imperative language features, whereas the CASH compiler supports compilation of virtually all of ANSI C to custom hardware [BAG05, BVCG04]. The incorporation of support for these features was left out of the current implementation primarily due to time constraints. A more detailed summary of these limitations and the reasons for them is provided in Section 5.4.

4.4.2 Relation to Original Work on Dataflow Computing

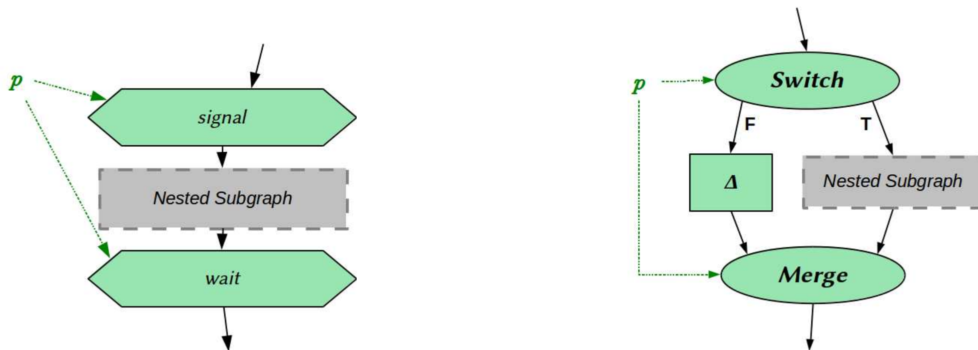
The design of the VSFG intermediate representation relies heavily on concepts developed during the early work done on dataflow computation in the 1970's and 1980's [AC86, Tra86, AN90]. The key distinction that could be made between the early work on dataflow and the VSFG IR is in the handling of control-directed dataflow:

[10] which, being statically scheduled, would not incur the same control-instruction overhead as the dataflow languages.

1. **Speculative execution via $\boldsymbol{MUX}$:** Early work on dataflow computing utilized the conditional schema from Figure 4.5a to implement control-directed dataflow [Tra86]. The VSFG instead relies extensively on the well-behaved MUX operator, to support speculative execution. Due to this, predicated operations like *load* and *store* must produce a don't care output even when their input predicate is false (i.e. they must be well-behaved independent of the conditional schema).

2. **Nested Subgraphs for representing function calls:** Early dataflow literature establishes the notion of '*user-defined functions*', which was analogous to the VSFG's representation of nested subgraphs [AC86]. However, instead of utilizing the conditional schema, the VSFG-S incorporates the predicated *signal* and *wait* operations to control the execution of such user-defined functions.

3. **Loops represented as atomic subgraphs:** Instead of the loop schema described by Traub (Figure 4.5b) [Tra86], the VSFG-S extracts loops into their own nested subgraphs, that execute atomically from the perspective of the parent graph thanks to the use of the *inGate* and *outGate* operations.



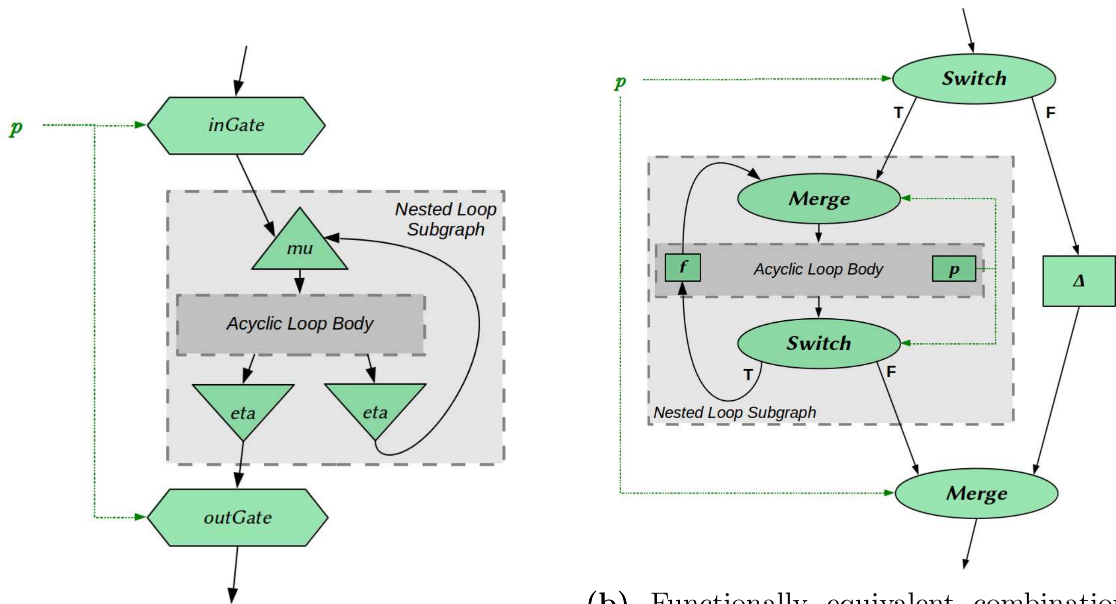
(a) Use of *signal* and *wait* to predicate the execution of a nested subgraph.

(b) Functionally equivalent conditional schema to Figure 4.21a. The Δ operation produces *don't care* as output.

Figure 4.21: A predicated subgraph implemented using *signal* and *wait* in the VSFG-S, and the functionally equivalent conditional schema described by Traub [Tra86].

However, these distinctions are merely abstractions that are useful for our purpose: compiling iterative programming languages to an intermediate representation that elides control-flow, and allows for aggressive speculation, control-dependence analysis and exploitation of multiple flows of control. For instance, the MUX operator is simply another primitive operation of the type shown in Figure 4.3a, defined to enable speculative execution. Furthermore, the predication of subgraphs using *signal* and *wait* (Figure 4.21a) is functionally equivalent to a conditional schema where the false path produces a don't care output, while the true path implements a *user-defined function* equivalent to the nested subgraph being predicated (Figure 4.21b). This is shown in Figure 4.21.

Similarly, the use of *eta* and *mu* to implement loops, while isolating the running loop from its parent graph using *inGate* and *outGate* operations (Figure 4.22a) is functionally equivalent to having a loop-schema enclosed within the conditional schema from Figure 4.21b, as shown in Figure 4.22b. In the same way as Budiu constructs the CDFG-based Pegasus IR from variants of the established basic dataflow operations [AC86, Tra86,



(a) Use of *inGate* and *outGate* to atomically execute a nested loop.

(b) Functionally equivalent combination of conditional and loop schemas to Figure 4.22a. The Δ operation produces *don't care* as output.

Figure 4.22: A nested loop subgraph implemented using *inGate* and *outGate* in the VSFG-S, and the functionally equivalent combination of conditional and loop schemas described by Traub [Tra86].

AN90], in order to represent imperative code [Bud03], the VSFG makes use of its own variants, for the same purpose, albeit utilizing a different organization, in order to overcome complex control-flow and expose ILP from control-intensive code.

4.5 Limitations of Static Dataflow Execution

The VSFG-S intermediate representation was developed with the static dataflow execution model in mind. This model was chosen for its potential for very high energy efficiency, as demonstrated in prior work [BVCG04, MCC$^+$06], but it also allows us to evaluate the VSFG-S in a simpler manner by implementing a high-level synthesis tool to generate static-dataflow custom hardware, instead of first having to develop, implement and tune a coarse-grained reconfigurable architecture like Wavescalar [SSM$^+$07].

Unfortunately, the static-dataflow execution model has one key limitation – it cannot dynamically invoke user-defined function calls [AC86]. While it can represent function calls as nested subgraphs, these subgraphs must be *flattened/expanded* into the parent graph at compile-time. Thus in the VSFG-S, each nested subgraph representing a function call would be implemented by *instantiating* a unique copy of the VSFG graph of that function at that location.

This implies that in a pure static-dataflow execution model, there would be no support for standard imperative language features like general recursion, since the number of times a recursive functions' subgraph is invoked is potentially large, and typically determined at runtime[11]. The development of the dynamic-dataflow model was motivated in part by

[11]Note that we already revert our acyclic, tail-recursive representation of loops back to a finite graph by

the need to overcome this key limitation of static-dataflow computing, thereby making the dataflow model more suitable for general-purpose computation [AC86].

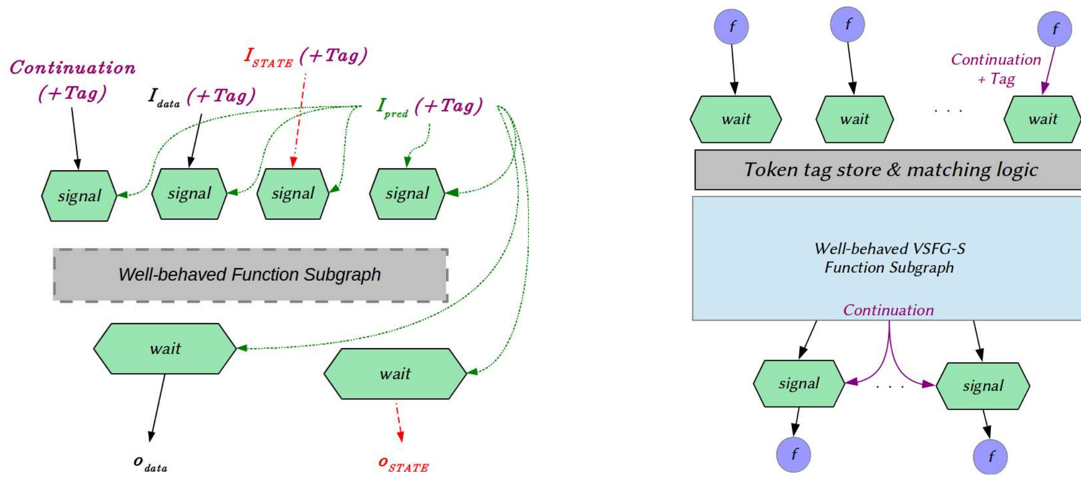
Unlike static dataflow, the dynamic dataflow approach associates a *tag* with each value or token in the graph, signifying its dynamic instance. Dynamic dataflow is (a) not constrained by the limitation of having only one token per dataflow edge, and (b) allows operations to consume input tokens out-of-order, by performing *tag-matching*. The token tagging and matching mechanisms allow a single instance of a function subgraph to be utilized to execute multiple independent function calls. General recursion can therefore also be supported by dynamic dataflow architectures, given sufficient memory space for the storage of tagged-tokens along each input arc to a recursive function. Note that this constraint is analogous to having sufficient *call-stack* space for a recursive program implemented on a conventional Von-Neumann architecture. Further details and discussion about both the static and dynamic dataflow execution models, and their implementation may be found in earlier work by Arvind, Traub, and Gao et al [AN90, AC86, GGP92, Tra86].

This key limitation of static-dataflow execution must be addressed if it is to be considered suitable for compiling unrestricted imperative high-level language code to spatial architectures. Previous work on compiling imperative code to static-dataflow hardware was undertaken by Budiu [Bud03], who incorporates a dynamic call-stack into his static dataflow model in order to implement function calls and support general recursive execution. The *caller* graph – i.e. the graph containing a function call in Budiu's Pegasus representation – must (1) allocate a stack frame, and (2) *push* all live values in the graph into this frame, before passing the arguments to the function call. Once the function call returns its outputs, all live values must be popped from the stack frame, and restored into the static-dataflow graph, after which the stack frame is freed, and execution proceeds as normal. Recursive functions adhere to the same procedure, allocating stack frames sequentially, and in control-flow order, in a similar fashion to the conventional imperative execution model.

Budiu's hybridization of the static-dataflow and imperative models is made possible by the fact that his Pegasus intermediate representation is based on the Control Flow Graph, and thus must strictly adhere to a single, sequential flow of control, at least at the coarse-grained, hyperblock level: while execution within a basic or hyper-block may occur concurrently in dataflow order, flow of control between blocks, and thus between caller and callee functions, must occur sequentially. Thus the last-in-first-out nature of the call-stack may still be utilized to implement function calls.

However, as execution in the VSFG is not constrained to a sequential or even a single flow of control, this stack-based approach cannot directly be ported to the VSFG, as its last-in-first-out nature does not guarantee retrieval of the correct operands in the absence of a single flow of control. Instead, I propose hybridizing the static-dataflow approach with the tagged-token dynamic dataflow approach, in order to add support for function calls and general recursion. This proposed approach is neither implemented nor validated, nor evaluated in this dissertation. These tasks are left for future work, while I focus on compiling non-recursive applications in this dissertation. However, a brief description is

reintroducing back-edges for the same reasons. However, if the number of general-recursive calls can be determined at compile-time, and the design is not resource constrained, there is no reason that recursive calls couldn't also be flattened into the original graph the desired number of times, in the same manner as loops are currently unrolled.



(a) Modifying subgraph communication to support true, non-inlined function calls.

(b) Implementing coarse-grained dynamic dataflow: incorporating token-storage and tag-matching, logic at subgraph boundary.

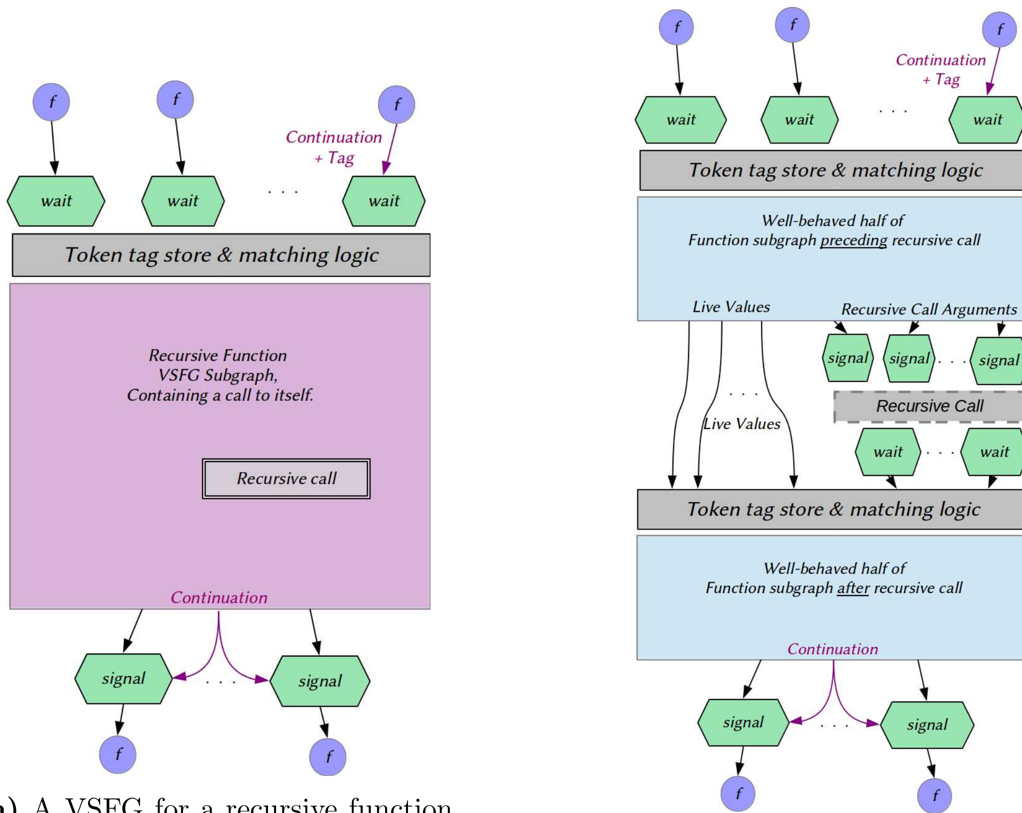
Figure 4.23: Proposed hybridization of static and dynamic dataflow, to retain the simplicity and efficiency of the former, while using the latter to support true function calls.

provided below, for completeness.

The goal of combining tagged-token dynamic dataflow with static dataflow would be to retain the generality and dynamic function call features of the former, with the efficiency and simplicity of the latter. To this end, the VSFG-S may be extended in the future to incorporate tag-matching at a coarse-grained level, where instead of incorporating tag-matching support for each operation in the VSFG, it may only be utilized at subgraph boundaries. For instance, instead of being flattened/inlined at each call site, a function subgraph may be implemented as shown in Figure 4.23.

For each call to a subgraph, the parent graph would dynamically generate a unique *tag* value. This tag would be passed along with each input argument to the function. The parent graph would pass an additional *continuation* or return address to the subgraph, as shown in Figure 4.23a. The *signal* operations in the parent graph would have fused places with corresponding *wait* operations at the entry to the function subgraph, as shown in Figure 4.23b. These *wait* operations would receive the incoming operands, along with their tags, from multiple potential call-sites, and insert them into the 'token tag store and matching logic', which may be implemented using content-addressable memory. Once a complete set of inputs to the subgraph has been received at this tag-matching logic (i.e. all inputs with matching tags+continuation have arrived), they may be issued to the well-behaved function subgraph, which would use *signal* operations at its exit edges to send output values back to corresponding *wait* operations in the correct parent graph specified by the *continuation* address.

The implementation of general recursion using such tagged-tokens may be performed as shown in Figure 4.24. The VSFG of a recursive function, as shown in Figure 4.24a, cannot be directly implemented as a VSFG-S suitable for static-dataflow execution, with dynamic-dataflow style tag-matching at its inputs. In order to implement a recursive function in our hybrid dataflow execution model, we must split its VSFG into two non-recursive, well-behaved regions: one that produces input arguments for the recursive self-call, and a second that consumes outputs from the returning recursive call, as shown



(a) A VSFG for a recursive function, containing a call to itself.

(b) Implementing recursion: split the function into two *halves*, and add tag-matching logic in the middle, after the recursive call.

Figure 4.24: Proposed hybridization of static and dynamic dataflow, showing how to implement a general recursive function.

in Figure 4.24b.

All values from the first half VSFG that are not arguments to the recursive call, are *live* values that must be held temporarily, until the recursive call returns. These values are therefore inserted into another instance of tag-matching, token-store logic at the input to the second half of the original VSFG, where they await the return of outputs from the current instance of the recursive call. Once the corresponding recursive call returns, the set of live values, along with the output from the recursive call, is inserted into the second well-behaved half of the VSFG, after which the function returns its outputs to the original calling parent graph.

Instead of incurring the cost of content-addressable logic for token storage and tag-matching for each operation in the VSFG, as is the case with the Wavescalar architecture, this hybrid approach should hopefully amortize this cost over a much larger set of dataflow operations by restricting tag-matching and token storage to the boundaries of VSFG-S subgraphs. Additionally, unlike the stack-based approach proposed by Budiu for his Pegasus IR [Bud03], this proposal does not necessitate constraining execution to a single-flow of control.

Additional work would be needed to fully flesh-out the details of how this hybrid approach may be implemented in custom or spatial hardware. For instance, a compiler might need to assess the relative overheads of inlining all calls to a function over implementing a

single body of the function with the associated tag-matching logic. Furthermore, how do we determine the best graph-cut when splitting a recursive function into its constituent halves? Finally, how suitable is the proposed approach for functions that make multiple recursive self-calls? In order to fully support general imperative languages for compilation to spatial architectures via the VSFG intermediate representation, validation, implementation and testing of this hybridization proposal is a high priority for future work beyond this dissertation.

4.6 Summary

In order to evaluate the potential improvements in ILP that the VSFG may provide for sequential, imperative applications implemented on spatial architectures, this chapter made the case for developing a high-level synthesis toolchain that compiles a VSFG-based IR directly to static-dataflow custom hardware. I described in detail the structure and semantics of the *VSFG-S*, discussing how loops and nested subgraphs may be represented for implementation as custom hardware. While a discussion of how to support true function calls and general recursion by incorporating some dynamic dataflow features is presented, implementation and validation of this approach is left for future work – currently the VSFG-S based HLS toolchain in-lines all function calls, and does not include support for recursion.

A brief comparison was also presented with the original dataflow computational model and operations, demonstrating that a lot of the new operations in the VSFG-S are essentially abstractions useful for eliding control-flow, and compiling imperative code to VSDG-like dataflow graphs. The presented implementations of predicated nested subgraphs and atomic loop subgraphs may easily be imitated using combinations of the original conditional and loop schemas presented in the early dataflow literature [AN90, AC86, Tra86].

Chapter 5 will now describe a high-level synthesis toolchain that compiles imperative code to the VSFG-S and then generates custom dataflow hardware for evaluation on an FPGA platform.

CHAPTER 5

A VSFG-Based High-Level Synthesis Toolchain

Chapter 4 presented the definition and operational semantics of the VSFG-S intermediate representation. The goal of the VSFG-S is to facilitate the implementation of sequential, imperative programs written in conventional high-level languages, onto spatial computation substrates using the static-dataflow execution model. The nature of the VSFG enables compile-time exposition of ILP from control-intensive code (as discussed in Chapter 3), while the static-dataflow model enables dynamic execution scheduling an energy-efficient exploitation of this ILP [BVCG04, MCC+06].

As highlighted in Chapter 4, development of a high-level synthesis tool that generates custom hardware descriptions from the VSFG-S IR is an effective and time-conserving way of evaluating the potential of this IR, without having to target a specific type of programmable spatial architecture like the ones described in Chapter 2. This chapter describes the design of this toolchain, after which Chapter 6 presents the evaluation results for the VSFG-S based hardware generated using this toolchain.

5.1 The Toolchain

I utilize the LLVM compiler infrastructure to implement this VSFG-based high-level synthesis toolchain. The toolchain takes the LLVM IR as input, generating VSFG-based static-dataflow hardware described using the Bluespec Hardware Description Language. The choice of the LLVM compiler infrastructure was motivated by several factors:

- **Ease-of-use:** The LLVM compiler is highly modular and well documented, making it a very convenient tool for academic research. Front-ends for many popular high-level languages are available that convert the input language code into the LLVM IR. All code analysis, optimization and transformation passes are implemented only on this LLVM IR, and thus can be developed independently of the input language.

  Furthermore, compiler *back-end* code generation and optimization passes can also be developed independently of the input language. An LLVM back-end would take in as input the generic LLVM IR representation of the input code, that may have undergone various back-end agnostic analysis and optimization passes, and generate output assembly or binary code targeting different architectures. Back-ends for

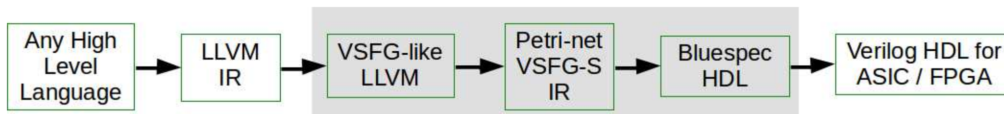


Figure 5.1: The language transformation steps involved in VSFG-based high-level synthesis. The representations shaded grey are the ones generated by the toolchain. It takes LLVM IR as input, and produces Bluespec SystemVerilog as output, which may subsequently be synthesized via Verilog HDL to either standard-cell logic or an FPGA.

different computational models (such as custom hardware [CCA+11]) or even other languages (for source-to-source translation) can also be developed.

- **Robust, established compiler:** As an established compiler undergoing rapid development and enhancement, LLVM provides front-ends for multiple high-level languages. Assuming that a VSFG-based spatial compiler is able to translate all of the LLVM IR to the VSFG IR, it would in theory be able to directly support the compilation of all high-level languages that have LLVM front-ends to spatial computation architectures.

  Furthermore LLVM provides as standard many well-tuned code analysis, transformation and optimization passes, that may be applied to the input code in its LLVM representation, before generating the code for a targeted back-end.

- **Static-single Assignment IR:** The LLVM IR is a simple, strongly-typed, static-single assignment (SSA) language. The SSA nature of LLVM simplifies dataflow analysis and optimization considerably, and the structure of a program represented using LLVM IR is essentially that of a Control-Data Flow Graph. The task of the toolchain is therefore primarily one of translating from this CDFG to the VSFG-S IR.

- **Simple, RISC-like instruction-set:** The LLVM instruction set is composed of simple operations, and is analogous to that of a RISC processor [LLV]. This makes it very easy to convert each non control-flow LLVM instruction into an equivalent dataflow operation in hardware[1].

Figure 5.1 presents an overview of the language transformation steps in the VSFG-based high-level synthesis toolchain, which generates and/or operates on the language representations shown within the grey region. The toolchain is essentially a back-end for LLVM, taking LLVM IR as input, and producing Bluespec SystemVerilog HDL output describing the application as VSFG-S IR based static dataflow hardware. This Bluespec HDL description is then compiled using the Bluespec *bsc* compiler [Nik04, Nik08] to generate Verilog HDL, that may then be further synthesized for implementation using a standard-cell library, or on an FPGA. There are two key reasons for selecting Bluespec HDL as the output language from the toolchain, instead of directly generating Verilog HDL:

[1]Note that in addition to scalar integer operations, the LLVM instruction-set also supports floating-point and vector arithmetic-logic operations. However, the described toolchain currently only supports the integer subset of the LLVM instruction-set.

- **Bluespec provides a higher level of abstraction than Verilog,** allowing for faster design experimentation and prototyping of hardware implementations.
- **Bluespec is well suited to expressing static-dataflow exection.** Instead of representing hardware at the register-transfer or structural level, Bluespec utilizes sets of guarded atomic actions, or *rules* to express the intended functionality for the hardware to be synthesized. The use of guarded atomic actions makes Bluespec highly suited to the implementation of dataflow-style dynamic execution scheduling. In particular, each transition in the Petri-net style execution semantics described in Chapter 4 for the VSFG-S can be easily mapped to an atomic rule in Bluespec HDL.

As shown in Figure 5.1, the LLVM IR representing the CDFG of the imperative input code is first *pre-processed* into an acyclic, VSFG-like format wherein all loops are normalized, extracted and transformed into tail-recursive functions, such that there are no explicit cycles remaining in the CFG. Then this pre-processed, acyclic LLVM IR is used to generate the equivalent Petri-net style VSFG-S representation as defined in Chapter 4. Next the VSFG-S IR is used to generate static-dataflow spatial hardware described using Bluespec SystemVerilog HDL, which is finally compiled using *bsc* to generate synthesizable Verilog. These steps are described in greater detail in the following sections.

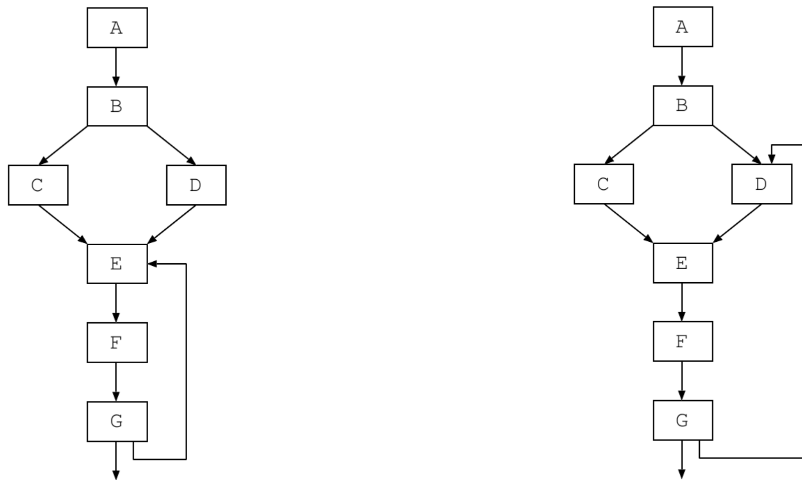
5.2 Conversion from LLVM to VSFG-S

The steps involved in the conversion of LLVM IR into the equivalent Petri-net style VSFG-S are as follows:

1. Transform and extract all loops into tail-recursive functions, such that the CFGs of all functions in the LLVM IR are now acyclic,
2. Introduce state-edges between side-effecting operations (including function calls), to enforce control-flow order,
3. Generate predicate expressions for each basic block in each acyclic function CFG,
4. Convert ϕ-nodes into MUX nodes,
5. Discard all branch instructions, and construct a VSFG-S from the remaining LLVM instructions, and the newly introduced predicate, MUX and state-edge operations.

5.2.1 Convert Loops to Tail-Recursive Functions

In order for a CFG to be transformed into an equivalent Value State Dependence Graph, and thus by extension, a Value State Flow Graph, all cycles within the CFG must be part of *natural* loops [Joh04]. A *natural* loop is one that has a single entry, or header basic block, such that none of the basic blocks that make up the body of the loop will have any control-flow predecessors outside the loop except for the header block, *and* all back-edges from within the loop enter the header block [ALSU06]. Essentially, all control flow entering a natural loop enters through the header block. Conversely, CFG cycles that



(a) A *reducible* CFG containing a natural loop.

(b) An *irreducible* CFG, with a CFG cycle but no distinct loop header.

Figure 5.2: Examples of reducible and irreducible code.

do not have a unique header node due to their being multiple control-flow entry points, are considered non-natural or *irreducible* code.

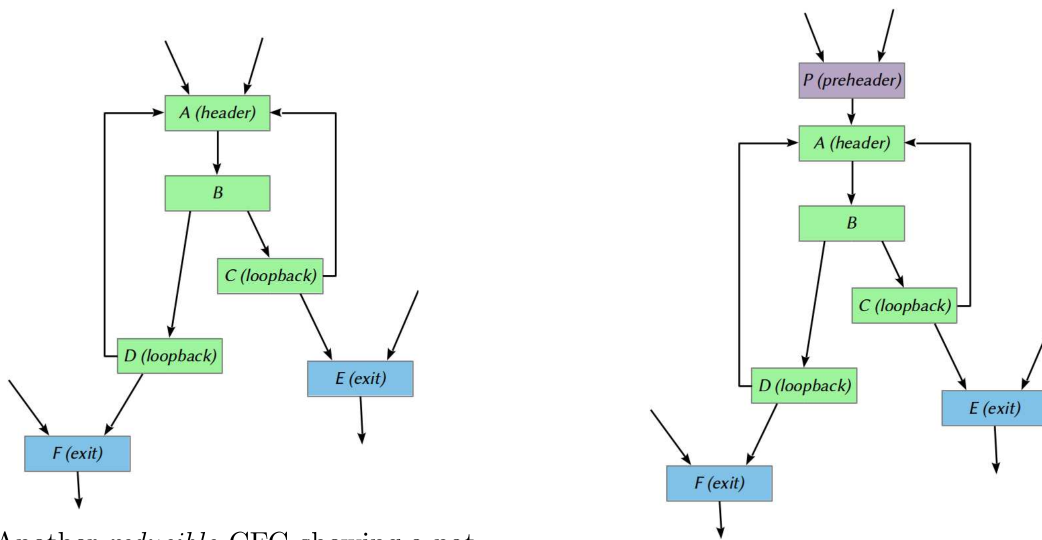
Figure 5.2a gives an example of a reducible CFG, showing a natural loop composed from the *E*, *F*, and *G* basic blocks. Block *E* is the loop header block, as control-flow only enters the loop at *E*, and the back-edge from the *loopback* block *G* also terminates at *E*. Figure 5.2b gives an example of irreducible code, where the *D*, *E*, *F*, and *G* basic blocks are part of a CFG cycle, but they do not constitute a natural loop, since control-flow can enter this cycle either from *C* (to *E*), or from *B* (to *D*). Both Figures are taken from [Joh04].

Irreduciblilty and its impact on VSDG formation are discussed in greater detail by Johnson [Joh04], who observes that it is not possible to compile irreducible code directy to a VSDG. Thankfully, natural loops represent the overwhelming majority of cycles in the control-flow graphs generated from modern high-level languages [SW11, ALSU06]. Furthermore, irreducible code may also be transformed into reducible code through various CFG transformations [UM02, JC97].

In order to extract such natural loops into their own tail-recursive functions, several transformations must be applied. For nested loops, the order of extraction is from the inner most loop to the outer-most.

1. **Loop preheader insertion:** For a natural loop that has a header block with multiple predecessor basic blocks, a new *preheader* basic block is created, such that the loop header will now only have one predecessor – the preheader block – while the header's original predecessors are now predecessors of the preheader block. Consider the natural loop CFG shown in Figure 5.3a, in which block *A*, the loop header, has multiple predecessors. Figure 5.3b shows the same CFG after the insertion of the preheader block.

2. **Merging all back-edges:** Next, all the back-edges are merged into one. This is done by introducing a new *loopback* block, as shown in Figure 5.4a (block *L*), and



(a) Another *reducible* CFG showing a natural loop.

(b) The same loop after preheader block insertion.

Figure 5.3: An example CFG showing another natural loop. Loop blocks are shown in green, while non-loop blocks are shown in blue. Each block introduced due to our transformations to the CFG are shown in purple.

re-directing the original back edges (from blocks C and D) to this block[2]. A new set of ϕ-nodes must be introduced in L for each loop-variant variable to select the appropriate value for the next iteration.

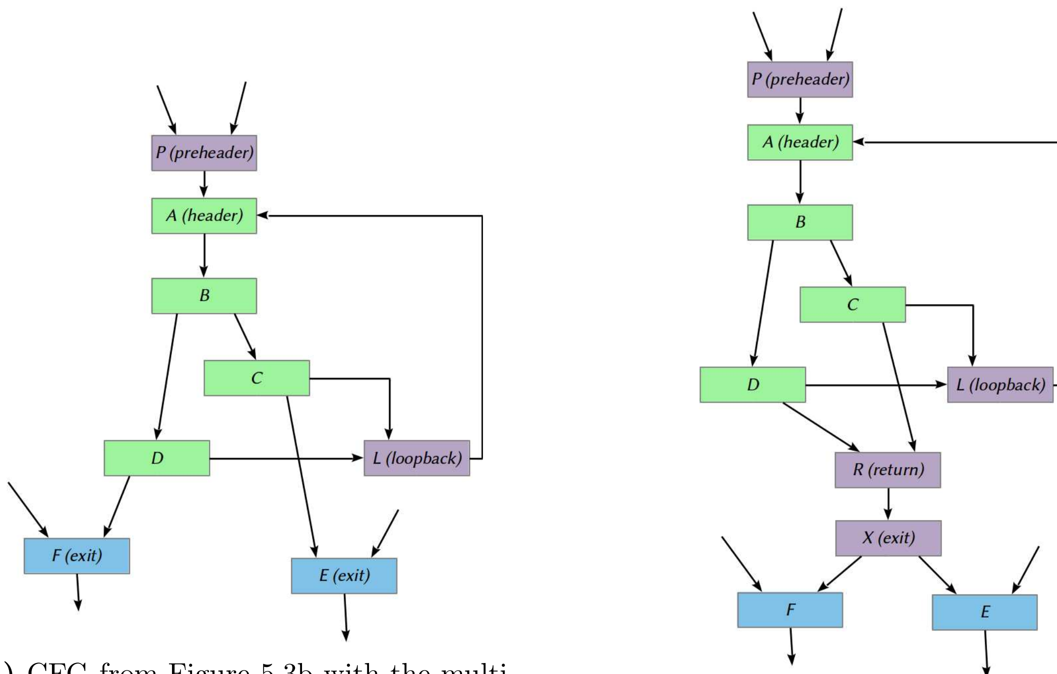
3. **Merging all loop-exit edges:** Then, all edges exiting the loop are also merged into a single exit edge. This is implemented by introducing two new basic blocks, as shown in Figure 5.4b: a *return* block R, and a new *exit* block X. The exit edges from both blocks C and D are redirected to the return block R, which has a single edge to block X. Block X then redirects control flow to the original exit blocks E and F.

   Note that a new *select* value must be introduced into the program in order to implement this correctly: a new ϕ-node is introduced into the R block, that indicates which of the unique loop exits is being taken (i.e. from D or from E). This *select* value is then used by the exit block X to conditionally redirect the control flow to either E or F.

4. **Loop extraction:** At this point, all of the blocks constituting the transformed loop can be extracted into their own function definition. As shown in Figure 5.5a, every block between the header block A and the return block R inclusive, is extracted into a new function, and replaced in the orignial CFG by a call to this new function.

   Before construction of the new function declaration, it is essential to identify all of the variables in the program that are live when control-flow enters the header block A – these variables must be added to the input argument list of the new function. Similarly, it is necessary to identify all of the variables that are updated within

[2] Given that we start from the inner-most loop in a loop nest, all nested inner loops would have already been extracted into their own tail-recursive functions, thus their back-edges are not considered here.



(a) CFG from Figure 5.3b with the multiple back-edges merged into one.

(b) CFG from Figure 5.4a after merging loop-exit edges. Note the introduction of the return block R and the exit block X.

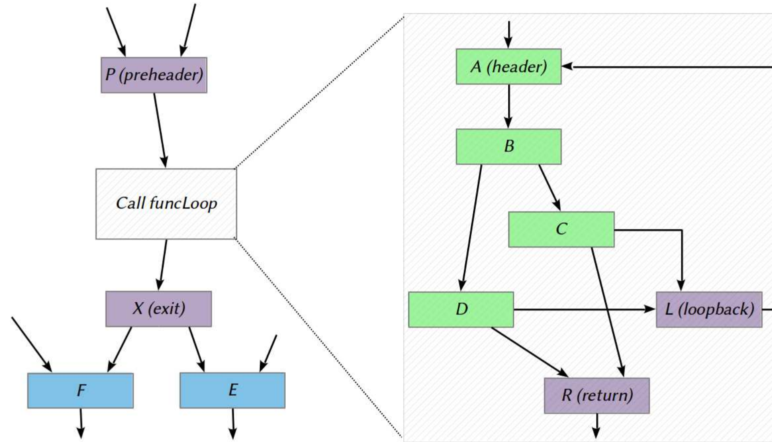
Figure 5.4: The loop CFG from Figure 5.3a after further transformations. Loop blocks are shown in green, while non-loop blocks are shown in blue. Each block introduced due to our transformations to the CFG are shown in purple.

the loop body, *and* are live when control-flow reaches the exit block X. For these variables, new stack memory is allocated in the preheader block P, and the pointers to these memory locations are added to the argument list of the new function.

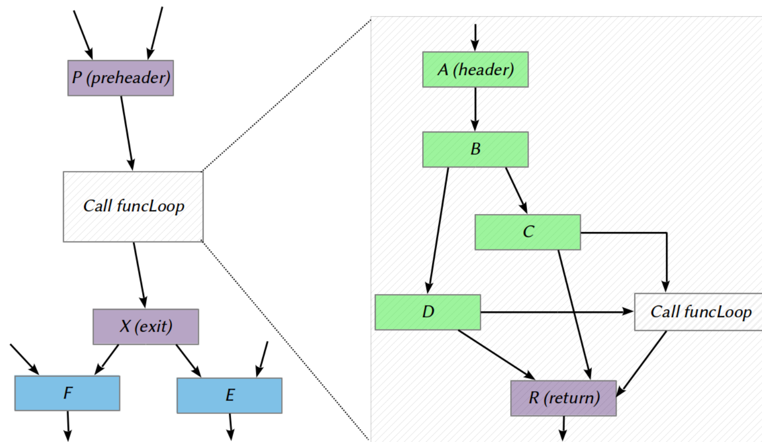
Within the loop body, at block R, just before returning control to the parent block, all of the updated values of these variables are stored to their respective stack locations. Corresponding load operations to retrieve the loop outputs from the stack are also inserted into block X. Thus communication between the parent function and the extracted loop function occurs through stack allocated memory[3]. The *select* value from the previous step that is computed in R, is set as the return value from this new function.

5. **Conversion to Tail-recursion:** This final step modifies the CFG of the extracted loop function by removing the back-edge exiting the loop-back block L, adding a recursive self-call to L, followed by an unconditional jump to the return block R, as shown in Figure 5.5b. However, before this can be implemented, the function definition in the previous step must again be modified – all loop variant values in the loop must also be added to the function's input arguments list, so that each time the tail recursive call is made, the updated values of the loop variants may be passed into the 'next iteration'.

[3]In the actual VSFG implementation, these stack allocation and access operations are identified and optimized away – instead, for each such stack pointer variable added to a function's argument list I consider the VSFG function subgraph as having an additional output port.



(a) CFG from Figure 5.4b with the basic blocks constituting the loop extracted into a separate function.



(b) CFG from Figure 5.5a with the loop back-edge replaced by a recursive self-call.

Figure 5.5: The loop CFG from Figure 5.4b after further transformations. Loop blocks are shown in green, while non-loop blocks are shown in blue. Each block introduced due to our transformations to the CFG are shown in purple.

The LLVM compiler provided considerable infrastructure that could be adapted to implement the above functionality for extracting natural loops into tail-recursive functions. LLVM includes a *-loop-simplify* transformation pass, that performs the first two transformations listed above. Another two useful passes are the *-loop-extract* and *-merge-return*, that together implement transformations 3 and 4 above. Thus, just by utilizing existing LLVM passes, 4 of the required 5 transformations can easily be performed.

Implementing transformation 5, i.e. replacement of the loop back-edge with a tail-recursive call, required modification of the existing *-loop-extract* pass to (a) add the loop-variant values to the extracted functions' input arguments lists, and (b) add the tail-recursive call to the loop-back block and remove its back-edge. This modified *-loop-extract* pass has been called the *-loop-2-tail-recurse* pass.

After the *-loop-simplify*, *-loop-2-tail-recurse*, and *merge-return* passes have been applied to the input LLVM IR, each function in the IR will have an acyclic CFG. At this point, the subsequent steps may be carried out in order to generate the predicate expressions and construct the state-edge graph for the VSFG.

5.2.2 Implement State-edges between State Operations

Once the tail-recursion transformations are applied, the LLVM IR CFG for each function in the input code will be acyclic. State-edges are now introduced between all state operations in the IR. Each side-effecting instruction (load, store and function call) in the LLVM IR is augmented with new state-token input and output ports. Then, within each basic block, the state output port of the first side-effecting instruction is connected to the state input of the second side-effecting instruction, and so on, until the end of the block. If there are no state operations in a block, the block's incoming state-edge is directly passed to its output.

Between basic blocks, the state-edge connectivity mirrors the structure of the acyclic CFG. The state-edge splits at the exit of a basic block when it has multiple possible successor blocks. Similarly, if a block has multiple predecessors, i.e. it represents a control-flow join point, then the state edges from each of the predecessor blocks must *join*, or synchronize before entering the current block. This synchronization is performed by introducing a MUX operation (as described in Section 5.2.4) for the state-token edges at the beginning of each basic block that has multiple predecessors.

An example is shown in Figure 5.6, which considers the now-acyclic CFG of the tail-recursive loop implementation from Figure 5.5b. The red dashed line indicates the state-edge. As can be seen, the state-edge *splits* at the exit of blocks B, C, and D, which means that the state-output port of the last side-effecting operation in each of these blocks will have a fan-out greater than 1 (fanout of 2 for each of these blocks). At control-flow join points, MUX operations are implemented to synchronize incoming state edges for both the tail-recursive call block and the return block R. Each MUX operation will wait for the state-token from the correct control-flow path to arrive before allowing the execution of subsequent side-effecting operations. In the VSFG-S, the correct control-flow path is determined using the MUX predicate inputs (not shown for clarity). Note that one of the state-operations in the *Call funcLoop* block will be the tail-recursive loop call, thereby ensuring that state operations are ordered correctly across loop iterations.

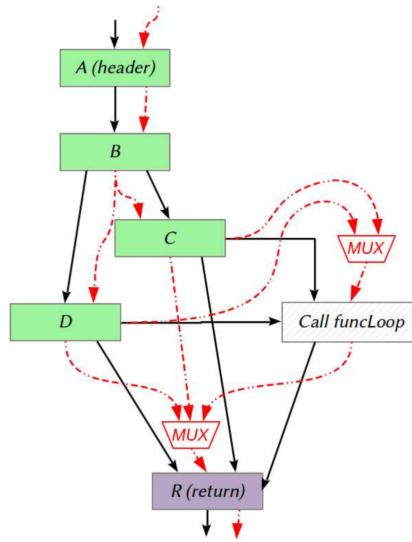


Figure 5.6: The tail-recursive function CFG from Figure 5.5b showing state-edges and MUX operations inserted between basic blocks (red dashed lines).

5.2.3 Generate Block Predicate Expressions

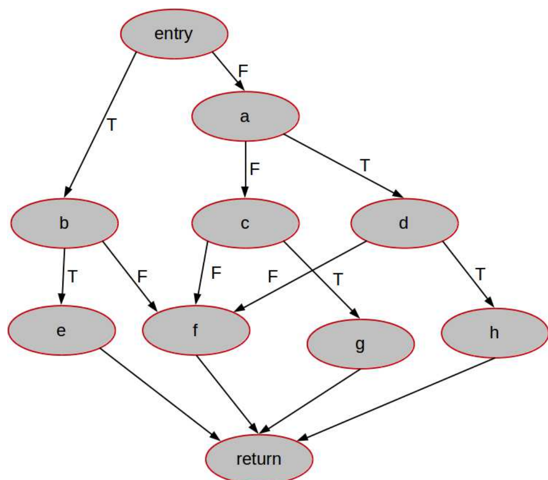
Predicate expressions are also generated for each basic block and for control-flow edge in the acyclic CFGs, as described in Section 3.6. A new set of boolean predicate operations is incorporated into the program representation, one for each basic-block. Then, each state operation in the CFG is augmented with a predicate input port, which is connected to the new predicate operation for its host basic block. These predicate operations are used not only to predicate the execution of state operations and function subgraphs, but also provide the predicate inputs to the state and value MUX operations.

Due to constraints of time, comprehensive boolean expression minimization for these predicate expressions (as described in Section 3.6) has not been incorporated into the toolchain at present. Instead, a simpler optimization is implemented based on dominance and post-dominance relations between basic blocks. For nodes in a directed graph with an entry node N_0 and an exit node N_∞, dominance and post-dominance are defined as follows (definitions adapted from Johnson [Joh04]):

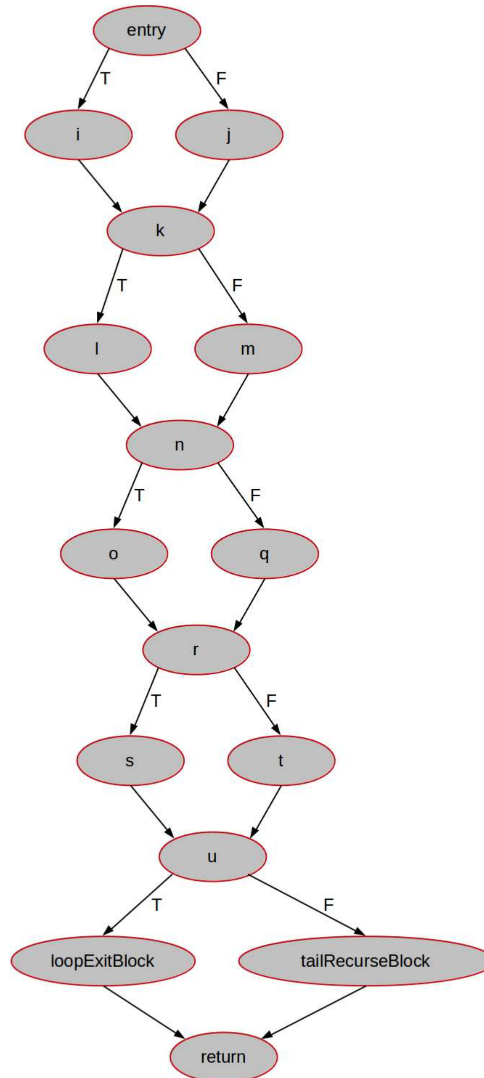
Definition 5.1. Dominance: In a directed graph with an entry node N_0, a node p is said to *dominate* node q if and only if every path from N_0 to q must traverse p. This relation is written as $p\ dom\ q$.

Definition 5.2. Post-dominance: In a directed graph with an exit node N_∞, a node q is said to *post-dominate* node p if and only if every path from p to N_∞ must traverse q. This relation is written as $q\ pdom\ p$.

Instead of applying boolean logic minimization to optimize predicate expressions and provide control-dependence analysis, as suggested in Chapter 4, I implement a simpler optimization which relies on identifying basic-blocks that are *control equivalent*. Two basic blocks are said to be control equivalent if every control-flow path from block N_0 to N_∞ that traverses one of the blocks is guaranteed to also traverse the other. This implies that after boolean minimization, the respective predicate expressions of both blocks would be identical.



(a) Acyclic CFG for a non-loop function.



(b) Acyclic CFG for a loop, extracted into its own, tail-recursive function.

Figure 5.7: Sample loop and non-loop CFGs from the AES benchmark in the CHStone Benchmark Suite [HTH+08]. These CFGs were generated using an LLVM pass, *after* preprocessing the benchmark code to extract loops into tail-recursive functions.

As an example, consider the example CFGs from Figure 3.14, shown again in Figure 5.7. From Figure 5.7a, we see that the *entry* and *return* blocks are control equivalent, since any path through this graph must traverse both. A more complex example is provided by Figure 5.7b: blocks *entry*, k, n, r, u, and *return* are control equivalent. As a result the predicate expression for the earliest block (*entry*) can directly be applied to the all of its control-equivalent blocks, as a means of simplifying their predicate expressions. Thus we have:

$$p(return) = p(u) = p(r) = p(n) = p(k) = p(entry) = inPred \tag{5.1}$$

Which is similar to the effect achieved by applying boolean minimization as discussed for Equations 3.4. Control equivalent blocks can be identified from the CFG through dominance analysis:

Definition 5.3. Control-equivalence of basic blocks: Two basic blocks p and q in a function CFG are said to be *control-equivalent* if and only if both p *dom* q, and q *pdom* p.

LLVM provides comprehensive analysis passes to evaluate dominance relations between basic blocks. These are used to provide a simple means of incorporating some control-dependence analysis into the VSFG. Currently, the VSFG toolchain only implements this control-equivalence based optimization of predicate expressions. Integration with boolean minimization tools like Espresso was not incorporated due to constraints of time, and is left for future work.

5.2.4 Replace each ϕ-node with a MUX

Once boolean expressions have been generated for each basic-block and conditional control-flow edge in each CFG, it becomes possible to convert all the ϕ-nodes within the LLVM IR into equivalent MUX nodes, as discussed in Section 4.3.1. In addition to replacing the *phi*-nodes, the newly introduced MUX operations for synchronizig state-edge joins (described in Section 5.2.2) are also provided with their appropriate predicate inputs.

5.2.5 Construct the VSFG-S

At this point, we have an acyclic LLVM CFG, with loops implemented using tail-recursion. Additionally, state operations have been augmented with state and predicate inputs, as well as a state output. State-edges have been introduced to sequentialize state operations in control-flow order, while predicates have been generated (and optionally optimized) for each basic-block. Finally, ϕ-nodes have been replaced with MUX operations. Having establised all of the necessary *value* and *state* dependencies, it is now possible to discard the explicit control-flow from the LLVM IR, and generate the equivalent VSFG-S Petri-net:

- All branch instructions are discarded from the LLVM IR.
- All remaining simple LLVM instructions are translated into their equivalent set of VSFG-S Petri-net transitions described in Chapter 4. ALU and *load*/*store* instructions become value and state transitions respectively, while ϕ-nodes become MUX transitions.

- All LLVM Call instructions to non-loop functions are replaced by the appropriate set of *signal* and *wait* transitions, one for each input and output argument of that function, as described in Section 4.3.2. This of course includes the new state and predicate inputs and outputs required for state operations.

- All LLVM Call instructions to tail-recursive (formerly extracted loop) functions are similarly replaced by a set of corresponding *inGate* and *outGate* transitions, as described in Section 4.3.3. The VSFG-S subgraph of each function is replicated for each of its call-sites in a parent graph, since our static-dataflow execution model doesn't currently support *true* function calls, as discussed in Section 4.5.

- Each value or state edge in the augmented LLVM CDFG is assigned a Petri-net place[4] if an instruction has a fanout of n, then n Petri-net output places will be introduced for its equivalent transition(s). Also, places are introduced at the input and output ports of each VSFG-S subgraph, *fused* with their corresponding *signal* and *wait*, or *inGate* and *outGate* operations in their parent graph (Section 4.3.2).

- After the appropriate degree of 'loop-unrolling' (via tail-recursive subgraph flattening) has been performed for each the extracted loop functions, dataflow loop-back edges can be reintroduced through the incorporation of *eta* and *mu* transitions. Both of these transformations are described in Section 4.3.3.

Once a completely defined VSFG-S Petri-net has been generated from the input LLVM IR, the next step is to translate it into an equivalent static-dataflow hardware description via the Bluespec SystemVerilog HDL.

5.3 Conversion from VSFG-S to Bluespec

As described in Chapter 4 each instruction/operation in the VSFG IR is represented using one or more Petri-net transitions in the VSFG-S IR. Similarly, each producer-consumer communication edge has an associated, unique Petri-net place. The Petri-net based representation and execution model of the VSFG-S makes it very easy to generate the equivalent Bluespec HDL.

Consider the short sample of LLVM IR input code shown in Figure 5.8a. After preprocessing this IR as described in Section 5.2, we end up with an acyclic VSFG-like LLVM representation that contains state and predicate edges, and can be represented as a dataflow graph as shown in Figure 5.8c. Once this is converted to the equivalent VSFG-S, as described in Section 5.2.5, we get the Petri-net representation shown in Figure 5.8d.

Broadly speaking, each transition in the VSFG-S Petri-net can be described using a single atomic *rule* in Bluespec, while each 1-bounded place[5] in the VSFG-S Petri-net maps to a corresponding 1-place *FIFO* element in the Bluespec HDL. The equivalent Bluespec HDL describing the Petri-net functionality of Figure 5.8d, is given in Figure 5.8e. The first few lines of Bluespec code (shaded blue) represents the declaration of 1-place FIFOs corresponding to each of the 1-bounded places in Figure 5.8d. Next, there are three atomic rules (shaded grey), implementing each of the three transitions from Figure 5.8d.

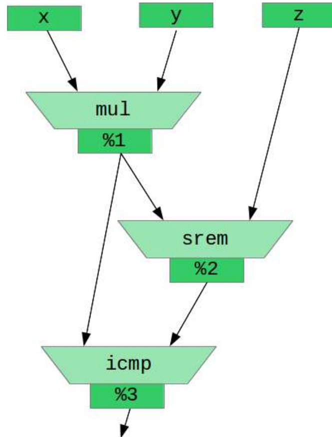
[4]Actually, a pair of places: a *value* place and its corresponding *acknowledgement* place, as discussed in Section 4.2

[5]Again, actually represented as a pair of places: a *value* place and its corresponding *acknowledgement* place, as discussed in Section 4.2

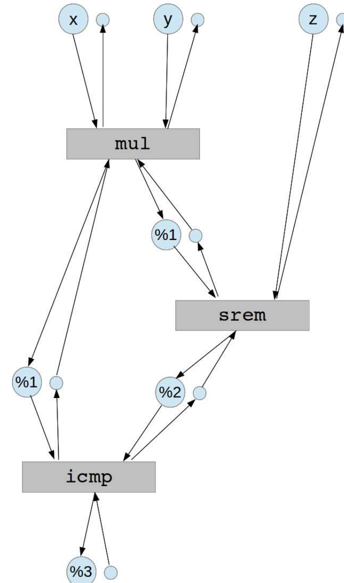
```
%1 = mul i32 %x, %y        ; <i32>
%2 = srem i32 %1, %z       ; <i32>
%3 = icmp slt i32 %2, %1   ; <i1>
```

(a) Sample LLVM IR.

(b) Symbol key for Figure.



(c) Dataflow graph suggested by the pre-processed, VSFG-like LLVM IR



(d) The equivalent VSFG-S Petri-net

```
FIFOF(int) x           ← mkFIFOF1;
FIFOF(int) y           ← mkFIFOF1;
FIFOF(int) z           ← mkFIFOF1;
FIFOF(int) srem_1      ← mkFIFOF1;
FIFOF(int) icmp_1      ← mkFIFOF1;
FIFOF(int) icmp_2      ← mkFIFOF1;
FIFOF(int) out_3       ← mkFIFOF1;

rule mul_inst;
    let val1 = x.first; x.deq;
    let val2 = y.first; y.deq;
    let rslt = val1 * val2;
    srem_1.enq (rslt);
    icmp_1.enq (rslt);
endrule

rule srem_inst;
    let val1 = srem_1.first; srem_1.deq;
    let val2 = z.first; z.deq;
    let rslt = val1 % val2;
    icmp_2.enq (rslt);
endrule

rule icmp_inst;
    let val1 = icmp_1.first; icmp_1.deq;
    let val2 = icmp_2.first; icmp_2.deq;
    let rslt = val1 > val2;
    out_3.enq (rslt);
endrule
```

(e) The corresponding Bluespec HDL

Figure 5.8: Sample LLVM IR code, and the subsequent three intermediate representations identified in the grey region of Figure 5.1.

A rule in Bluespec is able to *fire* when all of its implicit and explicit conditions are met. While the shown rules have no explicit conditions, implicit conditions are imposed by the status of the FIFOs being accessed from within each rule. As with the Petri-net transitions, each of the corresponding rules is able to fire when its input FIFOs are not empty, *and* its output FIFOs are not full. A firing rule then removes values from its input FIFOs and places a result value in each of its output FIFOs, thereby implementing the intended static-dataflow execution semantics. In Figure 5.8e, a *.first* operation on a FIFO accesses the contained value, while a *.deq* command removes a value from the FIFO. The former is analogous to reading a token from a value place, while the latter is equivalent to removing the token from the value place and placing one in the corresponding acknowledgement place.

Note however that a key issue aries when we maintain such a 1-to-1 correspondence between Petri-net places and Bluespec FIFOs: the number of storage elements required to store the output of an operation becomes proportional to the fanout of that output[6]. This can incur a significant area, power and energy overhead in custom hardware, particularly for the deeply pipelined style of circuits that are generated by this toolchain. One possible optimization is to instantiate a single output register for each output value in the VSFG-S. Then, each of the FIFOs in Bluespec can be declared to be of *void* type – this would mean that although they are still useful for performing flow-control between rules, they would not implement any storage for the output values. Instead of a *.first* operation performed on unique FIFOs, values would be read directly from the unified output register. Only the *.deq* operation would be retained to implement flow control.

Unfortunately, the Bluespec compiler is still experimental and under development. Implementing this optimization led to compilation failure when attempting to synthesize the resultant Bluespec into Verilog. Thus currently the toolchain incurs an area overhead proportional to the average fanout of values in the circuit, which must be accounted for when evaluating results, as discussed in Sections 6.2.3 and 6.2.4.

It is also important to note here that the Bluespec compiler implements its own internal scheduler to order the execution of rules. However, the Bluespec scheduler has no effect on either complexity or performance of the generated designs. This is because the generated Bluespec code is quite low-level, describing each instruction and its input/output FIFOs, thus there are no resource conflicts or contention between rules for the Bluespec scheduler to affect.

5.4 Current Limitations

As detailed in Chapter 2, in addition to addressing the *amenability* problem of spatial architectures by improving their performance on control-intensive sequential code, it is also important to address the *programmability* problem of spatial computation. Many recent projects address the programmability by developing sophisticated, fully automated high-level synthesis tools that are capable of compiling unrestricted imperative language code to generate custom hardware [BVCG04, SSM+07, VSG+10].

[6]Since each place (and its corresponding acknowledgement place) are implemented as a 1-place FIFO in Bluespec, operations/transitions that have a fan-out of $n > 1$, will therefore implement n 1-place FIFOs, one for each fan-out value. This can be quite wasteful, as the registers within each such FIFO will be storing identical values.

In the design of the VSFG IR this research also strives to provide a similar, complete coverage of imperative language features. The end-goal is analogous to the LLVM compiler IR in that unrestricted, unmodified high-level language code ought to be compiled to the VSFG, or the VSFG-S, which may then be processed through various back-end compilation passes for implementation on a wide variety of spatial architectures and substrates. As mentioned earlier, this problem is greatly simplified by the selection of the LLVM IR as our input representation, since numerous robust LLVM front-ends for various high-level languages already exist. Thus ideally we'd like to be able to compile unrestricted, unmodified LLVM IR to the VSFG IR.

However, largely due to constraints of time, the static-dataflow VSFG-S and the associated high-level synthesis toolchain have certain limitations on the types of code that can be compiled. These constraints and their associated reasons and solutions are summarised below:

- **No irreducible code:** As mentioned in Section 5.2.1, and detailed by Johnson [Joh04], irreducible code cannot be compiled to the VSDG, thus nor to the VSFG, by extension. Thankfully, irreducible code occurs very rarely [SW11, ALSU06], and when it does[7], transformations may be applied to make it reducible [UM02, JC97]. No such transformations have been incorporated into the current version of the VSFG-S HLS toolchain, but may easily be incorporated in the future.

- **No support for recursion:** While the pure-dataflow VSFG IR can represent general recursion, the VSFG-S IR does not. However, as discussed in Section 4.5, this support, as well as support for true function calls, may be added to the VSFG-S by hybridizing it with the dynamic dataflow execution model. This is left for future work – currently, our evaluation focuses on non-recursive code, and in-lines all called function subgraphs.

- **No system integration:** Currently the HLS toolchain does not incorporate support for system calls (including dynamic memory allocation, software exceptions, etc.) or external library calls. These are not limitations of the VSDG or VSFG IRs, but rather are common across most high-level synthesis toolchains, due to the often stand-alone, self-contained nature of the custom hardware that is typically generated. Other researchers frequently overcome this by implementing hybrid architectures composed of a conventional host processor combined with spatial-hardware components [MCC+06, MG07, VSG+10, CCA+11]. This hybrid approach enables full integration with modern system architecture, as all OS functionality, system and library calls are implemented on the host processor.

  Another approach would be to develop a fully programmable spatial substrate like Wavescalar [SSM+07], which would then implement all OS features and library code, though this requires recompilation of the OS and all libraries for the target spatial architecture. Both approaches may be explored in the future when developing programmable spatial architectures targeting the VSFG.

- **No Complex Data-structures:** Aside from dynamically sized data structures, the current toolchain also does not support complex, static data structures like

[7] Irreducibility is sometimes introduced by very aggressive code optimization passes. For instance running the gcc compiler with -O3 flags will occasionally produce irreducible code.

structs, *unions*, or multi-dimensional arrays. This support was left out only due to time constraints and may easily be incorporated in the future.

None of the above limitations are inherent to the VSFG IR itself. With more development, it should easily be possible to compile unrestricted LLVM IR code to the VSFG IR. In fact, with the exception of the issue of OS integration, system calls etc, all of the other issues can easily be addressed even for the custom hardware oriented VSFG-S IR.

Nevertheless, for now, these limitations constrain the scope of evaluation possible with the current VSFG-S HLS toolchain, as benchmark applications must be carefully selected that exclude the above features. The evaluation methodology, including the selected benchmarks are discussed next in Chapter 6.

5.5 Summary

This chapter briefly summarised the structure of the VSFG-S based high-level synthesis toolchain that is used to evaluate the performance and energy characteristics of VSFG-based spatial dataflow hardware. Also highlighted were various limitations of the prototype toolchain and means of overcoming them in the future.

Chapter 6 will now present an evaluation methodology, results and discussion of the performance and energy characteristics of the VSFG-based dataflow hardware, compared to an existing high-quality statically-scheduled high-level synthesis tool, as well as a simple in-order and a complex out-of-order processor.

CHAPTER 6

Evaluation Methodology and Results

Chapter 4 presented a justification for generating static-dataflow custom hardware based on the VSFG in order to evaluate its potential for exposing and exploiting ILP from control-intensive code. Then, Chapter 5 described the design and implementation of a prototype high-level synthesis toolchain for compiling sequential, imperative code to the VSFG, and then to custom hardware. In this Chapter, I present the methodology for, and results of, comparing the performance, power and energy characteristics for this generated custom hardware.

6.1 Evaluation Methodology

The objective of the following experimental evaluation is three-fold: (1) evaluate the potential of the VSFG for statically exposing ILP from control-intensive sequential code, compared to equivalent, conventional CDFG-based custom hardware, (2) understand the energy and area cost incurred for the observed improvements in ILP, and (3) estimate the performance characteristics of the VSFG compared to a conventional superscalar processor.

Using the toolchain described in Chapter 5, results for three versions of the VSFG-S based hardware are presented: VSFG_0 has no loop unrolling/flattening, VSFG_1 has all loops unrolled once, and VSFG_3 has all loops unrolled thrice. In a two-level nested-loop, for the VSFG_1, this would mean that there would be two copies of the outer-loop body VSFG, each containing a nested subgraph representing the inner-loop, which itself would have two copies of its loop-body. Thus for the VSFG_1, a two-level nested loop would implement *four* copies of the inner loop body, and similarly, the VSFG_3 would implement *sixteen* copies of the inner loop. This explosive growth in area would be impractical for real-world implementations, which would not unroll all loops blindly, instead relying on profile-driven selection of which loops to unroll. However, the objective here is to understand the maximum achievable performance.

6.1.1 Comparison with an Existing HLS Tool

To provide a baseline for comparison, we use LegUp 2.0, an established, high performance HLS tool [CCA+11] to generate CDFG-based, statically-scheduled custom hardware. There are several key reasons why LegUp was considered a suitable choice for this comparison:

- LegUp is a recent but mature project, aimed at developing a state-of-the-art, open-source high-level synthesis tool. As a result, it has a very active community of developers and researchers contributing to it, with prompt and readily available technical support and bug-fixing when needed.

- Like our VSFG-based toolchain, LegUp is based on the LLVM compiler infrastructure. This allows us to apply identical optimization and transform passes to the input code, and retain confidence that any observed differences in the characteristics of the generated hardware from both tools are not merely due to differences in the type or quality of compiler optimization passes applied.

- It has been demonstrated that LegUp produces hardware of comparable quality to commercial HLS tools like *eXCite* from Y Explorations Inc. [Inc10, CCA+11].

The input LLVM IR to both LegUp and the VSFG toolchain is compiled with $-O2$ flags, and with no link-time inlining or optimization. LegUp 2.0 does not support any loop unrolling, so results for only one version of LegUp generated hardware are provided[1].

Both tools were run with *operation-chaining* disabled – meaning that generated hardware is fully pipelined, with each instruction in the IR having its own output register instead of being merged as combinatorial logic with a predecessor or successor. This was done because enabling operation-chaining would have masked the ILP improvements by reducing the degree of pipelining in the generated hardware to match the achievable operating frequency.

6.1.2 Comparison with Pegasus/CASH

Given that this work is closely related to the Pegasus IR and the CASH compiler [Bud03], and relies on the same static-dataflow execution model, ideally the best way to demonstrate the advantage of the VSFG over CDFG-based IRs would have been to compare it with Pegasus/CASH for the selected benchmarks. Unfortunately, requests to obtain the CASH compiler were unsuccessful within a reasonable time-frame. Furthermore, even if the CASH compiler had been made available, there would be additional causes for concern regarding its suitability for our demonstration:

- The experimental compiler would have been almost a decade old, and largely unmaintained during this time. Debugging or obtaining support for it would have been difficult.

[1]A more recent version, LegUp 3.0 has incorporated limited support for pipelining and modulo-scheduling of carefully constrained inner loops. The VSFG-S IR allows for far more robust unrolling of loops from multiple levels of a loop nest. Furthermore, since the results reported by LegUp 3.0 do not improve the results of LegUp 2.0 for the benchmarks being considered [Leg13], making the effort to update our testing infrastructure to use this latest version was deemed unnecessary.

- The CASH compiler uses the SUIF compiler [WFW+94] front-end and analysis passes, while developing its own optimization passes. On the other hand, both LegUp and VSFG are based on LLVM and make extensive use of the existing standard optimization passes in LLVM. This similarity brings confidence that any differences in the observed characteristics of the generated hardware are not due to any potential shortcomings of either of our experimental research compilers.

Due to the shared LLVM-IR based starting point for both LegUp and my VSFG-based HLS toolchain, it is easy to verify that observed differences in performance, area, energy, etc. could only be due to at most two factors: either the differing program representations (VSFG vs. CDFG), or the execution models (statically-scheduled vs. static-dataflow).

6.1.3 Comparison with Conventional Processors

A comparison of the generated custom hardware is also made against the estimated performance and energy cost for an in-order Altera Nios II/f soft-processor, as well as an Intel Core i7 Nehalem out-of-order superscalar processor. The Nios processor is evaluated through timing-simulation on an Altera Stratix IV family FPGA, while Core i7 performance is approximated using the Sniper Interval Simulator [CHE11]. All benchmarks were compiled for both processors using $-O3$ flags, with the *gcc* and *nios2-elf-gcc* compilers respectively[2] in order to maximize performance.

Evaluating performance: Presenting a comparison between custom hardware generated from different HLS tools is easily justifiable, as all such synthesized hardware (from both LegUp and the VSFG-based HLS tools) is simulated on the same Altera Stratix IV FPGA. Unfortunately, providing a precise performance and efficiency comparison between synthesized custom-hardware and conventional processors is complicated by their vastly different implementation technologies and operating frequencies, as well as differences in the way they may be integrated into a system. The execution time for any application is composed of three factors:

$$\textit{Execution Time} = n \times CPI \times 1/f$$

Where n is the total number of instructions executed at runtime, CPI is the average cycles-per-instruction, and f is the clock frequency. The value of n varies between different instruction-set architectures and implementation philosophies (e.g. CISC vs RISC), while CPI and f are more implementation dependent parameters. To simplify performance comparison across architectures with different ISAs, *cycle-counts* provide an effective means of abstracting away the effects of instruction-set design and implementation:

$$\textit{Execution Time} = \textit{Cycle Count} \times 1/f \quad \text{where } \textit{Cycle Count} = n \times CPI$$

I utilize *cycle-counts* as a measure of performance because, assuming similar operating frequencies f, they provide a good analogy for the degree of ILP exploited across different architectures. Furthermore, with this approach, the issues in the VSFG IR affecting the

[2]The use of $-O3$ flags was recommended for the Nios II/f in the Nios II Software Developer's Handbook [Alt11].

exposition and exploitation of ILP may be considered separately from the implementation issues during high-level synthesis affecting achievable operating frequency f[3].

While the Nios II/f soft processor is also evaluated on the same Altera FPGA, its design has been carefully tuned to achieve a high operating frequency (f_{MAX}), often giving it as much as a $3\times$ frequency advantage over the unoptimized custom hardware. Similarly, a full-custom Core i7 processor operating at around 2.6 GHz possesses an approximately $20 - 40\times$ clock frequency advantage over FPGA-implemented custom hardware, which typically runs at between $70 - 150$ MHz for complex designs. Considering ILP as represented by cycle-counts separately from the issues of optimizing implementation frequency allows us to evaluate the potential of the VSFG IR without being too confounded by low-level implementation details, design trade-offs (such as optimising designs for high-frequency), and HLS toolchain optimization issues at this early stage of work[4].

Finally, in order to eliminate the effects of the differing memory-access latencies that each processor implementation experiences, all memory operations are constrained to have a latency of one clock cycle: the Sniper simulator is configured such that the simulated Core i7 uses perfect L1 caches (100% hit rate), and a hit latency of 1 cycle, while the Nios is configured to access local block RAMs (BRAMs) on the FPGA, again with 1 cycle access latency. This is done to match the 1 cycle BRAM access latency of the generated custom hardware.

Evaluating power and energy: As with the evaluation of performance, it is difficult to provide a fair comparison of power and energy efficiency between the synthesized custom hardware implemented on an FPGA and a full-custom processor like the Intel Core i7. In the first instance, I present an energy cost comparison with the Nios II/f soft-processor only, as it at least shares the same implementation substrate.

For further analysis, it is reasonable to assume that the in-order, six-stage pipeline of the Nios II/f should provide much higher energy-efficiency than the out-of-order Core i7, if both were implemented using the same underlying substrate. To estimate the energy cost of such an equivalent version of the Core i7 processor, I refer to the empirical relationship between power and performance for sequential processors presented by Grochowski et al. [GA06]:

$$Power = Perf^{\alpha} \qquad \text{where } \alpha = 1.75$$

Using the available cycle-count results for the Core i7 and the Nios to estimate their relative performance difference ($Perf$) allows us to estimate the approximate power dissipation of our hypothetical Core i7 processor, relative to the Nios. Assuming the same f_{MAX} for both the Nios and this hypothetical *soft*/FPGA version of the Core i7, an energy-efficiency comparison of the VSFG-based hardware with this out-of-order processor is also presented.

6.1.4 Selected benchmarks:

As an early-stage prototype, the VSFG HLS tool-chain can only compile applications with some constraints, as mentioned in Chapter 5. Currently there is no support for C language *structs* or multi-dimensional arrays, nor for floating-point arithmetic operations.

[3] i.e. minimizing the longest combinational critical-path between registers in the hardware.

[4] Recall that HLS is being used as a test case to evaluate the VSFG – the objective for now is not to produce a production quality, competitive HLS toolchain.

Additionally, like most HLS tools, there is no support for general recursion, dynamic memory-allocation, or external system/library calls.

Due to these limitations our choice of benchmark applications is constrained. Nevertheless, in order to provide a meaningful evaluation of performance on control-intensive sequential code, I have selected six benchmarks from the CHStone benchmark suite, in addition to two home grown kernels. The CHStone benchmark suite [HTH+08] was developed in order to facilitate research into high-level synthesis tools. It is composed of several common C-language benchmark applications that have been modified to be entirely self-contained, by removing all dynamic memory allocation, library or system calls, and general recursion. The following benchmark applications were selected from the CHStone suite:

- **dfadd:** Implements IEEE-standard double-precision floating-point addition using 64-bit integers. This benchmark contains only a single loop, but exhibits considerable data-dependent forward branching through *if* statements, as well as bitwise ALU operations.

- **dfmul:** Implements IEEE-standard double-precision floating-point multiplication using 64-bit integers. This benchmark exhibits similar control-flow structure to *dfadd*.

- **dfdiv:** Implements IEEE-standard double-precision floating-point division using 64-bit integers. In addition to considerable forward branching, *dfdiv* also contains several data-dependent loops nested within the main loop. Furthermore, it requires long latency integer division operations to be selectively performed based on data-dependent control flow.

- **dfsin:** Implements a double-precision floating-point *sine* function using 64-bit integers. This benchmark exhibits multiple nested data-dependent loops, as well as function calls to code from the *dfadd*, *dfmul*, and *dfdiv* benchmarks, thus also exhibiting data-dependent forward branches and bitwise ALU operations.

- **mips:** This benchmark is a simplified (30-instructions) MIPS processor simulator implementing a sorting program. It is composed of a main outer-loop, containing four inner-loops, one of which contains data-dependent forward branching in the form of a two-level nested *switch* statement.

- **adpcm:** Implements the CCITT G.722 Adaptive Differential Pulse Code Modulation algorithm for voice compression. This benchmark implements both ADPCM encoding and decoding, and contains significant data dependent forward branches, multiple levels of nested loops, as well as function calls.

In addition, to the above benchmarks from CHStone, the following two home-grown kernels were also used as benchmarks:

- **epic:** This is the *internal_int_transpose* function identified as being difficult to accelerate in spatial hardware by Budiu et al. [BAG05], and discussed in Section 3.3. It consists of a nested-loop that exhibits significant outer-loop parallelism, but insufficient concurrency, as well as a long latency modulus (%) operation within its inner-loop.

- **bimpa:** This is a spiking neural-network simulator for constructing and simulating a network of 1000 neurons, and was developed as part of the BIMPA project [BIM]. I modified this benchmark to flatten all *structs* and multi-dimensional arrays, as well as to statically allocate all memory. This benchmark exhibits complex-control flow in the form of multiple levels of loop-nesting, with dynamic data-dependent control-flow. This benchmark is also highly memory intensive, as unlike the CH-Stone benchmarks listed above, neural-network simulation is dominated by communication of *spike* values between neurons through array updates. Most of the loops in the benchmark are easily vectorizable [NFMM13], however, for the purposes of this work, no vectorization optimizations are applied – the benchmark is treated strictly as sequential, imperative code.

Table 6.1: Table showing Raw Cycle-counts for LegUp, various versions of the VSFG, as well as the Altera Nios II/f and Intel Nehalem Core i7 processors.

		Full subgraph-speculation, no predicate optimization			Full subgraph-speculation, basic predicate optimization			Predicated subgraphs, basic predicate optimization			Conventional Processors	
	LegUp	VSFG_0	VSFG_1	VSFG_3	VSFG_0	VSFG_1	VSFG_3	VSFG_0	VSFG_1	VSFG_3	Nios II/f	Core i7
epic	1078444	1062439	980783	972593	1062436	524122	320344	1062436	528218	265170	3399634	200174
adpcm	71349	56145	51580	51186	56145	51580	51186	57860	51580	51186	119794	42662
dfadd	2391	2073	1737	1737	1999	1590	1574	1623	1574	1574	16441	15994
dfdiv	3029	4485	3460	2929	4405	3380	2824	3235	2825	2639	36487	15120
dfmul	941	928	679	647	916	671	625	916	671	625	7074	14072
dfsin	105773	78349	78275	78275	72007	71896	71896	77906	73231	73231	1420558	104953
mips	13414	14489	13438	12953	14489	13438	12953	14489	13438	12953	31082	29998
bimpa	142386696	114361494	98179648	97430648	114361494	98179648	97430648	114361494	98179648	97430648	373347552	39664956

6.2 Results

6.2.1 Cycle Counts

Aside from varing the degree of loop unrolling between VSFG_0 and VSFG_3, the VSFG HLS toolchain was configured to generate three different versions of the VSFG:

- Full subgraph-speculation, no predicate optimization.
- Full subgraph-speculation, basic predicate optimization.
- Predicated subgraphs, basic predicate optimization.

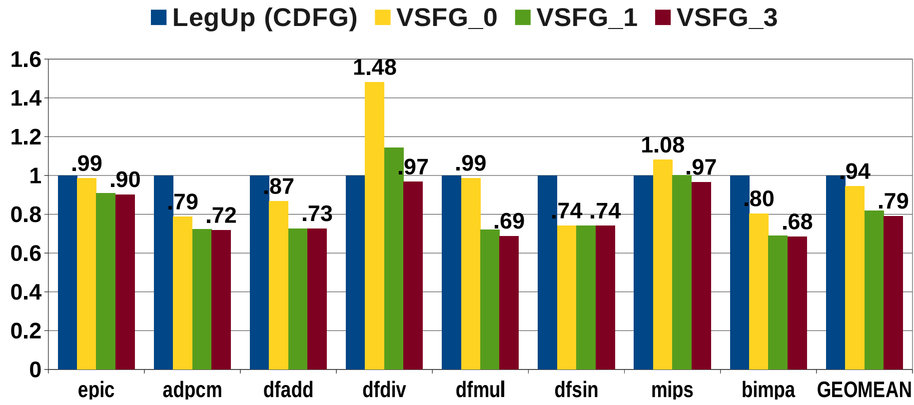
Table 6.1 shows the raw cycle-count results for all 3×3 versions of the VSFG, as well as for LegUp.

Table 6.2: Table showing Cycle-counts for VSFG with full subgraph-speculation, and no predicate optimization

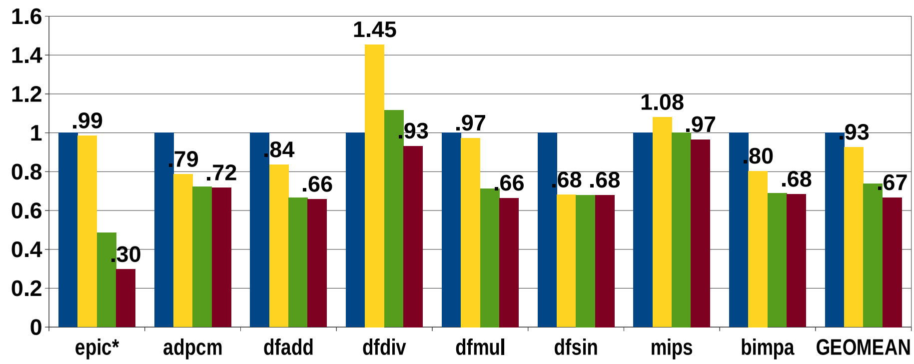
	LegUp	VSFG_0	VSFG_1	VSFG_3
epic	1078444	1062439	980783	972593
adpcm	71349	56145	51580	51186
dfadd	2391	2073	1737	1737
dfdiv	3029	4485	3460	2929
dfmul	941	928	679	647
dfsin	105773	78349	78275	78275
mips	13414	14489	13438	12953
bimpa	142386696	114361494	98179648	97430648

Full subgraph-speculation, no predicate optimization: For the very first evaluation of the generated VSFG-based hardware, no boolean expression minimization was performed on the predicate expressions generated for each basic-block in the original CFG. This meant that this implementation did not benefit from the control-dependence analysis described in Section 3.6. Nevertheless, in order to maximize performance, the VSFG implementations were initially configured to maximize speculative execution: all *signal* and *wait* operations were predicate promoted so that all nested subgraphs (except loop-subgraphs) would execute without waiting for their predicates to be computed.

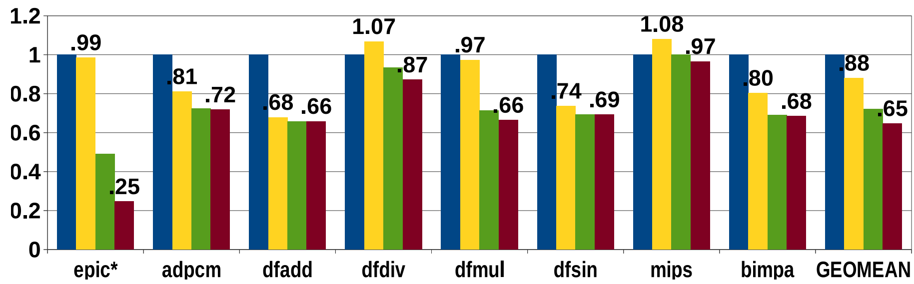
Table 6.2 shows the cycle-count results of this version of the VSFG (i.e. a subset of Table 6.1), while Figure 6.1a, shows all values normalized to the results for the hardware generated from the LegUp HLS tool. On average, the baseline VSFG_0 provides 6% lower cycle-counts than the equivalent LegUp generated hardware. Furthermore, as the degree of unrolling in the VSFG is increased from VSFG_0 to VSFG_3, there is a general improvement in performance for most benchmarks of an average 21%. However, the *epic* benchmark, despite a high degree of unrolling for both levels of the loop-nest, shows only a 10% improvement in cycle-counts relative to the baseline VSFG_0 and LegUp results. Additionally, the VSFG_0 results exhibit worse performance than LegUp for the *dfdiv* and *mips* benchmarks.



(a) Cycle Counts with speculative subgraph execution and without predicate optimization.



(b) Cycle Counts with speculative subgraph execution and predicate optimization.



(c) Cycle Counts with predicated subgraph execution and predicate optimization.

Figure 6.1: Performance Comparison (Cycle Count) Normalized to LegUp. Lower values are better.

```
/*=============================================                                 1
In-place (integer) matrix tranpose algorithm.                                   2
Handles non-square matrices, too!                                               3
=============================================*/                                 4
void internal_int_transpose(int* mat, int rows, int cols,                       5
   int modulus ){
  int swap_pos, curr_pos, swap_val;                                             6
                                                                                7
  for (curr_pos=1; curr_pos<modulus; curr_pos++) {                              8
    swap_pos = curr_pos;                                                        9
    do {                                                                        10
      swap_pos = (swap_pos * cols) % modulus;                                   11
    }                                                                           12
    while (swap_pos < curr_pos);                                                13
                                                                                14
    if (curr_pos != swap_pos) {                                                 15
      swap_val       =    mat[swap_pos];                                        16
      mat[swap_pos] =    mat[curr_pos];                                         17
      mat[curr_pos] =    swap_val;                                              18
    }                                                                           19
  }                                                                             20
}                                                                               21
```

Figure 6.2: The *'internal_int_transpose'* function from the *'epic'* Mediabench benchmark

Full subgraph-speculation, basic predicate optimization: To address these shortcomings, the basic predicate optimization described in Chapter 5 was incorporated into the generated VSFG-S hardware. Table 6.3 shows the cycle-count results of this version of the VSFG (another subset of Table 6.1), while Figure 6.1b shows the normalized cycle count results with this predicate optimization enabled. With this optimization, a dramatic improvement in the cycle count for the *epic* benchmark is observed as the degree of unrolling is increased. Figure 6.3 shows the CFG for the outer-loop of the *epic* benchmark kernel (code shown in Figure 6.2, and kernel described in Section 3.3). The *iterate* basic block contains the tail-recursive call of the outer-loop. Without predicate minimization, the predicate expression for the *iterate* block is given by the equation:

$$p(iterate) = p(condition)\,.\,\overline{p(condition, exit)}$$

where:

$$p(condition) = p(entry)\,.\,p(entry, condition)\; +\; p(swap)$$

where:

$$p(swap) = p(entry)\,.\,\overline{p(entry, condition)}$$

The computation of $p(entry, condition)$ is dependent on the result computed within the inner-loop. With predicate optimization however, the predicate expression for the *condition* and *iterate* blocks is simplified to:

$$\begin{aligned} p(condition) &= p(entry)\,.\,p(entry, condition)\; +\; p(entry)\,.\,\overline{p(entry, condition)} \\ &= p(entry) \end{aligned}$$

Substituting, we get:

$$p(iterate) = p(entry)\,.\,\overline{p(condition, exit)}$$

Thus initiation of the next outer-loop iteration is only dependent on the loop exit condition being computed within the *condition* block. As the inner-loop body, including its long-latency modulus operation, are no longer on the critical path for the computation of the outer-loop's loop-back predicate, multiple copies of the outer-loop may be activated concurrently as the degree of unrolling is increased between the VSFG_0 and VSFG_3 implementations. This means that the execution of multiple copies of the inner-loop, including its long latency modulus operator, may overlap, allowing for a dramatic reduction in required cycle counts for *epic*.

However, as can be seen from Figure 6.1b, all other benchmarks exhibit only a modest improvement in cycle-counts due to this optimization. The *dfdiv* and *mips* benchmarks still exhibit worse performance than LegUp hardware for VSFG_0.

Table 6.3: Table showing Cycle-counts for VSFG with full subgraph-speculation, and basic predicate optimization.

	LegUp	VSFG_0	VSFG_1	VSFG_3
epic	1078444	1062436	524122	320344
adpcm	71349	56145	51580	51186
dfadd	2391	1999	1590	1574
dfdiv	3029	4405	3380	2824
dfmul	941	916	671	625
dfsin	105773	72007	71896	71896
mips	13414	14489	13438	12953
bimpa	142386696	114361494	98179648	97430648

Predicated subgraphs, basic predicate optimization: Poor performance in the two exceptions (*dfdiv*, and *mips*) was found to be due to sub-optimal pipeline balancing in the static dataflow hardware [Gao91]. An example is shown in Figure 6.4a: even if the long latency path (64 cycles) is infrequently taken, the static dataflow nature of the graph means that the longer latency will still be incurred each time this loop body is triggered. In the case of *dfdiv*, there is a long latency divide operation (64 cycles, unpipelined) on one of the infrequently taken paths within its main loop body. Using predicated execution (Figure 6.4b) instead of speculation to trigger only the necessary path in each loop iteration can overcome this problem, but at the cost of any potential performance gain due to speculation on the frequent path[5].

Thanks to the hierarchical nature of the VSFG, we have third option – sub-graph predication. Unlike full predication, infrequent paths through the graph can be nested into their own subgraphs, and may then be selectively predicated, while the shorter, more frequently taken paths can still take advantage of speculation, as shown in Figure 6.4c. For the third version of the VSFG-based hardware, subgraph predication is implemented

[5] Predicated execution as shown here has been implemented using the conditional schema described in Section 4.2.1.

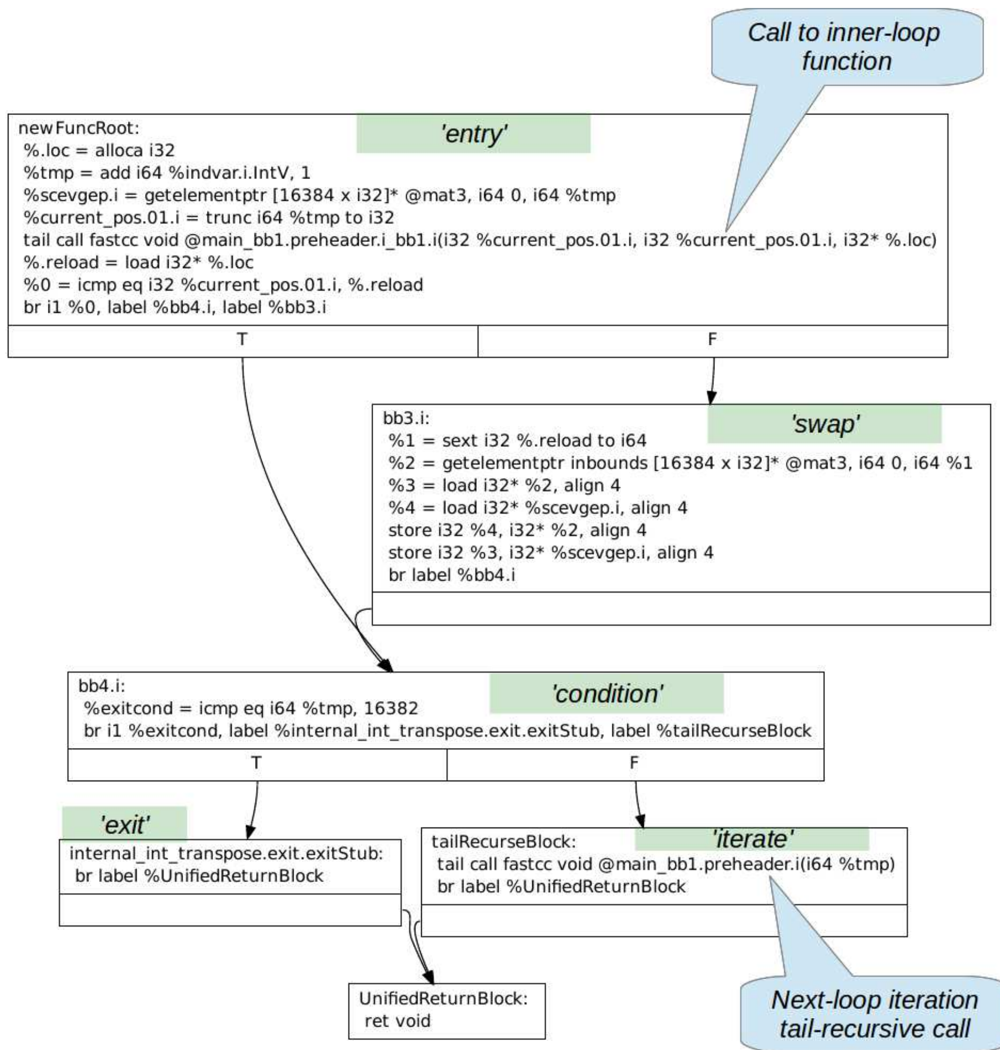


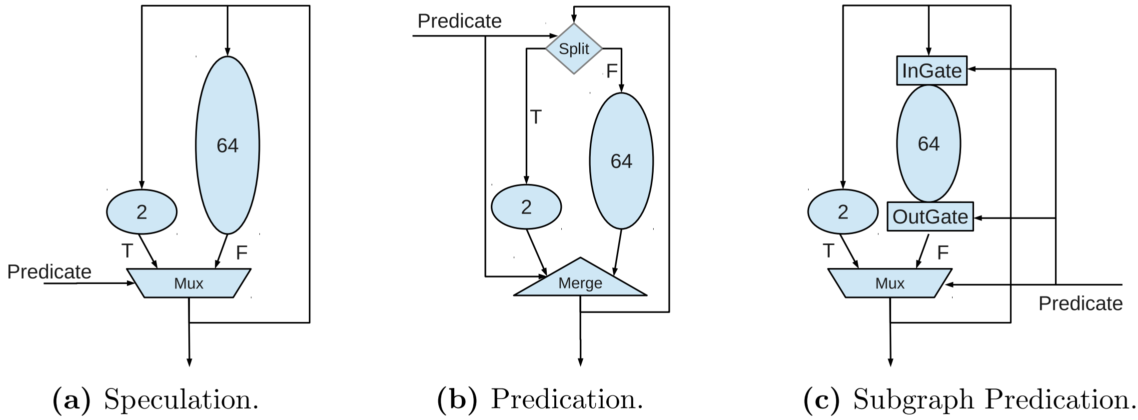
Figure 6.3: The LLVM CFG for the *epic* kernel outer-loop.

only on *pre-existing* subgraphs for all benchmarks, meaning that all non-loop subgraphs will now have their corresponding *signal* and *wait* operations in the parent graph be predicated[6]. Additionally, long-latency operations like division and modulus are also predicated.

As shown in Figure 6.1c, performance of VSFG_0 for *dfdiv* improves significantly, but there is no change for *mips*, as it contained no pre-existing subgraphs in its main loop that could be selectively predicated. However, it must be noted that the performance improvements do not apply for all benchmarks – *epic*, and *dfadd* exhibit slight improvements in VSFG_0, while performance for *dfsin* is slightly worse for all VSFG configurations, whereas *bimpa* and *dfmul* exhibit no change. On average, the VSFG_0 provides a 12% improvement in cycle count over LegUp, up from 7% for hardware with speculative subgraph execution. But this can largely be attributed to the performance improvement observed for *dfdiv*. Table 6.4 shows the raw cycle-count results of this version of the VSFG, extracted from Table 6.1.

Additional effort in identifying mutually-exclusive control-flow regions and nesting them into their own subgraphs is therefore warranted to further improve performance

[6]Recall that loop subgraphs are always predicated.



(a) Speculation. **(b)** Predication. **(c)** Subgraph Predication.

Figure 6.4: The Trade-off between Speculation and Predication. The VSFG enables a combination of both: Subgraph Predication.

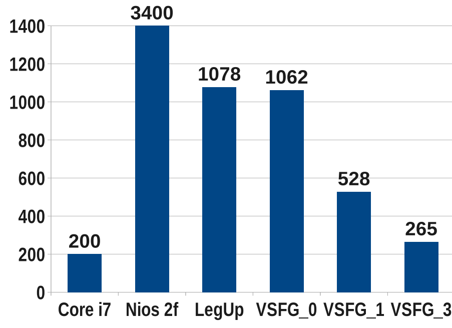
Table 6.4: Table showing cycle-counts for VSFG with predication of existing subgraphs and long-latency operations (/ and %), and basic predicate optimization.

	LegUp	VSFG_0	VSFG_1	VSFG_3
epic	1078444	1062436	528218	265170
adpcm	71349	57860	51580	51186
dfadd	2391	1623	1574	1574
dfdiv	3029	3235	2825	2639
dfmul	941	916	671	625
dfsin	105773	77906	73231	73231
mips	13414	14489	13438	12953
bimpa	142386696	114361494	98179648	97430648

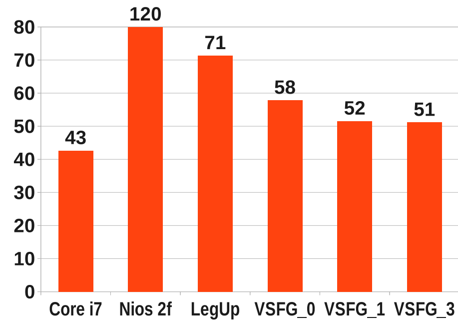
in the baseline VSFG_0 configuration. Further methods exist for pipeline balancing by performing optimal *dataflow software pipelining* [Gao89]. This has already been incorporated into previous work on Spatial Computation [Bud03], and should be incorporated into future versions of the VSFG toolchain as well.

Further performance improvements over VSFG_0 are achieved through aggressive multi-level loop unrolling. Figure 6.1c shows that VSFG_3 achieves 35% lower cycle counts than LegUp, by trading area for performance. Figure 6.5 compares LegUp and VSFG cycle counts to those of the Core i7 and Nios II/f processors. *epic*, *adpcm*, *dfsin*, and *bimpa* show the performance advantage of the superscalar Core i7 over the in-order Nios, as well as the CDFG-based LegUp. With unrolling, VSFG_3 is able to approach or exceed Core i7 cycle counts for all benchmarks with the exception of *bimpa*. Furthermore, for all benchmarks except *epic*, performance gains stagnate quickly between VSFG_1 and VSFG_3.

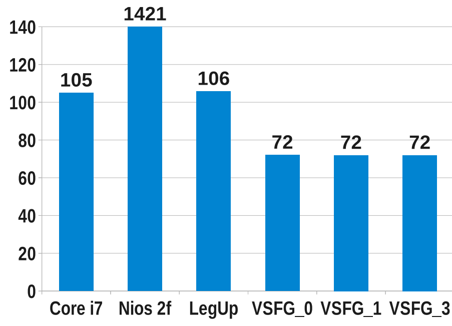
The reason for this is the lack of memory parallelism or reordering in the VSFG: all memory operations are constrained by the state-edge to occur strictly in sequential program order, whereas the Core i7 is able to dynamically disambiguate, re-order and issue multiple memory instructions each cycle. Yet despite this, only the memory intensive *bimpa* is where the Core i7 has a significant advantage over VSFG_3.



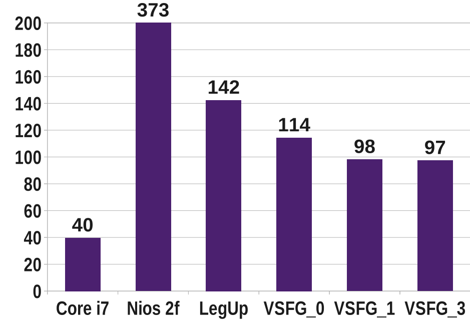
(a) epic (×1K cycles)



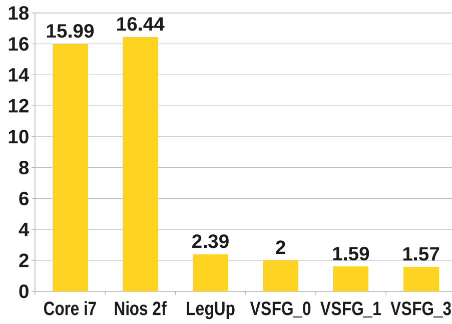
(b) adpcm (×1K cycles)



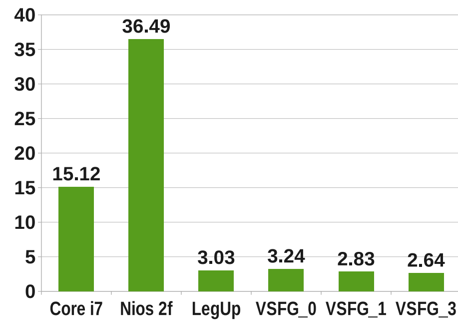
(c) dfsin (×1K cycles)



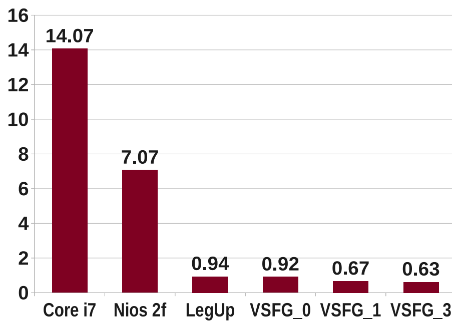
(d) bimpa (×1M cycles)



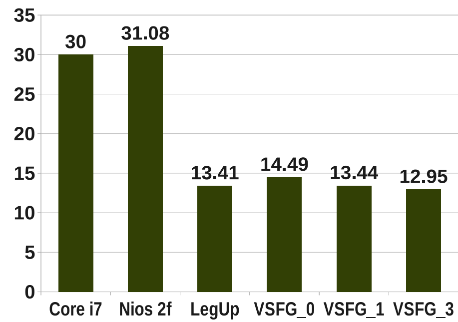
(e) dfadd (×1K cycles)



(f) dfdiv (×1K cycles)

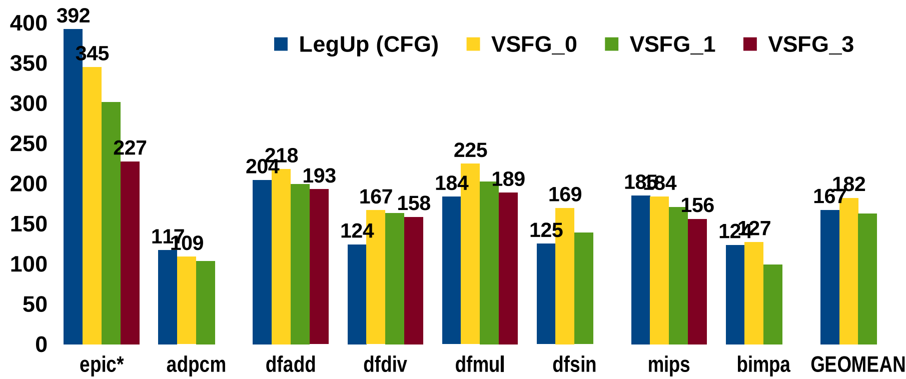


(g) dfmul (×1K cycles)

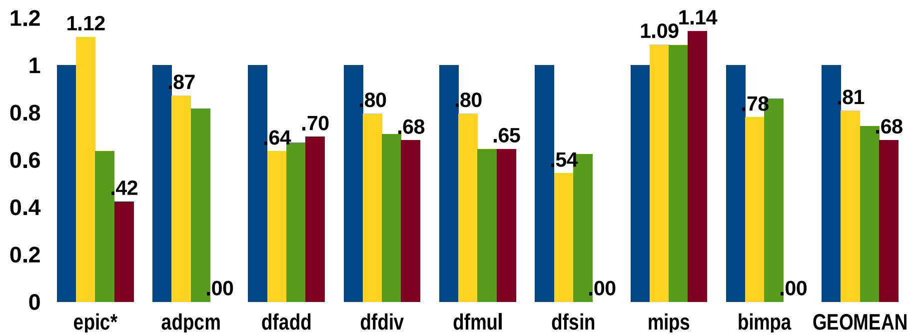


(h) mips (×1K cycles)

Figure 6.5: Performance Comparison (Cycle Count) vs an out-of-order Intel Nehalem Core i7 processor, and an Alteral Nios IIf in-order processor.



(a) f_{MAX} for benchmarks. Altera Nios II/f runs at 290 MHz.



(b) Normalized delay for benchmarks.

Figure 6.6: Frequency (f_{MAX}) and Delay comparison of LegUp and VSFG custom hardware. Note that some values for VSFG_3 are missing, as these circuits were too large to fit in our target FPGA. These have been removed from the Geomean calculation for VSFG_3.

As is common for existing high-level synthesis tools [SG08, CCA+11], incorporating static memory disambiguation based on alias-analysis is an essential first step towards exposing and exploiting more memory-level parallelism. Work by Budiu et al goes further, and incorporates a small load-store queue to perform dynamic memory disambiguation in their generated application-specific static dataflow hardware [BVCG04]. Note, however, that despite LegUp's utilization of static memory dismabiguation, the VSFG-based hardware performs better as LegUp-based hardware performance is primarily limited due to explicit control-flow in its implementation.

The *dfadd*, *dfdiv*, *dfmul*, and *mips* benchmarks perform poorly on the Core i7 primarily due to its high branch misprediction penalty – the i7 achieves only an 88% prediction accuracy on average for these benchmarks. Thanks to their many short, data-dependent branches, any speculation performance gains are quickly swamped by a 17 cycle penalty for each misprediction. As a result, both LegUp and VSFG perform better than the Core i7 for these benchmarks.

6.2.2 Frequency and Delay

Figure 6.6a shows the peak operating frequency (f_{MAX}) estimated for the circuits generated by both LegUp and the VSFG (predicated subgraphs version) by the Altera Quartus II (v. 13) tool after synthesis and place-and-route. For comparison, the Nios II/f achieves an f_{MAX} of 290MHz. The baseline VSFG_0 configuration achieves 15MHz higher fre-

quencies on average than LegUp. This was expected, as the dynamically scheduled VSFG does not have a centralized FSM, as discussed in Section 3.5. However, this advantage diminishes with the increase in circuit size that accompanies loop unrolling in the VSFG_1 and VSFG_3 configurations.

Unfortunately, despite the high degree of pipelining by both LegUp and in the VSFG, achievable f_{MAX} is inversely related to the size of the circuit being implemented. The primary reason for this is that all of the memory operations in spatial hardware are distributed across each generated circuit. Each such operation must access memory through a single, centralized memory controller. The combinational critical-path wire length thus increases with the total number of memory operations in the circuit. Thus as the degree of VSFG unrolling is increased, f_{MAX} decreases accordingly. The frequency values for the spatial hardware are thus often well below the achievable f_{MAX} for the carefully optimized Nios II/f design.

Figure 6.6b provides a comparison of the wall-clock execution time for all of the hardware configurations ($ExecutionTime = CycleCount \times 1/f_{MAX}$). On average, VSFG_0 and VSFG_3 demonstrate a 19% and 32% performance advantage over LegUp respectively. However, on a per-benchmark basis, it is important to note that the reduction in operating frequency can often overwhelm the ILP/cycle-count advantage provided by loop-unrolling, as is the case for the *mips* benchmark. This is another reason why static-memory disambiguation is an essential feature of modern HLS tools – it not only enables memory-level parallelism at runtime, but may also serve to partition physical memory, and hence the memory access network/arbitration tree, into smaller subsets. Instead of being connected to a single memory controller, memory operations may be divided amongst multiple controllers, one for each static partition, thereby simplifying the interconnect and improving f_{MAX}.

Other projects have optimized the memory access network in custom hardware for high frequency operation by either partitioning and distributing memory [HRRJ07] in this manner, or by pipelining the memory access network, while simultaneously optimizing for the most frequent accesses [VBCG06], or even by incorporating cache-like structures closer to each memory access operation [PBD+08, SVGH+11]. Thus in addition to incorporating memory disambiguation for parallelism, it is also necessary to consider the architecture and implementation of the centralized memory resource in spatial hardware. For the purpose of this dissertation, this effort is left for future work.

The LegUp tool instead deals with reduced f_{MAX} in a different manner. Instead of attempting to identify and pipeline the combinational critical path in a circuit, LegUp applies *operation-chaining*: reducing the cycle-count for hardware execution by removing registers between operations. This reduces the degree of pipelining in the circuit, but so long as f_{MAX} is not adversely affected, this translates into a direct improvement in execution time. A rudimentary version of operation-chaining was also incorporated into the VSFG HLS toolchain, and demonstrated similar improvements in cycle counts for a few of the benchmarks, without adversely affecting f_{MAX}. Unfortunately, my current implementation of operation-chaining is not yet robust enough to successfully compile for all benchmarks, thus those results are not provided here.

6.2.3 Resource Requirements

Recall from Section 5.3 that during the compilation from the VSFG IR to the Bluespec HDL, each VSFG transition is implemented as a set of one or more Bluespec *rules*, while each place (and its corresponding acknowledgement place) are implemented as a 1-place FIFO. Operations/transitions that have a fan-out of $n > 1$, will therefore implement n 1-place FIFOs, one for each fan-out value. In the actual hardware implementation, this can be quite wasteful, as the registers within each FIFO will store identical values. This complicates providing a fair comparison of the area, energy and power requirements between the circuits generated by LegUp, with those from the VSFG HLS toolchain.

A possible means of overcoming this is to store each output value in a separate register, and then utilize *void* type FIFOs to implement the Petri-net / static-dataflow style flow control. The *void* FIFOs do not incorporate registers to store any values, and only implement the state machine for performing the necessary flow control. Unfortunately, Bluespec currently remains an experimental compiler, and the VSFG HLS toolchain generates very large Bluespec modules for the selected benchmarks, often containing as many as 600-700 rules, thus pushing Bluespec to its limits. Implementing this optimization in my VSFG-to-Bluespec compilation pass caused the Bluespec compiler to fail during Verilog generation, despite the fact that no syntax or structural errors were reported. Of the eight selected benchmarks, only one was able to compile successfully to Verilog.

Thus, an alternative means of fairly evaluating area, energy and power for the VSFG-based hardware had to be devised: the area, energy and power metrics generated for the VSFG-hardware would have to be *scaled* by the estimated overhead incurred in the implementation due to the additional fanout FIFOs.

For FPGA implementations, the number of 'programmable logic blocks' or 'look-up tables' utilized is used as a measure of *area* or resource requirements for a given circuit. The structure of a typical Altera Stratix IV FPGA 'Adaptive Look-up Table' is shown in Figure 6.7. The hardware generated by the VSFG HLS toolchain will utilize a greater number of these ALUTs than equivalent hardware generated by the LegUp HLS tool, not because the VSFG requires any additional combinational logic, but because each instruction/operation/transition will require n sets of output registers instead of just 1 (where n is the fanout of a given instruction). Since there is only a single 1-bit register associated with each bit of output from the combinational logic, the VSFG must make use of $n-1$ additional ALUTs only for their registers, in order to implement the required fanout FIFOs. I attempt to account for this effect when presenting the area, power and energy results for the VSFG-based hardware, by computing a *scaling-factor* for each benchmark.

To generate this scaling factor, the LLVM-to-VSFG compilation pass was augmented to compute two new values for each benchmark. Assuming there are k instructions/operations in a benchmark:

- *Instruction-bits* is the total number of output values produced in the circuit by its constituent operations. It is the sum of the bit-width of the result of each operation in the circuit:

$$\textit{Instruction Bits} = \sum_{i=1}^{k} (\textit{bit-width})_i$$

- *Instruction Fanout-Bits* is the total number of 1-bit registers needed to store the

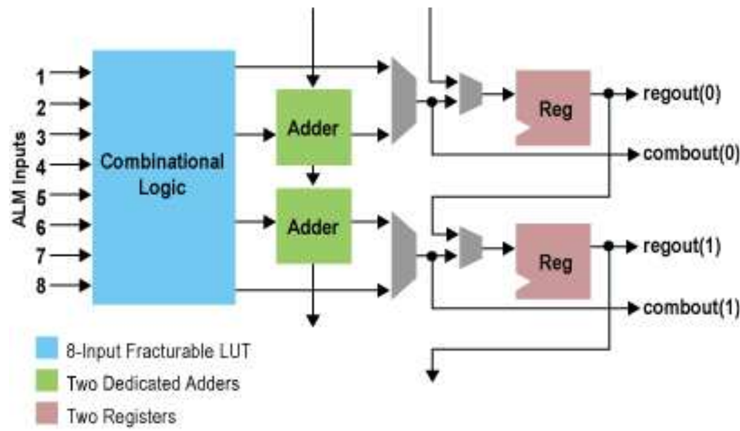


Figure 6.7: An Altera Stratix IV FPGA 'Adaptive Look-up Table' design.

instruction-bits, taking into account the fanout of each *instruction-bit*:

$$\textit{Instruction Fanout-bits} = \sum_{i=1}^{k} (\textit{bit-width})_i \times (\textit{fanout})_i$$

The scaling factor may then be computed as follows:

$$\textit{Scaling Factor} = \frac{\textit{Instruction Fanout-bits}}{\textit{Instruction Bits}}$$

This *Scaling Factor* represents the degree to which the fanout of all instructions in a VSFG-based hardware circuit will increase its ALUT utilization on an FPGA. Table 6.5 shows the *Instruction bits*, the *Instruction Fanout-bits*, and the *Scaling Factor* for all the benchmarks, while Table 6.6 shows the resource requirement for 7 of our 8 benchmarks in ALUTs before and after applying the scaling factor to the ALUT count of each VSFG implementation[7]. The VSFG area results are for the circuits implemented with predicated-subgraphs, since these exhibited the best average cycle-count results.

Table 6.5: Table showing the *Instruction bits*, the *Instruction Fanout-bits*, and the *Scaling Factor* for all benchmarks.

	Instruction-bits	Instruction Fanout-bits	Scaling Factor
epic	590	1069	1.81
adpcm	40767	55400	1.36
dfadd	15673	28040	1.79
dfdiv	16337	26759	1.64
dfmul	10320	16640	1.61
dfsin	38759	67376	1.74
mips	15134	26790	1.77
bimpa	14488	20007	1.38

Figure 6.8 shows the scaled results from Table 6.6 normalized to the LegUp circuit sizes. As can be seen, the VSFG_0 configuration has a similar resource requirement to

[7]Results for *bimpa* are not presented as it was too large to fit in the largest available FPGA, so its resource requirements could not be estimated. The VSFG_3 results for the *adpcm* and *dfsin* benchmarks are unavailable for the same reason.

Table 6.6: Table showing the resource requirements (in number of ALUTs) of the generated custom hardware from both the LegUp and VSFG HLS toolchains. Area is shown for the best performing version of the VSFG hardware (i.e. predicated subgraphs).

		Raw ALUT Counts				Scaled ALUT Counts		
	LegUp	VSFG_0	VSFG_1	VSFG_3	Sc. Factor	VSFG_0	VSFG_1	VSFG_3
epic	764	1569	4932	16575	1.81	866	2722	9148
adpcm	17214	44268	81087		1.36	32575	59669	
dfadd	6382	10098	19217	38048	1.79	5644	10741	21267
dfdiv	9843	14545	29849	68085	1.64	8880	18224	41567
dfmul	3088	5382	9556	18162	1.61	3338	5927	11264
dfsin	18805	36347	128487		1.74	20909	73914	
mips	3533	7993	14299	27000	1.77	4515	8078	15253

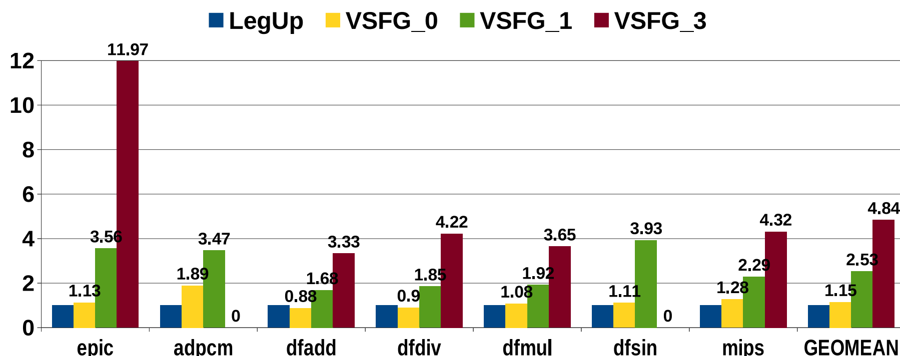


Figure 6.8: Resource requirements comparison for VSFG-based hardware compared to LegUp hardware. VSFG results have been *scaled* to remove the effects of extra fanout FIFOs.

the LegUp baseline, incurring a 15% penalty on average. The key reason for this is that the VSFG must implement additional hardware operations in order to perform control-directed dataflow and hierarchical subgraph communication in a dynamically scheduled environment – *MUX signal*, *wait*, *eta*, *mu*, *inGate*, and *outGate* operations must be implemented as part of the dynamically scheduled VSFG hardware. Additional resources are also required to implement the boolean predicate expressions, as well as the state-token being passed between state operations.

The statically-scheduled LegUp hardware also implements control-directed dataflow operations, equivalent to our *MUX*, *eta*, and *mu*, however these are only implemented as combinational logic, not requiring their own output registers, and thus also not always requiring their own ALUTs in the FPGA. Conversely, LegUp hardware must implement its static-schedule as a centralized, sometimes large and complex FSM that consumes additional area, whereas the static-dataflow VSFG hardware implements very simple, highly decentralized FSMs.

As the degree of unrolling is increased for the VSFG, its resource requirements grow dramatically – the VSFG_3 configuration requires an average of almost 5× more resources than the baseline LegUp implementation with no unrolling. The *epic* benchmark requres almost 12× more resources than the LegUp baseline, as it effectively implements 16 copies of the inner-loop, containing the resource intensive modulus operation.

In addition to the reduction in operating frequency that is experienced by larger circuits as discussed earlier, managing resource utilization is another reason why unrolling

of loops must be implemented more judiciously, perhaps by relying on execution profiling of code, in order to identify which loops would benefit from unrolling, and to what degree. As mentioned earlier, *epic* exhibits no inner-loop parallelism, thus only the outer-loop ought to be unrolled, which would have improved both resource requirements, and f_{MAX}, as well as reduced power dissipation, and improved energy efficiency, as discussed below.

6.2.4 Power and Energy

Power Dissipation:

Post-synthesis power estimation for each of the hardware circuits considered was performed using the PowerPlay Power Analysis tool built into the Altera Quartus II ver.13 compiler. In order to achieve high accuracy, Value Change Dump (.vcd) files were generated for all nets within each circuit evaluated from post-synthesis functional simulation using ModelSim. These '.vcd' files were provided as input to the PowerPlay Power Analysis tool. According to the Altera Quartus II Development Sotware Handbook, these results are expected to be accurate to within ±20% of a physical FPGA implementation of a circuit [Alt13].

All power results are given in miliwatts (mW). Power values are generated assuming a standard 250MHz operating frequency for all circuits, as well as the Nios II/f, and the extrapolated Core i7. Given prior work on pipelining and/or partitioning the memory arbitration tree for high frequency operations [HRRJ07, VBCG06, PBD+08, SVGH+11], this not an unreasonable assumption, and is helpful in simplifying our power and energy comparison. Reported power results have two main components: static and dynamic power dissipation. Equations 6.1 and 6.2 present the general formulae for static and dynamic power, respectively. In the equations, V_{DD} is the supply voltage, $I_{leakage}$ is the leakage current, k_{design} is a MOS fabrication technology dependent constant, f is the operating frequency, C is the average loading capacitance per transistor/resource, n is the total number of transistors/resources, and α is the average activity ratio for all transistors.

$$P_{static} = n.V_{DD}.I_{leakage}.k_{design} \tag{6.1}$$

$$P_{dynamic} = n.\alpha.C.V_{DD}^2.f \tag{6.2}$$

For our FPGA-based evaluation, V_{DD}, $I_{leakage}$, and k_{design} can be assumed to be common across all tested circuits. Therefore, static power is proportional to n, i.e. the area or ALUT resource requirements of a circuit. Dynamic power is primarily due to logic transitions, signal transmission and switching activity, thus is primarily dependent upon C, α, f, and n. PowerPlay presents dynamic power as composed of two components: (1) combinational logic, and (2) clock power. Dynamic clock power accounts for power dissipated by the FPGA clock tree, as well as all switching activity at the datapath registers in a given circuit.

Table 6.7 presents the raw static and dynamic power results reported for the circuits generated by LegUp[8]. The constituents of dynamic power are shown along with their total. As can be seen, in such deeply pipelined spatial hardware, clock-tree and register power is significantly greater than the power dissipated by the combinational LUT logic.

[8]Note that power results are not presented for *bimpa*, as this benchmark could not be synthesized successfully as it did not fit in the largest available Stratix IV FPGA.

Table 6.7: Table showing Raw Static and Dynamic Power Results (mW) for LegUp

	Raw Dynamic Power			Static Power				
	Clock	Comb.	**Total**	Raw	Used ALUTs	Total ALUTs	**Actual**	**Total**
epic	33.2	1.2	34.4	813.59	764	182400	3.41	37.81
adpcm	1166.37	146.16	1312.53	880.77	17214	182400	83.12	1395.65
dfadd	271.23	2.74	273.97	830.29	6382	182400	29.05	303.02
dfdiv	454.7	3.26	457.96	844.3	9843	182400	45.56	503.52
dfmul	158.62	6.15	164.77	822.46	3088	182400	13.92	178.69
dfsin	586.3	13.75	600.05	873.95	18805	182400	90.10	690.15
mips	290.67	1.55	292.22	826.73	3533	182400	16.01	308.23

For comparison, the Nios II/f processor exhibits a total power dissipation of 738.17 mW, of which 725.53 mW is dynamic power, and 12.63 mW is the actual static power.

PowerPlay reports the static power dissipated by the entire FPGA, thus actual static power utilized by a circuit is determined by scaling this value by the actual proportion of the FPGA being utilized by the circuit. The actual static power is thus calculated as follows:

$$Actual\ Static\ Power = Raw\ Static\ Power \times \frac{Used\ ALUTs}{Total\ ALUTs}$$

The total power reported in the last column is the sum of total dynamic and actual static power values.

Tables 6.8, 6.9, and 6.10 present the power values for the VSFG_0, VSFG_1 and VSFG_3 (with predicated subgraphs) respectively[9]. As with LegUp, combinational dynamic power forms only a small fraction of the total dynamic power. In addition to scaling static power by the circuit size, we must also adjust it by using the scaling factors provided in Table 6.5, in order to account for the overhead in VSFG circuits for the extra fanout FIFOs. Since static power is proportional to the amount of resources utilized n, it can be scaled as follows:

$$Scaled\ VSFG\ Static\ Power = \frac{Raw\ VSFG\ Static\ Power}{Scaling\ Factor} \times \frac{Used\ ALUTs}{Total\ ALUTs}$$

For dynamic power, I assume that the amount of combinational logic is equivalent between the VSFG and LegUp-based hardware, and thus the combinational component of dynamic power need not be scaled. However, the clock component of dynamic power must be scaled to remove the effects of the extra fanout FIFOs. For simplicity, I assume that C, α, f would remain unchanged in VSFG circuits that are implemented without the extra fanout FIFOs[10]. Thus dynamic clock power can also be scaled proportionally to the number of registers in the circuit. Scaled dynamic power is therefore given as:

$$Scaled\ VSFG\ Dynamic\ Power = \frac{Raw\ VSFG\ Clk\ Power}{Scaling\ Factor} + Raw\ VSFG\ Comb.\ Power$$

[9] As with LegUp, results are not presented for *bimpa*. In addition, VSFG_3 results for *adpcm* and *dfsin* are also not presented, as these circuits were also too large to be successfully synthesized to the available Stratix IV FPGAs.

[10] Note that the average loading capacitance C would also be reduced with the reduced number of fanout FIFOs, as well as due to the total circuit size being reduced leading to shorter transmission distances. However, I ignore this improvement for simplicity.

Table 6.8: Table showing Static and Dynamic Power Results for VSFG_0 (mW)

	Raw Dynamic Power			**Scaled Dynamic Power**		**Static Power**				
	Clock	Comb.	Sc. Factor	Scaled Clk.	**Total**	Raw	Used ALUTs	Total ALUTs	**Scaled**	**Total**
epic	74.99		1.81	41.39	41.39	814.71	1569	182400	3.87	45.26
adpcm	2531.17	31.61	1.36	1862.60	1894.21	968.15	44268	182400	172.90	2067.12
dfadd	2057.31	4.18	1.79	1149.94	1154.12	879.38	10098	182400	27.21	1181.33
dfdiv	570.27	9.13	1.64	348.16	357.29	850.63	14545	182400	41.41	398.71
dfmul	764.07	23.8	1.61	473.87	497.67	838.95	5382	182400	15.35	513.02
dfsin	2679.5	54.78	1.74	1541.42	1596.20	951.45	36347	182400	109.07	1705.27
mips	1017.84	0.08	1.77	574.99	575.07	850.28	7993	182400	21.05	596.12

Table 6.9: Table showing Static and Dynamic Power Results for VSFG_1 (mw)

	Raw Dynamic Power			**Scaled Dynamic Power**		**Static Power**				
	Clock	Comb.	Sc. Factor	Scaled Clk.	**Total**	Raw	Used ALUTs	Total ALUTs	**Scaled**	**Total**
epic	178.98	5.33	1.81	98.78	104.11	817.54	4932	182400	12.20	116.31
adpcm	6117.88	36.84	1.36	4501.94	4538.78	1176.55	81087	182400	384.89	4923.67
dfadd	4007.14	17.63	1.79	2239.80	2257.43	953.32	19217	182400	56.14	2313.57
dfdiv	1240.34	22.29	1.64	757.26	779.55	899.45	29849	182400	89.86	869.41
dfmul	1576.3	63.54	1.61	977.61	1041.15	869.5	9556	182400	28.25	1069.40
dfsin	4157.85	63.89	1.74	2391.86	2455.75	1110.81	128487	182400	450.13	2905.89
mips	1394.74	0.24	1.77	787.91	788.15	868.37	14299	182400	38.46	826.60

Table 6.10: Table showing Static and Dynamic Power Results for VSFG_3 (mw)

	Raw Dynamic Power			**Scaled Dynamic Power**		**Static Power**				
	Clock	Comb.	Sc. Factor	Scaled Clk.	**Total**	Raw	Used ALUTs	Total ALUTs	**Scaled**	**Total**
epic	688.31	11.7	1.81	379.89	391.59	844.56	16575	182400	42.36	433.95
dfadd	4585.13	23.44	1.79	2562.87	2586.31	1000.93	38048	182400	116.70	2703.01
dfdiv	2500.87	47.01	1.64	1526.84	1573.85	1012.24	68085	182400	230.68	1804.53
dfmul	2745.6	121.15	1.61	1702.80	1823.95	918.18	18162	182400	56.70	1880.65
mips	2251.99	0.41	1.77	1272.18	1272.59	911.45	27000	182400	76.22	1348.80

Figure 6.9a shows a comparison of the total power values of LegUp from Table 6.7, with the scaled total power values for VSFG circuits from Tables 6.8, 6.9, and 6.10, normalized to LegUp. In order to understand the impact of aggressive speculative execution on power dissipation in VSFG-based spatial hardware, I instrumented both the VSFG HLS toolchain, and the generated VSFG_0 hardware to estimate the amount of circuit activity overhead due to speculative execution in the baseline VSFG.

This was accomplished in two steps:

1. First, in the LLVM-to-VSFG compiler pass, I associate a *true activity count* and a *false activity count* with each basic block predicate expression in the VSFG. This is done in a similar manner to the instrumentation performed in Section 6.2.3. The *true activity count* represents the total number of instruction-bits that might switch each time the basic-block predicate holds, while the *false activity count* represents the total number of instruction-bits that might toggle each time the predicate is false, i.e. the activity of all the predicate promoted operations within the original basic block that are speculatively executed.

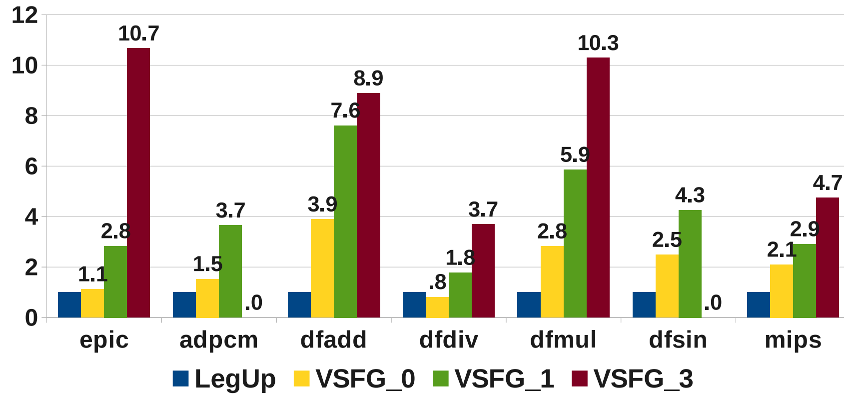
2. Next, in the VSFG-to-Bluespec generation phase, two additional accumulators are associated with each predicate expression in the generated hardware, labelled *valid activity bits*, and *mis-speculated activity bits*. Each time a predicate expression holds in the circuit, it adds its *true activity count* value to its *valid activity bits* register. Conversely, each time a predicate expression is false, it adds its *false activity count* value to its *mis-speculated activity bits* register. At the end of benchmark execution, all *valid* and *misspeculated activity bits* registers are accumulated into two global *valid* and *misspeculated activity* registers.

These values provide a measure of the degree of misspeculation overhead incurred by the VSFG_0 hardware. Table 6.11 shows the measured useful and misspeculated activity for 6 of the 8 benchmarks[11]. The last column shows what fraction of total activity was *useful*, i.e. not misspeculated. As can be seen, some benchmarks like *adpcm* and *epic* exhibit low speculation overhead, while for the remainder, a large fraction of the activity is due to mis-speculated execution. Figure 6.9b presents the misspeculated activity overheads graphically, normalized to the useful activity (green region) in each benchmark.

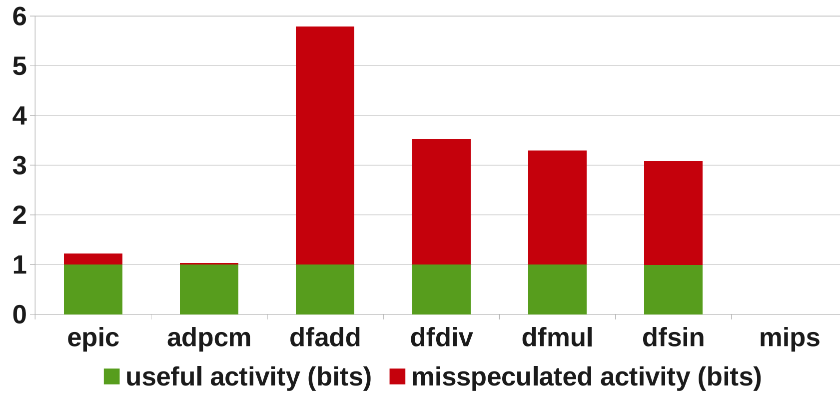
Table 6.11: Table showing Useful, Misspeculated, and Total activity in *Instruction-bits* for the VSFG_0

	Activity in *instruction-bits*			
	Useful	Mis-speculated	Total	Useful / Total
epic	7478389	1617588	9095977	0.82
adpcm	3504294	100992	3605286	0.97
dfadd	122651	586432	709083	0.17
dfdiv	93578	236223	329801	0.28
dfmul	59796	137056	196852	0.30
dfsin	3116600	6493479	9610079	0.32

[11]Unfortunately, the Bluespec compiler consistently crashed when *mips* was compiled after adding the mentioned hardware instrumentation.



(a) Power dissipation comparison of VSFG, normalized to LegUp.



(b) Activity overheads due to aggressive speculation in the VSFG_0.

Figure 6.9: Estimated Power Dissipation Comparison vs LegUp. The overheads for the VSFG_0 are proportional to the increase in activity due to aggressive speculation.

Comparing Figures 6.9b and 6.9a, we see that the VSFG_0 mostly exhibits a higher power dissipation than LegUp, echoing the pattern for speculation overhead in Figure 6.9b - *dfadd* for instance has both the highest speculation overhead and the highest normalized power dissipation for VSFG_0, while both *epic* and *adpcm* have much lower of both overheads. This again emphasizes the need to carefully balance between sub-graph predication and speculation. Existing work on profiling-driven hyperblock formation [MLC+92], could be adapted in the future for implementing careful subgraph extraction and/or refactoring into the VSFG HLS toolchain. However, for now, the exploration of how far the the energy overheads of speculation can be mitigated without compromising performance is left for future work.

As we increase the degree of unrolling in the VSFG_1 and VSFG_3 configurations, power dissipation increases significantly, despite the fact that circuit activity per unit area should decrease. The additional increase in power dissipation for VSFG_1 and VSFG_3 is driven primarily by non-computational overheads: dynamic power increases because clocking overhead grows proportionally to the registers in the circuit, while static power increases in proportion to the increased circuit size.

These results highlight the need to apply effective clock and power gating to the various subgraphs when implementing the VSFG in custom hardware. One extreme solution is the utilization of asynchronous logic by Budiu et al for their spatial computation im-

plementations [BVCG04], allowing for drastic reduction of dynamic power dissipation, since there is no need for a global clock tree driving all registers. Additionally, the degree of loop unrolling should be carefully balanced against the power overheads that may be incurred in addition to the area cost.

Energy Cost:

VSFG-based hardware is able to achieve high sequential performance, approaching that of a simulated Intel Nehalem Core i7 processor, as described in Section 6.2.1. For this advantage, the unrolled VSFG_1 and VSFG_3 implementations exhibit anywhere from $2-10\times$ higher power dissipation than the baseline LegUp hardware. In order to evaluate the energy cost of the VSFG-based hardware, I combine the cycle-counts for the best performing VSFG configuration from Section 6.2.1 (predicated subgraphs), with the power measurements at 250MHz presented in Tables 6.8, 6.9, and 6.10.

I also combine the cycle counts for Nios with its power dissipation of 738.17 mW at 250 MHz to estimate its energy dissipation. Finally, I estimate the energy cost of a hypothetical version of the superscalar Core i7 processor that can be implemented on an FPGA, also running at 250MHz. This is done by referring to the empirical relationship between power and performance for sequential processors presented by Grochowski et al. [GA06]:

$$Power = Perf^{\alpha} \qquad \text{where } \alpha = 1.75$$

Table 6.12: Table showing average speedup of Core i7 over Nios II/f

	Nios II/f	Core i7	Speedup
epic	3399634	200174	16.98
adpcm	119794	42662	2.81
dfadd	16441	15994	1.03
dfdiv	36487	15120	2.41
dfmul	7074	14072	0.50
dfsin	1420558	104953	13.54
mips	31082	29998	1.04
bimpa	373347552	39664956	9.41
		GEOMEAN	3.07
		BEST GEOMEAN	8.83

Table 6.12 shows the cycle counts of the Nios II/f and the Core i7 processor taken from Table 6.1, and presents their ratio as the speedup over Nios provided by the Core i7 for each benchmark. Computing the geometric mean of speedup for all benchmarks, we find that the Core i7 is on average $3.07\times$ faster than the Nios processor at the same clock frequency[12]. However, as noted earlier, the *dfadd*, *dfdiv*, *dfmul*, and *mips* benchmarks represent pathological cases for the Core i7 due to their data-dependent, unpredictable branching. Thus a more realistic estimate of average speedup excluding these

[12]Given that we're assuming that this hypothetical Core i7 would be implemented on an FPGA, an order of magnitude reduction in f_{MAX} from 2.66 GHz to 250 MHz is a reasonable ball-park estimation.

four benchmarks, and considering only the remainder is found to be 8.83× over the Nios. Nevertheless, by applying the rule above for the more conservative speedup value:

$$PowerFactor_{3\times} = 3.07^{1.75} = 7.12$$

Thus, power for our hypothetical superscalar processor is computed as:

$$Power_{3\times} = 7.12 \times 738.17mW = 5255.77mW$$

Combining the cycle count and power measurements, Figure 6.10 presents an energy cost comparison between LegUp, each VSFG version, as well as pure software implemented on both the Altera Nios and the hypothetical out-of-order processor. All values are normalized to the energy of the LegUp-based hardware. The mean energy cost for the 35% higher average performance of the VSFG_1 and VSFG_3 configurations is between $3 - 4\times$ over LegUp. However, it is also about $0.25 - 0.3\times$ the energy cost of the in-order Nios II/f processor, and about 6× more energy efficient on average than the out-of-order processor.

Thus, despite our blind unrolling of all loops, implementing no optimizations towards minimizing energy, and no exploitation of memory-level parallelism, the VSFG-based hardware is able to approach levels of parallelism achievable in a superscalar processor, while incurring only a fraction of the energy cost of even an in-order conventional processor. I expect further improvements in both performance and energy efficiency to be available once we incorporate (a) alias-analysis to parallelize memory access, (b) effectively balance speculation with predication.

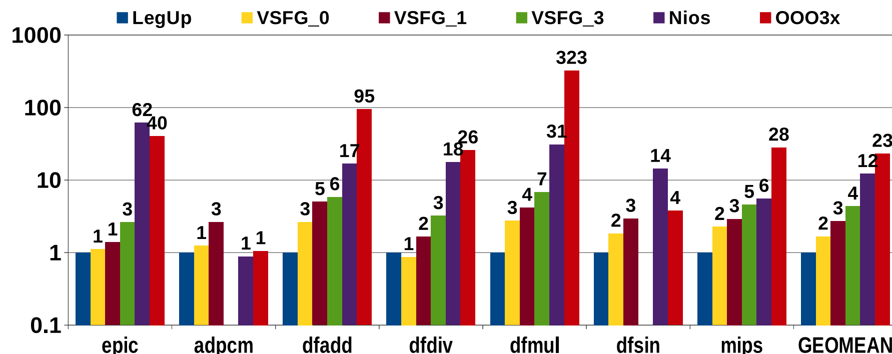


Figure 6.10: Energy Consumption comparison: VSFG vs LegUp vs Altera Nios II/f vs a hypothetical out-of-order processor based on the Core i7.

6.3 Estimating ILP

In order to more clearly illustrate the potential advantage of the VSFG in exposing instruction level parallelism, this section presents an estimation of the average instructions executed per cycle (IPC) by each of the implementations.

Due to the difficulty of making a fair comparison of IPC across the varying instruction sets (LLVM for LegUp and VSFG, x86 for the Core i7 and Altera's proprietary ISA for the Nios processor), I have chosen to instead approximate a measure of IPC by dividing the cycle-count results obtained for each type of processing element in Section 6.2.1 with the number of *useful*[13] LLVM instructions that are expected to execute within each

[13]i.e. excluding the misspeculatively executed instructions

benchmark. A similar form of instrumentation to that described in Section 6.2.4 (used to generate Figure 6.9b) was used to count the number of useful instructions executed by each benchmark.

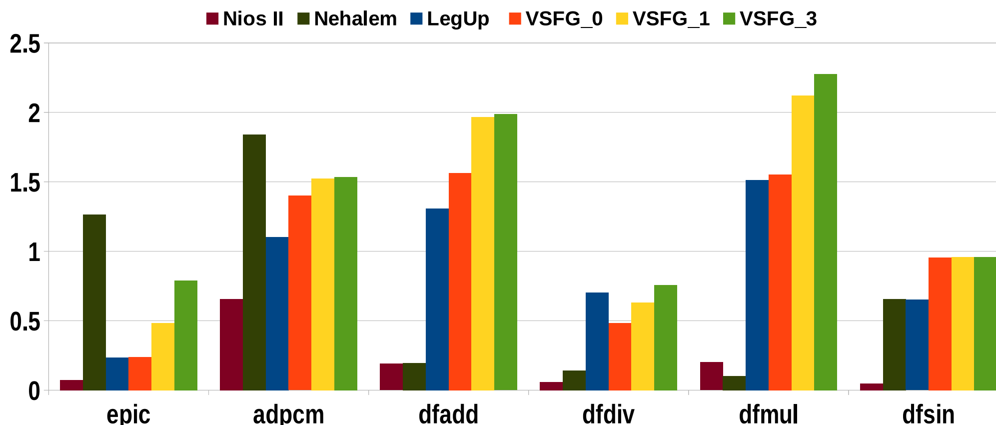


Figure 6.11: Estimated Instructions Per Cycle achieved by LegUp and VSFG (predicated subgraphs) implementations, as well as the Nios II/f and Core i7 processors.

The estimated IPC results are presented in Figure 6.11. Unfortunately these results could only be generated for six of the eight benchmarks, due to the toolchain's aforementioned fragility. As can be seen, increased loop unrolling typically allows the VSFG-based implementations to achieve better ILP than LegUp generated hardware, and far better than the in-order Nios. For the *epic* and *adpcm* benchmarks, the Core i7 still retains an advantage over the best VSFG version due to the single state-edge constraint on parallelism in the latter, while all versions of the VSFG exhibit better IPC than both processors for *dfsin*.

6.4 Summary

This chapter presented a performance, area, power and energy comparison of hardware generated from the VSFG-based high-level synthesis toolchain against hardware generated using LegUp. In addition, a performance and energy comparison was presented against two types of conventional processors: a simple, in-order Altera Nios II/f soft-processor, and a simulated hypothetical out-of-order soft-processor based on the Intel Nehalem Core i7.

The VSFG-based hardware demonstrated both higher cycle-counts and operating frequency than hardware generated by LegUp, a state-of-the-art academic high-level synthesis tool. Furthermore, with multi-level loop unrolling, the static-dataflow VSFG hardware is able to match, if not exceed the cycle-count performance of the Intel Nehalem Core i7 processor on control-intensive sequential code, except in cases of memory intensive code, where the VSFG performance is limited due to the strict sequentialization it currently imposes on all memory operations.

For this performance advantage, the best performing VSFG-based hardware incurs an average energy dissipation cost $3-4\times$ greater than LegUp-generated hardware, while the baseline VSFG_0 version incurs an average $2\times$ greater energy cost. However, I consider this a worst-case cost for the selected benchmarks, as no optimizations have yet been

applied to the VSFG to improve energy efficiency, nor have any memory disambiguation mechanisms been incorporated to improve memory-level parallelism. For future work on utilizing the VSFG to compile to spatial architectures, the following optimizations are considered essential:

- Efficiency would be improved by carefully balancing between speculation and predicated execution in the VSFG. One could start by adapting prior work on profile-directed hyperblock formation [MLC+92], and apply it to re-factor VSFG subgraphs and hierarchy, and then selectively applying predicate-promotion on the most frequently executing subgraphs.
- Utilizing profiling to identify and selectively unroll loops would help improve both area and energy efficiency.
- Incorporating static memory disambiguation is essential for improving performance scaling of the VSFG, particularly for memory intensive benchmarks. Performance improved in this way would also translate to lower energy requirements, as no additional power dissipation overhead is incurred from static-memory disambiguation.
- Incorporating dynamic memory disambiguation should also be explored, as done previously by Budiu et al [Bud03, BVCG04]. However, the increased power dissipation due to dynamic memory disambiguation would need to be balanced against the performance (and thus indirectly energy) improvements that it may provide.

Nevertheless, despite the lack of the aforementioned optimizations, these results demonstrate the ILP exposing potential of the VSFG intermediate representation. Through its aggressive support for speculative execution, control-dependence analysis, and exploitation of multiple-flows of control, the VSFG IR has been demonstrated to be a strong candidate for addressing the amenability issue with spatial computation. ILP is exposed statically within the representation, from sequential, control-intensive code, meaning that there is less need to incur the energy cost of dynamic mechanisms for overcoming control flow, as with earlier work on spatial architectures such as TRIPS [SNL+04], and DySER [GHN+12, GHS11].

Chapter 7 now concludes this dissertation, as well as discussing potential directions for future work based on the VSFG IR. To mitigate the effects of dark silicon, of particular interest would be the development of an energy efficient spatial architecture capable of not only accelerating numeric applications, but also matching, if not exceeding superscalar performance on sequential applications, while retaining its orders-of-magnitude energy efficiency advantage.

CHAPTER 7

Conclusions and Future Work

Computer architects are facing several critical issues that must be addressed in order to continue scaling performance with Moore's Law. In particular, the end of Dennardian scaling has led to the Utilization Wall: for a fixed power budget, with each technology generation, only an ever diminishing proportion of on-chip resources can be activated at any time. Also, due to the difficulty of parallelizing applications for shared-memory multicore processors, Amdahl's law further limits the speedups achievable in the manycore era.

Together, the Utilization Wall and Amdahl's Law result in the problem of Dark Silicon [EBSA+11], which constrains performance scaling far below the promise of Moore's Law. To mitigate the effects of the Utilization Wall, architects are increasingly looking towards spatial computation (application-specific custom hardware in particular), due to its potential for orders-of-magnitude improvements in energy efficiency over conventional processors. Unfortunately, while it achieves very high performance on numeric and multimedia style applications, spatial computation exhibits poor performance on sequential, control-intensive code relative to conventional processors. This means that despite the high efficiency, Amdahl's law still restricts achievable speed-ups.

In this dissertation, my goal has been to improve the performance of sequential, control-intensive code when implemented as spatial computation. Furthermore, this was to be achieved without overly compromising the inherent advantages of spatial computation, namely computational density, scalablilty, architectural simplicity, and of course high energy efficiency. Noting that control-flow was the primary constraint on achieving higher sequential performance [LW92], I devised a new compiler IR for spatial computation based on the Value State Dependence Graph [Law07, Joh04], as well as presenting eager, dataflow execution semantics for it. This Value State *Flow* Graph elides most control flow from the original imperative program representation, instead emphasizing only the true dataflow and state-ordering dependencies that must be respected for correct program execution. The VSFG also enables aggressive control-dependence analysis, and is capable of exposing and exploiting multiple flows of control.

As a case study for evaluating this new representation, I developed a high-level synthesis tool that compiles imperative code to the VSFG, then implements it as static-dataflow custom hardware. Hardware generated by this toolchain was compared with an existing, established academic high-level synthesis tool called LegUp, as well as with a simple in-order Altera Nios II/f soft processor and a simulated Intel Nehalem Core i7 processor. **The results show that the VSFG based hardware achieves a performance im-**

provement of as much as 35% over LegUp, at a 3× higher average energy cost. Performance (in cycle-counts) is comparable to that of the Core i7, while incurring only a fraction of the energy cost (1/4× the energy cost of an in-order Altera Nios II/f soft-processor, and 1/8× the energy cost of an extrapolated Core i7-like soft-processor).

7.1 Future Work

7.1.1 Incremental Enhancements

These results for the VSFG-based hardware are particularly promising given that they are achieved without any significant optimizations to the VSFG IR and hardware that could further improve both efficiency and performance. Listed below are some of the enhancements and optimizations suggested in this dissertation to further improve performance, energy-efficiency, and/or coverage of imperative language features:

- Hybridization of the VSFG-S IR (and associated toolchain) with the dynamic-dataflow execution model, to enable support for true function calls and general recursion.
- Incorporation of alias analysis based static memory disambiguation within the LLVM-to-VSFG compiler, in order to expose memory level parallelism at compile-time, thereby improving run-time performance as well as energy-efficiency.
- Incorporation of dynamic memory disambiguation hardware in the memory infrastructure to expose further memory level parallelism at run-time. This would further improve performance, though its energy impact must be evaluated.
- Optimization of memory-access network in hardware. Pipelining the memory access network, as well as the introduction of local *cachelets* closer to memory operations would also improve both performance and energy efficiency [HRRJ07, VBCG06, PBD+08, SVGH+11].

7.1.2 Mitigating the Effects of Dark Silicon

Spatial computation architectures such as Coarse-Grained Reconfigurable Arrays, and Massively Parallel Processor Arrays exhibit significant advantages over conventional processors, as they are far more scalable, energy-efficient, have higher computational density and lower design and verification costs. With worsening wire scaling [HMMH01], and the dark silicon problem, spatial architectures represent a potentially revolutionary improvement in computer architecture.

By developing the VSFG-S compiler IR specifically with the goal of implementing conventional, imperative programming languages onto spatial architectures, while achieving high sequential performance, I have demonstrated that two of the primary hurdles to the utilization of such spatial architectures – amenability and programmability – can now be resolved. I hope that the work presented in this dissertation will facilitate more pervasive, even ubiquitous utilization of such architectures, particularly to mitigate the effects of Dark Silicon. Several important future research directions are described below to further this end:

- Develop an efficient Coarse Grained Reconfigurable Array to directly implement VSFG-S static dataflow graphs. Ideally, the objective of this research would be to develop a complete replacement for conventional processors – similar to the Wavescalar [SSM+07] or TFlex [KSG+07] approaches – instead of implementing yet another hybrid architecture like Conservation Cores [VSG+10], DySER [GHS11] or Tartan [MCC+06], that relies on an attached conventional processor to implement part of the functionality. The recently developed *Triggered Instructions* architecture adopts a guarded atomic actions style execution model that is very similar to our static-dataflow approach, and may prove a good place to start [PPA+13].

- In addition to developing a new CGRA, the extensive existing research into the development of Massively Parallel Processor Arrays could benefit from the development of compilation back-ends via the VSFG. This would potentially allow the implementation of imperative code distributed onto such fine-grained message-passing many-cores while still achieving high sequential performance. Examples of such architectures include Loki [Bat14], Mamba [Cha12], and the MIT RAW [TLM+04].

  Existing work on compiling dataflow languages to coarse-grained Von Neumann / Dataflow hybrid architectures would be good places to seek inspiration on compiling to MPPAs [YAMJGE13, Ian88, GTK+02]. Notable work in this area utilizes coarse-grained dataflow models like Kahn Process Networks [LP02].

- Conventional high level synthesis tools rely on static scheduling to identify at compile time when any given operation should execute. Given this schedule, subsequent *allocation* and *binding* phases in the tool are used to implement area and energy optimizations by *reusing* hardware resources like adders, multipliers, registers etc. The VSFG-based HLS toolchain described in Chapter 5 does not presently consider the issues of allocation and binding, primarily because this work would have been outside the scope of this research.

  Nevertheless, consideration of how allocation and binding can be incorporated into a high-level synthesis tool that relies on dynamically scheduled execution merits further study, given the demonstrated performance advantages of such hardware.

- In addition to homogeneous spatial architectures like CGRAs, MPPAs, the structure and semantics of the VSFG are well suited for describing computation using heterogeneous components. Given that the VSFG is hierarchical, and that nested-subgraphs are isolated from each other, each nested-subgraph may be implemented on a completely different substrate from every other. A parent graph may execute on a CGRA, while its nested subgraph is implemented on an embedded processor, so long as the dataflow style communication semantics between the two subgraphs are maintained.

  Further pursuing this line of thinking might provide insightful new solutions to the design specification, implementation, and utilization of modern heterogeneous system-on-chip architectures.

- Currently the VSFG is tailored for the implementation of sequential languages. However, given the increasing and urgent shift towards parallel programming models and languages, an important topic for research would be to see if the VSFG

could be extended to express and implement such concurrent languages. For instance, message passing could be implemented by utilizing the existing *signal* and *wait* operations, while non-determinism could be incorporated by utilizing the *mu* operation beyond loop entry points, and independently of the *inGate* operation.

Furthermore, the key function of many shared memory parallel programming models like OpenMP and OpenCL is to allow the programmer to specify concurrency of operations. This explicit concurrency would manifest itself as a further partitioning of the VSFG state-edge beyond that already implemented by static-memory disambiguation at the compiler-level. Further research is warranted to assess if all features of such parallel programming models can be completely supported by extending the VSFG IR without compromising any of its inherent advantages.

The utility and impact of the VSFG-S IR can be extended dramatically if it can serve as a compilation target for multiple concurrent programming models, while also providing target implementation back-ends for custom hardware, CGRAs, MPPAs or even heterogeneous architectures.

Bibliography

[AAB+12] C. Auth, C. Allen, A. Blattner, D. Bergstrom, M. Brazier, M. Bost, M. Buehler, V. Chikarmane, T. Ghani, T. Glassman, R. Grover, W. Han, D. Hanken, M. Hattendorf, P. Hentges, R. Heussner, J. Hicks, D. Ingerly, P. Jain, S. Jaloviar, R. James, D. Jones, J. Jopling, S. Joshi, C. Kenyon, H. Liu, R. McFadden, B. Mcintyre, J. Neirynck, C. Parker, L. Pipes, I. Post, S. Pradhan, M. Prince, S. Ramey, T. Reynolds, J. Roesler, J. Sandford, J. Seiple, P. Smith, C. Thomas, D. Towner, T. Troeger, C. Weber, P. Yashar, K. Zawadzki, and K. Mistry. A 22nm High Performance and Low-Power CMOS Technology Featuring Fully-Depleted Tri-gate Transistors, Self-Aligned Contacts and High Density MIM Capacitors. In *VLSI Technology (VLSIT), 2012 Symposium on*, pages 131–132, June 2012.

[AC86] Arvind and David E. Culler. Dataflow Architectures. In Joseph F. Traub, Barbara J. Grosz, Butler W. Lampson, and Nils J. Nilsson, editors, *Annual Review of Computer Science Vol. 1, 1986*, pages 225–253. Annual Reviews Inc., Palo Alto, CA, USA, 1986.

[ACPP05] G. Ascia, V. Catania, M. Palesi, and D. Patti. Hyperblock Formation: A Power/Energy Perspective for High Performance VLIW Architectures. In *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on*, pages 4090–4093 Vol. 4, 2005.

[AHKB00] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock Rate Versus IPC: The End of the Road for Conventional Microarchitectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, pages 248–259, New York, NY, USA, 2000. ACM.

[AKPW83] J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of Control Dependence to Data Dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '83, pages 177–189, New York, NY, USA, 1983. ACM.

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[Alt11] Altera. The Nios II Software Developer's Handbook. In *Documentation: Nios II Processor Version 11.1*. Altera Corporation, 2011.

[Alt13] Altera. Section III: Power Estimation and Analysis. In *Quartus II Handbook Volume 3 Version 13.1.0*. Altera Corporation, Nov 2013.

[Amd67] Gene M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[AN90] K. Arvind and Rishiyur S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Trans. Comput.*, 39:300–318, March 1990.

[BAG05] M. Budiu, P.V. Artigas, and S.C. Goldstein. Dataflow: A Complement to Superscalar. In *Performance Analysis of Systems and Software, 2005. ISPASS 2005. IEEE International Symposium on*, pages 177 –186, march 2005.

[Bat14] Daniel Bates. Exploiting Tightly-Coupled Cores. Technical Report UCAM-CL-TR-846, University of Cambridge, Computer Laboratory, January 2014.

[BC11] Shekhar Borkar and Andrew A. Chien. The Future of Microprocessors. *Commun. ACM*, 54(5):67–77, May 2011.

[BDMF10] Geoffrey Blake, Ronald G. Dreslinski, Trevor Mudge, and Krisztián Flautner. Evolution of Thread-level Parallelism in Desktop Applications. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 302–313, New York, NY, USA, 2010. ACM.

[BIM] Biologically Inspired, Massively Parallel Architectures Project. `http://www.cl.cam.ac.uk/research/comparch/research/bimpa.html`. Accessed: 2014-02-10.

[BMMR05] C. Bazeghi, F.J. Mesa-Martinez, and J. Renau. μComplexity: Estimating Processor Design Effort. In *Microarchitecture, 2005. MICRO-38. Proceedings. 38th Annual IEEE/ACM International Symposium on*, pages 10 pp.–, 2005.

[BS00] J. Adam Butts and Gurindar S. Sohi. A Static Power Model for Architects. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 33, pages 191–201, New York, NY, USA, 2000. ACM.

[Buc93] Joseph T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*. PhD thesis, EECS Department, University of California, Berkeley, 1993.

[Bud03] Mihai Budiu. *Spatial Computation*. PhD thesis, Carnegie Mellon University, Computer Science Department, December 2003. Technical report CMU-CS-03-217.

[BVCG04] Mihai Budiu, Girish Venkataramani, Tiberiu Chelcea, and Seth Copen Goldstein. Spatial Computation. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS-XI, pages 14–26, New York, NY, USA, 2004. ACM.

[CCA+11] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. LegUp: High-Level Synthesis for FPGA-based Processor/Accelerator Systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 33–36, New York, NY, USA, 2011. ACM.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991.

[Cha12] Gregory A. Chadwick. Mamba: A Scalable Communication Centric Multi-threaded Processor Architecture. In *Proceedings of the 2012 IEEE 30th International Conference on Computer Design (ICCD 2012)*, ICCD '12, pages 277–283, Washington, DC, USA, 2012. IEEE Computer Society.

[CHE11] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulations. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, November 2011.

[CJH+12] Simone Campanoni, Timothy Jones, Glenn Holloway, Vijay Janapa Reddi, Gu-Yeon Wei, and David Brooks. HELIX: Automatic Parallelization of Irregular Programs for Chip Multiprocessing. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 84–93, New York, NY, USA, 2012. ACM.

[CLN+11] J. Cong, Bin Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Zhiru Zhang. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 30(4):473–491, April 2011.

[CM08] Philippe Coussy and Adam Morawiec. *High-Level Synthesis: from Algorithm to Digital Circuit*. Springer Publishing Company, Incorporated, 1st edition, 2008.

[Dal02] William J. Dally. Computer Architecture is All About Interconnect. HPCA Panel, February 2002, 2002.

[DD10] Peter J. Denning and Jack B. Dennis. The Resurgence of Parallelism. *Commun. ACM*, 53(6):30–32, June 2010.

[DeH96] A. DeHon. Reconfigurable Architectures for General-Purpose Computing. 1996.

[DeH00] A. DeHon. The Density Advantage of Configurable Computing. *Computer*, 33(4):41–49, 2000.

[DeH13] André M. DeHon. Location, Location, Location: The Role of Spatial Locality in Asymptotic Energy Minimization. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '13, pages 137–146, New York, NY, USA, 2013. ACM.

[DGnY+74] Robert H. Dennard, Fritz H. Gaensslen, Hwa nien Yu, V. Leo Rideout, Ernest Bassous, Andre, and R. Leblanc. Design of Ion-implanted MOSFETs with Very Small Physical Dimensions. *IEEE J. Solid-State Circuits*, page 256, 1974.

[EBSA+11] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM.

[Fin10] Michael Fingeroff. *High-Level Synthesis Blue Book*. Xlibris Corporation, 2010.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[GA06] Ed Grochowski and Murali Annavaram. Energy per Instruction Trends in Intel® Microprocessors. 2006.

[Gao89] Guang R. Gao. Algorithmic Aspects of Balancing Techniques for Pipelined Data Flow Code Generation. *J. Parallel Distrib. Comput.*, 6(1):39–61, February 1989.

[Gao91] GuangR. Gao. Algorithmic Aspects of Pipeline Balancing. In *A Code Mapping Scheme for Dataflow Software Pipelining*, volume 125 of *The Kluwer International Series in Engineering and Computer Science*, pages 41–59. Springer US, 1991.

[GDGN04] Sumit Gupta, Nikil Dutt, Rajesh Gupta, and Alexandru Nicolau. Loop Shifting and Compaction for the High-Level Synthesis of Designs with Complex Control Flow. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1*, DATE '04, pages 10114–, Washington, DC, USA, 2004. IEEE Computer Society.

[GGP92] G.R. Gao, R. Govindarajan, and P. Panangaden. Well-behaved Dataflow Programs for DSP Computation. In *Acoustics, Speech, and Signal Processing, 1992. ICASSP-92., 1992 IEEE International Conference on*, volume 5, pages 561–564 vol.5, Mar 1992.

[GHN+12] Venkatraman Govindaraju, Chen-Han Ho, Tony Nowatzki, Jatin Chhugani, Nadathur Satish, Karthikeyan Sankaralingam, and Changkyu Kim. DySER: Unifying Functionality and Parallelism Specialization for Energy-Efficient Computing. *IEEE Micro*, 32(5):38–51, September 2012.

[GHS11] Venkatraman Govindaraju, Chen-Han Ho, and Karthikeyan Sankaralingam. Dynamically Specialized Datapaths for Energy Efficient Computing. In *Proceedings of 17th International Conference on High Performance Computer Architecture (HPCA)*, 2011.

[GK07] H. Gadke and A. Koch. Comrade - A Compiler for Adaptive Computing Systems using a Novel Fast Speculation Technique. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 503–504, Aug 2007.

[GK08] Hagen Gdke and Andreas Koch. Accelerating speculative execution in high-level synthesis with cancel tokens. In Roger Woods, Katherine Compton, Christos Bouganis, and PedroC. Diniz, editors, *Reconfigurable Computing: Architectures, Tools and Applications*, volume 4943 of *Lecture Notes in Computer Science*, pages 185–195. Springer Berlin Heidelberg, 2008.

[GM08a] Daniel Greenfield and Simon Moore. Implications of Electronics Technology Trends to Algorithm Design. In *Proceedings of the 2008 International Conference on Visions of Computer Science: BCS International Academic Conference*, VoCS'08, pages 331–342, Swinton, UK, UK, 2008. British Computer Society.

[GM08b] Daniel Greenfield and Simon W. Moore. Fractal Communication in Software Data Dependency Graphs. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, pages 116–118, New York, NY, USA, 2008. ACM.

[Gov10] Madhu Sarava Govindan. *E^3: Energy-Efficient EDGE Architectures.* PhD thesis, University of Texas, Austin, 2010.

[Gre11] Peter Greenhalgh. Big.LITTLE Processing with ARM Cortex-A15 and Cortex-A7. http://www.arm.com/files/downloads/big.LITTLE_Final.pdf, ARM Ltd., 2011.

[GTK+02] Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. A Stream Compiler for Communication-Exposed Architectures. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, pages 291–303, New York, NY, USA, 2002. ACM.

[GWK04] Wenrui Gong, Gang Wang, and Ryan Kastner. A High Performance Application Representation for Reconfigurable Systems. In Toomas P. Plaks, editor, *ERSA*, pages 218–224. CSREA Press, 2004.

[Ham08] James Hamilton. Big.LITTLE Processing with ARM Cortex-A15 and Cortex-A7. http:http://perspectives.mvdirona.com/2008/11/28/CostOfPowerInLargeScaleDa 2008. Accessed: 2014-02-01.

[HCAA93] James Hicks, Derek Chiou, Boon Seong Ang, and Arvind. Performance Studies of Id on the Monsoon Dataflow System. *J. Parallel Distrib. Comput.*, 18(3):273–300, July 1993.

[HLK+99] Xuejue Huang, Wen-Chin Lee, Charles Kuo, D. Hisamoto, Leland Chang, J. Kedzierski, E. Anderson, H. Takeuchi, Yang-Kyu Choi, K. Asano, V. Subramanian, Tsu-Jae King, J. Bokor, and Chenming Hu. Sub 50-nm FinFET: PMOS. pages 67–70, 1999.

[HM08] Mark D. Hill and Michael R. Marty. Amdahl's Law in the Multicore Era. *Computer*, 41(7):33–38, July 2008.

[HMMH01] Ron Ho, Kenneth W. Mai, Student Member, and Mark A. Horowitz. The Future of Wires. In *Proceedings of the IEEE*, pages 490–504, 2001.

[HP06] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[HQW+10] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. Understanding Sources of Inefficiency in General-purpose Chips. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 37–47, New York, NY, USA, 2010. ACM.

[HRRJ07] Chao Huang, Srivaths Ravi, Anand Raghunathan, and Niraj K. Jha. Generation of Heterogeneous Distributed Architectures for Memory-Intensive Applications Through High-Level Synthesis. *IEEE Trans. Very Large Scale Integr. Syst.*, 15(11):1191–1204, November 2007.

[HRU+07] Wen-mei Hwu, Shane Ryoo, Sain-Zee Ueng, John H. Kelm, Isaac Gelado, Sam S. Stone, Robert E. Kidd, Sara S. Baghsorkhi, Aqeel A. Mahesri, Stephanie C. Tsao, Nacho Navarro, Steve S. Lumetta, Matthew I. Frank, and Sanjay J. Patel. Implicitly Parallel Programming Models for Thousand-core Microprocessors. In *Proceedings of the 44th annual Design Automation Conference*, DAC '07, pages 754–759, New York, NY, USA, 2007. ACM.

[HTH+08] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, Hiroaki Takada, and Katsuya Ishii. CHStone: A Benchmark Program Suite for Practical C-based High-level Synthesis. In *ISCAS'08*, pages 1192–1195, 2008.

[HW97] J. R. Hauser and J. Wawrzynek. Garp: A MIPS Processor with a Reconfigurable Coprocessor. In *Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines*, FCCM '97, pages 12–, Washington, DC, USA, 1997. IEEE Computer Society.

[H10] Urs Hlzle. Brawny Cores Still Beat Wimpy Cores, Most of the Time. *IEEE Micro*, 2010.

[Ian88] R. A. Iannucci. Toward a Dataflow/Von Neumann Hybrid Architecture. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, ISCA '88, pages 131–140, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[Inc10] Y Explorations (YXI) Inc. eXCite C to RTL Behavioral Synthesis 4.1(a). 2010.

[JC97] Johan Janssen and Henk Corporaal. Making Graphs Reducible with Controlled Node Splitting. *ACM Trans. Program. Lang. Syst.*, 19(6):1031–1052, November 1997.

[Jen91] Kurt Jensen. Coloured Petri Nets: A High Level Language for System Design and Analysis. In Grzegorz Rozenberg, editor, *Advances in Petri Nets 1990*, volume 483 of *Lecture Notes in Computer Science*, pages 342–416. Springer Berlin / Heidelberg, 1991.

[JM03] Neil Johnson and Alan Mycroft. Combined Code Motion and Register Allocation Using the Value State Dependence Graph. In *Proceedings of the 12th International Conference on Compiler Construction*, CC'03, pages 1–16, Berlin, Heidelberg, 2003. Springer-Verlag.

[Joh04] Neil E. Johnson. Code Size Optimization for Embedded Processors. Technical Report UCAM-CL-TR-607, University of Cambridge, Computer Laboratory, November 2004.

[JSMP13] José A. Joao, M. Aater Suleman, Onur Mutlu, and Yale N. Patt. Utility-based Acceleration of Multithreaded Applications on Asymmetric CMPs. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 154–165, New York, NY, USA, 2013. ACM.

[KBPS09] Jonathan G. Koomey, Christian Belady, Michael Patterson, and Anthony Santos. Assessing Trends Over Time in Performance, Costs, and Energy Use for Servers, 2009.

[KFJ+03] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 81–, Washington, DC, USA, 2003. IEEE Computer Society.

[KNM+08] Sami Khawam, Ioannis Nousias, Mark Milward, Ying Yi, Mark Muir, and Tughrul Arslan. The Reconfigurable Instruction Cell Array. *IEEE Trans. Very Large Scale Integr. Syst.*, 16(1):75–85, January 2008.

[KR06] Ian Kuon and Jonathan Rose. Measuring the Gap between FPGAs and ASICs. In *Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*, FPGA '06, pages 21–30, New York, NY, USA, 2006. ACM.

[KSG$^+$07] Changkyu Kim, Simha Sethumadhavan, M. S. Govindan, Nitya Ranganathan, Divya Gulati, Doug Burger, and Stephen W. Keckler. Composable Lightweight Processors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 381–394, Washington, DC, USA, 2007. IEEE Computer Society.

[KSP07] Srikanth Kurra, Neeraj Kumar Singh, and Preeti Ranjan Panda. The Impact of Loop Unrolling on Controller Delay in High Level Synthesis. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '07, pages 391–396, San Jose, CA, USA, 2007. EDA Consortium.

[KTR$^+$04] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. *SIGARCH Comput. Archit. News*, 32(2):64–, March 2004.

[KW02] Apostolos A. Kountouris and Christophe Wolinski. Efficient Scheduling of Conditional Behaviors for High-level Synthesis. *ACM Trans. Des. Autom. Electron. Syst.*, 7(3):380–412, July 2002.

[Lan92] William Landi. Undecidability of Static Analysis. *ACM Lett. Program. Lang. Syst.*, 1(4):323–337, December 1992.

[Law07] Alan C. Lawrence. Optimizing Compilation with the Value State Dependence Graph. Technical Report UCAM-CL-TR-705, University of Cambridge, Computer Laboratory, December 2007.

[Lee06] Edward A. Lee. The Problem with Threads. *Computer*, 39(5):33–42, May 2006.

[Leg13] LegUp 3.0 Quality of Results (CHStone). `http://www.legup.org:9100/perf/dashboard/overview.html`, 2013. Accessed: 2013-07-14.

[LLV] LLVM Language Reference Manual. `http://http://llvm.org/docs/LangRef.html`. Accessed: 2014-02-10.

[LMR07] Anton Lokhmotov, Alan Mycroft, and Andrew Richards. Delayed Side-effects Ease Multi-core Programming. In *Proceedings of the 13th International Euro-Par Conference on Parallel Processing*, Euro-Par'07, pages 641–650, Berlin, Heidelberg, 2007. Springer-Verlag.

[LNC13] Xiangyang Liang, Minh Nguyen, and Hao Che. Wimpy or Brawny Cores: A Throughput Perspective. *Journal of Parallel and Distributed Computing*, (0):–, 2013.

[LP02] Edward A. Lee and Thomas M. Parks. Readings in Hardware/Software Co-Design. chapter Dataflow Process Networks, pages 59–85. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[LPC12] Feng Li, Antoniu Pop, and Albert Cohen. Automatic Extraction of Coarse-Grained Data-Flow Threads from Imperative Programs. *IEEE Micro*, 32(4):19–31, 2012.

[LPN+13] Joo Hwan Lee, Kaushik Patel, Nimit Nigania, Hyojong Kim, and Hyesoon Kim. OpenCL Performance Evaluation on Modern Multi Core CPUs. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '13, pages 1177–1185, Washington, DC, USA, 2013. IEEE Computer Society.

[LW92] Monica S. Lam and Robert P. Wilson. Limits of Control Flow on Parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ISCA '92, pages 46–57, New York, NY, USA, 1992. ACM.

[MCC+06] M. Mishra, T.J. Callahan, T. Chelcea, G. Venkataramani, S.C. Goldstein, and M. Budiu. Tartan: Evaluating Spatial Computation for Whole Program Execution. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 163–174. ACM, 2006.

[MG07] M. Mishra and S.C. Goldstein. Virtualization on the Tartan Reconfigurable Architecture. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 323–330, 2007.

[MG08] Simon Moore and Daniel Greenfield. The Next Resource War: Computation vs. Communication. In *Proceedings of the 2008 International Workshop on System Level Interconnect Prediction*, SLIP '08, pages 81–86, New York, NY, USA, 2008. ACM.

[MLC+92] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. Effective Compiler Support for Predicated Execution Using the Hyperblock. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, MICRO 25, pages 45–54, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[MM09] J. Mak and A. Mycroft. Limits of Parallelism Using Dynamic Dependence Graphs. In *International Workshop on Dynamic Analysis*, pages 42–48, 2009.

[MTZ13] Daniel S. McFarlin, Charles Tucker, and Craig Zilles. Discerning the Dominant Out-Of-Order Performance Advantage: Is It Speculation or Dynamism? In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 241–252, New York, NY, USA, 2013. ACM.

[NFMM13] Matthew Naylor, Paul J. Fox, A. Theodore Markettos, and Simon W. Moore. Managing the FPGA Memory Wall: Custom Computing or Vector Processing? In *FPL*, pages 1–6, 2013.

[Nik04] Rishiyur Nikhil. Bluespec System Verilog: Efficient, Correct RTL from High Level Specifications. In *Formal Methods and Models for Co-Design, 2004. MEMOCODE '04. Proceedings. Second ACM and IEEE International Conference on*, pages 69–70, 2004.

[Nik08] RishiyurS. Nikhil. Bluespec: A General-Purpose Approach to High-Level Synthesis Based on Parallel Atomic Transactions. In Philippe Coussy and Adam Morawiec, editors, *High-Level Synthesis*, pages 129–146. Springer Netherlands, 2008.

[NRE04] R. Namballa, N. Ranganathan, and A. Ejnioui. Control and Data Flow Graph Extraction for High-level Synthesis. In *VLSI, 2004. Proceedings. IEEE Computer Society Annual Symposium on*, pages 187–192, 2004.

[OH05] Kunle Olukotun and Lance Hammond. The Future of Microprocessors. *Queue*, 3(7):26–29, September 2005.

[Ott08] Guilherme De Lima Ottoni. Global Instruction Scheduling for Multi-Threaded Architectures. 2008.

[Pan01] Preeti Ranjan Panda. SystemC: A Modeling Platform Supporting Multiple Design Abstractions. In *Proceedings of the 14th International Symposium on Systems Synthesis*, ISSS '01, pages 75–80, New York, NY, USA, 2001. ACM.

[PBD+08] Andrew R. Putnam, Dave Bennett, Eric Dellinger, Jeff Mason, and Prasanna Sundararajan. CHiMPS: A High-Level Compilation Flow for Hybrid CPU-FPGA Architectures. In *Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays*, FPGA '08, pages 261–261, New York, NY, USA, 2008. ACM.

[PHS85] Y. N. Patt, W. M. Hwu, and M. Shebanow. HPS, A New Microarchitecture: Rationale and Introduction. In *Proceedings of the 18th Annual Workshop on Microprogramming*, MICRO 18, pages 103–108, New York, NY, USA, 1985. ACM.

[PJ87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.

[PJS97] Subbarao Palacharla, Norman P. Jouppi, and J. E. Smith. Complexity-Effective Superscalar Processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, pages 206–218, New York, NY, USA, 1997. ACM.

[Plo81] Gordon D. Plotkin. A Structural Approach to Operational Semantics. Technical Report Technical Report DAIMI FN-19, Aarhus University, 1981.

[PMHS85] Y. N. Patt, S. W. Melvin, W. M. Hwu, and M. C. Shebanow. Critical Issues Regarding HPS, a High Performance Microarchitecture. In *Proceedings of the 18th Annual Workshop on Microprogramming*, MICRO 18, pages 109–116, New York, NY, USA, 1985. ACM.

[PNBK02] Hooman Parizi, Afshin Niktash, Nader Bagherzadeh, and Fadi J. Kurdahi. MorphoSys: A Coarse Grain Reconfigurable Architecture for Multimedia Applications (Research Note). In *Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, Euro-Par '02, pages 844–848, London, UK, UK, 2002. Springer-Verlag.

[Pol99] Fred J. Pollack. New Microarchitecture Challenges in the Coming Ggenerations of CMOS Process Technologies (keynote address)(abstract only). In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 32, pages 2–, Washington, DC, USA, 1999. IEEE Computer Society.

[PPA+13] Angshuman Parashar, Michael Pellauer, Michael Adler, Bushra Ahsan, Neal Crago, Daniel Lustig, Vladimir Pavlov, Antonia Zhai, Mohit Gambhir, Aamer Jaleel, Randy Allmon, Rachid Rayess, Stephen Maresh, and Joel Emer. Triggered Instructions: A Control Paradigm for Spatially-programmed Architectures. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 142–153, New York, NY, USA, 2013. ACM.

[PPM+06] Andrew Petersen, Andrew Putnam, Martha Mercaldi, Andrew Schwerin, Susan Eggers, Steve Swanson, and Mark Oskin. Reducing Control Overhead in Dataflow Architectures. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, PACT '06, pages 182–191, New York, NY, USA, 2006. ACM.

[PPM09] Hyunchul Park, Yongjun Park, and Scott Mahlke. Polymorphic Pipeline Array: A Flexible Multicore Accelerator with Virtualized Execution for Mobile Multimedia Applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 370–380, New York, NY, USA, 2009. ACM.

[PT05] David Pellerin and Scott Thibault. *Practical FPGA Programming in C.* Prentice Hall Press, Upper Saddle River, NJ, USA, first edition, 2005.

[RM11] M. Rostami and K. Mohanram. Dual-V_{th} Independent-Gate FinFETs for Low Power Logic Circuits. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30(3):337–349, March 2011.

[RRP+07] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 13–24, 2007.

[SBV95] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar Processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, ISCA '95, pages 414–425, New York, NY, USA, 1995. ACM.

[SG08] S. Singh and D. Greaves. Kiwi: Synthesis of FPGA Circuits from Parallel Programs. In *Field-Programmable Custom Computing Machines, 2008. FCCM '08. 16th International Symposium on*, pages 3–12, April 2008.

[SGM+06] Aaron Smith, Jon Gibson, Bertrand Maher, Nick Nethercote, Bill Yoder, Doug Burger, Kathryn S. McKinle, and Jim Burrill. Compiling for EDGE Architectures. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 185–195, Washington, DC, USA, 2006. IEEE Computer Society.

[SL05] Herb Sutter and James Larus. Software and the Concurrency Revolution. *Queue*, 3(7):54–62, September 2005.

[SLH90] Michael D. Smith, Monica S. Lam, and Mark A. Horowitz. Boosting Beyond Static Scheduling in a Superscalar Processor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, pages 344–354, New York, NY, USA, 1990. ACM.

[SNL+04] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Nitya Ranganathan, Doug Burger, Stephen W. Keckler, Robert G. McDonald, and Charles R. Moore. TRIPS: A Polymorphous Architecture for Exploiting ILP, TLP, and DLP. *ACM Trans. Archit. Code Optim.*, 1(1):62–93, March 2004.

[SSM+07] Steven Swanson, Andrew Schwerin, Martha Mercaldi, Andrew Petersen, Andrew Putnam, Ken Michelson, Mark Oskin, and Susan J. Eggers. The WaveScalar Architecture. *ACM Trans. Comput. Syst.*, 25:4:1–4:54, May 2007.

[SSV08] Scott Sirowy, Greg Stitt, and Frank Vahid. C is For Circuits: Capturing FPGA Circuits as Sequential Code for Portability. In *Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays*, FPGA '08, pages 117–126, New York, NY, USA, 2008. ACM.

[Sti11] Greg Stitt. Are Field-Programmable Gate Arrays Ready for the Mainstream? *IEEE Micro*, 31(6):58–63, November 2011.

[Sut05] Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobbs Journal*, 30(3):202–210, 2005.

[SVGH+11] Jack Sampson, Ganesh Venkatesh, Nathan Goulding-Hotta, Saturnino Garcia, Steven Swanson, and Michael Bedford Taylor. Efficient Complex Operators for Irregular Codes. In *HPCA 2011: High Performance Computing Architecture*, 2011.

[SW11] James Stanier and Des Watson. A Study of Irreducibility in C Programs. *Softw: Pract. Exper.*, page n/a, 2011.

[TA03] Jessica H. Tseng and Krste Asanović. Banked Multiported Register Files for High-frequency Superscalar Microprocessors. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ISCA '03, pages 62–71, New York, NY, USA, 2003. ACM.

[Tay12] Michael B. Taylor. Is Dark Silicon Useful?: Harnessing the Four Horsemen of the Coming Dark Silicon Apocalypse. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 1131–1136, New York, NY, USA, 2012. ACM.

[TDTD90] Walter F. Tichy, Peter J. Denning, Walter F. Tichy, and Peter J. Denning. Highly Parallel Computation. *Science*, 250:1217–1222, 1990.

[Tec12] Maxeler Technologies. Publications. `http://www.maxeler.com/publications/`, 2012.

[TLM+04] Michael Bedford Taylor, Walter Lee, Jason Miller, David Wentzlaff, Ian Bratt, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jason Kim, James Psota, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA '04, pages 2–, Washington, DC, USA, 2004. IEEE Computer Society.

[TPB98] Scott Thompson, Paul Packan, and Mark Bohr. MOS Scaling: Transistor Challenges for the 21st Century. *Intel Technology Journal*, 1998.

[Tra86] K. R. Traub. A Compiler for the MIT Tagged-Token Dataflow Architecture. Technical report, Cambridge, MA, USA, 1986.

[TWFO09] Georgios Tournavitis, Zheng Wang, Björn Franke, and Michael F.P. O'Boyle. Towards a Holistic Approach to Auto-parallelization: Integrating Profile-driven Parallelism Detection and Machine-learning Based Mapping. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 177–187, New York, NY, USA, 2009. ACM.

[UM02] Sebastian Unger and Frank Mueller. Handling Irreducible Loops: Optimized Node Splitting Versus DJ-graphs. *ACM Trans. Program. Lang. Syst.*, 24(4):299–333, July 2002.

[VBCG06] G. Venkataramani, T. Bjerregaard, T. Chelcea, and S. C. Goldstein. Hardware Compilation of Application-Specific Memory-Access Interconnect. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 25(5):756–771, November 2006.

[Ven11] Ganesh Venkatesh. *Configurable Energy-efficient Co-processors to Scale the Utilization Wall*. PhD thesis, University of California at San Diego, La Jolla, CA, USA, 2011. AAI3456370.

[VSG+10] 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. *ACM SIGARCH Computer Architecture News*, 38(1):205–218, 2010.

[Wal91] David W. Wall. Limits of Instruction-Level Parallelism. *SIGPLAN Not.*, 26:176–188, April 1991.

[Wan07] Jiacun Wang. Handbook of Dynamic System Modeling Jun 2007. chapter Petri Nets for Dynamic Event-Driven System Modeling. Chapman and Hall/CRC, 2007.

[WFW+94] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. *SIGPLAN Not.*, 29(12):31–37, December 1994.

[WM95] Wm. A. Wulf and Sally A. McKee. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995.

[WP94] Paul G. Whiting and Robert S. V. Pascoe. A History of Data-Flow Languages. *IEEE Ann. Hist. Comput.*, 16(4):38–59, December 1994.

[YAMJGE13] F. Yazdanpanah, C. Alvarez-Martinez, D. Jimenez-Gonzalez, and Y. Etsion. Hybrid Dataflow/Von-Neumann Architectures. *Parallel and Distributed Systems, IEEE Transactions on*, PP(99):1–1, 2013.

[YRHBL13] Paraskevas Yiapanis, Demian Rosas-Ham, Gavin Brown, and Mikel Luján. Optimizing Software Runtime Systems for Speculative Parallelization. *ACM Trans. Archit. Code Optim.*, 9(4):39:1–39:27, January 2013.

[ZK98] V. Zyuban and P. Kogge. The Energy Complexity of Register Files. In *Proceedings of the 1998 International Symposium on Low Power Electronics and Design*, ISLPED '98, pages 305–310, New York, NY, USA, 1998. ACM.