Design Automation for Streaming Systems

Eylon Caspi

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2005-27
http://www.eecs.berkeley.edu/Pubs/TechRpts/2005/EECS-2005-27.html

December 16, 2005

Copyright © 2005, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Design Automation for Streaming Systems

by

Eylon Caspi

B.S. (University of Maryland, College Park) 1996
M.S. (University of California, Berkeley) 2000

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

Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor John Wawrzynek, Chair
Professor Edward A. Lee
Professor Robert Jacobsen

Fall 2005

Design Automation for Streaming Systems

Copyright © 2005

by

Eylon Caspi

Abstract

Design Automation for Streaming Systems

by

Eylon Caspi

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor John Wawrzynek, Chair

RTL design methodologies are struggling to meet the challenges of modern, large system design. Their reliance on manually timed design with fully exposed device resources is laborious, restricts reuse, and is increasingly ineffective in an era of Moore's Law expansion and growing interconnect delay. We propose a new hardware design methodology rooted in an abstraction of communication timing, which provides flexibly timed module interfaces and automatic generation of pipelined communication. Our core approach is to replace inter-module wiring with streams, which are FIFO buffered channels. We develop a process network model for streaming systems (TDFPN) and a hardware description language with built in streams (TDF). We describe a complete synthesis methodology for mapping streaming applications to a commercial FPGA, with automatic generation of efficient hardware streams and module-side flow control. We use this methodology to compile seven multimedia applications to a Xilinx Virtex-II Pro FPGA, finding that stream support can be relatively inexpensive. We further propose a comprehensive, system-level optimization flow that uses information about streaming behavior to guide automatic communication buffering, pipelining, and placement. We discuss specialized stream support on reconfigurable, programmable platforms, with intent to provide better results and compile times than streaming on generic FPGAs. We also show how streaming can support an efficient abstraction of area, allowing an entire system to be reused with automatic performance improvement on larger, next generation devices.

Professor John Wawrzynek, Chair   Date

Acknowledgements

First and foremost, I owe an incalculable debt of gratitude to my wife, Denise, and children, Josh and Ross. They have seen me through all too many years of graduate school, late nights, busy weekends, conference trips, and deadlines. They picked me up every time I was down. Without them, this dissertation would not have been completed. Likewise, thanks to my parents, Ehud and Rachel, without whom this dissertation might never have begun. And thanks to the rest of the family, Amir, Heather, and Richard, for all their support when the going got tough.

I am indebted to a number of advisors and mentors. Thanks to my research advisor, John Wawrzynek, for supporting this exciting course of research, and for making sure we always had industry visibility. Thanks to professor Edward Lee for a brilliant course on concurrent models that helped center my work, and for advising on this dissertation. Thanks to my external committee member, professor Bob Jacobsen, for encouragement and for keeping this dissertation honest and accessible. Thanks to Seth Goldstein (CMU) for profound comments that forever improved my technical presentations. Thanks to Carl Ebeling (U. Washington) for helping me find my way between academia and industry. Thanks to Kees Vissers for guidance and encouragement, both technical and practical, at Berkeley and during my time at Chameleon Systems.

Extra special thanks goes to André DeHon (CalTech), who has been a mentor and a friend for years. During his post-doc at Berkeley, André co-founded our research group, taught our first course on reconfigurable computing, and guided the SCORE project. Even after his departure, he continued to provide technical and personal guidance, as well as compute resources for the experiments in Chapter 4. Perhaps the strongest compliment to him is my having been dubbed *Mini André*.

Thanks to all the past and present members of the BRASS research group, and particularly to those on the SCORE project, without whom my work would not exist. Thanks to Michael Chu, Randy Huang, Yury Markovskiy, Laura Pozzi, and André DeHon for building the SCORE software base. Special thanks to Joe Yeh, who valiantly wrote our entire suite of multimedia applications in a strange (and often broken) language, TDF. Thanks to Nick Weaver for helping to turn me onto reconfigurable in the first place. Thanks to John Hauser and Tim Callahan for being mentors. Thanks to Norm Zhou, Kip Macy, Yujia Jin, James Yeh, and all the other occupants of our cozy office, 441 Soda, for making my years at Berkeley so much fun. Thanks to Jon Kuroda for keeping our computers from becoming door stops. Thanks to the BRASS group's industry liaisons, especially Steve Trimberger (Xilinx), for keeping a watchful eye on us. And thanks to DARPA, NSF, California Micro, Xilinx, and STMicroelectronics for funding our work.

Additional thanks goes to the IC group at CalTech, including Michael Wrighton, Michael Delorimier, Rafi Rubin, Helia Naeimi, Nachiket Kapre, and Stephan Lavavej. They carried the torch in the later days of SCORE, gave me many good ideas, and stress tested TDF with new applications. Likewise, thanks to Osvaldo Colavin, Antoine Hue, and others at STMicroelectronics, who evaluated and tested our software.

Thanks to Steve Neuendorffer and Jörn Janneck (Xilinx) for numerous discussions about process network theory. Thanks to Tero Rissa for helping SCORE (and me!) get to Finland. Thanks to everyone who fed the "starving" graduate students, including the computer science social hour, countless industrial recruiting sessions, and our favorite hangouts, Bongo Burger, LaVal's, La Burrita, and Seven Palms. Thanks to the U.C. Regents for making Berkeley a great place to study.

And finally, thanks to all the wives (and husbands) of the people mentioned above, who went out of their way every day to make our work possible.

For my wife, Denise,

and our children, Josh and Ross

Contents

1 Introduction **1**

1.1 System Design Challenges 1

1.2 Limitations of RTL 4

1.3 Timing Tolerant System Composition 6

1.4 Stream Based Design 9

1.5 Related Streaming Models and Architectures 15

1.6 Contribution and Organization 22

2 Language and Theoretical Foundation **24**

2.1 Streaming Models 25

2.2 TDF Process Networks 30

2.3 TDF Language 38

2.4 TDFPN Denotational Semantics 54

3 Synthesis Methodology for FPGA **73**

3.1 Compilation Flow 74

3.2 Stream Protocol 77

3.3 SFSM Synthesis 77

3.4 Queue Implementation 80

3.5 Streaming Memories 86

3.6 System Composition 87

3.7 Stream Enabled Pipelining 89

4 Characterization for FPGA 96

4.1 Experimental Setup 97

4.2 Applications 99

4.3 Baseline Results 107

4.4 Logic Pipelining 114

4.5 Interconnect Pipelining 128

4.6 Summary 144

5 System Optimization 147

5.1 System Optimization Flow 150

5.2 Stream Buffer Bounds Analysis 153

5.3 Stream Aware Placement 183

5.4 Process Pipelining 193

5.5 Pipeline Balancing 194

5.6 Serialization 201

6 Streaming Programmable Platforms 204

6.1 Basic Platform 207

6.2 Paged Platform 215

6.3 Virtual Paged Platform 221

6.4 Page Partitioning 238

7 Conclusion 254

7.1 Summary 254

7.2 Open Questions, Future Work 258

7.3 History and Relation to SCORE 265

Bibliography **269**

Chapter 1

Introduction

1.1 System Design Challenges

The design of modern computational systems is driven by an incessant market demand for systems to have more functionality, be faster, be smaller, last longer on battery, and be cheaper. The industry's success in meeting such a tall order has been fueled by Moore's Law, namely the uncanny ability of process engineers to put more transistors on a chip each year. Technology improvements in recent years have reduced feature sizes on chip by an average of 11%-16% per year, leading to improvements in clock rates and to a doubling of transistor capacity every 1.5 to 2 years [ITRS, 2003].

Moore's law improvements also bring a host of difficulties associated with large system design and deep sub-micron (DSM) effects. Today's largest systems involve tens of millions of gates, or hundreds of millions of transistors[1], necessitating a highly modular design methodology and large teams of up to hundreds of engineers. One of the greatest design challenges is the growing dominance of interconnect delay, which owes to disproportionate improvements in wire speed versus logic speed at each technology generation [Agarwal *et al.*, 2000] [Sylvester and Keutzer, 1999]. Designers are forced to pipeline deeper to meet target clock rates, complicating designs further. The large and often unpredictable impact of distance on performance leads to an insidious *timing closure* problem, where design modifications intended to improve performance may actually diminish it due to changes in area and layout. These trends, and the inability of design automation tools to keep up, have resulted in a *design productivity gap*, where the number of transistors available on chip is growing 58% per year, but the number of transistors programmed per man-month is growing only 21% per year [ITRS, 1999].

Programmable platforms, such as field programmable gate arrays (FPGAs), have emerged as a popular alternative to application specific integrated circuits (ASICs). Since they are off-the-shelf components, FPGAs free system designers from many of the DSM effects involved in transistor level chip design, including parasitics extraction, crosstalk, signal integrity, clock distribution, and electromigration. FPGA based design also avoids the high cost of mask production for chip fabrication. Modern, high capacity FPGAs can emulate millions of logic gates using software programmable look-up tables, so they can accommodate a significant number of real world designs. Furthermore, due to their regularity, FPGAs tend to be early adopters of new technology generations, bringing with them the promise of technology scaling without the cost of fabricating a chip. Unfortunately, with such large device sizes, FPGA based design suffers from many of the same large system and DSM effects as ASIC design, including high design complexity, growing interconnect delay, and long run-times for tools such as place-and-route. For example, routing typically accounts for 45%-65% of the critical path on an FPGA [Xilinx, 2003a]. Thus, a productivity gap remains.

Despite the growing complexity and cost of system design, systems are regularly discarded and redesigned with each new device generation. Hardware design method-

[1] Some multi-million transistor systems—AMD Athlon 64 X2: 233M transistors [Mitchell, 2005], IBM Cell: 241M transistors [Gschwind *et al.*, 2005], nVidia GeForce 7800 GTX: 302M transistors [Kirk, 2005].

ologies tend to be very tied to the specific sizes and timings of a device. Consequently, significant redesign is necessary to take full advantage of the greater area on a new device, and to deal with disproportionate changes in logic and interconnect delays. Engineers strive to reuse at least some modules of a design, but this reuse does not mitigate the need for system redesign. The effort for engineering a system is not only growing, it is being largely repeated from scratch.

Microprocessor architectures have allowed some designers to press on, blissfully unaware of DSM difficulties and the productivity gap. Such architectures are programmed using instruction set architectures (ISA) that abstract away the underlying circuit implementation. This abstraction also allows microprocessor software to survive and scale on compatible, next generation processors, riding Moore's Law to better performance. However, processor based systems have limited performance and limited density (in terms of performance-per-dollar or performance-per-watt), so they cannot serve the entire market. The loss in density comes from supporting the ISA abstraction, and increasingly from extra area devoted to parallelizing instruction streams and overcoming long interconnect delay (*e.g.* large caches to hide memory access latency). It is predicted that future improvements in microprocessor performance will be strictly limited by interconnect delay, which limits the amount of state accessible in each clock cycle [Agarwal *et al.*, 2000]. To continue improving performance, designers will be forced to abandon the trend towards larger, more complex microprocessors, and instead consider more parallel architectures. Already today, many systems-on-chip (SoC) combine one or more simple processors with many other cores, and their designers are forced to deal with the hard issues of chip design.

We contend that many of the difficulties in modern system design stem from the prevalent use of register transfer level (RTL) design methodologies, which fully expose device timing. To increase productivity and design longevity, we must consider new abstractions that hide or otherwise assist in dealing with technology limiters such as interconnect delay and area. To this end, we shall propose an abstraction of timing based on *streams* as a replacement for long wires. We shall also show how streams support an efficient abstraction of area, given proper architectural support. These abstractions will be supported and enforced by a language and an automatic compiler for hardware. The abstractions enable a number of system-level analyses and optimizations that are not possible in RTL-based design, which serve to improve designer productivity and to promote reuse, retargeting, and scaling.

1.2 Limitations of RTL

The prevalent design abstraction for digital systems today is the *register transfer level* (RTL) description, which specifies a timed signal flow through logic and registers. For the sake of discussion, we distinguish the term *timing*, referring to picosecond measures such as signal delays and setup and hold times, from *timing behavior*, referring to integer measures of machine cycles. An RTL description fully specifies the timing behavior of a circuit, *i.e.* the cycle by cycle schedule of computation and communication. The design philosophy of RTL is that, to achieve a target clock rate, timing should be predictable and exposed to the designer, and the designer should manually pipeline the circuit accordingly. Design methodologies have improved by abstracting from *structural* descriptions, involving netlists of logic and registers, to *behavioral* descriptions, involving state machines. However, the basic paradigm of fully specified timing behavior persists.

The growing dominance of interconnect delay on chip poses a serious difficulty for RTL based design, since it makes timing unpredictable. For larger circuits, signal delays become dependent on placement, which is normally determined only after synthesis from RTL. Thus, it is not obvious how to best pipeline a circuit and choose its timing behavior. The industry has attempted a number of approaches for improving timing predictability in large circuits, with limited success. Making placement explicit before RTL, *i.e.* floorplanning, is possible but requires significant design effort. Physical synthesis attempts to feed timing information from placement back to guide RTL transformations and resynthesis. However, its improvement on clock rate is limited (*e.g.* 5%-20% in Synplify Premiere [Synplicity, 2005b]), since it is not allowed to introduce new pipeline stages or to modify visible timing behavior. Retiming can be modified to consider interconnect delay by performing it during [Singh and Brown, 2002] or after [Weaver *et al.*, 2003] placement, but it too is not allowed to introduce new pipeline stages or to modify visible timing behavior. The delays in question grow with each device generation, so ultimately, deeper pipelining is a necessity.

Large RTL designs are typically decomposed into small modules. On the surface, this seems to improve timing predictability, since inside a small module, interconnect delay would remain small and predictable. However, this approach merely exports the problem to the level of inter-module connections, which themselves need to be pipelined. Pipelining inter-module connections is usually not straightforward, since it breaks the expected timing behavior of module interfaces. In the fully-timed world of RTL design, a module's internal behavior is coupled to its communication behavior, which in turn is coupled to the behavior of other modules and the entire system. The use of modules as design units, in and of itself, does nothing to decouple the modules in time and nothing to address interconnect delay. In this sense, the philosophy of fully timed RTL design is actually obstructive to modularity—it limits module reuse in new timing environments, and it prevents modular optimizations that modify an individual module's timing behavior. A truly modular design methodology must be more tolerant of changes in timing behavior.

1.3 Timing Tolerant System Composition

Believing that predictable timing is a lost cause, there are a number of efforts to make timing more abstract to the designer. Most are based on timing tolerant interfaces between modules.

1.3.1 Asynchronous Circuits

Asynchronous circuits take timing tolerance to the extreme by removing clocks altogether and using handshaking to tolerate any delay. As a consequence, asynchronous circuits accommodate pipelining by simple insertion of *half buffer* elements [Lines, 1995]. Unfortunately, asynchronous circuits are not well supported by design tools, and they incur an area penalty for handshaking. A more practical approach is to introduce handshaking and asynchrony only at a coarser granularity, between modules. The GALS (*globally asynchronous locally synchronous*) approach involves conventional, synchronous design within modules and pipelined, asynchronous interconnect between modules. With this two-level hierarchy, interconnect between modules may be packaged as IP (intellectual property) or as a network on chip (NOC), *e.g.* Nexus [Lines, 2004].

1.3.2 Synchronous Standard Interfaces

It is not necessary to resort to asynchronous circuits in order to decouple the timing behavior of modules. A purely synchronous handshaking protocol would suffice. Several system on chip (SOC) interconnect standards exist which use such handshaking, including CoreConnect [IBM, 1999], AMBA [ARM, 1999], and OCP [OCP, 2001]. Such standards provide conventions for module interfaces, along with IP blocks to implement inter-module transport. However, only some packages provide a pipelined transport to enable long distance communication at high clock rates, *e.g.* the SONICS SiliconBackplane implementation of OCP [Sonics, 2005] point-to-point links.

A mere specification of interface conventions, as in the standards above, is only slightly better than none at all. It puts the burden of conforming to the interface on the module designer, including writing a state machine to sequence all the required signals. It also puts the burden of choosing interconnect implementation parameters on the designer, including pipelining depth, queue size, bit width, and so on. These manual choices ultimately obstruct reuse, since they must be reconsidered when porting to a next generation device, and changes may require reimplementation of modules. Making reuse practical would require standard interface packages to provide automation for choosing interconnect parameters and for regenerating module interfaces. Our streaming design methodology does provide that automation.

1.3.3 Latency Insensitive Design

Latency insensitive design (LID) [Carloni *et al.*, 2003] [Carloni *et al.*, 2001] is a more transparent approach to system composition, providing automatic inter-module pipelining without handshaking or modification of modules. Pipelining inter-module wires changes the arrival time of signals, and may cause the inputs of a module to arrive out of phase. To retain correct behavior, LID specifies that a module must stall until a complete, matched set of its inputs has arrived. Each module is embedded in a *shell* that stalls the module via clock gating and provides queueing to realign inputs in time. The required queue depths and schedule of stalls are in fact static and fully predictable from the interconnect pipelining depths. Thus, the process of choosing pipeline depths and programming the shells can be performed automatically after placement. This design methodology successfully avoids the reduction in clock rate associated with long distance, inter-module communication. Nevertheless, the addition of pipelining on feedback paths incurs a loss of throughput (data per clock period) and hence, a loss of system performance. This loss manifests easily on module interfaces that include feedback or hand-shaking, such as ready signals, and prevents those interfaces from operating at full throughput. Thus, for communication styles beyond simple, unidirectional wiring, LID may exacerbate rather than alleviate the effect of long interconnect.

In principle, it is possible to pipeline a handshaking protocol across a long distance at full throughput by *relaying* it. Rather than connecting two modules directly, a module may be connected to a protocol relay block that implements the same handshaking protocol. The relay buffers and retransmits transactions to another relay or to the target module. The first module may then go about its business, without waiting for the round trip delay of handshaking directly with the target module. Modules are thus decoupled in time, and each module is capable of transmitting or receiving at full throughput. This technique is used in asynchronous half buffer insertion, SONICS point-to-point links, as well as our own stream protocol. In contrast, LID relays raw wires rather than protocols. It transforms a system at too low a level to know that a protocol is being disturbed, so it cannot guarantee full throughput communication.

1.3.4 Bluespec

Bluespec [Bluespec, 2005] and its underlying term rewriting system [Arvind *et al.*, 2004] [Hoe and Arvind, 2004] provide a different approach to timing tolerant system composition. Their basic abstraction is that of global state being updated by guarded, atomic actions, termed *rules*. Rules may be local to a module or may relate the state of several modules, inferring communication. Thus, communication and synchronization between connected modules need not be specified as timed behaviors, like RTL, but are derived automatically from rules. Concurrency comes from evaluating all rules and applying as many non-conflicting actions as possible in each clock period. Timing tolerance comes from the allowance that an action may be deferred for several cycles, including an action that infers communication. Synthesis automatically generates all guard logic, interlock logic to prevent conflicting actions, and wiring between modules. To promote modular design and module reuse in Bluespec, a module's state is hidden behind *interface methods*, and modules communicate by invoking each others methods. In synthesis, those methods become guarded by the composition of rules across connected modules. The performance of a system depends on the combinational delay in the inferred composition of guards and actions, including inter-module wiring.

Unfortunately, pipelining in Bluespec remains a completely manual effort. Pipelining a module requires explicitly specifying new state for pipeline registers and factoring rules to use them. Likewise, decoupling inter-module connections in time requires declaring interface registers, queues, or some other intermediate, stateful elements. Specifying and exploring connection styles is substantially easier in Bluespec than in RTL. However, Bluespec does little to address growing interconnect delay. In fact, its reliance on analyzing global state is arguably obstructive to that end.

1.4 Stream Based Design

We seek a minimal set of abstractions beyond RTL to tackle some of the issues above concerning large systems, long interconnect delay, and design reuse. Our fundamental approach is to replace inter-module wires with *streams*, a particular form of buffered channels. Streams abstract communication timing, making a system more robust to delay, and supporting better modular design through timing-flexible interface specifications. Streams also abstract communication implementation, so they are amenable to automatic generation of efficient handshaking, transport, and buffering. Streams represent dataflow between modules, not mere protocols, so they enable a compiler to perform more system level analyses and optimizations. With language and tool support, a streaming system can be automatically retargeted to next generation process technologies or programmable devices, thus enabling entire system reuse.

We briefly describe our streaming model here, chiefly in the context of technology trends and design productivity. More complete specification of the model and its language are provided in Chapter 2.

1.4.1 Streaming System Composition

Modules are connected by a *stream* communication mechanism. A stream is a point-to-point, unidirectional, first-in-first-out (FIFO), buffered channel, which behaves as an ordered queue. A stream decouples its producer and consumer in time, up to the capacity of the stream queue. Flow control handshaking may be implemented by a simple pair of producer-ready / consumer-ready signals. A stream is easy to pipeline across long distances using protocol relaying or direct wire pipelining, yet it is still capable of transmitting at full throughput (we demonstrate a pipelined stream implementation in Chapter 3). In this sense, our approach provides a system composition methodology similar to OCP [OCP, 2001].

1.4.2 Timing Independence

We impose an additional constraint on streaming to guarantee timing independent system behavior. If an input desired by a module is not available, the module must stall. Specifically, the module may not test for input readiness and decide to do something else while waiting, since that may lead to timing-dependent behavior. The resulting system is a *process network* with *blocking read*, a formalism which has been proven to have deterministic behavior regardless of timing and scheduling [Kahn, 1974]. Thus, system correctness is ensured regardless of placement distance, stream pipelining, or stream implementation. Furthermore, system correctness is ensured regardless of any modification to the timing behavior of individual modules. A module may be pipelined or otherwise optimized, changing its timing behavior, without affecting system correctness or requiring modification of other modules. Thus, we support better modular design.

1.4.3 Language Integration

Requiring the module designer to conform to a stream protocol is laborious and error prone. Instead, we provide language support and automation for streaming. We turn the stream into an abstraction, expose it as an object in the syntax of a hardware description language, and implement it through synthesis. A compiler can then automatically generate handshaking signals and stall control for each module. Automatic generation of communication saves labor and is correct by construction. It also ensures conformance to the streaming discipline and the process network model. Thus, it enables compiler analyses and optimizations that might not be practical or permissible on an unrestricted RTL description.

1.4.4 Automatic Module Optimization

Since stream communication is robust to timing, a compiler is free to implement optimizations that change the timing behavior of a module. Such optimizations may include pipelining, rescheduling, area-time transformations, or changes of granularity (module decomposition/merging). We present a number of specific module optimizations throughout this document. In Chapter 3, we discuss a simple, automatic approach to module pipelining (applicable not only in our language but to any RTL module that conforms to our stream protocol, since it is based on retiming registers from streams). In Chapter 5, we discuss automatically implementing non-critical modules with serial arithmetic. In Chapter 6, we discuss automatically decomposing a large module into a collection of smaller, stream connected modules.

1.4.5 Automatic System Optimization

The stream topology of a system exposes its communication dependencies, and thus its available parallelism. A compiler can use this information to implement global optimizations such as stream pipelining, stream queue sizing, area-time tradeoffs, and throughput-aware placement. Such optimizations would not be possible in RTL based design, where system level information is limited, and changes in timing behavior are not tolerated. An RTL description binds a particular level of parallelism into its timed description, and it does not provide enough information for a compiler to make reliable, large scale changes. A streaming design has more information and more freedom. We present a number of specific, system-level optimizations in Chapter 5.

1.4.6 Reuse and Retargetting

The timing independent nature of streaming, as defined above, naturally supports better modular design and module reuse. Whereas an RTL module interface is specified in terms of timing behavior, a streaming module interface cares only about relative order of tokens. Consequently, a streaming module works in more contexts. A streaming module will continue to work on a next generation device that has different interconnect delay, where streams are pipelined deeper. More interestingly, a streaming module will work even if the transport implementation is changed, be it point-to-point wires, shared bus, or even shared memory. Modules may be optimized, independently or in concert, to better match the different timing and resources of a new device. Language integration can help automate the retargeting process by applying system level optimizations and regenerating both modules and streams. Thus, a streaming discipline with language integration supports entire system reuse, not just module reuse.

1.4.7 Platform Support

It is possible to provide specialized resources for streaming on a programmable platform. A streaming platform might provide a network on chip (NOC) for stream transport, custom stream queues, specialized programmable logic for stream handshaking, and streaming memory controllers. Any of these resources may be improved in a next-generation device, provided their stream interface is retained. Additionally, streaming modules may be implemented in specialized computational resources, such as MACs (multiply-accumulate), FFTs (fast Fourier transform), or microprocessor cores, provided they are equipped with streaming interfaces. Chip I/O may be incorporated in special, streaming nodes, which translate from the on-chip stream protocol to standard, off-chip protocols. Thus, a stream discipline can be the foundation for a complete system on a chip (SOC) or a programmable platform, including heterogeneous components and off-chip communication. We dedicate Chapter 6 to a discussion of streaming programmable platforms and techniques for compiling to use their structured resources.

1.4.8 Extension: Area Abstraction

With proper architectural support, streaming is a key enabler for *area abstraction*, which further supports design reuse and retargeting. We envision a hardware virtualization scheme, as in SCORE (Stream Computations Organized for Reconfigurable Execution) [Caspi *et al.*, 2000a] [Markovskiy *et al.*, 2002], whereby an arbitrarily large computation graph is partitioned and mapped onto a small, reconfigurable device as a sequence of configurations. Buffered streams are used to decouple connected modules that are not loaded in the same configuration. If inter-configuration stream buffers are large, then each configuration can run for a long time. The resulting reduction in reconfiguration frequency allows a system to tolerate and amortize long reconfiguration delays, which are typically thousands of cycles. In contrast, non-streaming virtual platforms such as WASMII [Ling and Amano, 1993] and TMFPGA [Trimberger *et al.*, 1997] must reconfigure at every cycle, or at best every few cycles, which requires larger, specialized reconfiguration hardware and more power.

The reconfigurable fabric on a SCORE device is partitioned into fixed size slices, termed *compute pages*, communicating through a streaming network on chip. The use of compute pages facilitates scheduling and placement by providing translatable, medium granularity, units of virtualization—much like virtual memory pages. A major challenge in compiling to SCORE is partitioning the original, stream connected modules into stream connected pages, while minimizing performance overhead and area fragmentation. We discuss techniques for page partitioning in Chapter 6.

1.4.9 Unbounded Stream Buffering

Our streaming model, based on process networks, allows producers and consumers to decide dynamically when to read and write to streams. This approach enables a natural description of applications having dynamic communication rates or I/O sizes, such as compression/decompression. We promote the notion that intermediate data should be encoded and buffered in streams where ever possible (as opposed to being explicitly stored in shared memory or ad hoc structures—even if they are accessed through streams). This approach makes inter-module dependencies visible to the compiler, enabling more system level analysis and optimization. The actual implementation of stream buffers may be chosen by a compiler to match the resources of the target device. A small stream buffer might be implemented in registers, whereas a large stream buffer might be implemented in memory.

Supporting dynamic rate stream access has the unfortunate consequence that programs may be specified which require unbounded buffering to function correctly. For example, a module may ignore one of its inputs for a long time, causing tokens to build up arbitrarily deep in the stream buffer. In a practical implementation with finite buffering, such programs may deadlock. This problem is not unfamiliar to RTL system designers, who in building systems with dynamic communication, must manually specify buffer implementations and sizes. In most real-world applications, dynamic buffering requirements are limited by compression formats, communication protocols, or some other constraint. Consequently, a designer can usually determine static buffer bounds by inspection or simulation.

If buffering is restricted to be in streams, then we can provide some automation for determining buffer sizes. In Chapter 5, we propose an analysis for determining minimum, deadlock-free sizes for stream buffers. If the analysis fails, a compiler can inform the designer and ask for explicit sizes on particular buffers. If buffer-related deadlock is still a concern, the implementation may detect it at runtime. A dynamically reconfigurable streaming platform such as SCORE [Caspi *et al.*, 2000a] [Markovskiy *et al.*, 2002] can dynamically reallocate buffers at run-time in the hopes of making additional progress after a deadlock.

1.5 Related Streaming Models and Architectures

The stream based design methodology proposed above is most effective for application domains that lend themselves naturally to stream decomposition. These so-called *streaming applications* are plentiful in the world of digital signal processing, media processing (audio/video), and communication. Streaming applications are typified by:

- large, sequentially accessed data structures (data streams),
- a (mostly) persistent computation structure,
- limited and infrequent reconfiguration or mode changes, and
- limited dataflow feedback.

Such applications can often be decomposed into a series of transformations on streams, which suggests efficient implementation in hardware as a pipeline of independent modules. Feedback is typically limited to be within modules, or infrequent between modules, so that concurrency exists to evaluate many modules simultaneously. The reliance on mostly sequential data access patterns means that implementations need not be bottlenecked by large, random access memories.

A large variety of models, languages, and architectures exist to support streaming applications. We review some of them here. Additional discussion of related models is presented in Chapter 2, with a more theoretical perspective.

1.5.1 Statically Schedulable Dataflow

A number of existing languages and synthesis engines are based on restricted but statically schedulable streaming models, such as synchronous dataflow (SDF) [Lee and Messerschmitt, 1987b] [Bhattacharyya *et al.*, 1996] and cyclo-static dataflow (CSDF) [Bilsen *et al.*, 1996]. Examples include SDF to VHDL synthesis in Ptolemy [Williamson, 1998], synthesis of synchronous circuits from Simulink [Simulink, 2005], and StreamIt for RAW [Gordon *et al.*, 2002a]. In these models, individual processes or modules repeat a *firing* step, or a loop of firing steps in CSDF, with each firing restricted to consume and produce a static number of tokens. The upside is that program behavior becomes fully analyzable, so that resources and schedules can be set at compile time, and simple transforms can parallelize the program to the target hardware (*e.g.* horizontal and vertical graph restructuring in StreamIt). The downside is that such programs cannot represent dynamics, so they are inherently incomplete. Dynamic communication rates can only be represented by building protocols out of streams (*e.g.* coupling a data stream with a presence stream), and statically sized buffers may need to be allocated as special nodes. Such manual structures are cumbersome, inefficient in implementation, difficult to analyze, and obstructive to retargeting. Alternatively, coarse grained dynamics can be added by using a meta-language to sequence a collection of statically schedulable kernels. However, the dataflow between kernels may be obfuscated in the meta-language, and system level analysis again becomes difficult or impossible. For these reasons, our streaming solution is based on a dynamic dataflow model, whose stylized dynamics are amenable to compiler analysis.

1.5.2 Dynamic Dataflow

A number of models and languages exist that support dynamic flow rates on streams. Boolean controlled dataflow (BDF) [Buck, 1993], integer controlled dataflow (IDF) [Buck, 1994], and dataflow process networks (DFPN) [Lee and Parks, 1995] are dynamic rate, process network models using functional, *i.e.* stateless processes. In these models, state exists only in stream contents. This modeling style is well suited for signal processing applications, where signal flow graphs are commonplace. However, it is largely foreign to hardware designers, who are used to the expressive power of finite state machines (FSMs) and data registers. Process state may be represented as feedback streams on functional processes, but this only obfuscates state from designers and synthesis tools. It is better to treat process state differently than data streams, since state never requires queueing, and it may be optimized by a large body of existing work in sequential synthesis (*e.g.* SIS [Sentovich *et al.*, 1992]). We are not aware of any efforts to map these stateless process models directly to hardware.

CAL [Eker and Janneck, 2003] and YAPI [de Kock *et al.*, 2000] [Stefanov *et al.*, 2004] are extended process network models that support process state. Our own model closely resembles CAL, with both models defining process behavior as actions guarded by process state and input presence. Both CAL and YAPI permit non-deterministic behavior which may depend on the arrival time of stream inputs. CAL does so by permitting a specification of priorities among actions, to resolve the case when multiple actions are simultaneously enabled. YAPI does so using a *probe* construct, which allows testing and reacting to the absence of inputs. In contrast, our model retains determinism by disallowing input testing and by constraining multiple enabled actions (as detailed in Chapter 2). These restriction ensure that behavior is faithfully retained across device generations and different implementations of stream interconnect.

The main challenge with dynamic dataflow models is that stream buffer requirements may be unbounded, and knowing whether they are unbounded is undecidable. Thus, it may be impossible for a compiler to allocate sufficient buffering resources at compile time without creating a possibility of run-time deadlock. Processor based implementations of dynamic dataflow can overcome this limitation by reallocating buffers at run-time, but hardware implementations generally cannot. Certain restricted forms of dynamic dataflow are guaranteed to need only bounded buffering, including well behaved dataflow graphs [Gao *et al.*, 1992], BDF graphs with bounded length schedules [Buck, 1993], and CHP in static token form [Teifel and Manohar, 2004]. Our streaming approach avoids such restrictions to improve expressiveness, so it may require unbounded buffering. To bound buffers for a hardware implementation, we propose in Chapter 5 a compile-time analysis of buffer bounds based on state space enumeration. If the analysis fails to bound buffers, then we rely on a designer to bound them explicitly.

1.5.3 Simulink

A number of efforts have appeared for compiling from Simulink [Simulink, 2005], a graphical design and simulation environment, to hardware, particularly to FPGAs. Examples include Synplify DSP from Synplicity [Synplicity, 2005a], System Generator from Xilinx [Xilinx, 2005a], DSP Builder from Altera [Altera, 2005], and the BEE design environment [BWRC, 2005]. A program in Simulink is a hierarchical graph of computational nodes, where nodes process in continuous time (*e.g.* integrals) or discrete time (*e.g.* FIR filters). A library of node generators is usually provided, which generates efficient hardware implementations of non-trivial tasks such as filters or FFTs. System composition is structural and timed, so that signal connections become wires, and ideal delays become registers. This structural interpretation is unfortunate, since the subset of allowed, synthesizable behaviors is usually equivalent to SDF, a highly optimizable model (or more generally, equivalent to a collection of SDF regions with unrelated clocks, which may be optimized separately). In principle, it should be possible to apply system level optimizations such as pipelining or parallelizing by node replication. Even so, those synthesizable behaviors suffer from the same inability to express dynamics as SDF, which makes them limited for specifying complete systems.

1.5.4 Imperative Languages

A number of efforts have appeared for synthesizing streaming applications to hardware from imperative languages such as C (without pointers) or Java. Synthesis from a single thread of control faces a difficult challenge of extracting parallelism. Most single-thread approaches are limited to exploiting data parallelism by unrolling and mapping loops, so their performance is limited (*e.g.* Garp C compiler [Callahan and Wawrzynek, 2000], DEFACTO [Diniz *et al.*, 2001], Xax [Snider, 2002]). Compaan [Turjan *et al.*, 2005] is unique in its ability to decompose Matlab code into a network of multiple, stream connected processes, so it can exploit a higher degree of concurrency.

Other efforts at synthesizing imperative languages make task level concurrency explicit, accepting a specification of multiple communicating threads. The most basic usage model merely uses the host language syntax and libraries to specify RTL behavior (*e.g.* SystemC 1.0 [OSCI, 2000], Handel-C from Celoxica [Handel-C, 2005]). More abstract approaches use the untimed semantics of communicating sequential processes (CSP) [Hoare, 1985], where inter-process connections are two-party rendezvous (*e.g.* Streams-C [Gokhale *et al.*, 2000], Sea Cucumber [Jackson *et al.*, 2003]). Typically, rendezvous is treated as a primitive from which more sophisticated mechanisms are built, such as queues or signals. Other approaches provide multiple communication abstractions, such as queues, stacks, and shared memory, and can automatically generate communication implementations (*e.g.* System C 2.1 [OSCI, 2005], Catapult C from Mentor Graphics [Catapult C, 2005]). These approaches provide useful automation for module design, but in a sense, they provide *too much freedom* at the system level. The availability of multiple mechanisms for communication is intended to promote efficient implementations, but that is a moniker for manual design choices. The resulting lack of a unifying design discipline or semantic model makes system level analyses difficult, and it ultimately obstructs retargeting, reuse, and scaling on next generation devices. Even when a design discipline is enforced, the semantic model is usually non-deterministic and hard to analyze.

In contrast, we propose synthesizing a more disciplined, minimalist model of streaming. By restricting the definition of streams, we enforce a strong semantic model (process networks) with well understood properties. This streaming discipline may be embedded in many possible languages, including the ones discussed above. For example, one may restrict Streams-C to use buffered streams without signals, or SystemC 2.0 to use only FIFO queues without a capacity test (thus enforcing blocking read). Programs thus restricted would become amenable to stream specific optimizations. Our particular choice of a process network interpretation of streaming is motivated by multiple factors, including its ability to deal with interconnect delay, and its facility in handling both static rate (*e.g.* signal processing) and dynamic rate (*e.g.* compression) problems. We revisit semantic models and our particular choices in Chapter 2.

1.5.5 Streaming Architectures

A variety of architectures exist for efficient implementation of streaming multimedia applications. Examples include Cheops [Bove, Jr. and Watlington, 1995], Philips video signal processor (VSP) [de Kock, 1999], Pleiades [Wan *et al.*, 2001], Imagine [Khailany *et al.*, 2001], TRIPs [Sankaralingam *et al.*, 2003], and Cell [Kahle *et al.*, 2005]. These architectures get their performance from several key features: (1) architectural support for data streams, including efficient, sequential memory access and high throughput transfer between modules; (2) concurrent evaluation of multiple processes (though not necessarily full concurrency—some processes may be time-shared in a multi-context module such as a microprocessor); and (3) domain specific features in the modules, such as MACs (multiply-accumulate) for signal processing. Efficient sequential transfers and multi-processing are orthogonal to domain specificity, and they are present in many general purpose concurrent architectures. It has been shown that streaming can be an efficient way to use multi-processor and concurrent architectures, including Monsoon [Shaw *et al.*, 1996], MIT RAW [Gordon *et al.*, 2002b] [Gordon *et al.*, 2002a], and Garp [Callahan *et al.*, 2000].

Unfortunately, the programming models for such architectures are often ad hoc, built from an arbitrary collection of communication and concurrency mechanisms such as message passing, and lacking a unifying semantic model[2]. This is the same

[2]Philips VSP [de Kock, 1999] is a refreshing example of a streaming processor whose programming model is based on a well founded, formal model of streaming, namely CSDF.

problem cited above for synthesis from C—too much freedom in the programming model. In such cases, compiler optimizations are limited to traditional, local optimizations within modules, while system optimization and correctness are left to the programmer. In some cases, strong semantic models exist for individual contexts of a subsystem but not for the control thread that sequences contextst in the subsystem. Once a program is manually partitioned and scheduled as a set of contexts, its dataflow is obscured from tools. Here again, the lack of a unifying model for total system behavior obstructs retargeting, reuse, and scaling on a next generation variant of the architecture. To address these shortcomings, we propose in Chapter 6 a set of programmable, reconfigurable architectures built expressly around our streaming discipline as a unified programming model.

1.6 Contribution and Organization

Our key contributions in this research work are as follows:

1. We present a streaming model for hardware design that, by abstracting communication timing, promotes better modular design, module reuse, and graceful handling of long interconnect delays. We provide both operational and denotational semantics (Chapter 2).

2. We present a concrete language *TDF* based on the streaming model, resembling a hardware description language with built-in streams (Chapter 2).

3. We implement a complete synthesis methodology for mapping TDF to a commercial FPGA, including automatic generation of streams and stream handshaking, as well as automatic pipelining of modules and stream interconnect (Chapter 3). We generate Verilog (RTL) as an intermediate form and rely on a

commercial back-end to map it to the FPGA. In this way, we exploit existing, state-of-the-art RTL tools.

4. We characterize the area and performance impact of stream support on an FPGA, using the automatic synthesis methodology and seven multimedia applications written in TDF (Chapter 4).

5. We propose several automatic, system-level analyses and optimizations for streaming systems, including automatic selection of stream buffer sizes; selection of stream pipelining depths; stream aware, throughput-optimized placement; and serialization of non-critical modules for area reduction (Chapter 5).

6. We propose several streaming programmable platforms that support our streaming model more efficiently than an FPGA, and that enable better system reuse and scaling. We extend our synthesis methodology to target these platforms, including partitioning for paged platforms such as SCORE (Chapter 6).

We believe that the streaming methodology proposed herein, or something resembling it, is necessary as a next generation of design abstraction for digital systems. Our methodology addresses a number of shortcomings of the prevalent design abstractions, which are based on RTL and system composition through standard interfaces. Our methodology can be viewed as a layer of abstraction on top of RTL, mapping to it and benefiting from its existing body of compilation techniques and software.

Chapter 2

Language and Theoretical Foundation

This chapter presents the details of our model and language for streaming computation. The term "streaming" has many meanings in the literature, each having different expressive power, implied run-time support, and consequences for compilation tools. Our specific model and language are motivated by an intent to implement computations as digital circuits and by the associated technology trends such as growing interconnect delay. These considerations lead us to choose hardware-centric definitions for streams, modules, and system composition.

We begin this chapter with an overview of existing streaming models and their respective design considerations (Section 2.1). We then present a new model for hardware-centric streaming, TDF Process Networks (TDFPN), and give its operational semantics (Section 2.2). We present TDF (*Task Description Format*), a language for streaming applications based on TDFPN (Section 2.3). Finally, we present denotational semantics for TDFPN, which formally capture the meaning of a complete system composition (Section 2.4).

The model and language described herein were originally developed for SCORE (*Stream Computations Organized for Reconfigurable Execution*), targeting a dynamically reconfigurable architecture [Caspi *et al.*, 2000a]. We distinguish our model from SCORE, since we consider only single context execution ("fully spatial" execution in SCORE terms). Thus the operational semantics of TDF process networks are a subset of those of SCORE. We describe SCORE as a logical extension of TDF process networks in Chapter 6 on streaming platforms. The TDF language and its use in SCORE are fully detailed in [Caspi, 2005].

2.1 Streaming Models

Streaming and dataflow based design environments abound today. A short and highly incomplete list of examples might include graphical design environments such as Simulink [Simulink, 2005], Ptolemy [Ptolemy, 2005], SPW [SPW, 2005], System Studio [Synopsis, 2005] (formerly COSSAP), and LabVIEW [LabVIEW, 2005], as well as text based languages such as Streams-C [Gokhale *et al.*, 2000], StreamIt [Gordon *et al.*, 2002a], Silage [Hilfinger, 1985], Sisal [McGraw *et al.*, 1985], Lucid [Ashcroft *et al.*, 1995], and Haskell [Jones, 2003]. Additional examples may be found in [Johnston *et al.*, 2004] and [Najjar *et al.*, 1999]. The use of visual or textual syntax is largely irrelevant to a program's meaning, so we shall refer to both forms as languages. All these languages share a superficial similarity in defining an application as a composition of stream-connected components. Components are also termed *actors*, *processes*, *nodes*, or *operators*. Stream connections expose the communication dependencies between components, and thus their available concurrency, enabling tools to produce efficient system implementations. However, these approaches offer wildly different definitions of streams, stream access methods, component structure, notion of time, *etc.* Consequently, they differ in expressive power, determinism of application behavior, amenability to compiler analysis, and runtime resource requirements. We take the pragmatic approach that a programming model should provide appropriate abstractions for the underlying implementation. In this research work, we are concerned with the implementation of computation as digital circuits in Silicon, where interconnect delay is increasing with each device generation. Consequently, we seek a streaming model that can abstract communication timing while retaining deterministic behavior.

2.1.1 Definitions of the Stream

Streaming languages differ, first and foremost, in their definition of the stream. That definition binds the meaning of system composition and, in turn, important system properties such as determinism and analyzability of deadlock. The general notion of a stream as a connection is insufficient, because it fails to specify important properties of the connection such as:

- Is it directional?
- Does it permit single or multiple writers?
- Does it permit single or multiple readers?
- Does it represent data values in time?
- Does it represent data values in order? (*e.g.* a FIFO queue does, a Petri net *place* [Murata, 1989] does not)
- Does it allow data values to be replicated or over over-written in transit? (*e.g.* a PGM *graph variable* [Kaplan and Stevens, 1995] allows replication and over-writing, like a register; a Streams-C *signal* [Gokhale, 2003] allows over-writing but not replication; a Kahn process network *channel* [Kahn, 1974] allows neither)

Lee and Sangiovanni-Vincentelli [Lee and Sangiovanni-Vincentelli, 1998] categorize many streaming languages into *models of computation* based on the language's definition of streams, or in their terms, of *signals*. Their models of computation include *continuous time*, where a signal is a function from time to value; *discrete event*, where a signal is set of (time, value) pairs; *synchronous reactive*, where a signal is a set of (quantized time, value) pairs; and *process network*, where a signal is an ordered sequence of values (*i.e.* a queue of tokens). These definitions of signals lead to a family of denotational semantics for proving properties about streaming models. We will use their categorizations for discussion here and their denotational framework later.

2.1.2 Notion of Time

Most of Lee and Sangiovanni-Vincentelli's models include an explicit notion of time, implying synchrony among all components. The synchrony is logical, and need not be absolute in the implementation. Nevertheless, its existence in the model would constrain any rescheduling in a parallel hardware implementation, where actors evaluate concurrently. Synchrony may become obstructive in the presence of long interconnect delay, where waiting for round trip delays would reduce performance.

The process network model is attractive for our purposes, because it abstracts away time and allows asynchrony among actors. Asynchrony is useful for dealing with long interconnect delay, so a stream consumer may be scheduled to run arbitrarily later than the stream producer. Process networks also tolerate unpredictability or changes in the timing and implementation of streams and actors. This timing independence is useful for porting designs to next generation devices, where the timing of actors and streams may be different. Another useful feature of process networks is that the queue definition of streams naturally accommodates computations where the amount of data transmitted between actors may vary dynamically, *e.g.* compression/decompression.

2.1.3 Stream Testing

A process network model, using queues for streams, may or may not allow *testing stream readiness*. Extended process network models such as YAPI [de Kock *et al.*, 2000] permit an actor to test whether an input stream has data available and to take action accordingly. This approach is useful for real-time reactive systems that deal with an unreliable environment, *e.g.* skipping a video frame when samples are lost. However, the resulting timing dependent behavior is non-deterministic and usually difficult to debug. Traditional process network models such as Kahn process networks (KPN) [Kahn, 1974] disallow testing stream readiness. The resulting semantics, termed *blocking read*, require that, once an actor issues a stream read, it will block until the requested data is available. Kahn proved that this simple restriction is sufficient to guarantee deterministic behavior regardless of timing or scheduling (assuming there is no randomness within individual processes).

2.1.4 Dynamic Consumption / Production Rates

Streaming languages may differ in the allowed *rate* of stream access. Kahn processes may choose to consume or produce data at any time, even based on dynamic, data-dependent decisions. For example, a process may choose to ignore certain input streams, causing tokens to build up arbitrarily deep in those streams. Consequently, Kahn process networks may require unbounded buffering for streams and may deadlock if implemented with buffers that are too small. Whether or not a particular network will deadlock is undecidable. This follows from the Turing completeness of KPN (one may build a Turing Machine in KPN using streams to implement tapes [Buck, 1993]). Deciding whether a Turing complete network will deadlock is equivalent to deciding whether it will halt, which is known to be intractable. Thus, correct execution of KPN in general requires dynamic scheduling and memory allocation.

Parks [Parks, 1995] proposes the approach of reallocating full buffers to be larger whenever a buffer related deadlock occurs. Clearly, this is impractical in a hardware implementation.

Alternatively, process networks may be restricted to avoid dynamic consumption/production rates and thus regain the decidability of memory bounds and of deadlock avoidance. In Synchronous dataflow (SDF) [Lee and Messerschmitt, 1987b], each actor is restricted to consume and produce tokens in fixed ratios, a policy enforced by associating an actor's consumption and production with an atomic *firing* step. In Cyclo-static dataflow, each actor is restricted to consume and produce tokens in a repeating pattern of fixed ratio firings. The static nature of these models permits static analyses for liveness (deadlock avoidance), memory bounds, and firing schedule. Statically schedulable models enable highly optimized mappings to arbitrary hardware, including microprocessors [Bhattacharyya *et al.*, 1996], digital circuits [Williamson, 1998], and multi-processor arrays [Gordon *et al.*, 2002a].

Interestingly, statically schedulable models may be extended into Turing Complete or non-deterministic models by the mere addition of certain canonical actors. Boolean controlled dataflow (BDF) [Buck, 1993] extends SDF with *switch* and *select* token routers, which are powerful enough to implement if-then-else decision making. Similarly, extending SDF with a *non-deterministic merge* yields a non-deterministic, undecidable model [Lee and Neuendorffer, 2005].

2.1.5 Actor Elaboration

Stream languages may allow actors to elaborate themselves into streaming subgraphs containing other actors. This elaboration is a streaming analogue of a function call, where one actor temporarily invokes another, or invokes a group of others. Static elaboration is available in most languages and corresponds simply to a hierarchy in the stream connection graph. More interesting is dynamic elaboration, which is supported in the Kahn-MacQueen language using coroutines [Kahn and MacQueen, 1977]. Dynamic elaboration is a highly expressive mechanism, but it may make a model hard to analyze or entirely undecidable. A restricted form of dynamic elaboration is available in Heterochronous dataflow (HDF) [Girault *et al.*, 1999], which is a statically schedulable model. Each HDF actor elaborates into a finite state machine; each state elaborates into an HDF subgraph; and so on in alternating, finite recursion. Each HDF state corresponds to a fixed consumption/production ratio, which is respected by the graph elaborated from that state. Although different states may have different ratios, the total state space is countably finite, so it corresponds to a finite number of fully-elaborated synchronous dataflow graphs.

Dynamic elaboration is not practical in a single context hardware implementation, such as an FPGA. Elaboration based on finite recursion (*e.g.* HDF) may be statically unrolled into hardware, but the resulting implementation will have many idle components. Efficient dynamic elaboration requires a time shared implementation, *e.g.* using microprocessors or dynamic reconfiguration.

2.2 TDF Process Networks

We now develop a particular streaming model suitable for implementing applications as digital circuits. In this section, we define the model and outline its operational semantics. In later sections, we will describe a language instance of the model and its denotational semantics.

2.2.1 Key Features

Determinism is a cornerstone for our model. A model is *determinate* or *deterministic* if its behavior is determined entirely by its inputs. From a technology standpoint, we desire determinism regardless of timing, so that an application will be robust to new timing characteristics when ported to a next generation device. Changes that may be appropriate when retargeting to a new device include replacing the stream transport with a new on-chip network; pipelining actors; implementing time-space trade-offs to match architectural throughput bottlenecks; and restructuring actors to match an architecturally favorable granularity. Without determinism, these changes are likely to modify system behavior and introduce bugs. With a deterministic model, a compiler is free to automatically implement any of those changes without affecting system behavior. Determinism also guarantees equivalence between hardware behavior and simulated behavior, regardless of timing and scheduling in the simulator. Thus, it enables debugging to be done in highly efficient, functional simulators.

We define the stream to be a FIFO queue as in process networks, *i.e.* a unidirectional, single producer, single consumer, first-in-first-out, buffered channel. Such a stream is an untimed abstraction of an inter-module wire, carrying a sequence of tokens from producer to consumer. We define stream access to use blocking read, so as to guarantee determinism regardless of the implementation and timing of streams and actors. An application's actors may be instantiated in hardware to run concurrently and persistently, driven by availability of input data and output space.

We wish for actors to support dynamic consumption/production rates, to naturally support compression/decompression operations. As a consequence, our model must incorporate stream buffering and allow for unbounded buffer sizes. A practical hardware implementation has only finite buffering resources and cannot support dynamic reallocation of buffers. Thus, we choose static buffer sizes for the implementation, and we accept that some programs with unbounded buffer requirements may fail. In practice, most applications have bounded requirements, even if those requirements appear to be data dependent. We will rely on compiler analysis and programmer annotation to choose static buffer sizes.

Actors should be defined in a manner familiar to hardware designers. Thus, we structure actor definitions around finite state machines (FSMs). In principle, it is possible to use functional actor definitions and add state in the form of feedback loops. However, this approach is less convenient for hardware design, and it will likely yield a poorer implementation than an FSM-based design optimized by sequential synthesis. A conventional FSM is a synchronous circuit, which evaluates inputs and sets outputs on every cycle of a clock signal. We extend that notion into a *streaming finite state machine* (SFSM), which *fires* only when input tokens and output space are available. Each state may specify which inputs and outputs are desired, comprising a state-specific *firing guard*. While the guard is unsatisfied, the actor blocks, *i.e.* stalls in the same state. When the guard is satisfied, the actor fires, *i.e.* consumes input tokens, evaluates a state-specific action, and produces output tokens.

2.2.2 SFSM Definition

We begin with a simplified definition of an SFSM, presented in analogy to an FSM, and extend the definition in Sections 2.2.3 and 2.2.4. An additional restriction is discussed in Section 2.4.4.

A conventional FSM (Mealy type) may be defined as:

$$\text{FSM} = (\Sigma, \sigma_0, \sigma, I, O, f_{NS}, f_O) \tag{2.1}$$

where:

- Σ is a set of states,
- σ_0 is the start state,
- σ is the present state,
- I is the space of input values (an M-tuple of bits),
- O is the space of output values (an N-tuple of bits),
- $f_{NS} : \Sigma \times I \rightarrow \Sigma$ is the *next state* function, and
- $f_O : \Sigma \times I \rightarrow O$ is the *output* function.

At each clock cycle, the FSM samples inputs $i \in I$, emits outputs $o = f_O(\sigma, i)$, and transitions to a next state $\sigma' = f_{NS}(\sigma, i)$. A system is a composition of FSMs via wire connections, and all FSMs step synchronously.

To extend the above model into an SFSM, we replace I and O with tuples of sequences, rather than tuples of values. A firing guard will look ahead into the input sequences to determine whether desired inputs are available. The output function will produce output sequences.

We use a notation for sequences based on [Lee and Parks, 1995] and [Lee, 1997]. A sequence $s \in S$ with $S = T^\omega$ is a possibly infinite, ordered list of values (tokens) of type T (ω being the first infinite ordinal). A specific sequence $s \in S$ is denoted by a bracketed list of values $s = [v_1, v_2, ...]$, with $v_i \in T$. Let $s.s'$ denote a concatenation of sequences s, s'. Let $s \sqsubseteq s'$ denote that s is a prefix of s' (*i.e.* $s' = s.s''$). For tuples of sequences, using boldface, let $\boldsymbol{s} \in S^M$ denote an M-tuple, let $\boldsymbol{s}.\boldsymbol{s'}$ denote a pointwise concatenation, and let $\boldsymbol{s} \sqsubseteq \boldsymbol{s'}$ denote a pointwise prefix. For simplicity, we will assume that all inputs and outputs of an SFSM carry values of the same type S.

An SFSM may be operationally defined as:

$$\text{SFSM} = (\Sigma, \sigma_0, \sigma, I, O, R, f_{NS}, f_O) \tag{2.2}$$

where:

- Σ is a set of states,
- σ_0 is the start state,
- σ is the present state,
- $I \subseteq S^M$ is the space of input sequences (an M-tuple of sequences),
- $O \subseteq S^N$ is the space of output sequences (an N-tuple of sequences),
- $R : \Sigma \rightarrow I$ is a collection of *firing rules*, which specify for each state a tuple of desired input sequences,
- $f_{NS} : \Sigma \times I \rightarrow \Sigma$ is the *next state* function, and
- $f_O : \Sigma \times I \rightarrow O$ is the *output* function, producing output sequences.

An SFSM evaluates arbitrarily often. At a given evaluation, an SFSM's input queues contain subsequences $\boldsymbol{i} \in I$, and its output queues contain subsequences $\boldsymbol{o} \in O$. The SFSM fires if and only if: (1) the present state's desired inputs are available, *i.e.* $R(\sigma) \sqsubseteq \boldsymbol{i}$, or equivalently $\boldsymbol{i} = \boldsymbol{i'}.R(\sigma)$, and (2) the present state's outputs $f_O(\sigma, R(\sigma))$ would fit in the output queues without overflow (in the abstract model with infinite buffering, test (2) would always be true). Upon firing, the state of the SFSM and queues are updated such that $\sigma' = f_{NS}(\sigma, R(\sigma))$, $\boldsymbol{o'} = \boldsymbol{o}.f_O(\sigma, R(\sigma))$, and $\boldsymbol{i'}$ is defined by $\boldsymbol{i} = \boldsymbol{i'}.R(\sigma)$. Otherwise, the SFSMs and queues retain their present state. A system is a composition of SFSMs and queues having a collective state including queue contents. A system behavior is an arbitrary interleaving of SFSM firings.

In a practical implementation, an SFSM and a queue may be able to exchange at most one token per firing. In such a case, every component sequence in R and f_{NS} would be empty or unit length, of the form $[v]$. Nevertheless, it is possible to accommodate longer input and output sequences in several ways, including (1) directly, using queues with windowed read/write, (2) by widening a stream into several parallel streams, or (3) by transforming multi-token states into sequences of single-token states.

2.2.3 Extending the Blocking Read

The SFSM model described above uses a multi-stream blocking read. In each state, the SFSM issues a blocking read simultaneously to every stream specified in $R(\sigma)$. This is a simple generalization of the Kahn-MacQueen blocking read [Kahn and MacQueen, 1977], which reads and blocks on a single stream at a time. Still, it suffers from the same non-compositionality as the Kahn-MacQueen model. A composition of two processes may not be expressible as a single process, since a single process cannot decouple the reads of the original processes. Consider, for example, the parallel composition of two identity processes, each of which copies input to output one token at a time. A process representing their composition could issue a blocking read to either of the original input streams, or to both inputs at once, but either approach couples the execution of the two processes. The single process cannot simulate, using blocking read, the original behavior of independently evaluating two processes. Likewise, the Kahn-MacQueen blocking read prevents expressing certain transformations on an SFSM, including certain decompositions into communicating sub-machines, and pipelining.

To regain compositionality, we extend the blocking read with *multiple firing rules*. Each state will be guarded to match *one or more* allowable input patterns. Each such pattern, or firing rule, may be associated with a different behavior. Thus, the above composition of two identities may issue a read that succeeds on the presence of the first input *or* second input, then emit a token to the first or second output accordingly. The SFSM model in equation (2.2) must be modified such that:

- $R \subseteq \Sigma \times I$ is a set of *firing rules*, which specify for each state one or more tuples of desired input sequences.

The SFSM operational semantics are modified such that the SFSM fires in state σ if *any* of that state's firing rules are satisfied, *i.e.* if $\exists \boldsymbol{r} \in R$ s.t. $\boldsymbol{r} = (\sigma, \boldsymbol{\rho})$ and $\boldsymbol{\rho} \sqsubseteq \boldsymbol{i}$.

Unfortunately, multiple firing rules open the door to non-deterministic behavior. A process may now be sensitized to the order of arrival of its inputs, choosing a different next state depending on that order. To retain determinism, we need a strategic restriction on the structure of firing rules, output functions, and next-state functions. We defer a complete discussion of such a restriction to Section 2.4.4, where we develop denotational semantics to capture the *meaning* of a computation. Part of the solution comes from the fact that a process is not really sensitized to the order of arrival, but rather to the order of consumption. If both inputs are available, the SFSM must choose which one to consume, and that choice is an arbitrary part of the blocking read implementation. While this may seem like a source of hidden non-determinism, it in fact prevents a programmer from writing one form of timing dependent behavior. In contrast, CAL [Eker and Janneck, 2003] allows multiple firing rules to be explicitly prioritized by the programmer, representing intentional timing dependence.

2.2.4 Restricted Value Matching

The definition $R \subseteq \Sigma \times I$ implies that a firing rule may match against a particular value of input, not just the presence of input. Value matching, or pattern matching, is possible in dataflow process networks (DFPN) [Lee and Parks, 1995], which are defined similarly, and in functional languages with streams such as Haskell and SML. However, our implementation of TDFPN permits only restricted value matching. The data type for each stream is extended with an end-of-stream value `eos`, and a firing rule may match against only two classes of values: `eos` or non-`eos`. We can redefine R as follows. For a given stream data type T, where $S = T^\omega$ and $I = S^M$, let $T^* = \{*, \texttt{eos}\}$, let $S^* = (T^*)^\omega$, and let $I^* = (S^*)^M$. The symbol "$*$" shall mean any non-`eos` value, and a rule matching "$*$" shall be shorthand for a collection of rules matching each non-`eos` value in the position of "$*$". The SFSM model in equation (2.2) must be modified such that:

- $R \subseteq \Sigma \times I^*$ is a set of *firing rules*, which specify for each state one or more tuples of desired input sequences.

Disallowing value matching simplifies firing guards in a circuit implementation. A guard that checks for values involves comparators, whereas a guard that checks only for presence involves testing one bit. In our synthesis methodology (Chapter 3), `eos` is encoded as an extra data bit, so matching `eos` involves testing only one bit. Disallowing value matching forces certain actors to be expressed differently. Specifically, it may require stateless actors to be re-expressed with state. For example, consider the canonical *select(s,t,f)* actor from BDF [Buck, 1993], which selectively passes a token from either input t or f to the output, depending on the value of the boolean control token from input s. *select* with value matching is stateless:

$$\begin{aligned}
&\boldsymbol{r_{st}} = (\sigma_0, [\mathtt{T}], [*], \bot), \quad && f_O(\sigma_0, [\mathtt{T}], [v_t], \bot) = [v_t], \quad && f_{NS}(\sigma_0, [\mathtt{T}], [v_t], \bot) = \sigma_0 \\
&\boldsymbol{r_{sf}} = (\sigma_0, [\mathtt{F}], \bot, [*]), \quad && f_O(\sigma_0, [\mathtt{F}], \bot, [v_f]) = [v_f], \quad && f_{NS}(\sigma_0, [\mathtt{F}], \bot, [v_f]) = \sigma_0
\end{aligned}$$

In contrast, *select* without value matching requires state. Input s is matched only for presence, and its value is used to branch to a state that reads t or to another state that reads f:

$$\begin{aligned}
&\boldsymbol{r_s} = (\sigma_s, [*], \bot, \bot), \quad && f_O(\sigma_s, [v_s], \bot, \bot) = [\bot], \quad && f_{NS}(\sigma_s, [v_s], \bot, \bot) = v_s ? \sigma_t : \sigma_f \\
&\boldsymbol{r_t} = (\sigma_t, \bot, [*], \bot), \quad && f_O(\sigma_t, \bot, [v_t], \bot) = [v_t], \quad && f_{NS}(\sigma_t, \bot, [v_t], \bot) = \sigma_s \\
&\boldsymbol{r_f} = (\sigma_f, \bot, \bot, [*]), \quad && f_O(\sigma_f, \bot, \bot, [v_f]) = [v_t], \quad && f_{NS}(\sigma_f, \bot, \bot, [v_f]) = \sigma_s
\end{aligned}$$

The stateful implementation of *select* emits output only every other firing. If the circuit implementation equates a firing with a clock cycle, then the stateful implementation operates at only half throughput. Nevertheless, it can be pipelined to full throughput by adding rules that read t or f simultaneously with the next s.

2.3 TDF Language

TDF (*Task Description Format*) is a language for designing streaming applications targeting hardware. The language is a fairly direct implementation of the process network model described in the previous section. In essence, TDF is a minimal hardware description language with streaming features. This level of abstraction is well suited for synthesis yet is still amenable to stream-related optimizations. The language is intentionally small, omitting many high level features such as aggregate data structures (records), type inheritance, and structured loops. Instead, we view TDF as an intermediate level language that might be compiled from a higher level language such as Streams-C [Gokhale *et al.*, 2000].

This section provides a brief overview of TDF. A more complete definition of the language is available in [Caspi, 2005].

2.3.1 TDF Overview

TDF has two levels of design: the *behavioral* language is used to specify SFSM actors, and the *compositional* language is used to instantiate and connect actors via streams. Compositions are themselves actors and may be instantiated within other compositions. Thus, a TDF application is a hierarchical graph of stream connected behavioral actors.

2.3.1.1 Types

TDF scalar types include booleans and integers. Integers are signed or unsigned bit vectors with explicit bit width, *e.g.* `unsigned[9]`. A TDF array, or memory segment, is a list of identically typed scalars. A TDF stream is a queue of identically typed scalars. The type of a computed expression may be inferred automatically from the types of its components, *e.g.* `a+b` has type `unsigned[9]` if `a` and `b` have type `unsigned[8]`.

2.3.1.2 Actor Prototypes

Each actor is declared with a prototype that lists its name, its streams, and its instantiation parameters. For example, a zero-length encoder might be declared as:

```
zle (input unsigned[7] i, output unsigned[8] o) {...}
```

2.3.1.3 Instantiation Parameters (`param`)

An actor may have instantiation parameters, which are inputs whose value is bound once, when the actor is instantiated. This is in contrast to stream inputs, whose value changes at run-time. Instantiation parameters may be used to specialize an actor with a particular initialization value, operating mode, or bit width. For example, a zero-length encoder with parameterized data width might be declared as:

```
zle (param unsigned[8] w,
     input unsigned[w] i, output unsigned[w+1] o) {...}
```

Parameter values are propagated to every use, and the uses undergo partial evaluation at compile time. Thus, in the example above, a compiler would generate a specialized datapath having the specified bit width. A compositional actor may pass its instantiation parameters to actors that it instantiates, and so on down the compositional hierarchy.

2.3.2 TDF SFSMs

The heart of TDF is the behavioral language used to specify SFSMs (*streaming finite state machines*). An SFSM is an extended finite state machine where each state has a specification of desired inputs and an action. All input and output occurs through streams. An SFSM may also have register variables to store local, persistent data state.

2.3.2.1 SFSM Firing

Each state includes an *input signature*, which is a comma separated list of input streams to wait for. On entry into a state, the SFSM issues blocking reads to the inputs specified in the present state's input signature. The SFSM then waits to be *enabled* by (1) the presence of tokens on the input streams and (2) the availability of space of the output streams. Once enabled, the SFSM may *fire*, *i.e.* consume the present state's inputs and evaluate the state's action. The process then repeats for the next state, or for the same state if no state transition is specified. We will say interchangeably that an SFSM, a state, or a signature is enabled if (1) and (2) are satisfied for the present state.

2.3.2.2 State actions

A state action is a loop-free sequence of C-like statements, evaluated when a state fires. An action may compute new values from input and register values, update register variables, produce output tokens, and transition to a next state via `goto`.

An if-then-else provides control for simple decision making. Temporary variables may be declared inside nested statement blocks to name intermediate values. The C preprocessor may be used for substitution macros.

2.3.2.3 Stream access

We use the convention from Silage [Hilfinger, 1985] that the name of an input stream in expressions refers to its most recently consumed value. The stream's history of values may be referenced using the `@` operator, such that `i@0` is the most recently consumed value (same as `i`), `i@1` is the previously consumed value, and so on. An assignment to an output stream denotes emitting an output token, *e.g.* `o=1`. Thus, a 4-tap FIR filter might be expressed as a single state SFSM with an action: `y = C0*x + C1*x@1 + C2*x@2 + C3*x@3`.

2.3.2.4 Example: ZLE

Figure 2.1 lists sample TDF for a zero-length encoder (ZLE) SFSM. A similar encoder is used in JPEG and MPEG encoding following quantization and zig-zag scan. The ZLE's job is to recognize runs of zero valued input tokens and to compress them into code words specifying the run length. Non-zero input tokens are passed untouched to the output stream. Symbolic states are useful for recording whether or not the encoder is presently in a run of zeros. State `start` is the normal state that passes non-zero tokens and looks for the first zero of a run. State `zeros` counts zero tokens in a run, using register `cnt`, until the run is terminated by consuming a non-zero or by reaching a maximum allowable run length. The state then emits a token specifying the run length. If a run is terminated by consuming a non-zero, then that non-zero is passed to the output in state `pending` before returning to state `start`.

```
#define ZERO_MASK 0b10000000    /* 128, mask for run length codewords */
#define MAX_ZEROS 0b01111111    /* 127, maximum allowable run length  */

zle (input unsigned[7] i, output unsigned[8] o)
{
  unsigned[8] cnt=0;

  state start (i) :
    if (i==0)             { cnt=1; goto zeros; }
    else                  { o=i; }

  state zeros (i) :
    if (i==0) {
      if (cnt==MAX_ZEROS) { o=ZERO_MASK|cnt; cnt=0; }
      else                { cnt=cnt+1; }
    }
    else                  { o=ZERO_MASK|cnt; cnt=0; goto pending; }

  state pending () :
    o=i;
    goto start;
}
```

Figure 2.1: Zero length encoder SFSM in TDF

2.3.2.5 Synthesizable Timing Model

The synthesis methodology of Chapter 3 equates an SFSM state with an RTL behavioral state, evaluating a state action in one cycle. The implementation of streams in Chapter 3 permits only one token to be consumed or produced on a stream per cycle. To reflect this timing model and rate restriction, the zero length encoder's production of the first non-zero following a run of zeros is implemented in a separate state `pending`, rather than emitting a second token in state `zero`. It is important to note that this timing model and rate restriction are not inherent to the TDF language and, in principle, need not be exposed to the TDF programmer. There is nothing preventing a compiler from rescheduling an SFSM by moving statements, creating new states, or applying loop scheduling techniques such as software pipelining. Such transformations would be useful for making an SFSM satisfy synthesis restrictions and for improving performance in general. The ZLE can be pipelined to avoid the cycle lost in state `pending` even with the restriction of one token per stream per firing. The production in `pending` should be moved to a modified state `start'` that reads a new input but emits output associated with a *prior* firing. The output would "catch up" to the input at the next zero input, where the original `start` emits nothing, and `start'` emits its prior output.

2.3.2.6 Termination

Streams are terminated by a special end-of-stream token. Termination of an output stream may be specified using the `close` statement, *e.g* `close(o)`. Termination of an input stream may be caught in an input signature, *e.g.* `state zeros(eos(i))`. A state may have multiple input signatures to specify a different action for receiving data than for receiving end-of-stream. This is a simple form of pattern matching built into the input signature, and it is deterministic, since only one signature would match the inputs. For example, the zero length encoder state `zeros(i)` could be augmented with a second signature `zeros(eos(i))` to properly terminate a run of zeros if the input stream closes. An SFSM may transition to a built-in state `done` to indicate no further action. If a state with no explicit end-of-stream handling tries to consume data from a closed input stream, the default behavior is to close all output streams and to terminate the SFSM. Thus, end-of-stream will be propagated to the rest of the application for proper termination.

2.3.2.7 Calls and Exceptions

A state action may include an inline call to another actor. Such a call is syntactic sugar for exchanging a set of tokens, one per stream, with a private instance of the called actor. In the spirit of a function call, the caller sends tokens, then waits for return tokens before proceeding. The call may be implemented by connecting the two actors via streams, in which case the calling state must be split into a sending state and a receiving state. Otherwise, the call may be implemented by inlining the called actor's code into the caller. A call syntax is particularly useful for handling exceptions, since the caller SFSM must wait for return tokens before proceeding. While waiting, the SFSM produces no other outputs and consumes no other inputs, thus stalling any dependent parts of the application until the exception is resolved. Since TDF behavior is deterministic regardless of timing, this mechanism provides a consistent and resumable way to halt a distributed application for exception handling.

2.3.2.8 Multiple Enabled Signatures

TDF syntax allows writing multiple cases for a state, each having a different input signature and action. If the signatures reference the same streams and differ only in matching data versus end-of-stream, then only one signature can match at a time, and behavior will be fully deterministic. If signatures reference different streams, then more than one may match at a time, *e.g.* `state foo (x): X; state foo (y): Y;`. This syntax is useful for expressing certain transformations on SFSMs where blocking read is insufficient, including composition, decomposition, and pipelining. Which signature is chosen for firing may depend on timing, since inputs need need not arrive at the same time. If the inputs of several signatures are all available, then an arbitrary choice must be made. TDF *does not* specify which signature will match if more than one signature is simultaneously enabled. If the actions of the multiple signatures do not properly match, then the resulting behavior will be non-deterministic. Unfortunately, it is difficult to express a minimal but sufficient restriction on how the actions should match in order to guarantee determinism. Our design philosophy is to prevent a programmer from using multiple enabled signatures, but to allow the compiler to generate them during mapping and optimization, where determinism is guaranteed by construction.

Consider, for example, an SFSM state that needs inputs `x` and `y` but which, for reasons of performance, wants to make forward progress even if only one input is available. The SFSM might be written as follows:

```
state foo  (x)   : X;    goto gety;
state foo  (y)   :    Y; goto getx;
state foo  (x,y) : X; Y; goto next;
state getx (x)   : X;    goto next;
state gety (y)   :    Y; goto next;
```

The resulting behavior will be deterministic if the effects of statements `X` and `Y` are independent of which one is evaluated first. For example, `X` and `Y` must not write incompatible values to the same register or output stream. It should be possible to construct a compiler analysis to recognize whether such multi-signature code is deterministic. The corresponding semantic restrictions are discussed in Section 2.4.4.

Now consider an SFSM that has been decomposed by clustering states into separate, communicating sub-machines. Only one sub-machine is active at a time, and it may transfer control to another sub-machine by sending a control token. When inactive, a sub-machine enters a wait state of the form:

```
state wait (m1) : goto return_from_m1;
state wait (m2) : goto return_from_m2;
```

Streams `m1` and `m2` denote a return of control from different, external sub-machines. The states that resume control, `return_from_m1` and `return_from_m2`, are unrelated, so we cannot expect that their effect on registers and streams will be independent of order. Furthermore, the waiting sub-machine cannot know which return token to expect first, since the other sub-machines may transfer control amongst themselves. The compiler analysis proposed above for order independence would seem to indicate non-deterministic behavior. Nevertheless, there is no question of order: there is only one control token among all the sub-machines, so no more than one signature will be enabled at a time. Here, determinism is guaranteed by construction.

2.3.3 TDF Composition

A TDF *compositional actor* contains statements to instantiate and connect other actors via streams. Hierarchical compositions are supported by instantiating and connecting other compositional actors. However, recursive compositions are not supported.

2.3.3.1 Instantiation and Connection

In the compositional language, a stream in an expression refers to the entire stream, not to an individual value. With this interpretation, we use conventional assignment

```
filter1 (input signed[16] i, output signed[16] o) {...}

filter2 (input signed[16] i, output signed[16] o) {...}

filter  (input signed[16] i, output signed[16] o)
{
  signed[16] t;
  filter1(i,t);
  filter2(t,o);
}
```

Figure 2.2: Composition of two filters in TDF

```
signed[16] funfilter1 (input signed[16] i) {...}

signed[16] funfilter2 (input signed[16] i) {...}

signed[16] funfilter  (input signed[16] i)
{
  funfilter = funfilter2(funfilter1(i));
}
```

Figure 2.3: Composition of two filters in TDF, functional form

statements and calls to build and connect a stream graph. A function call is used to instantiate an actor, *i.e.* add it to the graph, and the call arguments indicate stream connections. Assignment to a stream also indicates a connection. Feedback loops in the stream graph are perfectly legal. Intermediate streams may be declared inside a compositional actor as named local variables, with an optional specification of stream buffer size and initial contents[1].

[1]Initial stream contents are not presently supported in the synthesis methodology of Chapter 3 targeting FPGAs. They would also be difficult to support in an ASIC, where the startup value of storage cells is usually undefined. Instead, initial stream contents can be implemented by adding an initial state to the stream producer that injects initial values into the stream.

2.3.3.2 Example: Filter Composition

Figure 2.2 shows an example composition of two filters in series. An intermediate stream `t` is first declared. A call to each filter is used to instantiate and connect it, with `t` feeding the output of one filter into the input of the next. Figure 2.3 shows an alternate, functional syntax for composition. Each actor is declared to have a *return stream*, which is a stylized output stream. A call to the actor is then not only an instantiation, but an expression referring to the actor's return stream, which may be used syntactically like any other named stream. An actor's own return stream has the same name as the actor itself, and it may be used syntactically like any output stream. Composition in TDF using return streams resembles composition in a functional language like Haskell or SML. However, it is not truly functional, since the called actors may have local state. This syntax is merely being used to build a stream graph.

2.3.3.3 Fanout and Fanin

A TDF stream may have only one producer and one consumer. Fanning out a stream to several consumers requires an intermediate actor, implementing a token copy. TDF has a built-in `copy` actor to serve that purpose. The actor is polymorphic with respect to the type and number of streams connected to it, requiring only that the connected streams be compatible, *e.g.* `copy(i,o1,o2,o3)`. TDF also supports automatic inferencing of a copy operator for any stream that appears to be referenced by multiple consumers. Merging a stream from several producers also requires an intermediate actor, implementing an arbitrator. TDF has no built-in arbitrators. However, it has a built-in `cat` actor that merges streams by concatenating the bits of corresponding input tokens into a wider output token.

2.3.3.4 Memory Segments

TDF avoids the traditional mechanism of unified, shared memory, since shared memory obfuscates dependencies and often becomes a performance bottleneck. Instead, TDF supports separate, local memories, termed *segments*. A segment is associated with a particular actor that owns it. The segment may be declared inside that actor or passed to it as a `param` argument. TDF provides two ways to access a segment: as an array in an SFSM, or via streams. An SFSM can directly access an array variable using array subscripting. The array memory is treated as part of the SFSM's datapath, and the access latency is tolerable for relatively small arrays.

In general, we wish to abstract the timing of memory access and to improve its performance by pipelining and exploiting sequential access patterns. We also wish to decouple address generation from data consumption where possible, to create feed-forward flows. These goals are easily met in a streaming paradigm by accessing memories through streams. TDF has a family of built-in actors, termed *segment operators*, which provide a stream interface and an address generator for a segment, using one of the following access modes:

- Sequential read (source) — with a stream for data out,
- Sequential write (sink) — with a stream for data in,
- Sequential read/write (FIFO) — with streams for data in and out,
- Random access read — with streams for address in and data out,
- Random access write — with streams for address in and data in,
- Random access read/write — with streams for address in, read/write mode in, and data in and out.

Segment operators hide the architecture-specific implementation and timing of memory access. Segment operators may be instantiated and connected in a compositional actor using the usual syntax.

2.3.4 Generating TDF from a Higher Level Language

The TDF language is small but easy to synthesize. Hence, it is a suitable target for compiling from a higher level language. Features missing in TDF which would be useful in a higher level streaming language include:

- *Aggregate data types (records)* for streams and variables.
- *Type inheritance.* Ptolemy [Ptolemy, 2005] makes its actors polymorphic by defining a hierarchy of subsumption for stream types: a boolean is an integer, which is a float, which is a double. It is permissible to connect a producer of integers to a consumer of floats. TDF has a limited form of subsumption, permitting type upgrades on integers (bit vectors) from narrow to wide and from unsigned to signed.
- *Arithmetic expressions on streams* as a shorthand for composing synchronous dataflow graphs. *E.g.* `a=b+c` would denote a streaming adder.
- *Composition generators.* TDF presently has no way to build an FIR filter with a parameterized number of taps. A generator language with iteration or recursion would address this need, *e.g.* the `for` loop in Verilog. The iteration would be partially evaluated at compile time to form the stream composition.
- *Structured loops.* SFSMs with `goto` are "spaghetti code"—hard to read and hard to optimize. A higher level actor language should include structured loops such as "for," "do-while," or "repeat-until." Such loops are amenable to traditional optimization (strength reduction, lifting invariants) and loop scheduling (loop unrolling, software pipelining).

Imperative languages with stream read/write have emerged as a popular way to express streaming applications, *e.g.* Streams-C [Gokhale *et al.*, 2000], YAPI [de Kock *et al.*, 2000], TinySHIM [Edwards and Tardieu, 2005], and Catapult C [Catapult C, 2005]. The actors of such languages are relatively easy to translate into TDF SFSMs.

The key observation is that an SFSM state is akin to a basic block with a stream read header[2]. Consequently, an imperative language actor might be translated into TDF as follows:

1. Form basic blocks, forcing a new block at every stream read.
2. Optionally collapse if-then-else flows by repeatedly merging single-entry, read-free blocks into their predecessor.
3. Convert every such block into an SFSM state.

The above translation does not include loop optimizations and loop scheduling. Translating loops into TDF can benefit from many existing compiler techniques, particularly those of vectorizing and parallelizing compilers. A loop over an array is analogous to a streaming actor, where array accesses are linearized onto streams. A series of loops communicating through intermediate arrays is then analogous to a pipeline of stream connected actors. Compaan [Turjan *et al.*, 2005], which translates from Matlab to YAPI, includes a polytope-based analysis of array access patterns, so it can convert intermediate arrays into streams. Loop factoring is analogous to splitting an actor into a series composition of two actors. Loop fusion is analogous to merging a series composition of two actors into one. Such transformations are useful for managing the depth of streaming pipelines. Loop unrolling and vectorization are analogous to splitting an actor into data-parallel actors (*i.e.* vector lanes). Such transformations could be used to unfold a computation to match an available amount of hardware and to manage the granularity of individual actors, guided by an architectural area and timing model.

[2]An SFSM state action has loop free control flow and a DAG for local dataflow. A state action may include if-then-else structures and nested statement blocks. Nevertheless, those if-then-else structures synthesize into multiplexers in hardware, so they indicate predication rather than true branching.

2.3.5 Implementing BDF

For completeness, we demonstrate that TDF can implement Buck's boolean controlled dataflow (BDF) [Buck, 1993]. BDF is a particularly simple form of dynamic dataflow, derived by extending synchronous dataflow (SDF) with two canonical, dynamic rate actors: *select* and *switch*. To complete the reduction, we need to implement *select*, *switch*, and SDF actors as TDF SFSMs.

The *select* actor selectively passes a token from either of two data inputs to one data output, based on a boolean select input. The *switch* actor selectively passes a token from one data input to either of two data outputs, based on a boolean select input. These actors may be implemented in TDF using state sequences, where an initial state reads the select input and branches to two possible subsequent states. Figure 2.4 shows TDF code and corresponding TDFPN firing rules and functions for the *select* actor. Figure 2.5 shows the same for *switch*. The TDF implementations assume that data streams are unsigned with parameterized bit width, but those types may be easily substituted. The multi-state implementation of *switch* is chosen for symmetry with *select*, but it can alternatively be implemented in a single state with an `if` to conditionally write to either output stream.

Theorem 2.3.1. *TDFPN can implement BDF.*

Proof. BDF consists of streams, ideal delays, SDF actors, *select* and *switch*. Streams are built into TDFPN. An ideal delay corresponds to an initial token on a stream. An SDF actor can be expressed as a single state SFSM with one firing rule and unconditional output. Implementations of *select* and *switch* are shown in Figures 2.4 and 2.5. □

Corollary 2.3.2. *TDFPN is Turing Complete.*

Proof. BDF can implement a Turing machine [Buck, 1993], and TDFPN can implement BDF. □

```
select (param  unsigned[8] w,
        input  boolean     s,
        input  unsigned[w] t, input unsigned[w] f,
        output unsigned[w] o)
{
  state S (s) : if (s) goto T; else goto F;
  state T (t) : o=t;   goto S;
  state F (f) : o=f;   goto S;
}
```

$$\begin{array}{lll} \boldsymbol{r_s} = (\sigma_s, [*], \bot, \bot), & f_O(\sigma_s, [v_s], \bot, \bot) = \bot, & f_{NS}(\sigma_s, [v_s], \bot, \bot) = v_s ? \sigma_t : \sigma_f \\ \boldsymbol{r_t} = (\sigma_t, \bot, [*], \bot), & f_O(\sigma_t, \bot, [v_t], \bot) = [v_t], & f_{NS}(\sigma_t, \bot, [v_t], \bot) = \sigma_s \\ \boldsymbol{r_f} = (\sigma_f, \bot, \bot, [*]), & f_O(\sigma_f, \bot, \bot, [v_f]) = [v_t], & f_{NS}(\sigma_f, \bot, \bot, [v_f]) = \sigma_s \end{array}$$

Figure 2.4: TDF and TDFPN implementations of BDF *select* actor

```
switch (param  unsigned[8] w,
        input  boolean     s,
        input  unsigned[w] i,
        output unsigned[w] t, output unsigned[w] f)
{
  state S (s) : if (s) goto T; else goto F;
  state T (i) : t=i;   goto S;
  state F (i) : f=i;   goto S;
}
```

$$\begin{array}{lll} \boldsymbol{r_s} = (\sigma_s, [*], \bot), & f_O(\sigma_s, [v_s], \bot) = (\bot, \bot), & f_{NS}(\sigma_s, [v_s], \bot) = v_s ? \sigma_t : \sigma_f \\ \boldsymbol{r_t} = (\sigma_t, \bot, [*]), & f_O(\sigma_t, \bot, [v_t]) = ([v_t], \bot), & f_{NS}(\sigma_t, \bot, [v_t]) = \sigma_s \\ \boldsymbol{r_f} = (\sigma_f, \bot, [*]), & f_O(\sigma_f, \bot, [v_f]) = (\bot, [v_f]), & f_{NS}(\sigma_f, \bot, [v_f]) = \sigma_s \end{array}$$

Figure 2.5: TDF and TDFPN implementations of BDF *switch* actor

2.4 TDFPN Denotational Semantics

Whereas operational semantics deal with the execution of a program, denotational semantics deal with its underlying meaning. It is easy to reason about the meaning of an individual actor, but what is the meaning of an entire composition? What is its intended behavior? Is that behavior determinate and unique? A major difficulty of traditional parallel programming is that the meaning or intended behavior of a program is not obvious, lost in a myriad of synchronization primitives and implementation details. Streaming models, on the other hand, have been more amenable to formal modeling of denotational semantics [Kahn, 1974] [Lee and Sangiovanni-Vincentelli, 1998]. With those mechanisms, it becomes possible to prove important properties of a model or of a particular program, including whether it is determinate, will deadlock, or will run in bounded memory. Those properties translate into compiler analyses, optimizations, and synthesis methods that yield more efficient implementations.

In this section, we outline the denotational semantics for TDF process networks (TDFPN). We show that TDFPN is a special case of Lee's dataflow process networks (DFPN) and thus inherits important properties such as semantic equivalence (full matching between operational and denotational semantics), determinacy, and undecidability. We begin by reviewing mathematical mechanisms (Section 2.4.1) and DFPN denotational semantics (Section 2.4.2), then develop TDFPN denotational semantics (Sections 2.4.3, 2.4.4).

2.4.1 Fixed Point Semantics for Streaming

We adopt the *tagged signal* denotational model from [Lee and Sangiovanni-Vincentelli, 1998]. In this model, the meaning or behavior of a streaming system is captured by its communication traces, or *signals*, *i.e.* by the total history of communication on every inter-actor connection. In this view, an actor is a function F from its input traces to its output traces. Note that those traces may be infinite, such as an infinite sequence of tokens. An actor in this view is intrinsically functional, or stateless. Nevertheless, a stateful actor may be rewritten as a stateless actor by exposing its state as a feedback stream. The behavior of a system is then the solution to a system of equations relating all streams and actors, *e.g.* $Y = F(X)$, $Z = G(Y)$. A determinate system would have only one solution for a given set of inputs, so we need a more specific definition to distinguish among possibly many solutions. For mathematical convenience, we recast the system as a single function $\boldsymbol{F}$, composed of all actor functions, and recast the system of equations to be:

$$\boldsymbol{X} = \boldsymbol{F}(\boldsymbol{X}, \boldsymbol{I}) \tag{2.3}$$

where $\boldsymbol{I}$ is the tuple of input traces, and $\boldsymbol{X}$ is the tuple of intermediate and output traces. The behavior of a system may now be defined as the *least fixed point* solution to equation (2.3). The existence and uniqueness of a solution would be guaranteed by a family of fixed point theorems if the communication traces were defined in an appropriate topology and if the actor functions were continuous.

The Scott topology deals specifically with sequences. Its use in denotational semantics of process networks was introduced by Scott [Scott, 1970] and Kahn [Kahn, 1974], and it receives an excellent summary in [Lee, 1997]. We review it here for completeness. The key observation is that sequences with a prefix order comprise a complete partial order (CPO), where a fixed point theorem applies.

A *partial order* is a relation $\sqsubseteq$ on a set S that is:

- *reflexive* : $s \sqsubseteq s$,
- *transitive* : $(s \sqsubseteq s') \wedge (s' \sqsubseteq s'') \implies (s \sqsubseteq s'')$,
- *antisymmetric* : $(s \sqsubseteq s') \wedge (s' \sqsubseteq s) \implies (s = s')$,

for all $s, s', s'' \in S$. The order is partial, not total, since there may be elements $s, s' \in S$ that are unrelated: $s \not\sqsubseteq s'$, $s' \not\sqsubseteq s$. A set with a partial order relation is a *partially ordered set* or *poset*. Sequences with the *prefix* order form a poset, where $s \sqsubseteq s'$ means that s is a prefix of s', or equivalently that $s' = s.s''$ for some $s'' \in S$. Let $s.s''$ denote a sequence concatenation, and let $[v_1, v_2, v_3, ...]$ denote a particular sequence. Let $\lambda = []$ denote the empty sequence, which is a *bottom* element for the poset: $\lambda \sqsubseteq s \quad \forall s \in S$. Bottom is traditionally denoted by $\perp$. An *upper bound* u of two sequences $s, s' \in S$ is a common extension: $s \sqsubseteq u$, $s' \sqsubseteq u$. A *least upper bound* or *join* of two sequences, $s \sqcup s'$, is an upper bound that is a prefix of any other upper bound. A *lower bound* l of two sequences $s, s' \in S$ is a common prefix: $l \sqsubseteq s$, $l \sqsubseteq s'$. A *greatest lower bound* or *meet* of two sequences, $s \sqcap s'$, is a lower bound of which any other lower bound is a prefix. A *chain* $C \subseteq S$ is a possibly infinite, ordered list of increasing sequences $(s_1 \sqsubseteq s_2 \sqsubseteq ...)$. A *complete partial order* (CPO) is a poset with a bottom, where every increasing chain $C \subseteq S$ has a least upper bound $\sqcup C \in S$. Sequences with the prefix order and empty sequence as bottom form a CPO. Here, an increasing chain represents the repeated extension of a sequence through computation. Tuples of sequences with a pointwise prefix also form a CPO, so all our results extend naturally to tuples. Using boldface for tuples, let $\boldsymbol{s} \in S^M$ denote an M-tuple of sequences, let $\Lambda = \boldsymbol{\perp}$ denote a tuple of empty sequences (bottom for tuples), let $\boldsymbol{s}.\boldsymbol{s'}$ denote a pointwise concatenation, let $\boldsymbol{s} \sqsubseteq \boldsymbol{s'}$ denote a pointwise prefix, let $\boldsymbol{s} \sqcup \boldsymbol{s'}$ denote a pointwise join, let $\boldsymbol{s} \sqcap \boldsymbol{s'}$ denote a pointwise meet, and define $\boldsymbol{s} \in S^M$ to be *finite* whenever all its component sequences are finite.

A *monotonic* function $f : S \to S$ on a CPO is one where:

$$(s \sqsubseteq s') \implies (F(s) \sqsubseteq F(s')) \tag{2.4}$$

For sequences (and tuples thereof), this definition means that whenever the input to a monotonic function is extended (from s to s'), the output is also extended, or stays the same. A monotonic function on sequences is *causal* in the sense that when input is added, the function cannot change its mind and modify old output, it can only add to the existing output. This is an appropriate, practical limitation for computable functions.

A fixed point theorem for CPOs states that a solution to $X = F(X)$ always exists when F is monotonic. Thus, a system of monotonic actors will have a fixed point behavior in these denotational semantics. However, that behavior may not be unique, meaning the system may not be determinate.

A *continuous* function $f : S \to S$ on a CPO is one where, for any increasing chain $C \subseteq S$,

$$F(\sqcup C) = \sqcup F(C) \tag{2.5}$$

where $F(C)$ refers to the set (chain) inferred by applying F to each element of C. A continuous function is monotonic, as evidenced by considering any chain of two elements. The interesting part of this definition applies to infinite chains. For sequences (and tuples thereof), it means that the output of a continuous function with infinite input ($F(\sqcup C)$) is the same as its limit output with finite subsequences of that input ($\sqcup F(C)$). Again, this is an appropriate, practical limitation of computable functions, which provides that a function need not wait for its infinity of inputs before emitting output. In fact, it must not wait.

A fixed point theorem for CPOs states that a solution to $X = F(X)$ always exists and is unique when F is continuous. Thus, a system of continuous actors is determinate. Furthermore, the theorem states that the solution can be constructed by repeated applications of F to the bottom element:

$$X = F(F(...F(\bot)...)) \tag{2.6}$$

This construction suggests an operational semantics for evaluating process networks by repeatedly evaluating every actor on intermediate inputs and results, building up from initially empty streams.

Kahn [Kahn, 1974] describes a rudimentary actor language based on blocking reads and proves that the resulting actors are continuous. As a consequence, Kahn process networks are determinate. Kahn and MacQueen [Kahn and MacQueen, 1977] provide a more complete language for the same model, retaining continuity.

2.4.2 Dataflow Process Networks (DFPN)

The naive operational semantics suggested by equation (2.6) are not always practical. Continuity absolves an implementation from having to atomically process infinite inputs but not infinite outputs. If an actor function F is specified to emit an infinite output sequence, then (2.6) specifies that it must do so before proceeding to any other action. An implementation with these semantics may require infinite buffering for the output, and it may force other actors to stall indefinitely. A more practical operational semantics would instead produce finite output sequences in a series of steppings, perhaps one token at a time. These are the semantics associated with *firing*, as in synchronous dataflow (SDF) [Lee and Messerschmitt, 1987b], cyclo-static dataflow (CSDF) [Bilsen *et al.*, 1996], and boolean controlled dataflow (BDF) [Buck, 1993]. Unfortunately, any discrepancy between the operational and denotational semantics makes program analysis and optimization more difficult.

Lee [Lee, 1997] defines a streaming model, dataflow process networks (DFPN), where both the operational and denotational semantics incorporate firing. At each firing, a DFPN actor waits to be *enabled* by any of several finite input patterns, called *firing rules*, then consumes those inputs and emits finite outputs. Lee defines a dataflow actor from M inputs to N outputs to be a pair $A = (f, R)$, where

1. $R \subseteq S^M$ is a set of finite sequences, called *firing rules*, denoting possible input patterns,
2. $f : R \to S^N$ is a firing function, denoting the actor's reaction to each input pattern,
3. $f(\boldsymbol{r})$ is finite for all $\boldsymbol{r} \in R$, and
4. no two distinct $\boldsymbol{r}, \boldsymbol{r'} \in R$ are joinable.

Whenever a pattern $\boldsymbol{r} \in R$ appears as a prefix of the input $\boldsymbol{s}$, *i.e.* $\boldsymbol{r} \sqsubseteq \boldsymbol{s}$, the actor consumes that pattern and produces the finite output $f(\boldsymbol{r})$. The associated Kahn process, mapping infinite input sequences to infinite output sequences, is then:

$$F(\boldsymbol{s}) = \begin{cases} f(\boldsymbol{r}).F(\boldsymbol{s'}) & \text{if } \exists\, \boldsymbol{r} \in R \text{ s.t. } \boldsymbol{s} = \boldsymbol{r}.\boldsymbol{s'} \\ \Lambda & \text{otherwise} \end{cases} \tag{2.7}$$

This self-referential definition mirrors the operational semantics whereby the actor consumes inputs and produces outputs incrementally, through a series of firings. Nevertheless, the definition is mathematically well founded and denotational. Lee proves that, given (f, R), the process F exists, is unique, and is both monotonic and continuous[3]. Consequently, dataflow process networks are determinate.

2.4.3 Equivalence of TDFPN and DFPN

TDF process networks (TDFPN) and dataflow process networks (DFPN) are very similar, both based on a similar notion of firing rules. They differ in only two ma-

[3] Lee [Lee, 1997] shows that F in Equation (2.7) can be defined as the least fixed point of a functional, $F = \phi(F)$, and that the functional ϕ is continuous over a CPO of functions. The fixed point theorem for CPOs then indicates that F exists, is unique, and can be constructed by repeatedly applying ϕ to an initial function that generates empty sequences. This construction yields an operational semantics with firing.

jor respects: state and value matching. First, TDFPN actors are stateful, whereas DFPN actors are functional, or stateless. Nevertheless, state can be incorporated into a functional paradigm by exposing it as explicit an input and output of a function. Hence, this difference is in structure only, not in expressive power. Second, DFPN firing rules support arbitrary value matching, whereas TDFPN rules do not. We show that (1) TDFPN is a special case of DFPN, (2) TDFPN with one rule per state is equivalent to DFPN with sequential firing rules, and (3) TDFPN extended with arbitrary value matching is equivalent to DFPN. We can demonstrate the equivalence of the two models by transforming actors from one model to the other. The equivalence implies that TDFPN inherits all of the properties of DFPN, including its fixed point denotational semantics and determinacy.

The equivalence described herein is between DFPN and the abstract TDFPN, both with infinite buffering. TDFPN operational semantics allow for finite buffers, but that is an implementation issue. Parks [Parks, 1995] addresses the discrepancy between infinite denotational buffers and finite operational buffers with an elegant transformation on process networks. In the transformation, each stream is converted into a feedback loop carrying capacity tokens, and the stream producer is modified to wait for a capacity token before emitting to the stream. Parks' transformation applies equally well to TDFPN. We differ from his view only in having explicitly specified a capacity check in the operational semantics of SFSMs.

2.4.3.1 Equivalence Theorems

Theorem 2.4.1. *TDFPN is a special case of DFPN.*

Proof. See the reduction in Section 2.4.3.3. □

Theorem 2.4.2. *TDFPN extended with value matching is equivalent to DFPN.*

Proof. See the reductions in Sections 2.4.3.3 and 2.4.3.4. □

Theorem 2.4.3. *TDFPN with one rule per state is equivalent to DFPN with sequential firing rules.*

Proof. See the reductions in Sections 2.4.3.5 and 2.4.3.6. □

2.4.3.2 Notation

An SFSM was defined in equation (2.2) as:

$$\text{SFSM} = (\Sigma, \sigma_0, \sigma, I, O, R, f_{NS}, f_O) \tag{2.2}$$

For parallelism with DFPN notation, we assume a unified stream type S, such that $I = S^M$, $O = S^N$. We also assume that data state is rolled into FSM state, a convention commonly used in the literature of sequential synthesis. Thus, a TDFPN actor may be defined more simply as:

$$A = (\Sigma, \sigma_0, \sigma, R, f_{NS}, f_O) \tag{2.8}$$

Using the extended, multiple signature definition of R from Section 2.2.3 with the unified type S, we have: $R \subseteq \Sigma \times S$, $f_{NS} : \Sigma \times S^M \rightarrow \Sigma$, $f_O : \Sigma \times S^M \rightarrow S^N$. In comparison, a DFPN actor is defined as $A' = (f', R')$, where $R' \subseteq S^{M'}$ and $f' : R' \rightarrow S^{N'}$ (using primes to distinguish DFPN symbols from TDFPN symbols). In the reductions, we ignore end-of-stream.

2.4.3.3 Reducing TDFPN to DFPN

A TDFPN actor may be converted into a DFPN subgraph by exposing its state in a feedback loop, as in Figure 2.6. For convenience, we define projection functions on M-tuples: $\pi_i(\boldsymbol{s})$ is the i^{th} component, and $\pi_{j..k}(\boldsymbol{s})$ is the tuple of j^{th} through k^{th} components. A TDFPN actor $A = (\Sigma, \sigma_0, \sigma, R, f_{NS}, f_O)$ from M inputs to N outputs

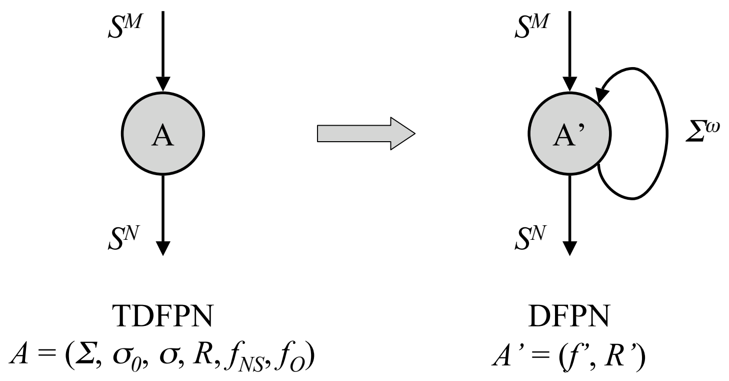


Figure 2.6: Reduction of a TDFPN actor to a DFPN subgraph, exposing state

is equivalent to a DFPN actor $A' = (f', R')$ from $M + 1$ inputs to $N + 1$ outputs, where:

- the first input and output carry state, *i.e.* the actor maps $(\Sigma^\omega \times S^M)$ to $(\Sigma^\omega \times S^N)$,
- the state input and output are tied in a feedback loop,
- the state feedback loop contains an initial token σ_0,
- $R' = R$ (see below),
- $f' : R' \to (\Sigma^\omega \times S^N)$ is defined by $f'(\boldsymbol{r'}) = (\ [f_{NS}(\pi_0(\boldsymbol{r'}))],\ f_O(\pi_{1..M}(\boldsymbol{r'}))\ )$.

The definition $R' = R$ (which is technically sloppy in its types) is shorthand for saying that we map each TDFPN firing rule $\boldsymbol{r} = (\sigma, \boldsymbol{s})$ to a DFPN firing rule $\boldsymbol{r'} = ([\sigma], \boldsymbol{s})$. The DFPN firing rule would match a single state token on its state input.

This reduction can be taken in the context of different restrictions on multiple enabled firing rules, which are discussed further in Section 2.4.4. TDFPN with one rule per state reduces to DFPN with no joinable rules TDFPN with multiple rules per state reduces to DFPN with joinable rules.

2.4.3.4 Reducing DFPN to TDFPN with value matching

A DFPN actor may be converted into a TDFPN actor, extended with value matching, having a single state. All of the DFPN actor's firing rules would associate with that state. Specifically, a DFPN actor $A' = (f', R')$ from M inputs to N outputs is equivalent to an extended TDFPN actor $A = (\Sigma, \sigma_0, \sigma, R, f_{NS}, f_O)$ from M inputs to N outputs, where:

- $\Sigma = \{\sigma_0\}$ for some dummy σ_0
- R is inferred from R' by mapping each $\boldsymbol{r'} \in R'$ to $\boldsymbol{r} = (\sigma_0, \boldsymbol{r'})$,
- $f_{NS} : \Sigma \times S^M$ is defined by $f_{NS}(\sigma_0) = \sigma_0$,
- $f_O : \Sigma \times S^M$ is defined by $f_O(\sigma_0, \boldsymbol{r'}) = f'(\boldsymbol{r'}) \;\; \forall \boldsymbol{r'} \in R'$.

This reduction, together with the reduction from TDFPN to DFPN in Section 2.4.3.3, proves the equivalence of TDFPN extended with value matching and DFPN.

2.4.3.5 Reducing DFPN with sequential firing rules to TDFPN with one rule per state

Our implementation of TDFPN lacks value matching, so the conversion of DFPN rules R' to TDFPN rules R in the previous section is not always possible. A direct conversion is possible only for DFPN rules that match "$*$" and "$\perp$". However, TDFPN can emulate more general DFPN rules by using a sequence of stream reads, value tests, and branches. Lee and Parks [Lee and Parks, 1995] formalize a restriction on DFPN firing rules, termed *sequential firing rules*, that permits this emulation. A DFPN actor $A' = (f', R')$ has sequential firing rules if the following procedure succeeds:

1. Find an input j such that $\forall \boldsymbol{r_i} \in R'$, $\exists v_i$ s.t. $[v_i] \sqsubseteq \pi_j(\boldsymbol{r_i})$. That is, find an input such that all firing rules require at least one token from that input (though each rule may require a different value v_i). If no such input exists, fail.

2. For the choice of input j, divide the firing rules into subsets, one for each specific token value v_i at the head of $\pi_j(\boldsymbol{r_i})$. If $[v_i] = [*]$, then the firing rule $\boldsymbol{r_i}$ should appear in all such subsets.

3. Remove the first element $[v_i]$ from every $\boldsymbol{r_i}$.

4. If all subsets have empty firing rules, then succeed. Otherwise, repeat these four steps for any subset with any non-empty firing rules.

This recognition procedure largely mirrors the decision tree required to evaluate sequential firing rules without using value matching. We can construct a TDFPN actor that emulates a DFPN actor with sequential firing rules as follows, in unison with the procedure above. Steps (1) and (3) infer a state that consumes one token from input j and compares it to all possible values $[v_i]$. Step (4) infers a choice between completing the match of the DFPN firing rule or branching to a next state to read another input. If the match is complete, the state should emit the corresponding DFPN output and return to the first state for matching firing rules. Otherwise, the state should branch to a next state inferred by step (2). There will be either one next state per value $[v_i]$, or a single next state if $[v_i] = [*]\ \forall i$. Each next state is built by iterating the construction on the remaining, modified firing rules. The resulting TDFPN actor has exactly one rule per state.

2.4.3.6 Reducing TDFPN with one rule per state to DFPN with sequential firing rules

A TDFPN actor with one firing rule per state can be converted into a DFPN actor with sequential firing rules by direct application of the procedure in Section 2.4.3.3. For each state σ_i, there is one TDFPN rule, $\boldsymbol{r_i} = (\sigma_i, \boldsymbol{s_i})$, inferring one DFPN rule, $\boldsymbol{r'_i} = ([\sigma_i], \boldsymbol{s_i})$, with: $\boldsymbol{s_i} \in (\{*\}^{\omega})^M$. The restriction on s_i indicates that it can only match strings of “*”, since TDFPN has no value matching. The resulting DFPN rules are sequential. They can be evaluated sequentially by first reading the state input, which is present and uniquely valued in each rule, then branching based on the value σ_i to a chain of states that read the rule’s other inputs, one “*” token at a time.

This reduction, together with its counterpart in Section 2.4.3.5 proves the equivalence of TDFPN with one rule per state and DFPN with sequential firing rules.

2.4.4 Multiple Enabled Firing Rules

The denotational semantics presented thus far (for DFPN and TDFPN) are limited to having one rule enabled at a time. Multiple enabled rules are desired for expressiveness, compositionality, and closure under certain actor transformations. We extend the denotational semantics to include multiple enabled rules.

2.4.4.1 Kahn Restriction

The continuity of a DFPN actor $A = (f, R)$ relies on the restriction that no two distinct firing rules $\boldsymbol{r}, \boldsymbol{r'} \in R$ be joinable (part (4) of the definition in Section 2.4.2). That is, no two rules have a common extension. Considering the actor’s input to be the common extension, this means that no two rules can appear on the input simultaneously. Determinism is guaranteed, because there is never a question of which rule to evaluate. We call this the *Kahn restriction*, since it is analogous to the Kahn-MacQueen blocking read—it has the same problem of preventing compositionality in the model. An actor representing the composition of two others actors must, in general, be able to simultaneously match against the input of either actor, and this requires joinable rules.

2.4.4.2 DFPN Restriction

To regain compositionality, Lee [Lee, 1997] relaxes the above restriction by allowing two firing rules to match simultaneously, provided they do not interfere. That is, both rules must fire, and the resulting output must be independent of which rule fired first. Technically, the Kahn restriction is replaced by:

5. for any $\boldsymbol{r}, \boldsymbol{r'} \in R$ that are joinable, $\boldsymbol{r} \sqcap \boldsymbol{r'} = \Lambda$ and $f(\boldsymbol{r}).f(\boldsymbol{r'}) = f(\boldsymbol{r'}).f(\boldsymbol{r})$,
6. if $\Lambda \in R$ then $f(\Lambda) = \Lambda$.

We call this the *DFPN restriction*. Recall that Λ is the tuple of empty sequences. The restriction $\boldsymbol{r} \sqcap \boldsymbol{r'} = \Lambda$ means that any two joinable (simultaneously enabled) rules must have no common prefix. If they had a common prefix, then firing one rule would consume the prefix required by the second rule and thus prevent it from firing. Forbidding a common prefix ensures that both rules can fire. The DFPN restriction is not directly applicable to TDFPN due to state[4]. We formulate an alternate version with state below.

2.4.4.3 Autmented DFPN Restriction

The DFPN restriction is not always efficient for parallel hardware implementations. If firings represent time steps in an implementation, then firing two joinable rules in sequence takes more time than firing both rules in tandem. Yet the DFPN restriction prohibits combining two joinable rules into one. To be concrete, consider two independent rules $\boldsymbol{r_1} = ([*], \bot)$, $\boldsymbol{r_2} = (\bot, [*])$. Processing both inputs requires two firings, even if both inputs are available at the same time. A more efficient implementation

[4] The DFPN restriction cannot be directly applied to a TDFPN actor reduced to a DFPN actor. In the reduction, every rule consumes a state token. Two joinable rules would have the same state and thus have a common prefix, which is technically prohibited by the DFPN restriction. The state is re-emitted in each firing, so in principle, every signature of the state could fire in turn. However, this requires remaining in the same state and so is not very useful.

might add a rule $\boldsymbol{r_{12}} = ([*],[*])$ and a function case $f(\boldsymbol{r_{12}}) = f(\boldsymbol{r_1}).f(\boldsymbol{r_2}) = f(\boldsymbol{r_2}).f(\boldsymbol{r_1})$ to process both inputs at once. This addition maintains the continuity of the process, since the output is identical regardless of which rules fire when both inputs are available. Nevertheless, it violates the DFPN restriction, since the common prefixes $\boldsymbol{r_1} \sqcap \boldsymbol{r_{12}} = \boldsymbol{r_1}$ and $\boldsymbol{r_2} \sqcap \boldsymbol{r_{12}} = \boldsymbol{r_2}$ are non-empty.

To support tandem firing of joinable rules in hardware, we desire a more general restriction on R and f. Call it the *augmented DFPN restriction.* Intuitively, we want any collection $R' \subseteq R$ of joinable firing rules having non-empty meet $\sqcap R'$ to have the property that, regardless of which rule is chosen for the first firing, the sequence of subsequent firings will generate equivalent behavior. We limit the look-ahead by requiring that the process generate the said equivalent behavior no later than the consumption of the join $\sqcup R'$, regardless of which rules fire. Consider a *maximal* set of joinable rules $R' \subseteq R$ with non-empty meet:

$$R' = \left\{ \boldsymbol{r'} \in R \mid (\sqcup R' \text{ exists}) \ \wedge \ (\sqcap R' \neq \Lambda) \ \wedge \ \neg(\exists \boldsymbol{r} \in R \setminus R' \text{ s.t. } \boldsymbol{r} \sqcup \boldsymbol{r'} \text{ exists}) \right\} \tag{2.9}$$

Let $Q_{R'}$ be the set of all sequences of rules (sequences of sequences!) whose concatenation equals $\sqcup R'$:

$$Q_{R'} = \{q \in R'^{\omega} \mid \bullet q = \sqcup R'\} \tag{2.10}$$

where $\bullet q$ is the concatenation of rules in sequence q. For any maximal R', we require:

$$\exists \boldsymbol{s} \in S^N \text{ s.t. } \forall q \in Q_{R'}, \ \bullet f(q) = \boldsymbol{s} \tag{2.11}$$

where $\boldsymbol{s}$ is the said equivalent behavior, and $f(q)$ means the sequence formed by applying f to every rule in sequence q. Note that the rules thus chained in $Q_{R'}$ need not all be in R', and in general cannot all be in R' (the non-empty common prefix $\sqcap R' \neq \Lambda$ means that the first firing of a rule from R' consumes the prefix required to fire any of the other rules from R', so the others cannot fire). The original DFPN restriction continues to hold for pairs of joinable rules having empty meet $\boldsymbol{r_1} \sqcap \boldsymbol{r_2} = \Lambda$. The augmented restriction could subsume the original restriction if we remove "with non-empty meet" ($\sqcap R' \neq \Lambda$) from the definition of R', but keeping the restrictions separate might be more efficient for compiler analysis.

2.4.4.4 TDFPN Restriction

In a TDFPN process, state is hidden. Hence, when discussing equivalent behavior for multiple enabled firing rules, it suffices to speak only of externally visible behavior. Operationally, two sequences of firings have the same externally visible behavior if they have identical consumption and production of data streams and identical final state. The sequence of intermediate states is irrelevant. We use this concept to adapt the augmented DFPN restriction for TDFPN.

We first develop new terminology to discuss sequences of firings. The notation below is in the DFPN domain, with $A = (f, R)$ being a DFPN actor reduced from an equivalent TDFPN actor. *Chaining* a sequence of firing rules means firing them in turn. Define two firing rules $\boldsymbol{r_1}, \boldsymbol{r_2} \in R$ to be *chainable* if a firing of $\boldsymbol{r_1}$ produces the state required to fire $\boldsymbol{r_2}$, *i.e.* if $\pi_0(f(\boldsymbol{r_1})) = \pi_0(\boldsymbol{r_2})$. By extension, a sequence of firing rules $[\boldsymbol{r_i}, ...]$ is *chainable* if $\pi_0(f(\boldsymbol{r_i})) = \pi_0(\boldsymbol{r_{i+1}})\ \forall i$. The input associated with chaining two rules $\boldsymbol{r_1}, \boldsymbol{r_2}$, is expressed by the *chained input* operator C:

$$\boldsymbol{r_1} \mathsf{C} \boldsymbol{r_2} = \boldsymbol{r_1}.(\perp, \pi_{1..M}(\boldsymbol{r_2})) \tag{2.12}$$

which concatenates the two rules' data inputs and takes the initial state from $\boldsymbol{r_1}$. Similarly, the output associated with chaining two rules $\boldsymbol{r_1}, \boldsymbol{r_2}$, is expressed by the *chained output* operator $\bar{\mathsf{C}}$:

$$\boldsymbol{r_1}\bar{\mathsf{C}}\boldsymbol{r_2} = (\bot, \pi_{1..N}(f(\boldsymbol{r_1}))).f(\boldsymbol{r_2}) \tag{2.13}$$

which concatenates the two rules' data outputs and takes the final state from $f(\boldsymbol{r_2})$. Chained input/output is *not* the same as concatenated input/output, since the chaining operators ignore intermediate state. C and $\bar{\mathsf{C}}$ are associative, so their application to a sequence q of chainable rules is well defined: $\mathsf{C}q$, $\bar{\mathsf{C}}q$.

The *TDFPN restriction* for multiple enabled firing rules is now formulated. Consider a TDFPN actor reduced to a DFPN actor $A = (f, R)$. Let $R' \subseteq R$ be a maximal set of joinable firing rules having non-empty meet $\sqcap R'$, as in Equation (2.9). Let $Q_{R'}$ be the set of all sequences of rules whose chaining equals $\sqcup R'$:

$$Q_{R'} = \{q \in R'^{\omega} \mid \mathsf{C}q = \sqcup R'\} \tag{2.14}$$

For any maximal R', we require:

$$\exists \boldsymbol{s} \in S^N \;\text{ s.t. } \forall q \in Q_{R'},\; \bar{\mathsf{C}}q = \boldsymbol{s}. \tag{2.15}$$

This statement captures concisely that, whenever multiple firing rules in R' are enabled, any chaining of rules that collectively consumes $\sqcup R'$ must produce the same chained output $\boldsymbol{s}$, regardless of intermediate state. Note that the rules thus chained in Q need not all be in R', and in general cannot all be in R'. This restriction is sufficient to guarantee determinacy of a TDFPN composition. Strictly speaking, it does not guarantee determinacy of the reduced DFPN composition, where state is exposed, and where different intermediate state implies multiple possible behaviors. Nevertheless, the data streams are fully determined by inputs and initial state.

The TDFPN restriction for multiple enabled rules is useful in pipelined implementations of operators. Consider a two-input actor that processes corresponding pairs of inputs using a rule $\boldsymbol{r} = (\sigma, [*], [*])$. Now consider pipelining the actor to process the two inputs staggered by one firing. The pipeline must be able to flush when inputs are unavailable, since trapping in-flight values in pipeline registers may deadlock the composition. Thus, the pipelined implementation might use three states (pipeline fill, stead state, flush) plus a feedback stream corresponding to pipeline register(s). Using inputs (state, in1, in2, pipedata) and outputs (state, out, pipedata), the firing rules and functions would be:

Pipeline fill:	$\boldsymbol{r_1} = ([\sigma_1], [*], \bot, \bot)$	$f(\boldsymbol{r_1}) = ([\sigma_2], \bot, [p])$
Steady state:	$\boldsymbol{r_2} = ([\sigma_2], [*], [*], [*])$	$f(\boldsymbol{r_2}) = ([\sigma_2], [o], [p'])$
Pipeline flush:	$\boldsymbol{r_3} = ([\sigma_2], \bot, [*], [*])$	$f(\boldsymbol{r_3}) = ([\sigma_1], [o'], \bot)$

where p, p', o, o' are functions of the input. Steady state and pipeline flush use the same state σ_2 to denote that the flush is triggered by the presence (or rather, absence) of data inputs, not by an iteration count or a termination value on the input. Rules $\boldsymbol{r_2}, \boldsymbol{r_3}$ have a non-empty common prefix, $\boldsymbol{r_2} \sqcap \boldsymbol{r_3} = \boldsymbol{r_3}$, which violates the DFPN restriction. However, our choice of $\boldsymbol{r_2} = \boldsymbol{r_1} \mathsf{C} \boldsymbol{r_3}$ and $f(\boldsymbol{r_2}) = \boldsymbol{r_1} \bar{\mathsf{C}} \boldsymbol{r_3}$ satisfies the TDFPN restriction and maintains continuity of the associated Kahn process F.

2.4.4.5 Fairness

The denotational semantics above say nothing of fairness or priority among multiple enabled rules. For instance, it is correct for an implementation of the pipelined example above to never use its steady state rule $\boldsymbol{r_2}$, and instead to only use fill $\boldsymbol{r_1}$ and flush $\boldsymbol{r_3}$ in alternation. Our denotational semantics are formulated to guarantee determinacy regardless of the choice and order among multiple enabled rules. We leave that choice, and its consequences to performance, to the operational semantics. The operational semantics may be tuned for performance or resources, but they will retain determinism. In contrast, models such as CAL [Eker and Janneck, 2003] permit explicit priorities among multiple enabled rules. Such priorities create timing dependent behavior and thus destroy determinacy. They provide a useful, expressive mechanism for the programmer, but they undermine a significant benefit of the denotational semantics.

2.4.5 Adding Non-Determinism

Deterministic behaviors comprise only a limited subset of possible, desired behaviors for real systems. For example, a system that is reactive in time may need to take a different action depending on when an event occurs, such as skipping a lost frame of video. Lee and Parks [Lee and Parks, 1995] note that non-determinism may be added to Kahn process networks by any of five methods: (1) allowing processes to test for input emptiness, (2) allowing processes to be internally non-determinate, (3) allowing more than one process to produce tokens on a channel, (4) allowing more than one process to consume tokens from a channel, and (5) allowing processes to share variables. However, all these methods break the denotational semantics defined above, by destroying continuity and/or preventing use of the fixed point theorem.

To deal with a real-time environment, it suffices to add non-determinism only in the actors that interact with the environment. Thus, the core computation can continue to be determinate and benefit from the denotational semantics. For example, a source actor may sample the environment at regular intervals and produce a token pair (value,presence) to denote whether the environment was ready. Such an actor is internally non-determinate (method (2) above), but its downstream consumers need not be. Alternatively, to avoid buffering non-present tokens, we may prefer a demand-driven source actor with unbuffered output. The source actor could test its output stream and emit a (value,presence) pair only when the downstream actor is ready to consume. Thus, the consumer receives a current sample whenever it is ready, without having to wade through stale, buffered samples. This approach is equivalent to extending Lee and Park's list of non-deterministic methods with: (6) allow processes to test for output fullness on channels with finite buffers. In general, finite buffers may introduce deadlock—arguably a form of non-determinism, though not a useful one. Nevertheless, introducing finite buffers only at primary inputs and outputs, with a mechanism for dropping samples (combining (1), (2), (6), (7)), is deadlock free.

Chapter 3

Synthesis Methodology for FPGA

Compiling a TDF program to hardware involves a large semantic gap between abstraction and physical implementation. For a given target, be it ASIC, FPGA, or programmable platform, the synthesis methodology must bind design decisions for every aspect of the language and the specific program, with target-specific constraints and optimizations. On a programmable platform, some decisions may already be bound into custom resources, *e.g.* the stream transport. On an FPGA, however, nearly all design decisions remain to be bound, including:

- Stream protocol
- Stream pipelining (for long interconnect)
- Queue implementation (for stream buffering)
- Queue capacities
- SFSM synthesis style
- Streaming memory control and allocation
- Primary I/O style

This chapter describes a complete synthesis methodology for mapping TDF to an FPGA. We begin with an overview of the compilation flow in Section 3.1. Sections 3.2

through 3.6 discuss the design decisions above and their respective implementations, from the level of stream protocol to system synthesis. Section 3.7 discusses pipelining of streams and processes. We defer the discussion of choosing queue capacities and pipelining depths to Chapter 6 on optimization. In this chapter, we focus primarily on hardware mechanisms and synthesis style.

3.1 Compilation Flow

To facilitate retargeting to different hardware, our compile flow uses the Verilog hardware description language as an intermediate form. This form is then compiled to a specific hardware target using conventional behavioral synthesis tools. The intermediate behavioral Verilog uses no device-specific libraries and is portable to any FPGA. However it uses idioms that infer particularly efficient structures on Xilinx Virtex/Spartan FPGAs, *e.g.* SRL16. Our tool flow targeting Virtex/Spartan is shown in Figure 3.1 and consists of three phases:

1. *Compile TDF to Verilog.* This step, using our custom compiler `tdfc`, includes TDF language processing, optimizations, and translation to Verilog. It automatically generates firmware for streams, stream buffers (queues), module firing control, and the netlists composing them all.

2. *Behavioral Synthesis.* We use a commercial Verilog compiler, Synplify Pro 8.0 [Synplicity, 2005c], with synthesis features such as FSM compilation, resource sharing, and retiming[1]. We specify a clock target of 200MHz, which the tool tries to meet by automatically applying additional, low-level optimizations such as logic replication.

[1]We use the following optimizations in Synplify Pro: FSM Explorer, FSM Compiler, Resource Sharing, Retiming, Pipelining. Here, pipelining refers to retiming delays into the pipeline registers of custom components such as multipliers and Block-RAMs.

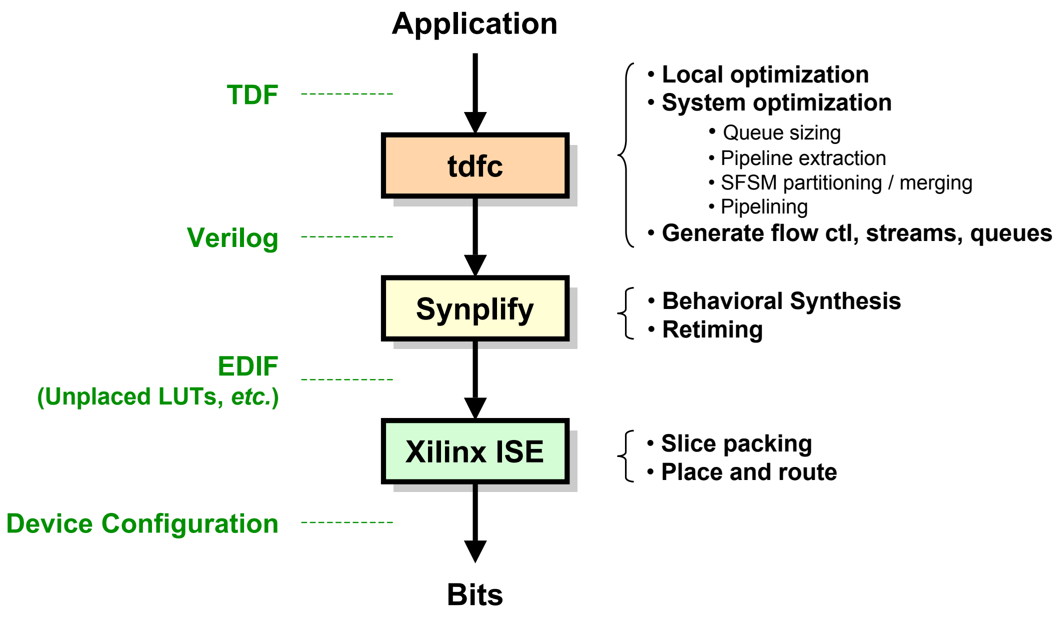


Figure 3.1: Tool flow targeting Xilinx Virtex/Spartan series FPGAs

3. *Place and Route.* We use a commercial tool suite, Xilinx ISE 6.3i [Xilinx, 2005b] for PAR. To accurately measure the used chip area, we constrain PAR to use the minimum feasible square area, rather than the entire chip. A scripted loop tries a progressively larger area constraint until PAR succeeds.

The TDF compiler `tdfc` is responsible for optimization and translation to Verilog. Its compiler passes are organized as follows:

1. *Parse, Link, Type-Check.* The TDF language front end.

2. *Canonical Form.* Transform away several language features that are mere syntactic sugar, including:

   - Infer and instantiate `copy` SFSMs for stream fanout,
   - Remove inlined calls from SFSMs by *exlining* (instantiate an external, stream-connected SFSM) or *inlining* (copy callee into caller),

- Remove array accesses from SFSMs by *exlining* (instantiate an external, stream-connected memory),
- Optionally expand states into basic blocks, for analysis.

3. *Flatten Hierarchy.* Form a single level composition of SFSMs and streaming memories.
4. *Bind Parameters.* Propagate values of `param` bind-once parameters into SFSMs.
5. *Local Optimization.* Perform traditional compiler optimizations on SFSMs such as constant folding, constant propagation, and unreachable state removal. Exhaustive local optimization is not necessary here, since the Verilog compiler will do it later.
6. *System Optimization.* Perform system-level optimizations that are unique to streaming systems, including:
   - Queue sizing
   - Pipeline extraction (Decompose SFSMs into pipelines of SFSMs)
   - Granularity transformations (SFSM partitioning and merging)
   - SFSM pipelining
   - Stream pipelining
7. *Code Generation.* Emit TDF, C++ for simulation, or Verilog.

The remainder of this chapter deals with Verilog code generation from TDF and pipelining. Additional system optimization is discussed in Chapter 6, including queue sizing, SFSM merging, and SFSM partitioning.

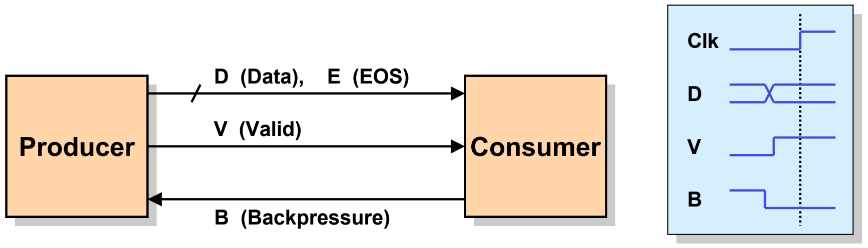


Figure 3.2: Stream protocol

3.2 Stream Protocol

Our stream protocol is a synchronous handshaking protocol, chosen for its simplicity and ease of pipelining and queuing. It is diagrammed in Figure 3.2. The stream producer emits bussed data D and a data presence bit V ("valid"). The stream consumer emits a (low-active) back-pressure bit B. V and $\neg B$ serve as "ready" signals for cycle-by-cycle flow control, and both must be asserted during a rising clock edge to commit a transaction. That is, a single token with value D is transferred at the clock edge if $V \wedge \neg B$. The use of single cycle transactions is intended to support a synthesis style that evaluates one TDF state per cycle. In addition to data, D includes a bit E to represent an out-of-band *end-of-stream* value. This representation allows end-of-stream to be transported and queued like any data token. Stream queueing and pipelining are described below, in Sections 3.4 and 3.7.

3.3 SFSM Synthesis

Our SFSM synthesis style evaluates one SFSM state per machine cycle. It does so by directly translating each SFSM state into a behavioral Verilog FSM state, with code to control that state's firing, stream transactions, and datapath action.

Each SFSM state begins with a firing guard to wait for readiness of the streams consumed or produced in that state. The guard combines incoming flow control bits from those streams, namely input valid, input end-of-stream, and output back-pressure. So long as the guard is false, the SFSM emits non-ready outgoing flow control and waits in the same state. When the guard becomes true, the SFSM emits ready outgoing flow control, to commit the stream transactions, and evaluates its state action. Figure 3.3 shows a sample SFSM, and Figure 3.4 shows the corresponding Verilog code for its firing control. The firing guard is implemented using a Veriog "if" statement. If the present state has more than one firing rule, then the implementation needs a separate guard for each rule, plus a priority mechanism to choose among multiple enabled guards. This can be implemented in Verilog by generalizing the "if" for one guard into an "if-then-else" for a sequence of guards.

One consequence of this firing control style is that, since an SFSM always waits for incoming flow control before asserting outgoing flow control, connecting two SF-SMs directly would cause deadlock. Instead, SFSMs must be connected through intermediate elements that assert flow control first, namely stream queues.

An SFSM is synthesized in two parts, (1) an FSM module for firing control, and (2) a datapath module for state actions. This division allows each part to be compiled and characterized separately to study of the costs of stream support. Figure 3.5 shows a schematic for a typical SFSM, showing the separation of FSM and datapath. Note that stream flow control and end-of-stream signals (V, B, E) are handled entirely in the FSM for firing control, while stream data signals (D) are handled entirely in the datapath. Thus, the sample code from Figure 3.4 belongs strictly in the FSM module. The FSM controls the datapath by sending its present state and an indication of which signature was chosen for firing. Those signals in turn control multiplexers in the datapath that choose stream output values and new data register values. If-then-else conditions are evaluated in the datapath, and their boolean values are sent back to the FSM for use in determining next state. With this structure, most of the statements

```
select (input boolean     s,  input unsigned[8] t,
        input unsigned[8] f, output unsigned[8] o)
{
  state S (s) : if (s) goto T; else goto F;
  state T (t) : o=t;   goto S;
  state F (f) : o=f;   goto S;
}
```

Figure 3.3: Example SFSM for controlled merging of two data streams (canonical *select* actor from Boolean Controlled Dataflow)

```
always @* begin
  t_b=1;  f_b=1;  o_v=0;  o_e=0;
  case (state_reg)
    state_S: ...
    state_T: ...
    state_F: begin
      if (f_v && !f_e && !o_b) begin
        f_b=0;  o_v=1;  o_e=0;
        ...
      end
    end
  endcase  // case (state_reg)
end  // always @*
```

Figure 3.4: Verilog firing control for the SFSM of Figure 3.3, showing state F which consumes from stream f and produces to stream o. Suffixes _v, _e, _b refer to the valid, end-of-stream, and (low active) back-pressure bits of each stream.

in an SFSM action can be translated directly to corresponding Verilog statements in the datapath and/or FSM, including `if` and nested statement blocks. TDF `goto`, which denotes an immediate control branch, is translated into a predication of the state action's remaining statements.

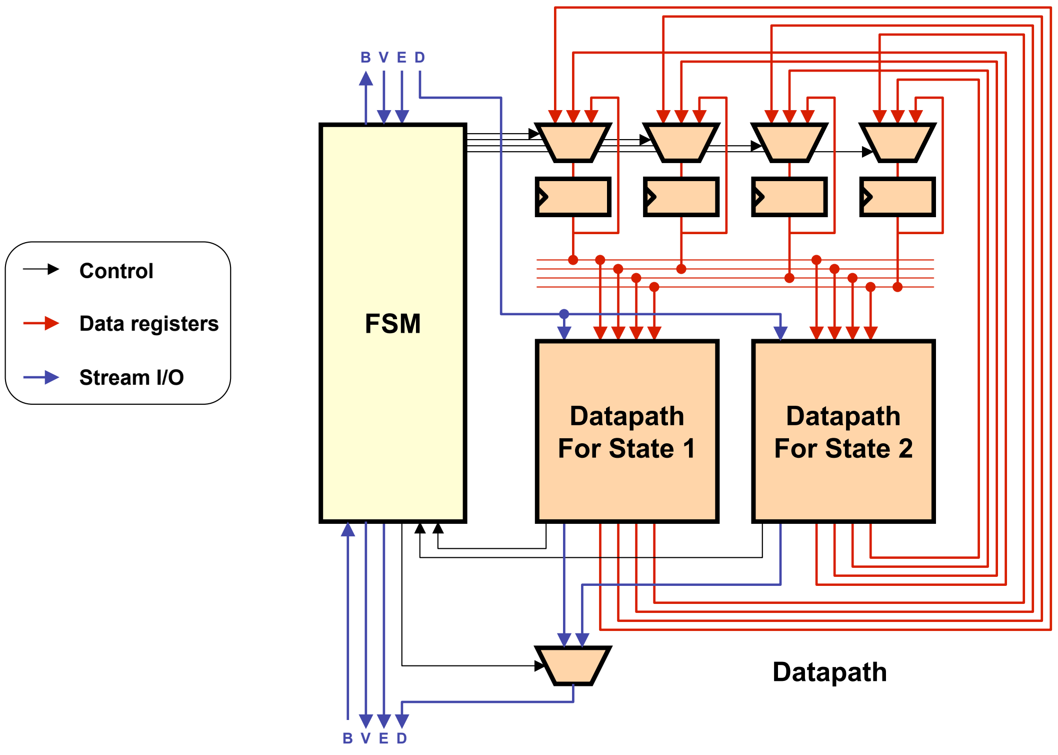


Figure 3.5: SFSM synthesis as FSM and datapath

3.4 Queue Implementation

Each stream must be buffered, using a queue, for purposes of correctness and performance. We will separate the issue of queue implementation from queue capacity, though clearly both are important. The choice of queue capacity for each stream is treated in Chapter 5. A queue implementation must consider the following design issues:

1. *Queues must temporally align data tuples that reach their destination at different times.* This is handled implicitly by stream flow control and firing guards.

2. *Queue flow control must not induce deadlock.*

   Our chosen style of SFSM firing control (Section 3.3) makes SFSMs wait for incoming flow control before firing and asserting outgoing flow control. To avoid deadlock, queues connecting SFSMs should assert their outgoing flow control unconditionally, or in some other way that avoids flow-control feedback loop.

3. *Queues should not bottleneck system throughput.*

   A queue should be able to consume and produce a token at every clock cycle, to match full throughput SFSMs. Also, a queue should have minimum pipeline delay from input to output, since adding pipeline delay to stream feedback loops may degrade system throughput. For this reason, fixed-latency delay lines are in general unacceptable as queues.

4. *Queues should not bottleneck system clock rate.*

   A queue should have a high internal clock rate and low input/output latencies. Also, a queue should have no combinational path between inputs and outputs, since such a path would would incur cumulative combinational delay from producer, queue, consumer, and interconnect. Instead, a queue should combinationally decouple the producer and consumer.

Each of these features requires area and delay, and some features are in conflict. Thus, a design is best served by having different kinds of queues for different circumstances. In some cases, it is appropriate to cascade several queues on one stream to serve multiple purposes. We present three kinds of queues for Xilinx Virtex/Spartan FPGAs, based on an enabled register, a shift register, and embedded memory.

3.4.1 Enabled Register Queue

The simplest and smallest queue is an enabled register, as shown in Figure 3.6. Such a queue can store one token, and a cascade thereof can store more. Data D, end-of-stream E, and valid V are simply registered, with the V register serving as a full/empty state bit. The registers are enabled in every circumstance except when the queue is full and its downstream is not ready, in which case the queue and its upstream must stall. Consequently, flow control is simply an AND gate: $i_B = o_V \wedge o_B$.

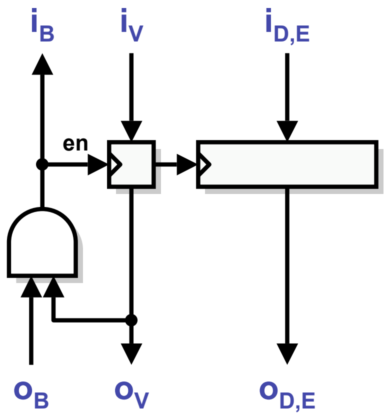


Figure 3.6: Enabled register queue with capacity one

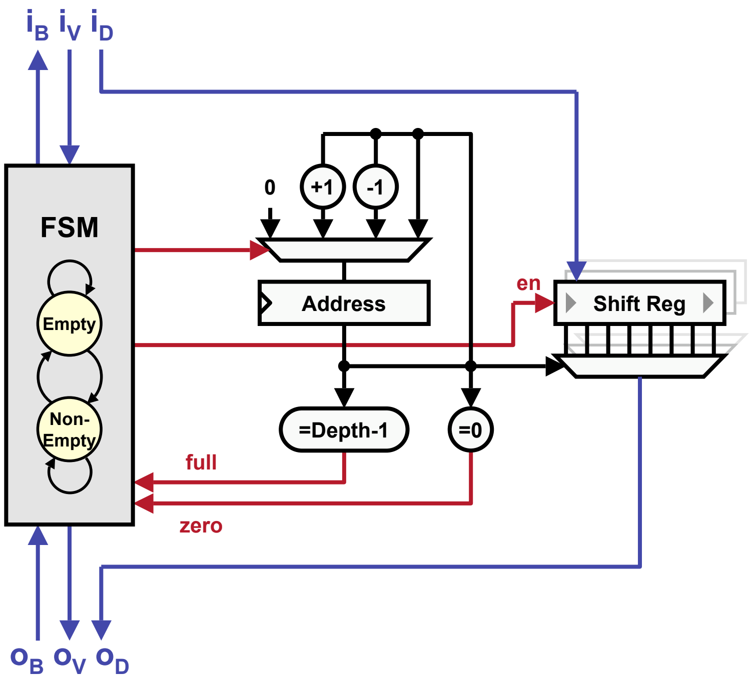
For a single stage, this queue is both small and fast, needing only an AND gate and $(w+2)$ registers for w bit data. If those registers are packed with other logic in an FPGA cell, the queue may be nearly free. However, this queue has a combinational feed-through of back-pressure, from B_o to B_i, violating queue design issue 4 above. That feed-through may slow the system clock rate if it spans multiple connected SFSMs, multiple enabled register stages, or long distances on chip. Consequently the enabled register queue is best suited for single-stage connection of small SFSMs.

If enabled register queues are used to connect SFSMs in a streaming feedback loop, the feed-through of back-pressure will create a combinational feedback loop, violating queue design issue 2. This loop is free of race conditions and will behave deterministically. However, it will deadlock if all queues become simultaneously full. Thus, streaming feedback loops should include at least one queue of a different kind with no feed-through.

3.4.2 Shift Register Queue

Xilinx Virtex/Spartan series FPGAs have an SRL16 mode which implements a shift register of depth up to 16 in just one 4-LUT cell per bit of data. This mode provides a basis for an efficient queue of depth up to 16, or deeper by chaining shift registers. Figure 3.7 shows schematics for such a queue. The queue uses a shift register for data storage, shifting data in at the front and out at a dynamic address corresponding to the queue occupancy. Fullness and emptiness are evaluated by comparing that address to queue bounds (with an offset). Flow control is handled by a state machine. For higher performance, the queue is enhanced with output registers for data, valid, and back-pressure, and with pre-computation of the next cycle's fullness and emptiness.

We have implemented such a queue in behavioral Verilog, fully parameterized for depth d and bit width w, using shift registers that infer SRL16s. Figure 3.8 shows the implementation speed and area for various parameters, pre-PAR, on Virtex-II XC2VP70 -7. The implementation is efficient, near or above 200MHz for all depths and widths up to 128. Thus, queue speed will seldom be the critical component of a streaming system. The SRL16 area is $(w+1)\lceil d/16 \rceil$ 4-LUT cells. For shallow depths, total queue area is typically 2-4 times the SRL area (*e.g.* a depth-16 queue for 16-bit data is 57 4-LUT cells, 17 of which are SRLs). For large depths, control structures are amortized, and total queue area becomes asymptotic to the SRL area (1/16 4-LUT cells per bit). Depth two is a special case where much of the queue structure is removed by partial evaluation, including the SRL16 being converted into



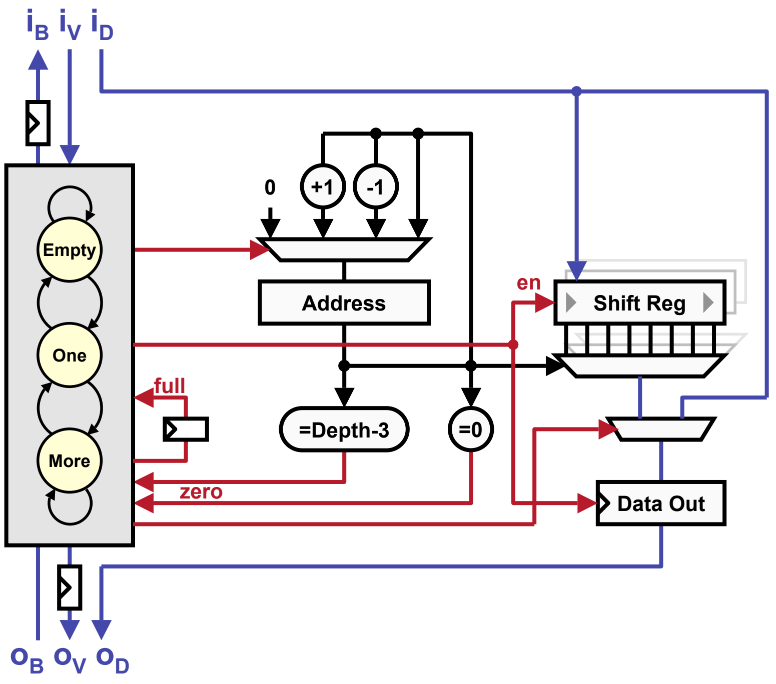
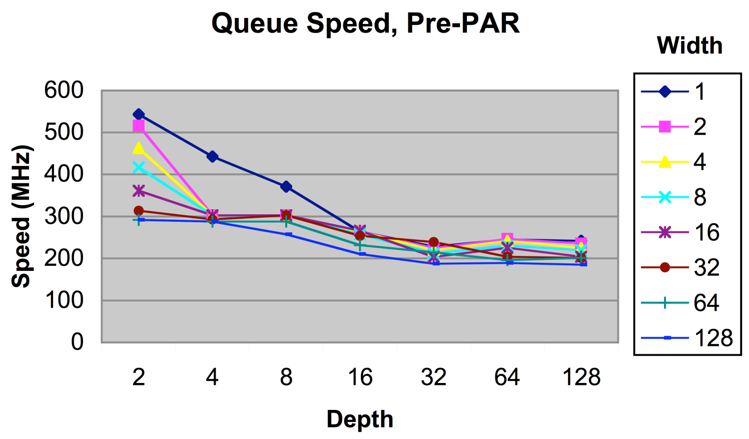
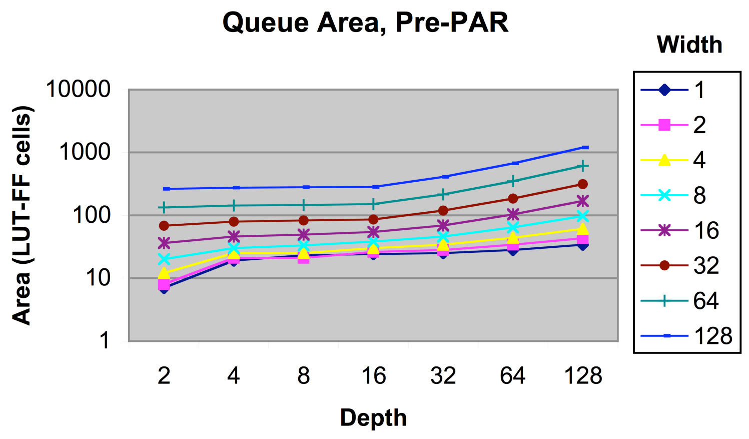


Figure 3.7: Shift register queue for stream buffering: (top) basic, (bottom) enhanced with registered output and pre-computation of the next cycle's fullness and emptiness.

(a)



(b)



(c)

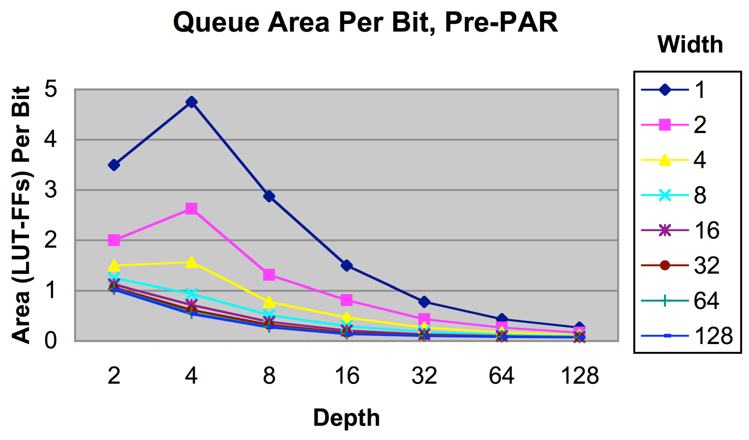


Figure 3.8: Performance of shift register queue, pre-PAR, on XC2VP70: (a) Speed, (b) Area, (c) Area per bit of storage

registers (*e.g.* a depth-2 queue for 16-bit data is 37 4-LUT cells).

It is useful to note that this shift register queue requires a minimum capacity of two to operate at full throughput (design issue 3). To avoid combinational feed-through of flow control signals (design issues 2 and 4), the queue disallows consuming when full. Consuming when full requires simultaneously producing, which would require conditioning input-backpressure on output back-pressure and creating a feed-through. A queue of unit capacity is always either empty or full, so under these conventions, it could never simultaneously consume and produce. Instead, it would ping-pong between consuming when empty and producing when full, achieving at most half throughput. A queue of capacity two has an intermediate state, neither empty nor full, that can operate at full throughput. Thus, a queue that simultaneously satisfies design issues 2, 3, and 4 must have a minimum capacity of two.

3.4.3 Block RAM Queue

Xilinx Virtex/Spartan series FPGAs feature embedded memory, termed *Block SelectRAM* or simply *Block RAM*, which provides a basis for efficient, large queues. Virtex 4 FPGAs further provide custom FIFO controllers for Block RAM based queues. Block RAM queues have asymptotically higher storage density than SRL based queues, which are limited to 16 bits per 4-LUT cell. That density advantage would be realized at high queue depths, probably at hundreds of elements or above. We presently do not implement Block RAM queues.

3.5 Streaming Memories

Memory blocks can be incorporated into a streaming system by accessing them through streams. This purpose is served by a family of TDF *segment operators*, described in Section 2.3.3.4 and more fully in [Caspi, 2005]. A segment operator abstracts the type and timing of a memory block behind a streaming interface. The interface is tuned for one of several possible access modes (sequential, random access, *etc.*) and may include an automatic address generator. A complete synthesis system would provide separate segment operator implementations for every access mode and memory type, including LUT-based RAM, Block RAM, and off-chip memory. We presently do not implement segment operators on FPGAs. The `tdfc` Verilog back-end emits them as black boxes.

3.6 System Composition

We synthesize a system as a composition of stream-connected *pages*, where a page contains:

- One or more SFSMs,
- An input queue for each incoming stream,
- Pipelining for each outgoing stream (described below in Section 3.7),
- An internal queue for each internal stream, (if the page contains multiple SFSMs).

Figure 3.9 shows such a page. In the simple case, a page contains only one SFSM and exists merely to properly associate queues and pipelining with streams and SFSMs. In the general case, a page refers to a physically localized cluster of SFSMs, where internal queues may be simpler and smaller than input queues, because they represent short-distance communication. For instance, internal queues might be enabled registers, while input queues might be SRL16 or Block RAM based. This notion of a page is also useful for modeling and targeting a *paged* architecture, where SFSMs must fit into fixed size partitions of reconfigurable fabric termed pages. An architectural page

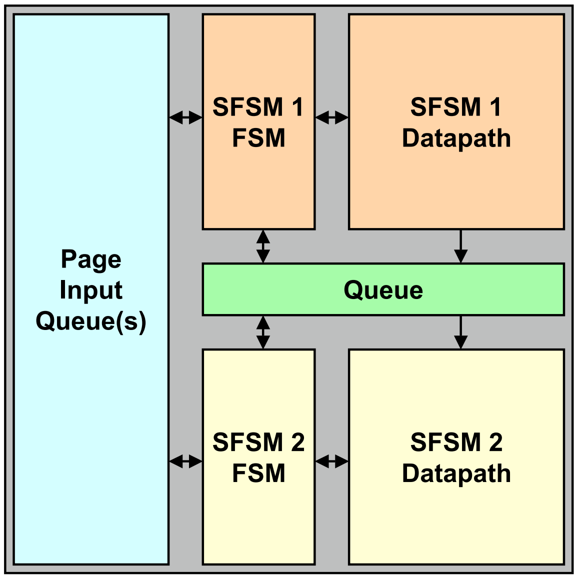


Figure 3.9: A sample page containing two SFSMs

may include custom resources for streams and queues. For example, in SCORE [Caspi *et al.*, 2000a], page input queues are implemented in custom resources, while page internal queues are implemented in reconfigurable fabric on the page. As such, our synthesis methodology for commercial FPGAs can be reused for paged architectures and for studies of best page size.

A complete streaming system is a composition of pages and streaming memories. Our synthesis methodology allows compilation of the entire system as well as separate compilation of pages and page contents (queues and SFSMs). This separation provides a means to characterize component costs and the potential benefit of introducing custom resources for some components.

3.7 Stream Enabled Pipelining

Pipelining is a general approach for achieving higher throughput by adding registers to break long combinational paths and increase clock rate. Adding registers to a circuit changes its sequential behavior. Hence, pipelining also involves modifying the circuit to recover an equivalent, correct behavior (*e.g.* control changes). Pipelining a single, performance critical component often requires modifying external components connected to it, to account for modified latencies and throughputs in the communication pattern. That modification may be very simple, *e.g.* a feed-forward component pipelined to depth N produces correct results N cycles later. In many cases, however, the required modification is complicated and requires multiple layers of correction. A notorious example is the pipelining of function units in a microprocessor. There, correctness requires significant external control for hazard detection, to match the timing of function inputs and outputs. Furthermore, achieving full throughput in a pipelined function unit requires issuing an instruction at every possible clock cycle, which in turn requires modification of other subsystems such as branch prediction and load/store units.

Streaming systems, and our particular synthesis style for them, support a simpler, more modular approach to pipelining. First, pipelining can be done independently on individual components (processes) without violating system correctness, since the underlying process network is naturally robust to changes in communication latency and throughput. Furthermore, it is possible to pipeline the streams rather than the processes, and then to retime registers into the processes. Such an approach can pipeline a process, to a limited extent, without rewriting its RTL or behavioral source implementation. These properties make the design and optimization of streaming systems more modular and easier to implement.

The primary requirement for pipelining a streaming system is that the pipeline stages support *bubbling*. A pipeline stage added inside a process, which stalls together with that process, may hold up data in-flight and subsequently deadlock a system. For correctness, an added pipeline stage must pass data through, as if it were firing as an independent process[2]. It must also be initialized to contain a bubble, *i.e.* no valid data, and to pass bubbles through like data. Any bubbles introduced by pipelining would propagate naturally through streams and processes, even around stream feedback loops, yielding a graceful degradation of throughput.

In the remainder of this section, we discuss four automated approaches to pipelining a streaming system: (1) interconnect pipelining, (2) interconnect relaying, (3) logic pipelining and retiming, and (4) logic relaying and retiming. These approaches will be introduced in order of conceptual complexity, simplest first. Each approach adds pipeline delay directly to streams, requiring no modification of process implementation. In a feed-forward stream network, the added pipeline delay is harmless to system throughput. However, in a feed-back stream network, the added pipeline delay may reduce system throughput in tokens-per-cycle. For this reason, pipelining around feedback loops is best done in conjunction with a system-level performance analysis. We propose such an analysis in Chapter 5.

3.7.1 Interconnect Relaying

A stream spanning a long distance on chip may be pipelined by relaying its signal through small, depth-2 queues, as in Figure 3.10(a). An automatic placer may then place those queues as necessary to avoid long, critically slow wires. We dub these queues *relay stations*, borrowing the name from latency insensitive design [Carloni *et al.*, 2003], which uses a similar approach. Relay stations should have no combina-

[2] A bubbling pipeline stage need not be implemented with its own firing control—it merely needs to appear that way, as observed at the output of the pipelined process. Sections 3.7.3 and 3.7.4 discuss two implementations of bubbling pipelining that do not add firing control to each pipeline stage.

tional feed-through of signals. Hence, the SRL queue introduced in Section 3.4.2 is an appropriate implementation choice. We note that a depth-2 queue is specifically required, because (1) pipelining using D flip-flops would introduce error in flow control, and (2) a depth-1 queue with full throughput and no combinational feed-through cannot be constructed for our stream protocol.

3.7.2 Interconnect Pipelining

A stream spanning a long distance on-chip may be directly pipelined using D flip-flips, as in Figure 3.10(b). An automatic placer may then place the registers as necessary to avoid long, critically slow wires. To correct for the staleness of pipelined flow control signals, the stream's queue must be placed downstream of the pipeline registers and be modified to avoid overflow. Specifically, with N pipeline stages, and consequently a $2N$-cycle round-trip staleness of flow control, the queue must signal fullness whenever it has $2N$ or fewer empty slots. In essence, the queue must *reserve* a capacity equal to the round trip pipeline delay. A shift register queue with reservation is only slightly more complicated than the original queue from Section 3.4.2, requiring one additional address comparator. Furthermore, the queue must have a depth of at least $2N + 1$ to avoid deadlock, and at least $2N + 2$ to permit full throughput.

Interconnect pipelining is not directly supported in most commercial FPGAs, including the Xilinx Virtex/Spartan series. Those devices have no dedicated interconnect registers. Instead, registers exist as an integral part of logic cells, and the entire cell is moved during placement. Reallocation of registers during or after placement might provide additional flexibility for breaking long distance path on these architectures. Several approaches have been proposed in the literature for performing retiming and register reallocation after or during placement in an RTL tool flow, pertaining to registered interconnect [Tsu *et al.*, 1999] [Singh and Brown, 2001] and

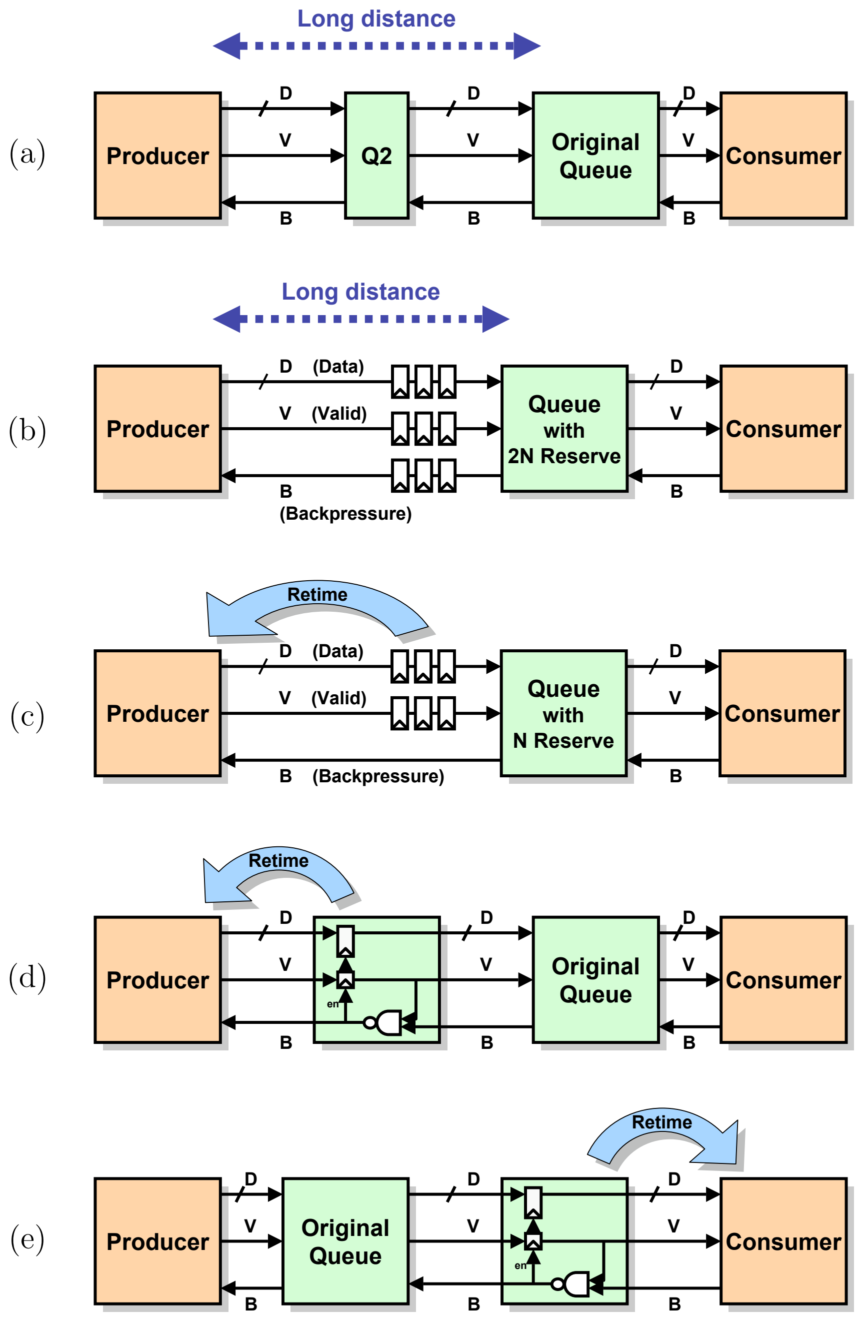


Figure 3.10: Stream enabled pipelining: (a) interconnect relaying, (b) interconnect pipelining, (c) logic pipelining and retiming, (d) logic relaying and retiming backward, (e) logic relaying and retiming forward

non-registered interconnect [Weaver *et al.*, 2003] [Singh and Brown, 2002]. However, those techniques are available commercially at the time of this writing.

Stream based interconnect pipelining is simpler by design. Pipelining of arbitrary depth can be added to a stream without affecting correctness and without resorting to *c-slowing* (which is required in [Weaver *et al.*, 2003]). Our basic approach applies stream pipelining before placement and uses a conventional, commercial tool flow thereafter. For improved results, stream pipelining can be applied after placement, to match the actual interconnect delay. This will require post-placement register allocation and post-placement specialization of the reservation parameter in queues (*e.g.* by modifying a register initial value in the placed configuration). In Chapter 5, we propose a way to integrate pipeline depth selection with placement, extending the methodology of [Singh and Brown, 2002] for streams.

Stream based interconnect pipelining incurs a particular, preventable area overhead on Xilinx Virtex/Spartan FPGAs using commercial tool flows. Since the unit of placement is a *slice* of two 4-LUT cells, a pipeline register that is mobile during placement must consume an entire slice. Consequently, interconnect pipelining beyond one level deep requires as much area as full interconnect relaying (at one level deep, the register is packed with the stream producer's logic and is effectively free). For example, with 16-bit data, a depth-2 SRL queue for relaying is 37 4-LUT cells, whereas a mobile pipeline stage is 38 4-LUT cells, with no area advantage. The area overhead of mobile pipelining slices could be avoided if pipeline registers were allocated only after placement, where space permits, as in [Weaver *et al.*, 2003].

3.7.3 Logic Pipelining and Retiming

An SFSM's datapath may be automatically pipelined by adding pipeline registers to one or more output streams and retiming them backwards into the datapath, as in Figure 3.10(c). Only the forward stream signals D, E, V need to be pipelined, not the back-pressure B. As with interconnect pipelining, staleness of pipelined flow control signals must be handled by placing a modified queue downstream of the pipeline registers, reserving a capacity equal to the round-trip pipeline delay. Specifically, with N pipeline stages, and consequently an N-cycle round-trip staleness of flow control, the queue must signal fullness whenever it has N or fewer empty slots. Note that N-deep logic pipelining and M-deep interconnect pipelining may be used together with a downstream queue that reserves $N+2M$ slots. A shift register queue with reservation is only slightly more complicated than the original queue from Section 3.4.2, requiring one additional address comparator. Furthermore, the queue must have a depth of at least $N+2M+1$ to avoid deadlock, and at least $N+2M+2$ to permit full throughput.

Stream based logic pipelining is useful but limited. It works best on perfectly pipelinable computations, such as a finite impulse response (FIR) filter or discrete cosine transform (DCT), where a single-state FSM exists solely for stream flow control and is otherwise independent of the datapath. Stream based logic pipelining has partial benefit whenever the datapath has non-cyclic components at its outputs or inputs. In that case, pipeline registers can be retimed back as far as a feedback cycle, or entirely past it, but not into it. This retiming based form of pipelining is strictly less powerful than loop scheduling techniques such as loop unrolling and software pipelining, since they deal with feedback loops directly. It is also possible for behavioral synthesis to introduce new feedback loops which obstruct retiming based pipelining, for instance via resource sharing.

3.7.4 Logic Relaying and Retiming

An SFSM's datapath may be automatically pipelined by adding a small queue to one or more of its streams and retiming the queue's registers into the the datapath, as in Figures 3.10(d), 3.10(e). The enabled register queue from Section 3.4.1 is particularly useful for this purpose, since it may be cascaded to provide arbitrarily many, retimable, enabled registers on the stream data line D. The AND gate of each enabled register queue would also be cascaded, slightly increasing flow control latency. There are both advantages and disadvantages to using logic relaying with enabled registers versus logic pipelining with D flip-flops. The comparison is discussed below and evaluated by experiment in Chapter 4.

Logic pipelining adds registers only at the SFSM outputs, whereas logic relaying adds registers at either the outputs (Figure 3.10(d)) or the inputs (Figure 3.10(e)). Retiming forward from the inputs may provide different optimization opportunities than retiming backward from the outputs, due to feedback loops within the SFSM, and due to the idiosyncratic interaction between retiming and other optimizations in behavioral synthesis and technology mapping. Even considering only backward retiming, such idiosyncracies sometimes make it possible for enabled registers to retime better than D flip flops (this is discussed further in Chapter 4).

The use of enabled registers rather than D flip-flops incurs an area disadvantage, since retiming enabled registers may leave a residue of a register-multiplexer cycle. Nevertheless, logic relaying also has an area advantage, since it does not require modifying the original stream queue. Logic pipelining, on the other hand, requires modifying the original stream queue with reservation, which requires a comparator and potentially more capacity. The resulting area overhead may be large for an originally shallow queue.

Chapter 4

Characterization for FPGA

In this chapter, we apply the synthesis methodology of Chapter 3 to compile seven multimedia applications to a commercial FPGA, the Xilinx Virtex-II Pro XC2VP70. From this case study, we seek to determine the merit and overheads of our streaming design methodology, including:

- What is the attainable application performance?
- What is the overhead for supporting streaming, *e.g.* the added area for stream queues?
- What is the speedup and area overhead for stream-enabled pipelining?

We gain additional insight by separately compiling and analyzing the components of applications, including individual pages, SFSMs, FSMs, and datapaths.

The chapter is organized as follows. Section 4.1 discusses the experimental setup, including the choice of chip, place-and-route options, and stream implementation parameters. Section 4.2 discusses our suite of seven multimedia applications. The next three sections present results for compiling the applications with increasing levels of optimization: Section 4.3 without pipelining, Section 4.4 with stream-enabled logic pipelining, and Section 4.4 with stream-enabled interconnect pipelining. Finally, Section 4.6 provides a summary of results.

4.1 Experimental Setup

For the studies in this chapter, we target the Xilinx Virtex-II Pro XC2VP70 FPGA, speed grade 7. Virtex-II Pro is a high performance device family, fabricated in 130nm, featuring logic speeds up to 400MHz, multi-standard chip I/O, integrated on-chip memory (Block Select RAM), and hard macros such as 18x18 bit multipliers. The XC2VP70, having 74K 4-LUT cells, is a medium-large device capable of mapping each of our applications on a single chip. It is not the largest device in its family, but it is the largest one available with the family's maximum speed grade of 7 (*speed grade* is Xilinx's term for binning devices by clock rate). We do not focus on the newer, 90nm, Virtex 4 family, primarily because its software tools are less mature. Our approach to stream-enabled pipelining relies heavily on retiming in commercial tools, which at the time of this writing, provided limited benefit on Virtex 4.

The methodology of Chapter 3 provides multiple implementation options for streams and does not prescribe how to choose among them. That choice is a matter of optimization. Before we can flesh out a complete optimization methodology, we need a design space exploration to compare the merit and costs of various implementation options, including queue types and pipelining depths. Consequently, the studies in this chapter are organized as a design space exploration for a subset of implementation options. In most cases, we uniformly apply a single implementation choice to every stream in an application. For instance, in every study, we choose a stream's main queue to be the enhanced, shift-register queue from Section 3.4.2 with depth 16. Smaller queue depths are possible, if the user or an automated analysis indicate that those depths do not introduce deadlock (such an analysis is discussed in Chapter 6). Anecdotally, we know that depth 16 is sufficient for every stream in our seven multimedia applications, except for eight feedback streams that hold a raster image row.

For the sake of realism and believability, we report mapping results after place-and-route (PAR). It is possible, and often instructive, to consider estimates of clock rate and area at earlier stages of compilation, such as after behavioral synthesis, or after slice packing. Nevertheless, PAR captures the additional delay of interconnect, which is increasingly important in large devices. We follow the tool flow of Figure 3.1, using Xilinx ISE 6.3i for PAR. The primary inputs and outputs of each compiled component are left as raw streams (valid, back-pressure, and data) and are allowed to float freely during PAR.

To capture accurate area estimates, we constrain placement to use a minimum size, square floorplan. Without such a constraint, the placer is free to use the entire device, and it typically consumes many slices for their routing resources without using their logic. An area constraint forces the placer to consider a realistically small area, as if on a smaller device. The result is typically somewhat slower than with unconstrained placement, since there is less freedom to bring connected components as close as possible. Unless otherwise specified, all areas reported in this chapter are floorplan areas, which include any unused resources within the floorplan.

We use a floorplan that is 20% larger than the area required for maximum slice packing (as reported by `map -c 1`). In the expanded floorplan, the slice packer is allowed to target the entire area, and it is not required to apply maximum slice packing. The expansion figure of 20% was chosen as a balance between aleviating PAR effects and bloating the area. The Xilinx *Development System Reference Guide* [Xilinx, 2003b] (p. 151) warns that using maximally packed slices in production leads to long PAR times and poor clock rates. Our methodology encountered precisely those effects when using a square area constraint with 0% or 10% expansion over the area for maximally packed slices. Using 30% or greater expansion, we observed little added benefit for PAR times or clock rates.

4.2 Applications

As a case study for our streaming design methodology, we use suite of seven multimedia applications written in TDF[1]. The applications range in size from a simple, linear filter to a standards-compliant MPEG encoder having hundreds of streams. Individual SFSMs represent computational kernels such as a 1-dimensional discrete cosine transform (DCT), a run length coder, or a Huffman coder. Most SFSMs require little control and have few states. The largest state machine belongs to an 8x8 matrix transpose, with 21 states. The seven applications are:

IIR (Figure 4.1). An infinite impulse response (IIR) filter for 16-bit data. An IIR filter normally has tight feedback loops that limit throughput. This implementation achieves high throughput by decomposing the filter into a cascade of IIR stages having loose feedback ($y[t] = x[t] + Ay[t-4] + By[t-8]$) and compensating finite impulse response (FIR) stages. The loose feedback loops can be retimed to a higher clock rate than the original, direct implementation, since they have a greater ratio of registers to combinational delay. The remaining feed-forward structure can be easily pipelined.

JPEG Decode (Figure 4.2). A JPEG image decoder for 8-bit, gray-scale images. The heart of this decoder is a processing pipeline for Huffman and run-length decoding, dequantizing, zig-zag scan, and 2D inverse DCT. The two-dimensional IDCT is implemented by cascading two one-dimensional IDCTs (Loeffler-Ligtenberg-Moschytz [Loeffler *et al.*, 1989]) separated by an 8x8 matrix transpose, using banks of 8 parallel streams.

[1]Special thanks to Joe Yeh for authoring the seven TDF applications used in this chapter.

JPEG Encode (Figure 4.3). A JPEG image encoder for 8-bit, gray-scale images. The heart of this encoder is a processing pipeline for 2D DCT, zig-zag scan, quantizing, run-length coding, and Huffman coding. The two-dimensional DCT is implemented by cascading two one-dimensional DCTs (Loeffler-Ligtenberg-Moschytz [Loeffler *et al.*, 1989]) separated by an 8x8 matrix transpose, using banks of 8 parallel streams.

MPEG Encode IP (Figure 4.4). A standards-compliant, MPEG-1 video encoder using a fixed frame pattern {IPP}. The input is a sequence of uncompressed YUV color frames, raster-scanned into separate luminance and chrominance streams. The heart of this encoder is a processing pipeline for motion estimation, 2D DCT, zig-zag scan, quantization, run-length coding, and Huffman coding. The motion estimation engine uses streams as wide as 64 bits to represent a column of eight pixels in one token. The two-dimensional DCT is implemented as for JPEG above.

MPEG Encode IPB (Figure 4.4). A standards-compliant, MPEG-1 video encoder using a fixed frame pattern {IBBPBB}. The encoder structure is similar to MPEG Encode IP, but it includes a second motion estimation engine to handle bidirectional (B) frames. A more detailed description of the encoder can be found in [Yeh, 2005].

Wavelet Encode (Figure 4.6). A wavelet image encoder for 8-bit, gray-scale images. The encoder uses three stages of filtering and sub-sampling to split the image into seven frequency components, which are emitted to separate output streams. Each stage applies a low-pass filter, whose output is encoded through quantization, run-length coding, and Huffman coding, as well as a high-pass filter, whose output is passed to the next stage. Each two-dimensional image filter is implemented as a cascade of two one-dimensional filters.

Wavelet Decode (Figure 4.7). A wavelet image decoder for 8-bit, gray-scale images. The decoder structure is essentially the same as the encoder structure, only in reverse. The decoder uses three stages of scaling and compositing to recombine seven frequency components, from separate input streams, into a raster-scanned image stream. Inputs are decoded through Huffman decoding, run-length decoding, and dequantizing, and are then recombined. Each two-dimensional recombination stage is implemented as a cascade of two one-dimensional stages.

Figures 4.1- 4.7 show the stream-connected page graphs of every application, and Table 4.1 summarizes structural statistics of those graphs. In the figures, arcs represent streams; diamond nodes terminate primary input/output streams; and rectangular nodes represent pages, each containing an SFSM or a segment operator. Strongly connected components are shown with a red outline. The graphs depict a form ready for direct mapping to hardware, *i.e.* after the TDF source is processed for canonicalization, flattening, and parameter binding. Consequently, these flat graphs do not show the original hierarchy of the TDF compositional operators.

A salient feature of our applications, and of streaming applications in general, is that streams partition the application into a mostly feed-forward structure. Table 4.1 indicates that two of our seven applications have no stream feedback, and three have feedback only between one SFSM and one segment. The largest strongly connected component (SCC) contains only three SFSMs and one segment. Feedback certainly exists within SFSMs, but it is largely confined to that level. The latency of feedback loops typically determines application performance. Consequently, for high performance, feedback loops cannot be pipelined arbitrarily deep, and they must be kept physically localized during partition, place, and route. A mostly feed-forward streaming topology makes optimization easier by confining feedback loops to a small granularity—mostly inside SFSMs—where loops can be localized and not interfere with system-level placement. The majority of streams, being feed-forward, can then be readily pipelined, as in Section 3.7.

			Streams			Streams	Largest SCC
Application	SFSMs	Segments	In	Local	Out	in Feedback	(SFSMs+Segs)
IIR	8	0	1	7	1	0	0+0
JPEG Decode	9	1	1	41	8	7	3+1
JPEG Encode	11	4	8	42	1	0	0+0
MPEG Encode IP	80	16	6	231	1	4	1+1
MPEG Encode IPB	114	17	3	313	1	8	1+1
Wavelet Encode	30	6	1	50	7	9	2+0
Wavelet Decode	27	6	7	49	1	23	1+1
Total	279	50	27	733	20	51	3+1

Table 4.1: Application structural statistics

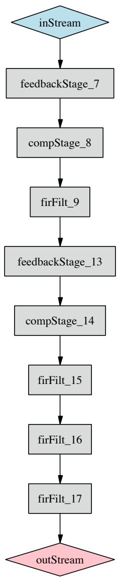


Figure 4.1: Page graph for application "IIR"

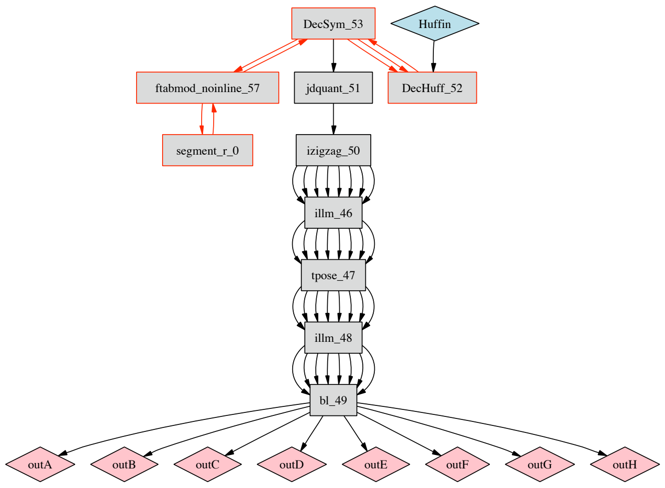


Figure 4.2: Page graph for application “JPEG Decode”

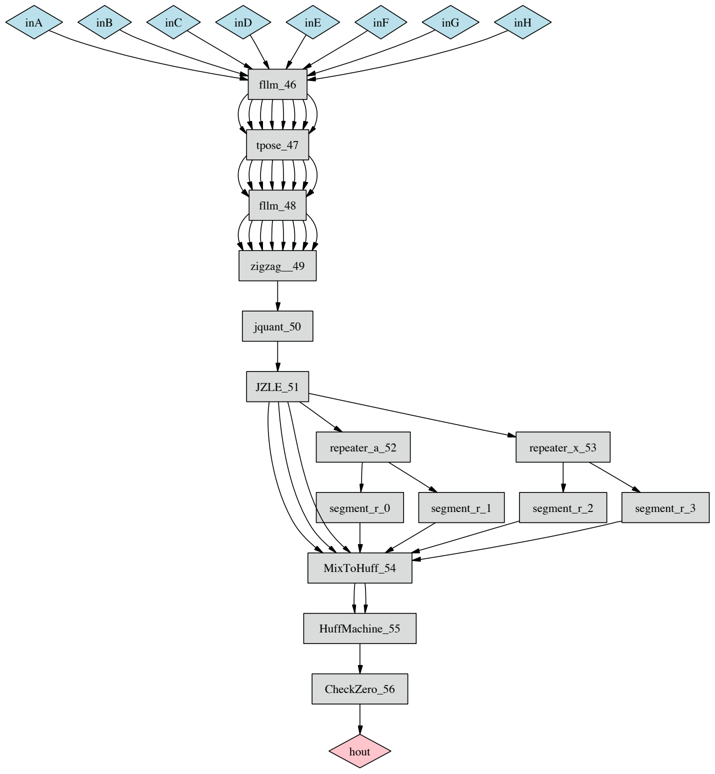


Figure 4.3: Page graph for application “JPEG Encode”

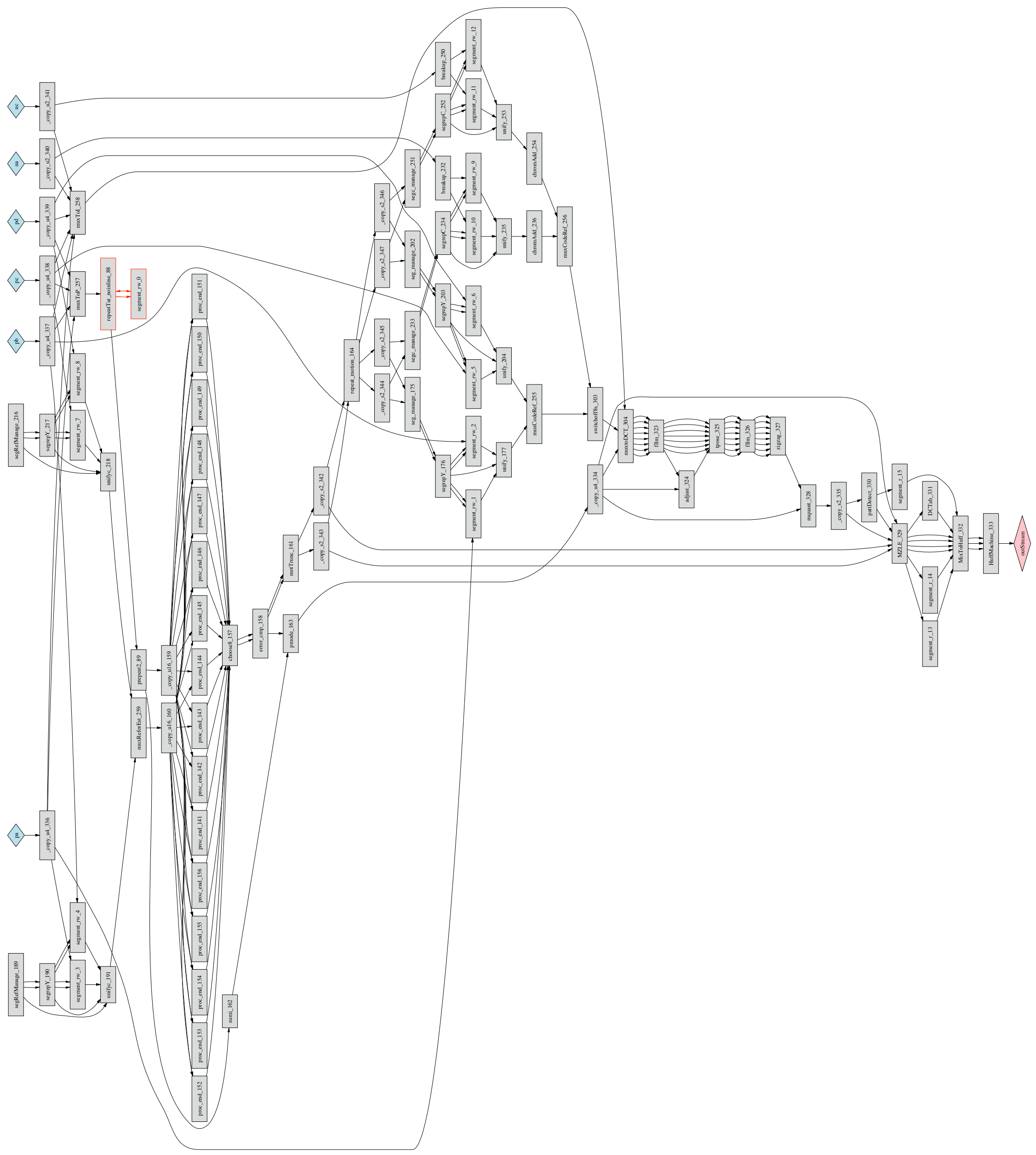


Figure 4.4: Page graph for application "MPEG Encode IP"

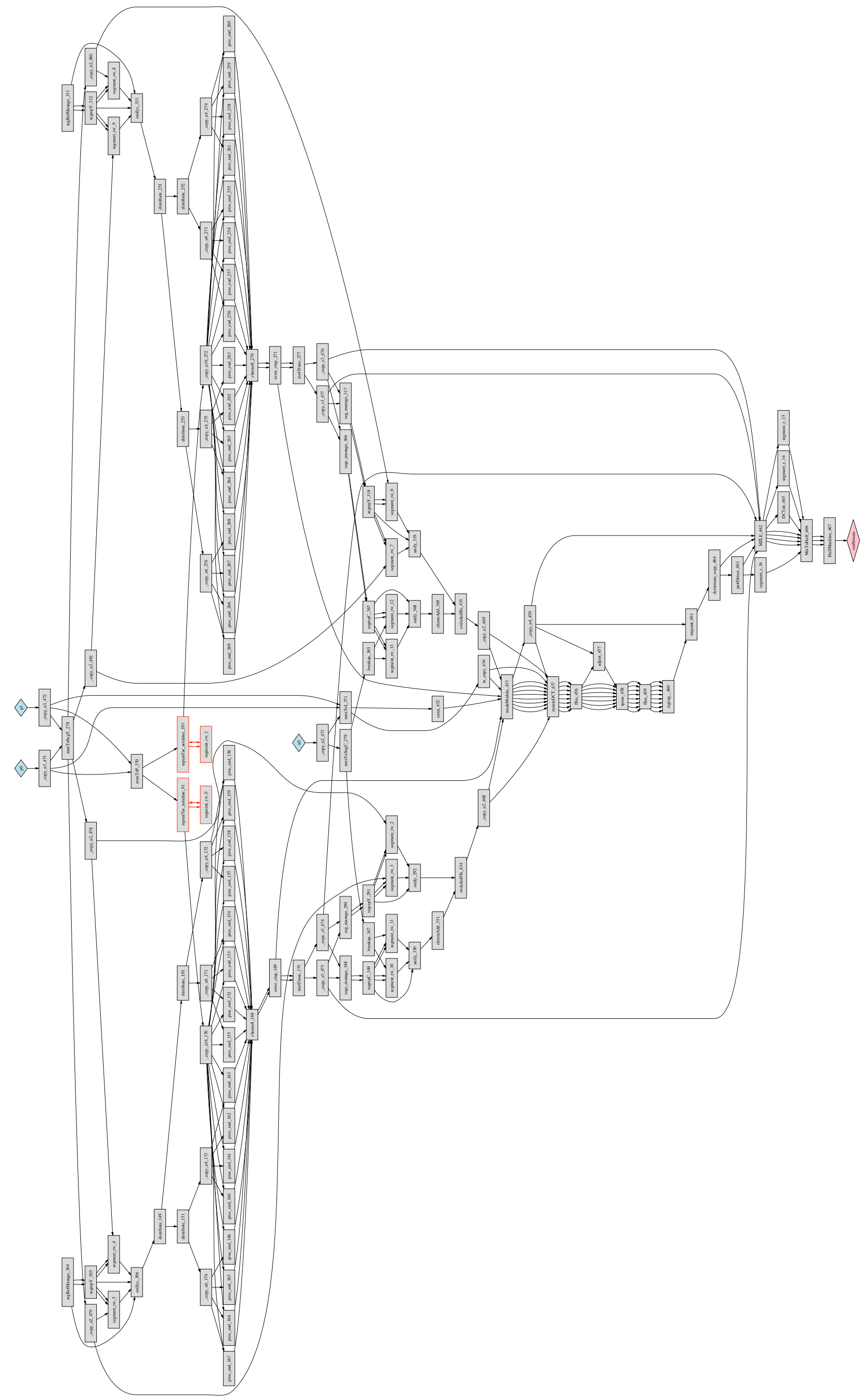


Figure 4.5: Page graph for application “MPEG Encode IPB”

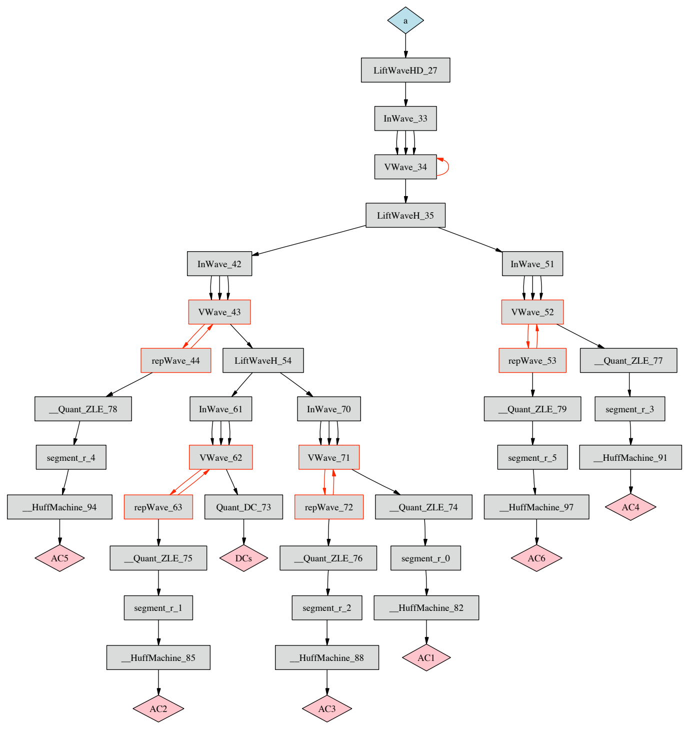


Figure 4.6: Page graph for application "Wavelet Encode"

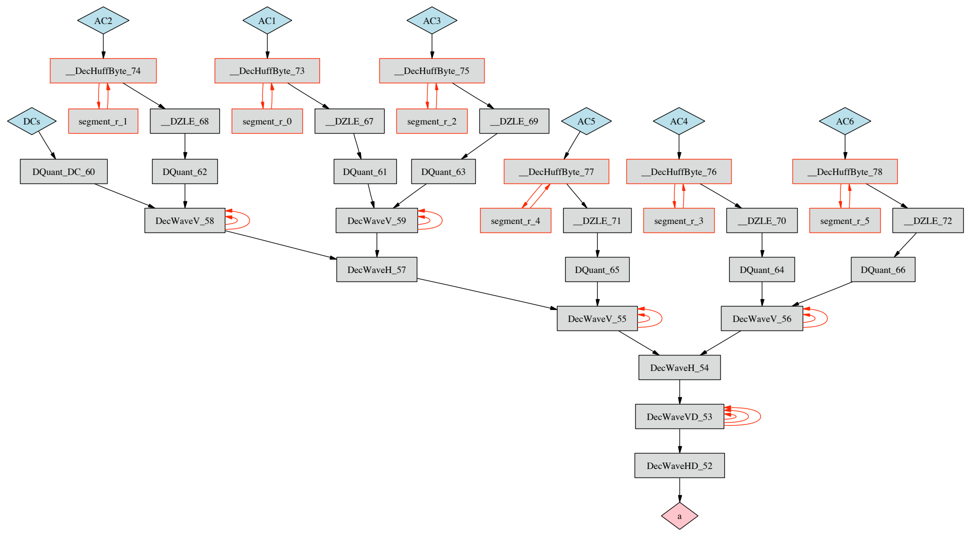


Figure 4.7: Page graph for application "Wavelet Decode"

4.3 Baseline Results

In this section, we present the results of mapping applications to the XC2VP70 FPGA, using the methodology of Chapter 3, with no automatic pipelining. These results represent a baseline case of no stream-based optimization.

4.3.1 Compiling Applications

Table 4.2 shows the baseline area and clock rate for our seven multimedia applications, post-PAR. Application floorplan areas range from about 2000 4-LUT cells for IIR to about 66,000 4-LUT cells for MPEG Encode IPB. Application speeds range from 46.9MHz for MPEG Encode IP up to 166.3MHz for IIR. Application speed tends to be limited by one critical page. Pipelining that page—the subject of Section 4.4—can produce a speedup of 2x or more.

		Application		Pages	
Application	SFSMs	LUT-FFs	Clock (MHz)	LUT-FFs	Clock (MHz)
IIR	8	1,922	166.3	2,110	174.2
JPEG Decode	9	7,442	47.0	8,080	59.2
JPEG Encode	11	6,728	57.5	7,394	59.8
MPEG Encode IP	80	41,472	46.9	54,532	47.3
MPEG Encode IPB	114	65,772	50.3	80,440	39.7
Wavelet Encode	30	8,320	105.7	9,666	118.7
Wavelet Decode	27	8,712	108.8	9,830	127.2
Total	279	140,368	46.9	172,052	39.7

Table 4.2: Application area and speed; unoptimized, post-PAR, on XC2VP70.

4.3.2 Compiling Application Pages

Compiling the pages of an application separately provides a means to identify particular pages and system effects worthy of optimization. Since an application is merely a composition of pages, its area and speed should correspond closely to the area and speed of its pages. Any discrepancy is an opportunity for optimization.

Table 4.2 shows, for each application, the cumulative area of its separately compiled pages, and the speed of its slowest page. We find that application area is smaller than cumulative page area by 8%-24%. We hypothesize that this effect is due to inter-page optimizations during synthesis and technology mapping of applications, such as propagation of constants and don't-cares across streams, and simultaneous covering of producer and consumer pages. The opportunity for such optimizations would be greatly reduced using pipelined streams.

We also find that application speed is typically slower than critical page speed by 5%-21%. We hypothesize that this result is due to two effects: (1) the unpipelined stream implementation introduces combinational delay between producer and consumer, and (2) the application encounters more interconnect delay than any separately compiled page, by virtue of being larger. These effects can be mitigated by stream-enabled pipelining, whose results are discussed in Sections 4.4 and 4.5 below.

4.3.3 Page and Component Speeds

Compiling the components of pages separately provides a means to study how those components contribute to the speed of the whole. We study the 279 pages aggregated from all seven multimedia applications. Figure 4.8 shows the clock speed of each page and its components, from separate compilation. Pages are ordered along the horizontal axis, from fastest to slowest. For each page position on the horizontal axis, the figure displays a tuple of speeds: the speed of the page, its SFSM (without queues), its FSM, and its datapath.

The "Page" line of Figure 4.8 indicates that page speeds range from 40MHz to 315MHz. Pages can be quite fast, even in this unoptimized, baseline study. We

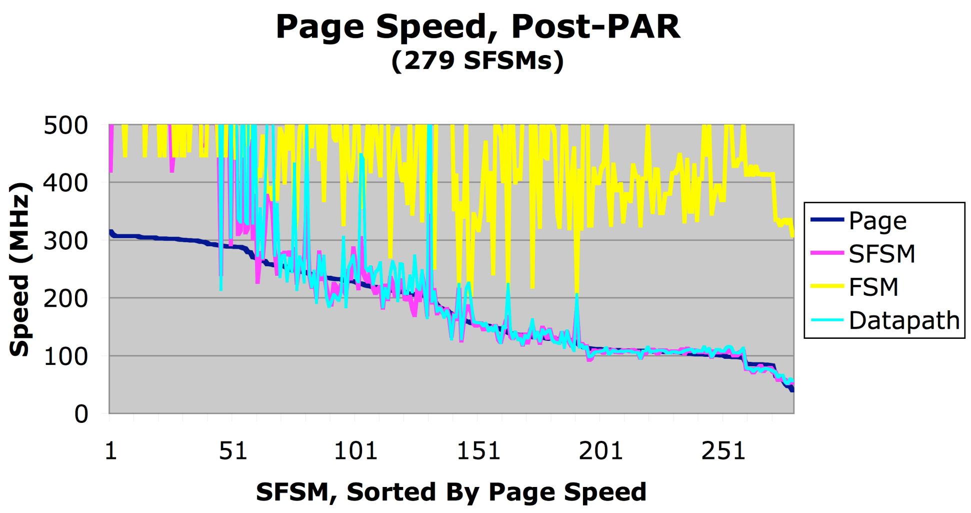


Figure 4.8: Speeds for 279 SFSMs, ordered by page speed; unoptimized, post-PAR, on XC2VP70.

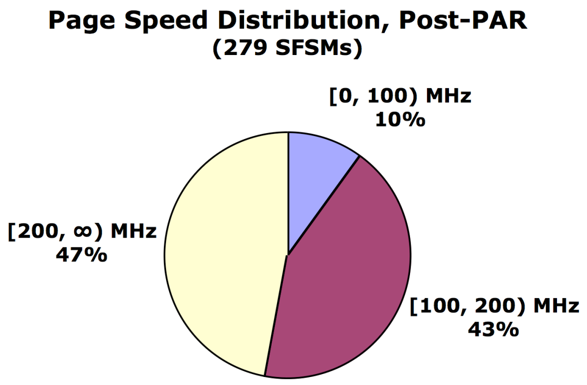


Figure 4.9: Distribution of page speeds for 279 SFSMs; unoptimized, post-PAR, on XC2VP70.

find that 47% of pages are over 200MHz, 43% are 100-200MHz, and only 10% are under 100MHz (this breakdown is also shown in Figure 4.9). To improve application performance, it would suffice to optimize only those few, slowest pages. In those pages, and in fact in most pages, speed is dominated by the datapath.

The "FSM" line of Figure 4.8 indicates that FSMs are very fast, always over 200MHz, and never the slowest part of a page. These are the finite state machines used for stream flow control and sequencing within an SFSM. We conclude that the performance cost of stream flow control is negligible.

Queue speed is not shown in Figure 4.8. Nevertheless, we know from Section 3.4.2 that SRL-based queues are over 200MHz (before PAR) for the depths and widths used in this study. Thus, a queue cannot be the critical component of a page except in the fastest pages. For those cases, we can examine the "SFSM" line of the figure, which denotes a page without its queues. In most cases, the "Page" line tracks the "SFSM" line, indicating that the page's queues are no slower than its SFSM. We shall assume that queues are critical only if "Page" speed is slower than "SFSM" speed by 5% or more, allowing some slack for differences in optimization during separate compilation. Under this criterion, queues are critically slow in 34% of pages, and only in fast pages (123-315MHz).

In the remaining 66% of pages, the critically slow component is the datapath. Those datapaths were not hand-optimized in TDF, nor were they automatically optimized by the TDF compiler for this baseline study. Fortunately, many of them are easy to pipeline. The slowest pages tend to be entirely or mostly feed-forward, including DCT, IDCT, and token distributors/sequencers for MPEG motion estimation. Those pages are slow simply because they were coded in TDF using a single, non-pipelined state, and that state determines the clock rate. Section 4.4 presents the results of optimizing those pages by automatic, stream-based logic pipelining.

4.3.4 Page and Component Areas

Compiling the components of pages separately provides a means to study how those components contribute to the area of the whole. We study the 279 pages aggregated from all seven multimedia applications. Figure 4.10 shows the area of each page and its components, from separate compilation. Pages are ordered along the horizontal axis, from largest to smallest. For each page position on the horizontal axis, the figure displays a tuple of areas: the area of the page, its SFSM (without queues), its FSM, and its datapath. Figure 4.11 is a cumulative version of Figure 4.10. For each page position on the horizontal axis, the figure displays the cumulative area of the page (or component) plus all pages (or components) to the left.

The "SFSM" line of Figure 4.10 indicates that SFSM areas range from 24 to 2,380 4-LUT cells. Most of these SFSMs, 86.7%, are smaller than 512 4-LUT cells. This area distribution is by design, indicating that the TDF programmer had a preferred range of module granularity. The smallest SFSMs tend to have a minimal datapath or no datapath at all, including a token copy with fanout of 2, and a comparison to zero. The largest SFSMs tend to be dominated by a complicated datapath, including DCT, IDCT, an MPEG motion vector selector that chooses the *(vector,error)* pair with least error (full of comparators and multiplexers), and an 8x8 matrix transpose (full of multiplexers).

Module granularity has some important consequences to the implementation. Small modules incur a higher area overhead for stream support. Large modules amortize that overhead but may instead suffer performance degradation from long routing delay. Striking a balance in module size is analogous to the traditional problem of choosing a page size for virtual memory. The area overhead of queues further increases module size. The "Page" line of Figure 4.10, which includes queue area, shows page areas ranging from 60 to 2,888 4-LUT cells. At thousands of 4-LUT cells,

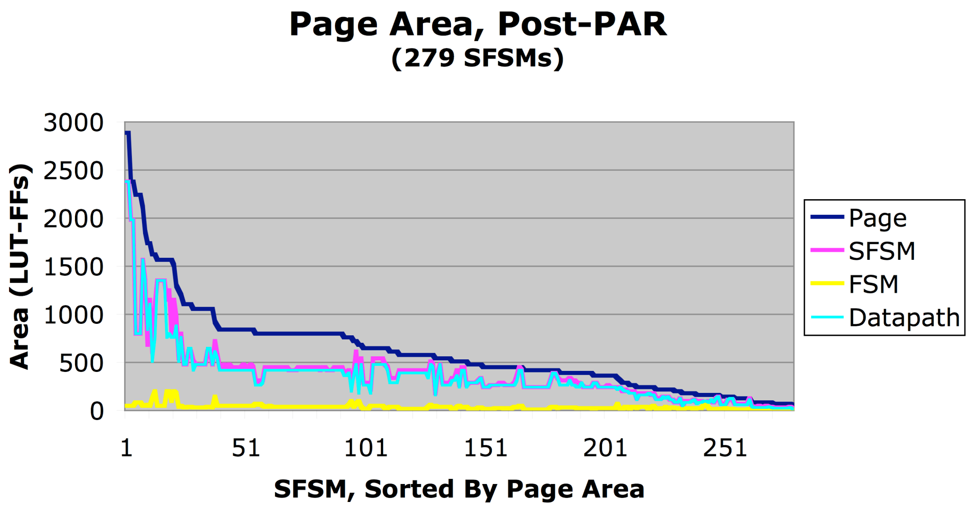


Figure 4.10: Areas for 279 SFSMs, ordered by page area; unoptimized, post-PAR, on XC2VP70.

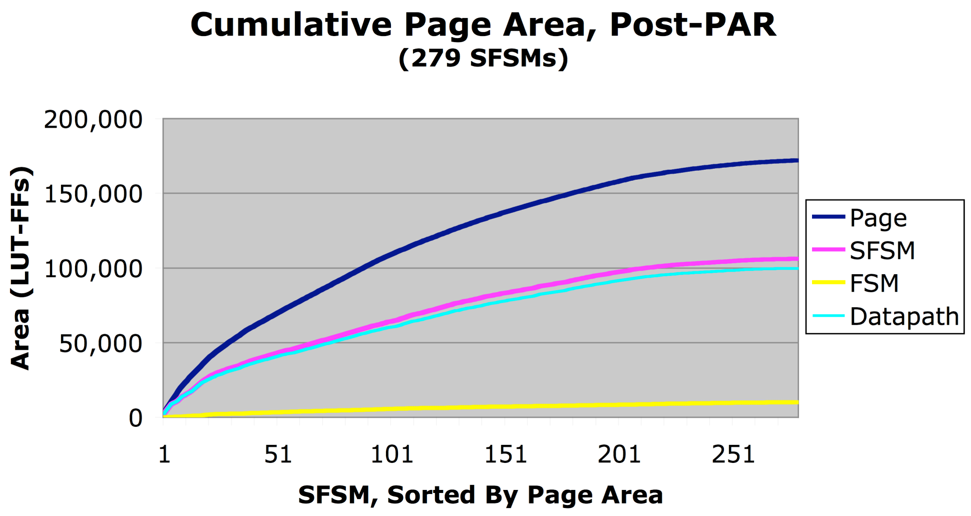


Figure 4.11: Cumulative areas for 279 SFSMs, ordered by page area; unoptimized, post-PAR, on XC2VP70.

Application	% Area FSM	% Area Datapath	% Area SFSM	% Area Queue	% Area Page
IIR	3.4%	72.3%	72.3%	27.7%	100.0%
JPEG Decode	7.0%	63.0%	71.3%	28.7%	100.0%
JPEG Encode	7.5%	55.2%	63.1%	36.9%	100.0%
MPEG Encode IP	5.5%	57.5%	60.3%	39.7%	100.0%
MPEG Encode IPB	5.2%	56.7%	59.5%	40.5%	100.0%
Wavelet Encode	10.1%	61.9%	68.0%	32.0%	100.0%
Wavelet Decode	8.5%	62.5%	70.4%	29.6%	100.0%
Total	5.9%	58.0%	61.7%	38.3%	100.0%

Table 4.3: Area breakdown based on separately compiled components; unoptimized, post-PAR, on XC2VP70. Queue area is derived as 100% minus SFSM area.

our largest pages are large enough to incur non-trivial routing delay. Nevertheless, many of the the large datapaths are entirely or mostly feed-forward, so they would be easy to partition. In fact, they would be naturally partitioned by pipeline registers, as in our proposed automatic, stream-enabled logic pipelining. Chapter 6 discusses additional, automatic techniques for transforming module granularity.

By aggregating page and component areas, we can find the relative contribution of each kind of component to the total area. Figure 4.11 shows the aggregate areas for all 279 pages. Table 4.3 breaks down the aggregate areas for each of the seven applications. We find that FSMs contribute only 5.9% of the aggregate area. Hence, we conclude that the area cost of stream flow control is negligible. We find that queues contribute 38.3% of the aggregate area. Possible ways to reduce the relative area of queues include: (1) optimizing the queue implementation, (2) reducing queue depths, and (3) modifying the granularity of SFSMs to reduce the number of streams, *e.g.* by merging SFSMs.

In summary, we find the following results from baseline, unoptimized compilation. The cost of stream flow control is negligible. The FSMs responsible for flow control are never performance-critical and contribute only 5.9% of the total area. Stream queues are seldom performance critical (critical in only 34% of the fastest pages), but they do contribute a sizable 38% of the total area. And finally, datapaths tend to limit performance and need to be optimized. Datapath optimization by logic pipelining is the subject of the next section.

4.4 Logic Pipelining

In this section, we present the results of mapping applications to the XC2VP70 FPGA, using the methodology of Chapter 3, with automatic, stream-enabled pipelining of logic. We explore the use of logic pipelining as well as logic relaying at different depths. Section 4.4.1 considers the case of uniform pipelining, where all streams are pipelined identically. Section 4.4.2 considers the case of page-specific pipelining, where each page is pipelined separately.

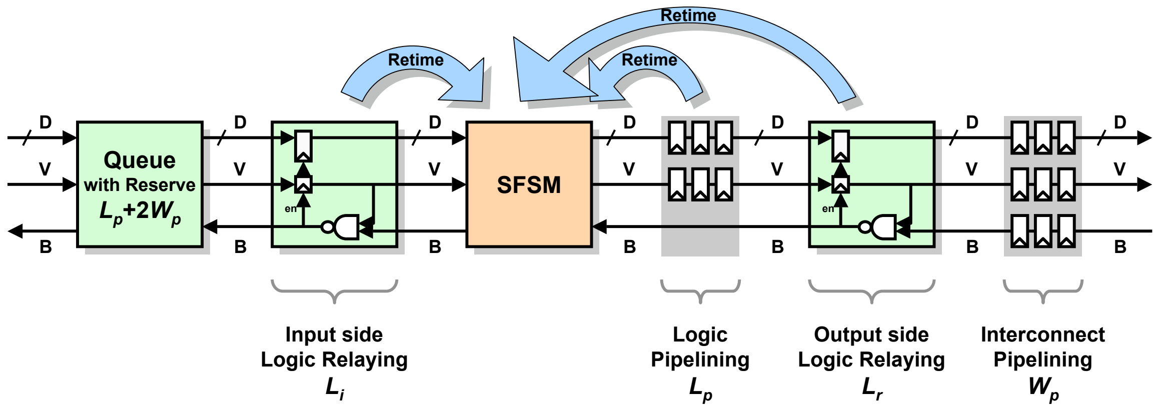


Figure 4.12: Stream enabled pipelining, parameterized by depths $L_i + L_p + L_r + W_p$

Stream-enabled logic pipelining and logic relaying involve placing registers on streams and then retiming those registers into SFSM datapaths. Figure 4.12 shows the placement of registers on the input and output streams of a hypothetical SFSM to be pipelined. On the input side, we consider logic relaying of depth L_i. On the output side, we consider logic pipelining of depth L_p and logic relaying of depth L_r. Interconnect pipelining of depth W_p is also shown in the figure, but its use is deferred to Section 4.5. In this section, each instance of logic pipelining is labeled by its depths $L_i + L_p + L_r$ and its *aggregate* depth L. Assuming that all streams are pipelined identically, each stream queue must be modified to reserve L_p slots to correct for the staleness of pipelined flow control signals. That is, each queue will assert back-pressure whenever L_p or fewer slots are empty.

4.4.1 Uniform Logic Pipelining

The pipelining parameter space $L_i + L_p + L_r$ is large. We begin by exploring it along one dimension at a time. We search for viable, single-variable *trends* by holding two depths constant, sweeping the third depth, and observing the effect on application speed. A viable trend would be one that yields a speedup that (1) is non-zero, and (2) increases monotonically with pipelining depth. It is possible that particular forms of pipelining do not work well for particular applications, due to circuit topology, or due to the interaction between retiming and other optimizations in synthesis and technology mapping. To observe these effects, we study the speeds for single-variable trends as reported by Synplify, before place-and-route.

Figure 4.13 shows application speeds for seven single-variable trends: $0+0+Lr$, $1+0+Lr$, $0+Lp+0$, $1+Lp+0$, $Li+0+0$, $Li+0+1$, and $Li+1+0$. We make the following observations:

Speed does not always increase monotonically with pipelining depth.

In many cases, adding one level of pipelining actually decreases performance. This typically happens at high depths, *e.g.* in the trend $1 + 0 + L_r$, several applications are slower with $L_r = 3$ than with $L_r = 2$. We attribute this effect to idiosyncratic interactions between retiming and other optimizations during

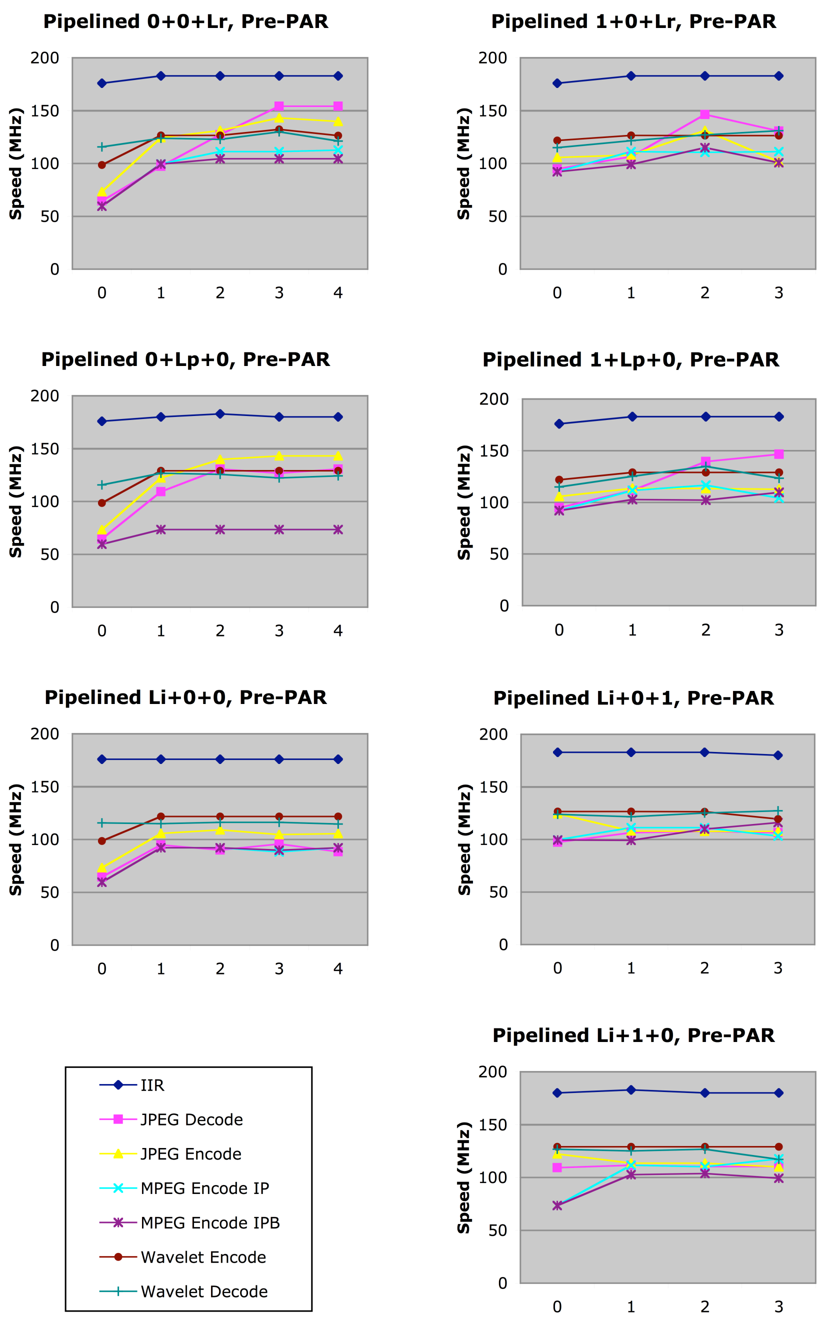


Figure 4.13: Application Speed for different logic pipelining trends; pre-PAR, on XC2VP70.

synthesis and technology mapping. At high pipelining depths, timing driven optimizations have more choices, and thus may stumble on more, detrimental interactions.

Some applications need input-side registers. In the trend $0 + L_p + 0$, which has no input-side registers, MPEG Encode IP and IPB are limited to 73MHz, regardless of L_p. Adding one level of input-side logic relaying, in the trend $1 + L_p + 0$, allows those applications to reach 116MHz and 109MHz, respectively. This phenomenon is due to a particular SFSM which cannot be effectively pipelined by retiming registers from the outputs backward, and which instead requires retiming registers from the inputs forward. Similarly, in the same two trends, JPEG Decode attains higher clock rates with one level of input-side logic relaying than without. Ironically, JPEG Encode encounters a non-monotonicity: adding one level of input-side logic relaying slows it down.

One input-side register is enough. The three trends that vary L_i demonstrate a speedup for $L_i = 1$ but typically no additional speedup for $L_i > 1$.

The two most viable single-variable trends appear to be $0+0+L_r$ and $1+L_p+0$. They demonstrate little non-monotonicity and, in general, higher speeds than the other trends. We study these trends in greater detail in Figures 4.14-4.17 and their corresponding Tables 4.4-4.7.

Figure 4.14 / Table 4.4 show the speed of our seven multimedia applications, after place and route on XC2VP70, for logic pipelining trends $0 + 0 + L_r$ and $1 + L_p + 0$. The missing entries for MPEG Encode IPB indicate cases where that application, expanded by pipelining, no longer fit on the target chip. We find that stream-enabled logic pipelining accelerated all the applications beyond 100MHz, except for MPEG Encode IP and IPB (more on MPEG below).

Figure 4.15 / Table 4.5 show the speedup of our seven multimedia applications, for logic pipelining trends $0{+}0{+}L_r$ and $1{+}L_p{+}0$, relative to the unpipelined case 0+0+0. It is evident that different applications receive different benefit from logic pipelining, and no one parameterization is best everywhere. Some applications, notably the JPEGs, are particularly well suited to stream-enabled logic pipelining, and benefit with either trend. They continue to speed up at aggregate pipeline depths up to 4, attaining speedups of up to 2.7. Some applications perform substantially better with one trend, notably MPEG Encode IP, which achieves its best speedup of 1.85 with trend $1+L_p+0$. Other applications see little speedup or even non-monotonicity. The average speedup across all seven applications is 1.4-1.5 and is similar for aggregate pipeline depths 2-4 across both trends.

Figure 4.16 / Table 4.6 show the area of our seven multimedia applications, after place and route on XC2VP70, for logic pipelining trends $0+0+L_r$ and $1+L_p+0$. They are included for completeness.

Figure 4.17 / Table 4.7 show the area expansion of our seven multimedia applications, for logic pipelining trends $0+0+L_r$ and $1+L_p+0$, relative to the unpipelined case $0+0+0$. The area overhead of the two trends is substantially different. Trend $0+0+L_r$ rises steeply with L_r, reaching up to 1.51x and averaging 1.31x at depth 4. Trend $1+L_p+0$ is small and relatively flat with L_p for most applications, averaging only 1.08x at aggregate depth 4. This difference verifies that adding and retiming enabled registers (trend $0+0+L_r$, output-side logic relaying) costs more area than adding and retiming D flip-flops (trend $1+L_p+0$, logic pipelining). Retimed enabled registers often leave a residue circuit (a register-multiplexer cycle), whereas retimed D flip-flops do not. Also, retimed enabled registers are less likely to pack well with logic, since they have an additional input (enable) that D flip-flops do not. We conclude that uniform, output-side logic relaying should be avoided in Virtex-II technology, due to its additional area, except in those cases where it provides a real and desired performance advantage, *e.g.* JPEG Encode.

MPEG Encode IP and IPB, unlike our other applications, suffer from a peculiar sensitivity to area constraint. Pipelined or not, they do not perform well when placed and routed in the usual square floorplan that is 20% larger than the area for maximum slice packing. However, they can attain over 100MHz when allowed to consume the entire chip, with no area constraint (see Table 4.18). The cause of this sensitivity to area constraint is not clear. The MPEGs are the largest applications in our application suite. Their sensitivity may be the result of greater graph connectivity, combined with greater application size, resulting in an inability to bring connected components close enough during area-constrained placement.

4.4.2 Page-Specific Logic Pipelining

Uniform pipelining is a simplification, and potentially a costly one. Pipelining every SFSM to the same depth means that some SFSMs are over-pipelined, incurring area overhead, while others are under-pipelined, slowing application performance. Ideally, each SFSM should be pipelined separately, to the minimum depth required to meet a timing target. Separate optimization is entirely feasible within our streaming design methodology, since SFSMs are decoupled by streams. In this section, we develop simple heuristics for separate optimization of SFSMs via page-specific logic pipelining.

It is straightforward to demonstrate that over-pipelining an SFSM leads to unnecessary area overhead and little performance improvement. Figure 4.18 shows the average speedup of 279 pages (one SFSM per page), compiled separately with varying depths of logic pipelining. Figure 4.19 shows, in correspondence, the average area expansion of those 279 pages. In each figure, pages are binned on the horizontal axis by their initial, non-pipelined speed: 0-100MHz, 100-200MHz, and over 200MHz. We find that the greatest speedup occurs for initially slow pages. Initially fast pages, on

	Logic Pipelining $L_i + L_p + L_r$								
Application	0+0+0	0+0+1	0+0+2	0+0+3	0+0+4	1+0+0	1+1+0	1+2+0	1+3+0
IIR	166.3	177.1	179.9	180.8	185.2	161.7	174.8	181.3	181.4
JPEG Decode	47.0	71.4	101.3	117.3	129.9	81.2	106.8	127.9	126.5
JPEG Encode	57.5	76.8	116.9	124.2	119.3	64.0	104.6	106.7	98.4
MPEG Encode IP	46.9	56.4	53.2	55.3	60.4	63.7	86.8	59.1	57.4
MPEG Encode IPB	50.3	63.8							
Wavelet Encode	105.7	126.3	123.9	136.5	127.4	119.4	132.4	118.1	124.1
Wavelet Decode	108.8	117.8	122.7	127.4	119.5	107.8	126.6	115.2	123.2

Table 4.4: Application speed (MHz) with logic pipelining; post-PAR, on XC2VP70.

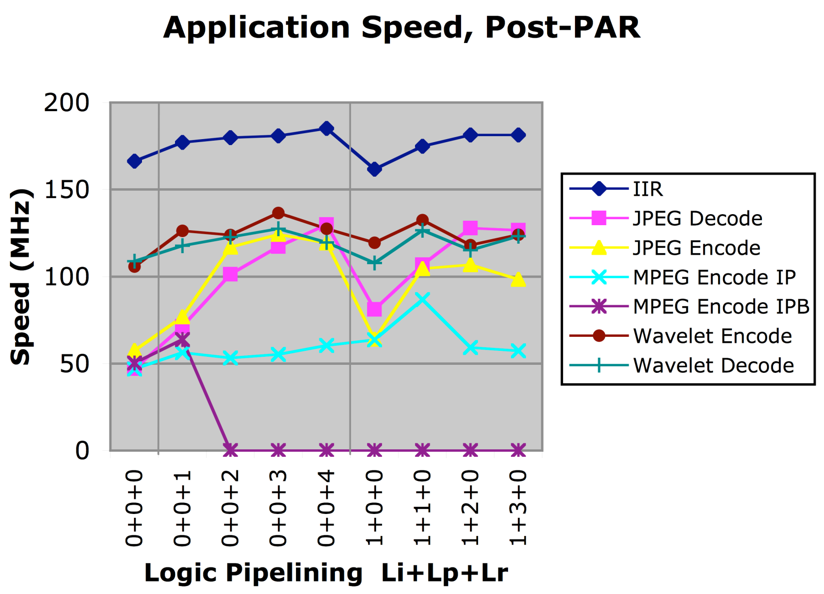


Figure 4.14: Application speed with logic pipelining; post-PAR, on XC2VP70.

	Logic Pipelining $L_i + L_p + L_r$								
Application	0+0+0	0+0+1	0+0+2	0+0+3	0+0+4	1+0+0	1+1+0	1+2+0	1+3+0
IIR	1.00	1.06	1.08	1.09	1.11	0.97	1.05	1.09	1.09
JPEG Decode	1.00	1.52	2.16	2.50	2.77	1.73	2.27	2.72	2.69
JPEG Encode	1.00	1.34	2.03	2.16	2.08	1.11	1.82	1.86	1.71
MPEG Encode IP	1.00	1.20	1.13	1.18	1.29	1.36	1.85	1.26	1.22
MPEG Encode IPB	1.00	1.27							
Wavelet Encode	1.00	1.20	1.17	1.29	1.21	1.13	1.25	1.12	1.17
Wavelet Decode	1.00	1.08	1.13	1.17	1.10	0.99	1.16	1.06	1.13
All (Geo. Mean)	1.00	1.23	1.39	1.48	1.49	1.19	1.51	1.42	1.42

Table 4.5: Application speedup from logic pipelining; post-PAR, on XC2VP70.

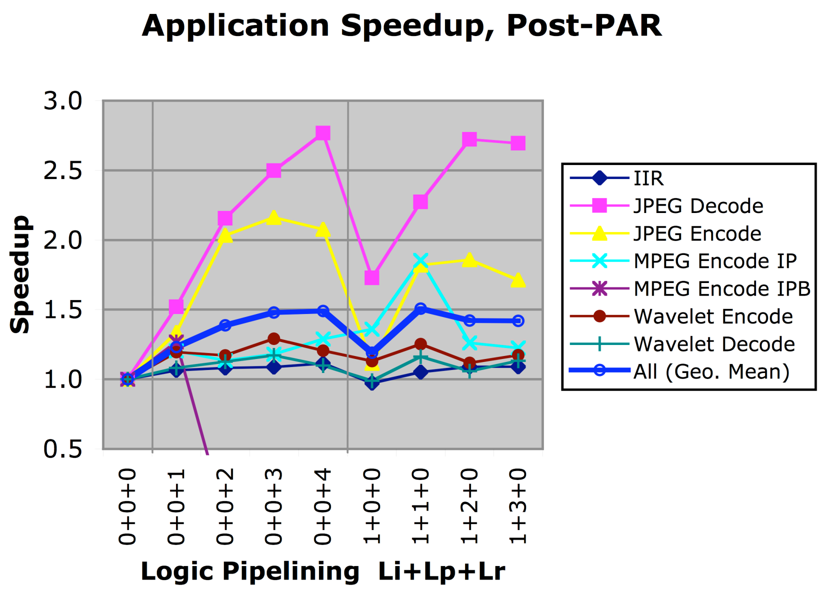


Figure 4.15: Application speedup from logic pipelining; post-PAR, on XC2VP70.

	Logic Pipelining $L_i + L_p + L_r$								
Application	0+0+0	0+0+1	0+0+2	0+0+3	0+0+4	1+0+0	1+1+0	1+2+0	1+3+0
IIR	1,922	1,984	1,984	2,244	2,380	1,922	1,984	2,112	2,312
JPEG Decode	7,442	7,812	9,112	10,224	11,250	7,812	7,812	7,812	7,938
JPEG Encode	6,728	7,080	8,064	8,844	9,522	6,962	7,080	6,962	7,080
MPEG Encode IP	41,472	42,924	45,300	52,800	60,160	45,000	45,602	45,300	46,208
MPEG Encode IPB	65,772	66,912							
Wavelet Encode	8,320	8,450	8,712	9,522	10,512	8,450	8,580	8,320	8,580
Wavelet Decode	8,712	9,112	9,112	9,112	9,248	8,712	9,112	8,978	8,978
Total	140,368	144,274	82,284	92,746	103,072	78,858	80,170	79,484	81,096

Table 4.6: Application area (LUT-FFs) with logic pipelining; post-PAR, on XC2VP70.

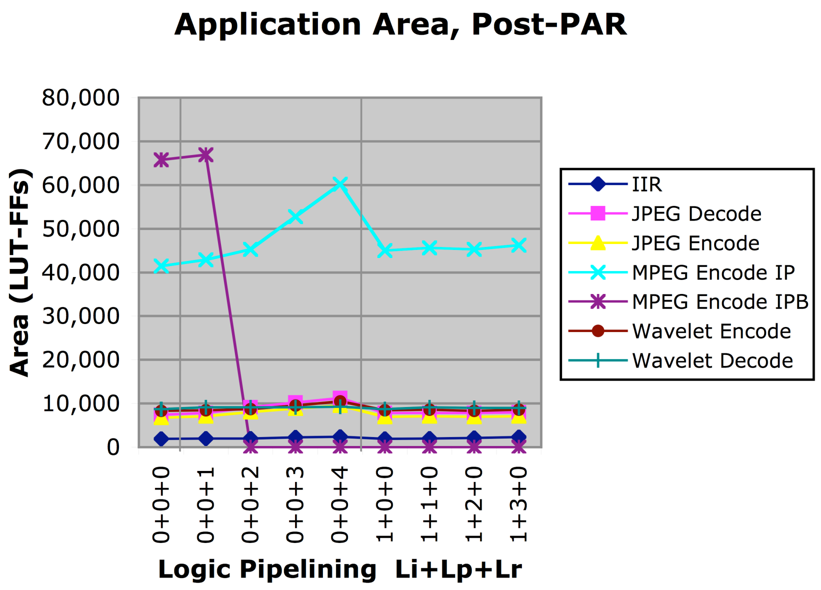


Figure 4.16: Application area with logic pipelining; post-PAR, on XC2VP70.

	Logic Pipelining $L_i + L_p + L_r$								
Application	0+0+0	0+0+1	0+0+2	0+0+3	0+0+4	1+0+0	1+1+0	1+2+0	1+3+0
IIR	1.00	1.03	1.03	1.17	1.24	1.00	1.03	1.10	1.20
JPEG Decode	1.00	1.05	1.22	1.37	1.51	1.05	1.05	1.05	1.07
JPEG Encode	1.00	1.05	1.20	1.31	1.42	1.03	1.05	1.03	1.05
MPEG Encode IP	1.00	1.04	1.09	1.27	1.45	1.09	1.10	1.09	1.11
MPEG Encode IPB	1.00	1.02							
Wavelet Encode	1.00	1.02	1.05	1.14	1.26	1.02	1.03	1.00	1.03
Wavelet Decode	1.00	1.05	1.05	1.05	1.06	1.00	1.05	1.03	1.03
All (Geo. Mean)	1.00	1.04	1.10	1.21	1.31	1.03	1.05	1.05	1.08

Table 4.7: Application area expansion from logic pipelining; post-PAR, on XC2VP70.

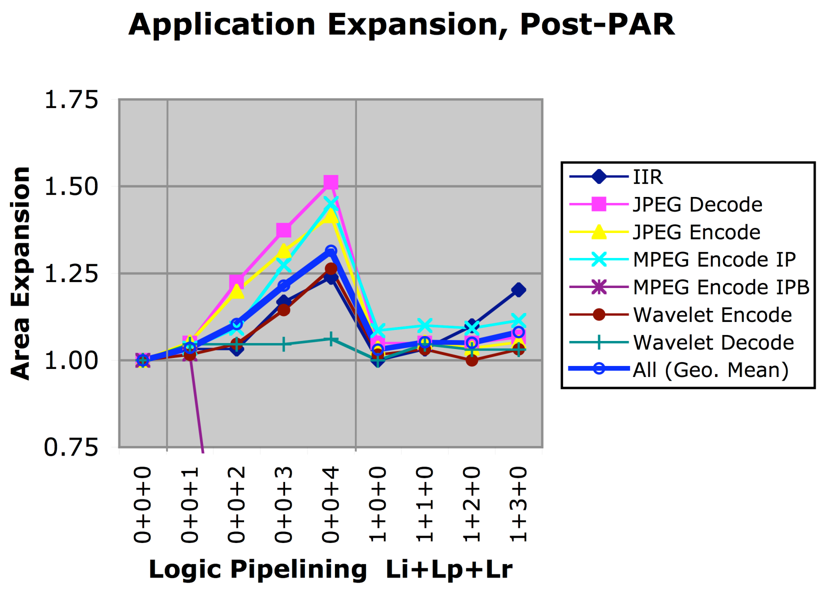


Figure 4.17: Application area expansion from logic pipelining; post-PAR, on XC2VP70.

average, see no speedup or even a slight slowdown. At the same time, those initially fast pages see the greatest area expansion from pipelining—up to 1.9x per page. They need not be pipelined at all. In the uniform pipelining case, such pages contribute unnecessary area overhead.

We propose a heuristic approach to system optimization by separate optimization of each SFSM. We take the simplifying assumption that application speed is precisely the speed of the slowest page. We dub that page the *critical page*. To optimize application speed, it suffices to repeatedly optimize the critical page. Each time a critical page is sped up, a different page becomes critical, and the process repeats. The process ends when the present critical page can no longer be sped up. This procedure for system optimization is codified in Algorithm 4.1.

```
{$P$ is a priority list of pages, sorted by speed, slowest first}
{$p, p'$ are pages}

**repeat**
  $p \leftarrow \mathrm{pop}(P)$
  $p' \leftarrow \mathrm{improve}(p)$
  $\mathrm{insert}(P, p')$
**until** $\mathrm{speed}(p') \leq \mathrm{speed}(p)$
```

Algorithm 4.1: Page-Specific Logic Pipelining

In the algorithm, procedure “improve” refers to optimizing an individual page. The improvement to the page may be incremental, for example, adding one level of pipelining. If the page remains critical, or if it becomes critical again later, it will be incrementally improved again. If no improvement is possible, “improve” does nothing, and system optimization ends. In principle, “improve” may consider any kind of optimization. In practice, it will consider a limited search space, such as choosing a depth for stream-enabled logic pipelining with a particular, single-variable trend.

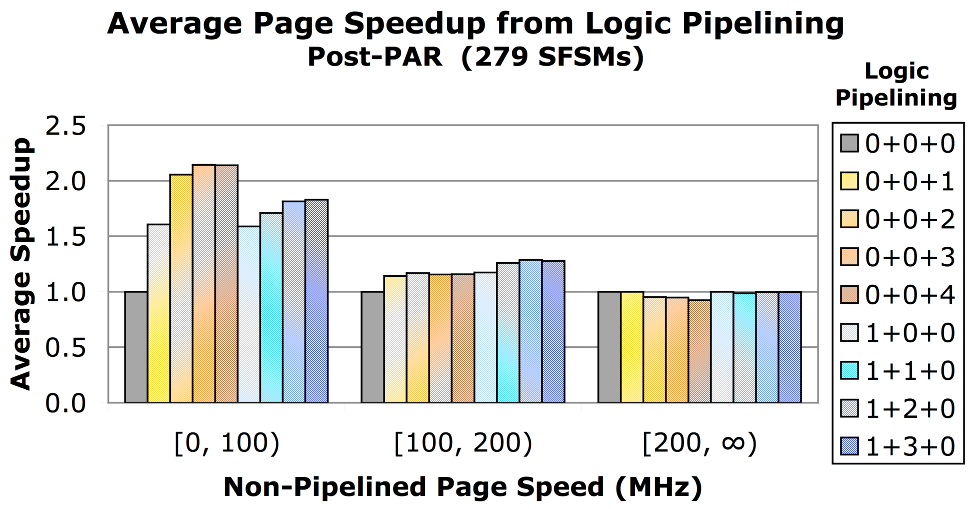


Figure 4.18: Page speedup from logic pipelining, binned by non-pipelined speed; post-PAR, on XC2VP70.

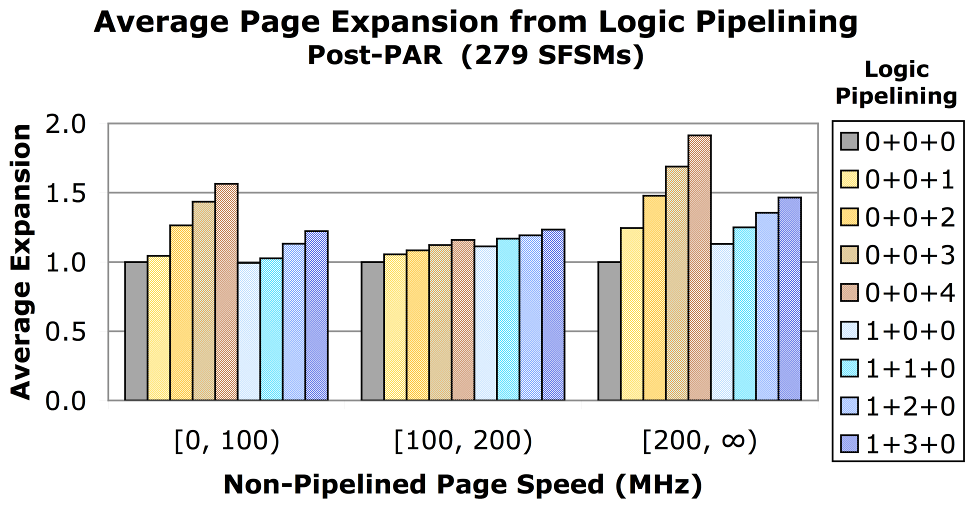


Figure 4.19: Page area expansion from logic pipelining, binned by non-pipelined speed; post-PAR, on XC2VP70.

In the algorithm, procedure “speed” refers to the speed of an individual page. Evaluating speed may involve full recompilation and place and route. Consequently, the fewer number of “improve” steps, the better. For the study below, we ran Algorithm 4.1 with full place and route for each page.

We consider three heuristics to implement “improve”. Each heuristic chooses new parameters for stream-enabled logic pipelining for a particular page. To limit the search space, we consider the trend $0 + 0 + L_r$ with $L_r \leq 4$ and the trend $1 + L_p + 0$ with $L_p \leq 3$. The heuristics are:

Max: Choose the pipelining parameterization that yields maximum page speed. Consider all depths of either trend (brute force search).

Greedy L_r: Increase pipelining depth by one on the trend $0 + 0 + L_r$.

Greedy L_p: Increase pipelining depth by one on the trend $1 + L_p + 0$.

Due to idiosyncrasies in back-end optimizations, page speed does not always increase monotonically with pipelining depth. Consequently, the greedy heuristics for “improve” may add a level of pipelining that actually slows a page down. In such a case, system optimization would end early. This shortcoming may be ameliorated with look-ahead, *i.e.* adding more than one level of pipelining at a time in “improve” to ensure a speed improvement.

For our seven applications, page-specific logic pipelining consistently yields better performance at less area than uniform pipelining. Figures 4.20-4.23 and their corresponding Tables 4.8-4.11 show the results of page-specific as well as uniform logic pipelining for our seven applications, after place and route on XC2VP70. For each application and each pipelining approach, we report critical page speed (Figure 4.20), critical page speedup (Figure 4.21), cumulative page area (Figure 4.22), and cumulative page area expansion (Figure 4.23). Figures 4.24-4.25 replot the geometric mean, across all applications, of speedups and area expansions. In generating these results, Algorithm 4.1 was evaluated based on post-PAR page speeds, and all results are reported post-PAR.

We find that page-specific logic pipelining using "Max" (brute force) produces better performance than every instance of uniform logic pipelining, using less area than most. It yields an average speedup of 1.83 and an average expansion of only 1.15. Heuristic "Greedy L_r" performs slightly worse than "Max" but still outperforms almost every instance of uniform logic pipelining. It yields an average speedup of 1.70 and an even smaller average expansion of 1.09. Heuristic "Greedy L_p" does not yield as high a speedup, though it has the lowest expansion, at 1.03. It is particularly ineffective for the MPEGs, where non-monotonicity causes the search to end early. We also note that JPEG Decode incurs an unusual, higher-than-average area expansion with heuristics "Max" and "Greedy L_r." That expansion owes simply to the fact that the application's critical page(s) (*e.g.* IDCT) comprise a particularly large fraction of the application area.

In summary, we find that page-specific logic pipelining offers an effective, automatic way to pipeline streaming applications for a fairly low area cost. The "Greedy L_r" heuristic offers an average speedup of 1.83 with an average area overhead of 15%.

Separate optimization of SFSMs is not a complete approach to system optimization, since it ignores system level effects. Such system level considerations include:

System clock speed equals critical page speed only if pages are combinationally decoupled. However, with page-specific pipelining, many pages remain unpipelined and may have high combinational delay on their stream connections. Composing such pages may lengthen the critical path and reduce application clock speed—unless streams are explicitly pipelined, *e.g.* using stream-enabled interconnect pipelining. That additional step would decouple pages and ensure the validity of separate page optimization, but it would cost additional area.

System throughput in tokens-per-cycle may be reduced if pipeline delay is added to stream feedback loops. Hence, pipelining should not be used blindly everywhere. We have ignored this case because stream feedback is rare in our applications. Feedback is present inside SFSMs, but those internal loops are not affected by our stream-based logic pipelining.

Stream-based system optimizations may further improve performance and area. Examples include (1) area-time transformations to match the throughputs of connected SFSMs, and (2) SFSM granularity transformations to reduce the cost of stream support.

Chapter 5 discusses system level optimization in detail and revisits some of the issues above.

4.5 Interconnect Pipelining

In this section, we present the results of mapping applications to the XC2VP70 FPGA, using the methodology of Chapter 3, with automatic, stream-enabled pipelining of logic and interconnect. We quantify the benefit and cost of interconnect pipelining. For brevity, we consider only uniform pipelining, where every stream is pipelined identically, and we omit interconnect relaying. Figure 4.12 shows the addition of interconnect pipelining of depth W_p to a hypothetical stream, in relation to other components. Assuming that all streams are pipelined identically, each stream queue must be modified to reserve $L_p + 2W_p$ slots to correct for the staleness of pipelined flow control signals. That is, each queue will assert back-pressure whenever $L_p + 2W_p$ or fewer slots are empty.

	Uniform Logic Pipelining $L_i + L_p + L_r$									Page Specific Pipelining		
Application	0+0+0	0+0+1	0+0+2	0+0+3	0+0+4	1+0+0	1+1+0	1+2+0	1+3+0	Max	Greedy Lr	Greedy Lp
IIR	174.2	162.8	164.7	173.5	173.7	183.1	158.7	162.3	176.1	188.3	174.2	188.3
JPEG Decode	59.2	94.1	114.0	118.3	122.0	75.8	103.1	107.3	125.2	153.0	153.0	125.2
JPEG Encode	59.8	93.3	130.7	127.3	134.7	77.6	108.1	83.7	102.4	146.9	130.7	108.1
MPEG Encode IP	47.3	80.9	102.3	105.1	102.2	51.0	62.2	77.7	83.5	117.5	104.3	51.9
MPEG Encode IPB	39.7	90.3	99.9	100.5	99.4	55.7	73.8	80.2	80.5	118.1	101.6	83.0
Wavelet Encode	118.7	138.6	134.8	131.5	138.3	135.6	131.1	134.4	125.9	138.6	138.6	136.0
Wavelet Decode	127.2	138.0	136.7	124.8	122.9	118.7	136.0	130.4	130.2	146.8	140.6	129.9

Table 4.8: Critical page speed (MHz) with logic pipelining; post-PAR, on XC2VP70.

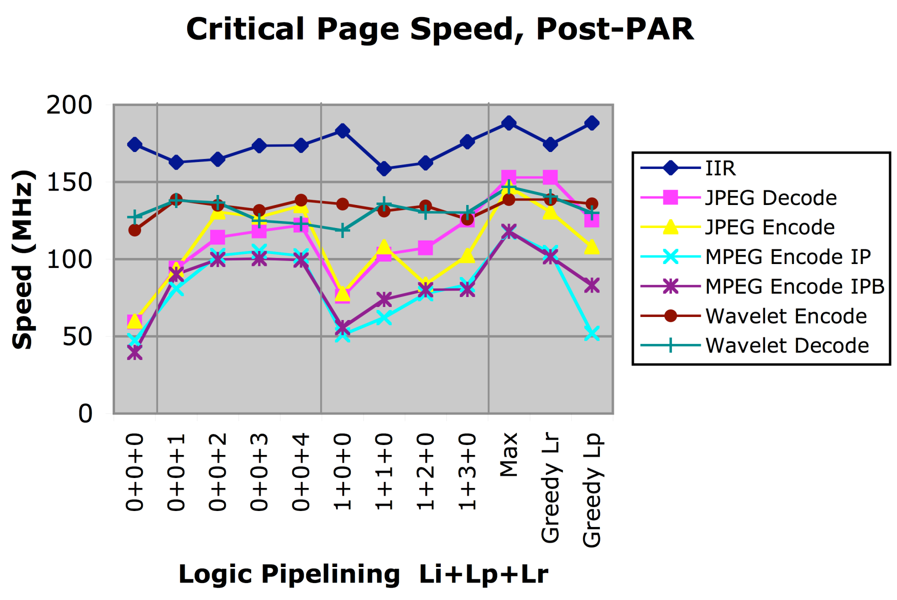


Figure 4.20: Critical page speed with logic pipelining; post-PAR, on XC2VP70.

	Uniform Logic Pipelining $L_i + L_p + L_r$									Page Specific Pipelining		
Application	0+0+0	0+0+1	0+0+2	0+0+3	0+0+4	1+0+0	1+1+0	1+2+0	1+3+0	Max	Greedy Lr	Greedy Lp
IIR	1.00	0.93	0.95	1.00	1.00	1.05	0.91	0.93	1.01	1.08	1.00	1.08
JPEG Decode	1.00	1.59	1.93	2.00	2.06	1.28	1.74	1.81	2.11	2.58	2.58	2.11
JPEG Encode	1.00	1.56	2.18	2.13	2.25	1.30	1.81	1.40	1.71	2.45	2.18	1.81
MPEG Encode IP	1.00	1.71	2.16	2.22	2.16	1.08	1.32	1.64	1.77	2.48	2.21	1.10
MPEG Encode IPB	1.00	2.27	2.52	2.53	2.50	1.40	1.86	2.02	2.03	2.98	2.56	2.09
Wavelet Encode	1.00	1.17	1.14	1.11	1.16	1.14	1.10	1.13	1.06	1.17	1.17	1.15
Wavelet Decode	1.00	1.08	1.07	0.98	0.97	0.93	1.07	1.02	1.02	1.15	1.11	1.02
All (Geo. Mean)	1.00	1.42	1.60	1.59	1.61	1.16	1.35	1.37	1.46	1.83	1.70	1.41

Table 4.9: Critical page speedup from logic pipelining; post-PAR, on XC2VP70.

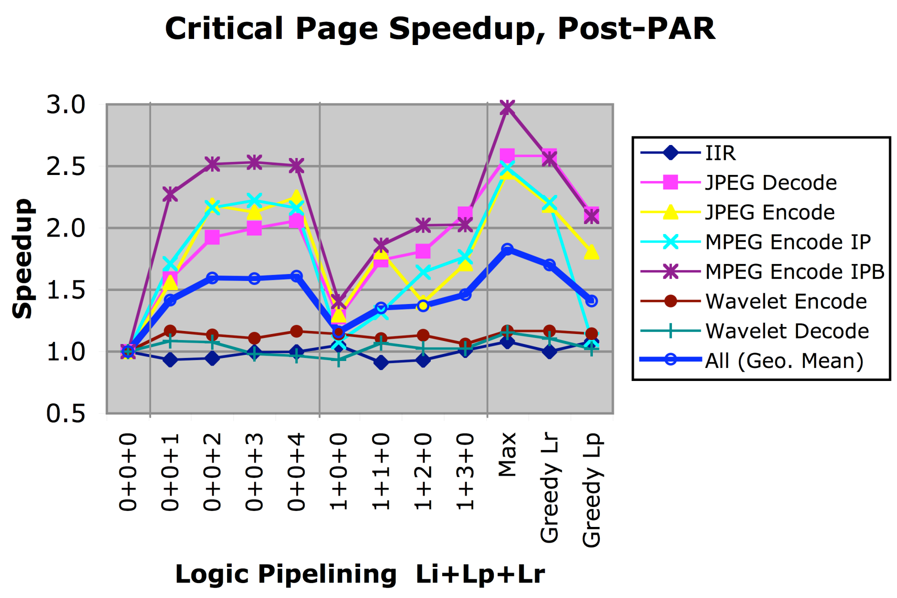


Figure 4.21: Critical page speedup from logic pipelining; post-PAR, on XC2VP70.

	Uniform Logic Pipelining $L_i + L_p + L_r$									Page Specific Pipelining		
Application	0+0+0	0+0+1	0+0+2	0+0+3	0+0+4	1+0+0	1+1+0	1+2+0	1+3+0	Max	Greedy Lr	Greedy Lp
IIR	2.1	2.2	2.4	2.6	2.7	2.2	2.3	2.5	2.6	2.3	2.1	2.2
JPEG Decode	8.1	10.2	11.0	12.4	13.1	8.9	10.2	9.6	9.7	11.3	11.3	8.7
JPEG Encode	7.4	9.1	9.9	10.7	11.3	8.0	8.9	9.0	9.1	8.7	8.3	7.7
MPEG Encode IP	54.5	63.7	72.6	80.9	88.3	61.2	65.5	69.4	72.4	64.6	56.7	54.7
MPEG Encode IPB	80.4	94.0	108.2	120.8	131.6	92.8	99.6	104.9	111.2	94.1	82.1	80.8
Wavelet Encode	9.7	10.6	11.5	12.3	13.5	10.0	10.7	11.0	11.6	10.5	10.0	9.8
Wavelet Decode	9.8	10.6	11.1	11.4	12.2	10.4	10.8	11.0	11.7	10.1	10.4	9.9
All (Geo. Mean)	172.1	200.4	226.6	251.2	272.8	193.4	208.1	217.4	228.2	201.6	180.9	173.8

Table 4.10: Cumulative page area (thousands of LUT-FFs) with logic pipelining; post-PAR, on XC2VP70.

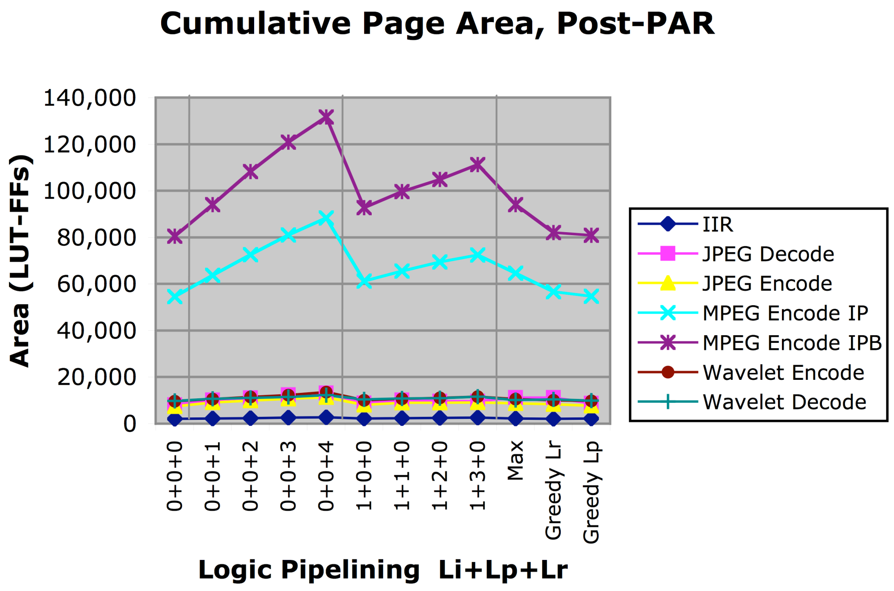


Figure 4.22: Cumulative page area with logic pipelining; post-PAR, on XC2VP70.

	Uniform Logic Pipelining $L_i + L_p + L_r$									Page Specific Pipelining		
Application	0+0+0	0+0+1	0+0+2	0+0+3	0+0+4	1+0+0	1+1+0	1+2+0	1+3+0	Max	Greedy Lr	Greedy Lp
IIR	1.00	1.06	1.12	1.21	1.29	1.03	1.10	1.19	1.25	1.07	1.00	1.05
JPEG Decode	1.00	1.26	1.36	1.54	1.62	1.10	1.26	1.19	1.20	1.39	1.39	1.08
JPEG Encode	1.00	1.24	1.34	1.45	1.53	1.08	1.21	1.21	1.22	1.18	1.13	1.05
MPEG Encode IP	1.00	1.17	1.33	1.48	1.62	1.12	1.20	1.27	1.33	1.19	1.04	1.00
MPEG Encode IPB	1.00	1.17	1.35	1.50	1.64	1.15	1.24	1.30	1.38	1.17	1.02	1.00
Wavelet Encode	1.00	1.09	1.19	1.27	1.39	1.03	1.10	1.14	1.20	1.09	1.04	1.02
Wavelet Decode	1.00	1.08	1.13	1.16	1.24	1.05	1.10	1.12	1.19	1.03	1.06	1.00
All (Geo. Mean)	1.00	1.15	1.25	1.37	1.47	1.08	1.17	1.20	1.25	1.15	1.09	1.03

Table 4.11: Cumulative page area expansion from logic pipelining; post-PAR, on XC2VP70.

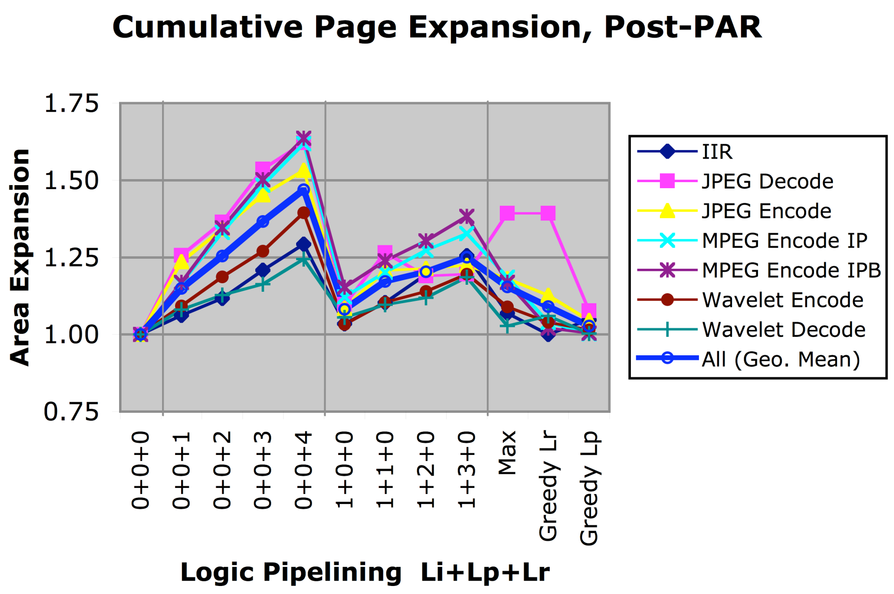


Figure 4.23: Cumulative page area expansion from logic pipelining; post-PAR, on XC2VP70.

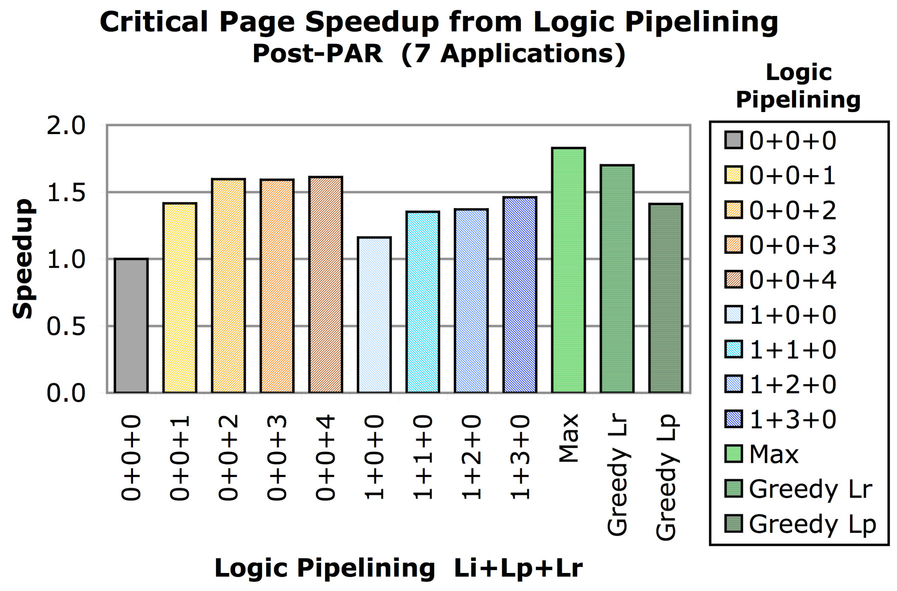


Figure 4.24: Average critical page speedup from logic pipelining (geometric mean of 7 applications); post-PAR, on XC2VP70.

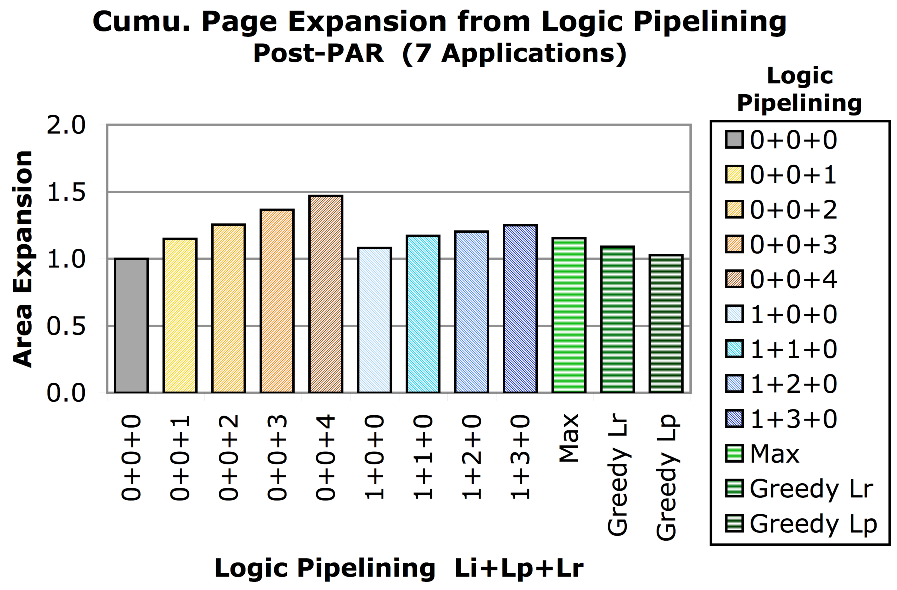


Figure 4.25: Average cumulative page area expansion from logic pipelining (geometric mean of 7 applications); post-PAR, on XC2VP70.

Routing delay is an increasingly dominant limiter to the performance of modern systems. As CMOS device technology scales, logic delay improves faster than wire delay. The resulting disparity forces designers to choose between (1) scaling the clock rate to match the slower scaling of wire delay, or (2) scaling the clock rate to match the faster scaling of logic delay but reaching fewer transistors in each clock cycle and having to pipeline deeper.

Modular design can ameliorate the problem of routing delay, since a well-packed, small module will use shorter routes. For our streaming applications, we can demonstrate that the routing delay of an application's critical page, compiled separately, is usually lower than the routing delay of the application compiled as a whole. Table 4.12 shows the routing delay in each application's critical path, ranging from 45% to 56%, depending on logic pipelining. These routing delays are in line with the Xilinx *Constraints Guide* [Xilinx, 2003a] (p. 147), which states that on FPGAs, "Routing delays typically account for 45% to 65% of the total path delays." Table 4.13 shows the routing delay of each application's critical page. Table 4.14 and Figure 4.26 show the ratio of application routing delay to critical page routing delay. That ratio is greater than one for most applications, with most instances of logic pipelining. An application's clock speed would be improved if its high routing delay could be reduced to that of its critical page.

To benefit from reduced routing delay in modular design, we must ensure that system composition does not lengthen the critical path. First, we must ensure that each module has little or no combinational logic delay on its interfaces. Second, we need a way to pipeline long module-to-module routes. Both requirements can be met using stream-enabled interconnect pipelining. Stream-enabled logic pipelining does not suffice on its own, since (1) it may leave combinational logic delay on output streams, and (2) it does not directly address long routes. Interconnect pipelining, on the other hand, places non-retimable registers directly on stream wires, breaking the

	Logic Pipelining $L_i + L_p + L_r$								
Application	0+0+0	0+0+1	0+0+2	0+0+3	0+0+4	1+0+0	1+1+0	1+2+0	1+3+0
IIR	77.6%	31.9%	48.1%	45.1%	73.4%	32.4%	61.0%	48.1%	39.8%
JPEG Decode	58.7%	57.1%	47.3%	57.0%	51.1%	42.0%	36.1%	57.7%	63.4%
JPEG Encode	48.1%	56.3%	33.8%	40.7%	42.9%	61.1%	39.6%	39.2%	45.3%
MPEG Encode IP	49.5%	82.1%	77.3%	69.3%	83.7%	59.4%	49.9%	69.5%	75.1%
MPEG Encode IPB	50.9%	65.5%							
Wavelet Encode	31.3%	36.3%	39.4%	31.3%	33.9%	38.5%	33.9%	45.7%	46.8%
Wavelet Decode	36.9%	40.8%	44.2%	54.1%	53.4%	36.9%	62.3%	52.2%	58.4%
Average	50.4%	52.9%	48.4%	49.6%	56.4%	45.1%	47.1%	52.1%	54.8%

Table 4.12: Routing delay in application critical path, with logic pipelining; post-PAR, on XC2VP70.

	Logic Pipelining $L_i + L_p + L_r$								
Application	0+0+0	0+0+1	0+0+2	0+0+3	0+0+4	1+0+0	1+1+0	1+2+0	1+3+0
IIR	58.8%	32.9%	30.3%	33.2%	31.8%	31.1%	32.3%	33.1%	33.5%
JPEG Decode	37.5%	37.9%	71.6%	77.0%	73.0%	62.5%	53.4%	52.9%	38.7%
JPEG Encode	41.0%	56.8%	35.9%	38.5%	37.5%	55.7%	57.6%	49.7%	35.4%
MPEG Encode IP	57.1%	46.1%	52.8%	39.0%	39.2%	65.3%	68.6%	64.5%	48.4%
MPEG Encode IPB	67.0%	39.6%	41.0%	41.3%	42.3%	69.7%	53.1%	49.2%	65.6%
Wavelet Encode	26.3%	29.6%	34.2%	35.7%	30.7%	29.0%	31.3%	31.3%	35.6%
Wavelet Decode	30.1%	37.9%	46.2%	42.9%	57.2%	33.6%	46.9%	39.6%	50.8%
Average	45.4%	40.1%	44.6%	43.9%	44.5%	49.6%	49.0%	45.8%	44.0%

Table 4.13: Routing delay in critical page, with logic pipelining; post-PAR, on XC2VP70.

	Logic Pipelining $L_i + L_p + L_r$								
Application	0+0+0	0+0+1	0+0+2	0+0+3	0+0+4	1+0+0	1+1+0	1+2+0	1+3+0
IIR	1.32	0.97	1.59	1.36	2.31	1.04	1.89	1.45	1.19
JPEG Decode	1.57	1.51	0.66	0.74	0.70	0.67	0.68	1.09	1.64
JPEG Encode	1.17	0.99	0.94	1.06	1.14	1.10	0.69	0.79	1.28
MPEG Encode IP	0.87	1.78	1.46	1.78	2.14	0.91	0.73	1.08	1.55
MPEG Encode IPB	0.76	1.65							
Wavelet Encode	1.19	1.23	1.15	0.88	1.10	1.33	1.08	1.46	1.31
Wavelet Decode	1.23	1.08	0.96	1.26	0.93	1.10	1.33	1.32	1.15
All (Geo. Mean)	1.13	1.28	1.08	1.13	1.26	1.00	0.99	1.17	1.34

Table 4.14: Ratio: application routing delay vs. critical page routing delay, with logic pipelining; post-PAR, on XC2VP70.

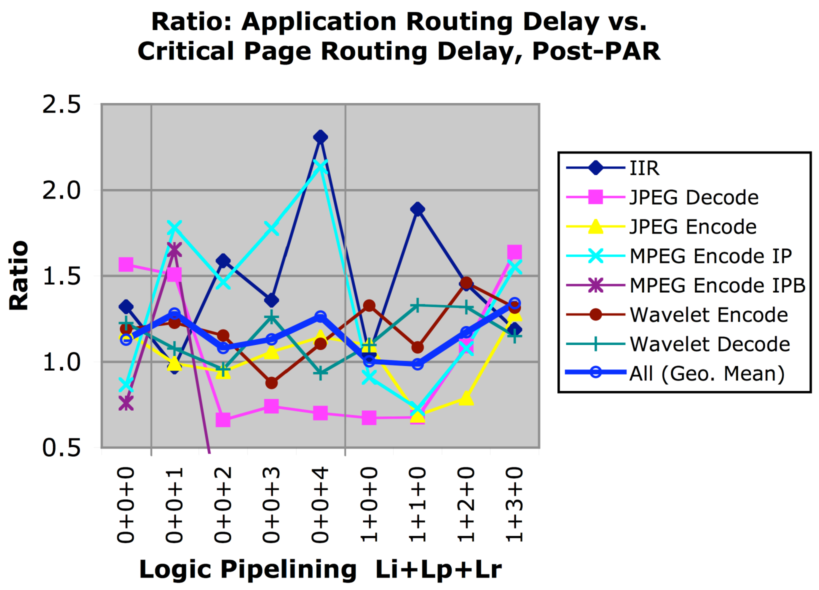


Figure 4.26: Ratio of routing delay in application versus critical page, with logic pipelining; post-PAR, on XC2VP70.

combinational paths of logic as well as routing.

Figure 4.27 / Table 4.15 show speedups for our seven applications using interconnect pipelining of depths $W_p = 1$ and 2, relative to the case of no interconnect pipelining ($W_p = 0$). We consider several instances of aggressive, uniform logic pipelining, since routing delay is more dominant at higher clock speeds. Surprisingly, we find little speedup from interconnect pipelining, not more than 4%, and in some cases yielding a slowdown. We have verified that this limitation is *not* due to (1) non-monotonic, poor optimizations in synthesis or technology mapping, nor to (2) poor slice packing of interconnect pipelining registers, which may tie the registers to producer or consumer logic and make them less mobile during placement[2]. Interestingly, this limitation seems to be a consequence of using a constrained floorplan for placement. The results reported above use a square floorplan that is 20% larger than the area required for minimum slice packing. Removing the floorplan and using the entire chip enables more meaningful speedups for interconnect pipelining. It has the double benefit of improving speed for the case of no interconnect pipelining and simultaneously enabling more benefit with interconnect pipelining.

Figure 4.28 / Table 4.16 show speedups for our seven applications using interconnect pipelining of depths $W_p = 1$ and 2, relative to the case of no interconnect pipelining ($W_p = 0$), placed and routed with no area constraint on XC2VP70. Figure 4.29 / Table 4.17 show the corresponding area expansions. For logic pipelining trend $0 + 0 + L_r$, we find that a single level of interconnect pipelining works best. It yields an average speedup of 15%, that number being heavily biased by a high speedup to MPEG Encode IP. A second level of interconnect pipelining produces less speedup, apparently because its greater area suffers the same congestion effects as the constrained floorplan above. For logic pipelining trend $1 + L_p + 0$, we find that interconnect pipelining produces nearly no speedup on average. Nevertheless, with this trend, most applications are already faster than with the other trend, even before interconnect pipelining.

Our results above indicate that interconnect pipelining produces a relatively small performance improvement on Virtex II-Pro. A more meaningful performance improvement is had simply by providing more area for place-and-route. Area-constrained PAR suffers from two problems that lead to poorly performing circuits. First, high logic utilization prevents a placer from bringing all connected components close together, creating some unnecessarily distant paths (we have observed paths crossing half the chip or more). Second, routing congestion leads to longer routes and, in some cases, to allocating long, slow wires when short, fast wires would have sufficed. This

[2]It is possible to constrain slice packing so that interconnect pipelining registers are never packed with producer or consumer logic. Such a constraint would make the registers fully mobile during placement, and ostensibly help to overcome routing delay. In practice, this approach did not substantially improve the speed of any of our application except the MPEGs. Furthermore, this approach incurs an area overhead proportional to the number of streams.

wire allocation problem is specific to segmented routing architectures, where long wires are slower due to higher capacitance. Our stream-enabled interconnect pipelining may exacerbate these problems by consuming additional resources and creating higher utilization and congestion.

There are several ways to reduce utilization and congestion in a fixed area, to better enable interconnect pipelining. First, we can use page-specific logic pipelining in place of uniform logic pipelining, since it requires less area. Second, we can consider changing the way in which interconnect registers are allocated. In the conventional tool flow for Virtex/Spartan FPGAs, interconnect registers are allocated directly in the netlist, packed into slices (with logic if possible), then placed and routed. Weaver *et al.* [Weaver *et al.*, 2003] propose changing the tool chain to allocate interconnect registers *after* placement. Register need would be determined by post-placement retiming, estimating routing delay from Euclidean distance in the placement. Free registers would then be scavenged from the existing placement and connected by incremental rerouting. This approach has almost no area overhead and would suit us well.

For completeness, we include the absolute speed and area of our applications with interconnect pipelining. Figure 4.30 / Table 4.18 show the speeds of our seven applications using interconnect pipelining of depths $W_p = 1$ and 2, relative to the case of no interconnect pipelining ($W_p = 0$), placed and routed with no area constraint on XC2VP70. Figure 4.31 / Table 4.19 show the corresponding areas.

	Logic Pipelining $L_i + L_r + L_p$ / Interconnect Pipelining W_p											
	0+0+2			0+0+3			1+2+0			1+3+0		
Application	0	1	2	0	1	2	0	1	2	0	1	2
IIR	1.00	1.02	1.03	1.00	1.00	0.96	1.00	1.04	0.95	1.00	1.02	0.99
JPEG Decode	1.00	1.04	1.15	1.00	1.17	0.90	1.00	0.92	0.93	1.00	0.99	0.77
JPEG Encode	1.00	0.98	1.06	1.00	0.85	1.05	1.00	0.97	0.94	1.00	1.05	1.01
MPEG Encode IP	1.00	1.05	0.95	1.00	0.94	1.05	1.00	1.03	0.83	1.00	1.03	1.12
MPEG Encode IPB												
Wavelet Encode	1.00	1.03	1.06	1.00	0.89	0.90	1.00	1.05	1.10	1.00	1.06	1.01
Wavelet Decode	1.00	1.03	1.00	1.00	0.92	0.93	1.00	1.04	1.09	1.00	0.97	0.98
All (Geo. Mean)	1.00	1.02	1.04	1.00	0.95	0.96	1.00	1.01	0.97	1.00	1.02	0.97

Table 4.15: Application speedup from interconnect pipelining; post-PAR, on XC2VP70.

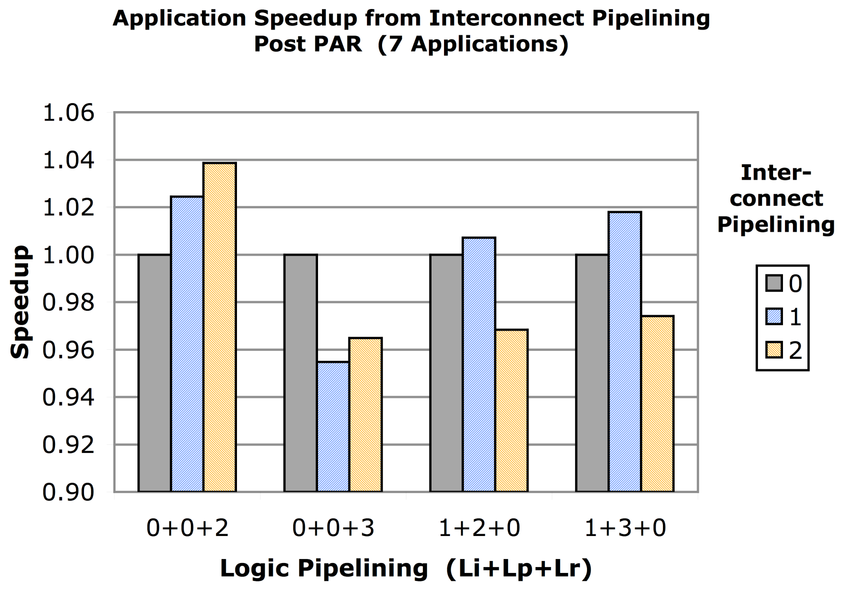


Figure 4.27: Average application speedup from interconnect pipelining (mean of 7 applications); post-PAR, on XC2VP70.

	Logic Pipelining $L_i + L_r + L_p$ / Interconnect Pipelining W_p											
	0+0+2			0+0+3			1+2+0			1+3+0		
Application	0	1	2	0	1	2	0	1	2	0	1	2
IIR	1.00	0.95	1.00	1.00	1.07	1.03	1.00	1.02	0.97	1.00	1.01	0.99
JPEG Decode	1.00	1.13	1.04	1.00	0.87	0.81	1.00	1.17	1.17	1.00	1.27	1.04
JPEG Encode	1.00	1.13	1.10	1.00	0.93	0.83	1.00	1.02	1.03	1.00	0.92	1.00
MPEG Encode IP	1.00	1.63	1.00	1.00	2.86	2.45	1.00	1.01	0.95	1.00	1.05	0.93
MPEG Encode IPB							1.00					
Wavelet Encode	1.00	1.07	1.04	1.00	0.99	0.94	1.00	0.97	0.99	1.00	1.00	1.02
Wavelet Decode	1.00	1.08	1.05	1.00	0.96	1.00	1.00	0.97	1.03	1.00	0.77	0.99
All (Geo. Mean)	1.00	1.15	1.04	1.00	1.15	1.08	1.00	1.02	1.02	1.00	0.99	0.99

Table 4.16: Application speedup from interconnect pipelining; post-PAR without area constraint, on XC2VP70.

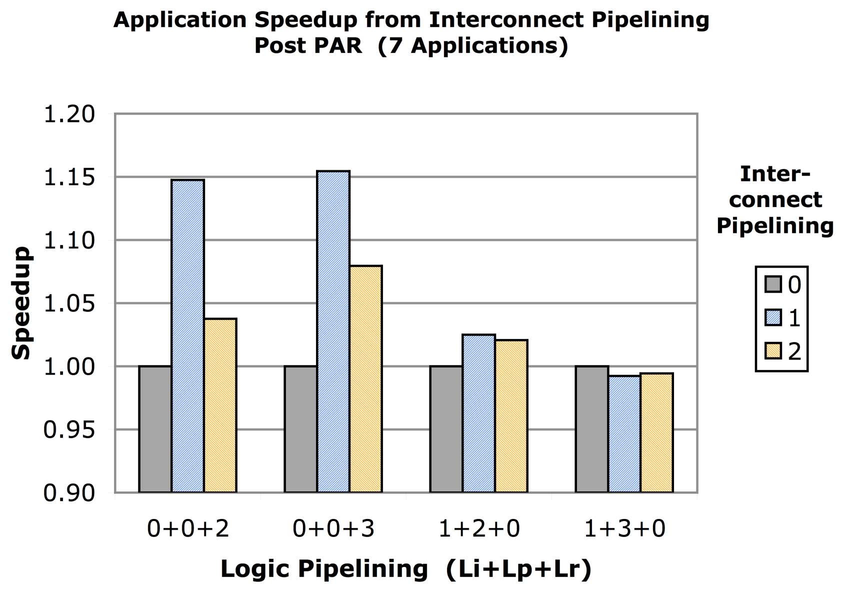


Figure 4.28: Average application speedup from interconnect pipelining (mean of 7 applications); post-PAR without area constraint, on XC2VP70.

	Logic Pipelining $L_i + L_r + L_p$ / Interconnect Pipelining W_p											
	0+0+2			0+0+3			1+2+0			1+3+0		
Application	0	1	2	0	1	2	0	1	2	0	1	2
IIR	1.00	1.11	1.20	1.00	1.07	1.16	1.00	1.07	1.11	1.00	1.05	1.11
JPEG Decode	1.00	0.94	1.15	1.00	0.96	0.96	1.00	1.07	1.15	1.00	1.10	1.22
JPEG Encode	1.00	0.92	1.12	1.00	1.07	1.16	1.00	1.10	1.16	1.00	1.06	1.14
MPEG Encode IP	1.00	1.13	1.27	1.00	1.13	1.13	1.00	1.08	1.14	1.00	1.06	1.13
MPEG Encode IPB	1.00						1.00	1.00		1.00		
Wavelet Encode	1.00	1.05	1.15	1.00	1.06	1.15	1.00	1.09	1.15	1.00	1.07	1.12
Wavelet Decode	1.00	1.06	1.15	1.00	1.08	1.15	1.00	1.07	1.15	1.00	1.07	1.14
All (Geo. Mean)	1.00	1.03	1.17	1.00	1.06	1.12	1.00	1.07	1.14	1.00	1.07	1.14

Table 4.17: Application area expansion from interconnect pipelining; slice-packed without area constraint, on XC2VP70.

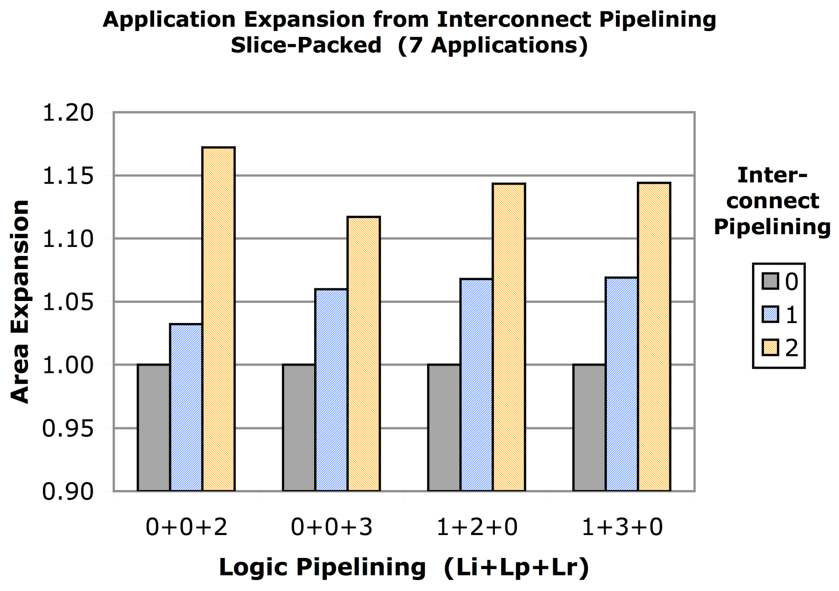


Figure 4.29: Average application area expansion from interconnect pipelining (mean of 7 applications); slice-packed without area constraint, on XC2VP70.

	Logic Pipelining $L_i + L_r + L_p$ / Interconnect Pipelining W_p											
	0+0+2			0+0+3			1+2+0			1+3+0		
Application	0	1	2	0	1	2	0	1	2	0	1	2
IIR	182	173	181	172	185	177	182	186	177	184	186	182
JPEG Decode	101	114	105	132	115	107	123	145	144	119	151	123
JPEG Encode	105	119	116	125	117	104	105	106	107	111	102	111
MPEG Encode IP	59	96	59	29	83	71	101	101	96	99	104	92
MPEG Encode IPB							44					
Wavelet Encode	128	138	134	135	133	126	133	129	131	129	130	131
Wavelet Decode	119	129	125	132	127	132	126	123	130	126	97	125

Table 4.18: Application speed (MHz) with interconnect pipelining; post-PAR without area constraint, on XC2VP70.

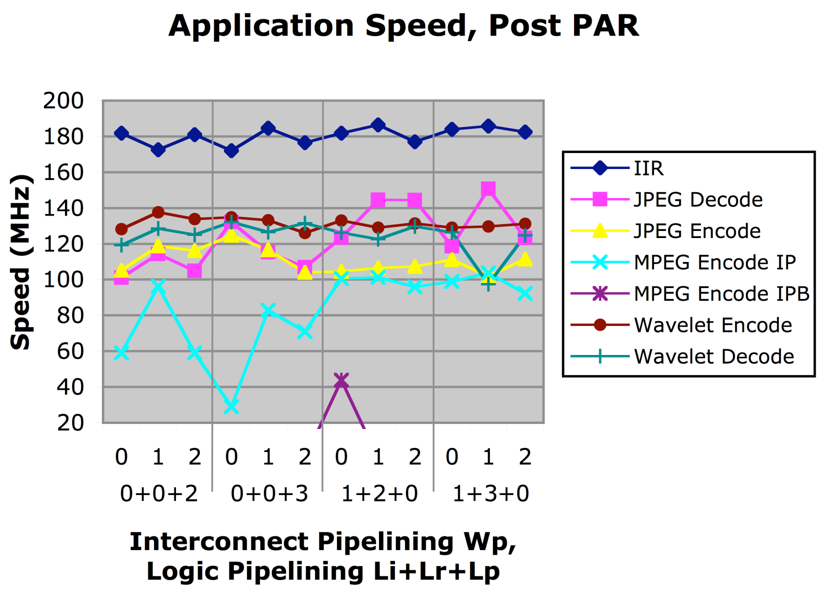


Figure 4.30: Application speed with interconnect pipelining; post-PAR without area constraint, on XC2VP70.

	Logic Pipelining $L_i + L_r + L_p$ / Interconnect Pipelining W_p											
	0+0+2			0+0+3			1+2+0			1+3+0		
Application	0	1	2	0	1	2	0	1	2	0	1	2
IIR	2.3	2.5	2.7	2.5	2.7	2.9	2.5	2.6	2.7	2.6	2.8	2.9
JPEG Decode	10.8	10.1	12.4	12.0	11.5	11.5	8.5	9.1	9.8	8.8	9.6	10.7
JPEG Encode	9.5	8.8	10.7	10.1	10.8	11.7	7.2	7.9	8.4	8.0	8.4	9.1
MPEG Encode IP	52.2	59.0	66.2	58.4	65.9	66.2	48.0	51.6	54.7	51.7	54.9	58.4
MPEG Encode IPB	66.2						66.2	66.2		66.2		
Wavelet Encode	10.3	10.8	11.8	11.2	11.8	12.9	9.0	9.8	10.4	9.9	10.5	11.1
Wavelet Decode	9.3	9.9	10.7	9.9	10.6	11.3	9.4	10.1	10.8	10.1	10.8	11.5
Total	160.6	101.1	114.6	104.0	113.2	116.5	150.7	157.3	96.8	157.2	97.1	103.7

Table 4.19: Application area (thousands of LUT-FF cells) with interconnect pipelining; slice-packed without area constraint, on XC2VP70.

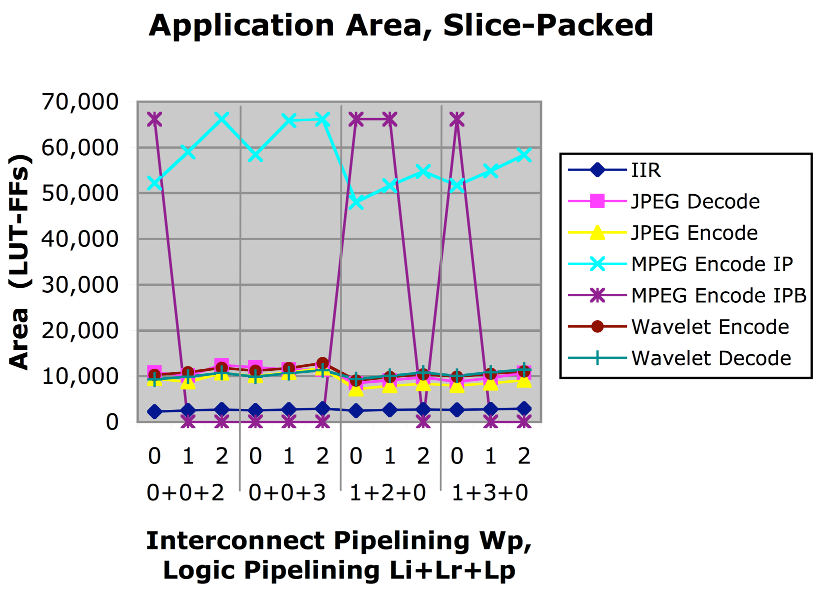


Figure 4.31: Application area with interconnect pipelining; slice-packed without area constraint, on XC2VP70.

4.6 Summary

We have used the methodology of Chapter 3 to compile seven multimedia applications to a commercial FPGA. The applications are written in TDF and structured as stream-connected process networks. Individual processes, or SFSMs, range in complexity from simple token fanout to DCTs and Huffman coders. The applications are characterized by having a mostly feed-forward flow of stream connections, making them suitable for deep, system level pipelining.

Baseline compilation, without stream based optimizations, revealed the following. Stream flow control is fast and cheap on an FPGA. The FSMs responsible for stream flow control comprise 6% of total page area and are never the limit to performance. Stream queues are moderately expensive, comprising 38% of total page area. They limit performance in about 1/3 of all pages, those being the fastest pages. The dominant component remains the datapaths, which comprise 58% of total page area and limit performance in 2/3 of all pages. Application performance is typically dominated by one or a few slow datapaths that merit optimization.

Stream-enabled logic pipelining is an automated approach to optimizing SFSMs by systematically adding registers to stream links and retiming them into the datapath of the producer or consumer. Uniform pipelining, where all streams are pipelined identically, provides average application speedups of 1.4x-1.5x using aggregate pipelining depths of 2-4. We concentrate on two trends for the type and location of registers: (1) trend $0+0+L_r$ uses L_r levels of enabled registers on datapath outputs; (2) trend $1 + L_p + 0$ uses L_p levels of D flip flops on datapath outputs plus one level of enabled register on datapath inputs. The two trends provide comparable performance but very different areas, owing to the different type of registers used. Retiming D flip-flops requires near zero area in FPGA technology, since the registers are packed together with logic. Retiming enabled registers, on the other hand, leaves a residue, typically a multiplexer-register pair, which requires additional logic cells. Accordingly, the D flip flop trend, $1 + L_p + 0$, incurs an area cost of only 5% for its best depths $1+1+0$, $1+2+0$. The enabled register trend, $0+0+L_r$, incurs an area cost proportional to pipelining depth, which is substantially larger: 10% and 21% for its best depths $0+0+2$, $0+0+3$ (these so-called best depths are those that yield most of the available speedup—not necessarily the absolute maximum—without undue area overhead).

Page-specific pipelining, where each SFSM is pipelined separately, yields higher performance and lower area than uniform pipelining. We optimize a system incrementally by repeatedly repipelining the presently slowest page, and we consider three heuristics for page repipelining. The "Greedy L_r" heuristic, which adds one level of pipelining in trend $0+0+L_r$, yields an average application speedup of 1.41x with an area overhead of only 3%—a speedup on par with uniform pipelining but in less area. The "Greedy L_p" heuristic, which adds one level of pipelining in trend $1 + L_p + 0$, yields an average application speedup of 1.7x with an area overhead of 9%—an area on par with uniform pipelining but with greater speedup. The exhaustive page optimizer "Max" beats wins in both speedup and area, yielding an average application speedup of 1.83x with an area overhead of 15%.

Stream-enabled interconnect pipelining is an automated approach to pipelining long, on-chip interconnect, by systematically adding registers to stream wires. We find this approach to be very sensitive to area and routing congestion. It provides negligible speedup using a minimum square floorplan but meaningful speedup when using the entire chip. With logic pipelining depths $0 + 0 + 2$ and $0 + 0 + 3$, a single level of interconnect pipelining provides 15% speedup for an area cost of 3% and 6%, respectively. A second level of interconnect pipelining provides less, rather than more, speedup.

Our methodology is hampered by two problems in the commercial FPGA tool flow. First is the problem of non-monotonicity in retiming-based pipelining. When compiling behavioral verilog, adding one level of retimable pipelining registers may actually decrease performance. Thus, performance is non-monotonic with respect to pipelining depth. This non-monotonicity impairs efficient searching of the pipelining parameter space for optimization. We hypothesize that the non-monotonicity is due to idiosyncratic interactions between retiming and other optimizations during behavioral synthesis and technology mapping. Such other optimizations include timing-driven replication and LUT covering.

Second is the problem of interconnect register allocation. Existing FPGAs, including the Xilinx Virtex/Spartan series, do not have architectural interconnect registers. Instead, they rely on registers in the user design, along with timing-driven placement, to reduce long routes. Registers that are explicitly allocated for pipelining the interconnect must be packed with logic before placement, incurring extra area, and without knowledge of actual interconnect delay. Our stream enabled interconnect pipelining methodology, which allocates registers in that manner, thus has limited benefit. A better approach would be to allocate interconnect pipelining registers *after* placement, as in [Weaver *et al.*, 2003]. With that approach, performance would be improved by choosing register locations with knowledge of routing delay, and area would be saved by scavenging unused registers from the existing placement.

Chapter 5

System Optimization

In this chapter, we discuss a number of system optimizations that are specific to streaming systems. In general, such optimizations are not possible or practical with RTL-based systems, whose behavior is too unstructured and hard to analyze. A streaming discipline, enforced by a structured language, guarantees more limited behavior that is amenable to analysis and optimization. We can exploit the timing independence of process networks in optimizations that change the cycle-by-cycle behavior of modules, module interfaces, and communication. Possible optimizations include not only pipelining of stream communication, but also module pipelining and area-time transformations. The stream graph topology is a key resource for guiding those transformations, since it, together with the stream discipline, exposes inter-module dataflow and available system concurrency. To guide those transformations, we rely on the stream graph. We are thus able to analyze system-wide behaviors, throughputs, latencies, and even deadlock.

A primary challenge in stream based design is choosing stream parameters. Streams are buffered collectively in queues and pipeline registers. A designer normally prefers to keep those structures as small as possible, to save area and power[1]. However, stream buffering must be large enough to: (1) avoid deadlock due to buffer overflow, (2) pipeline communication over long distances on chip, (3) balance pipeline delays on reconvergent paths, and (4) smooth out dynamic variations in consumption/production rates. In traditional, RTL based design, communication buffering is chosen manually, and properties (1)-(4) are verified by inspection or simulation. Nevertheless, in a disciplined streaming paradigm, it is possible to apply some analysis and automation to the problem. We systematically tackles issues (1)-(3). Section 5.1 describes how we factor the issues into separate analyses. Section 5.2 proposes a compiler analysis to find minimum, deadlock-free buffer sizes. Section 5.5 proposes a retiming-based approach for balancing pipeline delays on reconvergent paths, to avoid throughput loss due to data misalignment in time. Section 5.3 proposes a stream-aware approach for placement, which models the effect of pipelined interconnect delay on system throughput.

Automatic module pipelining is normally not possible in RTL based design, since pipelining changes the timing behavior of a module. We showed in Chapter 3 that a streaming discipline accommodates several forms of module pipelining, thanks to its timing independence. The choice of pipelining *depth* poses an interesting trade-off, since deeper pipelining may improve clock rates but actually degrade system performance due to pipeline bubbles. An RTL description does not distinguish pipeline control and bubbles from other structures, so it gives a compiler no basis for optimization of pipeline depths. A streaming system, on the other hand, associates pipeline

[1]Keeping stream buffers small allows a design to use smaller queue implementations. For example, in our synthesis methodology of Chapter 3:

- a queue of depth in the hundreds uses block RAM;
- a queue of depth in the tens uses SRL16 shift registers;
- a queue of depth two uses D flip-flops as shift registers;
- a queue of depth one uses an enabled register and a gate;
- a queue of depth zero implies that the producer and consumer can be merged with no queue, or that intermediate results can be pipelined in a depth-one, enabled register queue.

bubbles with the presence or absence of tokens, so it is amenable to an analysis of pipelined throughput and system performance. In Section 5.4 we show how to use the throughput models developed for stream-aware placement in order to identify throughput-critical processes and decide how deep to pipeline them. Combined with module synthesis, our approach can estimate the point at which additional pipelining begins to degrade system performance.

Another challenge in large system design is how to take advantage of the fact that some components operate infrequently or are altogether idle for long periods. A designer may wish to serialize or time-share non-critical components to save area, or to turn them off to save power. An RTL compiler cannot infer most intermittent behavior, since it must make the conservative assumption that every wire, register, and combinational component is active at every clock cycle. Consequently, a designer must verify intermittent behavior by inspection or simulation and manually implement any transformations. In a streaming dataflow paradigm, wires are replaced by streams, and token flow determines module activity. It becomes possible to construct analytic or approximate models for activity rates of modules and streams. Those models may then be used to guide area saving transformations. In Section 5.6, we discuss how to model activity in process networks and use it to guide serialization transformations.

Synchronous Dataflow (SDF) scheduling techniques [Bhattacharyya *et al.*, 1996] [Gaudiot and Bic, 1991] provide a key starting point for modeling throughput and activity in process networks. For example, if a measure of *tokens per firing* is available for each actor, then SDF *balance equations* may be used to determine actor activity across an entire system. We apply and adapt SDF techniques with *average* rates to model first order process network behavior.

The analyses and optimizations discussed in this chapter are sketches. We do not present complete implementations. Instead, we develop conceptual foundations and theory for the proposed analyses, and we show preliminary results where possible.

5.1 System Optimization Flow

Given a process network $G = (V, E)$, we can define the choice of stream parameters as three assignments:

- An assignment of logic pipelining depths: $L : E \rightarrow \mathbb{N}$,
- An assignment of interconnect pipelining depths: $W : E \rightarrow \mathbb{N}$,
- An assignment of stream buffer sizes: $B : E \rightarrow \mathbb{N}$.

Logic pipelining L corresponds to the pipelining methodology of Sections 3.7.3 and 3.7.4, wherein registers are added to streams and retimed into processes. The aggregate pipelining depth L may be broken into: $L = L_i + L_p + L_r$, where L_i is input-side logic relaying, L_r is output-side logic relaying, and L_p is D flip-flop based logic pipelining.

Interconnect pipelining W corresponds to the pipelining methodology of Sections 3.7.2 and 3.7.1. The aggregate pipelining depth W may be broken into: $W = W_p + W_r$, where W_r is interconnect relaying, and W_p is D flip-flop based interconnect pipelining. Interconnect relaying is implemented in relay stations, which are two element queues, so W_r will be a multiple of two.

Stream buffer size B refers to the total buffering in queues, including a stream's main queue and any relays. Pipelining in D flip-flops also provides buffering, but we need not tally it separately, since it is mirrored in the reservation capacity of a downstream queue. The aggregate buffer size can be broken into: $B = B_q + L_i + L_r + W_r$, where B_q is the size of the stream's main queue, and the other components are from relaying. Buffering in relays is fixed for particular purposes, but B_q can provide excess capacity as needed. Buffering must be large enough to accomplish several goals, for correctness and performance:

1. The main queue must provide sufficient reservation capacity to balance D flip-flop based pipelining: $B_q \geq L_p + 2W_p$. Using the shift register queue from Section 3.4.2, we further need an excess of one to avoid deadlock and an excess of two for full throughput: $B_q \geq L_p + 2W_p + 2$.

2. Total buffering B must be large enough to avoid artificial deadlock from buffer overflow.

3. Total buffering B should be large enough to balance pipeline delays of reconvergent paths and realign data in time.

4. Total buffering B should be large enough to smooth out dynamic variations in consumption/production rates.

We now propose an optimization flow that chooses L, W, and B in multiple phases. Each phase is intended to meet a particular requirement of L, W, B, among those described above. The only requirement not addressed in this flow is buffering to smooth dynamic rates (4). The optimization flow is as follows:

1. Serialization	
2. Buffer bounding	(affects B)
3. Process pipelining	(affects L)
4. Pipeline balancing (for logic)	(affects B)
5. Synthesis	
6. Stream aware place and route	(affects W)
7. Pipeline balancing (for interconnect)	(affects B)
8. Buffer parameter setting	(affects B)

Serialization. (Section 5.6) This pass serializes infrequent and non-critical processes and streams to save area.

Buffer Bounding. (Section 5.2) Deadlock avoidance can be formulated as finding a lower bound on the size of every stream buffer. Any additional buffering is harmless to correctness and can be used to meet other requirements, such as pipelining. We propose an approach to buffer bounding based on state space enumeration.

Process Pipelining. (Section 5.4) Logic pipelining depths are chosen with two goals in mind: (1) improve system clock rate, and (2) avoid decreasing system throughput in tokens-per-cycle. We extend the pipeline depth selection approach from Section 4.4.2, which addresses (1), with a throughput model to address (2).

Pipeline Balancing. (Section 5.5) If pipeline delays on reconvergent paths are not balanced, then data may arrive mismatched at the point of reconvergence, leading to stalling and throughput loss. We address this by providing additional buffering on the different paths. We provide a retiming-based approach for resizing stream buffers to balance pipelines.

Synthesis. (Chapter 3) This step generates a netlist suitable for place and route, via translation to Verilog and behavioral synthesis.

Stream Aware Place and Route. (Section 5.3) Interconnect pipelining depths are chosen after placement to match actual interconnect delay. In addition, we modify the placement engine to know that stream connections may be pipelined and need not be kept localized. Feed forward streams can be pipelined arbitrarily deep without affecting system throughput. However, feedback streams cannot. We provide throughput models to identify which streams are most critical to throughput and should not be pipelined.

Buffer Parameter Setting. This final step ensures that stream buffers are large enough to provide reservation for D flip-flop based pipelining: $B_q \geq L_p + 2W_p + 2$. It then sets the reservation parameters of every stream's main queue. Since this step happens after queues have been placed and routed, it must be implemented as template specialization. That is, each queue is specialized by modifying register initial values in its placed configuration.

5.2 Stream Buffer Bounds Analysis

There is a fundamental, semantic gap between process network abstract models and implementations. The abstract model provides conceptually unbounded buffering for streams, but an implementation must use finite buffering. An implementation buffer that is too small may overflow and deadlock a computation that would otherwise continue. We call this induced, overflow related deadlock a *bufferlock*, following the terminology of SCORE [Caspi *et al.*, 2000a]. A designer, compiler, or run-time system is then faced with the task of choosing buffer sizes that are sufficiently large to avoid bufferlock. This task is tractable for restricted models such as SDF. For TDFPN and any Turing complete process network model, this task is, in general, intractable, since determining the memory bounds of a Turing machine is undecidable. Nevertheless, the memory requirements of most programs are bounded *a-priori* by subtle constraints in compression, communication, or file protocols (*e.g.* maximum word or block sizes in compression). Consequently, designers have always been able to choose buffer sizes manually. We seek to automate that process.

In this section, we describe an analysis to choose minimum, deadlock free, stream buffer sizes for process networks. We first address the existence of minimum sizes and why they are independent of other implementation parameters. We devise a test for bufferlock based on state space exploration, using an adaptation of Henzinger's interface automata [de Alfaro and Henzinger, 2001]. We reformulate the dynamic scheduling methodology of Parks [Parks, 1995] into an abstract interpretation and demonstrate how to use it to choose static buffer sizes. The proposed analysis may fail to determine static bounds for some streams, since that job is technically undecidable. For those particular streams, the compiler must ask a designer for explicit buffer sizes.

5.2.1 Existence of Buffer Bounds

A process network imposes a partial order on the schedule of actor firings, based on token flow. Any particular schedule consumes and produces tokens in a particular order, and thus requires a particular amount of buffering resources. Likewise, a particular amount of buffering resources constrains the possible choice of schedules. In this sense, stream buffers behave like any resource that constrains a scheduling problem. Our synthesis methodology in Chapter 3 produces a self-scheduled implementation, where actors fire concurrently as soon as flow control allows. Nevertheless, that scheduling policy, like any other, is still limited by stream buffer space.

Parks [Parks, 1995] notes that expanding a stream buffer may increase, but not decrease, the set of possible schedules. That is, the addition of a new buffer slot permits all of the schedules that were possible before, and possibly new ones. If a particular assignment of buffer sizes induces bufferlock using a particular scheduling policy, then one may try larger buffers to enable more schedules under the same policy. The actual scheduling policy is immaterial. Parks specifically suggests always increasing the smallest full buffer after encountering bufferlock. If a schedule exists that works with finite buffers, then this approach will eventually find it[2]. We refer to this approach as *Parks' algorithm.*

We refer to a bufferlock-free schedule as a *permissible* schedule. We refer to a buffer size assignment that produces permissible schedules as a *permissible* buffer size assignment. Note that a buffer size assignment may be permissible under one scheduling policy but not under another.

The first, *i.e.* least, permissible buffer size assignment under Park's algorithm

[2] It is possible that *no* permissible schedule exists using finite memory under a particular scheduling policy, or perhaps under *any* policy. Programs which require truly infinite memory are, in a sense, uninteresting, since they are not practically computable. An interesting question is whether a compiler analysis could recognize such programs. The answer, in general, is no, due to the undecidability of memory bounds for process networks.

forms a lower bound for permissible buffer sizes. Any additional expansion is guaranteed to produce permissible schedules under the same scheduling policy.

In a hardware implementation of process networks, the choice of queue implementation and stream pipelining comprise part of the scheduling policy. Those choices affect delay, and thus schedule. However, Park's algorithm is agnostic of schedule and thus agnostic of those choices. Using the stream pipelining methodology of Chapter 3, adding some forms of stream pipelining requires expanding the downstream buffer. Nevertheless, if that buffer were already large enough to support permissible schedules, expanding it cannot hurt. In short, a permissible assignment of buffer sizes (assuming one exists) can be chosen before and independently of any other stream parameters.

5.2.2 Undecidability of Buffer Bounds

An interesting question is whether a compiler analysis can be constructed to determine stream buffer bounds. The absolute lowest bounds are not necessary—any bounds will suffice to guarantee bufferlock-free execution. Given such a guarantee, a compiler can synthesize a reliable implementation in near minimum resources. Unfortunately, it is theoretically impossible to construct such a compiler analysis for the general case. That is, determining a permissible, bounded memory schedule for a process network is undecidable. Buck [Buck, 1993] proved this undecidability for BDF. Parks [Parks, 1995] extended that result to DFPN and KPN by implementing BDF. The same result applies for TDFPN, since TDFPN can implement BDF (Theorem 2.3.1).

Theorem 5.2.1. *The problem of deciding whether a BDF graph can be scheduled with bounded memory is undecidable.*

Proof. The theorem and proof are due to Buck [Buck, 1993]. Buck proved that BDF can implement a Universal Turing Machine (UTM). It follows that determining whether a BDF graph deadlocks, *i.e.* terminates, in bounded memory is at least as hard as determining whether a UTM accesses a bounded or unbounded length of tape. Buck proved that an algorithm for the latter problem (UTM bounded memory) could be used to solve the halting problem, which is known to be undecidable. Consequently, the BDF bounded memory problem is undecidable. □

Theorem 5.2.2. *The problem of determining whether a TDFPN program can run without bufferlock using bounded stream buffers is undecidable.*

Proof. This follows from the undecidability of BDF (Theorem 5.2.1) and the reducibility of BDF to TDFPN (Theorem 2.3.1). The phrasing "can run without bufferlock using finite stream buffers" is equivalent to Buck's phrasing "can be scheduled in bounded memory." □

The undecidability of stream buffer bounds means that no static, *i.e.* compile time, choice of buffer sizes can universally guarantee bufferlock-free execution. There are several ways to deal with this undecidability. One is to provide a dynamic, rather than static, mechanism for choosing buffer sizes. Parks' algorithm [Parks, 1995] and SCORE [Caspi *et al.*, 2000a] use a simple approach of expanding buffers at run-time whenever bufferlock occurs. However, buffer reallocation is not practical in a digital circuit implementation of process networks. General memory allocation is a task best left for a microprocessor, which is available in Park's implementation as well as SCORE. Our task is somewhat simpler, since we specifically allocate queues[3]. Nevertheless, dynamic buffer allocation in hardware would incur a high area overhead. so we prefer to use only static allocation.

[3] One may envision a dynamic queue allocation scheme in hardware using a bank of queues to provide spill capacity for bufferlocked streams. A controller would need to detect bufferlock, identify a bufferlocked stream, and reroute that stream to a spill queue. Queues may be chained to provide additional capacity.

The undecidability of stream buffer bounds means only that *some* buffers in *some* programs cannot be bounded. For example, a TDFPN program might have an SDF equivalent subgraph, which is certainly bounded. It is useful to bound as many buffers as possible, even if some remain unbounded. The remaining set of unbounded buffers may be small enough to afford dynamic allocation in hardware. Alternatively, the remaining set of unbounded buffers can be presented to a designer in request for a manual specification of buffer sizes. Thus, an automatic but imperfect analysis of stream buffer bounds can still reduce the amount of work required by a designer.

5.2.3 Abstract Parks Algorithm

Parks' algorithm for dynamically scheduling process networks in bounded memory is roughly as follows. Let $G = (V, E)$ be a process network graph and $b : E \to \mathbb{N}$ be a buffer size assignment. Try to evaluate G with b. If the evaluation bufferlocks, then increase b and try again. The prescribed increase of b is to expand the size of the smallest full buffer by one.

Park's algorithm is structured around a decision procedure that determines whether G bufferlocks with a particular b. Call it $\mathit{live}(G, b)$. While Park's original intent was to evaluate G and collect its results, the same approach can be used with a pure decision procedure and no results. Suppose we had a compiler analysis for $\mathit{live}(G, b)$, *i.e.* an abstract interpretation rather than a full evaluation. Parks' algorithm could then be used to find permissible buffer bounds for G by starting with $b \equiv 0$ and iterating to increase b. As noted above, such a decision procedure would be undecidable in general. However, a conservative version of the procedure could be constructed which assumes bufferlock whenever it is too difficult to prove liveness. If the procedure determines liveness, then Parks' algorithm is done. If the procedure determines or assumes bufferlock, then Park's algorithm tries another iteration. After enough iterations, Park's algorithm gives up. The original, run-time version would give up due to lack of buffer memory. The abstract, compile-time version might give up due to lack of memory for the analysis, or due to lack of time. It stands to reason that, as buffers grow larger, the analysis becomes more expensive in memory and/or time. Thus, it makes sense that an inconclusive analysis would give up at some large enough b.

In the next few sections, we describe an abstract analysis for $live(G, b)$ based on state space enumeration and automata composition. We first discuss pairwise composition, then efficient, whole system composition. This analysis, in conjunction with the abstract Park's algorithm, provides a conservative analysis for stream buffer size bounds. Applying the analysis to separate partitions of a system gives it the opportunity to bound at least some streams. Thus, it can provide partial assistance to a designer, even if it cannot bound all streams.

We note that Parks' algorithm does not distinguish intrinsic deadlock from artificial deadlock (bufferlock). Intrinsic deadlock is unrelated to buffer sizes and would happen even with infinite buffers. A simple example is to connect a source of *false* tokens to an actor that requires *true* tokens. Another example is any feedback loop of actors that contains no initial token, *i.e.* no SDF-style ideal delays and no actor willing to emit a first token. Parks' algorithm would apply increasingly larger buffers and never find a non-deadlocked solution. The special case of deadlock with no full buffers is easy to identify as intrinsic (as in the feedback loop example above). However, in the general case, intrinsic deadlock would create plenty of full buffers upstream of the point of deadlock. Some restricted models of process networks do have static analyses for determining intrinsic deadlock (*e.g.* the SDF notion of *liveness* [Bhattacharyya *et al.*, 1996]). However, we are not aware of any such analysis for general process networks.

5.2.4 Deadlock Analysis via Automata Composition

A process network with finite buffers is a system with finite state. Consequently, it should be amenable to formal verification techniques based on state space enumeration. In particular, we are interested in determining whether a given system has deadlock states, *i.e.* states that are not enabled to fire. Recall that a process is enabled to fire (in a given state) if its desired inputs are available, and if its intended outputs will fit in its output buffers. We say that a system is enabled to fire (in a given state) if some actor is enabled. A system is deadlocked if no actor is enabled. This interpretation is valid with functional processes, as in SDF, BDF, and DFPN, as well as with stateful processes, as in TDFPN and KPN. A system always has state, namely in in its stream buffers, even if the individual processes do not.

To simplify deadlock analysis, we use an abstract interpretation of system state that does not include data values. We are concerned only with data presence. This is a reasonable simplification for TDFPN, where firing rules are based only on presence, not value. Abstract system state will include queue occupancy and the FSM state of SFSMs. Abstract state transitions will reflect token consumption and production. Any dynamic decisions based on data values will be modeled as non-deterministic transitions.

We develop a finite state automaton and automata composition to represent abstract process state and token flow. Our automata are based on de Alfaro and Henzinger's *interface automata* [de Alfaro and Henzinger, 2001], which in turn are based on Lynch's *I/O automata* [Lynch and Tuttle, 1987]. We begin with a direct use of interface automata. We then generalize to *multi-action* interface automata, which more closely match our operational semantics and yield smaller structures for analysis.

5.2.5 Interface Automata for Process Networks

We use de Alfaro and Henzinger's interface automata [de Alfaro and Henzinger, 2001] to model token flow in process networks.

5.2.5.1 Automaton Definition

An *interface automaton* P is a tuple:

$$P = (V_P, V_P^i, \mathcal{A}_P^I, \mathcal{A}_P^O, \mathcal{A}_P^H, \mathcal{T}_P) \tag{5.1}$$

where:

- V_P is a set of states,
- $V_P^i \subset V_P$ is a singleton set of *initial* states,
- $\mathcal{A}_P^I, \mathcal{A}_P^O, \mathcal{A}_P^H$ are mutually disjoint sets of *input*, *output*, and *internal* (hidden) actions,
- $\mathcal{A}_P = \mathcal{A}_P^I \cup \mathcal{A}_P^O \cup \mathcal{A}_P^H$ is the set of all actions, and
- $\mathcal{T}_P \subseteq V \times \mathcal{A}_P \times V$ is a set of *transitions*.

An action $a \in \mathcal{A}_P$ is *enabled* in state $v \in V_P$ if a transition $(v, a, v') \in \mathcal{T}_P$ exists. That transition may represent token consumption ($a \in \mathcal{A}_P^I$), token production ($a \in \mathcal{A}_P^O$), or an internal change not involving token flow ($a \in \mathcal{A}_P^H$). Let $\mathcal{A}_P^I(v)$, $\mathcal{A}_P^O(v)$, $\mathcal{A}_P^H(v)$ respectively denote the set of input, output, and internal actions enabled at state $v \in V$, and let $\mathcal{A}_P(v) = \mathcal{A}_P^I(v) \cup \mathcal{A}_P^O(v) \cup \mathcal{A}_P^H(v)$. A state need not be enabled to accept all inputs, *i.e.* $\mathcal{A}_P^I(v) = \mathcal{A}_P^I$ is not required. A state with no enabled actions is a *terminal* state: $v \in V$ s.t. $A_P(v) = \emptyset$. An *execution fragment* e of P is a possibly infinite, alternating sequence of states and actions, $e = [v_0, a_0, v_1, a_1, ...]$, such that $(v_i, a_i, v_{i+1}) \in \mathcal{T}_P \; \forall i$. This definition of execution permits a non-deterministic choice when multiple transitions are available in a state.

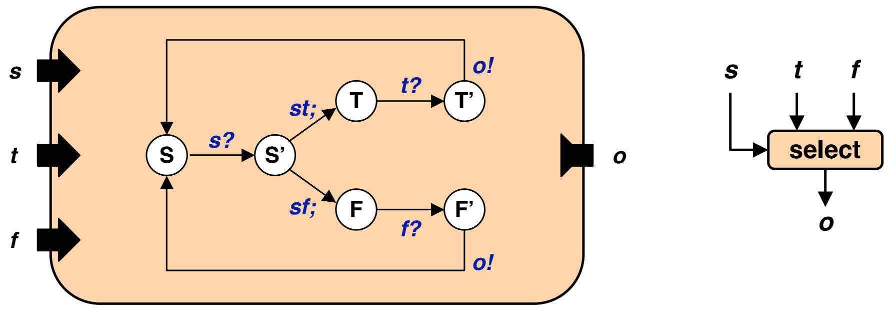


Figure 5.1: Interface automaton for BDF *select* actor

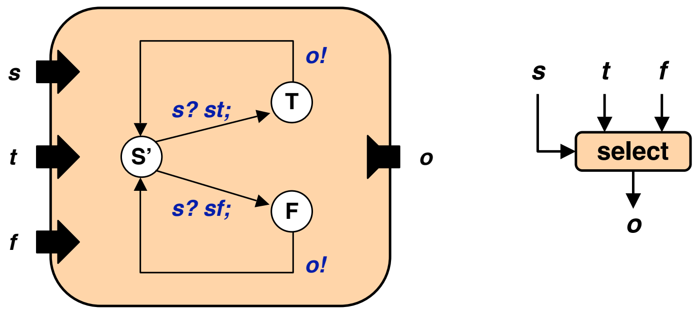


Figure 5.2: Multi-action interface automaton for BDF *select* actor

Figure 5.1 shows a sample, single-action automaton for the BDF *select* actor. Each node represents an automaton state $v \in V_{\text{select}}$, and each arc represents a transition $(v, a, v') \in \mathcal{T}_{\text{select}}$ labeled by a. Using CSP notation, we denote an input action as "a?", an output action as "a!", and an internal action as "a;". The automaton shown is derived from the three-state TDF implementation of *select* from Figure 2.4. Interface automata permit only on action per transition. To construct the automaton, each TDF state was split into a sequence of automaton states separated by a single action. Actions "s?", "t?", "f?" denote consuming from input streams s, t, f. Action "o!" denotes producing to output stream o. Actions "st;", "sf;" denote the internal action of determining whether the most recent value consumed from s is true or false. A more natural translation from TDF would permit multiple actions per transition, as in Figure 5.2. We will extend interface automata to deal with such multi-action transitions in Section 5.2.6.

5.2.5.2 Automata Composition

Automata may be composed to represent synchronized execution. Dataflow between automata is represented by having an output action in one automaton match an input action in the other automaton. The composition is restricted so that both component automata must evaluate that shared action simultaneously. These composition semantics represent a CSP-style rendezvous. To represent buffered streams, we will later introduce additional automata for queues.

Two interface automata P, Q are *composable* if:

$$\begin{aligned} \mathcal{A}_P^H \cap \mathcal{A}_Q &= \emptyset, & \qquad \mathcal{A}_P^I \cap \mathcal{A}_Q^I &= \emptyset, \\ \mathcal{A}_P \cap \mathcal{A}_Q^H &= \emptyset, & \qquad \mathcal{A}_P^O \cap \mathcal{A}_Q^O &= \emptyset. \end{aligned}$$

That is, P and Q may not share a common internal action, input action, or output action. However, an output action of one may be an input action of the other. Such shared actions represent dataflow between the automata and become internal actions in the composition. Let $\mathit{shared}(P, Q) = \mathcal{A}_P \cap \mathcal{A}_Q$ be the set of shared actions. Under the restrictions above, we have: $\mathit{shared}(P, Q) = (\mathcal{A}_P^O \cap \mathcal{A}_Q^I) \cup (\mathcal{A}_P^I \cap \mathcal{A}_Q^O)$.

The composition of two composable interface automata P, Q is a product automaton:

$$P \otimes Q = (V_{P\otimes Q}, V^i_{P\otimes Q}, \mathcal{A}^I_{P\otimes Q}, \mathcal{A}^O_{P\otimes Q}, \mathcal{A}^H_{P\otimes Q}, \mathcal{T}_{P\otimes Q}) \tag{5.2}$$

where:

- $V_{P\otimes Q} = V_P \times V_Q$,
- $V^i_{P\otimes Q} = V^i_P \times V^i_Q$,

- $\mathcal{A}^I_{P\otimes Q} = (\mathcal{A}^I_P \cup \mathcal{A}^I_Q) \setminus \mathit{shared}(P,Q)$,
- $\mathcal{A}^O_{P\otimes Q} = (\mathcal{A}^O_P \cup \mathcal{A}^O_Q) \setminus \mathit{shared}(P,Q)$,
- $\mathcal{A}^H_{P\otimes Q} = (\mathcal{A}^H_P \cup \mathcal{A}^H_Q) \cup \mathit{shared}(P,Q)$,
- $\mathcal{T}_{P\otimes Q}$ is defined by:

$$\begin{aligned}
\mathcal{T}_{P\otimes Q} &= \{((u,v),a,(u',v')) \mid (u,a,u') \in \mathcal{T}_P \ \wedge \ a \notin \mathit{shared}(P,Q) \ \wedge \ v \in V_Q\} \\
&\cup \{((u,v),a,(u',v')) \mid (v,a,v') \in \mathcal{T}_Q \ \wedge \ a \notin \mathit{shared}(P,Q) \ \wedge \ v \in V_P\} \\
&\cup \{((u,v),a,(u',v')) \mid (u,a,u') \in \mathcal{T}_P \ \wedge \ (v,a,v') \in \mathcal{T}_Q \ \wedge \ a \in \mathit{shared}(P,Q)\}
\end{aligned}$$

Although interface automata resemble FSMs, a product automaton is not the same as a direct product of FSMs. A product FSM allows independent and unsynchronized transitions for all components. A product automaton restricts transitions to enforce synchronization. Thus, a product automaton may have as many states as a product FSM but only a subset of its transitions. We can safely ignore any unreachable states in the product automaton.

Figure 5.3 shows an example automata composition for a pair of processes connected by a single stream x. The producer A alternates between consuming primary input i and producing x. The consumer B alternates between consuming x and producing primary output o. In the composition, A and B run in lock step, simultaneously producing and consuming x.

Figure 5.4 shows an example automata composition for a pair of processes connected by two streams, x and y. The two processes access x and y in an incompatible order and subsequently deadlock. Specifically, the producer A produces x before y, while the consumer B consumes y before x. The resulting composition has few permissible transitions and many unreachable states, shown in grey. Deadlock manifests as terminal states that have no outgoing transitions, shown in red. There is one

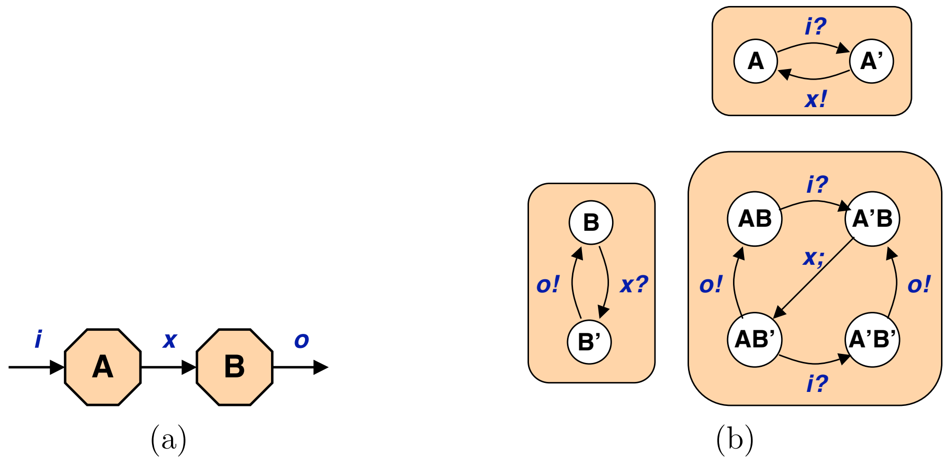


Figure 5.3: Composition of two processes connected by a single stream, (a) process view, (b) interface automata view

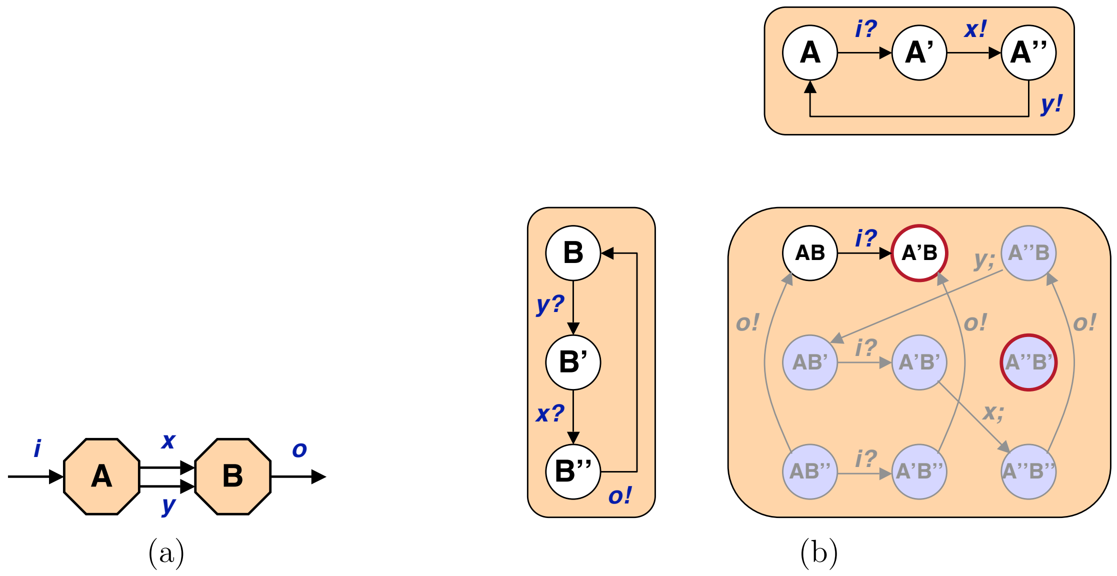


Figure 5.4: Composition of two processes connected by a pair of streams, exhibiting deadlock, (a) process view, (b) interface automata view

reachable deadlock state.

5.2.5.3 Deadlock

A composition may have terminal states that are intentional, *i.e.* the product of terminal component states, or unintentional and due to deadlock. We distinguish the two cases as follows. Let $\mathit{terminal}(P) = \{v \in V_P \mid \mathcal{A}_P(v) = \emptyset\}$ denote the set of terminal states in P. We define[4] the set of deadlock states in a composition $P \otimes Q$ to be:

$$\mathit{deadlock}(P, Q) \;=\; \mathit{terminal}(P \otimes Q) \;\setminus\; (\mathit{terminal}(P) \times \mathit{terminal}(Q)) \tag{5.3}$$

A composition $P \otimes Q$ is *live* if it contains no reachable deadlock states.

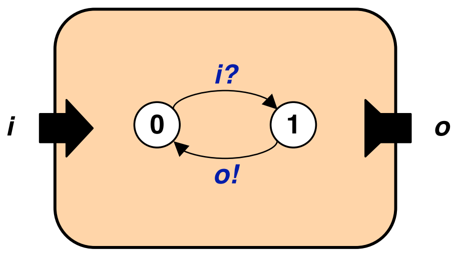
Our notion of deadlock is a special case of De Alfaro and Henzinger's notion of *illegal* states. An illegal state in a composition $P \otimes Q$ is one where P has an output transition that is not accepted by Q as input, or vice versa. In a process network interpretation, such states are not illegal so long as P or Q can continue to make other transitions. That is, one automaton can stall while the other proceeds. For a liveness analysis, we are concerned only with illegal states where neither P nor Q can

[4] The definition of $\mathit{deadlock}(P, Q)$ above is not associative. It does not properly distinguish the deadlock states from one composition as contributing to the deadlock of a larger composition. For a multi-way composition of automata P_i, we desire:

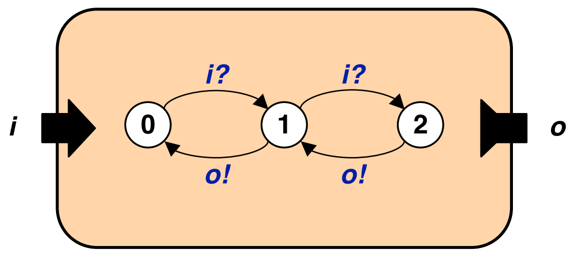
$$\mathit{deadlock}(P_i) \;=\; \mathit{terminal}(\bigotimes_i P_i) \;\setminus\; \prod_i \mathit{terminal}(P_i)$$

An associative operator $\mathit{deadlock}(P, Q)$ suitable for repeated pairwise composition can be defined with the help of a recursively defined "proper terminal" operator *pterminal*:

$$\begin{aligned} \mathit{pterminal}(A) &= \begin{cases} \mathit{terminal}(A) & \text{if } A \text{ is uncomposed} \\ \mathit{pterminal}(P) \times \mathit{pterminal}(Q) & \text{if } A = P \otimes Q \end{cases} \\ \mathit{deadlock}(P, Q) &= \mathit{terminal}(P \otimes Q) \;\setminus\; \mathit{pterminal}(P \otimes Q) \end{aligned}$$

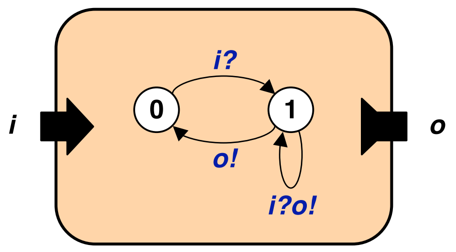


(a)

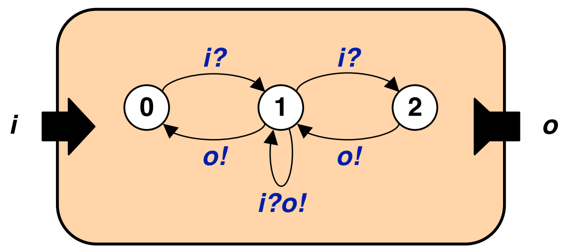


(b)

Figure 5.5: Simplified interface automata for (a) enabled register queue of depth 1, (b) shift register queue of depth 2



(a)



(b)

Figure 5.6: Multi-action interface automata for (a) enabled register queue of depth 1, (b) shift register queue of depth 2

proceed, *i.e.* deadlock states.

5.2.5.4 Stream Buffers

To resolve the deadlock of Figure 5.4, it suffices to buffer stream x with a single element queue. That buffer space would allow producer A to produce x and continue even before consumer B consumes x.

Automata may be constructed to represent stream queues. A queue of depth d is conceptually a chain of $d+1$ states that reflect occupancy, ranging from empty to full. Figure 5.5 shows automata for queues of depth 1 and 2, corresponding to the depth-1 enabled register queue and depth-2 shift register queue from Chapter 3. The actual queue implementations include states that can simultaneously consume and produce tokens. The corresponding automata would need transitions with multiple actions, shown in Figure 5.6. However, conventional interface automata permit only one action per transition, so we rely on a simplified form.

Figure 5.7 shows the pair of processes from Figure 5.4 composed with a depth-1 queue to buffer stream x. We compute the composition of A, B, and queue Q in two steps, first composing $A \otimes Q$, then $(A \otimes Q) \otimes B$. Interface automata composition is associative, so the alternate order of composition $A \otimes (Q \otimes B)$ would yields the same result. The resulting composition has no reachable deadlock states, so it is live.

5.2.5.5 Deficiency of Single Action Transitions

The fact that interface automata permit only one action per transition is obstructive for modeling process networks. A process that evaluates several actions at once, such as reading two input streams simultaneously, must be serialized into a chain or DAG of automata states connected by single-action transitions. This transformation creates large automata and even larger compositions, leading to a liveness analysis with high run-times and memory requirements. We implemented a systematic conversion of TDF SFSMs to interface automata[5] and found that an automaton typically has 10 to 20 times more states than its SFSM. A composition of N processes would have 10^N to 20^N more states than a product SFSM, which becomes intractable very quickly. Our implementation of two-process composition yielded unbuffered compositions with over 8,000 states, with over 100,000 states using a depth-1 queue per stream, and with over 300,000 states using a depth-2 queue per stream. And that was for a mere pair of processes. Clearly, we need smaller automata to retain a tractable liveness analysis.

[5] The interface automaton for a TDF SFSM can be derived as follows. In general, each SFSM state must be split into a chain or a DAG of automata states connected by single-action transitions.

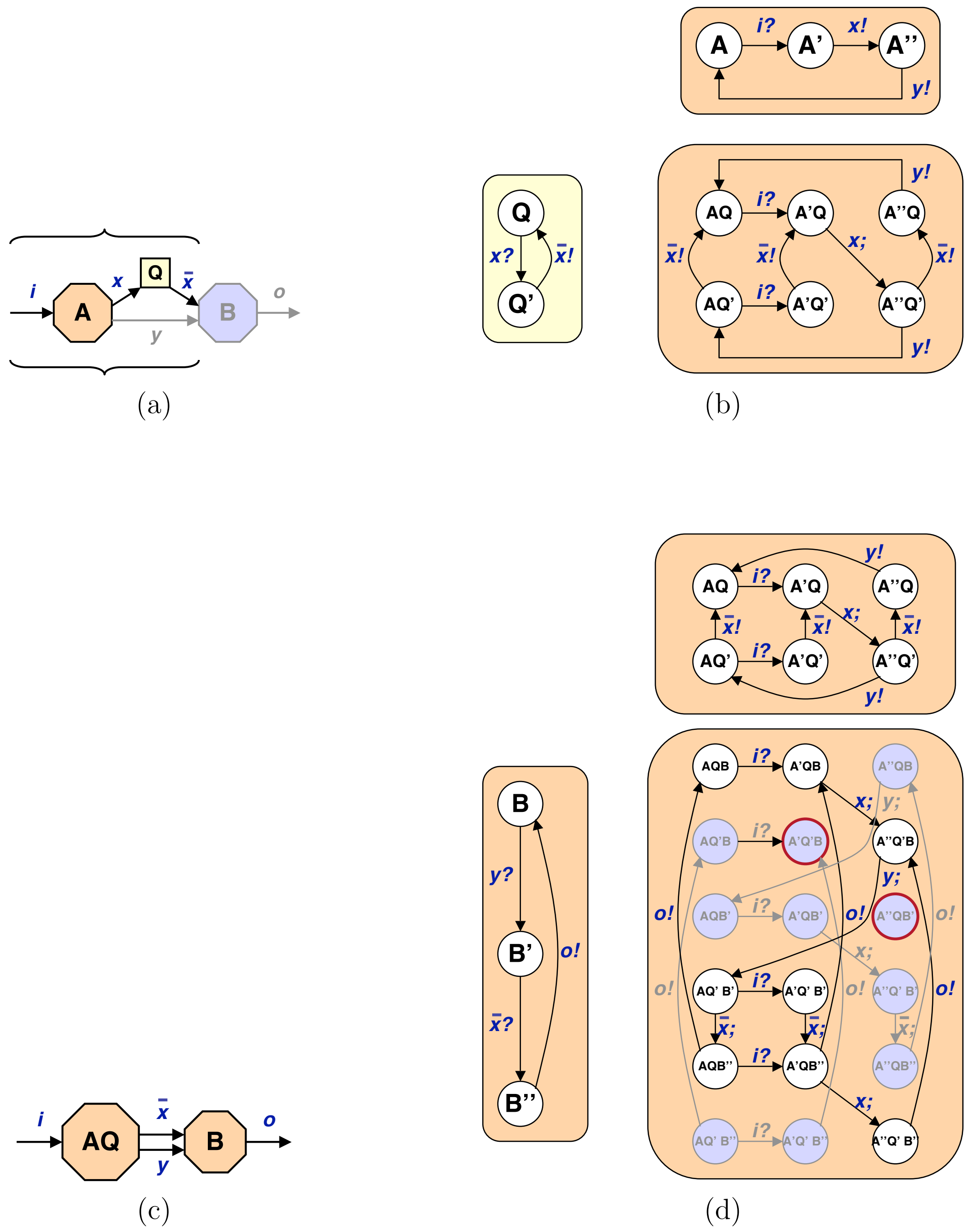


Figure 5.7: Composition of two processes connected by a pair of streams and a stream buffer. Showing initial composition $A \otimes Q$, (a) process view, (b) automata view, and second composition $(A \otimes Q) \otimes B$, (c) process view, (d) automata view.

Serialization into single-state actions creates another problem, namely that an interface automaton does not reflect the true, cycle-by-cycle schedule of its original process. Likewise, an automata composition does not reflect the cycle-by-cycle behavior of the original process network. Each schedule yields a particular order of token production and consumption and, consequently, a particular buffer usage. It is not clear that an assignment of buffer sizes that yields a live automata composition will also yield a live process network, since the two have different schedules[6]. Also, it is not clear how to synthesize an efficient SFSM from an automaton or a composition thereof. For example, if a liveness analysis indicates that a group of processes may be composed without queues, then a designer may wish to merge the processes into a composite process with unified control (the simpler approach of directly connecting the processes is disallowed by the synthesis methodology of Chapter 3). A direct translation of the composite automaton into TDF state flow would have too many states and be too slow. Merging the web of states to improve performance is difficult and possibly of limited benefit.

We circumvent the difficulties above by extending interface automata to handle multiple actions per transition.

Entry into an SFSM state induces one automaton state. The inputs in the state's firing signature are serialized and induce a chain of automata states connected by single input transitions. Alternatively, to represent possible reordering of consumption from multiple streams, the chain may be replaced by a decision tree of single input transitions. If the state has multiple firing signatures, they must correspond to sequential firing rules (see Chapter 2) and be converted into a decision tree of single input transitions. Thereafter, each SFSM state action must be broken down into basic blocks, where if-then-else branchings induce automata transitions on non-deterministic, internal actions. Within each basic block, the statements of the SFSM state action are serialized into a chain of automata states representing single-output transitions. This procedure may generate many automata states for each SFSM state, at least $N + M$ for an SFSM state with N inputs and M outputs.

[6]It may be argued that a process network needs buffers no larger than those of the corresponding automata composition, since its possible schedules differ from those of the automata composition only in making some actions simultaneous. There is no issue of correctness, since the simultaneity is still subject to the precedence constraints of the original dataflow. Nevertheless, simultaneity may allow output actions to happen closer to the input actions that consume them, thus requiring less buffering. We do not have a proof.

5.2.6 Multi-Action Interface Automata

We extend de Alfaro and Henzinger's interface automata [de Alfaro and Henzinger, 2001] to allow transitions labeled by multiple actions. This extension enables a more direct translation of processes into automata, avoiding unnecessary serialization of actions, and retaining the operational, cycle-by-cycle behavior. The composition semantics of multi-action interface automata are chosen to reflect and retain the operational semantics of concurrent processes in hardware.

5.2.6.1 Automaton Definition

A *multi-action interface automaton* P is a tuple:

$$P = (V_P, V_P^i, \mathcal{A}_P^I, \mathcal{A}_P^O, \mathcal{A}_P^H, \mathcal{T}_P) \tag{5.4}$$

which is defined identically to an interface automaton except:

- $\mathcal{T}_P \subseteq V \times \wp(\mathcal{A}_P) \times V$ is a set of *transitions*.

A transition $(v, \mathfrak{a}, v')$ is associated with a *set* of actions $\mathfrak{a} \subseteq \mathcal{A}_P$ rather than with a single action. When modeling process networks, a multi-action transition can represent an atomic firing, including reading and/or writing several streams. For example, an SDF actor needs only one automaton state with one self-looping transition. Our definition is restricted to modeling firings that consume/produce at most one token on each stream. This restriction is in line with the synthesis methodology of Chapter 3, where the implementation of streams is also restricted to one token per cycle. Firings that consume/produce more than one token on a stream could be modeled using a *multiset* of actions on each transition, but that is beyond the scope of our work.

Figure 5.2 shows a multi-action automaton for the BDF *select* actor. The transition "$i?t!$" from state T models a firing involving one input and one output (likewise for transition "$i?f!$" from state F). The two transitions from state S model a non-deterministic choice between consuming a true or false value from input stream s.

Figure 5.6 shows multi-action automata for queues. The self-loop transition in state "1" models a firing in full throughput mode, where the queue simultaneously emits a buffered value and consumes a new one.

5.2.6.2 Automata Composition

The composition of two composable, multi-action interface automata P, Q is a product automaton:

$$P \otimes Q = (V_{P\otimes Q}, V^i_{P\otimes Q}, \mathcal{A}^I_{P\otimes Q}, \mathcal{A}^O_{P\otimes Q}, \mathcal{A}^H_{P\otimes Q}, \mathcal{T}_{P\otimes Q}) \tag{5.5}$$

which is defined identically to the composition of interface automata except:

$$\begin{aligned}
\mathcal{T}_{P\otimes Q} &= \{((u,v), \mathfrak{a}, (u',v')) \mid (u,\mathfrak{a},u') \in \mathcal{T}_P \ \wedge \ \mathfrak{a} \cup \mathit{shared}(P,Q) = \emptyset \ \wedge \ v \in V_Q\} \\
&\cup \{((u,v), \mathfrak{a}, (u',v')) \mid (v,\mathfrak{a},v') \in \mathcal{T}_Q \ \wedge \ \mathfrak{a} \cup \mathit{shared}(P,Q) = \emptyset \ \wedge \ v \in V_P\} \\
&\cup \{((u,v), \mathfrak{a}, (u',v')) \mid \exists \mathfrak{a}_P, \mathfrak{a}_Q \text{ s.t. } \mathfrak{a} = \mathfrak{a}_P \cup \mathfrak{a}_Q \ \wedge \ (u,\mathfrak{a},u') \in \mathcal{T}_P \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge \ (v,\mathfrak{a},v') \in \mathcal{T}_Q\}
\end{aligned}$$

The first line of this definition permits P to transition alone, provided its transition contains no shared actions. The second line permits the same for Q. The third line permits P and Q to transition together on any combination of shared and unshared actions. This last ability reflects the operational concurrency of two self-scheduled processes in hardware, which are allowed to step simultaneously without communicating with each other. Simultaneous transition on unshared actions is not possible in conventional interface automata, where just one action must be chosen for each composite transition. There, advancing both P and Q on unshared actions requires a sequence of two transitions, one for P and one for Q. In short, a composition of

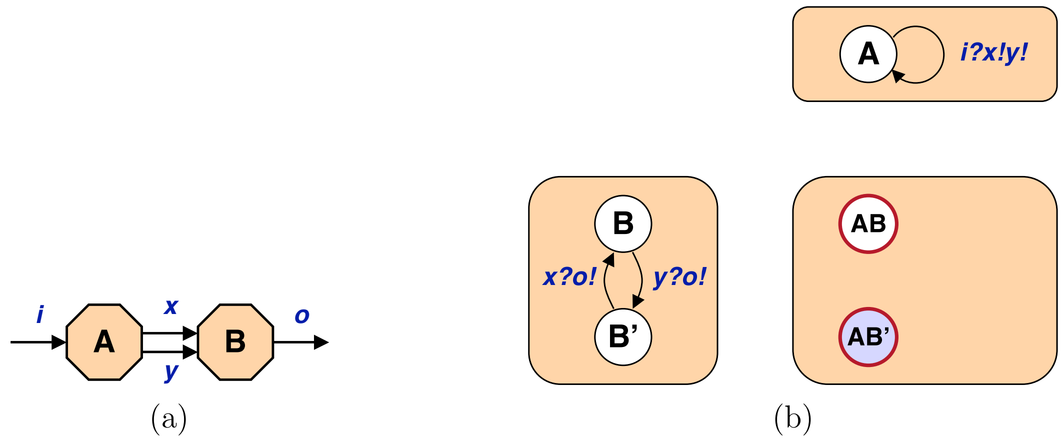


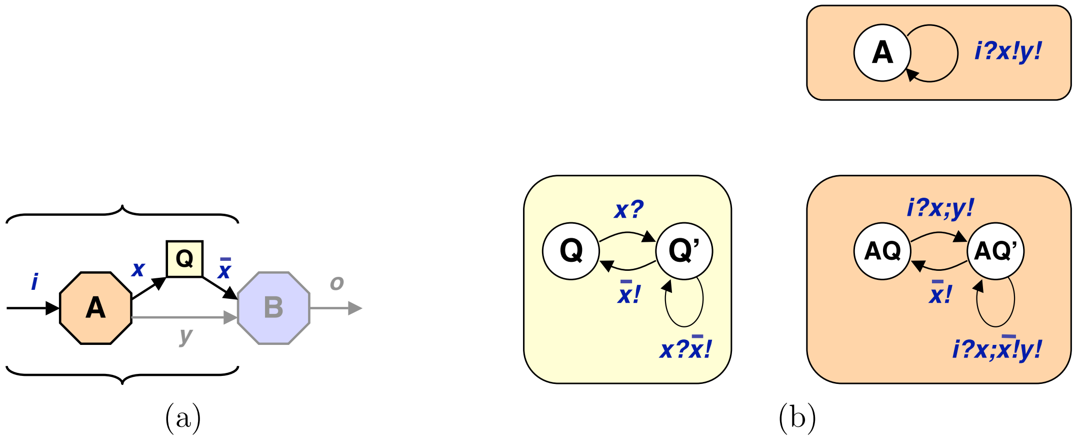
Figure 5.8: Composition of two processes connected by a pair of streams, exhibiting deadlock, (a) process view, (b) multi-action interface automata view

multi-action interface automata reflects the cycle-by-cycle behavior of a composition in hardware, whereas composition of conventional interface automata does not.

Deadlock and liveness for multi-action interface automata are defined identically as for interface automata. The use of queue automata for stream buffering remains the same. The basic idea of liveness analysis embedded in an abstract Park's algorithm remains the same.

Figure 5.8 shows an example composition that exhibits deadlock using multi-action interface automata. Two processes are connected using a pair of streams x, y. The producer A repeatedly consumes from primary input i and produces to x and y. The consumer B alternately consumes from either x or y, and always produces to output o. Composite execution should force A to run at half throughput, limited by the alternating nature of B. Using multi-action interface automata composition, we find that A is not enabled to simultaneously produce x and y, since B does not read them simultaneously. The result is deadlock.

Figure 5.9 resolves the deadlock of Figure 5.8 by adding a depth-1 queue to buffer stream x. We compute the composition of A, B, and queue Q in two steps, first



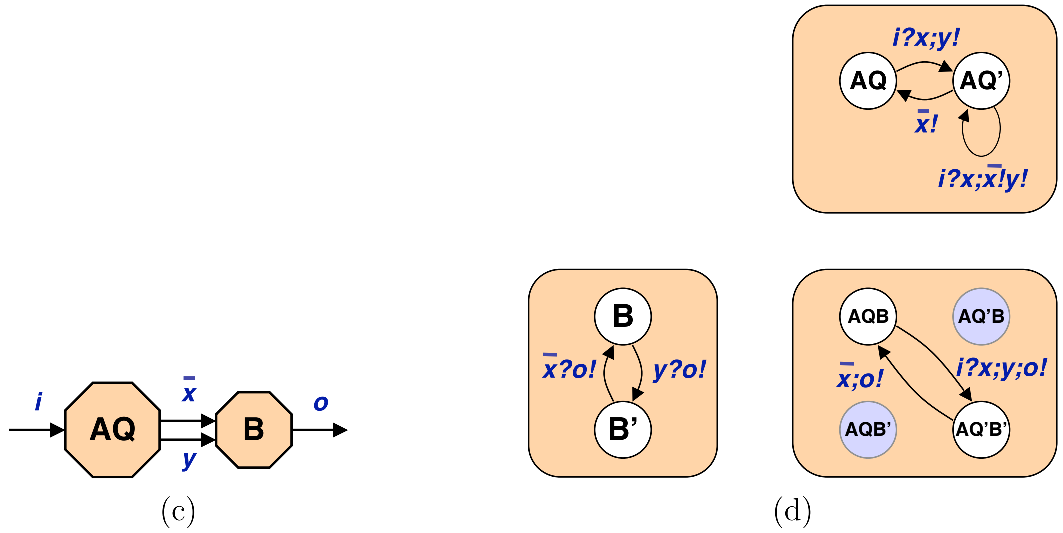


Figure 5.9: Composition of two processes connected by a pair of streams and a stream buffer, using multi-action interface automata. Showing initial composition $A \otimes Q$, (a) process view, (b) automata view, and second composition $(A \otimes Q) \otimes B$, (c) process view, (d) automata view.

composing $A \otimes Q$, then $(A \otimes Q) \otimes B$. The composition behaves as expected, alternating between processing buffered x (labeled $\overline{x}$) and y, with consumption of input i reduced to half throughput. This example also demonstrates that multi-action compositions can be very small, two states in this case, and substantially smaller than the corresponding conventional interface automata compositions (*e.g.* compare the multi-action Figure 5.9 to the single-action Figure 5.7, depicting very similar process networks).

5.2.6.3 Statically Schedulable Dataflow

SDF and CSDF graphs have particularly well behaved compositions using multi-action interface automata. An SDF actor [Bhattacharyya *et al.*, 1996] is a process with a single state (or equivalently, no state) and a single firing rule that consumes and produces a static number of tokens. The automaton for that process has a single state with a single, self-loop transition. Recall that our representation is limited to consuming/producing one token per stream per firing, corresponding to *single-rate* (unit rate) SDF actors. The composition of such automata, representing a homogeneous dataflow graph with no ideal delays, would have just one product state and one transition. An ideal delay corresponds to a unit capacity stream buffer, and at least one is required on any feedback loop to avoid deadlock. An SDF graph with N ideal delays (implemented as depth-1 queues) would have an automata composition with at most 2^N states. However, in steady state operation, the buffers always have an occupancy of one. A synchronized firing of all actors in the graph is balanced, producing as many tokens as it consumes, borrowing and returning a token to every buffer. Thus, the automata composition for an SDF graph with N ideal delays has only one steady state[7].

[7]An ideal delay in an SDF graph always has an initial token. However, TDF presently does not support initial contents for stream buffers. Instead, a stream buffer corresponding to an ideal

A CSDF actor [Bilsen *et al.*, 1996] is a process with a loop of K states, each one consuming/producing some subset of the actor's streams. Again, our representation is restricted to consuming/producing at most one token per stream per firing. A composition of such CSDF actors, each having K_i states, will have a steady-state loop of at most $\mathrm{lcm}(\{K_i\})$ states (the least common multiple of all K_i). The composition may require stream buffering to realign tokens in time, as in the example of Figures 5.8 and 5.9. It may also include ideal delays, corresponding to more unit capacity buffers. Nevertheless, the steady state behavior of the automata composition remains a loop as above.

Our automata composition also recognize some statically schedulable behaviors with state branchings. Consider an SDF actor that occasionally needs an extra cycle to think. That behavior is represented by a data-dependent branch to a second state that neither inputs nor outputs tokens. The I/O behavior of the actor has a static ratio of inputs to outputs, so it is statically schedulable. An automata composition involving this actor will reveal that no additional buffering is necessary. Whenever the actor transitions to its second state, other actors connected to it simply stall. Next, consider an actor that repeatedly evaluates one of two state sequences. Each sequence consumes and produces the same number of tokens, but in a different order and at different times. Again, the I/O behavior of the actor has a static ratio of inputs to outputs, so it is statically schedulable. Our automata composition will discover that small, statically sized buffers suffice to realign the tokens in time. Dynamic dataflow is characterized by non-deterministic branches that yield a dynamic ratio of inputs to outputs. For those cases, our automata composition will deadlock for any finite buffer sizes. Unfortunately, our automata ignore data values, so they fail

delay must be initialized by an initial state of the stream producer, which emits the initial token. Consequently, an SDF actor in TDF may need two states: an initialization state and a steady state. The automata composition of such actors also has only two states: a product initialization state, and a product steady state.

Application	SFSMs	Internal Streams	SFSM States Max	SFSM States Product	Product States Using Queue Depths 0	1	2
IIR	8	7	1	1	1	128	2,187
JPEG Decode	9	41	21	10,080	10,080	2.2×10^{16}	3.7×10^{23}
JPEG Encode	11	42	21	25,200	25,200	1.1×10^{17}	2.8×10^{24}
MPEG Encode IP	80	231	21	1.1×10^{31}	1.1×10^{31}	3.7×10^{100}	1.7×10^{141}
MPEG Encode IPB	114	313	21	6.3×10^{50}	6.3×10^{50}	1.1×10^{145}	1.4×10^{200}
Wavelet Encode	30	50	7	3.6×10^{16}	3.6×10^{16}	4.0×10^{31}	2.6×10^{40}
Wavelet Decode	27	49	6	1.1×10^{13}	1.1×10^{13}	6.4×10^{27}	2.7×10^{36}
Total	279	733					

Table 5.1: Maximum sizes of whole application automata compositions

to recognize some statically schedulable compositions that contain data-dependent branching. They cannot recognize that a counter-based loop has a fixed number of iterations unless it is fully unrolled into a loop of states. Also, they cannot schedule a BDF style if-then-else graph, which is well behaved because it controls multiple, dynamic-rate actors with matching, boolean-valued tokens [Buck, 1993].

5.2.7 Efficient System Composition

System level liveness analysis involves the composition of every process and every queue in the system. The resulting compositions may be very large, possibly too large to process on a conventional, desktop computer. Table 5.1 estimates the maximum size of product automata for the seven TDF multimedia applications from Chapter 4. Using multi-action interface automata, each SFSM automaton has as many states as its SFSM. The product automaton has a state count equalling, at most, the product of SFSM state counts. That number may be trivially small, *e.g.* 1 for IIR, which is a feed-forward SDF graph. It may also be intractably large, *e.g.* 10^{50} for MPEG Encode IPB. The number of possible states gets even larger with stream buffering. Every buffer slot infers another queue state, which in the product automaton, infers a replication of the entire unbuffered composition. Using a depth-1 queue for every stream yields up to 128 states for IIR, and up to 10^{145} for MPEG. Using a depth-2 queue for every stream yields up to 2,187 states for IIR, and up to 10^{200} for MPEG. The actual product automaton may be smaller after removal of unreachable states, particularly if the process network contains large SDF or CSDF regions. Nevertheless, the asynchrony provided by buffering always creates a state explosion, so buffer-rich compositions tend to have 50%-100% of the maximum number of possible states.

A tractable, system level, liveness analysis can keep automata compositions small in several ways: (1) keep stream buffers as small as possible, (2) partition the process network into separately analyzed regions (*divide and conquer*), and (3) apply partial analysis early to avoid fully building deadlocked compositions (*branch and bound*).

5.2.7.1 Keeping Buffers Small

Buffer sizes are chosen by the abstract Parks' algorithm. Buffers initially have zero capacity and are expanded whenever a liveness analysis indicates deadlock. Parks' suggested always increasing the smallest full buffer by one. Identifying which buffers are full in a particular deadlocked state may be done by marking the full state of every buffer automaton and retaining that marking during automata composition. Bufferlock usually involves a cascade of buffer overflows, originating at one buffer and propagating upstream via back-pressure. Expanding any of the upstream buffers will allow a process network to make some forward progress, but it will not resolve the bufferlock. The bufferlock must be resolved at its point(s) of origin. Thus, a smarter implementation of Park's algorithm might try to identify an originating full buffer and expand it first, even if it is not the smallest full buffer. One way to find an originating full buffer is to start at an arbitrary full buffer and follow downstream links through other full buffers as far as possible. The connection between full buffers may be coincidental. That is, the cascade of overflows moving backwards from one origin may reach a second, independent origin that was already full. Nevertheless, resolving the origin furthest downstream will eventually reveal the other origin.

5.2.7.2 Divide-and-Conquer

A useful technique for keeping automata compositions small is to partition the process network into separately analyzed regions. We can apply the abstract Park's algorithm separately for each region to bound its stream buffers. The streams connecting those regions remain unanalyzed, and consequently unbounded. They can be handled in one of several ways:

1. *Bound the stream using a different analysis.*
   One approach is to use topological properties. For example, bridge streams require no buffering (more on this below). This suggests that process networks should always be partitioned at bridge streams first.

2. *Bound the stream using a programmer annotation.* That is, let the user choose a buffer bound.

3. *Provide run-time buffer expansion for the stream.* This approach requires additional hardware for bufferlock detection and buffer reallocation. It is impractical on an FPGA or ASIC, but it is possible on a SCORE architecture.

A divide-and-conquer approach to liveness analysis requires assurance that composing two live partitions will not introduce new deadlock. In general, this is not possible. Composing a partition's automaton with anything that has shared actions (*i.e.* is connected via streams) can only restrict the automaton's transitions. It may leave some states with no enabled transitions, indicating to a deadlock. There is one special case that is guaranteed to produce no deadlock—a single stream connection. If the downstream process is not ready to consume, then the upstream process simply waits for it to become ready. There is no other synchronization between them to cause deadlock. No buffering is required, though the same reasoning holds if the stream is buffered and the buffer is full—the upstream simply waits until the downstream drains one element from the buffer. The processes in question may be compositions. A more precise proof follows for the interface automata view.

Theorem 5.2.3. *Let P and Q be live, multi-action interface automata with one shared action. Then $P \otimes Q$ is also live.*

Proof. Deadlock requires both P and Q to be stalled, so we prove that at least one of them is always enabled to transition. Suppose P and Q share an action a, P is in state p, and Q is in state q. If P has any enabled transitions not involving a ($\exists \mathfrak{a} \in \mathcal{A}_P(p)$ s.t. $a \notin \mathfrak{a}$), then P can transition independent of Q, and there is no deadlock. Likewise for Q. The only remaining case is if P and Q each have only one enabled transition, and that transition involves a ($\mathcal{A}_P(p) = \mathfrak{a}_p$, $a \in \mathfrak{a}_p$, $\mathcal{A}_Q(q) = \mathfrak{a}_q$, $a \in \mathfrak{a}_q$). In this case, P and Q transition simultaneously, so there is no deadlock. $\square$

Theorem 5.2.3 implies that every bridge stream in a process network requires no buffering. A bridge is a graph edge whose removal would disconnect the graph (forming P and Q of the theorem). Thus, a reasonable first step for efficient, system level, liveness analysis is to partition the process network at bridge streams.

A similar result holds for any tuple of streams that may be merged to form a bridge stream. Such streams may have been logically separated by a programmer for clarity and convenience, but they should be remerged by a compiler before other analyses. A tuple of streams may be merged if all of the following conditions hold:

- All streams are produced by the same process,
- All streams are consumed by the same process,
- Tokens are produced to all streams in synchrony,
- Tokens are consumed from all streams in synchrony.

For TDFPN, we can define synchronized production/consumption as follows. Streams i, j are consumed in synchrony if: $\forall \boldsymbol{r} \in R$, $|\pi_i(\boldsymbol{r})| = |\pi_j(\boldsymbol{r})|$. Streams i, j are produced in synchrony if: $\forall \boldsymbol{r} \in R$, $|\pi_i(f_O(\boldsymbol{r}))| = |\pi_j(f_O(\boldsymbol{r}))|$.

5.2.7.3 Branch and Bound

A branch-and-bound approach to liveness analysis would attempt to discover deadlock early, before a deadlocked composition has been built in its entirety. It can then immediately move on to the next step of the abstract Parks' algorithm. We consider several ways to do that.

The liveness of an automata composition is determined by verifying that no path of transitions exists from the initial state to any deadlock state. As soon as one path is found, liveness analysis may terminate and return false. It suffices to build the automata composition incrementally, along the path search. In fact, it suffices to record only which product states were visited, not their transitions. A path search may be performed forward or backwards. A forward search would start at the initial state and build the composition along forward transitions, looking for deadlock states. This approach has the advantage of never visiting unreachable states. A backwards search would begin by identifying all deadlock states, including possibly unreachable ones (by computing $\mathcal{A}(v)$ for every product state v). For each deadlock state, the search would build the composition along backwards transitions, looking for the initial state.

A composition of multiple automata is normally computed by repeated, pairwise composition. However, this approach obstructs early detection of deadlock. Only the final pairwise composition can be built incrementally around a path search. The intermediate compositions must be built in their entirety in order to compute the final composition. It would be useful to discover something about deadlock or liveness during one of the intermediate compositions, and avoid having to compose all pairs. However, as we noted above, liveness of an intermediate composition does not guarantee liveness of the next composition (unless the pair is connected by a single stream—Theorem 5.2.3). Similarly, a reachable deadlock state in an intermediate composition may not be reachable in the next composition. Hence, it seems that liveness analysis must be deferred until the final pairwise composition. There are two ways to apply partial liveness analysis earlier: (1) avoid pairwise composition, computing the multi-way composition directly and applying the path search immediately, or (2) cull deadlock out of the intermediate pairwise compositions.

Direct Multi-Way Composition. A composition of multiple automata can, in principle, be computed directly and incrementally around a path search. Pairwise composition is associative, so the multi-way composition is unique. The definition of multi-way product states is obvious. Transitions may be tested at each product state as follows, generalizing the definition of $\mathcal{T}_{P\otimes Q}$. A transition of any individual component automaton is accepted if it contains no shared actions. A simultaneous transition of any pair of component automata is accepted if the two component transitions share no actions with other components. Similarly for higher tuples of component automata. These tests seem to require computing shared action sets for all possible subsets of component automata; those shared action sets may be memoized for use in other product states.

Culling Reachable Deadlock. De Alfaro and Henzinger [de Alfaro and Henzinger, 2001] suggest a way to cull intermediate pairwise compositions. Their original approach is based on *illegal* states, but we adapt it to deal with our deadlock states. A composition $P \otimes Q$ is considered *compatible* if there exists an environment (*i.e.* a next composition) that prevents $P \otimes Q$ from reaching its deadlock states. An environment can only constrain the inputs and outputs of $P \otimes Q$. If $P \otimes Q$ can reach a deadlock state via internal transitions only, then clearly no environment could prevent its deadlock. To represent non-deadlocking behavior, the composition is culled by removing any states that can reach deadlock via internal transitions. The remaining states, termed *compatible* states, represent cases where an environment *might* prevent $P \otimes Q$ from reaching deadlock. Those states can then undergo further, pairwise composition. If deadlock is discovered in a subsequent composition, it would indicate that either (1) the compatible states were insufficient in the actual environment, or (2) an independent deadlock exists in the actual environment. De Alfaro and Henzinger prescribe the most optimistic approach, which is to evaluate the entire chain of intermediate, pairwise compositions, and to give each one the chance to steer prior compositions away from their deadlock. Thus, every intermediate, pairwise composition is computed and then culled. If any culled composition is empty, then deadlock is unpreventable, and the analysis terminates. Otherwise, a non-empty, final, culled composition has deadlock free schedule and can be safely implemented[8].

The culling approach provides two forms of savings for a liveness analysis. First, the culling keeps intermediate compositions small. Second, the analysis terminates as soon as any intermediate, pairwise composition is culled to empty. At that point, the abstract Parks' algorithm can attempt buffer expansion. Buffers eligible for expansion include any full buffers from culled deadlock states.

The culling approach has the advantage of recognizing some cases of *local deadlock*. Parks' algorithm for dynamic scheduling responds only to total system deadlock, where all processes are stalled. Likewise, our abstract interpretation of deadlock as a product state with no enabled transitions also corresponds to total system deadlock. These approaches miss any case where a subgraph of the process network deadlocks, while another subgraph continues to run. A culled composition represents composed behavior that prevents local deadlock in prior compositions. If a culled composition is empty, it indicates that *any* of the local deadlocks in prior compositions would be possible in the unculled composition.

[8]A culled composition can be implemented verbatim as a single, large SFSM. Implementing the composition as separate SFSMs (one per process) does not strictly enforce non-entry into culled states, so it may yet permit reaching deadlock states.

5.3 Stream Aware Placement

Process networks are naturally robust to long interconnect delay. Their streams may be pipelined across long distances, yet their behavior remains deterministic. In conventional design flows, communication pipelining is fixed in RTL and becomes a constraint for placement. In Chapter 3, we proposed an approach for choosing stream pipeline depths after placement, to match actual interconnect delay. That approach provides some improvement in performance and area compared to pipeline depths fixed in RTL. However, a conventional placer does not know that stream-related wires may be pipelined, so it misses certain opportunities for optimization. A conventional placer is usually timing driven and will strive to keep all register-to-register paths within timing constraints. It does not know that a feed-forward stream can violate the target clock period and be pipelined without degrading system throughput. That freedom might allow the placer to push the stream endpoints apart and bring other, more timing-critical circuits together to improve the clock rate. Nor does the placer know that pipelining a stream feedback loop may degrade system throughput, or it may not, depending on the nature of the feedback. Non-critical stream feedback loops can actually be pipelined to tolerate placement across large distances.

In this section, we discuss how to extend placement with knowledge of pipelinable streams. Our basic approach is to reuse the existing notion of *net criticality*, which denotes to a placer the need to keep a connection short. Given a process network, we build a throughput model and use it to derive net criticalities. A similar approach works for circuit partitioning, which typically adds interconnect delay to inter-partition connections (*e.g.* partitioning across multiple chips, across MegaLABs in an Altera APEX FPGA, or across SCORE pages). The conversion of throughput information into net criticalities has been proposed by Singh and Brown [Singh and Brown, 2002] for placing sequential circuits on FPGAs. We adapt their approach to deal with process networks. To do so, we derive throughput models for process networks.

5.3.1 Adding Throughput Awareness to a Placer

Timing driven placement is based on a notion of *slack* time. To meet a target clock period ϕ, the total combinational delay $\sum_i d_i$ along any register-to-register circuit path must be at most ϕ. The *slack* of a path is the delay that can be added in interconnect without exceeding ϕ: $\mathit{slack} = \phi - \sum_i d_i$. A placer can use slack to prioritize which components to keep proximal. Paths with high slack can cross large distances, whereas paths with small slack cannot. In VPR (Versatile Place and Route) [Betz and Rose, 1997] [Betz *et al.*, 1999], the criticality of a net ranges from 0 to 1 and is computed as: $\mathit{crit} = 1 - \mathit{slack}/\phi$. A *net* refers to a route between circuit components.

Retiming is a circuit transformation that moves registers, and which can improve the attainable clock period by equalizing delays among register-to-register paths [Leiserson *et al.*, 1983]. Combining retiming with placement provides some means to address the fact that interconnect delays are not known until placement. Singh and Brown [Singh and Brown, 2002] propose making a placer aware of the additional timing slack enabled by post-placement retiming. They call that the *cycle slack*. With retiming, the attainable clock period is limited by delay on circuit cycles (feedback loops). Retiming is not able to add registers to cycles, but it is able to redistribute the registers to equalize combinational delay. For a given cycle c containing N registers and $\sum_i d_i$ combinational delay, the attainable clock period with retiming is $\phi_c = (1/N) \sum_i d_i$, called the *delay to register ratio* (DRR). The attainable clock period ϕ of the entire circuit is the maximum DRR (MDR) over all cycles. For a given clock period ϕ, the slack available on a cycle with retiming can be thought of as follows. Each register on the cycle adds ϕ to the cycle's time budget. Each combinational element subtracts d_i from the cycle's time budget. The time left over is $N\phi - \sum_i d_i$. The average time left over for each of the N register-to-register paths is $\phi - (1/N)\sum_i d_i$, the so-called *cycle slack*. If a given net is on several cycles, then its available slack is the minimum cycle slack of all such cycles. The cycle slack of a net can then be turned into a net criticality[9], *e.g.* for VPR: $crit = 1 - CycleSlack/\phi$.

A similar analysis of cycle throughputs is used in Latency Insensitive Design (LID) [Carloni and Sangiovanni-Vincentelli, 2000] for composing synchronous modules with pipelined interconnect. A synchronous module contains internal registers and is assumed to produce valid data on every clock cycle. Inter-module wires may be automatically pipelined to cross long distances, but those pipeline registers introduce bubbles into the otherwise valid computation. Bubbles propagate around a circuit like data, but they represent nil data and a consequent loss of throughput. Carloni and Sangiovanni-Vincentelli show that the number of bubbles in a feedback cycle remains constant during circuit operation. This property is similar to the one in retiming whereby the number of registers in a feedback cycle remains constant, as both properties are based on the splitting/joining of movable elements (bubbles or registers) across fanouts/fanins. A feedback cycle with M modules and N pipeline registers produces valid data, on average, every $(M+N)/M$ clock periods. That value

[9] Singh and Brown describe two practical adaptations for computing net criticalities for VPR. They tune criticalities as: $crit = 1 - \beta \cdot CycleSlack$, where β is chosen adaptively so that no more than 5% of all nets have a criticality over 0.9. They also approximate the computation of cycle slack. A net may belong to arbitrarily many cycles, and finding the minimum cycle slack of all such cycles is difficult. Instead, they advocate analyzing a random subset of such cycles. The cycle slack of a cycle C can be computed as the total edge weight along C in the *cycle rate graph*, a graph derived from the retiming graph by replacing each edge weight $w(e_{uv})$ with a slack contribution weight, $w'(e_{uv}) = \phi w(e_{uv}) - d(e_{uv}) - d(v)$, where $d(v)$ and $d(e_{uv})$ are the combinational delays of circuit element v and the route of e_{uv}. The available slack of net e_{uv} is the minimum cycle slack of cycles containing e_{uv}, but it may be approximated from a random subset of cycles as follows. Choose a random node s. Compute the minimum cycle slack of cycles containing s and e_{uv} as: $w'(v \to s) + w'(s \to u) + w'(e_{uv})$, where $x \to y$ denotes the shortest (least weight) path from x to y and $w'(x \to y)$ denotes its total edge weight. Repeat for several random nodes s and take the least result.

is the ratio of total data to valid data on the cycle, termed the *cycle mean* (CM). The system throughput is constrained by the slowest cycle and is equivalent to the *maximum cycle mean* (MCM)[10]. We note that LID's CM and MCM are analogous to Singh and Brown's DRR and MDR, with ϕ factored out. The LID literature does not address how to tune a placer for minimum throughput degradation, *i.e.* minimum MCM. Nevertheless, it is possible to apply Singh and Brown's approach, defining cycle slack as $(MCM - CM)$ periods, and converting it into a net criticality for an inter-module wire as: $crit = 1 - ((MCM - CM)/MCM)$.

5.3.2 Process Network Throughput Models

The LID model represents a particularly simple process network, where every process consumes and produces a single token per stream per firing. A similar throughput model can be applied to more general process networks, where token rates may be non-unit and non-static. We demonstrate this by (1) modeling the throughput of single-rate SDF, and (2) transforming other process network models into SDF, either by known techniques or by approximation. Where ever throughput and cycle slack can be defined, net criticalities can be derived for a placer.

5.3.2.1 Single Rate SDF

In single-rate SDF (SRSDF), each actor has a single firing rule that consumes and produces one token per stream per firing. All actors are stateless except for the *ideal delay* element, for whom each firing produces the value consumed in the previous firing. To evaluate the graph, tokens are conceptually propagated from ideal delays,

[10] The maximum cycle mean (MCM) problem is well known in the literature, dating back at least to Karp's Algorithm [Karp, 1978]. In a weighted edge graph, the mean of a cycle C is its weight divided by its length: $\lambda(C) = \frac{\sum_{e \in C} w(e)}{|C|}$. The maximum cycle mean of a graph G is: $\lambda(G) = \max_{C \in G} \lambda(C)$. For LID's computation of MCM as throughput, $w(e)$ denotes pipeline delay, and would be equal to the pipeline depth of inter-module route e plus one for the module at the beginning of e.

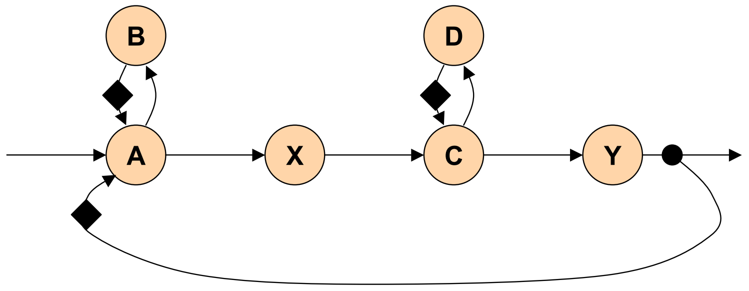


Figure 5.10: SRSDF graph with throughput 4, limited by cycle $\{AXCY\}$

through actors, and back to ideal delays. A direct hardware implementation would realize each actor as a combinational element and each ideal delay as a D flip-flop with initial value. A more efficient hardware implementation would add pipeline registers at actor outputs or on long distance streams. Most such pipeline registers are uninitialized and represent bubbles in a LID-style throughput analysis. However, they may be configured to hold initial tokens, like ideal delays. A cycle C containing M tokens and N pipeline delays produces a valid token, on average, every $T_C = (M/N)$ clock periods. The graph produces a valid token, on average, every $T = \max_C\{T_C\}$ clock periods. We dub this the *token period*, in analogy to the clock period. The cycle slack in periods is $(T - T_C)$. A net criticality can then be computed as: $crit = 1 - ((T - T_C)/T)$. Our throughput analysis is a pipelined generalization of Singh and Brown's clock rate analysis, with our T, T_C, and $(T - T_C)$ being directly analogous to their ϕ, ϕ_c, and *CycleSlack*.

Figure 5.10 shows a sample SRSDF graph with three base cycles. Suppose the implementation provides a single level of pipelining between actors, and that level can hold an ideal delay value (*e.g.* an enabled register queue from Chapter 3). Then we have the following cycle throughputs in clock periods per token: $T_{\{AB\}} = 2$, $T_{\{CD\}} = 2$, $T_{\{AXCY\}} = 4$, $T = \max_C\{T_C\} = 4$. The cycle slack of cycle $\{AB\}$ is two periods, so a placer may add up to two levels of interconnect pipelining to its nets without degrading system throughput. Likewise for cycle $\{CD\}$. However, cycle $\{AXCY\}$ is critical and has zero slack.

Ideal delays represent initial tokens in stream buffers. They can be inferred from a process network description such as TDF in two places. First, there may be a specification of initial stream contents. Second, process initial states may inject initial tokens into streams. A comprehensive recovery of initial tokens would need to distinguish initial states in the state flow graph of each process and, possibly, simulate their interaction.

5.3.2.2 Multi-Rate SDF

Multi-rate SDF (MRSDF) differs from single-rate SDF (SRSDF) in that an actor firing may consume or produce more than one token. Connecting actors with unequal rates requires that some actors fire more often than others. Non-uniform firing rates invalidate using a LID-style throughput analysis on the original graph. However, it is possible to unroll an MRSDF graph into an equivalent SRSDF graph, and apply the analysis there. We can then derive slacks and criticalities for the original MRSDF graph.

The steady state behavior of an MRSDF graph is captured by a *balanced schedule*, wherein a repeating pattern of actor firings returns the graph to its original state, leaving the same number of tokens on the same streams. The number of firings of each actor in a balanced schedule may be found by solving *balance equations* [Bhattacharyya *et al.*, 1996]. For each stream $A \rightarrow B$, assuming A produces N_A tokens per firing, and B consumes N_B tokens per firing, we pose the equation: $q_A N_A = q_B N_B$. Quantities q_A and q_B represent the number of firings of A and B in a balanced schedule, and the equation represents a balance of production and consumption in the stream $A \rightarrow B$. The set of equations for a graph may be represented and solved as an eigenvector problem: $\Gamma \boldsymbol{q} = \boldsymbol{0}$, where $\boldsymbol{q}$ is the *repetitions vector*, and Γ is the *topology* *matrix* ($\Gamma_{i,j}$ equals the number of tokens-per-firing added to stream i by actor A_j, positive for production and negative for consumption) SDF theory provides liveness conditions under which a solution $\boldsymbol{q}$ exists and is unique, within a scaling factor. Scaling is permissible, because any repetition of a valid schedule is also a valid schedule. We are normally interested in the least integer solution of $\boldsymbol{q}$.

An MRSDF graph may be unrolled into an *acyclic precedence graph* (APG) that represents the token flow in one instance of a balanced schedule [Lee and Messerschmitt, 1987a]). The precedence graph contains q_A instances of each actor A, representing firings, and connecting to instances of A's neighbors in correspondence with token flow. Tokens conceptually flow from output ports of ideal delays, through actor instances, to input ports of ideal delays. The precedence graph can be interpreted as an SRSDF graph if ideal delay port pairs are merged to form feedback cycles. Williamson [Williamson, 1998] proposes mapping the equivalent SRSDF graph directly to hardware, with concurrent realizations of all actor instances. Our synthesis methodology of Chapter 3 implements the instances of an actor A not concurrently but serially, in one hardware realization of A. We can represent this serialization by adding precedence edges between instances of A (often, these edges are superfluous, as serialization is already implied by other dataflow edges).

The equivalent SRSDF graph may used to reason about the effect of interconnect delay on throughput. Consider clustering the instances of each actor A, to represent to a placer that those instances are co-located. Original streams $A \rightarrow B$ may now appear replicated between clusters, connecting instances of A and B. They will be merged later. If a placer makes A and B distant, then all replicated streams between A and B will incur the same interconnect delay. If those streams constitute a feedback cycle between A and B, then the cycle will incur interconnect delay multiple times, as its path crosses back and forth between multiple instances in the different clusters. The multiplicity of a replicated stream will be used to scale its apparent cycle slack

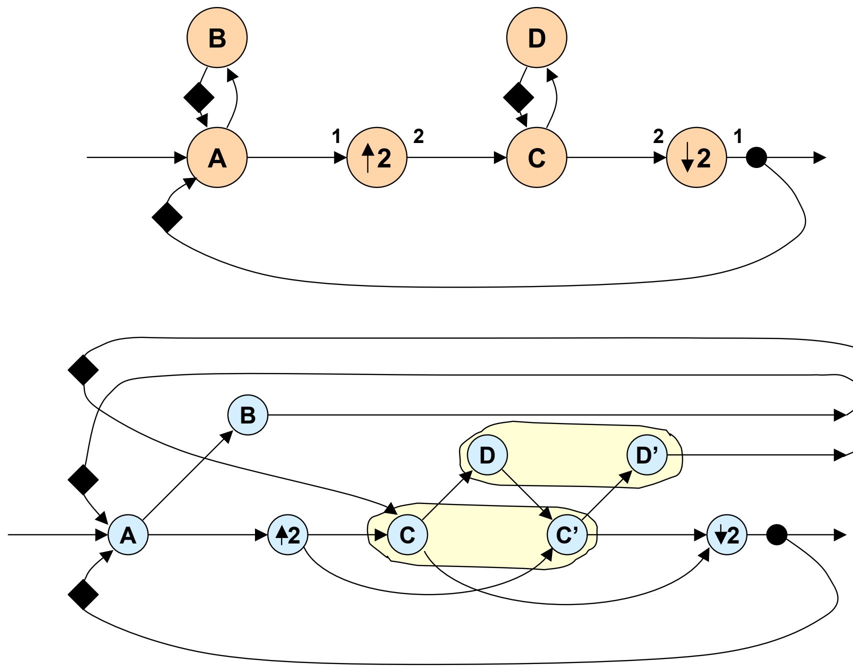


Figure 5.11: MRSDF graph (top) and equivalent SRSDF graph (bottom) with throughput 6, limited by cycle $\{A \uparrow_2 CDC' \downarrow_2\}$

in the computation of net criticality.

We now collapse the equivalent SRSDF graph to match the structure of the original MRSDF graph, computing slacks and criticalities in the process. We first determine cycle slacks for all streams in the SRSDF graph. We merge replicated actor instances. We merge replicated stream instances, remembering a stream's multiplicity, and taking its new cycle slack to be the minimum slack of its replicants. Finally, we compute a net criticality for the merged stream. Suppose a given stream has multiplicity k and a cycle slack of $(T - T_C)$ periods after pipelining actors. Then a VPR net criticality might be computed as: $crit = 1 - ((T - T_C)/T)^k$.

Figure 5.11 shows a sample MRSDF graph and its equivalent SRSDF graph with actor clustering. We do not show serialization edges, as they are unnecessary in this case. Actors $\uparrow_2$ and $\downarrow_2$ denote up-sampling and down-sampling by a factor of two, respectively. Suppose the implementation provides a single level of pipelining between actors, and that level can hold an ideal delay value (*e.g.* the enabled register queue from Chapter 3). Then we have the following cycle throughputs in clock periods per token: $T_{\{AB\}} = 2$, $T_{\{A\uparrow_2 C\downarrow_2\}} = 4$, $T_{\{A\uparrow_2 CDC'\downarrow_2\}} = 6$, $T_{\{CDC'D'\}} = 4$, $T = \max_C\{T_C\} = 6$. The cycle slack of cycle $\{AB\}$ is 4 periods, so a placer may add up to four levels of interconnect pipelining to its nets without degrading system throughput. However, all other nets are critical. Consider the original stream $C \to D$, which is replicated with a multiplicity of two. In the SRSDF graph, stream $C' \to D'$ has a cycle slack of 2 periods from cycle $\{CDC'D'\}$. Stream $C \to D$ has a cycle slack of zero from cycle $\{A \uparrow_2 CDC' \downarrow_2\}$. Consequently, the slack of stream $C \to D$ after merging replicated streams is also zero.

5.3.2.3 Average Rate SDF

A dynamic rate dataflow graph can be modeled as an MRSDF graph with *average* consumption/production rates. For example, an actor might, on average, consume 0.5 tokens per firing and produce 0.75. Average consumption/production rates may be collected empirically in simulation or derived by compiler analysis. They are independent of any schedule. Balance equations may be posed with non-integer values for tokens per firing, N_A. However, the resulting firing counts $\boldsymbol{q}$ may be related by irrational ratios and thus have no integer solution. To apply MRSDF-to-SRSDF unrolling, we need integer rates N_A and integer firing counts $\boldsymbol{q}$. We propose an *approximate* unrolling, as follows.

Consider an average-rate MRSDF graph G with repetitions vector $\boldsymbol{q}$. Choose a scaling factor α and form a graph G' representing α firings of G. Each actor $A' \in G'$ represents α firings of the corresponding actpr $A \in G$, with consumption/production rates: $N_{A'} = \alpha N_A$. This scaling must also be done for ideal delays, forming chains thereof. Graphs G and G' have the same repetitions vector, since any repetition of a balanced schedule is also balanced (the balance equations, $\Gamma \boldsymbol{q} = \mathbf{0}$ and $(\alpha\Gamma)\boldsymbol{q} = \mathbf{0}$, have the same eigenvector $\boldsymbol{q}$). Now form an integer-rate graph G'', where every actor $A'' \in G''$ has integer production/consumption rates $N_{A''} = \lfloor N_{A'} \rfloor$. We unroll G'' into an equivalent SRSDF graph *inexactly*. Graph G'' may have no balanced schedule due to its truncated production/consumption rates, but we adopt $\boldsymbol{q}$ as an approximation. Choose an arbitrary number of schedule repetitions to unroll, and obtain truncated firing counts. One way to do this is to normalize $\boldsymbol{q}$ such that its greatest component equals one, then use the firing counts $\boldsymbol{q}'' = \lfloor \alpha \boldsymbol{q} \rfloor$. Form the acyclic dependency graph (APG) of G'' by simulating a schedule of $\boldsymbol{q}''$ firings. The APG must now be closed to form an equivalent SRSDF graph, but the actors at its output may be the wrong ones to connect in feedback. We visit APG nodes in reverse topological order from the outputs. An instance $A''_i \in$ APG of an actor $A'' \in G''$ that fed an ideal delay in G'' can be connected to the corresponding ideal delay at the APG input, thus closing a feedback loop. An instance $A''_i \in$ APG that does not correspond so is culled. The resulting, *approximate* SRSDF graph can be analyzed for throughputs, slacks, and criticalities in the normal way.

5.3.2.4 CSDF, TDFPN

The average rate SDF model can estimate, to first order, the behavior of process networks with sequenced (stateful) actors and/or dynamic rates. However, it fails to capture the actual, cycle-by-cycle behavior of stateful processes. The cycle-by-cycle schedule of consumption and production determines when pipeline bubbles appear and when they force process stalls. Analyzing that level of behavior would yield more accurate throughputs than those based on average rates. Our throughput analysis can be extended to processes with static sequences of states, such as cyclo-static dataflow (CSDF) [Bilsen *et al.*, 1996], since they can be unrolled into an equivalent SRSDF graph. A cyclo-static actor is defined by a repeating sequence of k firings, each one consuming and producing a static, but possibly different, number of tokens. Its firings appear in the equivalent SRSDF graph as actor instances, chained by precedence edges. In principle, it should be possible to represent actors having dynamic state flow, such as TDFPN SFSMs, and to unroll them in approximation. A possible approach is to build a one-hot representation of each SFSM, with a boolean activation token for every state. Such a representation could be built using dynamic actors from boolean controlled dataflow (BDF) [Buck, 1993] or integer controlled dataflow (IDF) [Buck, 1994] and unrolled as an average rate SDF graph.

5.4 Process Pipelining

In Section 4.4.2, we proposed an approach for choosing logic pipelining depths for processes to optimize the system clock rate. That approach does not take into account that pipelining a process in a stream feedback loop may degrade system throughput. The cycle slack model introduced in Section 5.3.2 provides a mechanism for identifying which processes should not be pipelined. The cycle slack of a process is the minimum cycle slack of all cycles to which the process belongs. Equivalently, it is the minimum cycle slack of the process's adjacent streams. A process with a cycle slack of $s > 0$ periods may accomodate $\lfloor s \rfloor$ levels of additional pipelining without degrading system throughput. A process with zero cycle slack cannot accommodate pipelining without degrading tokens-per-clock-period throughput. However, pipelining it may improve the clock period and thus improve the overall, tokens-per-second throughput. In this case, pipelining is warranted only if the improvement in clock period outweighs the loss of per-period throughput. Consider a zero-slack process A in a system with throughput T, measured in clock periods per token. Adding d pipeline levels to A yields a throughput degradation of $(T + d)/T$ and a given clock speedup. Then pipelining is warranted if: $\textit{speedup} > (T + d)/T$. This condition can be added as a

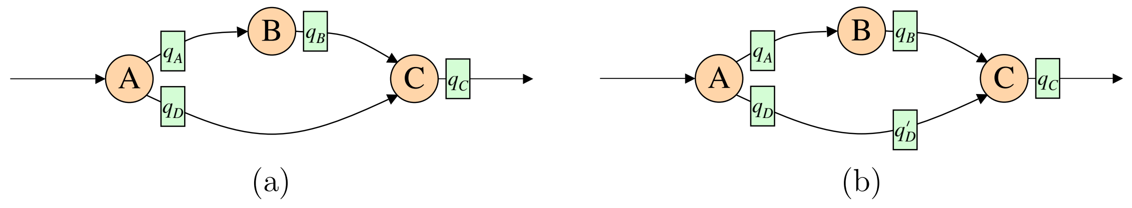


Figure 5.12: Process network with (a) unbalanced and (b) balanced pipeline delay on reconvergent paths

termination condition for Algorithm 4.1. That is, the algorithm should add one level of pipelining to the slowest process until either (1) the pipelined process becomes slower instead of faster, or (2) the throughput degradation exceeds the clock speedup. A process in a feed forward stream topology may be pipelined freely, having an effectively infinite cycle slack.

5.5 Pipeline Balancing

If pipeline delay on reconvergent paths is not balanced, then data may arrive mismatched at the point of reconvergence. The consumer would have to stall, waiting for a matched input set, and throughput would be lost. In sequential circuits, such mismatches are handled by adding registers to equalize pipeline delays. In process networks, the same goal can be achieved by adding buffering capacity. Balancing by adding buffering has several advantages to adding delays. It may be free if every stream buffer must be allocated to some quantized capacity. For example, the SRL16 based queue from Section 3.4.2 has the same area up to capacity 16. In such cases, adding buffering may be performed after place and route by setting queue parameters. Also, adding buffering may have better average delay and average throughput than adding delays if dynamic rates are involved.

Figure 5.12(a) shows an example process network with unbalanced pipeline delays. Suppose processes A, B, C are single-rate SDF actors, implemented as combinational cores, with output pipelining in unit-depth, enabled register queues (from Section 3.4.1). The reconvergent paths A-B-C (top) and A-C (bottom) have mismatched pipeline delays of two and one clock periods, respectively. With unit-depth buffers everywhere, each firing of A must wait for C to have consumed its previous input set, which means waiting an extra period for B. The best hardware schedule is only half throughput, forever alternating between firing B, then firing C together with the next A. Nevertheless, this is a feed forward network, so it should be able to operate at full throughput. We can balance the paths by adding a second buffer slot to path A-C (bottom), as in Figure 5.12(b). With two buffer slots on either path, the process network can fire at full throughput. In principle, it is not necessary to add pipeline delay to achieve balance. We can either expand buffer q_D to two elements (adding no delay) or cascade it with a second, unit-depth buffer q'_D (adding unit delay). The latter configuration would be preferred if q_D were intended for logic relaying and retiming, so as not to interfere with it.

We formulate pipeline balancing as a two phase process: (1) retime to determine where buffer capacity is needed, and (2) allocate buffers at those locations. Buffer allocation is normally possible only before place and route. After place and route, it may be possible to expand some buffers by configuration specialization, assuming those buffers had been allocated but restricted to less than maximum size. Otherwise, buffer reallocation would require an iteration of place and route.

To determine where buffer capacity is needed for balance, we use a form of architecturally constrained retiming. Retiming is a circuit transformation that moves registers in a sequential circuit. It was first introduced by Leiserson, Rose, and Saxe [Leiserson *et al.*, 1983] for improving system clock rate by balancing combinational delay among all register-to-register paths. Retiming naturally maintains a balance of register delays on reconvergent paths, since it replicates a delay when moving it across a fanout, and it merges delays when moving them (in unison) across a fanin. The formulation of retiming as a constraint satisfaction problem permits adding constraints to require that register delays be moved into particular locations in the circuit. In our case, those locations will correspond to logic and interconnect pipeline registers and stream buffers, whose allocation is mandated by other analyses (place and route, *etc.*). As retiming moves delays into those locations, it will naturally move balancing delays into other paths.

We reinterpret register motion as token flow, and we adapt retiming to balance pipeline delays in a process network. This interpretation assumes that every process is a single-rate SDF actor, firing at every clock period, and always consuming and producing one token per stream. This is a conservative but correct approximation for dynamic rate process networks, provided they consume and produce no more than one token per stream per firing. For generality and improved accuracy, the analysis can instead be applied to a representative, single-rate SDF graph derived by unrolling the process network, as described in Section 5.3.2.

5.5.1 Retiming

Retiming [Leiserson *et al.*, 1983] is formulated as a constraint satisfaction problem on the *lag* r_i of each combinational node i. Every node begins with a lag $r_i = 0$. Moving a register delay backwards across a node makes it evaluate one cycle later, increasing the lag by one. Moving a register delay forwards across a node makes it evaluate one cycle earlier, decreasing the lag by one. For convenience, the circuit is modeled as a *retiming graph*, where every node is a combinational component, and every edge is a two-terminal net weighted by the number of register delays on that net. Let $w_i(e)$ and $w(e)$ denote the number of delays on edge e before and after retiming, respectively. On a given edge $e_{u,v}$, the difference in endpoint lags indicates the number of delays moved into that edge during retiming: $w(e_{u,v}) - w_i(e_{u,v}) = r_v - r_u$. The number of delays on each edge must be non-negative: $w(e_{u,v}) \geq 0$. Combining these two relations yields the lag constraints: $w(e_{u,v}) = (r_v - r_u + w_i(e_{u,v})) \geq 0 \ \ \forall e_{u,v}$. Circuit retiming for optimizing clock rate involves additional constraints to ensure that combinational delays meet the target clock period. However, those constraints are not necessary for pipeline balancing, where we ignore combinational delay altogether. Retiming also requires that every cycle in the graph have at least one delay. However, this can be verified independently, since retiming cannot change the number of delays in a cycle.

Architecturally constrained retiming [Singh and Brown, 2001] is implemented by adding constraints of the form: $k_{u,v} \leq w(e_{u,v}) \leq k'_{u,v}$. Such constraints indicate the allowable number of delays on an edge. An edge corresponding to a net that cannot be pipelined requires: $w(e) = 0$. An edge corresponding to a single pipeline delay requires: $w(e) = 1$. An edge corresponding to a buffer of up to 16 elements requires: $0 \leq w(e) \leq 16$. An unconstrained edge may retime to use an arbitrary number of delays, for which new buffering resources would need to be allocated.

The constraint system described above is a special case of integer programming with known, polynomial time solutions. All constraints are of the form: $r_v - r_u \leq k$. Efficient solutions for two-variable integer programming include [Kannan, 1980] and [Feit, 1984]. In addition, it is possible to solve the system in polynomial time while optimizing a cost function that is a linear combination of lags r_i. A typical use is to minimize the total number of registers, $\sum_e w(e)$, using the substitution: $w(e_{u,v}) = r_v - r_u + w_i(e_{u,v})$. Efficient solutions for the constrained optimization problem include the mixed integer-linear programming (MILP) approach of Leiserson *et al.* [Leiserson *et al.*, 1983] and simplex / convex hull approaches. Retiming for pipeline balancing is simpler than conventional circuit retiming, since it does not involve constraints on non-integer, combinational delays. Our problem is also simplified because it need not consider register initial values, register enable, or register set/reset.

5.5.2 Feed-Forward Pipeline Balancing

Our interpretation of retiming for process networks equates each register delay with a buffer slot for a token. Architecturally constrained retiming moves tokens from arbitrary initial locations to locations where buffering is required or available. For convenience, a stream can be elaborated in the retiming graph into a chain of edges, each representing a different kind of buffering. Logic relaying, logic pipelining, and interconnect pipelining denote required buffering in D flip-flops, generating constraints of the form: $w(e) = L(e)$, $w(e) = W_p(e)$. Interconnect relaying denotes buffering of at least one and at most two in each relay station: $W_r(e)/2 \leq w(e) \leq W_r(e)$. An SRL16 based stream queue denotes buffering of 1 to 16: $1 \leq w(e) \leq 16$.

Tokens can be retimed from primary inputs forward, or from primary outputs backward. A convenient way to represent this is to leave primary I/Os unconstrained and allow retiming to add as much lag as necessary. Formally, this can be done by constraining a single, reference I/O to have zero lag and to retime everything else for minimum delay. This approach differs from traditional circuit retiming, which ties all primary I/Os to a common "host" node, to force them to retime synchronously, and to prevent modifying input-to-output latencies. There is no requirement of I/O synchrony or latency preservation in process networks, since every I/O is a stream with independent flow control. Nevertheless, it is possible to constrain particular streams to retime synchronously, the designer wishes it.

Retiming from primary I/Os is, in general, sufficient only for pipelining feed-forward streams. This is a consequence of the property that retiming cannot change the number of register delays in a feedback cycle. Retiming cannot push new tokens into a cycle, so it cannot help an under-buffered cycle satisfy its buffering constraints. To avoid posing an unsatisfiable retiming problem, constraints associated with feedback loops should be hidden from retiming altogether, *e.g.* by collapsing each strongly connected component into an individual node. The remaining retiming problem balances only feed-forward streams. Unfortunately, leaving feedback streams unbalanced may incur an otherwise avoidable loss of throughput. In the next section, we propose an approach for balancing feedback cycles by exposing retimable tokens within them.

5.5.3 General Pipeline Balancing

Pipeline balancing by retiming involves pushing tokens from initial locations into buffer locations. Pushing tokens from primary I/Os is insufficient, because it cannot push new tokens into feedback cycles. Instead, we must expose initial token locations directly within feedback cycles, and in sufficient number. Initial tokens may be identified from a program description either as initial buffer contents or as tokens emitted in the initial state(s) of each process. However, there may be insufficiently many such tokens to satisfy the buffering constraints of a feedback cycle. This mismatch indicates that the feedback cycle will operate at less than full throughput—a small number of tokens will propagate around a larger number of buffers, and the output buffer(s) of the feedback cycle will not have a valid token at every clock period. If the network had a known, static throughput of T periods per token, then we could model the schedule of those tokens by *c-slowing* the network by T, *i.e.* converting each token into a chain of $c = T$ tokens, treating the first as valid and the remaining $T - 1$ as bubbles. These tokens can then be retimed to satisfy buffering constraints. Any buffer retimed to hold b balancing delays would, in practice, need only $\lceil b/T \rceil$ buffer slots. We develop and generalize this interpretation of retiming for general process networks.

In Section 5.3.2 we describe how to unroll a process network into a representative, single rate SDF graph, and how to find its throughput T. The representative graph for a dynamic rate process network is based on average consumption/production rates, so we have only its average throughput T. We can retime this graph to obtain buffer sizes that are balanced on average. Deviation from the average rates might lead to a temporary pipeline imbalance (under-buffering), but the effect on average throughput would be small. The effect of such deviations might be hidden by a small, judicious increase of buffer sizes beyond the prescribed averages.

We apply pipeline balancing to a process network as follows. Form the equivalent, average rate SDF graph G. Find its throughput T in clock periods per valid token. Form a retiming graph for pipeline balancing of G. Replicated streams in G (connecting unrolled actor instances) are given identical buffering constraints. Precedence edges are given a constraint: $w(e) \geq 1$, indicating that the endpoints are not concurrent. Apply a *c-slow* transformation with $c = T$. The *c-slow* transformation, initially proposed by Leiserson *et al.* [Leiserson *et al.*, 1983], converts every register delay into a chain of c delays, comprising c independent and non-interacting sets of data. In our interpretation, one of these sets represents valid tokens, and the remaining $T - 1$ sets represent bubbles. Now retime to satisfy buffering constraints. Every edge representing a pipeline delay will have been satisfied with $w(e) = k$, regardless of stream replication. Every edge representing a balancing buffer will have accumulated a certain number of delays, but only the delays corresponding to valid tokens require buffering capacity. A non-replicated stream whose edge has b delays will hold at most $\lceil b/T \rceil$ valid tokens. A replicated stream may hold valid tokens on some or all of its replicants. To distinguish valid and bubble tokens, we assign to each token a label equal to its delay distance from an arbitrary reference input, modulo T. Valid tokens will all have an identical label t ($0 \leq t < T$). Nevertheless, execution makes them advance in unison through every t, so we must consider every t when counting valid tokens. Suppose each edge e_m of a replicated stream has w_t^m tokens of label t. Then the stream buffer will contain at most: $\max_t\{\sum_m w_t^m\}$ valid tokens.

5.6 Serialization

Multi-rate and dynamic rate systems typically have components operating at different frequencies, with some components frequently idle. A designer may with to serialize the implementation of such components to save area, or to improve the component's clock rate. For example, an infrequent adder may be implemented in byte-serial, nibble-serial, or bit-serial fashion. The serialized adder will be smaller but slower. Similarly, a low frequency stream may be serialized to transmit each token as a sequence of narrower tokens over correspondingly fewer wires. Serialization affects the delay, firing rate, and/or throughput of a serialized component. It must be be applied judiciously to avoid degrading the system throughput. We can use the slack and activity models from Section 5.3.2 to guide which components should be serialized and by how much.

Serializing a process by a factor α is equivalent to converting a single state into a sequence of α states. The transformation increases firing rate by a factor α and reduces cycle slack by α periods. We can verify whether the transformation degrades system throughput as follows. Suppose the representative SDF graph has a repetitions vector $\boldsymbol{q}$, normalized so that its greatest component is one. Equating a firing with a clock period, the normalized $\boldsymbol{q}$ represents actor activity, *i.e.* the probability of firing per clock period. The increased activity of a serialized actor should not exceed one firing per period. An actor A can be serialized by a factor α without degrading system throughput if: $\alpha q_A \leq 1$ and $CycleSlack_A \geq \alpha$. If serialization is applied only to a particular state whose probability is p (per firing), then the transformation increases average firing rate by a factor $p\alpha$ and reduces cycle slack by $p\alpha$ periods. Thus, a probability-p state of an actor A can be serialized by a factor α without degrading system throughput if: $p\alpha q_A \leq 1$ and $CycleSlack_A \geq p\alpha$.

A low frequency stream may be serialized using a time-division multiplexing

(TDM) scheme. A stream serialized by a factor α transmits each token as a sequence of α sub-tokens, each sub-token having $1/\alpha$ as many bits. The serialized stream may then be implemented with $1/\alpha$ as many wires (not accounting for the overhead of flow control). The transformation increases token frequency by α and reduces cycle slack by α periods. Suppose the representative SDF graph has a repetitions vector $\boldsymbol{q}$, normalized so that its greatest component is one, to represent actor activity. Equating a firing with a clock period, the frequency of a stream $A \to B$, in tokens per period, is: $f = q_A N_A = q_B N_B$, where N_A is the average number of tokens produced to the stream by A per firing, and N_B is the average number of tokens consumed from the stream by B per firing. If this frequency is less than one token per period, then serialization is desriable. However, the increased frequency of a serialized stream should not exceed one token per period. Serialization may be implemented transparently to the producer and consumer processes, using a serializing output buffer at the producer and a parallelizing input buffer at the consumer. In this case, a stream $A \to B$ having token frequency f may be serialized by a factor α without degrading system throughput if: $\alpha f \leq 1$ and $CycleSlack_A \geq \alpha$. If stream serialization is implemented with modification to the producer and consumer, then their respective effects on system throughput must also be considered. For example, if stream consumption is implemented by serializing the consuming state into α states, then that state must be tested for its effect on system throughput using the approach in the previous paragraph.

The converse to serialization is parallelization. Parallelizing a critical process or stream may be desirable for improving system performance. For instance, a process may be replicated or loop-unrolled to consume/produce more tokens per period. Similarly, a stream may be widened to transmit more tokens per period. In general, such transformations are complicated by hazards and limited by data dependencies. For example, parallelizing a stateful process is no easier than vectorizing a loop in an imperative language. Parallelization is vastly simplified in statically schedulable dataflow models and has been implemented in many forms, *e.g.* the MRSDF-to-SRSDF unrolling of Williamson [Williamson, 1998], and the horizontal and vertical transformations in StreamIt [Gordon *et al.*, 2002a]. The slack and activity models from Section 5.3.2 can be used to identify critical processes and streams and guide where to apply parallelization with maximum benefit and minimum area.

Chapter 6

Streaming Programmable Platforms

Programmable platforms have emerged as an attractive alternative to ASIC and custom chip design in many application domains. A platform is an off-the-shelf part, typically including one or more programmable cores for computation and a variety of programmable off-chip interfaces. A platform based system is defined primarily as software, separated from the generic hardware of the platform. The platform itself may be general purpose, *e.g.* Xilinx Virtex 4 series FPGA [Xilinx, 2005c], or domain specific, *e.g.* Cradle CT3600 series for video [Cradle, 2005], *e.g.* Intel IXP series for networking [Intel, 2005].

The primary advantage of platform based design is that it avoids the difficulty and cost associated with fabricating a new chip, including multi-million dollar mask sets, transistor layout and sizing, signal integrity, parasitics, electromigration, and other deep submicron effects. System cost is instead shifted into a per-part cost to pay for the off-the-shelf platform. Traditionally, platform based design was more profitable than an ASIC only at very low volumes. However, as mask costs continue to rise, the break-even point at which platforms cease being more profitable shifts to larger volumes, encompassing more application domains.

Platforms can play a key role in enhancing design reuse. By its nature, a platform separates system design into a software and hardware layer. Ideally, a next-generation, compatible platform would innovate the hardware and run the same software at higher performance. This has been the case for uniprocessors, where architectural families such as Intel x86 and IBM System 390 have retained legacy software for decades. However, modern platforms are larger and more heterogeneous, and they usually lack the proper design abstractions to support software longevity. Where ever their programming model exposes device details, it forces a system designer to tie software to the architecture, and thus undermines software reuse. For example, inter-core communication that was hand-scheduled as DMA transfers on a shared bus will likely not be reusable on a next-generation device with a network on chip (NOC).

We have argued that a streaming discipline provides a key abstraction of communication to enable retargeting and reuse of software. Stream connected modules are agnostic of the timing and implementation of streams, so a platform is free to innovate stream related hardware, *e.g.* providing a better network on chip (NOC) or larger stream buffers. The platform need only retain a compatible stream interface for every module.

A streaming discipline also enables an abstraction of area. A next generation platform is usually larger, providing more programmable cores or a larger reconfigurable fabric. Software that was designed to use a particular number of cores, ALUs, or gates cannot normally benefit from additional hardware. The algorithms and schedule of communication are already bound. The missing ingredient is an abstraction of area and a methodology to map a computation of arbitrary size into a device of actual size—analogous to the way a uniprocessor ISA hides the number of function units, but at the system level. A streaming design has the advantage of timing independence, whereby stream connected modules may be separated by large delay. In fact, stream connected modules need not be resident and active at the same time. This flexibility suggests an abstraction of area by *virtualization*, where an arbitrarily large network of stream connected modules is time shared on an arbitrarily small number of module processors. A virtualized platform needs few mechanisms beyond those of a basic streaming platform: a multi-context configuration store, a configuration controller, and software to partition and schedule the virtually large design.

In this chapter, we discuss how to accommodate efficient, scalable streaming in a programmable platform. In Section 6.1 we discuss a basic approach, extending an FPGA-like platform with custom resources for streams. In Section 6.2 we discuss a *paged* FPGA-like platform, which is partitioned into stream connected slices called pages. This partitioning is indended to take advantage of locality within modules, separating local and global interconnect into a physical, two-level hierarchy. Such a platform would support faster compile times as well as fault tolerance through resource sparing. In Section 6.3 we discuss a virtual, paged platform that incorporates an abstraction of area. The platform is dynamically reconfigurable, using run-time support to map many virtual pages into the available, physical pages. The virtualized platform and run-time support are defined more completely in SCORE [Caspi *et al.*, 2000a].

A major challenge for targeting a paged platform is partitioning a computation into stream connected pages. Such a partitioning must be sensitive to the possible effects of inter-page communication delay on system throughput. The actual communication delay may be unknown at compile time, particularly with virtualization (where the number of pages is unknown) or resource sparing (where the relative layout of pages is unknown). In Section 6.4, we discuss techniques for automatically restructuring a process network of arbitrarily sized, stream connected modules into a network of fixed-size, stream connected pages. The optimal size of a page remains an open research question, which could be answered by using our proposed techniques to map a set of benchmark applications to a variety of page sizes.

A number of so-called streaming platforms exist, including Pleiades [Wan *et al.*, 2001], Imagine [Khailany *et al.*, 2001], and TRIPs [Sankaralingam *et al.*, 2003]. However, their notion of a stream differs from ours. While it may be possible to map our streaming model, TDFPN, to those architectures, the results may not be optimal. The platforms proposed in this chapter are built from the ground up to support the streaming model and synthesis methodology of Chapters 2 and 3 on a reconfigurable fabric.

6.1 Basic Platform

A minimalist approach to creating a streaming, programmable platform is to add some custom resources to an FPGA. A field programmable gate array (FPGA) is a highly generic programmable platform, capable of emulating arbitrary digital logic using look-up tables (LUTs) and registers. Modern FPGAs include many custom resources to improve efficiency, including carry chains, SRAM memory, multipliers, and even entire microprocessors. We might consider continuing this trend by adding resources to make the streaming methodology of Chapter 3 more efficient. The key ingredients would be stream interconnect, stream buffers, and stream interfaces for other custom resources like memory and processors.

6.1.1 Stream Interconnect

Existing FPGA interconnect is typically segmented into different lengths to best manage wire and stub capacitance. Short wires have low capacitance and switch fastest. Long wires have high capacitance and switch slowest. To create long routes, wires are cascaded through intermediate buffers or pass transistors, or pipelined through logic blocks. Our streams may be pipelined in FPGA logic blocks using registers or relay stations (minimal, two-element queues). However, pipelining streams in logic blocks has several problems: (1) it wastes logic resources, (2) it consumes additional, local interconnect for controlling relay blocks, and (3) it is inefficient in existing tool flows (as we discovered in Section 4.5). Thus, we should consider adding pipelining resources to at least some of the interconnect. Streams may then be overlaid on top of, and without disturbing, the existing FPGA interconnect.

6.1.1.1 Registered Interconnect

Pipelining streams with registers can be supported by adding registers directly to the interconnect. Registered routing switches have been proposed for several reconfigurable architectures, including HSRA (fat tree interconnect) [Tsu *et al.*, 1999], conventional FPGAs [Singh and Brown, 2001], and SFRA ("corner turn" switches) [Weaver *et al.*, 2004]. Their approaches rely on retiming, during or after placement, to take advantage of interconnect registers. Retiming at such a late stage has the problem that the existing placement may be unable to accommodate the new resources required by retiming. Specifically, retiming may create register delays on non-critical paths to balance parallel, critical paths. To accommodate those delays without modifying placement, the above architectures incorporate input registers in logic blocks. Furthermore, backwards retiming of registers with initial values may require additional control, so it is usually forbidden, limiting the efficacy of retiming. Stream pipelining avoids these problems, since it does not require post-placement retiming of arbitrary logic. Balancing pipeline delays on parallel streams is desirable for avoiding pipeline bubbles, but those delays can be accommodated in stream buffers, not in generic logic blocks. Also, pipelining a stream does not involve backwards retiming of initial values, even if the stream has initial contents.

6.1.1.2 Relayed Interconnect

Pipelining streams with relay stations can be supported by adding custom relay blocks to the interconnect. A relay station is a minimal, two-element queue that combinationally decouples all inputs from all outputs. Its implementation can be very small, using two data registers, a data multiplexer, and a three-state FSM[1]. The FSM and register enable can be shared across all bits of a multi-bit stream. This sharing requires all bits to arrive at the same time, so it is best served with bussed interconnect, described next.

6.1.1.3 Bussed Interconnect

If a stream is pipelined with simple registers, then all its bits are, in principle, equal. They may be routed and registered independently on an FPGA-style, bit level interconnect. Nevertheless, bits must arrive synchronized at any object having a stream interface, including relay stations and queues. Bits arriving out of sync would require either (1) additional registering to realign in time, or (2) separate flow control. We prefer to avoid those overheads, keeping just one set of flow control bits per stream, and amortizing the cost of that flow control over all the data bits of the stream. The easiest way to guarantee synchronized arrival is to constrain all bits of a stream to take the same route. That structure can be exploited in hardware using *bussed routes*, wherein the interconnect switches and registers of an entire word are controlled by a shared configuration. Bussed routes can thus amortize the area of route configuration across all bits of a multi-bit stream. We might imagine a basic stream width in hardware, say four data wires plus valid and back-pressure wires. A narrower stream would underutilize the data bits, whereas a wider stream would have redundant flow control.

[1] A possible implementation for a relay station is the shift register queue from Figure 3.7(bottom). At depth two, it would be optimized to need no shift register and no address register.

FPGAs have traditionally shunned bussed interconnect, since it leads to some fragmentation, *i.e.* unused bits on a bus. Fragmentation is particularly expensive in FPGAs, where the area of programmable interconnect dominates chip area, easily taking 97-98% [DeHon, 1996]. Nevertheless, bussing streams may be more compelling than bussing general purpose interconnect. Streams represent inter-module communication, which typically has fewer narrow control signals than random logic, and consequently less fragmentation. It is possible to apply bussing only to the stream interconnect and not to the local interconnect that handles random logic. Furthermore, the area saved by bussing streams is reflected in more resources, including interconnect switches, relays, stream buffers, and any stream interface. The optimum bit width for bussing remains an open research question, particularly if bussing is restricted to streams. The optimum may lie in providing a mix of stream widths, for example a boolean stream and an eight-bit data stream. Ye *et al.* [Ye and Rose, 2005] study bussed interconnect for conventional FPGAs (without streaming), finding that converting 50% of all tracks into 4-bit busses can provide a system area savings of about 10%.

6.1.2 Stream Buffers

Our mapping of streaming applications to an FPGA (Chapter 4) found that stream buffers comprise 38% of application area, assuming stream buffers are implemented in Xilinx logic blocks using SRL16 mode. This relatively high fraction suggests that a streaming platform should provide custom resources for buffering streams. Our basic requirement is a FIFO queue with an appropriate stream interface, including valid, back-pressure, and reservation (as discussed in Section 3.7.2). Interestingly, Xilinx Virtex 4 FPGAs include FIFO controllers for block RAM, including an “almost full” signal that could serve as back-pressure with reservation. Adapting their controller for our stream protocol would require minimal additional logic. However, Block RAM is efficient only for large buffers (in fact, the smallest block RAM FIFO supported in Virtex 4 is 512 elements deep). The common case for our streaming applications is small buffers of no more than 16 elements. A complete streaming platform would need to provide efficient resources for buffers of this size. Stream buffers should be large enough to accommodate post-placement pipeline balancing (*e.g.* additional 1-2 elements) and reservation capacity for stream-based logic pipelining (*e.g.* additional 3-4 elements). In addition, for general process networks, stream buffers need a certain minimum capacity for correctness and an excess to smooth out dynamic rate variations. Thus, a cumulative capacity of 16 seems appropriate. We have also mentioned above the need for custom relay stations blocks, which are two-element queues.

6.1.3 Stream Interfaces

A streaming platform should provide streaming interfaces for most, if not all, custom resources, including multipliers, memory, embedded processors, and off-chip I/O. Modern FPGAs already provide input and/or output registers for hard blocks such as multipliers and memory, in recognition of the need to pipeline routes to these blocks. It would not take much more circuitry to provide a full stream interface for these blocks. Often these blocks are word oriented and will amortize one instance of stream flow control for an entire data word (*e.g.* 18-bit multiplier on Xilinx Virtex II, 8- to 48-bit DSP48 on Xilinx Virtex 4). A streaming block interface is easier to use than manually pipelining an RTL description to accommodate a block's mandatory input/output registers. Nevertheless, the stream interface can be made optional, by disabling or ignoring flow control, to give a designer the choice of using a block conventionally via RTL.

A stream interface requires, at minimum, flow control signals and input/output queues. An input or output queue can be modeled as a unit-depth, enabled register queue, as in Section 3.4.1. This structure is only minimally larger than an input/output register, requiring an AND gate and a one-bit valid register. The enabled register queue has a combinational path connecting input and output back-pressure, so it does not fully decouple input from output. However, as we verified in Chapter 4, flow control is fast and seldom in the critical path, so pipelining logic with enabled register queues is effective. In fact, pipelining a custom block with a fixed number of stages can be modeled as a cascade of enabled register queues, with stages of the block between them. To pipeline back-pressure on block inputs and outputs, it suffices to use relay stations. The required flow control resources are minimal and could be built directly into the block. For more flexibility, flow control could be implemented in adjacent LUTs or in specialized "flow control" blocks.

6.1.3.1 Arithmetic Blocks

Stream flow control for an unpipelined arithmetic block such as a multiplier is fairly simple. For the multiplier, we have: $\mathit{fire} = i_v^1 \wedge i_v^2 \wedge \neg o_b$, $o_v = \mathit{fire}$, $i_b = \neg \mathit{fire}$. For a block with an input N-tuple $\boldsymbol{i}$ and an output M-tuple $\boldsymbol{o}$, we have: $\mathit{fire} = \prod_{n=1}^{N}(i_v^n)\prod_{m=1}^{M}(\neg o_b^m)$, $i_b^n = \neg \mathit{fire}\ \forall n$, $o_v^n = \mathit{fire}\ \forall m$. Pipelining can be modeled as a cascade of enabled register queues.

6.1.3.2 Memory Blocks

A conventional block RAM interface in an FPGA involves signals for data, address, and access modes. It often requires requires input or output registering to guarantee proper timing. A streaming memory interface packages the same signals into streams. Such an interface is naturally synchronous and pipelined, since it must be surrounded by input/output queues (enabled register queues or relay stations). If the queues are integrated into the memory block, then their registers can serve the purpose of input/output registering for proper timing. Thus, the incremental cost of the streaming interface is small, merely in flow control.

A streaming memory interface simplifies memory access in several ways. First, streaming flow control is a natural way to deal with unknown memory latency. Thus, a streaming memory can be transparently implemented as DRAM with refresh. In contrast, FPGA block RAM is usually SRAM, to guarantee a fixed access latency. More generally, a streaming interface can hide the implementation of memory altogether, be it SRAM, DRAM, a composition of memories slowed by interconnect delay, or even an off chip memory. In contrast, RTL access to memory requires separate interfaces and control for every memory implementation. A streaming memory interface can also integrate an address generator for common access modes. This would obviate the address and mode signals, leaving only streaming data. A sequential or strided access pattern is common in digital signal processing and may be worth specializing. A FIFO access pattern is particularly valuable and can serve as a large stream buffer.

6.1.3.3 Off-Chip I/O

Communication with external components can be handled in several ways. A stream interface can be mapped directly to device pins, to support streaming communication between devices. This may be useful for supporting transparent partitioning of a large computation across multiple, concurrent devices. To support standard off-chip interfaces (USB, PCI, *etc.*), the on-chip interface controller should have a streaming interface within the platform. This notion is similar to streaming memories, packaging data and protocol-specific signals into streams. Likewise, the streaming interface can be specialized for common access modes. In this way, streams can simplify and abstract an off-chip interface, provide buffering for it, and naturally handle unknown or dynamic rates and latencies.

6.1.3.4 Embedded Processors

Microprocessors usually communicate with other components through specialized bus protocols or DMA (direct memory access). While memory access is usually well integrated into the processor instruction pipeline, other forms of communication are not, requiring polling, stalling, or interrupts. Efficient streaming can be fully integrated into a processor instruction pipeline, as suggested for SCORE [Caspi *et al.*, 2000a]. We add a stream read instruction `sread(s,v)` and a stream write instruction `swrite(s,v)`, where `s` denotes a stream identifier and `v` denotes a token value (either one may be a register or, where appropriate, an immediate). These instructions are entirely analogous to memory load and store, with stream identifiers `s` constituting a kind of address space. Single-cycle and non-blocking stream access can be supported with the same techniques as memory access, using load/store units and an instruction reorder buffer. Stream writes can be handled in a write buffer, without stalling the instruction stream. Stream reads can wait for a token while other instructions proceed, possibly with speculation. If a stream is not ready for a long time, then a stalled stream access can trigger an exception and a context swap, to be continued later. The stream space can be virtualized using a *stream look-aside buffer* (SLB), analogous to a translation look-aside buffer (TLB). This allows handling a larger number of streams, protecting stream access for different processes, and hiding the implementation of each stream. Some stream accesses may be routed to stream ports, while others may be routed directly to a local memory buffer.

6.1.4 Compilation Issues

The netlist given to place and route (PAR) must distinguish between stream connections and conventional connections. Stream connections can be automatically pipelined, and they may be restricted to particular wires. The two kinds of connections can be routed independently if stream interconnect is very distinct from conventional interconnect, *e.g.* registered, bussed, and connected to custom resources such as queues. However, stream interconnect may be identical to conventional interconnect except for the presence of additional routing registers, as in [Singh and Brown, 2001]. In this case, it is easy to see how connections of one kind could be implemented on routes of the other kind. For performance, PAR should begin by allocating stream connections to stream routes and conventional connections to conventional routes, but it will have the freedom to spill excess connections to the other kind of route. Ultimately, stream connections must reach logic blocks, so there must be bridges between stream and conventional interconnect. Those bridges may be the logic blocks themselves.

The compilation flow for a streaming platform is essentially identical to the flow targeting FPGAs, including the optimization flow of Section 5.1. The only difference is that logic synthesis and PAR must be made aware of new custom resources.

6.2 Paged Platform

The natural layout for a module based design is one where each module is localized, but different modules may be distant. This layout does not use stream interconnect within modules, nor conventional interconnect between modules. A reconfigurable architecture can exploit that structure to save interconnect area by introducing a two-level hierarchy in hardware. The reconfigurable fabric would be sliced into fixed size *pages*, with interconnect between pages being strictly streaming, and interconnect within pages being strictly non-streaming. Pages, resembling miniature FPGAs, would be laid out in a regular, two-dimensional grid, connected by a streaming network on chip (NOC).

A paged architecture can provide benefits to system area, compile time, and software longevity. It may save area in interconnect, since it need not provide both streaming and non-streaming interconnect everywhere. It may reduce place and route time by dividing the problem into two levels, within a page and between pages. It would also promote software longevity by enabling a next generation device to innovate the streaming NOC without obviating paged software. However, a paged architecture also incurs overhead associated with partitioning a computation into pages, affecting system area, performance, and compile time. Partitioning leads to area fragmentation, *i.e.* underuse of resources in each page, and to communiation delay between pages. It also adds to compile time. The total effect of paging on area, performance, and compile time remains an open question and would be a function of the page definition (size, I/O, *etc.*).

6.2.1 Page Definition

A page is a fixed-size slice of reconfigurable resources with a streaming interface. It has a particular size and a particular number of stream ports. Each stream port can be equipped with custom resources such as a stream queue. A page would be configured to run one or more streaming processes. Compilation and resource allocation within each page is no different than for a non-paged platform. The key operational difference is that modules must fit within pages, or be partitioned to fit within pages, since communication between pages is strictly streaming.

Page size determines the basic granularity for streaming modules. A small page mandates small modules, whereas a large page permits large modules (or multiple small modules). The choice of page size poses an interesting trade-off. A larger page lowers the area cost of streaming, since that area will be amortized over larger modules. However, a page that is too large suffers from internal interconnect delay. It may also lose density if it contains multiple modules and must implement the streams between them in generic, LUT-based resources.

Figure 4.10 in Chapter 4 shows the distribution of SFSM areas on a Xilinx Virtex-II Pro FPGA for seven streaming multimedia applications. Discounting stream queues, we found that 87% of SFSMs are smaller than 512 4-LUTs. This suggests that a page with 512 4-LUTs would accommodate most SFSMs without having to partition them. Particularly large SFSMs would still need to be partitioned. Particularly small SFSMs could be clustered to fill pages.

6.2.2 Heterogeneous pages

A paged architecture may contain more than one kind of page. Particularly useful page types include memory pages, processor pages, and off-chip I/O pages. Heterogeneous pages are a convenient way to package custom resources with streaming interfaces, as in a non-paged architecture. In a paged architecture, those resources connect only to the global stream interconnect.

6.2.3 Hierarchical Place and Route

Conventional place and route considers an entire system at once. First, all components are placed using distance-based timing estimates. Then, all components are routed based on detailed timing estimates and availability of detailed routing resources. Both parts of the problem are NP-complete, and even approximate solutions have super-linear complexity with respect to the number of components. A paged architecture mandates a hierarchical approach to place and route: (1) partition into pages, (2) for each page, place and route its contents, and (3) place and route indivisible pages. The hierarchical approach can be faster than monolithic place and route in several respects.

Suppose we are mapping a netlist of size N to a paged architecture with P pages of equal size. Following ideal partitioning, there will be P instances of place and route on netlists of size N/P (within pages) and one instance of place and route on a netlist of size P. Suppose there are relatively few pages, s.t. $P \leq N/P$. Suppose the complexity of monolithic place and route is $O_{\text{monolithic}} = O(f(N))$, where f is super-linear in N. Then the complexity of hierarchical place and route will be lower:

$$\begin{aligned}
O_{\text{hierarchical}} &= P \cdot O(f(N/P)) + O(f(P)) \\
&\leq P \cdot O(f(N/P)) + O(f(N/P)) \\
&= (P+1) \cdot O(f(N/P)) \\
&\leq ((P+1)/P) \cdot O(f(N)) \\
&\approx O(f(N)) \\
&= O_{\text{monolithic}}
\end{aligned}$$

This is a simplified analysis, since the complexity of place and route depends also on the number of nets. However, the number of nets is bounded between N and N^2, so a similar analysis would hold.

Of course, hierarchical place and route requires an initial step of partitioning, which adds complexity. This obfuscates the question of whether the monolithic or hierarchical approach is faster. If partitioning can be made fast, while place and route only get harder with each device generation, then the hierarchical approach eventually becomes faster. For general circuits, the problem of area balanced, I/O constrained partitioning is known to NP-complete, but there are efficient, linear approximations based on min-cut/max-flow [Yang and Wong, 1994]. In Section 6.4, we propose a heuristic, throughput aware approach for partitioning process networks.

6.2.4 Area Fragmentation

Partitioning a computation into pages invariably introduces some area fragmentation, or underuse of resources in each page. Partitioning for maximum area utilization is usually the wrong thing to do, since it may separate communicating modules over long distances and degrade system throughput. Even if partitioning did target maximum utilization, it would still be limited by the I/O capacity of each page. A page may be able to accommodate the area of an additional module but not its I/O streams, in which case the area remains unused. Thus, any mismatch between modules and pages leads to some fragmentation.

Area utilization in pages is a function of both the partitioning approach and the page definition. Clearly, performance optimal partitioning would waste more area than area optimal partitioning. A mismatched ratio of page area to page I/O would also lead to fragmentation. A theoretically optimal ratio may be guided by Rent's Rule, an empirical power law relating circuit size and I/O: $IO = CN^p$, where $p \in (0, 1)$ is the Rent parameter, and C is a constant [Landman and Russo, 1971] [Lanzerotti *et al.*, 2005]. Rent's Rule based, programmable interconnect is studied in [DeHon, 2001].

6.2.5 Fault Tolerance

A paged architecture supports a limited form of fault tolerance through resource sparing. That is, an entire page can be marked as faulty and removed from the resource pool. Other pages remain unaffected, since they share no resources with the faulty page. Furthermore, pages are translatable. Thus, avoiding one faulty page does not require recompiling any page contents, only modifying the final placement and routing of indivisible pages. In contrast, FPGAs route through the entire area of the fabric, so a local fault can interfere with very distant components. Avoiding a fault in an FPGA usually requires a complete, device-wide place and route. Online fault tolerance would possible in a multi-context platform if page PAR were performed online, as for the virtual paged platform in Section 6.3.

6.2.6 Compilation issues

The biggest challenge in compiling to a paged architecture is partitioning and packing to fill pages. The job of a partitioner is to transform a process network of arbitrarily sized processes into a process network of fixed size pages. Small processes may be packed together, while large processes must be partitioned into communicating subprocesses. Page partitioning must be done early in the compilation flow to preserve knowledge of processes. We discuss page partitioning further in Section 6.4.

The compilation flow for a paged architecture is modified from the flow of Section 5.1 in two ways: it includes page partitioning, and it splits place and route into two phases (within and between pages). The overall flow is as follows:

1. *High level analysis.*
   (serialization, buffer bounding)

2. *Page generation.*
   (page partitioning, process pipelining, synthesis within pages, PAR within pages)

3. *Page composition.*
   (stream aware PAR of indivisible pages, stream pipeline balancing, buffer parameter setting)

6.3 Virtual Paged Platform

Streams support efficient computation in the presence of long communication delay, by pipelining and buffering that communication. Just as streams permit communication between devices, they also permit communication between different contexts of a single, time-shared device. An individual context need not know how many other contexts are being time shared. Thus, streaming and context swapping provide a means for abstraction of area, or virtualization. Stream based virtualization provides two key benefits to system design: efficient run-time reconfiguration (RTR), and automatic technology scaling. A streaming application can be automatically partitioned into different contexts, using streams to buffer intermediate data between contexts, and using token flow to guide reconfiguration. With this automation, an application can be retargeted to different size platforms, automatically benefiting from higher performance on larger platforms. With proper software and hardware support, streaming is a key enabler for software longevity and scaling for large, reconfigurable systems.

Virtualization can be added to a paged reconfigurable architecture with only incremental cost. The primary requirements for supporting virtualization are a large context store, reconfiguration control, and algorithms for context partitioning and scheduling. This section is dedicated to those mechanisms. Many of the ideas and mechanisms were originally developed for SCORE, including an instance of a streaming, paged, virtual platform [Caspi *et al.*, 2000b] [Caspi *et al.*, 2000a] [Markovskiy *et al.*, 2002] [Markovsky, 2004]. We redevelop them here as incremental additions to our streaming model and paged architecture.

6.3.1 Stream Based Virtualization

Run time reconfiguration (RTR) is a general technique for exploiting the power of reconfigurable platforms by mapping a large application as a collection of separate but interacting device contexts (*e.g.* [Eldredge and Hutchings, 1994] [Jones *et al.*, 1995] [Wirthlin and Hutchings, 1996] [Luk *et al.*, 1997] [Hudson *et al.*, 1998] [Swaminathan *et al.*, 2002]). Compared to single context execution, RTR can:

- Reduce the size of the platform required to solve a *phased* or *multi-mode* problem, with different contexts containing the datapath of different phases/modes.
- Reduce the size of a platform required to solve a *multi-rate* problem by time-multiplexing low-throughput components and keeping high-throughput components persistent.
- Reduce area and improve performance by specializing a general application to a particular mode or data set, either by instantiating different contexts, or by modifying contexts.
- Offer time-space trade-offs to a system designer, so a problem with modest throughput requirements can be implemented in economical hardware.

Unfortunately, RTR is not well supported in commercial architectures and commercial tool flows. There is no accepted, unifying model for how to describe the decomposition of an application into contexts and how to implement their sequencing. The designer is usually left with the entire job of partitioning into contexts, scheduling contexts, and triggering and managing reconfiguration in hardware.

Stream based virtualization provides a *systematic* and *efficient* way to use RTR. The decomposition and phasing of a streaming application as contexts is implied by the application's stream structure, which exposes communication dependencies, throughputs, and dynamics. That information enables software tools to automatically cluster streaming modules into different contexts and to sequence them based on token flow. Such automation reduces the system designer's job and hides the hardware mechanisms used for managing reconfiguration. Efficient time multiplexing is supported by making stream buffers large enough to allow each context to run for a long epoch. This allows a system to tolerate and amortize long reconfiguration times—thousands of cycles or more. In contrast, non-streaming approaches to virtualization must reconfigure as often as every clock period, which requires specialized hardware support that is area and power intensive (*e.g.* WASMII [Ling and Amano, 1993], TMFPGA [Trimberger *et al.*, 1997]).

Stream based virtualization also supports automatic performance scaling on next-generation hardware. RTR works by time multiplexing a large application on limited hardware. Ideally, application performance would be higher on larger hardware, since more parts of the application could execute concurrently. With the advent of automatic partitioning and scheduling, a streaming application could be automatically retargeted to larger hardware. Thus, an entire system could be retargeted to a larger, next-generation platform and automatically enjoy performance improvement, riding Moore's Law without redesign. Such improvement can continue until the platform is large enough to contain the entire application in one context.

6.3.2 Virtual Paging, Binary Compatibility

The primary challenge in retargeting a streaming application to a next-generation platform is recompilation, including partitioning and scheduling new contexts. Recompilation is certainly easier than redesign, but it still has drawbacks, namely that it precludes binary compatibility between platform generations. Binary compatibility in uniprocessors has proven to be an invaluable way to retain legacy software across device generations, providing performance scaling for shrink-wrapped software with zero involvement from the original software developer. In uniprocessors, performance scaling is enabled by the ISA's abstraction of hardware size (such as number of function units), and binary compatibility is enabled by supporting that abstraction in hardware (issue logic, reservation stations / reorder buffers). A binary compatible, reconfigurable platform would need a similar unit of abstraction *in hardware* that is supported across device generations. That unit can be a page, as in Section 6.2.

A paged, virtual platform is largely identical to a paged platform, but it runs large applications using RTR. The application and reconfigurable fabric are both sliced into fixed size pages. A scheduler is required for sequencing the application's *virtual pages* on the platform's *physical pages*. A device with more physical pages will accommodate more of the application's virtual pages at once, improving performance. The larger device will need a new schedule, but it can reuse the same compiled page contexts, provided the two devices have compatible page definitions. Binary compatibility across a family of page-compatible devices can now be provided by encoding an application as a collection of page contexts and a graph for connecting them. A device-specific schedule can be generated at load time by a device-specific operating system, or offline in advance. Either of these choices is sufficient to enable a business model of shrink-wrapped software for reconfigurable platforms. Comparing to uniprocessors, consider that x86 software is usually distributed either as a single binary or as a collection of binaries optimized for different x86 variants (P4, PIII, etc.), yet any of those binaries will work on next year's processor.

Figure 6.1 shows a sample, streaming application—JPEG Encode—mapped to two page compatible devices of different size. The devices are shown with reconfigurable pages marked CP (*compute pages*), streaming memories marked CMB (*configurable memory block*), and a fat-tree stream network, following SCORE conventions. The application is first partitioned into pages (6.1(a)), with each named block denoting a group of up to four pages. On a small device with few pages (6.1(b)), application execution is phased using RTR, with each phase reading from and writing to intermediate stream buffers. The RTR schedule repeats indefinitely, or until the input image(s) are exhausted. On a large device with many pages (6.1(c)), application execution is fully spatial in a single context.

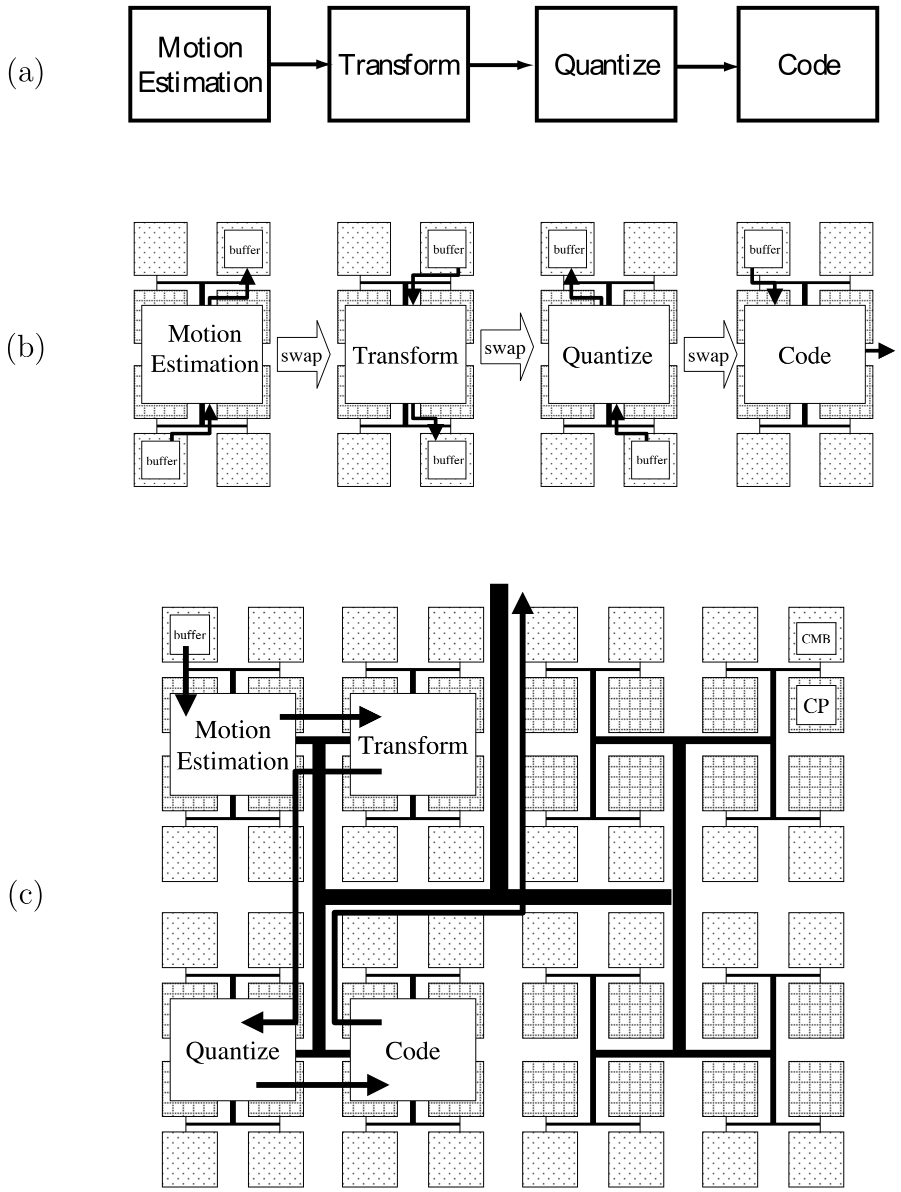


Figure 6.1: JPEG encoder on a virtual paged platform, (a) application, (b) time multiplexed on small device, (c) fully spatial on large device

Paged, virtual hardware is highly analogous to paged, virtual memory in microprocessors. In either case, a virtually large resource (memory *vs.* fabric) is time shared at a single granularity called a page. Pages are fixed size, translatable, and non-overlapping, which vastly simplifies the resource management problem of where to load new pages. Page size is retained across a family of compatible devices, and each device has a different number of pages. The application designer need not know about pages at all. The compiler need not know the number of pages, though it may need to know the page size to apply certain optimizations (data structure layout *vs.* computation layout). Only the operating system, which decides the final schedule and placement of pages, needs to know both page size and number of pages. A scheduling policy is used to decide which pages to evict and reload, and when to do that. Finally, the page size is chosen by architects to balance the overheads of page reloading (worse for small pages) and fragmentation (worse for large pages).

6.3.3 Hardware Requirements

The primary hardware requirements for adding virtualization to a paged, streaming platform are (1) sufficient memory for multiple contexts, and (2) a reconfiguration controller for RTR. The complexity of the configuration controller depends on how much scheduling needs to be done at run-time. The controller may be anything from a static, round-robin sequencer, to a full fledged microprocessor for dynamic scheduling. In addition, it is useful to have (3) deadlock detection hardware, which can inform a controller/scheduler that a running context can make no further progress and should be swapped.

6.3.3.1 Memory

A virtual paged platform needs storage for three kinds of structures: (1) contexts, (2) stream buffers, and (3) user segments. Structures associated with a non-resident context can be temporarily transferred to off-chip memory to free space. However, this lengthens the context swap time and requires a more sophisticated reconfiguration controller.

Context Store. A page context consists of a configuration and state, which must be saved during a context swap. Context must be saved for every kind of page, including reconfigurable fabric pages, streaming memories, and custom cores (MACs, *etc.*). It may be useful to separate configuration, which is static, from state, which is dynamic, and thus avoid storing duplicate configurations. However, most reconfigurable architectures use a unified context format to simplify context loading.

Stream Buffers. Each device context includes large stream buffers for streams communicating with other contexts. Those streams represent intermediate data generated by a previous context or being generated for a future context. Intermediate data that is not used in the present context, *i.e.* whose producer and consumer are not loaded, need not remain on chip.

User Segments. A device context may include memory segments defined as part of the original computation. Examples include tables of constants and scratch memory.

Unified On-Chip Memory. To simplify the architecture, all three kinds of structures can be stored in a unified kind of on-chip memory. During normal operation, the memory would use stream interfaces to implement stream buffers and segment operators. During reconfiguration, the memory would provides context load and store for pages, possibly also through stream interfaces. All these functions can be served by the streaming memories discussed in previous sections. To provide high storage capacity on chip, streaming memories can be implemented as embedded DRAM (as in HSRA [Tsu *et al.*, 1999]). The stream interfaces would naturally hide the DRAM refresh latency. Multiple, independent memories can be used to reconfigure multiple pages in parallel, reducing reconfiguration time.

Page Input Queues. An active streams on-chip may be buffered in one or more different places, including a streaming memory, the stream network (*e.g.* relay stations), and a page input queue. We associate the main queue for each stream with the stream's consumer, as a page input queue. This allows the stream contents to be saved with the page during a context swap. A page input queue can also used to drain the stream network before a context swap, in order to capture in-flight tokens as part of the context being saved. For this purpose, the queue must have an excess capacity equivalent to the maximum, round-trip latency of the stream network. This requirement is similar to reservation for stream pipelining (Section 3.7.2), but the excess capacity must be provided even if the stream uses pipelining mechanisms that do not normally require downstream reservation, such as relay stations.

6.3.3.2 Reconfiguration Control

A context swap involves a sequence of operations that must be managed by a reconfiguration controller:

- Halt all pages
- Drain data from the stream network into stream buffers
- Save page contexts to memory
- Choose the next context for each page
- Load page contexts from memory
- Load network configuration from memory
- Restart all pages

Draining the Stream Network. Draining the network means that network state need not be saved during a context swap. In-flight tokens are instead collected in page input queues, to be stored as page context. Draining the stream network can be done simply by waiting for a fixed number of clock periods corresponding to the maximum, round-trip network latency.

Saving and Loading Contexts. These steps may be non-trivial and multi-phased. If streaming memories are used for context store, then they must be temporarily configured into a context swap mode. Context swap mode resembles a segment operator in sequential read or sequential write mode. However, it must not overwrite the state of a real segment operator that was previously active. The segment operator's state, including any address and bounds registers, can first be saved into the *same memory* (so as not to involve the network), before the memory begins to swap contexts for other pages. Additional phasing is needed if a streaming memory provides context store for more than one other page, or if its contents need to be spilled to / recovered from off-chip memory.

Choosing The Next Context. This step is equivalent to scheduling, or to implementing a pre-computed schedule. The simplest case is round-robin scheduling for a cycle of contexts. This case is relatively easy to implement with a small, hardware controller, or with distributed control for each page. However, it represents a static schedule that cannot respond to any dynamics in the application's dataflow. More sophisticated, dynamic scheduling requires a correspondingly more sophisticated reconfiguration controller. In the extreme, the reconfiguration controller can be a microprocessor with scheduling software, as in SCORE. The microprocessor can be a priveleged page that also runs streaming modules during normal operation.

Partial Reconfiguration. Thus far, we have described reconfiguration as an atomic, device wide reconfiguration, implemented by halting and swapping every component. *Partial reconfiguration* would involve swapping only some components.

This is useful if a schedule calls for pages to be resident in several contiguous contexts. *Live partial reconfiguration* would involve halting and swapping some components while others continue to run. This is useful for allowing persistent pages, such as off-chip I/O interfaces, to continue to function during reconfiguration. Live partial reconfiguration requires that active pages not interfere with the swapping of other pages, nor with reconfiguration of the stream network. That may be accomplished by cutting active pages off the stream network during reconfiguration, or by supporting live, partial reconfiguration of the stream network.

6.3.3.3 Deadlock Detection

In a phased RTR implementation, a context may reach a point where it can make no forward progress, because its input buffers are empty or its output buffers are full. Keeping such a context resident is largely a waste of time. It is useful to add a hardware mechanism to detect this case. It is not sufficient to check for all empty inputs or all full outputs, since a single buffer can stall the context. In general, it is necessary to verify that every page has stalled for a period longer than the round-trip latency of the stream network. This can be implemented by emitting a stall signal from every page, computing a wired AND of those signals, and detecting a true-valued result for a certain time-out period. The stall signals can be streams, but it is appropriate to specialize them, as they are infrequent. One way to compute a page stall signal is to detect inactivity on all the page stream ports. However, this method would yield a false positive for any page that computes for a while without communicating.

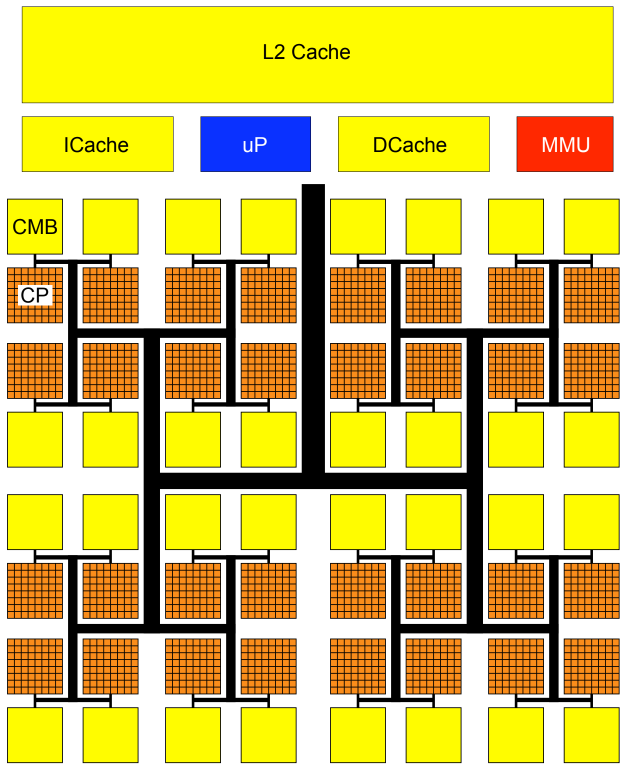


Figure 6.2: A hypothetical SCORE architecture

6.3.3.4 Example: SCORE

The concepts discussed thus far are embodied in SCORE (*Stream Computations Organized for Reconfigurable Execution*) [Caspi *et al.*, 2000a]. SCORE is a streaming compute model and associated reconfigurable architecture using virtual paging. The SCORE compute model extends TDFPN with a mechanism to dynamically create, destroy, or modify TDFPN graphs. This mechanism is embodied in *Streaming Turing Machine* (STM) graph nodes, which build graphs and communicate with that graph while it runs. STM nodes normally live on a microprocessor, where resource allocation is easier to handle than in hardware.

Figure 6.2 shows a typical SCORE architecture. A *Compute Page* (CP) corresponds to the pages discussed above, containing reconfigurable fabric, stream ports, and input queues. A *Configurable Memory Blocks* (CMB) corresponds to the streaming memories discussed above, implementing context store, large stream buffers, and segment operators. The stream network is a circuit-switched, pipelined, fat tree, network on chip. The reconfiguration controller is a full fledged microprocessor, used also for scheduling, running STMs, and boot-strapping (operating system, file I/O, *etc.*). The actual size of CPs and CMBs is an architectural parameter, though nominally, SCORE uses 512 4-LUTs per CP and 2 Mbits per CMB. The figure shows one CMB per CP, with each CMB wired to reconfigure the CP next to it. That ratio is also an architectural parameter, and it may be higher (more CMBs) to support the streaming requirements of small devices.

6.3.4 Compilation Issues

Compilation for a virtual, paged, streaming platform is divided into two major phases: page generation, and page scheduling. Page generation involves partitioning the application into pages and compiling page configurations. Page scheduling involves assigning the pages to device contexts and compiling those contexts. Only the second phase knows the actual number of pages in the target device.

The compilation flow for a virtual paged architecture is similar to the flow for a non-virtual page architecture, but extending single-context page composition into multi-context scheduling. Some parts of scheduling may be deferred as late as runtime. Extending the flow from Section 6.2, we have:

1. *High level analysis.*
   (serialization, buffer bounding)

2. *Page generation.*
   (page partitioning, process pipelining, synthesis within pages, PAR within pages)

3. *Page scheduling.*
   (temporal partitioning, memory allocation, stream aware PAR of indivisible pages, stream pipeline balancing, buffer parameter setting, time slice sizing)

6.3.4.1 Page Generation

Page generation for a virtual paged architecture is largely the same as for a non-virtual paged architecture. The primary challenge is that page partitioning must be done without knowing the actual communication delay between pages. That delay depends on the device size, which is unknown at compile time. Device size affects context assignment and placement. In the worst case, communication delay may be indefinite for a stream that crosses a context boundary. For this reason, it is important that partitioning strive to avoid page-to-page feedback loops, and otherwise minimize communication frequency on feedback loops. Page partitioning is discussed further in Section 6.4.

6.3.4.2 Page Scheduling

Page scheduling maps a virtually large graph of stream connected pages into a set of device contexts that will be time multiplexed on one device. Scheduling involves four major components:

1. *Temporal Partitioning.*
   This step assigns pages to contexts. The assignment must respect the physical limitations of what fits in a context (number of pages, number of stream buffers) and apply a scheduling policy to minimize total execution time.

2. *Memory Allocation.*
   Given a temporal partitioning, this step allocates the stored structures for each partition into the streaming memories on chip, and decides which structures to

swap off chip. These structures include contexts, stream buffers, and segments. The allocation must respect streaming memory constraints of size and number of physical segment operators, while minimizing the cost of off-chip swapping.

3. *Place and Route.*
   Given a temporal partition, this step places the partition's indivisible pages (including streaming memories) within a context and routes them on the stream network. We assume this step includes pipeline balancing and buffer parameter setting. If network configurations are normally stored in streaming memories, then the results of this step must be overlaid into the memory images.

4. *Time Slice Sizing.*
   This step decides how long a given context will run before the next context swap, termed a *time slice* or *epoch.* The epoch is usually determined by the time required to exhaust inter-context input and output buffers.

The four components of page scheduling can be done offline, online, or in some combination. In general, we equate offline with static and online with dynamic. Online scheduling is useful for responding to the dynamic behavior of an application, which may temporarily make some pages dormant and others more active, or even change the page graph. In principle, a dynamic schedule may save time in cases where a static schedule would waste it. However, a dynamic schedule requires time online to make decisions, so it may ultimately reduce run-time performance.

Markovskiy *et al.* [Markovskiy *et al.*, 2002] [Markovsky, 2004] study a spectrum of static versus dynamic scheduling. Figure 6.3 depicts the four components of scheduling on an axis that may be cut to denote which component is static and which is dynamic. The position of the cut corresponds to various flavors of scheduling. The figure depicts a fifth component of scheduling, "timing", refering to scheduling cycle

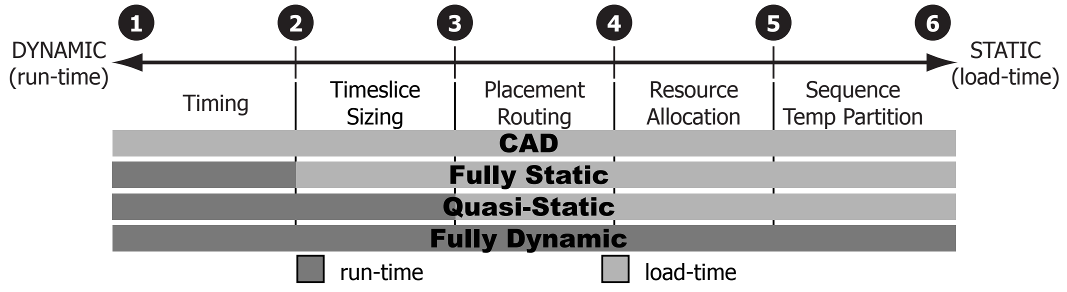


Figure 6.3: Spectrum of static *vs.* dynamic scheduling

by cycle behavior. In our streaming synthesis methodology, this level of scheduling is implicitly determined by flow control and is always dynamic. Markovskiy *et al.* go on to implement a fully static, quasi-static, and fully dynamic scheduler for SCORE. Their key result is that a static scheduler saves significant run-time overhead compared to a dynamic scheduler, and furthermore, it generates a superior schedule thanks to more comprehensive, offline graph analysis. The quasi-static scheduler improves on the static scheduler by using deadlock detection hardware and terminating a time slice immediately if all pages are stalled.

Temporal partitioning, *i.e.* assigning pages to contexts, is a particularly interesting optimization problem for virtual paged platforms. The most important criterion for this assignment is to avoid streaming feedback loops between contexts. Feedback loops generally cannot be pipelined, and consequently, cannot accumulate many tokens in stream buffers. Often, there is only one token propagating around a feedback loop. Thus, an inter-context feedback loop would make a context exhaust its input buffers and require swapping contexts almost immediately, yielding an execution dominated by context swap time. This situation is equivalent to thrashing in virtual memory, where a program accesses a working set that is larger than physical memory. In our case, the working set is the feedback loop that did not fit into one context.

A secondary criterion for temporal partitioning is to prevent page stalls due to mismatched communication rates. If a context contains pages with mismatched rates, then some pages will frequently stall waiting for slow input or output. Every such stall lengthens the total execution time. On the other hand, if a context contained only rate matched pages, then no page would stall. A context containing low throughput pages might run for a longer time slice than one containing high throughput pages, because it would take longer to exhaust its buffers. Nevertheless, with fewer stalls, total runtime would be improved. Markovskiy *et al.* [Markovskiy *et al.*, 2002] [Markovsky, 2004] show that page utilization—the inverse of stall rate—can be modeled using SDF balance equations. Balance equations are posed based on the average consumption and production rate of each page (tokens per firing) and solved to compute relative firing rates, in the form of a repetitions vector $\boldsymbol{q}$. The firing rates are normalized so that the greatest count is one, corresponding to the architectural maximum of one firing per clock period. These normalized firing rates now indicate utilization, and any rate less than one indicates stalling. The total utilization of a context is additive across all pages: $\sum_i \boldsymbol{q}_i$. Markovskiy *et al.* provide a brute force approach to find a multi-way partitioning with maximum utilization, as well as several heuristics based on graph topology. A key result is that, for the multimedia applications studied—the same as our seven application in Chapter 4 which have minimal feedback—topology based heuristics (min-cut, topological sort) yield utilizations close to the theoretical maximum.

6.3.4.3 Bufferlock

An additional component of page scheduling is bufferlock detection and recovery. Bufferlock is an artificial deadlock associated with an overflow of an undersized buffer, studied in Chapter 5. In general, bufferlock is an undecidable, and hence unpreventable artifact in dynamic rate process networks. Parks [Parks, 1995] prescribes a dynamic approach for dealing with bufferlock, simply by expanding the smallest full buffer when bufferlock occurs. While a single context platform cannot do this, a paged virtual platform with a dynamic scheduler certainly can. Buffer expansion can be accomplished most simply by chaining the full buffer with a secondary buffer, implemented in another streaming memory. However, even this seemingly simple reallocation typically requires evicting a streaming memory from the temporal partition. Thus, bufferlock recovery requires computing a new schedule online. Nevertheless, the new schedule can be retained and reused as a static schedule thereafter (until the next bufferlock), to avoid the overhead of fully dynamic scheduling.

6.3.4.4 Fault Tolerance

Online scheduling can be used to support fault tolerance by avoiding faulty pages or faulty stream routes. A fault in one page would not affect other pages or the stream network, so that page can simply be removed from the available resource pool. Similarly, a faulty route on the stream network can be avoided during online place and route of indivisible pages. The new schedule can be retained and reused as a static schedule thereafter, to avoid the overhead of fully dynamic scheduling.

6.3.4.5 Hardware Assisted Online Scheduling

The place and route component of scheduling may be very computationally expensive, so much as to swamp execution time if it were performed online. However, this step can be accelerated using the reconfigurable platform itself. Wrighton *et al.* [Wrighton and DeHon, 2003] [Wrighton, 2003] describe hardware assisted placement using a systolic implementation of simulated annealing in FPGAs. Huang *et al.* [Huang, 2004] [Huang *et al.*, 2003] describe hardware assisted routing using a modification to the programmable interconnect of an FPGA. These approaches accelerate PAR by up to thousands of times compared to software implementations, yielding PAR times on the order of seconds. Those times are quite reasonable for online scheduling in many circumstances, certainly for the case of bufferlock recovery.

6.4 Page Partitioning

The primary challenge in compiling to a paged streaming platform is page partitioning. While it is possible for a designer to manually target every SFSM to fit in a page, that process is laborious and undermines design reuse. Instead, we desire an automatic approach for restructuring an application, converting a network of arbitrarily sized, stream connected SFSMs, into a network of fixed size, stream connected pages. This task is complicated by the fact that precise page-to-page communication delay is not known during partitioning. Communication delay depends on page placement, and worse, on device size (number of pages), which may not be known at compile time. With virtualization, communication delay also depends on the page schedule—it may be indefinitely long for a stream that crosses between contexts. Page partitioning must strive to avoid page-to-page feedback loops, since loops with long communication delay would degrade system throughput. When loops cannot be contained within a page, the partitioner must strive to only expose low frequency streams, to minimize their effect on system throughput. Thus, page partitioning for process networks is more complicated than traditional circuit partitioning. In this section, we discuss new techniques for page partitioning.

6.4.1 Page Partitioning Flow

Page partitioning can be thought of as two tasks: *decomposing* components larger than a page, and *packing* components to fill pages. If every SFSM is smaller than a page, then page packing can be posed as a clustering problem, extending the throughput aware techniques of Section 5.3 to model the impact of inter-page delay on system throughput. To avoid fragmentation (underuse of page area), we should expose as many graph components as possible. This can be done by partially decomposing SFSMs, even ones that are already smaller than a page. SFSM decomposition must be careful not to introduce critical feedback loops between pages. In general, this is difficult, because an SFSM contains control loops and data dependencies. Instead, we propose decomposing an SFSM in two phases: *pipeline extraction*, which extracts loop-free pipelines from the SFSM datapath, and *SFSM decomposition* of the remaining cyclic core with knowledge of rates. Each extracted pipeline can be further decomposed into a DAG of datapath operators, to provide additional freedom for page packing (each such operator is a small, single rate, SDF actor).

The multi-phase flow for page partitioning, illustrated in Figure 6.4, is:

1. *Pipeline Extraction* (Section 6.4.2)
2. *SFSM Decomposition* (Section 6.4.3)
3. *Page Packing* (Section 6.4.4)

We describe these phases next. We also consider an approach for simultaneous SFSM decomposition and page packing in Section 6.4.5.

6.4.2 Pipeline Extraction

Pipeline extraction is a means to partially decompose an SFSM, without introducing feedback loops, by hoisting parts of the datapath out as input or output filters. Our success with stream enabled logic pipelining (Sections 3.7.3-3.7.4 and 4.4) indicates that SFSMs often contain pipelinable datapath components connected to stream input and output. Circuit level pipelining decouples those components from the SFSM for the purpose of placement. We wish to emulate this decoupling earlier in the flow,

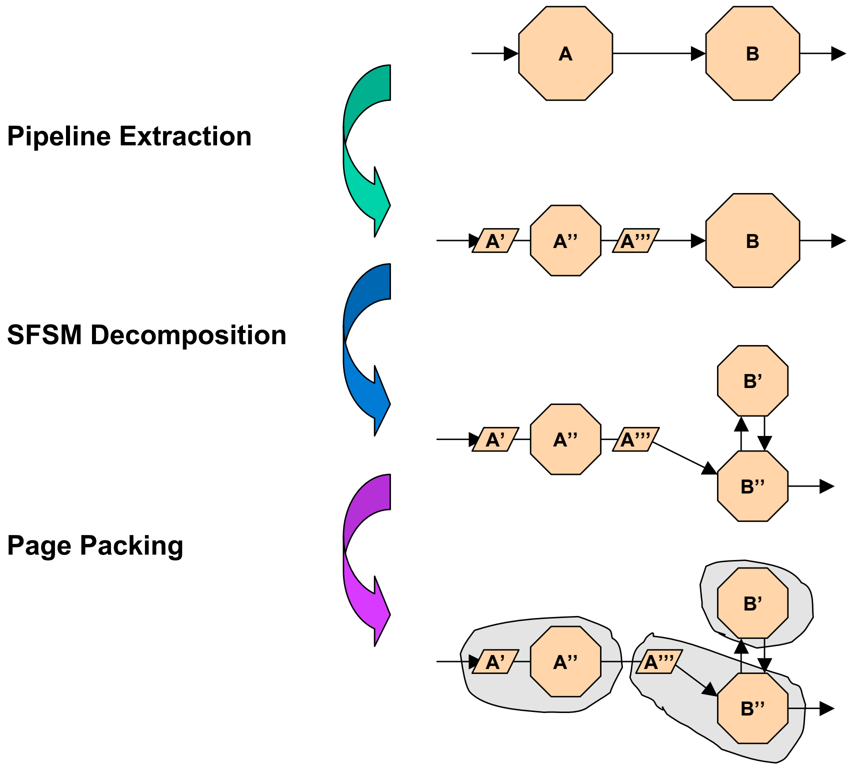


Figure 6.4: Page Partitioning Flow

for use with page partitioning. Conceptually, at the TDF source level, we wish to extract control-independent def-use chains originating at stream input operations or terminating at stream output operations.

Pipeline extraction is analogous to loop factoring. An SFSM's control flow is essentially an infinite loop containing a switch statement based on state, with each state case being a statement list that begins with stream reads. Input pipeline extraction can be posed as hoisting subexpressions without loop-carried dependencies out to an earlier loop, and modifying the stream read to receive data from that loop. Similarly, output pipeline extraction is hoisting out to a later loop. The resulting loops represent SFSMs in series connection. For example, consider an SFSM where only one state `I` reads input `i`: `state I(i): { t=i+1; ... }`. The expression `i+1` can be extracted into an input pipeline, leaving a smaller SFSM: `state I(i'): { t=i'; ... }`. An extracted subexpression may span several statements, even across to another state. An extracted subexpression may also be common to several states, but it would be chosen so that no state is needed in the extracted pipeline. An extracted pipeline is merely a DAG of datapath operations corresponding to the extracted subexpression. It can be implemented as a single-state, fully pipelineable SFSM. It can also be fully exposed to the page packer as a DAG of SDF actors, to provide more freedom and to reduce fragmentation.

Our early experiments indicate that pipeline extraction is a valuable technique. Among the seven multimedia applications from Chapter 4, roughly 30% of application area is in pure pipelines, *i.e.* single state SFSMs with no loop-carried dependencies. Among them are our largest and slowest SFSMs, such as discrete cosine transforms. Pipeline extraction would fully decompose those SFSMs. A simple implementation of pipeline extraction is also able to extract some components from nearly all SFSMs. However, several large, cyclic cores remain that need to be decomposed by secondary means in order to fit on a page.

Pipeline extraction can be used as a pipelining technique for general optimization, independent of page partitioning. It can be used in conjunction with stream enabled logic pipelining for better coverage. The two approaches, though similar, are likely to discover slightly different opportunities for pipelining, since one has visibility of source code and control flow, while the other has visibility of the circuit implementation.

6.4.3 SFSM Decomposition

SFSM decomposition seeks to partition a large SFSM into communicating, page size SFSMs with minimum inter-page communication. Traditional state machine decomposition techniques, *e.g.* for minimum logic [Devadas and Newton, 1989] or minimum I/O count [Kuo *et al.*, 1995], are generally inappropriate, because they introduce cycle-by-cycle feedback between sub-machines, and they do not model datapaths. Tight feedback would devastate performance in the presence of inter-page delay. In principle, datapaths can be modeled as boolean logic and merged into the state machine, but this precludes the use of hardware datapath resources such as carry chains and multipliers, so it is inadvisable. We need a new technique for performance oriented decomposition of a state machine with datapath (FSMD), under area and I/O constraints, in the presence of long communication delay.

6.4.3.1 State Clustering

We model SFSM decomposition as state clustering. We treat each state and its action as an indivisible unit, and we cluster those units into pages. Our intent is to contain control flow within a page whenever possible, and to minimize transfer of control between pages, since such a transfer would incur high delay. This concept is similar to Fisher's trace scheduling [Fisher, 1981], which optimizes common paths and loops of control flow (*i.e.* traces) in sequential code. It is also similar to the power-oriented FSM decomposition technique of Benini *et al.* [Benini *et al.*, 1998], which clusters states into partitions and saves power since only one partition is active at a time. In a state clustering formulation, only the partition containing the present state is active, while other partitions wait. With virtual paging, inactive partitions might even be paged out to free resources.

State clustering for minimum inter-page transitions can be formulated as an area-constrained, I/O-constrained, min-cut partitioning of the state flow graph, where each transition edge is weighted by its state transition probability. The total cut weight equals the probability of inter-page transition. Figure 6.5 shows a sample state flow graph for clustering. Each node denotes a state and its action. Each black

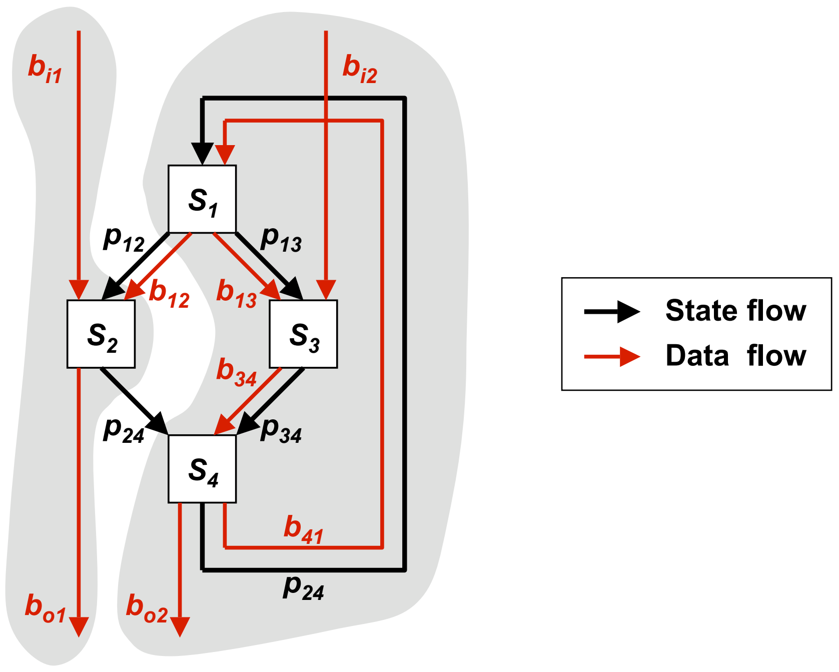


Figure 6.5: SFSM decomposition as state clustering

edge denotes a state transition, weighted by its probability p. Each red edge denotes dataflow of b bits through registers or streams. A min-cut to minimize transition probability might try to contain the loop S_1-S_3-S_4. However, the page I/O limitation might prevent this partitioning, if b_{12} and b_{24} are large enough.

6.4.3.2 Graph Conversion

We associate each cut edge with an inter-cluster stream. A cut state transition edge infers a control transfer stream. An active producer sends a token on this stream before becoming inactive, and an inactive consumer receives the token to become active. A cut dataflow edge is more complicated. Suppose dataflow edges are def-use chains. A register definition (assignment statement) generally does not know which of several uses will actually be evaluated. Sending a value token to all uses requires those uses to consume and clear the token, whether they need it or not. Otherwise, the token would remain on the stream and block future communication. This scheme requires all sub-machines to remain active to clear tokens, wasting power in a single-context execution, and wasting time in a multi-context execution. To make matters worse, a register use generally does not know which of several definitions was actually evaluated. Thus, it cannot know which tokens to clear without additional, control flow information. Thus, def-use chains cannot infer data streams.

Registers and streams constitute shared state among all sub-machines. In a sense, we need a sequential consistency model for multiple accessors of that state. There are at least two ways to reason about maintaining consistency: distributed shared memory, and closures. In a distributed, shared memory model, every register and stream have a home page, and values are transfered between homes and accessors. We can think of an uncached model, where every read/write communicates with a home page, or a cached model, where values are forwarded among accessors before returning to a home page. Arbitrary forwarding is not necessary, since sub-machines are not truly concurrent. Instead, value forwarding is better modeled as a transmission of closure (total state) between pages during a control transfer. To reduce communication, a transfer may contain only part of the closure, if the remainder will not be used until control returns. Parts of the closure may also be sent ahead, past the immediate next state, to avoid forwarding values through intermediate pages that do not actually need them. Knowing where to forward values requires control flow analysis. Having chosen a particular implementation of sequential consistency, that implementation can be modeled directly in the state clustering graph.

For example, we implement a simple, request-reply model for shared registers and streams (equivalent to uncached, distributed, shared memory). Every register and stream has a home page, and every external access requires a request-reply transaction between pages. We model request-reply transactions directly in the state flow,

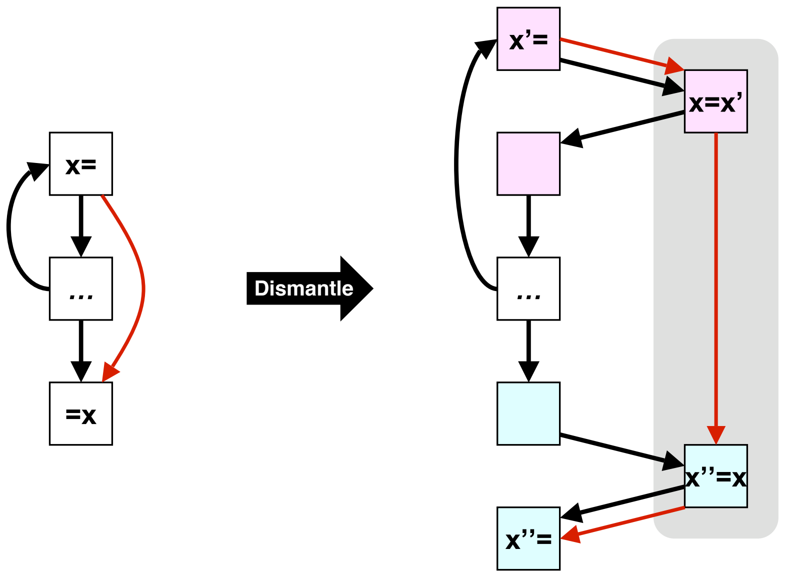


Figure 6.6: Dismantling register access to model request-reply for SFSM decomposition

as a source level transformation. We dismantle each state action into a sequence of states, with a request-reply-receive subsequence for every register/stream access. We pre-cluster all the reply nodes of a register/stream before graph partitioning to denote that object's home page. We also pre-cluster all the request and receive nodes of each original action statement. After partitioning, a request-reply-receive subsequence that crosses page boundaries will infer an inter-page transfer of control and data, while a subsequence within a page will be collapsed back into a single state. Figure 6.6 illustrates this dismantling and pre-clustering for a write and subsequent read of a register x. Note that all data edges are either within pre-clusters or between corresponding request-reply-receive nodes, so a cut data edge infers a proper stream.

Dismantling stream consumption/production requires additional care. Stream production is modeled as a register write. Stream consumption is modeled as a register write of *input history register(s)*, while a use of a stream in a state action is modeled as a read of input history register(s). Signatures with multiple inputs are dismantled into sequences of single-input states using sequential firing rule evaluation (Section 2.4.3.5). Subsets of signatures that are not sequential (*e.g.* `state foo(x):X; state foo(y):Y`) cannot be dismantled and are instead pre-clustered, so all associated input streams will have the same home page. Dismantling increases the state count by about 100x (after removing empty states), but state sequences within a cluster can be remerged after state clustering.

A state cluster is turned into a sub-machine SFSM by adding a special "wait" state. Whenever the sub-machine transfers control to another sub-machine, it transitions to its wait state to await a return of control. The wait state has a separate firing signature for every return path, each consuming a (control,data) tuple. Inter-page I/O requirements can be reduced by grouping all the control transfer streams between a pair of sub-machines and binary-encoding them to denote next state. The firing logic of the wait state may also be simplified by factoring it into several wait states, depending on which next states are expected.

6.4.3.3 Graph Partitioning Formulation

The state clustering problem can be formally stated as a multi-way graph partitioning problem. We unify state and data edges by creating one kind of edge with two weights: a dataflow edge has probability $p = 0$ and breadth b, while a state flow edge has probability p and breadth $b = 1$ for its control transfer token.

Area Constrained, I/O Constrained, Cut Optimal, K-Way Partitioning. Given a graph $G = (V, E)$ with node area $A : V \to \mathbb{R}^+$ (where $\mathbb{R}^+$ is the positive reals), edge weight $w_p : E \to \mathbb{R}^+$ (denoting probability p), edge weight $w_b : E \to \mathbb{R}^+$ (denoting data breadth b), partition area $A_{\max} > 0$, partition I/O count $B_{\max} > 0$, and partition count K, find a subset cover $\mathcal{P} \subset \wp(V)$ consisting of K disjoint subsets $P \subset V$ satisfying the area constraint: $A(P) \leq A_{\max}\ \forall P$, satisfying the I/O constraint: $w_b(P) \leq B_{\max}\ \forall P$, and minimizing total cut weight: $w_p(\mathcal{P})$. We define a partition cutset as: $\mathcal{E}(P) = \{e_{uv} \in E \ : \ (u \in P) \oplus (v \in P)\}$, the total cutset as: $\mathcal{E} = \{e_{uv} \in E \ : \ \exists P \in \mathcal{P} \text{ s.t. } (u \in P) \oplus (v \in P)\}$, and the constraint/minimization terms as: $A(P) = \sum_{v \in P} A(v)$, $w_b(P) = \sum_{e \in \mathcal{E}(P)} w_p(e)$, $w_p(P) = \sum_{e \in \mathcal{E}(P)} w_b(e)$, $w_p(\mathcal{P}) = \sum_{e \in \mathcal{E}} w_b(e) = (1/2) \sum_{P \in \mathcal{P}} w_p(P)$.

It is possible to approximate the K-way partitioning problem by iterated bipartitioning. Each iteration finds a partition P subject to a modified, balanced area constraint, $A_{\min} \le A(P) \le A_{\max}$, and removes it from V.

Area Balanced, I/O Constrained, Cut Optimal Bipartitioning. Given $G, A, w_p, w_b, A_{\min}, A_{\max}, B_{\max}$, find a partition $P \subset V$ satisfying: $A_{\min} \le A(P) \le A_{\max}$, satisfying: $w_b(P) \le B_{\max}$, and minimizing: $w_p(P)$.

The simpler problem of area-balanced bipartitioning, without optimizing a secondary weight w_p, is known to be NP-complete [Yang and Wong, 1994]. We therefore consider heuristic approaches.

6.4.3.4 Implementation and Preliminary Results

Our first attempt at state clustering tackled the iterated bipartitioning problem using an adaptation of Wong's flow-based balanced bipartitioning (FBB) [Yang and Wong, 1994]. FBB satisfies area and I/O constraints, but it does not directly minimize a secondary weight (probability). To find a partition, FBB computes a min-cut of the I/O weight. If the partition does not satisfy the area constraint, then it is expanded or shrunk by pulling one node across the cut and merging it, then computing the next best cut. The iteration continues until either the area constraint is satisfied or the I/O constraint is violated. FBB considers only one weight w, which can be the I/O weight w_b (without optimizing probabilities) or the probability weight w_p (without constraining I/O). Applying FBB to probability weights, we found that the multimedia applications from Chapter 4 can be partitioned with no more than 1%-2% of dynamic state transitions going between pages. This result is encouraging, since it would tolerate high inter-page delays with little performance loss. For example, if a transition within a page takes 1 clock period, while a transition between pages takes 10 clock periods, Amdahl's Law[2] indicates a total slowdown of only about 10%-20%. However, this result does not constrain I/O, and doing so would increase the probability cut.

We can adapt FBB to simultaneously constrain I/O and optimize probabilities as follows. First, apply FBB to probability weights w_b and iterate until either finding a valid partition or failing when no further node merging is possible (when the graph has two nodes). Upon failing, start over, apply FBB to I/O weights w_p, iterate until the I/O constraint is violated, and choose the intermediate cut with best probability cut weight. The quality of intermediate cuts depends on which node is chosen to merge across the cut, and it may be improved with multiple tries and look-ahead. It is also possible to use a mixed weight in the first phase, $w = (c)w_b + (1-c)w_p$ with $c \in [0,1]$, in the hopes of finding an I/O constrained solution earlier, with better probability cut.

DeHon [DeHon, 2002] provides an algorithm for the K-way partitioning problem. The graph nodes are first ordered in one dimension, then covered using disjoint intervals, or *extents*. The ordering step is heuristic, bringing nodes closer to create natural clusters that are tightly bound by large edge weights. The covering step then chooses extents that respect area and I/O constraints. DeHon implements a spectral ordering, which minimizes total squared distance within the one dimensional order, but with distances scaled by a mixed weight: $w = (c)w_b + (1-c)w_p$, $c \in [0,1]$. The covering step uses a dynamic programming approach to choose the set of valid extents with least cost, that cost being either the total probability cut (performance driven) or the

[2] Amdahl's Law computes the change in system performance when only part of the execution time is modified. If a fraction p of execution time T is scaled by a factor f, then the new execution time is: $T' = Tpf + T(1-p)$, and the scaling of execution time is: $T'/T = pf + (1-p) = 1 - p(1-f)$. In our case, with 2% of execution time scaled by 10x, slowdown is: $T'/T = (1 - 0.02(1-10)) = 1.18$.

total number of pages (area driven). The dynamic programming formulation computes the best midpoint for splitting each extent, considering extents in order from narrowest to widest. When splitting an extent $[s, e]$ at midpoint m (where s, e, m are start/end/midpoint indexes within the order), we have: $A[s, e] = A[s, m] + A[m, e]$ and: $w[s, e] = w[s, m] + w[m, e] - 2w([s, m] \leftrightarrow [m, e])$. Applying this partitioning approach to JPEG Encode from Chapter 4 yields total probability cuts under 2%, similar to our optimistic FBB results that lacked I/O constraints.

6.4.3.5 Area Model

SFSM decomposition requires an accurate area model $A(v)$ for every state and its action. In Chapter 4, we found that datapaths constitute roughly 94% of SFSM area[3]. Thus, it may suffice to model only datapath area. In our experiments, we estimated areas from individual datapath modules parameterized by bit width (n-bit adders, n-bit comparators, *etc.*). However, that estimate did not account for resource sharing between states nor for any optimizations in the Verilog back-end. In principle, datapath sharing between states can be incorporated into the graph partitioning problem by modifying the definition of partition area $A(P)$. Partition area with datapth sharing remains monotonic in the number of nodes $|P|$. Back-end optimizations can be approximated by incorporating their effect into datapath module area estimates. Rather than taking areas from separately compiled datapath modules, it may be possible to compile entire, unpartitioned SFSMs, pretend that their respective area is a superposition of datapath module areas, and solve for those module areas.

[3] From Table 4.2, datapaths constitute 58.0% of application area, while SFSMs constitute 61.7% of application area. Thus, datapaths constitute (58.0/61.7) = 94% of SFSM area.

6.4.4 Page Packing

The goal of page packing is to cluster small components into pages, minimizing throughput loss due to inter-page delay, with a secondary goal of minimizing fragmentation. We do not know the precise inter-page delay during packing, since it will depend on placement, device size, and virtual page schedule. Instead, we can assume a constant inter-page delay δ. We apply the same ideas from throughput aware placement in Section 5.3.2 and extend them to deal with partitioning. To model throughput, we unroll the process network into a representative SRSDF graph and use a cycle slack analysis. Before partitioning, we pre-cluster replicant nodes and streams to properly capture their respective area and bit width.

6.4.4.1 Graph Partitioning Formulation

The page packing problem can be formally stated as a multi-way graph partitioning problem on a representative SRSDF graph. Our graph representation is similar to a retiming graph, but we associate a node's delay with its outgoing edges, rather than with the node itself. Also, we treat all delays as pipeline delays, measured in whole clock periods. Stream buffers may be represented as additional graph nodes with non-zero area. Small buffers, intended for implementation in page fabric, would be naturally clustered into pages. Large or unbounded buffers, intended for implementation in other resources such as streaming memories, must be prevented from clustering into pages.

Area Constrained, I/O Constrained, Throughput Optimal, K-Way Partitioning. Given a graph $G = (V, E)$ with node area $A : V \to \mathbb{R}^+$ (where $\mathbb{R}^+$ is the positive reals), edge weight $w_b : E \to \mathbb{R}^+$ (denoting data breadth b), edge weight $w_r : E \to \mathbb{R}^+$ (denoting number of tokens r), edge weight $w_d : E \to \mathbb{R}^+$ (denoting pipeline delay d), inter-partition delay $\delta > 0$, partition area $A_{\max} > 0$, partition

I/O count $B_{\max} > 0$, and partition count K, find a subset cover $\mathcal{P} \subset \wp(V)$ consisting of K disjoint subsets $P \subset V$ satisfying the area constraint: $A(P) \leq A_{\max}\ \forall P$, satisfying the I/O constraint: $w_b(P) \leq B_{\max}\ \forall P$, and minimizing the partitioned token period T' (average clock periods per token). We define a partition cutset as: $\mathcal{E}(P) = \{e_{uv} \in E\ :\ (u \in P) \oplus (v \in P)\}$, the total cutset as: $\mathcal{E} = \{e_{uv} \in E\ :\ \exists P \in \mathcal{P}\ \text{s.t.}\ (u \in P) \oplus (v \in P)\}$, and the constraint/minimization terms as: $A(P) = \sum_{v \in P} A(v)$, $w_b(P) = \sum_{e \in \mathcal{E}(P)} w_p(e)$, $w'_d(e) = w_d(e) + \delta$ if $e \in \mathcal{E}$ or $w'_d(e) = w_d(e)$ otherwise, $T'_C = (\sum_{e \in C} w'_d(e))/(\sum_{e \in C} w_r(e))$ for any cycle $C \subseteq E$, $T' = \max_c T'_c$ over all cycles $C \subseteq E$.

It is possible to approximate the K-way partitioning problem by iterated bipartitioning. Each iteration finds a partition P subject to a modified, balanced area constraint, $A_{\min} \leq A(P) \leq A_{\max}$, and removes it from V. To properly compute throughput for subsequent partitions, it may be more appropriate not to remove P, but to give it zero area: $A(v \in P) = 0$, and to update the delay of inter-partition edges: $w'_d(e) = w_d(e) + \delta$.

Area Balanced, I/O Constrained, Throughput Optimal Bipartitioning. Given $G, A, w_b, w_r, w_d, \delta, A_{\min}, A_{\max}, B_{\max}$, find a partition $P \subset V$ satisfying: $A_{\min} \leq A(P) \leq A_{\max}$, satisfying: $w_b(P) \leq B_{\max}$, and minimizing: T'.

6.4.4.2 Implementation

It is attractive to recast the page packing problem as a min-cut partitioning problem, since we have already developed min-cut algorithms for SFSM decomposition (Section 6.4.3). A possible cut weight is the clock periods "over budget" that an edge would have if cut. Consider a graph with token period T and an edge e with cycle-specific token period $T(e)$ and cycle slack $(T - T(e))$. Cutting the edge adds δ periods. If $\delta \leq (T - T(e))$, then the graph throughput remains unaffected. If $\delta > (T - T(e))$, then the cycle slack is exceeded by $X = \delta - (T - T(e))$ periods, and the graph throughput degrades. That excess is a reasonable cut weight $w_p(e) = X$ for a throughput oriented, min-cut partitioner. However, that weight is only an approximation, as it is not properly additive over a cut of multiple edges. For example, if two edges on the same cycle C are cut, the cycle's excess over budget is: $X' = 2\delta - (T - T_C)$, whereas the total cut weight is: $2X = 2\delta - 2(T - T_C)$, which is an underestimate. Similarly, if two edges on unrelated cycles C_1, C_2 are cut, The system excess over budget is: $X' = \max\{X_1, X_2\} = \delta - T + \max\{T_{C1}, T_{C2}\}$, whereas the total cut weight is: $X = (X_1 + X_2) = 2(\delta - T) + T_{C1} + T_{C2}$, which is an overestimate.

A more accurate approach for page packing might be to adapt the performance driven partitioning of Liu *et al.* [Liu *et al.*, 1997]. Their approach is circuit level, using a retiming analysis to model clock rate. However, we can generalize from clock period ϕ to token period T, as before. Their algorithm provides bipartitioning under area constraint, I/O constraint, clock period constraint, and latency constraints. It is posed as finding a vector of boolean, partition membership variables, using Lagrangian relaxation to solve a constrained minimization of inter-partition wire count. We can convert their clock period (token period) constraint into an optimization by repartitioning within a binary search for the smallest feasible period. We do not need the latency constraints. And we can address area balancing/fragmentation by adding a minimum area constraint. We can then perform iterated bipartitioning for page packing.

6.4.4.3 Pipeline Recomposition

We proposed that extracted pipelines be fully exposed as DAGs of ALU-level actors, to provide more freedom for page packing. Following page packing, the pipelines within each page must be recomposed and implemented as SFSMs. They may be pipelined for high clock rate by stream enabled logic pipelining, or by register/queue insertion before recomposition. Logic pipeline depths should be chosen before page packing, so their effect on system throughput will be considered during packing. However, clock-optimal pipeline depths should be based on synthesis, which is more accurate after page packing. A hybrid approach might choose preliminary depths before page packing (possibly based on preliminary, pre-packing synthesis), then choose final depths after page packing.

6.4.5 Unified SFSM Decomposition and Page Packing

Page partitioning would be simpler and probably more effective if performed in fewer phases. It may be possible to unify SFSM decomposition and page packing by exposing SFSM states in a fine-grained process network, then clustering those states as a consequence of page packing. We do not wish to implement states as separate processes, only to analyze them as such during page packing. State clusters would be converted back into SFSMs after partitioning. The challenge lies in devising an SFSM-to-PN translation such that: (1) inter-state communication affects system throughput in a meaningful way, (2) inter-state communication infers streams after partitioning, and (3) state clusters can be recomposed into efficient, sub-machine SFSMs.

Chapter 7

Conclusion

We conclude this work with a summary of the entire document, a discussion of open questions and future work, and a brief history of the project.

7.1 Summary

RTL design methodologies are failing to keep up with Moore's Law technology improvements and modern design challenges. The RTL paradigm of manual design with fully exposed device timing is laborious, error prone, and unable to deal effectively with increasingly dominant interconnect delay. It inherently undermines reuse by tying designs to device timing and other parameters, necessitating significant redesign at every new device generation.

We have proposed an alternative, streaming design methodology, rooted in an abstraction of communication timing. We introduced the stream, a FIFO channel with blocking read and buffering, as a timing tolerant mechanism for composing modules into systems. Streams naturally accommodate long or unpredictable interconnect delay. The asynchronous nature of streams supports a more modular design methodology, making modules compatible in more timing contexts, and admitting optimizations that change module timing behavior, such as pipelining. At the same time, the streaming discipline and connection topology expose system-level information about module dependencies and concurrency, enabling system-level analyses that drive optimizations of modules and communication. By supporting streaming system design with a language and compiler, we not only facilitate design, we also make a design retargetable across device generations, promoting reuse at the module and system levels.

At the heart of our work is a formal model and a language for stream based design. We introduced TDF Process Networks (TDFPN), a model of concurrent, *streaming finite state machines* (SFSMs) whose actions are guarded by state and input/output readiness. Like other process network models, ours is deterministic regardless of timing and scheduling. It supports dynamic dataflow rates and requires potentially unbounded buffering for streams. However, for practical implementation in hardware, we require that stream buffers be statically sized by analysis and/or programmer annotation. Our model strongly resembles Lee's dataflow process networks (DFPN) [Lee and Parks, 1995], and we proved equivalence between variants of TDFPN and DFPN. We adapted and extended the denotational semantics of DFPN for TDFPN, with allowance for state, and with care to retain determinism in the presence of multiple, enabled firing rules. We introduced TDF (*Task Description Format*), a concrete language for TDFPN, which resembles a behavioral hardware description language with built-in streams.

We have developed a synthesis methodology to compile TDF programs to a commercial FPGA. Our methodology is retargetable, using Verilog as an intermediate form and relying on commercial back-end tools to complete the device mapping. For efficiency, we concentrate specifically on Xilinx Virtex-II Pro series FPGAs. Our synthesis methodology includes automatic generation of streams, stream buffers, and stream flow control for SFSMs. We developed several kinds of stream buffers to serve different purposes, including (1) an enabled register for registering outputs or retiming into logic, (2) a relay station for pipelining long distance streams, (3) a shift register queue (using SRL16) for medium capacity buffers, and (4) a RAM-based queue for large capacity buffers. We also developed mechanisms to pipeline streams and SFSMs for operation at high clock rates. Our SFSM pipelining approach is limited, based on retiming registers from streams, but it is sufficient for many signal processing tasks that are dominated by feed-forward datapaths.

We evaluated our synthesis methodology by using it to compile seven multimedia applications to the Xilinx XC2VP70 FPGA. Our applications include MPEG video coders, JPEG and wavelet image coders, and an IIR filter. We found that stream flow control is small and fast, comprising 6% of application area, and never limiting performance. We found that SRL16-based stream buffers comprise a non-trivial 38% of application area, but they seldom limit performance (they are usually over 200MHz, limiting only 1/3 of the fastest SFSMs). Application area and performance is otherwise dominated by datapaths, of which about 10% are under 100MHz and require pipelining. We found that pipelining SFSMs by retiming registers from streams into datapaths is effective, providing application speedups up to 2.7x, and 1.4x-1.5x on average, for an area overhead as little as 5%. Pipelining is particularly effective when combining input and output retiming and when choosing separate depths for every SFSM/stream. We found that pipelining streams to span long distances has limited effect on an FPGA, apparently due to increased utilization and congestion from adding pipelining resources. In response, we suggested a mechanism to allocate interconnect registers after placement, requiring fewer resources. After removing area constraints to relieve congestion, we found that one level of interconnect pipelining provided 15% speedup using only 3-6% additional logic cells. However, our applications did not benefit from any more than one level of interconnect pipelining. This last observation suggests that existing FPGAs, at speeds of 100-200MHz, do not yet need deeply pipelined routes. Nevertheless, as interconnect delay continues to grow, future, larger FPGAs will be forced to provide more efficient hardware and software mechanisms for pipelined communication.

We proposed a comprehensive, system-level, optimization flow to support efficient mapping to FPGAs. These optimizations rely implicitly on streaming dataflow structure, so they would not be possible in RTL-based design flows. We tackled the issue of communication buffering and pipelining in several phases, providing lower bounds for correctness, and adding buffering/pipelining to best retain throughput in the presence of long interconnect delay. First, we developed an analysis of minimum, deadlock free buffer sizes, based on state space enumeration and automata composition. The analysis may fail to bound some streams, since it is answering an undecidable question, but it is structured to bound at least some streams and to request explicit bounds for the rest. We developed an analysis of token throughput for process networks, using it to identify non-critical paths that can tolerate pipelining, and critical paths that would degrade throughput if pipelined. We applied this throughput model to develop a stream-aware approach to placement, which knows that streams may be pipelined and seeks to minimize the effect of long, pipelined routes on system throughput. We also applied the throughput model to guide SFSM pipelining and area-time transformations, so as to exploit non-critical paths and avoid over-pipelining critical feedback loops. We developed a retiming-based analysis to balance pipeline depths, even after placement, so as to retain maximum throughput.

Finally, we discussed efficient platform support for streaming. Programmable platforms are invaluable to future systems development, since they obviate the need for chip development (which is expensive, complicated, and slow), and since they can promote better reuse by casting a system as software. Today's platforms, such as FPGAs, use the same RTL-based design methodology as ASICs, so their productivity and performance will not scale well into the future without new techniques such as streaming. We proposed incremental support for streaming by adding specialized resources to an FPGA, including pipelined stream interconnect, stream buffer queues, and streaming memories. We proposed more comprehensive support for streaming in a *paged* platform, which decomposes an FPGA into a two-level hierarchy: conventional inside a page, and streaming between pages. A two-level hierarchy may improve circuit area, power, and delay, and may also improve place and route times, but it introduces a new challenge of partitioning an application into stream-connected pages. We proposed a multi-phase approach for page partitioning, first decomposing large SFSMs, then composing small ones to fill pages. We also proposed a *virtual* paged platform, as in SCORE [Caspi *et al.*, 2000a], which further supports reuse and scaling by abstracting the platform's area, or number of pages. With proper run-time support, this approach would allow an application to run on a family of different device sizes, and to automatically scale to better performance on larger, next-generation devices. Virtualization is a natural extension of streaming and requires little hardware support beyond the conventional, paged platform.

7.2 Open Questions, Future Work

A new design methodology cannot be fully established and tested within the scope of one Ph.D. dissertation. Although we have established a theoretical foundation and preliminary implementation for streaming, there are many unimplemented features and unanswered questions. A substantial body of future work is suggested in Chapters 5 and 6 on optimizations and platforms. Most of the optimizations discussed therein were implemented only in rudimentary form, or not at all. The streaming platforms of Chapter 6 have not been built, save for the SCORE platform, which exists only in simulation. We leave this for future research. Here, we discuss a handful of additional topics for future work.

7.2.1 Composition Without Intermediate Queues

Our stream composition methodology (Chapter 3) forbids direct connection of SFSMs without an intermediate queue. This requirement may waste area when a stream needs no buffering capacity, and it may introduce unnecessary pipeline delay. For example, a composition of small, static rate processes often needs no pipelining or buffering. One way to avoid intermediate queues to compose the processes in question into a single SFSM. This is the prescribed method for recomposing extracted pipelines after page partitioning—an SRSDF sub-graph of ALU operations is easily clustered into a single state SFSM. It is also possible to compose stateful SFSMs into a single, product SFSM with unified control, based on multi-action automata composition from Section 5.2.6. Still, it would be simpler for tools to merely connect two SFSMs together.

The ban on direct connections is an artifact of our stream protocol and firing guards, which require every SFSM to wait for incoming flow control before deciding to fire and asserting outgoing flow control. This convention is necessary for an SFSM to synchronize the readiness of multiple streams. However, this convention also means that two SFSMs wired directly together would wait for each other and deadlock. It may be possible to modify flow control generation to allow deadlock-free, direct connection. For example, a chain of stateless, 1-in-1-out SFSMs might use: $i_b = o_b$, $o_v = i_v$. However, such modifications must be specialized based on the stream topology, and they do not work well with fanin/fanout. Direct connection with unrestricted topology requires a more sophisticated, multi-phase stream protocol. A general protocol might request data, wait for acknowledgment, then commit the transaction—similar to four-phase handshaking in asynchronous circuits. Such a protocol would incur area overhead, since it requires either more control wires or more sophisticated control on the original valid/backpressure wires. It would also forfeit the ability to pipeline streams by simple insertion of D flip-flops, as well as the ability to pipeline SFSMs by retiming those flip-flops. The resulting overheads and complexity may be worse than our original stream protocol with intermediate queues.

7.2.2 Structural Datapath Synthesis

Our synthesis methodology for FPGAs (Chapter 3) uses behavioral Verilog as an intermediate form, relying on commercial tools to complete the mapping. This approach allows us to avoid complicated datapath synthesis, to easily retarget to new devices, and to leverage state-of-the-art optimizations in commercial tools. However, it leaves us with less control over the resulting SFSM implementation, *e.g.* datapath sharing across states. Without a structural netlist, the TDF compiler cannot have an accurate area model for page partitioning nor an accurate timing model for pipelining and scheduling datapaths. Ultimately, it would be useful for the TDF compiler to do its own datapath synthesis and optimizations. An example of structural synthesis for process networks may be taken from the TinySHIM compiler [Edwards and Tardieu, 2005]. More powerful techniques for datapath synthesis may be borrowed from hardware compilers such as GarpCC [Callahan and Wawrzynek, 2000] and CASH [Venkataramani *et al.*, 2004].

7.2.3 SFSM Scheduling

A TDF SFSM resembles a manually scheduled, behavioral state machine. Nevertheless, it is fully amenable to automatic rescheduling, thanks to the timing tolerant nature of our streaming discipline. SFSM state flow is akin to basic block flow in imperative languages. Consequently, it can be analyzed and optimized by a wealth of existing compiler techniques for loop scheduling, including strength reduction, loop unrolling, software pipelining, and modulo scheduling (*e.g.* [Callahan and Wawrzynek, 2000]). These techniques are useful for improving clock rate and throughput, and for implementing time-space trade-offs to match the throughput of several SFSMs. In addition, loop factoring, loop fusion, and code hoisting are useful for creating or merging pipelines of SFSMs. We were not able to implement these techniques within the scope of this project. They are perhaps best incorporated into a higher level compiler that generates TDF.

7.2.4 Viability of Stream Buffer Bounding

Our analysis of stream buffer bounds (Section 5.2) is based on enumeration of a potentially very large state space. The viability of this approach depends on being able to keep the enumeration small, by culling or partitioning the state space (we discussed a number of techniques to do so). One of our primary techniques is to abstract away data values, so that the state space represents only named SFSM states and queue occupancy. However, this simplification fails to distinguish end-of-stream from valid data, and it fails to identify static behaviors based on constant data values (fixed loop bounds, enumeration constants for multi-way branches, *etc.*). For example, a loop with a fixed iteration count, which terminates by comparing the loop counter to a constant, will be mistaken to have a data-dependent iteration count. It remains to be seen how often such idioms appear and cause our analysis to fail to bound a stream. Some idioms can be transformed away before buffer bounds analysis, *e.g.* identifying and unrolling a fixed iteration loop into a sequence of states. It may also be useful to apply value-range analysis followed by partial evaluation.

It should be possible to extend our buffer bounds analysis to distinguish value classes or ranges by splitting every stream action and every value test action into a *set* of actions. For example, we may distinguish inputing data (`i?`) from inputing end-of-stream (`eos(i)?`). Queue automata would need to be modified to properly enqueue value classes, which requires substantially more states. For example, a depth-N, shift register queue without value classes requires $N+1$ states, but with k values classes it requires $\sum_{n=0}^{N} k^n$ states. A queue with two value classes, such as data versus end-of-stream, requires $(2^{N+1}-1)$ states. Automata compositions would grow quickly, and buffer bounds analysis would be practical only on a small partition of the system with shallow queues.

7.2.5 Optimum SFSM and Page Size

SFSM granularity, or size, determines the ratio between compute resources and stream resources. In general, we wish to favor compute resources, since anything else is overhead. Small SFSMs require relatively more resources for flow control and stream buffering, so we prefer to cluster them into large SFSMs. However, an SFSM that is too large begins to suffer from internal interconnect delays, which reduce clock rate and increase power consumption. The optimum size of an SFSM remains an open research question and would be technology dependent. However, automatic composition and decomposition would be able to reshape SFSMs to match any target granularity, providing better scaling across device generations. In fact, automatic composition/decomposition could be used in a design space exploration to find an optimum SFSM granularity. The automation required is largely the same as page partitioning (Section 6.4), though it need not obey strict limits on area and I/O.

Page granularity, including size and number of stream ports, poses a similar trade-off. Small pages require relatively more stream ports and queues, while large pages incur internal interconnect delay and may lose more area to fragmentation. The optimum page granularity remains an open research question for SCORE. Automatic page partitioning could be used to find an optimum page granularity for a given architecture and technology.

7.2.6 Exploiting Decidable Programs

Our synthesis approach conservatively treats all TDF programs as general, dynamic rate process networks. We rely on dynamic self-scheduling using hardware flow control, and we require an expensive analysis of buffer sizes that is undecidable and may fail. These costs may be alleviated by identifying and special casing programs that conform to decidable models such as SDF [Bhattacharyya *et al.*, 1996], CSDF [Bilsen *et al.*, 1996], or HDF [Girault *et al.*, 1999]. Such programs could be fully analyzed at compile time for deadlock and resource bounds using fast, decidable algorithms. Such programs would function reliably with static schedules, which admit smaller hardware implementations with less reliance on general queues and flow control. Furthermore, such programs would be easier to parallelize or restructure to match the resources of a given target platform. Decidable and static rate structures are common in signal and media processing, and they constitute large portions of our seven applications from Chapter 4. The difficulty lies in recovering such structures in the presence of unrestricted state flow. Recovering decidable behavior, or constructing an alternate language that guarantees decidable behavior, are both open research questions.

Static rate behavior may be obfuscated in TDF by sequencing. An actor where every loop of states consumes and produces the same number of tokens is SDF, even though the equivalent SDF firing has been expressed as a sequence of state firings. It should be possible to automatically collapse the state flow of such an actor into a single state having one firing rule and one action, making its SDF nature explicit. However, the context of the actor in a surrounding graph may prohibit the transformation. Specifically, merging state sequences may reorder the actor's I/O operations (since consumption is restricted to happen at TDF state entry), and that reordering may yield incorrect behavior if a reordered input/output stream pair is connected by an external feedback loop.

Dynamic rate behavior may still conform to heterochronous dataflow (HDF) and be decidable. An HDF actor has one or more states, with each state corresponding to an SDF actor. In a composition of HDF actors, a product state can be elaborated into an SDF graph and analyzed for consistency, *i.e.* balance of consumption and production, and liveness[1]. The composition remains balanced with state flow if each actor is restricted so that a transition to a next state having different I/O rates is permitted only after a complete iteration of a balanced schedule. This restriction allows HDF programs to incorporate some dynamic rates while remaining fully decidable. However, this restriction also makes HDF actors context sensitive and less modular, since their allowable transitions depend on other actors.

HDF behavior in TDF is obfuscated by several issues, which need to be transformed away. First, each TDF actor must be HDF, *i.e.* each state must have one input signature and a static number of output tokens. A multi-signature state may be transformed into a set of single-signature states if its firing rules are sequential (Section 2.4.3.5). A dynamic stream write that is guarded by an if-then-else in TDF may be transformed into an unconditional, HDF-compatible, write by moving its branch of the if-then-else to a separate state. Second, every product state of the application must elaborate into a consistent and live SDF graph. SDF behavior, specifically balance, may be obfuscated by state flow, but it can often be recovered by merging state sequences, as suggested above. Third, actor transitions must remain unrestricted, so the balanced schedule of every elaborated SDF graph must contain at most one firing of any actor that has multiple states with different I/O rates. In general, we cannot expect this to be true. However, it will be true if every firing consumes and produces

[1] The product state of an HDF composition refers to the collective state of all HDF actors. In contrast, the product state in our TDFPN buffer bounds analysis (Section 5.2) also includes the occupancy of every stream buffer. Comparing the two approaches, HDF analysis involves a smaller state space and is decidable, but it requires additional time for consistency and liveness checks at every product state.

at most one token per stream[2], which is the case when using the synthesizable timing model of Chapter 3 (before merging state sequences to expose SDF). In some cases, it is possible to circumvent HDF restrictions on state transitions by clustering actors and scheduling the clusters hierarchically, described next.

A dynamic rate actor could be "hidden" in an SDF graph by clustering it with other actors, so that the cluster (1) looks SDF to the outside, and (2) has a local, HDF schedule with at most one firing of the dynamic rate actor per iteration. The top level schedule may call for the cluster to fire multiple times, but each such firing triggers the cluster's local schedule, which does not restrict the frequency of actor state transitions. The result is a static, hierarchical schedule. Finding compatible clusters is not always possible. One approach is to grow a cluster to include primary inputs/outputs, so the cluster looks like an SDF source/sink to the outside. This is likely to be possible for the seven multimedia applications from Chapter 4, whose dynamic rate actors are de/compression stages close to primary inputs/outputs. A systematic TDF to HDF translation might begin by identifying static rate actors (SDF), then attempting to cover each dynamic rate actor by the largest, compatible cluster.

7.3 History and Relation to SCORE

The TDF language and compiler began life as part of the SCORE project. SCORE (Stream Computations Organized for Reconfigurable Execution) [Caspi *et al.*, 2000a] attempted to provide a compute model and architectural support to bring software longevity and scaling to reconfigurable architectures. Our motivation was based on the observation that reconfigurable architectures have significant performance and

[2] Consider an SDF graph where every actor consumes and produces either zero or one token per stream per firing. If the graph has a balanced schedule, then the minimum schedule will prescribe either zero or one firing of each actor.

density advantages over microprocessors, yet they lack the ability of microprocessors to retain software across device generations. With Moore's Law area expansion and disproportionately growing interconnect delay, the job of constantly recreating software and systems would only get harder. We saw the need to abstract area and timing, and the benefit of streaming as a key ingredient. We defined an architecture with virtual, paged hardware and went on to study various approaches to page scheduling. At the time, we could not obtain applications in an appropriate, process network form. So we defined a minimal application language, TDF, and began work on complete applications and automatic compilation.

SCORE took shape as a simulated, highly parameterized device, in conjunction with run-time software for instantiating, scheduling, and executing paged computation. The TDF compiler's initial targets were POSIX threads (conventional software) and simulated hardware pages, both using similar, C++ code generation. The actual fabric of a SCORE compute page remained only loosely defined as 512 4-LUTs. Initial work on hardware compilation assumed a page fabric based on the HSRA [Tsu *et al.*, 1999], a reconfigurable architecture and test chip recently developed in our research group. Area and timing estimates were based on HSRA datapath modules, *e.g.* an n-bit adder taking so many LUTs and so many nanoseconds. Those estimates were sufficient for studying page scheduling [Markovskiy *et al.*, 2002] and for our early work on manual and automatic page partitioning.

Simulation and estimates can only go so far. Our area and timing results from hardware compilation had questionable value, since our fabric definition was incomplete, and our compiler lacked optimizations. Furthermore, the process technology on which our estimates were based (HSRA in 0.4μm) was quickly becoming outdated. We needed a more realistic hardware target to pursue further research on page partitioning and optimum page sizing. Yet there were no plans to build a physical SCORE device.

We decided to target a hypothetical SCORE device having a commercial FPGA fabric for its pages. Individual page contexts would be compiled and evaluated using commercial FPGA tools, yielding realistic area and timing figures. In addition, we could leverage state-of-the-art optimizations in a commercial compiler by letting it perform behavioral synthesis, rather than writing our own optimizations and synthesis. Thus, we began work on a Verilog code generator and back-end targeting the Xilinx Virtex-II Pro FPGA fabric.

Two realizations soon set in. First, efficient synthesis from TDF to FPGA-based pages was large enough to fill a dissertation. The details required to make streaming efficient in hardware were not trivial, and anything less than a best effort would obfuscate the study of page partitioning. Second, studying pages in isolation would tell us nothing about inter-page interconnect and system level effects (area, throughput, *etc.*), yielding incomplete guidance for building a real SCORE architecture. We still needed a more complete hardware target to pursue further research on page partitioning and SCORE.

Emulating a SCORE device in an FPGA was an option. This had been attempted early in the SCORE project, only to realize that the firmware for emulating a stream network, configurable memory blocks, and reconfiguration control left too little space for compute pages. Thus, this avenue had been abandoned.

Ironically, we now had to put SCORE aside in order to study it further. We decided to target and study the only complete architecture available: a conventional, commercial FPGA. Mapping TDF applications to an FPGA would allow us to characterize all the costs and benefits of streaming, including stream transport and detailed area and timing. However, it would require considering only a single context execution, temporarily shelving the core concept of hardware virtualization from SCORE. On the other hand, our streaming methodology would have immediate application to FPGA based design, for facilitating large system design and overcoming interconnect delay. And our results for guiding SCORE development would be current and relevant.

The work in this document represents our effort at building the necessary scaffolding to study next generation, stream based, reconfigurable architectures. We have provided a language, applications, and a compiler targeting FPGAs. Our results demonstrate the benefit of streaming on conventional architectures and suggest ways to improve area, performance, and power with new architectural support. Our vision for future architectures remains SCORE-like, as elaborated in Chapter 6. However, we had to visit a non-SCORE architecture along the way to make our point. With this scaffolding in place, future researchers may find it easier to ask the big questions about software longevity and scaling, automatic compilation, and optimum architecture points.

In the meantime, we have attempted to provide a meaningful level of design automation for existing reconfigurable architectures, to facilitate large system design, to deal with growing interconnect delay, and to promote better modular design and reuse.

Bibliography

[Agarwal *et al.*, 2000] 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 International Symposium on Computer Architecture*, pages 248–259, 2000.

[Altera, 2005] DSP builder. `http://altera.com/products/software/products/dsp/dsp-builder.html`, 2005. Software from Altera Corp.

[ARM, 1999] ARM Limited. *AMBA Specification (Rev 2.0)*, May 1999. `http://www.arm.com/products/solutions/AMBA_Spec.html`.

[Arvind *et al.*, 2004] Arvind, Rishiyur S. Nikhil, Daniel Rosenband, and Nirav Dave. High-level synthesis: An essential ingredient for designing complex ASICs. In *Proceedings of the International Conference on Computer Aided Design (ICCAD 2004)*, San Jose, CA, November 7–11, 2004.

[Ashcroft *et al.*, 1995] E.A. Ashcroft, A.A. Faustini, R. Jagannathan, and W.W. Wadge. *Multidimensional Programming.* Oxford University Press, New York, 1995.

[Benini *et al.*, 1998] Luca Benini, Giovanni De Micheli, and Frederik Vermeulen. Finite-state machine partitioning for low power. In *Proceedings of the 1998 IEEE International Symposium on Circuits and Systems (ISCAS '98)*, May 1998.

[Betz and Rose, 1997] Vaughn Betz and Jonathan Rose. VPR: A new packing, placement, and routing tool for FPGA research. In Wayne Luk, Peter Y. K. Cheung, and Manfred Glesner, editors, *Proceedings of the International Conference on Field-Programmable Logic and Applications*, number 1304 in LNCS, pages 213–222. Springer, August 1997.

[Betz *et al.*, 1999] Vaughn Betz, Jonathan Rose, and Alexander Marquardt. *Architecture and CAD for Deep-Submicron FPGAs.* Kluwer Academic Publishers, 101 Philip Drive, Assinippi Park, Norwell, Massachusetts, 02061 USA, 1999.

[Bhattacharyya *et al.*, 1996] Shuvra S. Bhattacharyya, Praveen K. Murthy, and Edward A. Lee. *Software Synthesis from Dataflow Graphs*, chapter Synchronous Dataflow. Kluwer Academic Publishers, 1996.

[Bilsen *et al.*, 1996] Greet Bilsen, Marc Engels, Rudy Lauwereins, and Jean Peperstraete. Cyclo-static dataflow. *IEEE Transactions on Signal Processing*, 44(2):397–408, February 1996.

[Bluespec, 2005] Bluespec, Inc. *Bluespec Overview*, 2005. `http://bluespec.com/pdfs_and_docs/Bluespec_overview.pdf`.

[Bove, Jr. and Watlington, 1995] Vincent Michael Bove, Jr. and John A. Watlington. Cheops: A reconfigurable data-flow system for video processing. *IEEE Transactions on Circuits and Systems for Video Technology*, 5(2):140–149, April 1995.

[Buck, 1993] Joseph T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory using the Token Flow Model.* PhD thesis, University of California, Berkeley, 1993. ERL Technical Report 93/69.

[Buck, 1994] Joseph T. Buck. Static scheduling and code generation from dynamic dataflow graphs with integer-valued control streams. In *28th Asilomar Conference on Signals, Systems, and Computers*, November 1, 1994.

[BWRC, 2005] INSECTA. `http://bwrc.eecs.berkeley.edu/Research/Insecta/`, 2005. Software from Berkeley Wireless Research Center, U.C. Berkeley.

[Callahan and Wawrzynek, 2000] Timothy J. Callahan and John Wawrzynek. Adapting Software Pipelining for Reconfigurable Computing. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2000.

[Callahan *et al.*, 2000] Timothy J. Callahan, John R. Hauser, and John Wawrzynek. The garp architecture and C compiler. *IEEE Computer*, 33(4):62–69, April 2000.

[Carloni and Sangiovanni-Vincentelli, 2000] Luca P. Carloni and Alberto L. Sangiovanni-Vincentelli. Performance analysis and optimization of latency insensitive systems. In *Proceedings of the 37th Design Automation Conference (DAC '00)*, pages 361–367, 2000.

[Carloni *et al.*, 2001] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(9):1059–1076, September 2001.

[Carloni *et al.*, 2003] L.P. Carloni, K.L. McMillan, A. Saldanha, and A.L. Sangiovanni-Vincentelli. A methodology for correct-by-construction latency-insensitive design. In A. Kuehlmann, editor, *The Best of ICCAD—20 Years of Excellence in Computer-Aided Design*, pages 143–158. Kluwer Academic Publishers, 2003.

[Caspi *et al.*, 2000a] Eylon Caspi, Michael Chu, Randy Huang, Nicholas Weaver, Joseph Yeh, John Wawrzynek, and André DeHon. Stream Computations Organized for Reconfigurable Execution (SCORE): Introduction and Tutorial. `http://www.cs.berkeley.edu/projects/brass/documents/score_tutorial.html`, short version appears in FPL'2000 (LNCS 1896), 2000.

[Caspi *et al.*, 2000b] Eylon Caspi, Michael Chu, Randy Huang, Nicholas Weaver, Joseph Yeh, John Wawrzynek, and André DeHon. Stream computations organized for reconfigurable execution (SCORE): Extended abstract. In *Proceedings of the International Conference on Field-Programmable Logic and Applications*, LNCS, pages 605–614. Springer-Verlag, August 28–30 2000.

[Caspi, 2005] Eylon Caspi. Programming SCORE. Technical Report UCB/EECS-2005-25, EECS Department, University of California, Berkeley, December 16, 2005.

[Catapult C, 2005] Catapult C. `http://www.mentor.com/products/c-based_design/catapult_c_synthesis/`, 2005. Software from Mentor Graphics Corp.

[Cradle, 2005] Cradle CT3600 multi-processor DSP. `http://www.cradle.com/products/sil_3600_family.shtml`, 2005.

[de Alfaro and Henzinger, 2001] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 109–120, Vienna, Austria, 2001.

[de Kock *et al.*, 2000] E.A. de Kock, G. Essink, W.J.M. Smits, P. van der Wolf, J.-Y. Brunel, W.M. Kruijtzer, P. Lieverse, and K.A. Vissers. Application modeling for signal processing systems. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 402–405, 2000.

[de Kock, 1999] Erwin de Kock. *Video Signal Processor Mapping.* PhD thesis, University of Eindhoven, the Netherlands, 1999.

[DeHon, 1996] André DeHon. *Reconfigurable Architectures for General-Purpose Computing.* PhD thesis, MIT, 545 Technology Sq., Cambridge, MA 02139, September 1996.

[DeHon, 2001] André DeHon. Rent's Rule Based Switching Requirements. In *Proceedings of the System-Level Interconnect Prediction Workshop (SLIP'2001)*, pages 197–204. ACM, March 2001.

[DeHon, 2002] André DeHon. Page covering. `http://www.cs.caltech.edu/courses/cs137/2002/spring/slides/day13.ppt`, May 20, 2002. Lecture 13 of course *CS137: Electronic Design Automation*, California Institute of Technology.

[Devadas and Newton, 1989] Srinivas Devadas and A. Richard Newton. Decomposition and factorization of sequential finite state machines. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(11):1206–1217, November 1989.

[Diniz *et al.*, 2001] Pedro Diniz, Mary Hall Park, Joonseok Park, Byoungro So, and Heidi Ziegler. Bridging the gap between complation and synthesis in the DEFACTO system. In *Proceedings of the 14th Workshop on Languages and Compilers for Parallel Computing Synthesis (LCPC '01)*, October 2001.

[Edwards and Tardieu, 2005] Stephen A. Edwards and Olivier Tardieu. SHIM: a deterministic model for heterogeneous embedded systems. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 264–272, 2005.

[Eker and Janneck, 2003] Johan Eker and Jörn Janneck. CAL language report: Specification of the CAL actor language. Technical Memorandum UCB/ERL M03/48, Electronics Research Laboratory, University of California, Berkeley, December 2003.

[Eldredge and Hutchings, 1994] James G. Eldredge and Brad L. Hutchings. Density enhancement of a neural network using fpgas and run-time reconfiguration. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 180–188, April 1994.

[Feit, 1984] Sidnie Dresher Feit. A fast algorithm for the two-variable integer programming problem. *Journal of the ACM*, 31(1):99–113, 1984.

[Fisher, 1981] Joseph Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 30(7):478–490, July 1981.

[Gao *et al.*, 1992] G. R. Gao, R. Govindarajan, and P. Panangaden. Well-behaved dataflow programs for DSP computation. In *Proceedings of the International Conference on Acoustics, Speech and Signal Processing (ICASSP '92)*, volume 5, pages 561–564, March 1992.

[Gaudiot and Bic, 1991] Jean-Luc Gaudiot and Lubomir Bic, editors. *Static Scheduling of Data-Flow Programs for DSP*, chapter 18, pages 501–526. Prentice Hall, 1991. Chapter by: Edward A. Lee.

[Girault *et al.*, 1999] A. Girault, B. Lee, and E.A. Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions on Computer Aided Design*, 18(6):742–760, June 1999. Also available as Technical Memorandum UCB/ERL M97/57, University of California, Berkeley.

[Gokhale *et al.*, 2000] Maya B. Gokhale, Janice M. Stone, Jeff Arnold, and Mirek Kalinowski. Stream-oriented FPGA computing in the Streams-C high level language. In *FCCM '00: Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, 2000.

[Gokhale, 2003] Maya Gokhale. *sc2 Reference Manual*, 2003. sc2 version 1.4beta. `http://www.streams-c.lanl.gov/content/streams-c/green/downloads/sc2_1.4beta/sc2.pdf`.

[Gordon *et al.*, 2002a] Michael Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Christopher Leger, Andrew A. Lamb, Jeremy Wong, Henry Hoffman, David Z. Maze, and Saman Amarasinghe. A stream compiler for communication-exposed architectures. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[Gordon *et al.*, 2002b] 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. The raw microprocessor: A computational fabric for software circuits and general purpose programs. *IEEE Micro*, March/April 2002.

[Gschwind *et al.*, 2005] Michael Gschwind, Peter Hofstee, Brian Flachs, Martin Hopkins, Yukio Watanabe, and Takeshi Yamazaki. A novel SIMD architecture for the cell heterogeneous chip-multiprocessor. In *Hot Chips 17*, August 14–16, 2005.

[Handel-C, 2005] Handel-C. `http://celoxica.com/technology/c_design/handel-c.asp`, 2005. Software from Celoxica, Inc.

[Hilfinger, 1985] P. Hilfinger. A high-level language and silicon compiler for digital signal processing. In *Proceedings of the Custom Integrated Circuits Conference (CICC '85)*, pages 213–216, Los Alamitos, CA, 1985. IEEE Computer Society Press.

[Hoare, 1985] C. A. R. Hoare. *Communicating Sequential Processes.* International Series in Computer Science. Prentice-Hall, 1985.

[Hoe and Arvind, 2004] J.C. Hoe and Arvind. Operation-centric hardware description and synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(9):1277–1288, September 2004.

[Huang *et al.*, 2003] Randy Huang, John Wawrzynek, and André DeHon. Stochastic, Spatial Routing for Hypergraphs, Trees, and Meshes. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, pages 78–87, February 2003.

[Huang, 2004] Randy Ren-Fu Huang. *Hardware-Assisted Fast Routing for Runtime Reconfigurable Computing.* PhD thesis, University of California at Berkeley, 2004.

[Hudson *et al.*, 1998] Rhett D. Hudson, David I. Lehn, and Peter M. Athanas. A run-time reconfigurable engine for image interpolation. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 88–95, 1998.

[IBM, 1999] IBM Corporation. *The CoreConnect Bus Architecture*, September 1999. White Paper. `http://www-03.ibm.com/chips/products/coreconnect/`.

[Intel, 2005] Intel IXP network processors. `http://www.intel.com/design/network/products/npfamily/`, 2005.

[ITRS, 1999] International technology roadmap for semiconductors. `http://public.itrs.net/Files/2001ITRS/Links/1999_SIA_Roadmap`, 1999.

[ITRS, 2003] International technology roadmap for semiconductors. `http://public.itrs.net/Files/2003ITRS/Home2003.htm`, 2003.

[Jackson *et al.*, 2003] Preston A. Jackson, Brad L. Hutchings, and Justin L. Tripp. Simulation and synthesis of CSP-based interprocess communication. In *11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '03)*, 2003.

[Johnston *et al.*, 2004] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, 36(1):1–34, 2004.

[Jones *et al.*, 1995] Chris Jones, John Oswald, Brian Schoner, and John Villasenor. Issues in wireless video coding using run-time-reconfigurable FPGAs. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 85–89, April 1995.

[Jones, 2003] S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press, 2003.

[Kahle *et al.*, 2005] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM Journal of Research and Development*, 49(4/5):589–604, July/September 2005.

[Kahn and MacQueen, 1977] Gilles Kahn and David B. MacQueen. Coroutines and networks of parallel processes. In *Proceedings of the IFIP CONGRESS 77*, pages 993–998. North-Holland Publishing Company, 1977.

[Kahn, 1974] Gilles Kahn. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP CONGRESS 74*, pages 471–475. North-Holland Publishing Company, 1974.

[Kannan, 1980] Ravindran Kannan. A polynomial algorithm for the two-variable integer programming problem. *Journal of the ACM*, 27(1):118–122, 1980.

[Kaplan and Stevens, 1995] D.J. Kaplan and R.S. Stevens. Processing graph method 2.0 semantics. Technical report, Naval Research Laboratory, June 1995.

[Karp, 1978] Richard M. Karp. A characterization of the minimum cycle mean in a digraph. *Discrete Mathematics*, 23:309–311, 1978.

[Khailany *et al.*, 2001] Brucek Khailany, William J. Dally, Ujval J. Kapasi, Peter Mattson, Jinyung Namkoong, John D. Owens, Brian Towles, Andrew Chang, and Scott Rixner. Imagine: Media processing with streams. *IEEE Micro*, 21(2):35–46, March/April 2001.

[Kirk, 2005] David B. Kirk. Multiple cores, multiple pipes, multiple threads—do we have more parallelism than we can handle? In *Hot Chips 17*, August 14–16, 2005.

[Kuo *et al.*, 1995] Ming-Ter Kuo, Lung-Tien Liu, and Chung-Kuan Cheng. Finite state machine decomposition for I/O minimization. In *Proceedings of the 1995 IEEE International Symposium on Circuits and Systems (ISCAS '95)*, April 1995.

[LabVIEW, 2005] Labview. `http://www.ni.com/labview/`, 2005. Software from National Instruments Corp.

[Landman and Russo, 1971] B. S. Landman and R. L. Russo. On pin versus block relationship for partitions of logic circuits. *IEEE Transactions on Computers*, 20:1469–1479, 1971.

[Lanzerotti *et al.*, 2005] M. Y. Lanzerotti, G. Fiorenza, and R. A. Rand. Microminiature packaging and integrated circuitry: The work of E. F. Rent, with an application to on-chip interconnection requirements. *IBM Journal of Research and Development*, 49(4/5), 2005.

[Lee and Messerschmitt, 1987a] Edward A. Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, C-36(1):24–35, January 1987.

[Lee and Messerschmitt, 1987b] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.

[Lee and Neuendorffer, 2005] Edward Lee and Stephen Neuendorffer. Concurrent models of computation for embedded software. *IEE Proceedings–Computers and Digital Techniques*, 152(2):239–250, March 2005.

[Lee and Parks, 1995] Edward A. Lee and Thomas M. Parks. Dataflow process networks. In *Proceedings of the IEEE*, pages 773–801, May 1995.

[Lee and Sangiovanni-Vincentelli, 1998] Edward Lee and Alberto Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computed-Aided Design for Integrated Circuits and Systems*, 17(12):1217–1229, December 1998.

[Lee, 1997] Edward A. Lee. A denotational semantics for dataflow with firing. Technical Memorandum UCB/ERL M97/3, Electronics Research Laboratory, University of California, Berkeley, January 1997.

[Leiserson *et al.*, 1983] Charles Leiserson, Flavio Rose, and James Saxe. Optimizing synchronous circuitry by retiming. In *Third Caltech Conference On VLSI*, March 1983.

[Lines, 1995] Andrew Lines. Pipelined asynchronous circuits. Master's Thesis CS-TR-95-21, California Institute of Technology, 1995.

[Lines, 2004] Andrew Lines. Asynchronous interconnect for synchronous SoC design. *IEEE Micro*, 24(1):32–41, January/February 2004.

[Ling and Amano, 1993] X. P. Ling and H. Amano. Wasmii: a data driven computer on a virtual hardware. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 33–42, April 1993.

[Liu *et al.*, 1997] Lung-Tien Liu, Minshine Shih John Lillis, and Chung-Kuan Cheng. Data flow partitioning with clock period and latency constraints. *IEEE Transactions on Circuits and Systems–I: Fundamental Theory and Applications*, 44(3):210–220, March 1997.

[Loeffler *et al.*, 1989] C. Loeffler, A. Ligtenberg, and G. S. Moschytz. Practical fast 1-D DCT algorithms with 11 multiplications. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP '89)*, pages 988–991, 1989.

[Luk *et al.*, 1997] Wayne Luk, Nabeel Shirazi, and Peter Y. K. Cheung. Compilation tools for run-time reconfigurable designs. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 56–65, April 1997.

[Lynch and Tuttle, 1987] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 137–151, Vancouver, British Columbia, Canada, 1987.

[Markovskiy *et al.*, 2002] Yury Markovskiy, Eylon Caspi, Randy Huang, Joseph Yeh, Michael Chu, John Wawrzynek, and André DeHon. Analysis of QuasiStatic Scheduling Techniques in a Virtualized Reconfigurable Machine. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, pages 196–205, February 2002.

[Markovsky, 2004] Yury Markovsky. Quasi-Static Scheduling for SCORE. Master's thesis, University of California at Berkeley, December 2004.

[McGraw *et al.*, 1985] J.R. McGraw, S.K. Skedzielewski, S.J. Allan, R.R. Oldehoeft, J. Glauert, C. Kirkham, W. Noyce, and R. Thomas. SISAL: Streams and iteration in a single assignment language: Reference manual version 1.2,. Manual M-146, Rev. 1, Lawrence Livermore National Laboratory, Livermore, CA, March 1985.

[Mitchell, 2005] Charles W. Mitchell. AMD multi-core processors, maximizing performance in a power constrained world. In *VIA Technology Forum (VTF 2005)*, June 1–3, 2005.

[Murata, 1989] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[Najjar *et al.*, 1999] Walid A. Najjar, Edward A. Lee, and Guang R. Gao. Advances in the dataflow computational model. *Parallel Computing*, 25(13-14):1907–1929, 1999.

[OCP, 2001] OCP International Partnership. *Open Core Protocol Specification, Release 1.0*, 2001. `http://www.ocpip.org/socket/ocpspec/`.

[OSCI, 2000] Open SystemC Initiative (OSCI): Synopsys Inc., CoWare Inc., Frontier Inc. *SystemC Version 1.0 User's Guide*, 2000. `http://www.systemc.org`.

[OSCI, 2005] Open SystemC Initiative (OSCI). *Draft Standard SystemC Language Reference Manual*, April 2005. SystemC 2.1. `http://www.systemc.org`.

[Parks, 1995] Thomas M. Parks. *Bounded Scheduling of Process Networks*. UCB/ERL-95-105, University of California at Berkeley, 1995.

[Ptolemy, 2005] Ptolemy II. `http://ptolemy.eecs.berkeley.edu/ptolemyII/`, 2005. Software from U.C. Berkeley.

[Sankaralingam *et al.*, 2003] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Jaehyuk Huh, Changkyu Kim, Doug Burger, Stephen W. Keckler, and Charles R. Moore. Exploiting ILP, TLP, and DLP using polymorphism in the TRIPS architecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA '03)*, pages 422–433, June 2003.

[Scott, 1970] D. Scott. Outline of a mathematical theory of computation. In *Proceedings of the 4th annual Princeton conference on Information sciences and systems*, pages 169–176, 1970.

[Sentovich *et al.*, 1992] Ellen M. Sentovich, Kanwar Jit Singh, Luciano Lavagno an d Cho Moon, Rajeev Murgai, Alexander Saldanha, Hamid Savoj, Paul R. Stephan, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical Memorandum UCB/ERL M92/41, Electronics Research Laboratory, University of California, Berkeley, May 4, 1992. `ftp://ic.eecs.berkeley.edu/pub/Sis/SIS_paper.ps.Z`.

[Shaw *et al.*, 1996] Andrew Shaw, R. Arvind, and Paul Johnson. Performance tuning scientific codes for dataflow execution. In *PACT '96: Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques*, Washington, DC, 1996.

[Simulink, 2005] Simulink. `http://www.mathworks.com/products/simulink/`, 2005. Software from The Mathworks, Inc.

[Singh and Brown, 2001] Deshanand P. Singh and Stephen D. Brown. The case for registered routing switches in field-programmable gate arrays. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, pages 161–169, February 2001.

[Singh and Brown, 2002] D. P. Singh and S. D. Brown. Integrated retiming and placement for field programmable gate arrays. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA '02)*, Monterey, California, February 24–26, 2002.

[Snider, 2002] Greg Snider. Performance-constrained pipelining of software loops onto reconfigurable hardware. In *FPGA '02: Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, pages 177–186, 2002.

[Sonics, 2005] Siliconbackplane III. `http://www.sonicsinc.com/sonics/products/siliconbackplaneIII/`, 2005. Sonics Inc.

[SPW, 2005] SPW. `http://www.coware.com/products/spw4.php`, 2005. Software from CoWare, Inc.

[Stefanov *et al.*, 2004] Todor Stefanov, Claudiu Zissulescu, Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. System design using kahn process networks: The compaan/laura approach. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, 2004.

[Swaminathan *et al.*, 2002] Sriram Swaminathan, Russell Tessier, Dennis Goeckel, and Wayne Burleson. A dynamically reconfigurable adaptive viterbi decoder. In *FPGA '02: Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, pages 227–236, 2002.

[Sylvester and Keutzer, 1999] Dennis Sylvester and Kurt Keutzer. Rethinking deep-submicron circuit design. *IEEE Computer*, 32(11):25–33, November 1999.

[Synopsis, 2005] System studio. `http://www.synopsis.com/products/cocentric_studio/`, 2005. Software from Synopsis, Inc.

[Synplicity, 2005a] Synplify DSP. `http://www.synplicity.com/products/synplifydsp/`, 2005. Software from Synplicity, Inc.

[Synplicity, 2005b] Synplify premiere. `http://www.synplicity.com/products/synplifypremier/`, 2005. Software from Synplicity, Inc.

[Synplicity, 2005c] Synplify pro. `http://www.synplicity.com/products/synplifypro/`, 2005. Software from Synplicity, Inc.

[Teifel and Manohar, 2004] John Teifel and Rajit Manohar. Static tokens: Using dataflow to automate concurrent pipeline synthesis. In *Proceedings of the Tenth International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC 2004)*, pages 17–27, April 2004.

[Trimberger *et al.*, 1997] Steve Trimberger, Dean Carberry, Anders Johnson, and Jennifer Wong. A time-multiplexed FPGA. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 22–28, April 1997.

[Tsu *et al.*, 1999] William Tsu, Kip Macy, Atul Joshi, Randy Huang, Norman Walker, Tony Tung, Omid Rowhani, Varghese George, John Wawrzynek, and André DeHon. HSRA: High-Speed, Hierarchical Synchronous Reconfigurable Array. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, pages 125–134, February 1999.

[Turjan *et al.*, 2005] Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. Solving out-of-order communication in kahn process networks. *The Journal of VLSI Signal Processing*, 40(1):7–18, May 2005.

[Venkataramani *et al.*, 2004] Girish Venkataramani, Mihai Budiu, Tiberiu Chelcea, and Seth Copen Goldstein. C to asynchronous dataflow circuits: An end-to-end toolflow. In *Thirteenth International Workshop on Logic and Synthesis (IWLS '04)*, June 2–4, 2004.

[Wan *et al.*, 2001] Marlene Wan, Hui Zhang, Varghese George, Martin Benes, Arthur Abnous, Vandana Prabhu, and Jan Rabaey. Design methodology of a low-energy reconfigurable single-chip DSP system. *Journal of VLSI Signal Processing Systems*, 28(1-2):47–61, 2001.

[Weaver *et al.*, 2003] Nicholas Weaver, Yury Markovskiy, Yatish Patel, and John Wawrzynek. Post-placement c-slow retiming for the xilinx virtex FPGA. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, pages 185–194, 2003.

[Weaver *et al.*, 2004] Nicholas Weaver, John Hauser, and John Wawrzynek. The SFRA: A corner-turn FPGA architecture. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, March 2004.

[Williamson, 1998] Michael C. Williamson. *Synthesis of Parallel Hardware Implementations from Synchronous Dataflow Graph Specifications.* PhD thesis, University of California, Berkeley, 1998.

[Wirthlin and Hutchings, 1996] Michael J. Wirthlin and Brad L. Hutchings. Sequencing run-time reconfigured hardware with software. In *Proceedings of the International Symposium on Field Programmable Gate Arrays*, pages 122–128, February 1996.

[Wrighton and DeHon, 2003] Michael Wrighton and André DeHon. Hardware-Assisted Simulated Annealing with Application for Fast FPGA Placement. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, pages 33–42, February 2003.

[Wrighton, 2003] Michael Wrighton. A Spatial Approach to FPGA Cell Placement by Simulated Annealing. Master's thesis, California Institute of Technology, June 2003.

[Xilinx, 2003a] Xilinx, Inc. *Constraints Guide*, ISE 6.2.03i edition, 2003.

[Xilinx, 2003b] Xilinx, Inc. *Development Systems Reference Guide*, UG000 v3.5.1 edition, April 30, 2003.

[Xilinx, 2005a] System generator for DSP. `http://www.xilinx.com/ise/optional_prod/system_generator.htm`, 2005. Software from Xilinx, Inc.

[Xilinx, 2005b] Xilinx ISE 6.3i. `http://www.xilinx.com/products/design_resources/design_tool/`, 2005. Software from Xilinx, Inc.

[Xilinx, 2005c] Xilinx virtex 4 FPGAs. `http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex4/`, 2005.

[Yang and Wong, 1994] Honghua Yang and D. F. Wong. Efficient network flow based min-cut balanced partitioning. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD '94)*, pages 50–55, November 1994.

[Ye and Rose, 2005] Andy G. Ye and Jonathan Rose. Using bus-based connections to improve field-programmable gate array density for implementing datapath circuits. In *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 3–13, 2005.

[Yeh, 2005] Joseph Yeh. *Complexity-Quality Tradeoffs for Real-Time Signal Compression.* PhD thesis, University of California, Berkeley, 2005.