Leveraging Latency-Insensitive Channels to Achieve Scalable Reconfigurable Computation

by

Kermin Elliott Fleming, Jr.

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2013

Author .....
Department of Electrical Engineering and Computer Science
January 14, 2013

Certified by .....
Arvind
Professor, Electrical Engineering and Computer Science
Thesis Supervisor

Certified by .....
Joel S. Emer
Professor, Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Leslie A. Kolodziejski
Chairman, Department Committee on Graduate Students

Leveraging Latency-Insensitive Channels to Achieve Scalable Reconfigurable Computation

by

Kermin Elliott Fleming, Jr.

Submitted to the Department of Electrical Engineering and Computer Science
on January 14, 2013, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Traditionally, FPGAs have been confined to the limited role of small, low-volume ASIC replacements and as circuit emulators. However, continued Moore's law scaling has given FPGAs new life as accelerators for applications that map well to fine-grained parallel substrates. Examples of such applications include processor modelling, compression, and digital signal processing.

Although FPGAs continue to increase in size, some interesting designs still fail to fit in to a single FPGA. Many tools exist that partition RTL descriptions across FPGAs. Unfortunately, existing tools have low performance due to the inefficiency of maintaining the cycle-by-cycle behavior of RTL among discrete FPGAs. These tools are unsuitable for use in FPGA program acceleration, as the purpose of an accelerator is to make applications run faster.

This thesis presents *latency-insensitive channels*, a language-level mechanism by which programmers express points in their their design at which the cycle-by-cycle behavior of the design may be modified by the compiler. By decoupling the timing of portions of the RTL from the high-level function of the program, designs may be mapped to multiple FPGAs without suffering the performance degradation observed in existing tools. This thesis demonstrates, using a diverse set of large designs, that FPGA programs described in terms of latency-insensitive channels obtain significant gains in design feasibility, compilation time, and run-time when mapped to multiple FPGAs.

Thesis Supervisor: Arvind
Title: Professor, Electrical Engineering and Computer Science

Thesis Supervisor: Joel S. Emer
Title: Professor, Electrical Engineering and Computer Science

Acknowledgments

To my reader, thank you. May you gain as much knowledge in reading about this thesis as gained in writing it. Hopefully time will prove that the ideas contained in this thesis are both generally useful and applicable to a wide range of computational problems.

Finishing this docotoral thesis is the major, and perhaps the last, milestone of my academic career. Completing a work of this size gives the author pause to reflect on the path taken and the people encountered along it. First and foremost, I would like to thank my advisors Arvind and Joel Emer for providing me with a truly complete education. Through their guidance I have evolved from an undergraduate with limited techincal expertise to a young researcher, a transformation to which the following pages will bear witness. Graduate school is a time of techinical development and learning; however, it is also a time of personal growth. I thank Arvind and, to a lesser extent, Joel for providing me with many opportunities to travel overseas and explore the world. These opportunities have given me an international education through experience, and as a result, I have become a more complete human being.

This thesis is the culmination of a long collaboration with Joel Emer and the members of VSSAD: Michael Adler, Angshuman Parashar, and Michael Pellauer. In the many summers that we spent together at the Intel facility in Hudson, we created the LEAP FPGA operating system, which helped guide the development of critical portions of this thesis. I would like to thank Joel in particular for training my thinking towards solving some of the general problems facing reconfigurable logic.

My early days at MIT were spent working on Airblue in association with Man Cheuk Ng, Mythili Vutukuru, Sam Gross, and Hari Balakrishnan, of MIT, in association with Jamey Hicks and John Ankorn of Nokia Research Center Cambridge on the Airblue wireless research platform. I would like to especially thank Jamey, from whom I first learned how to develop hardware from a software perspective, and John who, presciently, suggested that Airblue only be an example in my thesis. Chun-chieh Lin, with whom I developed the H.264 implementation presented in this thesis, was

also influential in steering me towards thinking about latency-insensitive design as a hardware design methodology.

Last, but not least, I would like to thank my family. My parents, Ike and Debbie, and my sister Kimberly have been supportive during my long tenure at MIT. The many care packages that they sent were helpful in getting through, quite litterally, the long night of graduate school. I would also like to thank Nadan Cho, who has made the last few years brighter and happier than they would otherwise have been.

Kermin Fleming, October 17th, 2012

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Field-programmable gate arrays (FPGAs) were originally intended to provide a replacement for ASICs in small or low-volume designs. However, as FPGAs have grown in both size and capability, they have matured from their original role to become algorithmic computation platforms in their own right. Indeed, many recent academic and industrial research projects [17] [39] [9] [47] have targeted FPGAs, without the intention of producing, or even emulating an ASIC. Rather than trying to precisely emulate some circuit, the design goal for these programs is purely functional, that is, to produce the answer to a problem of interest as quickly as possible. These projects target FPGAs to take advantage of the performance benefits offered by the FPGA over general purpose processors. As Moore's law continues to offer greater numbers of transistors and general-purpose processor performance fails to scale along with it, it is likely that FPGAs will enjoy even greater penetration into the domain of general computation.

Unlike general purpose processors, which have a theoretically unlimited capacity to describe programs, FPGA programs must fit within the physical resources offered by the FPGA. As a result of this area limitation, some interesting programs may not fit into a single FPGA. If a program cannot fit onto a given FPGA, the programmer is faced with a handful of choices. The programmer may use a larger single FPGA or refine the program to reduce area, neither of which may be possible. A third possibility is to partition the program among multiple FPGAs. This option is typically

feasible from an implementation perspective, but often has serious drawbacks. Manual partitioning may obtain high performance, but requires time-consuming design effort – the programmer must modify the design such that it may be partitioned, develop inter-FPGA communications hardware, and generally take on tasks that are as or more difficult than writing the original program. Tool-based partitioning [67] [3] [54], while automatic, may suffer severe performance degradations of more than an order of magnitude. Neither of these options is attractive.

The crux of the multiple FPGA problem lies in the way that hardware systems are described: modern register-transfer languages (RTLs) do not adequately convey high-level properties of hardware programs to the compiler, precluding significant compiler assistance. For example, large RTL systems are frequently described in a latency-insensitive style [13] [17] [39], which is amenable to multiple FPGA implementation. Latency-insensitive designs are typically implemented with concrete RTL FIFOs between modules, such that the timing of data transportation in these inter-module FIFOs does not impact the functional correctness of the modules or, in aggregate, of the entire program. If a compiler could understand the high-level behavior of such a program, then multiple FPGA partitioning would be straightforward: inter-module latency-insensitive FIFOs could be stretched across FPGA boundaries while intrinsically preserving the functional correctness of the design. However, current RTLs obfuscate this high-level behavior. In general, RTL compilers cannot decide when it is safe to modify the cycle-over-cycle behavior of a program. As a result, compilers devolve to preserving the cycle-over-cycle behavior of the RTL as a correctness criterion and limit themselves to the optimization of combinational logic.

To attack the multiple FPGA problem, this thesis introduces a new language-level construct: the latency-insensitive channel. Latency-insensitive channels have operational behavior similar to FIFOs, but may have dynamically variable latency and buffering. Latency-insensitive channels thus allow the programmer to directly describe points in a program where the compiler *may* choose to alter the timing behavior of the system. In particular, the compiler may choose any physical implementation of the latency insensitive channel to meet compilation goals, including mapping a

program to multiple FPGAs.

Programs described using latency-insensitive channels have a natural decomposition into latency-insensitive modules, entities that communicate only by way of latency-insensitive channels. Once decomposed, mapping the program to multiple FPGAs is conceptually straightforward, since the implementation of all inter-module communications channels is left to the latency-insensitive module (LIM) compiler. The compiler places modules on to a set of FPGAs coupled by some physical interconnect. The compiler then synthesizes a program-specific, inter-module communications network on top of the physical FPGA interconnect. The structures of this synthesized network, including program-specific interconnect multiplexors, marshalling structures, and arbitration logic, are optimized for the latency-insensitive channels and behavior of the compiled program. The automatic synthesis of this network represents a significant design-effort savings over a manually produced inter-FPGA network: whereas the LIM compiler can automatically regenerate a network in response to changes in the user program, for example the introduction of a new inter-FPGA channel, a hand-design network would require significant re-engineering.

Latency-insensitive channels give a natural partitioning of hardware programs across multiple FPGAs. However, real designs must also make use of external resources, such as memory. Moreover, good use of FPGA platform resources is essential to high-quality multiple-FPGA implementation. Performance gains in parallel computing come from making use of more resources, including memory, to accomplish a task in parallel. The performance gain due to these increased computational resources can be significant: for some truly parallel workloads, super-linear performance gains can be obtained. When hardware programs scale to multiple FPGAs, they also gain access to more resources – ranging from look-up-tables (LUTs) in the FPGA fabric to extra external DRAM banks. These resources must be leveraged to improve the performance of multiple-FPGA implementations. For example, increasing the size of the memory caches inside the program to utilize extra fabric resources. However, traditional RTL compilers are not generally able to make such design modifications, particularly when doing so implies a perturbation in the cycle behavior of the original

RTL.

By raising the level of abstraction available to the programmer above the level of RTL and explicitly decoupling resource interfaces from program RTL, programs partitioned across multiple FPGAs can automatically make use of the extra resources available across platforms. This thesis will describe a scalable, transparent mechanism based on latency-insensitive channels which provides automatic, programmer-oblivious access to resources. As in the case of parallel software running on a multi-core processor, granting access to these additional resources can dramatically accelerate FPGA programs; as in software, super-linear speed-ups in real-world applications are possible and will be demonstrated in this thesis.

Scaling designs across multiple FPGAs has traditionally been viewed as a difficult, time- and performance-consuming task. This thesis will demonstrate not only that multiple FPGA implementations can be generated automatically, by way of minor modifications to existing hardware programs, but that these multiple FPGA implementations can be superior to single FPGA implementations across several performance metrics. Thus, the work presented in this thesis provides four potential benefits to hardware designs:

- Wall-clock runtime of the program can decrease, due to improved clock frequency and increased access to FPGA resources.
- Programs can be scaled to handle larger problem sizes, again due to increased access to resources.
- Synthesis times are reduced both by the smaller size of design partitions and the opportunity for parallel synthesis.
- Partial recompilation is available in earnest because only those FPGAs that host modified modules need to be rebuilt.

Different programs experience different combinations of these salutary effects.

1.1 Thesis Organization

The thesis commences in Chapter 2.1 with a discussion of the properties that hardware systems need to have in order to permit automatic multiple FPGA implementation. This description of a model of computation for multiple FPGAs is followed by a discussion of prior work related to the thesis in Chapter 3.

After the discussion of these preliminaries, the thesis proceeds to the exposition of the LIM compiler, an automated tool capable of mapping latency-insensitive modules onto environments consisting of multiple FPGAs. Chapter 4 describes the operation and flow of the LIM compiler. The key activity of this compiler is synthesizing a communications network between the latency-insensitive modules, including support for channels that cross FPGA boundaries. To be practically useful, the network must not only provide high-bandwidth and low-latency communication, but also it must impose a low area overhead in terms of FPGA resource usage. Chapter 5 describes an efficient router architecture for inter-FPGA communications.

Efficient, high-performance networking hardware is essential to quality multiple-FPGA synthesis. However, the *parameterization* of this hardware is equally important. The compiler plays a major role in parameterizing the inter-FPGA network hardware for each application. Chapter 6 discusses program analysis techniques and compiler optimizations that are applied during compilation to improve network throughput. This chapter will also discuss mechanisms for program instrumentation and *feedback-driven* optimization.

Multiple FPGA programs are not comprised solely of latency-insensitive modules: like software programs, they must interface to memory and I/O devices. Chapter 7 describes the abstraction of these device-specific interfaces, which do not conform to the latency-insensitive model of computation.

The thesis concludes with the evaluation of multiple-FPGA implementations of several large latency-insensitive designs. Many hardware designs can be couched in the latency-insensitive model of computation, and the designs described in these chapters demonstrate both the utility of the latency-insensitive paradigm and the prac-

ticality of the approach presented in this thesis. These designs represent whole system hardware implementations drawn from recent academic and industrial research: Chapter 8 describes Airblue, a set of libraries for implementing wireless baseband transceivers, including an implementation of 802.11g; Chapter 9 describes HASim [47], a framework for building cycle-accurate multi-processor simulators, which, on multiple FPGAs, can scale to hundreds of cores; and Chapter 10 describes H.264 a high-performance video compression scheme. These chapters may be omitted if the reader is interested only in understanding the implementation of the LIM compiler.

It is important to note, in reading about these applications, that all of them were written targeting a single FPGA, long before the conception of the compiler presented in this thesis. The multiple FPGA implementations presented in this thesis use the *same, unmodified* source as single FPGA implementations, with the exception of minor re-parameterizations where relevant. Although I helped to implement Airblue and H.264 at MIT, HAsim was primarily developed at Intel.

All three of these examples experience some benefits when implemented on multiple FPGAs. Both HAsim and Airblue scale to implement larger programs, while all three applications enjoy faster throughput for some test cases when mapped to multiple FPGAs. For Airblue, throughput improvement is super-linear, in some cases. All three examples enjoy both faster compilation and faster incremental recompilation.

Finally, the thesis describes some planned extensions to the LIM compiler and concludes in Chapter 11.

1.2 Thesis Contributions

By providing new, abstract language-level constructs that allow the programmers to express the intrinsic behavior of their hardware programs, this thesis will demonstrate not only that existing hardware programs have properties that admit of multiple FPGA partitioning, but also that by explicitly exposing these properties, high-quality multiple FPGA implementations can be produced automatically. The following list summarizes the contributions of this thesis:

- The formulation and application of a model of computation for designs with explicit latency-insensitive channels (Chapter 2.1).
- A compiler mapping designs with explicit latency-insensitive channels to multiple FPGAs (Chapter 4).
- An area-efficient, high-performance network architecture for latency insensitive channels (Chapter 5).
- Optimizations for inter-FPGA networks (Chapter 6).
- Techniques for providing automatic and scalable access to resources, such as memory, across multiple FPGAs (Chapter 7).
- Numerous large hardware designs obtained through application of the compiler (Chapters 8, 9, and 10).

Chapter 2

Model of Computation

Traditional RTL has a strong structural correspondence to the physical reality of a hardware system. Designs are described in terms of registers, wires, logic gates, memory, and clocks. Even though system behavior can be complex, the semantic model of synchronous RTL is simple: at each clock cycle, values are read from register, some combinational computation occurs, and the results are written back to register. Compilers of RTL descriptions must honor these semantics: they must be *cycle-accurate*, that is, they must guarantee a bijective mapping between the cycle-over-cycle behavior of the original RTL and whatever implementation that they produce.

Synchronous systems described using RTL synthesize well, in the sense that language-level constructs have a strong correspondence to the synthesized hardware produced by a physical compiler. Moreover, the designer has tight control over the design and tool flow, even until the final cell-level realization of the system. However, the cycle-accurate model of computation and tight design control come at a cost: they severely constrain the freedom of the compiler to alter the design. In RTL-based designs, increasing the latency of even one FIFO by one cycle can break a design, while inserting a new pipeline stage can represent a major engineering effort. In general, RTL compilers cannot infer that making these sorts of cross-cycle-timing modifications will preserve cycle-accuracy or functional correctness, even if the modifications, in fact, do preserve the functional correctness of the design. Since the compiler cannot prove that modifications will maintain cycle-accuracy, it cannot assist the program-

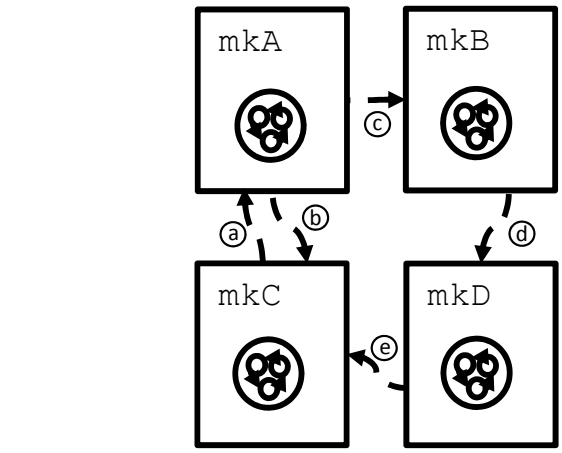
mer beyond combinational optimization.

The difficulty in automatic reasoning about when it is safe to modify the cycle-over-cycle behavior of any part of an RTL design is the direct cause of low performance in existing multiple FPGA partitioning schemes. Since existing tools cannot reason about design behavior as communicated through RTL, they must pay the enormous cost of maintaining cycle accuracy across multiple FPGAs. Typically, an order of magnitude in wall clock performance is lost, even if the design, at the high level, could be well-partitioned across FPGAs by a human.

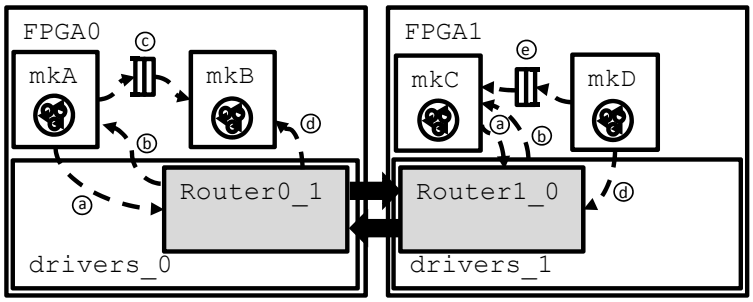
In the past, when gates were few and dear, the level of implementation control available in RTL was necessary. However, modern FPGAs offer enormous implementation capacity. As the usage of FPGAs shifts to algorithm acceleration from circuit emulation, the value in automating the design process becomes even more clear: real performance gains occur at the algorithm level. Design automation, even at the cost of slightly suboptimal performance, is essential to free the designer to focus on the high-level aspects of *program* behavior, as opposed to the low-level details of RTL.

2.1 A Model Of Computation for Multiple FPGAs

This thesis seeks to raise the level of abstraction available to hardware programmers above traditional RTL, in an effort to simplify the scaling of fine-grained parallel programs to multiple FPGAs. To achieve this goal, the thesis introduces a novel, language-level means of communicating information about design behavior: the *latency-insensitive channel*. The latency-insensitive channel has operating behavior and interface similar to the RTL FIFOs commonly used in hardware design – either a simple enqueue or dequeue operation, depending on the endpoint. However, unlike the RTL FIFO, which has fixed buffering, fixed latency, and a fully specified behavior, the latency-insensitive channel makes only two basic guarantees. First, the channel guarantees FIFO delivery of messages. Second, the channel guarantees that at least one message can be in flight at any point in time. These properties imply that the channel *may* have dynamically-variable transport latency and arbitrary, but



(a) Application



(b) Partitioned Implementation of 2-1(a)

Figure 2-1: A sample program comprised of four modules and its mapping to an environment consisting of two FPGA platforms. Latency-insensitive channels between modules located on the same platform may be connected by a simple FIFO, while latency-insensitive channels that transit platforms are communicate via a service provided by the platform runtime. The platform drivers may also provide other services, for example, memory, which can also be accessed by latency-insensitive channels.

non-zero, buffering. When a programmer instantiates a latency-insensitive channel, they assert that *these variations do not impact the functional correctness of their design.*

The latency-insensitive channel abstraction gives the compiler great freedom in choosing the physical implementation of the channel. For example, the compiler may choose to implement the channel using a fixed-buffer, fixed-latency RTL FIFO. Alternatively, the compiler is free to choose a more complex implementation, including implementations in which the endpoints of the latency-insensitive channel are physically far apart. This freedom of choice permits the compiler to automatically map designs described in terms of latency-insensitive channels to multiple FPGAs,

as shown in Figure 2-1. This thesis will use the syntax of Soft Connections [43], presented in Section 4.2.1, to denote latency-insensitive channels. Soft Connections simply designate to the compiler the endpoints of the latency-insensitive channels.

Latency-insensitive channels give great freedom to the compiler writer to transform hardware designs. However, if the use of latency-insensitive channels overly burdens the program writer, they are of little practical value. This thesis will show, via program examples, that the task of inserting latency-insensitive channels into a design does not significantly increase program complexity. Indeed, hardware designers frequently use FIFOs with latency-insensitive properties, but programming languages communicating these properties and compilers capable of leveraging these properties do not exist. Subsequent chapters will explore several large hardware programs which were originally expressed in RTL, but in a latency-insensitive style. As such, many of the hardware FIFOs in these designs were actually programmer-specified physical bindings of latency-insensitive channels. Thus, trivial textual substitution is sufficient to lift these physical FIFOs into the latency-insensitive channel abstraction, permitting the automatic mapping of these designs to multiple FPGAs. In practice, the design automation benefits obtained by using explicit latency-insensitive channels far outweigh the minimal programmer effort to explicitly demarcate latency-insensitive channels.

Once a program is expressed in terms of latency-insensitive channels, compiling that program to multiple FPGAs requires partitioning the program into *latency-insensitive modules*, modules that communicate only by way of latency-insensitive channels. In this thesis, programs are expressly constrained to be composed of latency-insensitive modules. Internally, modules may have whatever behavior the programmer sees fit, provided that the internal implementation honors the potential variability of the latency-insensitive interface. For example, once data has been obtained from an interface channel, it might pass through a traditional latency-sensitive hardware pipeline. Alternatively, although this thesis will consider only hardware latency-insensitive modules, modules could be implemented in software running on an FPGA-based soft-core, or even on an attached general-purpose processor. Module

internals do not matter; rather, it is the latency-insensitive interface that is fundamental.

The latency-insensitive channel formulation presented in this thesis is related to the burgeoning field of latency-insensitive, or elastic, design [7] [13] [40]. Work in this field, which will be discussed in detail in Section 3, can be grouped into two areas, latency insensitive design as a methodology and as a formalism for producing latency-insensitive implementations of arbitrary RTL. As a design methodology, latency insensitive design benefits programs in several ways, including lowered design times and simplified design-space exploration. This thesis treats latency-insensitive design primarily as a design methodology, but differs from previous works in that it provides explicit primitives which permit users to communicate information about latency-insensitive points in the design to the compiler.

Latency-insensitive design also exists as a formalism [6] [8] [61] for converting arbitrary latency-sensitive RTLs, which are referred to as synchronous state machines (SSMs) in the relevant literature, into latency-insensitive physical implementations. The main requirement of these transforms is that they preserve cycle-accuracy, such that the exact behavior of the original RTL implementation may be resolved. In contrast, the latency-insensitive channels formulated in this thesis are not required to preserve cycle-accuracy and may change design behavior at the cycle level to meet compilation goals, including mapping a program across multiple FPGAs. Latency-insensitive channels, as formulated in this thesis, are a programming abstraction for communication between points in a program, just as threads are a programming abstraction for parallel computation. The burden of reasoning about correctness, which has stymied previous, cycle-accurate tools, is pushed to the programmer, and the programmer must determine whether a particular point in the program can use a latency-insensitive channel safely.

2.2 Compilation and Execution of Latency-Insensitive Modules

Once a program has been formulated in terms of latency-insensitive modules, it must be mapped onto an environment for execution. An environment is an aggregation of platforms, each of which can execute latency-insensitive modules, joined together by an interconnect capable of carrying latency-insensitive channels. Depending on the underlying implementation of the particular module, it may be mapped to a variety of platforms for execution. For example, software modules can be executed on a general purpose processor or perhaps a soft-core on an FPGA. Hardware modules can be mapped to an FPGA or, if no FPGA is available or for simulation purposes, hardware modules may be executed on a general purpose CPU inside of a hardware simulator, as will be demonstrated in Chapter 7.

Platforms are subject to a handful of simple requirements, primarily related to inter-platform connectivity. Each platform in the environment must be linked by a pair of bi-direction, FIFO channels to at least one other platform. These inter-platform channels must have guaranteed, in-order delivery. An environment, then, is a strongly-connected network of platforms.

Given a program comprised of latency-insensitive modules and an execution environment, the goal of a latency-insensitive module (LIM) compiler is to map the modules of the program on to the platforms of the environment. The first step in this mapping is to place specific modules on to specific platforms. Modules are treated as indivisible by the compiler, and only whole module placements are possible. Once this placement is known, the next task of the compiler is to synthesize an inter-module communication network, that is to choose a *physical* implementation for each latency-insensitive channel in the program. A significant contribution of this thesis is the automatic generation of an area efficient, high-speed, inter-FPGA network connecting latency-insensitive modules. This thesis also presents several automatic and user-directed optimizations which improve the performance of the generated network.

2.3 Accessing Resources Across FPGAs

Real programs, both in hardware and in software, must make use of external devices. In software, dealing with physical devices, at least at a high level, can be straightforward. Software programs have general, well-defined, and highly-developed interfaces to devices, and interfaces such as sockets, virtual memory, and file systems are all relatively easy to use, to the point of being accessible to novice programmers. Moreover, these interfaces are stable across many machines, even those with radically different underlying physical hardware, permitting programs to be ported between machines, often with very little programmer intervention.

In contrast, device interfaces on the FPGA remain primitive. Programmers aiming to run programs on FPGAs must manually manufacture interfaces to raw, complex physical devices, and they must develop these interfaces for each FPGA platform they wish to use, even though the usage of physical devices is the approximately the same across many designs. These design issues represent a significant engineering effort and are only exacerbated by multiple FPGAs.

To illustrate the magnitude of the platform-interface problem, consider a design that has been fully debugged in simulation, but must be ported to an FPGA for evaluation. Porting the design to an FPGA requires bringing up various physical interfaces such as memory, clocks, and high-speed I/O and incorporating them into the existing design. These devices typically have non-trivial interfaces. Small errors, such as timing mismatches between the physical and simulation devices, are easy to make. Although trivial in scope, these errors can be extremely difficult to debug due to the limited visibility into and the lack of high-level debugging tools available for the FPGA. Without high-level debugging tools, finding errors in physical device interface logic degenerates to using an oscilloscope or, at a coarser grain, physical LEDs on the target board. These practical difficulties in FPGA use have limited the adoption of FPGAs as a general computation infrastructure.

A second issue with FPGA design, even when targeting single FPGAs, is design portability. Physical devices are inherently *platform-specific* and their interfaces are

fundamentally unportable. Moreover, if the programmer is not careful, any piece of user code touching these physical interfaces can inherit their unportability. This issue is particularly serious in latency-sensitive designs, in which circuits may well evolve, during the FPGA porting stage, to depend on particular device timings. These limitations have made it difficult to move designs from one FPGA platform to another, resulting in designs with limited re-usability and short life-spans.

Although design portability is an issue in single FPGA implementation, in multiple FPGA implementations, the problem of portability is a first-order concern. To use multiple FPGAs in any serious way, most, if not all, of a user design must be mappable to all of the FPGAs in a multiple FPGA system. However, as noted previously, any portion of the design that depends on a specific external resource, must be mapped to a specific FPGA. If enough parts of the user design are tightly coupled to specific FPGA resources, then multiple FPGAs are of limited use.

The solution to the physical interface problem and the platform portability problem is the same: latency-insensitive device abstraction. Just as in software, each FPGA platform provides a library of abstract, latency-insensitive interfaces to the physical devices available on that particular platform. Programs written using these abstract interfaces are insulated from behavioral and timing details of the underlying physical device. Both off-chip SRAM and off-chip DRAM can be captured using the same interface, and a program using this interface can alternatively use either SRAM or DRAM, depending on the platform, without a need for program modification. Because all platforms provide the same abstract interface, programs using these interfaces are portable.

In a multiple FPGA partitioning, each FPGA platform may have an instance of an external device, and a goal of a multiple FPGA compiler is to allow a program to make automatic use of these new resources across FPGAs. Abstract resource interfaces also provide a solution to the resource scaling problem. Once resource interfaces are expressed in an abstract manner, the compiler can be made aware of these resources and assist in scaling resources usage automatically, underneath of the abstraction layer.

Chapter 3

Related Work

This thesis presents a compiler that leverages explicit, language-level latency-insensitive channels to automatically produce multiple FPGA implementations. Much of the work in the thesis is novel, but it draws clear inspiration from prior research in the fields of FPGA operating systems, multiple-FPGA compilation, and latency-insensitive design.

The chapter begins with two sections discussing existing work conceptually related to the thesis: multiple FPGA compilation in Section 3.1 and latency-insensitive computing in Section 3.2. Both of these fields are relatively new, though latency-insensitive computing shares many theoretical ideas with the much earlier dataflow computing. This thesis represents a synthesis of ideas from both fields. Section 3.3 outlines the LEAP FPGA operating system, a general framework for supporting the use of external devices in single FPGA implementations. This thesis builds directly on the ideas of LEAP, extending it to support multiple FPGAs.

3.1 Existing Multiple FPGA Compilers

Due to their generality and the incumbent overhead thereof, FPGAs cannot typically emulate the computational capacity realizable on a contemporary ASIC, and several FPGAs are required to emulate larger, next-generation ASIC designs. To solve this circuit emulation problem, multiple FPGA compilers appeared shortly after the in-

vention of the FPGA, and there exist a number of commercially available tools [26] [29] capable of emulating RTL designs using multiple FPGAs. Because these tools are intended for verification, they are required to maintain the cycle accurate behavior of all signals in the original RTL design. Existing partitioning tools are differentiated by whether they provide dedicated [67] or multiplexed [3] [27] chip-to-chip wires, but both styles of multiple-FPGA emulation incur significant performance degradation due to the maintenance of cycle accuracy.

Dedicated-wire partitioning tools resemble extended versions of traditional place-and-route tools, and include inter-FPGA link delays in the circuit-level timing equations for the determination of setup and hold times. The result is that the design clocks are greatly slowed, since the delays of chip-to-chip wires are much longer than on-chip delays. However, dedicated-wire partitioners produce board-level wires that have a physical meaning, a property which may be useful in certain debugging regimes. Dedicated-wire partitioners represent a true transliteration of the original RTL: no effort is made at optimizing data transportation at FPGA boundaries. As a result, inter-chip data transportation may be extremely inefficient in terms of goodput, due to the slow clock and the transportation of unused data. Another drawback of dedicated-wire partitioning style is difficulty in timing closure: one FPGA failing to make timing can require the repartitioning and recompilation of the entire design.

Multiplexed-wire or virtual-wire [3] partitioners differ from dedicated-wire partitioners in that inter-FPGA communication is abstracted from the emulation of the RTL design. Multiplexed-wire partitioners operate by first running a single clock cycle of the partitioned design on each FPGA, then propagating the resulting values across multiplexed inter-device links. When this inter-FPGA data exchange is complete and all FPGAs are synchronized, each FPGA steps another clock cycle and the process repeats. As with dedicated-wire partitioners, multiplexed-wire partitioners incur performance overhead due to the maintenance of the cycle accuracy. This overhead is manifest in the need to freeze the emulation clock while transporting data between FPGAs and in the need to synchronize FPGAs at the end of each time step. As a result, these partitioning tools do not typically exhibit high

performance, achieving cycle-accurate operating speeds of a few megahertz [55] [59]. Despite their drawbacks, multiplexed-wire partitioners represent a significant advance over dedicated-wire partitioning tools both because they permit more wires to cross between FPGAs and because they largely avoid the timing closure problems of the dedicated wire partitioners.

TIERS [54] offers an interesting optimization for multiplexed-wire partitioners: clock-skew-based pipelining. In the basic multiplexed-wire implementation, all FPGAs operate on the same global clock cycle. However, this need not be the case: FPGAs can operate out of lock-step, provided that their inputs for a given cycle are available. In this manner, global synchronization can be obviated in favor of a distributed, data-flow style synchronization, improving overall emulation throughput. Similar pipelining effects are exploited in HAsim, described in Chapter 9.

Multiple FPGA compilation remains popular in industry, particularly in the context of emulating processors for the purpose of verification. IBM [2] has recently constructed a multiple FPGA platform for verifying a recent Bluegene processor. Intel [53] has also implemented a cycle-accurate version of its Nehalem core across a network of FPGAs for verification purposes. The IBM TwinStar system opts for full-system emulation of the original RTL, while the Intel Nehalem implementation admitted of some FPGA-friendly RTL modifications at the sub-cycle level. Both of these systems have dozens of FPGAs and run at effective speeds in the megahertz range.

This thesis presents an approach that is fundamentally different from existing tools and methodologies for multiple FPGA emulation: the compiler presented in this thesis is not required to maintain the cycle behavior or any relationship to the cycle behavior of an unpartitioned, single FPGA design. Latency-insensitive channels allow designers to explicitly annotate locations in designs at which it is safe to change cycle-by-cycle behavior of the design. As a result, partitions across these channels are free to run independently of one-another and operate on data as soon as it becomes available. Designs partitioned in this manner can take advantage of the natural pipeline parallelism inherent in hardware designs at a much finer grain than existing

optimized partitioners, such as TIERS [54]. Furthermore, the approach presented in this thesis enjoys an inter-FPGA bandwidth advantage over existing partitioners because only useful data is transported between FPGAs because only explicitly enqueued latency-insensitive channels are permitted to cross between FPGAs.

3.2 Latency-Insensitive Design

Research into latency-insensitive design is divided into two areas: latency-insensitive design as design methodology and latency-insensitive design as a formal methodology for implementing arbitrary RTL designs. The former treatment places emphasis on the development time gains derived from latency-insensitive implementations, while the former works typically seek to convert existing RTL, referred to in the literature as synchronous sequential machines (SSMs), into a latency-insensitive representation of the SSM, thereby obtaining some useful property, such as timing closure.

As a methodology [13] [17] [39], latency-insensitive design has a number of advantages over traditional RTL implementation styles. Latency-insensitive design facilitates both modular refinement and architectural exploration. Latency-insensitive modules are easier to substitute for one another because changing the timing of a module in a latency-insensitive design does not impact the functional behavior of the other modules. For this reason, latency-insensitive design also facilitates module reuse. These properties stand in contrast to typical RTL implementations in which changing the cycle-over-cycle behavior of a single component of a design, for example, by inserting a pipeline stage, can precipitate a cascade of changes through the design, constraining exploration and preventing meaningful design reuse at the fine grain. Because latency-insensitive modules are isolated from one another in terms of timing behavior, development of a latency-insensitive design can be partitioned among many design teams.

Latency-insensitive design is related to the classical Khan process networks (KPN) [33] model of distributed computation. KPNs are comprised of sequential processes connected by unbounded FIFO queues. Writes to the queues are always non-blocking,

while reads are blocking. The latency-insensitive channel model of computation differs subtlety from KPNs, in that latency-insensitive channels have non-blocking reads, but this difference has enormous theoretical impact. The implication of non-blocking reads is that latency-insensitive modules are fully general, and may have any internal implementation, hardware or software. However, in exchange for this generality, useful properties of KPNs, including determinism, are not guaranteed for latency-insensitive channel-based designs.

The circuit design literature discusses a form of latency-insensitive design, called globally asynchronous, locally synchronous (GALS) design [57]. As the scale of integration and clock frequency increases, routing clock across an entire chip becomes power-inefficient and complicates timing closure. GALS arose as a solution to the problem of clock distribution in synchronous circuits. Effectively, GALS designs are partitioned into several clock islands, which communicate by way of an asynchronous, latency-insensitive network. In some formulations, clock islands may be gated when they are idle to conserve power. Most exploration into GALS has focused on on physical, circuit-level issues as opposed to models of computation.

Latency-insensitive channels subsume GALS: one possible physical implementation of a latency-insensitive channel is a synchronizer crossing between two differently-clocked modules. The compiler presented in this thesis is capable of recognizing that the endpoints of a latency-insensitive channel reside in different clock domains and will automatically instantiate the necessary domain-crossing logic. Moreover, it should be possible to extend this functionality to automate the generation of clock islands.

Although this thesis advocates the direct description of programs in terms of latency-insensitive channels, there are other ways in which such designs may arise: formal methodologies [6] [8] [61] for transforming arbitrary RTLs (SSMs) into latency-insensitive implementations. Transformation from an SSM to a latency-insensitive implementation is achieved by programmatically introducing new control circuitry and FIFO interconnects between partitions of the original SSM in a way that maintains the cycle-accurate behavior of the original SSM.

Originally [6], latency-insensitive transform techniques were proposed to aide in

timing closure for ASIC designs. By decoupling the cycle behavior of the original SSM from the physical circuit clock, these techniques permit the introduction, for example, of extra registered buffer stations, which, in the context of SoC designs, can dramatically improve overall system timing by eliminating long wire paths.

Latency-insensitive transformation tools are also applicable to FPGA-based designs. Recently [35] and [25] have applied latency-insensitive transformation techniques to aide in the FPGA timing closure and resource optimization problems. Here, the latency-insensitive transformation permits those portions of the original design that do not map well to FPGAs, for example content-addressable-memories, to be replaced either with multiple-cycle latency-insensitive implementations that do map well to FPGAs or with equivalent software implementations. In both applications of the latency-insensitive transformation, the number of physical cycles necessary to complete an operation increases relative to the original SSM, potentially reducing performance. However, clock frequency increases in the latency-insensitive implementation may counter some of this performance degradation. Latency-insensitive transforms may also be applied to produce multiple-FPGA mappings. In this respect, modern latency-insensitive transforms may be considered as the extremely refined successors of the original multiple FPGA partitioning tools, which obtained latency-insensitive designs through coarse clock manipulation.

Carloni [6] [7] appears to be the first to attempt the automatic transformation of arbitrary SSMs to latency-insensitive implementations. This approach takes a set of wire-connected SSMs and introduces wrappers to convert the SSMs into a latency-insensitive network consisting of wrapped SSMs connected by FIFO-like relay stations. The main requirement of this transform is that the SSMs be patient, that is that they may be stalled via external signal, a form of distributed flow control. However, as noted previously in this chapter, it is difficult to automatically discover whether an SSM is patient. Should an SSM not be patient, patience can be coerced through clock-gating.

Cortadella, Kinishevsky et. al. [8] [10] and Vijayaraghavan and Arvind [61] generalize the work of Carloni by presenting full-system transformations for trans-

lating arbitrary synchronous RTLs into latency-insensitive implementations through the introduction of latency-insensitive primitives. These latency-insensitive primitives replace primitive elements, such as gates and registers, of the original SSM and are connected using a latency-insensitive communication protocol. Cortadella et. al. introduce a latency-insensitive construction based around transparent latches and circuit handshaking, while Vijayaraghavan advocates the use of a protocol based on bounded FIFOS. Cortadella, Kinishevsky et. al. further refine [11] the latency-insensitive transformation to use dataflow-style data speculation, thereby improving throughput.

Attempts have also been made at manual conversion of SSMs to latency-insensitive designs. Pellauer et al. [45] provide a set of constructs, called A-Ports, that resolve the timing of processor models to FPGA timing, with the goal of resolving the cycle-accurate behavior of the modelled processor. Programmers insert A-Ports to track the timing of specific signals and events in their modelled processor, while other signals in the model implementation remain untracked. Although A-Ports are not an automatic transformation, they offer greater potential performance since parts of the processor model which are not of interest may run independent of the A-port scheme. A-Ports themselves use the latency-insensitive channels advocated by this thesis as an internal communications mechanism.

The above transformational techniques all maintain some notion of the original cycle behavior of the base SSM. The model of computation presented in this thesis, in Chapter 2.1, expressly allows the compiler to modify the cycle behavior of the original design, disregarding the bijective-mapping requirement of existing cycle-accurate latency-insensitive transformations, in exchange for increased performance. In the context of multiple FPGAs, there are powerful technical motivations to discard this requirement: inter-FPGA links have long latencies, which can only be hidden by effective pipelining. The approach presented in this thesis can take advantage of the natural, pipelined behavior of hardware designs, even in the context of systems with high degrees of inter-FPGA feedback, while cycle-preserving transformations generally cannot.

Although the compiler presented in this thesis provides no inherent guarantees relating to cycle-accuracy, it can be readily composed with cycle-preserving transformation tools, should the cycle-accuracy of some signals be required. Indeed, one of the example codes presented in this thesis, HAsim (Chapter 9), uses the A-Ports technique. In association with these transform tools, the compiler presented in this thesis can be used to verify any synchronous design, including those designs written in a latency-sensitive style. However, there is no free lunch: as the number of cycle-accurate signals increases, multiple FPGA implementations derived from latency-insensitive channels will degrade in performance until they reach parity with traditional cycle-accurate multiple FPGA partitioning tools. Conversely, the closer a latency-insensitive channel-based multiple FPGA implementation is to pure data flow, the faster it will be.

3.3 The LEAP FPGA Operating System

This thesis leverages the LEAP [42] FPGA operating system and extends it to support multiple FPGAs. LEAP was developed to speed the implementation of FPGA-based systems and to ease the burden of migration between FPGA platforms by providing abstract, latency-insensitive channel-based device interfaces, between FPGA programs and FPGA platforms. Since all LEAP designs use the same interfaces, interfaces need be implemented only once per FPGA platform. Conversely, because physical interfaces are held constant across platforms, all LEAP programs can run on all LEAP-supported FPGA platforms. This second property of LEAP platforms is essential to the automatic implementation of latency-insensitive programs across multiple FPGAs.

In the LEAP virtualization framework, physical devices, such as memory, expose scalable, abstract, latency-insensitive request-response interfaces to the programmer. An example of a standard interface provided by LEAP is memory, which will be given a detailed treatment in Chapter 7. In LEAP, user-level designs interface to memory by means of a simple read-request, read-response, write interface [1]. Designs may

instantiate as many of these memory interfaces as the choose. At compile time, LEAP instantiates a set of caches and marshalling logic for each exposed user memory and ties these to the specific physical memory attached to the target platform.

In addition to memory, LEAP provides other generally useful device libraries and services to the FPGA programmer. The remote request-response protocol, RRR [41], is an automated mechanism for connecting FPGA programs to software running on a host PC. LEAP also provides an implementation of the Unix Standard I/O library, including an FPGA-based implementation of `printf`. LEAP also provides a single-FPGA implementation of Soft Connections [43], a syntax for describing both named, latency-insensitive point-to-point channels and named, latency-insensitive rings, which permit multiple ring stops to share a form of broadcast communication. The LIM compiler adopts Soft Connections, described in full in Section 4.2.1, as a syntax for latency-insensitive channels.

LEAP originally targeted single FPGA implementation. However, the platform abstraction advocated by LEAP can be scaled to implementations that span multiple FPGAs. Often, simple services or those with low performance requirements can be implemented directly on multiple FPGAs, without modifying the original LEAP source. On a single FPGA, LEAP device interfaces and services are typically implemented using Soft Connection point-to-point links or Soft Connection rings. Since the LIM compiler treats Soft Connections as the fundamental unit of communication between FPGAs, it can automatically scale some existing single-FPGA LEAP services to a multiple-FPGA implementation. For example, the LEAP STDIO library uses a ring to provide `printf` services to the user design. Scaling this service to more FPGAs requires only that the STDIO rings be able to span multiple FPGAs.

A good multiple-FPGA operating system will go beyond simply providing services on each FPGA. Rather, it will automatically claim and expose to the program any new resources that become available as a result of adding FPGA platforms to the system. In multi-core processors, threads mapped to different cores make use of not only the additional processor, but also additional cache resources. As in a multiprocessor, each FPGA typically has its own set of local resources, for example a memory interface. To

provide similar functionality in the context of multiple-FPGAs, performance critical services, such as memory, require a multiple-FPGA specific extension of the LEAP OS. Chapter 7 will present a mechanism by which these resources may be made available to user programs automatically, *without requiring program modification.*

Chapter 4

Multiple FPGA Compilation

Abstractly, the LIM compiler operates on the latency-insensitive modules and channels described in Chapter 2.1. However, all compilers must have a concrete syntax to operate on. The designs described in Chapters 8, 9, and 10 were written in Bluespec System Verilog [4], a commercially available high-level synthesis language. To support these designs, the LIM compiler operates on Bluespec System Verilog. The internal RTL representation of the latency-insensitive modules handled by the LIM compiler is simply Bluespec[1], while latency-insensitive channels are denoted using the Soft Connections [43] syntax, an extension to the base Bluespec syntax, which will be described in Section 4.2.1. Thus, the current implementation of the LIM compiler is best viewed as an extension of Bluespec System Verilog and its compiler, though the compilation scheme presented in this chapter generalize to all hardware descriptions augmented with latency-insensitive channels.

Implementations produced by the LIM compiler execute on top of some environment comprised of one or more FPGAs. Producing such implementations requires detailed knowledge of the FPGA environment configuration and the physical capabilities of each FPGA, for example, the availability of memory resources. Section 4.2.2 details how this information is conveyed to the compiler. Section 4.3 describes in detail the implementation of the LIM compiler and its internal algorithms. The LIM

[1]Bluespec, itself, supports a notion of modules, but these modules may have arbitrary, latency-sensitive interfaces. The latency-insensitive modules operated on by the LIM compiler have no direct syntactic representation in Bluespec.

compiler relies on Bluespec, a closed-source, commercial tool, as a subroutine. Section 4.4 describes the way in which the LIM compiler integrates Bluespec and the implementation issues that result from Bluespec's limited external interface.

4.1 Compiler Overview

The goal of the LIM compiler is to map a program comprised of latency-insensitive modules on to an environment comprised of multiple FPGAs. The chief property of latency-insensitive modules that permits this mapping is that their interfaces are comprised solely of latency-insensitive channels, the implementation of which may be chosen by the compiler without affecting the functional correctness of the original user program. Thus, communicating modules may reside on the same FPGA and communicate via a simple hardware FIFO or on two different FPGAs and communicate over some complex network interconnect.

Conceptually, the LIM compiler takes user source, described in terms of RTL, Bluespec, augmented with latency-insensitive channels, Soft Connections, as input and produces a set of programming files for a target execution environment. Because the LIM compiler operates on a commercial language, in addition to its own algorithms, the LIM compiler must make heavy use of Bluespec as a subroutine. The LIM compiler relies on two passes of the Bluespec compiler. The first pass of the Bluespec compiler produces a kind of primitive intermediate representation of user program, upon which the LIM compiler operates. The LIM compiler then analyzes this program representation and synthesizes a new Bluespec program for each FPGA platform in the target execution environment. These generated platform programs consist of a stylized version of the latency-insensitive modules mapped to that platform combined with a synthesized network which provides a physical implementation of the latency-insensitive channels of those modules. The second pass of the Bluespec compiler produces Verilog implementations of programs produced by the LIM compiler, and this Verilog is compiled by an FPGA-vendor tool chain to produce a final implementation for each FPGA.

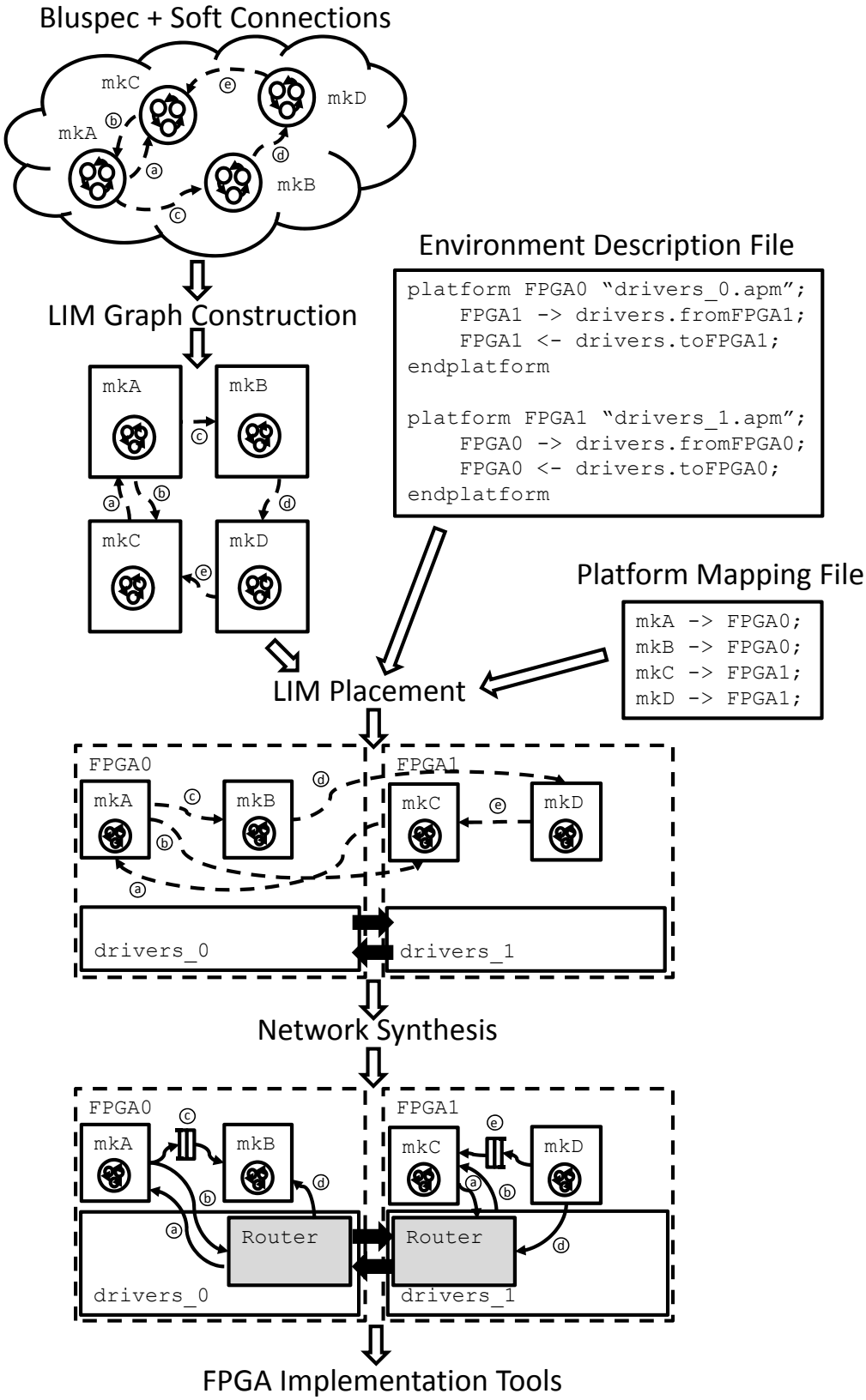


Figure 4-1: Complete LIM compiler flow, with syntax examples. Dotted lines represent logical connections, while elaborated connections are solid.

Figure 4-1 shows the flow of the LIM compiler along with its source inputs. The compiler is composed of three main phases, which occur sequentially after an initial parsing of the user program. These three phases are: the construction of graph representation of the latency-insensitive modules and channels of an input program; the mapping of that graph to a physical execution environment consisting of multiple FPGA platforms and their various physical resources; and the synthesis of a inter-module communications network for carrying data between latency-insensitive modules. At the end of these three phases, the LIM compiler produces a Bluespec program for each FPGA in the execution environment. The LIM compiler then proceeds to invoke Bluespec and the FPGA synthesis tool chain to produce a physical implementation.

Building a Latency Insensitive Module Graph

LIM compilation begins after an initial pass of the Bluespec compiler. This Bluespec invocation produces a set of log files, which can be parsed by the LIM compiler to produce a representation of the user program latency-insensitive modules and the latency-insensitive channels associated with each module. Once this data has been obtained from Bluespec, the LIM compiler builds a dataflow graph representation of the program: the LIM graph. In the LIM graph, edges correspond to the latency-insensitive channels, while vertices correspond to the synchronous RTL internals of the latency-insensitive modules. This graph will be processed by subsequent phases and eventually used to produce a physical implementation.

Mapping LIM Graph to Execution Environment

Given a LIM graph representing a program, the LIM compiler must next map this graph on to an execution environment, that is, a physical topology of FPGAs strongly connected by inter-chip communications channels. Just as the quality of cell-level placement in traditional ASIC and FPGA flows can determine performance, so too can placement of latency insensitive modules have significant impact on the performance of a multiple-FPGA implementation. In addition to considering inter-module

communications and area constraints, placement must also take into account various physical resources, such as memory, that may be available on the physical platforms. This thesis largely relies on programmer assistance to place modules on FPGAs, though automation of this process is an important area of future work.

Synthesizing Inter-Module Communication Network

Once the modules have been placed on FPGA platforms, the compiler must synthesize a communications network connecting the latency-insensitive modules. Portions of the network may be simple: individual latency-insensitive channels between modules placed on the same FPGA can be implemented as fixed-buffer FIFOs. Channels may also be mapped to a more complicated, shared network infrastructure. For example, all the channels crossing between a pair of connected FPGAs share the single, multiplexed physical interconnect between the FPGAs. Although conceptually simple, generating a high-quality, high-throughput inter-module network involves significant optimization effort on the part of the compiler. Details of these optimizations will be outlined in this chapter and elaborated in Chapter 6.

Once these three operations are completed, the LIM compiler generates a Bluespec program for each FPGA platform in the execution environment. The synthesized programs consist of a source-level representation of the latency-insensitive modules mapped to the particular platform, the physical device drivers associated with that platform, and the synthesized communications network. The generated programs are bundled with resource interface code and compiled for a second time using the Bluespec compiler, producing a Verilog implementation for each FPGA that can then be passed to a vendor-provided synthesis tools to produce an FPGA programming file. The Verilog implementation compiled by the vendor tools is fully elaborated, and all latency-insensitive channels have been mapped to RTL. Thus, to the vendor synthesis tools, the source for each FPGA platform is just a set of Verilog modules, the same as any single-FPGA design.

4.2 Inputs to the LIM Compiler

The LIM compiler requires three kinds of inputs: a Bluespec program, annotated with Soft Connections, a description of the execution environment to which that program will be mapped, and a mapping file describing the physical placement of program modules within the execution environment. The following sections describe in detail the syntax and semantics of these input types, and can be omitted without greatly impacting the subsequent discussion of the compiler implementation in Section 4.2.2.

4.2.1 Soft Connections: Describing Latency-Insensitive Channels

As noted in Chapter 2.1, the chief difficulty in automating latency-insensitive design is that latency-insensitivity is a high-level property. Designers leveraging this style in traditional RTLs typically connect modules with guarded FIFOs and predicate module execution on either the availability of ingress data or the availability of egress buffer in these FIFOs. The use of hardware FIFOs in this manner is effectively programmer-specified physical elaboration of a latency-insensitive channel.

Of course, only some FIFOs in a design will exhibit the latency insensitive property. Because a design may have many FIFOs and because the latency-insensitive property of a specific FIFO is generally undecidable, it is unlikely that automated reasoning can distinguish the latency-insensitive points in a design. In reality, not only can tools not recognize latency-insensitivity, they may not even be able to discern that FIFOs exist – the typical FIFO description in RTL is nothing more than wires and registers. The primitive, low-level nature of RTL syntax is one of the reasons that hardware compilers have traditionally been extremely limited in the kinds of optimizations that they can perform: to make any change to the design, the compiler must infer not only that a FIFO exists but also that the FIFO is semantically latency-insensitive. Therefore, for a compiler to automatically leverage latency-insensitive properties, some annotation is needed to allow the designer to convey information about latency-insensitivity to the compiler.

This thesis adopts the syntax of Soft Connections [43] to describe latency insensitive channels. Soft Connections are attractive as the basis for a latency insensitive compiler both because they offer a simple syntax and because a large library of designs, including those described in Chapters 8, 9, and 10, make use of them. Because Soft Connections themselves are latency-insensitive constructs, designs described using them can be directly partitioned among multiple FPGAs by splitting the designs across the Soft Connections.

An example of this Soft Connection syntax is shown in Figure 4-3. `mkSend` and `mkRecv` are straightforward, with the send endpoint injecting messages and with the receive draining those messages. This syntax is convenient because it provides a simple way for logically separate modules to communicate while abstracting and encapsulating the physical interconnect effecting that communication. In the context of the multiple FPGA compiler presented in this thesis, only Soft Connections channels are treated as latency-insensitive. Other FIFOs, for example those provided in the base language, retain their original fixed-implementation behavior.

In addition to point-to-point channels, Soft Connections also provide a broadcast mechanism: the ring. Modules participating in the broadcast instantiate named ring stops. At compile time, ring stops are aggregated by name and connected to one another via latency-insensitive point-to-point links. The ordering of this aggregation is not guaranteed by the compiler, and the compiler may choose an ordering to optimize other design goals, a freedom which will be used in subsequent sections of this chapter to optimize multiple FPGA implementations. Rings provide a low-cost implementation for low-bandwidth data transport and a convenient way to describe library implementations in which the number of communicators may be statically unknown. For example, statistics collection and many of the other services offered by the LEAP FPGA operating system are implemented using Soft Connection rings. Because the inter-ring-stop interconnect is implemented using latency-insensitive channels, rings may be directly partitioned across FPGAs.

Soft Connection rings are a purely physical transport, and protocol implementation on top of the physical layer is the responsibility of the implementor. Libraries

for several basic protocols, including a token-ring, have been implemented on top of Soft Connection rings. Figure 4-2 shows an example of a Soft Connection ring syntax and one possible physical implementation, a chain of FIFOs.

Soft Connections for Modularity

Soft Connections were originally intended to improve design modularity. Most existing HDLs are hierarchical, that is, the entire design is a logical tree rooted at a single top-level module. However, hardware designs are not necessarily isomorphic to trees: many designs require communication between modules whose ancestors may be on separate branches of the module hierarchy. In traditional RTLs, modules in separate branches of the module hierarchy communicate by exporting interface wires through each parent until a common ancestor is arrived at, at which point their wires can be connected.

In addition to requiring tedious, error-prone port replication, hierarchical wiring destroys design modularity. Substitute modules may need to communicate with different parts of the design. In a hierarchical design, these substitutions require modifying each intervening layer for each possible module combination. For a design library like Airblue, discussed in Chapter 8, where each of the dozen modules in a wireless transceiver has a handful of alternatives, modifying the module hierarchy for each combination is onerous.

Soft Connections as a Communication Paradigm

Soft Connections solve the hierarchical wiring problem by introducing a named send/receive primitive into the HDL. These sends and receives are matched by name at compile time, and the compiler automatically inserts the intervening wiring and state between the send/receive pair. Soft Connections were intended to be an abstract, latency-insensitive transport interface between modules and are intentionally divorced from a physical implementation. Indeed, in addition to a simple FIFO implementation, Soft Connections originally supported a shared, tree-like network of point-to-point channels. Although, on a single FPGA, a FIFO-based implementation of point-to-point

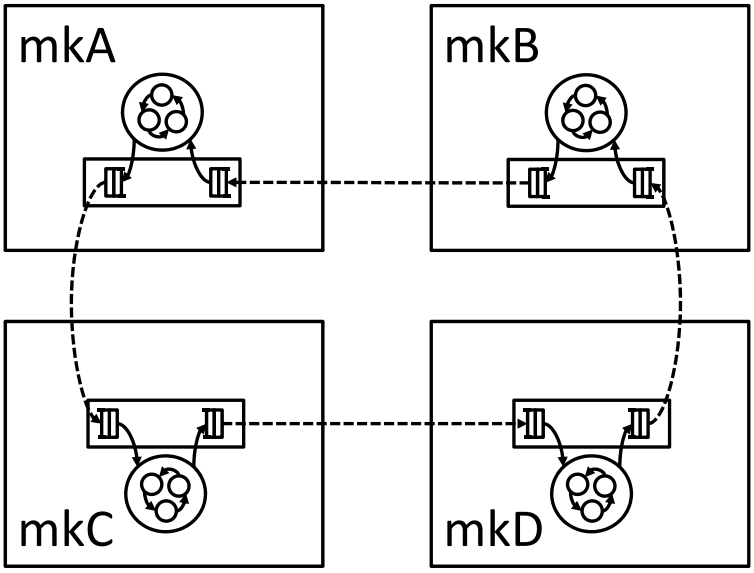
```
module mkA;
 chainA <- mkChain("Stats");
endmodule

module mkB;
 chainB <- mkChain("Stats");
endmodule

module mkC;
 chainC <- mkChain("Stats");
endmodule

module mkD;
 chainD <- mkChain("Stats");
endmodule
```

(a) Ring Syntax



(b) Logical Implementation of 4-2(a)

Figure 4-2: A sample ring declaration and its corresponding logical implementation. Ring stop ordering is chosen by the LIM compiler.

connections is optimal from a performance perspective, the LIM compiler extends the concept of a network of Soft Connections in the automatic synthesis of inter-FPGA networks.

4.2.2 Describing Multiple FPGA Execution Environments

The goal of the LIM compiler is to map a set of latency-insensitive modules on to an execution environment consisting of multiple FPGA platforms. In order to achieve this mapping, the compiler must have a detailed knowledge of the targeted execution environment, including the topology of the FPGA environment, the physical devices,

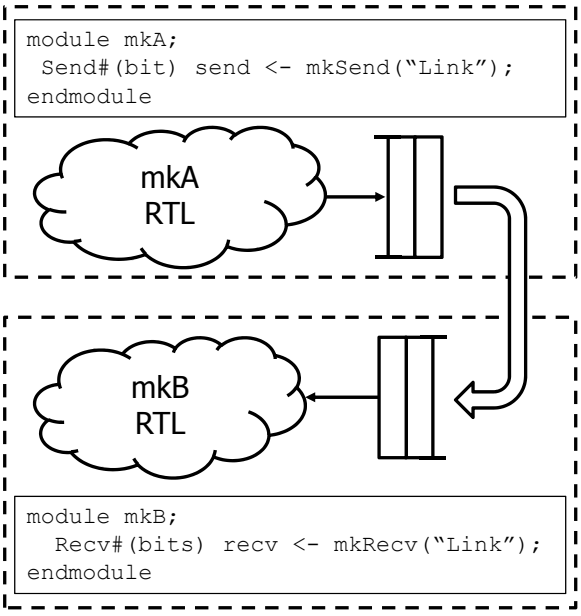


Figure 4-3: A pair of modules connected by a latency-insensitive channel. The arrow represents an automatically generated connection. Users may specify a minimum buffering as an argument to the channel constructor. Here, the tag "ToB" is used to match connections during the compilation flow.

such as memory, attached to each FPGA, and the latency-insensitive interface libraries used to communicate with these devices. This information is conveyed to the LIM compiler by way of the environment description file.

Figure 4-4 shows an example of the syntax used to describe the configuration of a multiple FPGA environment to the LIM compiler. In this example, which corresponds to Figure 4-7, there are two FPGA platforms connected by a single interconnect. This simple description language has two important features: the platform declaration and the arrow notation.

The platform declaration conveys two important pieces of information: the platform service description file and the inter-platform interconnections of the platform. The platform service description file contains all service libraries necessary to build the interfaces to the physical devices, such as memory, attached to the FPGA platform.

Much of the previous discussion of latency-insensitive modules has focused on the structure and properties of the input hardware program. However, hardware programs do not operate in a vacuum: they require access to external resources. For

```
platform FPGA0 drivers_0.apm;
    FPGA1 -> drivers.fromFPGA1;
    FPGA1 <- drivers.toFPGA1;
endplatform

platform FPGA1 drivers_1.apm;
    FPGA0 -> drivers.fromFPGA0;
    FPGA0 <- drivers.toFPGA0;
endplatform
```

Figure 4-4: A sample FPGA environment with two FPGA platforms.

example, some of the Airblue (Chapter 8) designs need access to a radio front-end, while HAsim(Chapter 9) and H.264(Chapter 10) both benefit from access to a large, fast memory store. Although physical devices are necessary in the implementation of real systems, they do not conform to the latency-insensitive model of computation described in Section 2.1. Thus, physical devices are not directly part of the user program, in much the same way that software programs do not directly interface with physical memory or I/O, instead using abstract interfaces and relying on the operating system and underlying hardware to preserve the abstraction.

The LIM compiler chooses to encapsulate physical devices inside of *services*, latency-insensitive interfaces to the devices. Service libraries, included as part of the environment description file, serve the primary purpose of converting synchronous, wired device interfaces into the latency-insensitive channel-based representation operated on by the LIM compiler. Thus, the services offered by each platform can be viewed as an extra latency-insensitive module from the perspective of both the hardware program and the LIM compiler, with the exception that these modules cannot move between platforms.

An example of service interface is the latency-insensitive request-response interface to external memory, shown in Figure 7-1. All latency-insensitive modules interacting with memory instantiate this interface. Instantiating the interface produces, within the instantiating module, several new ring stops for rings carrying requests and re-

sponses to memory. When modules using memory are mapped to an FPGA providing a memory resource, their ring stops will be connected to the physical memory specific to that platform. On the other hand, if no external memory resource is available on the local platform, the LIM compiler automatically routes the interface ring stops to some memory resource on a remote FPGA platform, just like any other latency-insensitive channel. This provides a correct and portable, if lower performance, implementation. Chapter 7 will discuss the implementation details of platform services in full.

The platform declaration also contains a description of the platform's inter-FPGA connections. The arrow operators are used to denote connections between FPGAs, with left arrow representing an incoming connection and right arrow representing an outgoing connection. Each FPGA platform is given a symbolic name in its declaration, which is used in conjunction with the arrow notation to describe connections between FPGAs. The second argument to the arrow operator represents the hierarchical path in the platform service description where the inter-FPGA interconnection device can be found. As noted in Section 2.1, these inter-FPGA interconnections are themselves latency-insensitive channels. Like latency-insensitive channels they must guarantee in-order message delivery. However, the inter-FPGA interconnects are not visible to the user program in the form of a platform service and are not described using the Soft Connections syntax. Rather, the inter-FPGA interconnects are reserved for use by the LIM compiler in generating the inter-module communications network. Each interconnect will eventually be shared by many channels crossing between to FPGA platforms. When viewed in aggregate, the platform declarations form a physical network of FPGAs to which the logical user program will be mapped. The arrow notation permits the description of directed graphs of inter-FPGA interconnects, but currently the LIM compiler requires a bi-directional inter-FPGA interconnect between each pair of communicating FPGAs.

Environment description files are created once per environment configuration and may be shared by all designs targeting the environment. Additionally, environment descriptions are both independent of the program and transparent to the programmer – designs may be targeted to different multiple FPGA environment simply by

```
mkA -> FPGA0;
mkB -> FPGA0;
mkC -> FPGA1;
mkD -> FPGA1;
```

Figure 4-5: A platform mapping for file targeting the execution environment of Figure 4-4.

changing the environment description file input to the LIM compiler. Thus, new multiple FPGA implementations can be described with *seconds* of programmer effort, though, of course, the compilation process may take a significant amount of time. Transparency and portability are achieved through the use of platform virtualization, which will be described in detail in Chapter 7.

4.2.3 FPGA Mapping File

In addition to the platform description file, the current implementation of the LIM compiler requires programmer assistance in the specification and placement of latency-insensitive modules. Thus, in addition to a platform description file, a program-specific platform mapping file is also required as an input to the compiler. Figure 4-5 gives an example of the relatively simple language used for describing the placement of modules across FPGAs, in which the arrow is used to denote the mapping of a latency-insensitive module to a specific platform. This sample mapping file corresponds to the program example shown in Figure 4-6 and again in Figure 4-7. Left-hand arguments are named platforms from the environment description file, while right-hand arguments are the names of program latency-insensitive modules. The mapping file may also be used by the compiler to obtain the names of the latency insensitive modules in the user program.

4.3 Compiler Implementation

LIM compilation consists of three main phases: constructing the LIM graph, placing the LIM modules on to the execution environment, and synthesizing the inter-FPGA

communication network. Figure 4-1 shows the flow of the LIM compiler, including the detailed input and output of each phase. The following sections will elaborate on each of these stages giving details of the algorithms and sub-phases involved in each.

4.3.1 LIM Graph Construction

The first phase in LIM compilation is the construction of a graph representation of the input program, the LIM graph. In the LIM graph, edges correspond to latency-insensitive channels, while vertices correspond to the synchronous RTL internals of the latency-insensitive modules. The vertices of the graph, the latency-insensitive modules, are described explicitly by the programmer in the mapping file. However, the edges of the graph, the latency-insensitive channels, are inferred from the user program directly, through the process of channel discovery.

Latency-insensitive channel discovery commences with an initial Bluespec compilation pass over the user program. As Bluespec compiles the program source, it will emit a string containing information about each Soft Connection endpoint that it encounters[2]. This endpoint meta-data includes the module to which the endpoints belong and the endpoint type. These emitted strings are stored in log files on a per-module basis for future use by the LIM compiler.

At the end of the initial compilation pass, the Bluespec compiler has generated a log file for each module, which contains a description of the connections of the module. The LIM compiler then proceeds to construct the LIM graph, beginning by parsing the Bluespec-produced log files. At the end of this parse, the LIM compiler posses a nearly complete LIM graph: each latency-insensitive module of the user program is decorated with its named ingress (receive) and egress (send) latency-insensitive channels. The compiler completes the construction of the LIM graph of by matching the names of the Soft Connections to form graph edges.

The matching stage mates the point-to-point channels discovered during the com-

[2]Some Soft Connections may have endpoints which reside in the same module. Although the compiler does not handle this case specially, since latency-insensitive modules are indivisible and cannot by partitioned between FPGAs, these channels will eventually be implemented to simple RTL FIFOS.

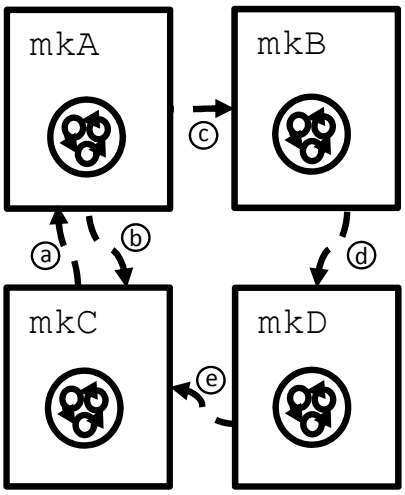


Figure 4-6: A sample LIM graph produced by the first phase of the LIM compiler.

pilation pass, filling in the edges of the LIM graph. Channels are matched by name, by iterating over the set of channel names for each channel. When a match is made, a new graph edge is inserted in the LIM graph between the source and sink modules.

Ring endpoint matching is postponed until after physical placement as an optimization. The ordering of ring stops is left to the compiler and could be determined while matching the point-to-point stops. The problem in the an early choice of ring ordering is that, at LIM graph construction time, all orderings are equally good. Once placement occurs, some of the choices of ordering may turn out to be suboptimal, resulting in needless inter-FPGA crossings. Consider Figure 4-9: the order ACDB minimizes the number of inter-FPGA crossings, while the ordering ADCB results in two spurious crossings. Although ordering the stops is deferred, the existence of ring stops is noted in the LIM graph so that the placement phase can observe them.

At the end of the matching phase, the LIM compiler possesses a largely-complete, logical graph of all the communications in the user program, wherein vertices represent latency-insensitive modules and edges represent matched latency-insensitive channels. This graph is used by subsequent compilation phases both to place the modules onto the execution environment and to generate the communication network.

It is possible at the end of the matching stage to have some point-to-point channels which do not have matches. If any channels are not matched at the termination of this

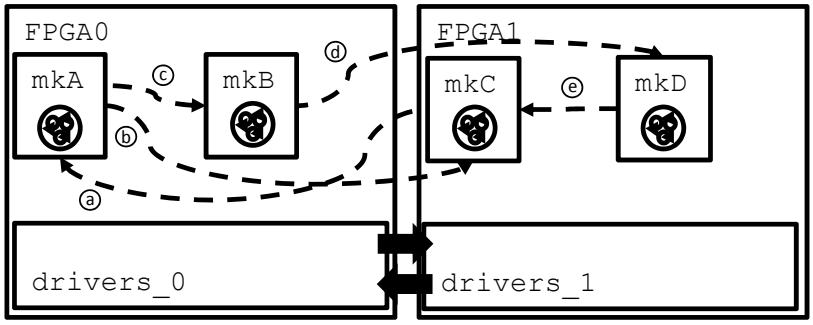


Figure 4-7: The modules of Figure 4-6 have been placed into an environment consisting of two FPGA platforms. The dotted lines represent the logical LIM graph, which has not yet been physically elaborated.

stage, then the program is erroneous and compilation terminates with a message about unmatched channels. However, some channels, particularly those associated with physical services like memory, may not be used by a given program. To accommodate this case, the Soft Connections syntax provides an explicit primitive for optional channels. The processing of optional channels is the same as the processing of normal channels, with the exception that unmatched optional channels do not result in the compiler reporting an error.

4.3.2 Module Placement

Once the set of latency-insensitive modules has been discovered and the LIM graph built, the logical representation of the user program can be mapped on to the target execution environment. In an ideal implementation of a LIM compiler, this mapping of modules to platforms would be fully automated: the compiler would use estimates of module area and inter-module communications to place modules across FPGAs. However, the programs considered in this thesis are comprised of only a handful of modules, and so the LIM compiler presently supports only a manual mapping. As programs and environments scale in size, automatic placement will be necessary. Thus, automatic placement of modules is an important future work, and a strategy for these algorithms will be discussed in Chapter 11.

Given the description of the environment, a mapping file, and the graph of modules produced by the channel discovery phase, the current compiler proceeds to map

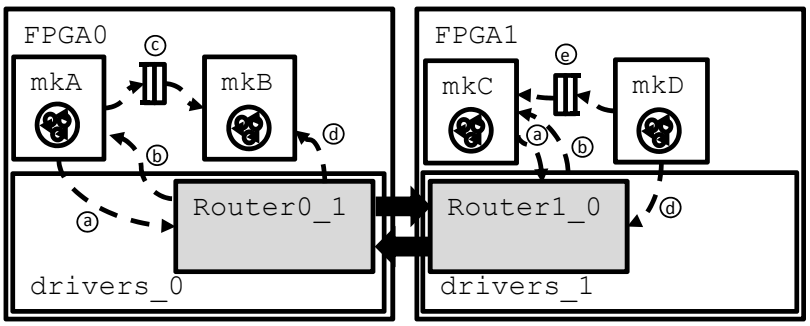


Figure 4-8: A fully elaborated multiple FPGA design. The logical latency-insensitive channels of Figure 4-7 have been replaced with RTL implementations. Modules placed on the same FPGA communicate via RTL FIFOs, while remote modules communicate by way of the synthesized router infrastructure.

program modules on to platforms in the environment, by marking modules with the platforms on which they have been placed. This information will be used in subsequent stages both to route signals among the FPGAs and to synthesize the inter-FPGA routers.

4.3.3 Network Synthesis

At this point, the LIM compiler has determined the topology of a program comprised of latency-insensitive modules and has mapped these modules down onto a physical topology of FPGA platforms. The next step of LIM compilation is to generate the network between latency-insensitive modules. There are several individual steps in the synthesis process. First, the physical communication topology of the program must be completed. In the original LIM graph, communications channels were all identical logical objects. However, in the physical realization of the program, channels may need to cross several FPGAs to move data between modules. The compiler must therefore route channels connecting remote modules across intervening FPGAs.

Once the physical communication topology is completely specified, the routers multiplexing the inter-FPGA links can be synthesized. Abstractly, these routers simply multiplex the physical interconnect between a pair of FPGAs among all latency-insensitive channels crossing between that pair of FPGAs. However, instantiating a simple multiplexor may be highly suboptimal because the physical interconnect be-

tween FPGAs typically consists of a wide, bit-parallel interface. The goal of router synthesis is to partition this wide interface in a way that maximizes the throughput of the channels routed between a pair of FPGAs. A brief sketch of this process is given here, but the subject will be treated more fully in Chapter 6.

Finally, once the physical topology and communication network are completely specified, the compiler generates a Bluespec program for each FPGA platform, representing the latency-insensitive modules mapped to that platform and the synthesized network carrying the latency-insensitive communication channels of those modules. These programs can be used to produce a physical, multiple-FPGA implementation of the original program.

Channel Routing

After the program graph has been mapped on to a particular physical substrate, it may be the case that some modules which communicate with one another are not placed together on the same FPGA. In this case, the compiler must insert some new latency-insensitive channels at intervening FPGA boundaries to form a path between the communicating modules. An example of this routing is shown in Figure 4-10. At the end of this routing phase of compilation, the logical graph of communication implied by the user source is completely mapped to the physical platform upon which it will execute.

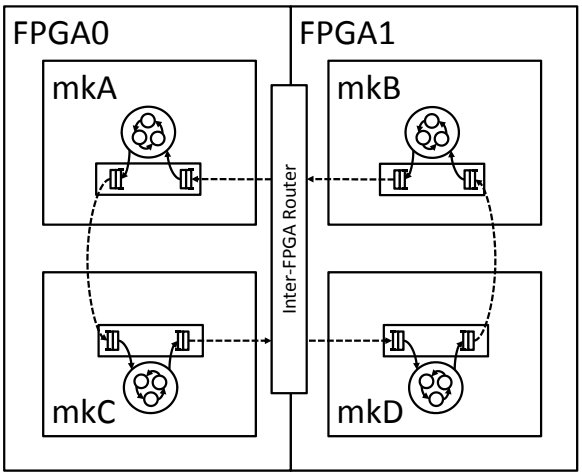
Given a set of non-local channel endpoints, the current compiler implementation routes point-to-point channels by finding the path with the shortest number of inter-FPGA hops on the platform graph. The compiler then dissolves the original edge in the LIM graph and inserts a new set of edges and vertices. The new vertices correspond to null latency insensitive modules, while the vertices correspond to the new latency-insensitive channels required to route the message across the environment. In the code generation phase, these insertions to the LIM graph will result in the instantiation of a latency-insensitive channel crossing between the two appropriate routers on each intermediate platform between the original modules. During the final RTL generation stage, these new channels will be compiled to RTL FIFOs connecting the

appropriate ingress and egress routers, allowing the point-to-point channel to traverse the FPGA boundaries.

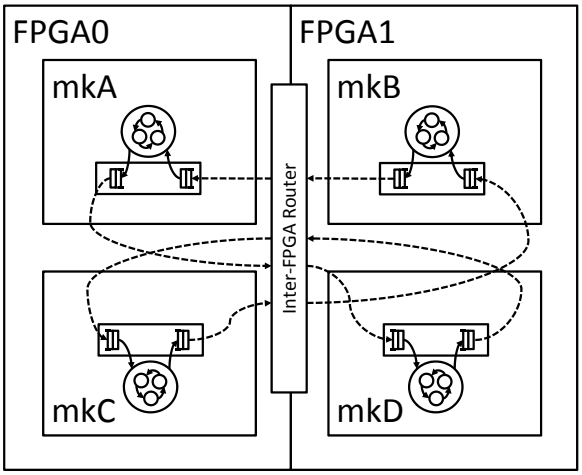
Greedily routing channels by choosing the shortest-hop path is both simple to implement and latency-minimizing in the best case. The problem with always choosing the shortest-hop path for each channel during the routing step is that inter-FPGA bandwidth and loading is completely ignored. The compiler will happily allocate all channels between two FPGAs along the shortest-hop path, even if that path has low bandwidth or becomes oversubscribed due to the allocation. Although the programs examined in this thesis have not yet encountered a performance penalty due to this simple allocation, it is easy to imagine that performance degradation could occur in some programs and for some environments. A future refinement to the basic algorithm is to incorporate some form of max-flow, min-cut algorithm which can account for bandwidth between the FPGAs. Such an algorithm would also be useful in the placement problem of the previous section.

During the LIM graph construction phase, the compiler deferred choosing the ordering of ring stops. Now that physical placement is known, a ring endpoint ordering and routing can be produced. Ideally, the compiler will choose an ordering of the stops such that the number of inter-FPGA crossing encountered along the ring is minimized. Thus, the global routing of rings reduces to finding a Hamiltonian cycle among the platforms hosting the ring stops: physically local ring stops are connected in sequence with the first and last endpoint connected to the preceding and succeeding platforms in the Hamiltonian cycle of platforms.

Of course, not all topologies of platforms and ring stops will have a Hamiltonian cycle. If this is the case, then the compiler, as in the point-to-point case, must insert virtual latency-insensitive modules on some of the platforms to close the cycle. An example of a fully routed ring is shown in Figure 4-10. Since this topology does not contain a natural Hamiltonian cycle, the compiler injects a latency-insensitive module on the middle FPGA to induce a cycle. In the physical implementation, this results in a single extra RTL FIFO transiting the middle FPGA.



(a) A Hamiltonian cycle of platforms for the ring in Figure 4-2. This implementation minimizes inter-FPGA crossings.



(b) A non-Hamiltonian cycle of platforms for Figure 4-2. This implementation is correct, but sub-optimal in terms of area and throughput.

Figure 4-9: Ordering ring-stops to induce a Hamiltonian cycle of platforms minimizes inter-FPGA interconnect and maximizes throughput. However, achieving such an ordering requires that the placement of modules is known.

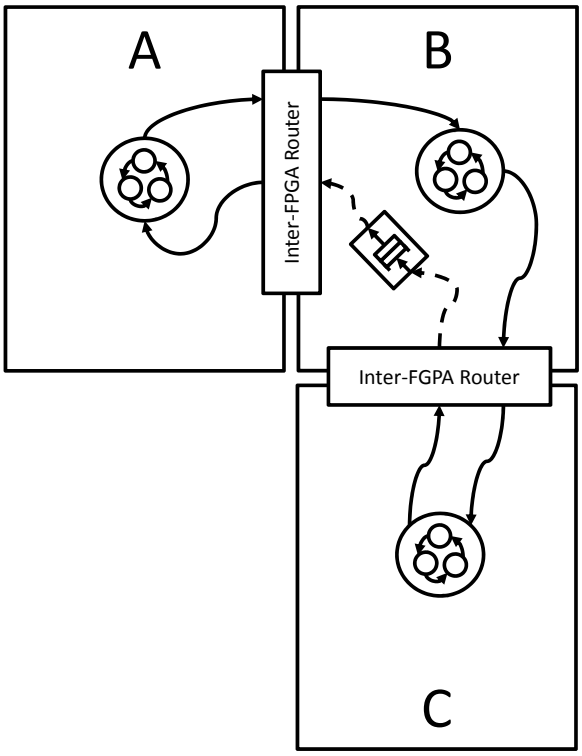


Figure 4-10: An example of channel routing. Here, the chain traverses A,B, and C. To close the chain, an additional channel, denoted with dashed lines, is inserted across B. A similar process is followed for point-to-point channels.

Router Generation

After placement and channel routing, the LIM graph is fully elaborated: the vertices of the graph have been logically partitioned into the subgraphs mapped to each platform and the edges in the graph are either local to a subgraph or the cross between two subgraphs. At this point, the only task remaining in producing a multiple FPGA implementation is to elaborate the elements of the inter-FPGA communications network. Edges within the same subgraph are simple: they are local to a single FPGA and can be implemented with a simple RTL FIFOs. Edges that span two platforms are also seemingly straightforward: all the edges between two platform subgraphs are tied to a router that multiplexes the physical interconnect between the two platforms.

Conceptually, network synthesis is straightforward. However, the implementation of the inter-FPGA routers bears some consideration. Simply building a naive router that time-multiplexes the interconnect between two FPGAs among all the channels crossing between them is often extremely wasteful, due to typical channel parameters. In particular, most inter-FPGA interconnects operate at roughly 10 Gbps. Modern FPGAs have dozens of these high-speed interconnects, which can be aggregated for higher-bandwidth. This means that a user design operating at 100 MHz must produce between 40 and 200 bits of data per inter-FPGA link per cycle to make full use of the offered inter-FPGA bandwidth. However, most user channels have narrow bitwidth and cannot make full use of the inter-FPGA interconnect. As a result, program performance can generally be improved by partitioning the physical interconnect into multiple smaller lanes which can operate in parallel. If an inter-FPGA interconnect is split into several lanes, the assignment of channels to lanes immediately becomes an important decision problem.

Chapter 6 explores various algorithms for the parameterization of and the allocation of channels to routers. However, the behavior of these operations as a compiler phase is uniform. The algorithms use the fully placed and routed communication graph along with other program information to produce a lane sizes and an allocation of channels to lanes. Each lane produced in this phase will eventually instantiate

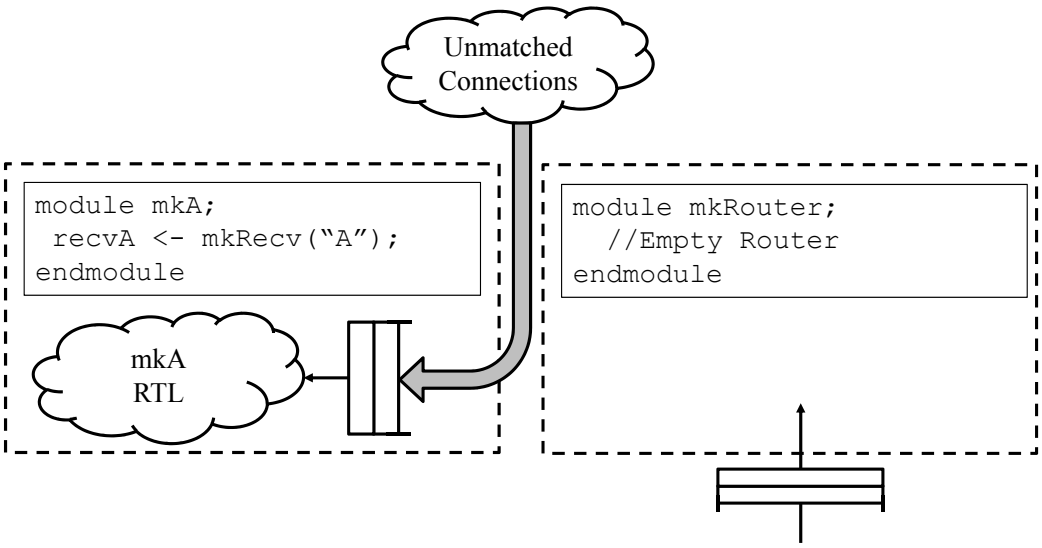
a lane-specific router during the code generation phase. It is important to note that the router configurations need not be symmetric: each inter-FPGA interconnect may have a different number of lanes and a different numbers of channels.

Once the network routers are parameterized, network synthesis is complete, and the LIM compiler can generate Bluespec programs for each physical platform. The generated program consists of the set of latency-insensitive modules placed on the platform, the synthesized network, and the platform-specific services. The modules and devices services are copied almost verbatim from the original source, though they are placed within wrappers. To produce the network code, the compiler selects appropriate components, i.e. routers and scheduling logic, from the highly parametric router library described in Chapter 5. These components are then connected to the latency-insensitive modules by inserting appropriate Soft Connections into the synthesized router description. During the subsequent Verilog generation phase, the Bluespec compiler will convert these Soft Connection to RTL FIFOs.

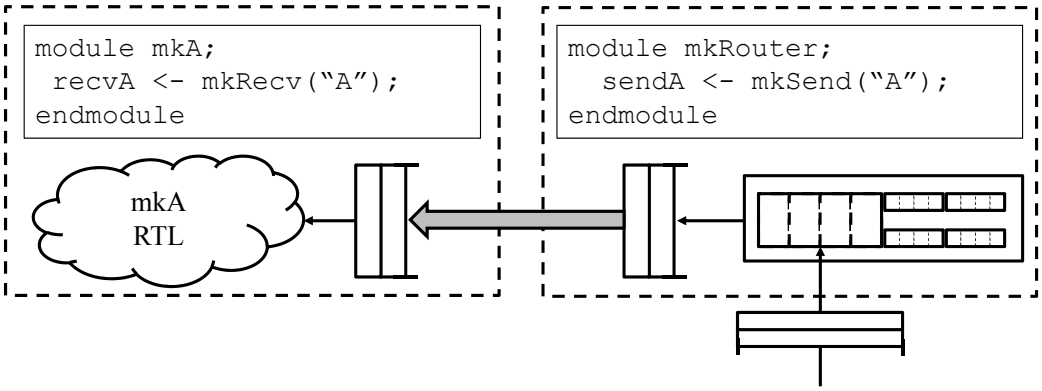
To improve the readability of the generated code and to facilitate implementation exploration, most of the code produced during the code generation phase is drawn from a highly parameterized Bluespec library. However, some code must still be produced by the compiler itself, to work around limitations of the Bluespec type system. For example, in the case of the variable width lanes created during router generation, Bluespec's type system is not sufficiently descriptive: Bluespec does not easily admit of a variable number of variable width arguments. In this case, the code generation phase must synthesize and inject the full module code, rather than just instantiating a parametric module out of the router library.

4.3.4 RTL Generation

At this point, the LIM compiler has produced a full Bluespec code describing the physical implementation of each FPGA platform in a complete multiple FPGA implementation. In the last stage of the LIM compiler, these codes, which contain the synthesized network, the original module source, and the platform interface libraries, are recompiled by the Bluespec compiler to RTL on a per FPGA basis. The Verilog



(a) First Compilation Pass



(b) Second Compilation Pass

Figure 4-11: An example of two-pass compilation. In the first pass, unmatched channels are discovered. In the second compilation pass, formerly unmatched channels are matched to the generated router code, channeling FPGA platforms.

generated by this final step can be simulated or passed to back-end synthesis tools to produce FPGA programming files for each FPGA platform in the system.

In addition to the differences in source to be compiled, the second Bluespec compilation differs from the first in its handling of Soft Connections. In the first compilation pass, the placement of the latency-insensitive modules was unknown, and Soft Connections represented logical latency-insensitive channels. However, in the second pass, modules have been placed on physical platforms and all latency insensitive channels have been fully elaborated and bound to a physical implementation. Thus, all Soft Connections refer to strictly local communication within a single platform during the

second pass, whether this communication is between two modules placed on the same FPGA or between a module and the inter-FPGA communication infrastructure for that platform. The compiler chooses a physical implementation of the channels during the second pass: simple RTL FIFOs. An example of this difference in handling is shown in Figure 4-11. In this case, the program contains receive channel"A" which is initially un-elaborated in the channel discovery phase, Figure 4-11(a). However, in the second Bluespec compilation pass, the synthesized router contains the endpoint to this channel and during this pass, the Soft Connection between the program and the router infrastructure will be replaced with an RTL FIFO, as shown in Figure 4-11(b).

Although the current compilation needlessly recompiles many portions of the user program during the second compilation phase, much of this work can be avoided. User modules do not change substantially when passing through the LIM compiler. In theory the modules could simply be linked against the synthesized routers, provided that the LIM compiler maintained sufficient intermediate compilation information between the two compilation passes. Indeed, even the physical FPGA synthesis of the user latency-insensitive modules for FPGA is constant between the passes, a fact which will become important in automatic placement algorithms, which will require high-quality synthesis area estimates.

4.4 Interfacing to the Bluespec Compiler

The LIM compiler operates on Bluespec System Verilog augmented with Soft Connections syntax. Unfortunately, the source of the Bluespec compiler is not publicly available. However, Bluespec does provide a set of compiler interfaces so that permit the LIM compiler to make use of Bluespec to parse source files and to construct a representation of the latency-insensitive channels within the source program . Because the LIM compiler must use this reduced interface, some complications in compilation arise.

Bluespec provides a useful mechanism for implementing compiler-like interfaces: `ModuleContext`. `ModuleContext`, as an implementation of the Haskell state monad,

permits the construction and mutation of arbitrary data structures within the compiler at compile time. These structures are hidden from the program writer and remain within the `ModuleContext` until extracted. Thus, new language primitives can be implemented by providing users modules that store state with the `ModuleContext`. By wrapping program modules with extraction code, a representation of the primitives instantiated during compilation can be written to a log file and this representation can be examined off-line by an external compiler, for example, the LIM compiler. Although this interface is poor in the sense that it does not offer full visibility into the operation of the Bluespec compiler, it does offer compiler writers some capacity to create new language-level primitives and offer them to the programmer without writing a new compiler front-end and without the programmer becoming aware of the implementation of the extra-language primitives.

The previous sections noted that the LIM compiler needed some programmer annotation to determine latency-insensitive modules. In an ideal LIM compiler, this assistance would not be necessary; the compiler would infer the existence of latency-insensitive modules directly from the channel-annotated source. However, inferring modules requires access to a low-level intermediate representation of the hardware program, which Bluespec does not provide. As a result, although the LIM compiler can extract information about latency-insensitive channels, it cannot determine latency-insensitive modules without programmer assistance. This assistance is provided by way of the explicit declaration of latency-insensitive modules in the mapping file described in Section 4.2.2. Due to this need for programmer assistance, there is a strong correspondence between some Bluespec modules and the latency-insensitive modules of the LIM compiler. In an ideal compiler, this correspondence between source modules and latency-insensitive modules would not necessarily exist. The LIM compiler currently makes no distinction between the definition and instance of latency-insensitive modules: the compiler instantiates each declared module exactly once. However, Bluespec modules not declared in the mapping file are permitted both to have multiple instantiations and to be latency-sensitive.

The designs considered in this thesis are comprised of coarse-grained latency in-

sensitive modules which do not require replication. However,the lack of differentiation between instance and definition represents a deficiency of the current LIM compiler. This could be corrected by adding support for explicit primitives for latency insensitive modules into Bluespec, by way of `ModuleContext`, or by writing a full compiler front-end capable of extracting latency-insensitive modules directly from source.

4.5 Conclusion

The LIM compiler takes as input a program written in terms of latency-insensitive modules and a description of the execution environment to which that program will be mapped. The compiler then, through a series of steps, produces a set of programming files that partition the original program across the FPGA platforms of an execution environment. To achieve this partitioning, the compiler builds a graph representation of the design comprised of vertices representing latency-insensitive modules and edges representing latency-insensitive channels. Next, the compiler maps this graph onto the execution environment and synthesizes a network between the modules of the graph. Finally, the compiler produces an RTL implementation of the multiple FPGA system.

While, the current implementation of the LIM compiler can produce fully functional implementations partitioned across an arbitrary configuration of FPGAs, it is incomplete. Due to limitations in language parsing and a strong dependence on the Bluespec compiler, the compiler requires the programmer to explicitly annotate latency-insensitive modules. Additionally, the mapping of latency-insensitive modules to the execution environment requires manual assistance. Many of these limitations are surmountable and represent an important future work.

Chapter 5

Router Architecture

The preceding chapter discussed the process of automatically converting high-level programs annotated with latency-insensitive channels into multiple FPGA implementations. An important part of this compilation process is the the automatic synthesis of an inter-module communication network. For modules located on the same FPGA, the synthesized network is nothing more than simple RTL FIFOs. However, the network between FPGAs is substantially more complicated. The key technical challenge of inter-FPGA networking is to produce a high-quality router to multiplex the inter-FPGA interconnect among the potentially many latency-insensitive channels crossing the interconnect, as shown in Figure 5-1. A good inter-FPGA router provides the illusion of FIFOs with back-pressure to the user latency insensitive channels, while offering high performance, in terms of latency and throughput, consuming few FPGA resources, and, most importantly, preserving the functional correctness of the original latency-insensitive program.

This chapter will first prove the functional correctness of inter-module networks synthesized by the LIM compiler. The chapter then introduces a high-quality inter-FPGA router satisfying the requirements of the proof. This router is built around the SRAM Multi-FIFO (SMF) [18], a novel, FPGA-efficient buffering resource. Figure 5-2 shows the organization of the router, which has been partitioned into two halves: the ingress and egress routers. Although the SMF is the core of the router, the router architecture consists of three pipelined layers built on top of some phys-

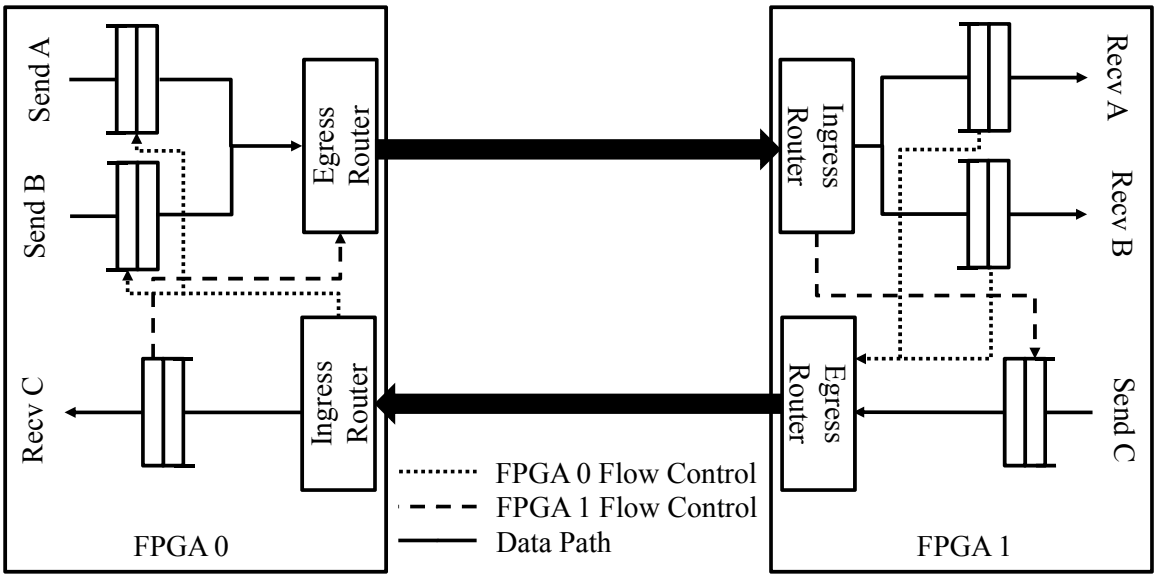


Figure 5-1: Logical Architecture of inter-FPGA channels.

ical interconnect: channel marshalling, channel routing, and lane merging. Channel marshalling converts the variable width data types carried by the latency-insensitive channels into uniformly-sized network packets. Channel routing provides flow control, packet buffering, and arbitration of the shared inter-FPGA interconnect. Lane merging extends the router to support multiple lanes, allowing channels mapped to different lanes to concurrently transmit on different portions of the same wide physical interconnect. The remainder of the chapter will describe, in detail, the behavior and physical implementation of the router.

5.1 Correct Latency Insensitive Networks

For the latency-insensitive model of computation, network correctness means preserving the illusion of latency-insensitive channels, or, more precisely, the guaranteed, in-order delivery of messages. This is a relatively simple requirement, particularly when mapping a program to a single FPGA. Some complexity arises when mapping a design to multiple FPGAs, because many channels can cross between FPGAs and must share a single inter-FPGA interconnect. This multiplexing of the physical interconnect between the FPGAs can introduce deadlocks if the network is not properly constructed.

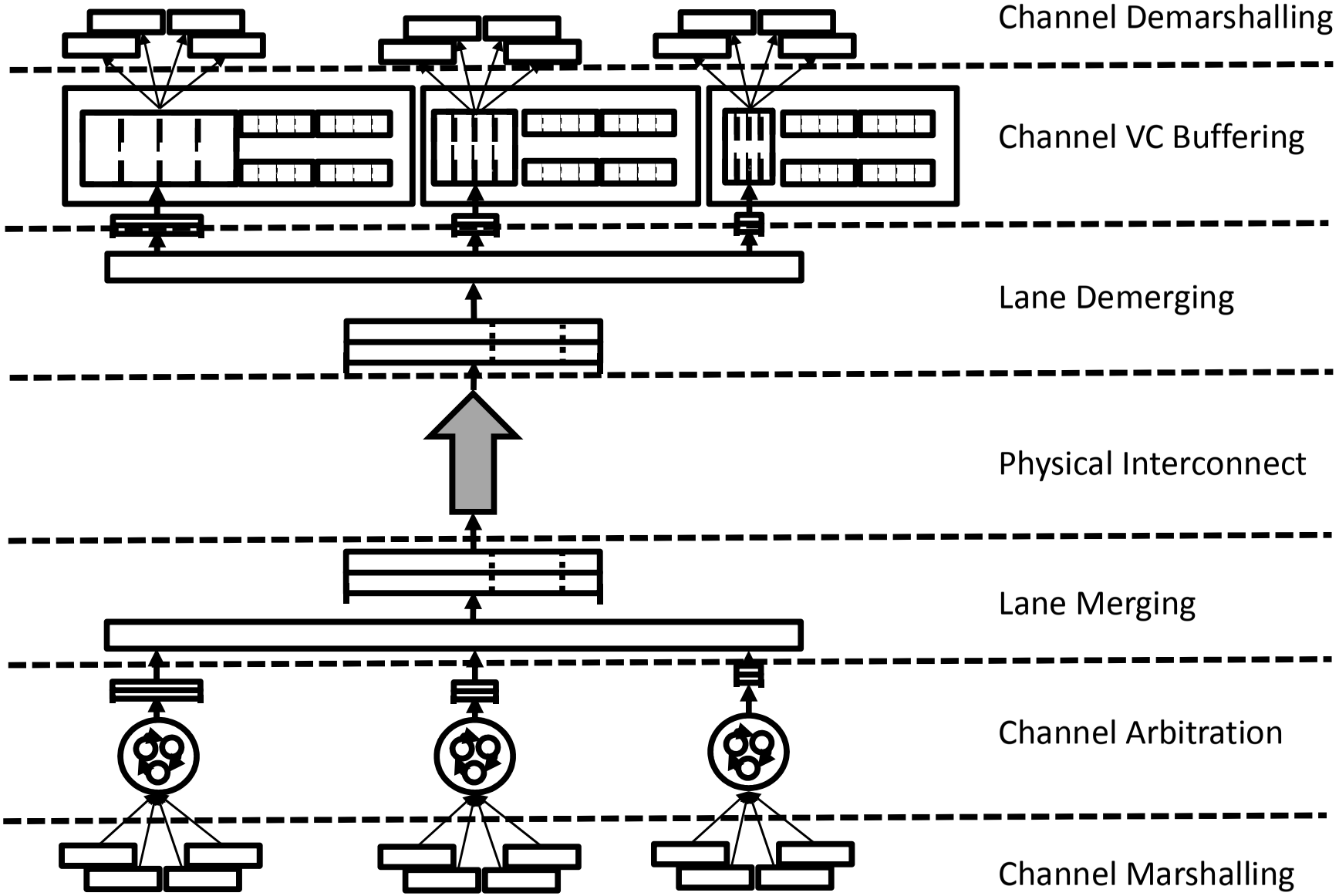


Figure 5-2: A complete FPGA router, partitioned into ingress (top) and egress (bottom) halves, exhibiting all router layers. The router depicted has three parallel lanes of different, statically determined width. This router services twelve independent channels, with four channels statically allocated to each lane.

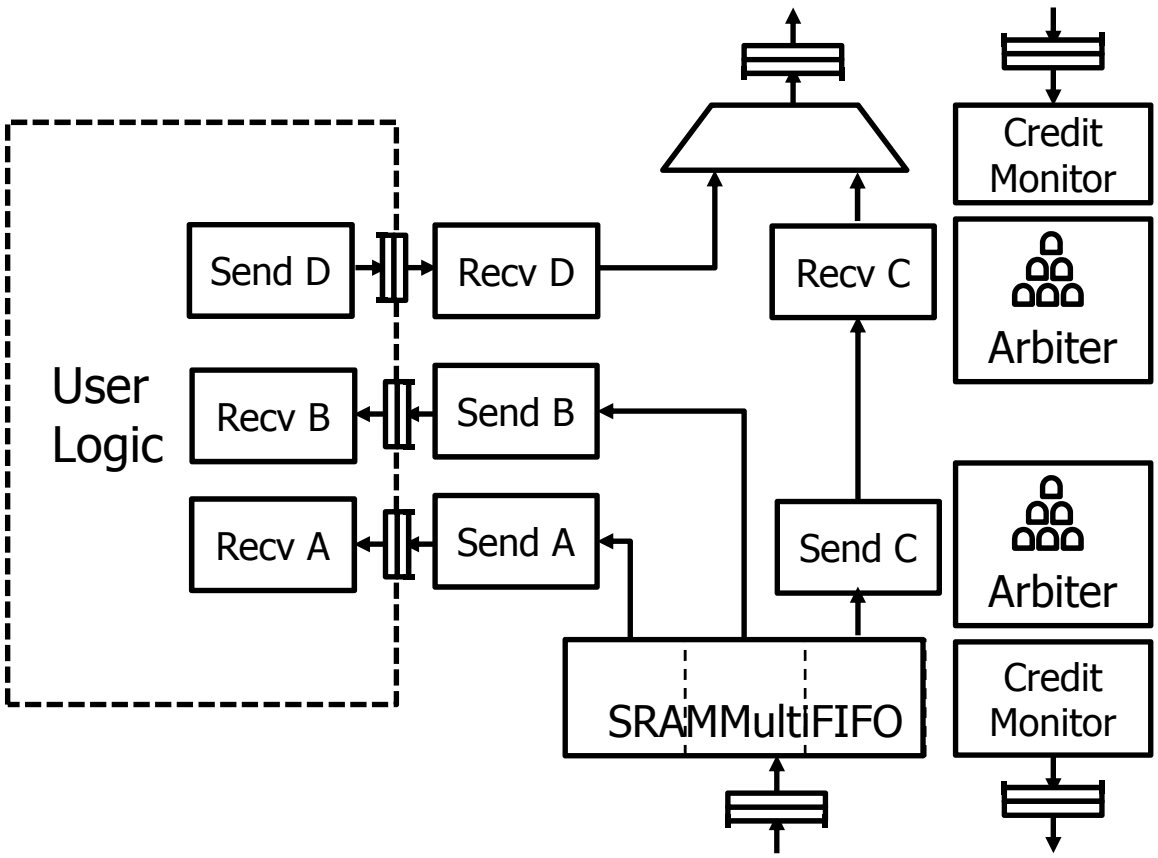


Figure 5-3: An example of a synthesized router connecting an FPGA with to two other FPGAs (not shown). Channel "Recv C" is routed through the FPGA.

However, the networks synthesized by the LIM compiler are *deadlock-free*.

Deadlocks arise in shared interconnect when dependent packets are forced to share the same routing paths, which can cause the packets to block each other. Figure 5-4 shows a simple example of how such a deadlock can arise: head of line blocking. In this case, channels A and B are dependent – a value from each is required for the multiplication to occur. However, because B has momentarily produced data faster than A and filled the shared inter-FPGA FIFO, the system has deadlocked.

To avoid deadlock resulting from shared paths in a network, virtual channels are introduced to break dependence cycles [12]. Virtual channels are independent buffer resources which multiplex the shared path. Because the virtual channel buffers are independent, the virtual channels do not block each other. Choosing the number of virtual channels at the router requires knowledge of the specific communication

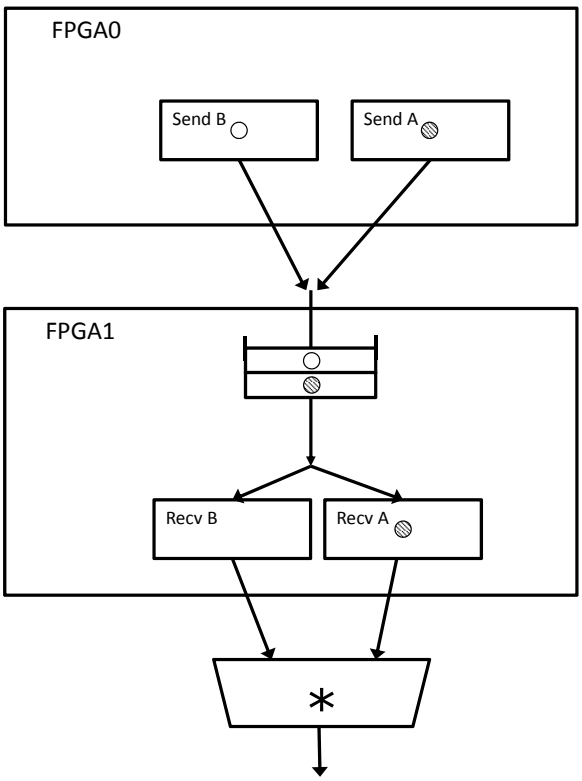


Figure 5-4: A deadlock arising from a data dependence in a shared resource.

being carried out on the network. In traditional computer architectures, deciding the number of virtual channels is a tractable problem since the communications behavior, usually some form of coherence protocol, is known statically and dependencies can be explicitly broken at design time by inserting a minimal number of virtual channels. However, reasoning about the communications dependencies of an arbitrary latency-insensitive program is difficult. Therefore, the LIM compiler must allocate a virtual channel to each latency-insensitive channel crossing between FPGAs. Virtual channel allocation alone is not sufficient to ensure deadlock freedom, because full virtual channels can still cause head-of-line blocking across the shared physical channels. To resolve this issue flow control across is introduced on each virtual channel, ensuring that every message injected into the shared physical interconnect will eventually be drained.

Together, universal virtual channel allocation and flow control are sufficient to guarantee that the multiple FPGA implementations produced by the LIM compiler

do not introduce deadlocks into previously deadlock-free latency-insensitive designs. This property is an easy corollary of the Dally-Seitz theorem [12], which guarantees that virtual channels can multiplex a shared physical substrate without deadlocking, so long as the the dependency graph of the virtual channels is acyclic. Since the network formulation presented in this thesis provides a virtual channel for each latency-insensitive channel, it must trivially have no dependence cycles. Thus, dependent channels are prevented from blocking one another and deadlock is avoided.

5.2 Channel Marshalling

The first stage in inter-FPGA routing is channel marshalling. Router widths are fixed statically to simplify the implementation of router hardware, but latency-insensitive channels, whose widths are determined by the type of the data carried by the channel, may be arbitrarily wide. Therefore, a packetization scheme is needed to map wide channels in to a fixed-width packet format. Since all communications channels and channel widths are statically determined at compile time, the compiler can infer a bit-optimal packet protocol for each channel.

The LIM compiler will instantiate an optimal protocol for each channel based on the width of the data transported by that channel and the width of the router to which the channel has been allocated. These protocols are specific instantiations of a header-body packet schema in which the header contains information about the packet length, type, and virtual channel. There are three possibilities for implementation, each of which is shown in Figure 5-5. First, in the case that the data width is wider than the physical channel, marshalling and de-marshalling logic is automatically inserted. This logic takes the form of a shift-register. Since headers tend to be only a few bits wide, a portion of the payload is packed in with the header to improve bandwidth utilization. The second case is that the data width is small enough to fit into a single body word, but too large to be packed in with the header. In this case, no shift register is needed, but a separate header word is needed. Finally, if the data width is sufficiently small, the packet header and body will be bit-packed together into a

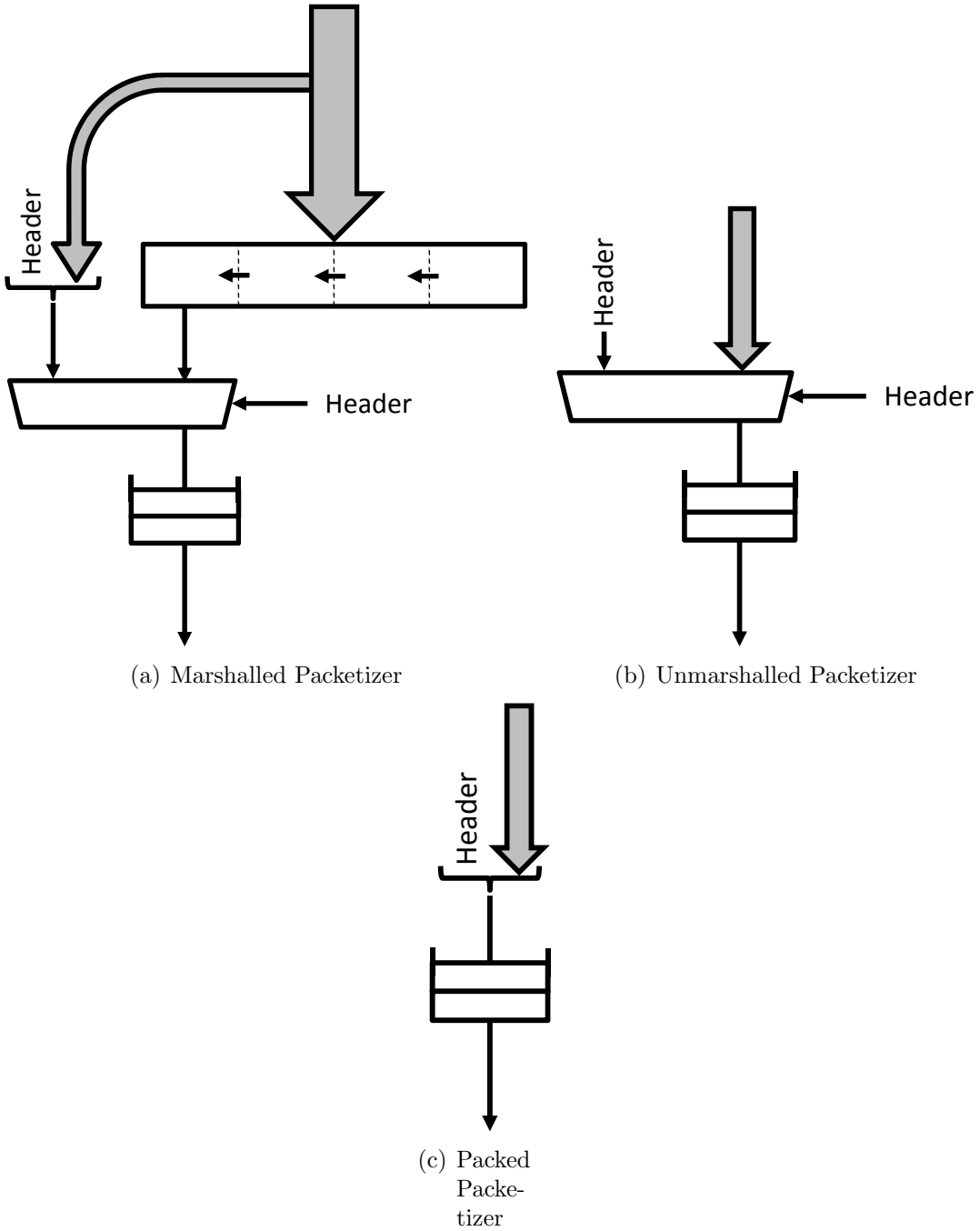
single word transmission. Since the data communicated between FPGAs tends to be narrow, this is a significant performance optimization.

5.3 Efficient Virtual Channel Buffering: the SRAM-MultiFIFO

LIM-compiler-generated networks require flow control on a per channel basis to guarantee functional correctness in multiple FPGA implementations. This is a seemingly costly proposition, since the high latency on the inter-FPGA interconnect implies the need for large amounts of buffering to utilize the full interconnect bandwidth. The inter-FPGA latency appears to create a cost-performance tradeoff between buffering per virtual channel and the performance and area of the router, since too little buffering can cause the sender to stall before the receiver even begins to receive packets, while too much buffering appears to reduce the FPGA area available to the user design. Indeed, a naive, register-based flow control implementation with buffering sufficient to cover a round-trip latency of 16 cycles requires half the area of a large FPGA. Physically, this kind of implementation does not scale beyond a pair of FPGAs.

The problem with the register-based design is that it is unnecessarily parallel and therefore needlessly wasteful of resources. In any cycle, any of the registers in any of the buffers can potentially supply a data value to transmit. However, observe that the inter-chip bandwidth between FPGAs is limited to a single, though wide, data word per cycle. This bandwidth limitation means that to sustain the maximum rate across the channel, exactly one channel needs to be enqueued or dequeued in any given cycle. Therefore, a structure with low parallelism, but with high storage density, is sufficient to sustain nearly the maximum throughput of the physical channel.

Most modern FPGAs are rich in SRAM, with a single chip containing megabytes of storage. Although large amounts of memory are available, the bandwidth to each slice of this memory is limited to a single word per cycle. Because inter-FPGA



(a) Marshalled Packetizer

(b) Unmarshalled Packetizer

(c) Packed Packetizer

Figure 5-5: Channel packetization hardware. Typical headers are only a handful of bits. The style of packetization is automatically chosen by the compiler at compile time.

communication is similarly constrained, in terms of bandwidth, virtual channels with relatively large buffers can be mapped onto the resource-efficient SRAM without significant performance loss. This optimized storage structure, depicted in Figure 5-6, is called the SRAMMultiFIFO (SMF) [18]. Since the SMF maps many FIFOs onto a single SRAM with a small number of ports, it must introduce an arbiter to choose which FIFO will use the SRAM port in a given cycle. FIFOs mapped to the SMF have uniform and constant size which simplifies control logic at the cost of potentially unused storage space. Thus, the SMF may increase the amount of buffering in the inter-FPGA network over a registered implementation. However, this area penalty is inconsequential due to the large number of SRAM resources on the typical FPGA. Because the bandwidth of the SMF is limited, book-keeping logic can also be time multiplexed and book-keeping meta-data stored in area efficient LUT-RAM.

The SMF, itself, is fully pipelined, and each mapped FIFO can utilize the full bandwidth of the SRAM. However, FPGA-based SRAM typically have a single cycle of read latency, which requires that channels reading data out of the SMF provide extra buffering in the case that they require full pipelining. When channels win the SMF arbitration, they receive notification from the SMF in order to update their own internal meta-data, so that they do not request more data than they can buffer. In practice, only high-bandwidth channels require extra buffering, since only these channels have enough traffic to exploit full pipelining.

Area usage for SMF and a functionally similar register-based FIFO implementation are shown in Table 5.1. The SMF scales in BRAM usage as FIFO depth or the number of channels increases, but typical designs consume only around 2% of slices on a Virtex-5 LX-330T. The low area usage of the SMF-based router makes it amenable to FPGA platforms with a high-degree of inter-platform interconnection. On the other hand, registered buffer schemes can quickly exhaust large amounts of area. Moreover, the largest implementable registered FIFO router has no better performance than a more resource efficient, but deeper SMF router, despite the inherent parallelism of the registered implementation.

The density of the SMF fundamentally changes the way that inter-FPGA commu-

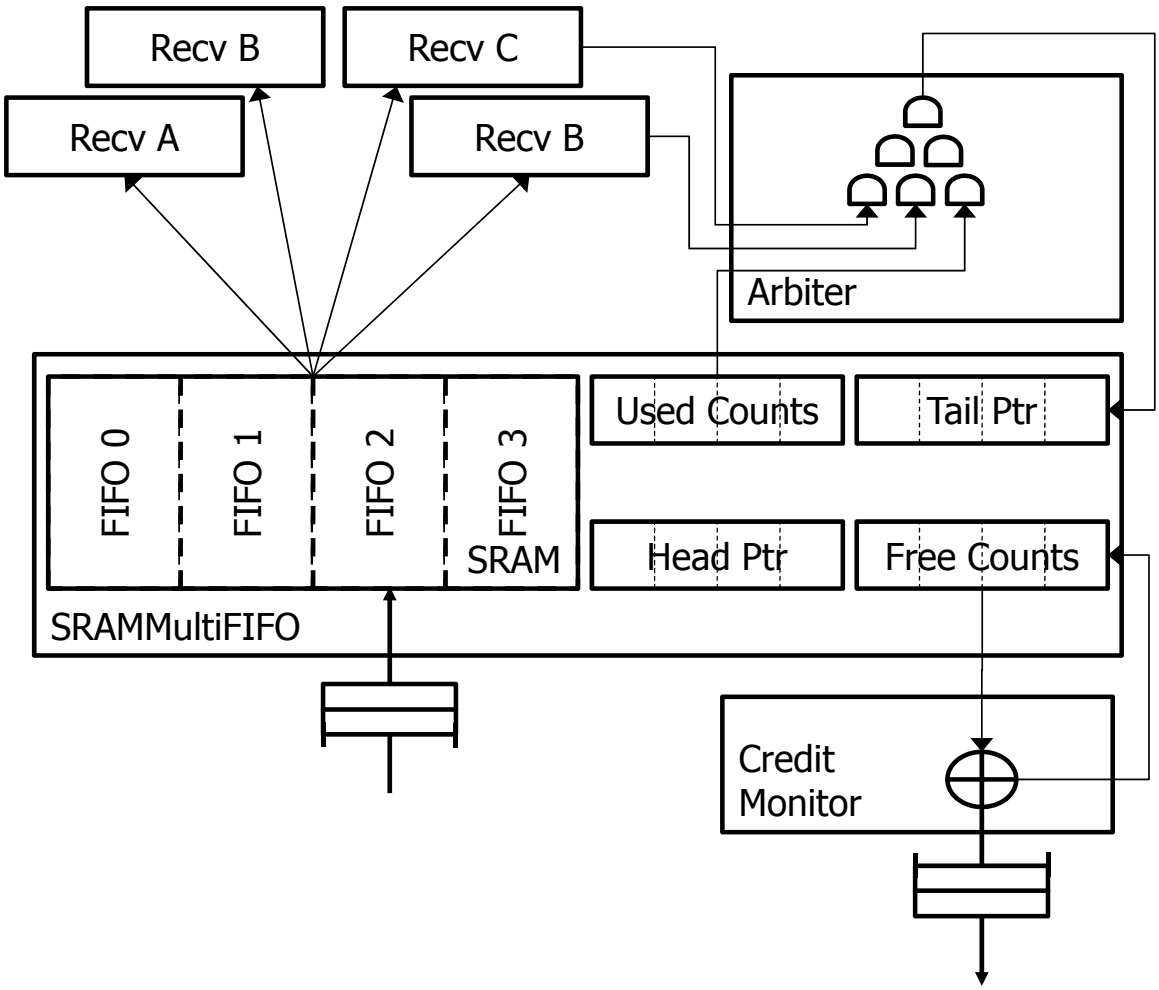


Figure 5-6: FIFOs are folded onto a single logical SRAM resource. Each FIFO in the SRAM represents a buffer for a single virtual channel

	LUTS	Registers	BRAM	Relative Performance
Registered FIFOs, depth 8	10001	22248	0	1
Registered FIFOs, depth 32	25494	68813	0	1.11
SRAM Multi-FIFOs, depth 32	4996	4778	2	1.09
SRAM Multi-FIFOs, depth 128	5225	4850	8	1.11

Table 5.1: Synthesis and performance metrics for various router architectures. Results were produced by mapping a simple HAsim dual core processor model to two FPGAs. In this design, 29 individual channels and 1312 bits cross the inter-FPGA boundary.

nication network are designed. Unlike processor network on chips, which multiplex virtual channels and offer extremely limited in-network buffering to conserve area, SMF based routers can liberally allocate virtual channels to each connection traversing the inter-FPGA channel without a significant area or performance penalty. As a result, issues related to shared virtual channels [36] do not apply to the routers and routing protocols built around the SMF. Because SMF provides deep buffers, each flow-controlled inter-chip channel can sustain the full interconnect bandwidth across high-latency physical interconnects. Deep buffers also reduce flow control traffic, minimizing throughput loss.

Routers use a simple block-update flow control scheme. Each virtual channel source keeps a conservative count of the number of free buffer spaces available at the corresponding virtual channel sink. Each time a packet it sent, this count is decremented. The virtual channel sink maintains a count of the free buffer space available, which is updated as user logic drains data out of the virtual channel. When this free space counter passes a threshold, it is set to zero and a bulk credit message is sent to the virtual channel source. These credit messages are given priority over other latency-insensitive channels to improve throughput.

5.4 Routing In Parallel: Multiple Lane Routers

With the channel marshalling and the SMF-based routing layers described in the previous sections, the LIM compiler can construct fully functional inter-FPGA routers.

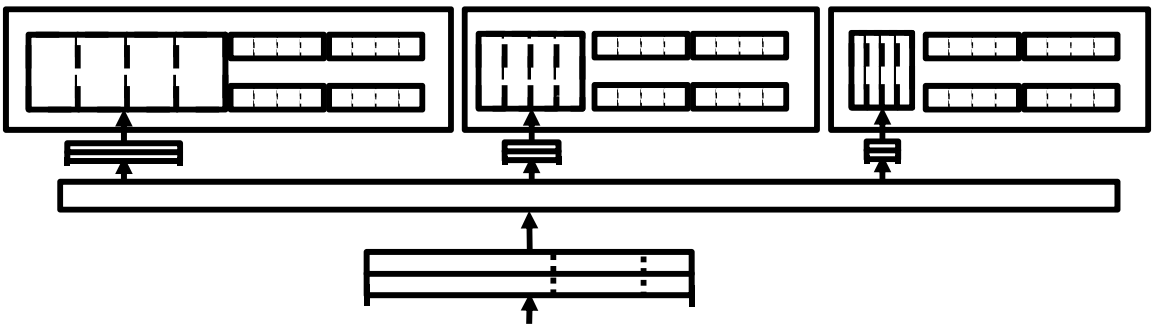


Figure 5-7: De-merger microarchitecture. Each lane is steered to a separate SRAM-MultiFIFO

To build such a single-lane router, the LIM compiler simply instantiates an SMF and ties all inter-FPGA channels to that that single SMF. This single-lane router is sufficient to build any inter-FPGA network required by the LIM compiler. However, the physical properties of FPGA systems and the properties of latency-insensitive programs suggest that the naive single-lane router architecture will be suboptimal in most cases.

Modern inter-FPGA interconnects tend to have high-bandwidth and switch at very high frequencies, particularly if the interconnect primitive is implemented in silicon. As compared to the interconnect, which achieves ASIC-like performance, user programs switch at relatively low frequency. Thus, to take advantage of the full bandwidth offered by the inter-FPGA interconnect, the interconnect interface must be widened, in terms of bits, until its bandwidth in the slow program clock domain is matched to the physical interconnect bandwidth. For example, consider the ACP platform, described in Section 7.5. The HAsim models of Chapter 9 clock no faster than 80MHz on the ACP FPGA platform, described in Chapter 7. The low-voltage differential signaling (LVDS) inter-FPGA channel provided by the ACP is 256 bits wide and clocks at 200MHz. Thus, to match the bandwidth of the interconnect, the router interface must be widened to at least 512 bits in the program clock domain. The ACP interconnect is implemented in the FPGA fabric, but for the silicon interconnects implemented in recent FPGAs, the device interface must be widened by a multiple of four or five to deal with the clock ratio differential.

Inter-FPGA interconnects exposes a wide interface, but this bandwidth is wasted

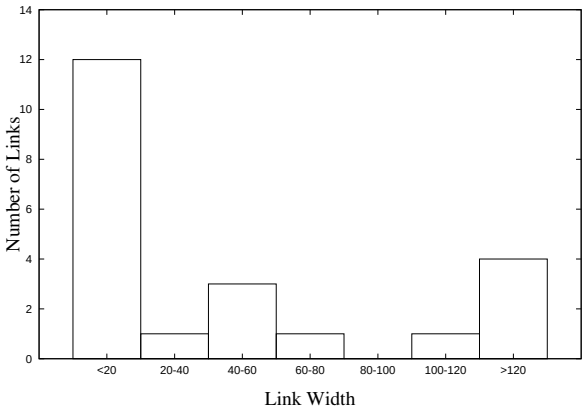


Figure 5-8: Widths of inter-FPGA channels in a two-FPGA partitioning of HAsim.

if all the exposed bits are not used to transport useful data in each cycle. Thus, the router infrastructure built on top of the physical interconnect must be tuned to the behavior of the user program channels to minimize bandwidth wastage. If all the channels in a program are wide, then the naive, single-lane router architecture is sufficient to maximize performance. However for a typical program like HAsim, most latency-insensitive channels are narrow, as shown in Figure 5-8. All of the channels are much narrower than typical inter-FPGA interconnect interfaces, and most channels are less than 20 bits wide. Not only do narrow channels occur frequently, but also, they bear most of the traffic in the design, as show in Figure 5-9. Thus, for HAsim, a single-lane router architecture is likely to be suboptimal. Of course, HAsim is a single example of a hardware design and is not necessarily representative of all programs of interest. However, narrow channels are common in hardware designs, since engineers try to economize for data sizes to improve design metrics, like area and frequency.

Since latency-insensitive channels are generally narrow and the inter-FPGA interconnect is wide, it is clear that a good router implementation must allow multiple channels to transmit data simultaneously on parallel lanes. The simplest framing of this problem is, for a m channels and a router with n lanes, to select n channels and transmit their data on the n parallel lanes of the router. Indeed, a single lane router produced by the LIM compiler is a base case of this problem in which $n = 1$. If n is greater than one, then a crossbar is needed to select among the m lanes.

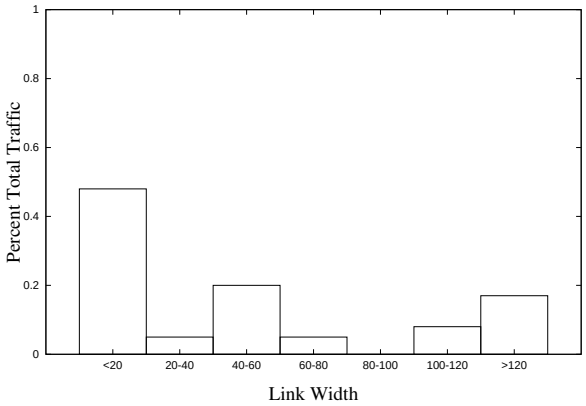


Figure 5-9: Activity of inter-FPGA channels in a two-FPGA partitioning of HAsim.

A crossbar router is optimal in terms of bandwidth usage: all router lanes are guaranteed to be filled, if there are enough channels with data. The difficulty in building a crossbar is that the crossbar selection network scales at least linearly with the number of lanes that must be allocated. For wide inter-FPGA interconnects, there may be as many as ten lanes, making a crossbar prohibitively expensive to implement. Not only is the crossbar costly to implement, it is likely to be wasteful of resources: most channels do not have much traffic. Figure 5-10 shows a traffic distribution for HAsim, in which the four most-heavily-loaded channels account for more than 80% of all inter-FPGA traffic.

Instead of synthesizing a costly crossbar to obtain a parallel router implementation, the LIM compiler synthesizes routers consisting of several independent lanes to which channels are statically assigned at compile time. Each lane has a set of dedicated channel marshalers and an arbiter, which together produce some packetized, fixed-width data format specific to that lane, using the packetization schemes discussed in the previous section. These lane packet streams are merged into a single transmission bundle for physical transport. At the merger, each lane is extend to include a valid bit denoting whether the lane contains valid information in the particular transmission bundle. Bundles are injected into the physical layer on demand, when at least one lane has some data to send.

On the ingress side, the corresponding de-merger, shown in Figure 5-7, takes the bundle and steers the data and the valid bit for each lane into a FIFO. In the second

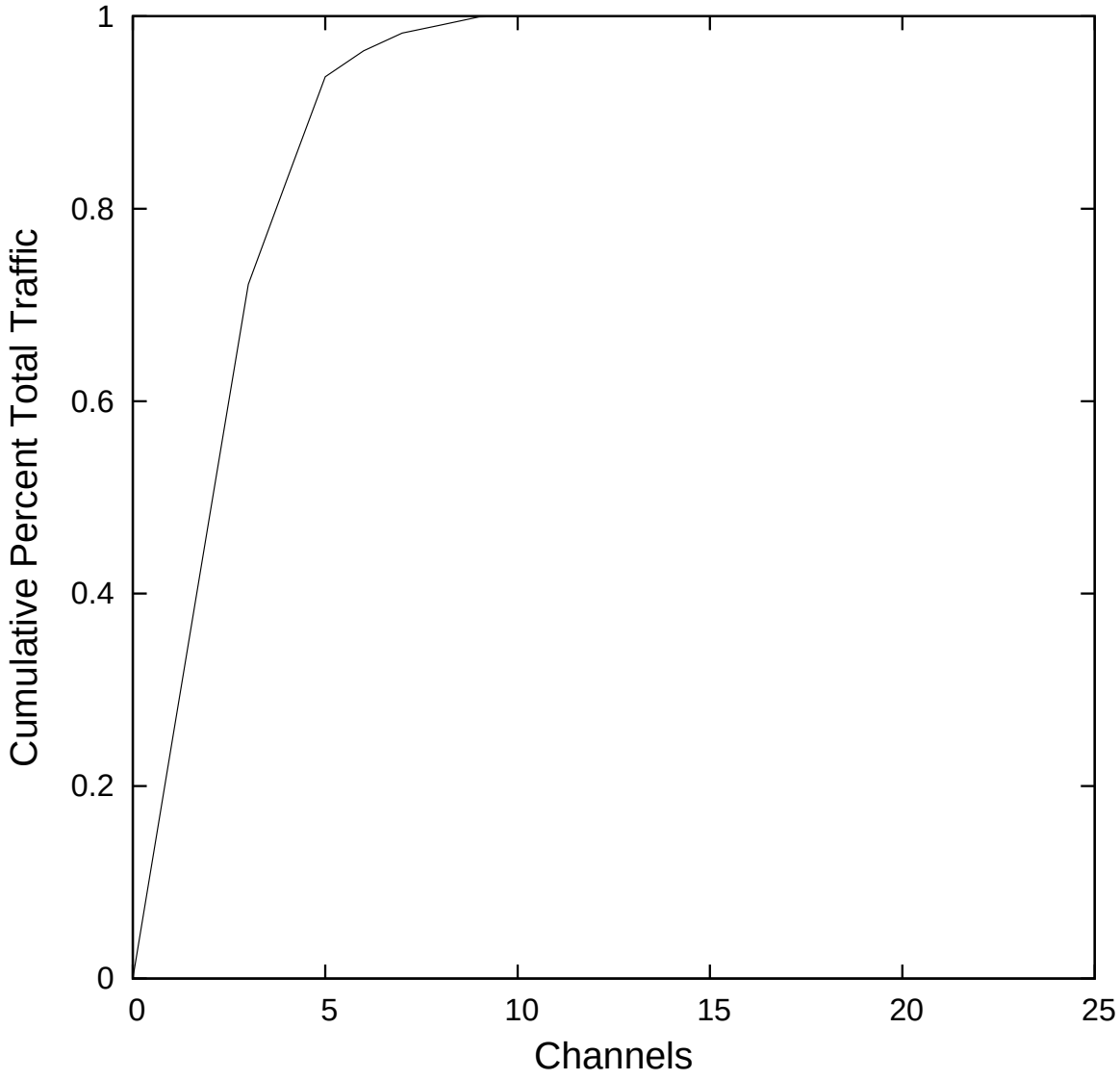


Figure 5-10: A cumulative distribution of the channel traffic loads in HAsim. HAsim is dominated by a handful of heavily loaded channels.

cycle, the valid bit is examined, and data is either passed up to the channel routing layer or dropped if it is invalid. Buffering in the de-merger is systolic. The intention in providing a separate FIFO is to ensure that separate lanes do not block each other, which can result in bandwidth loss. Note that if the lanes did block each other, performance would be lost, but correctness would still be maintained because higher level flow control guarantees the eventual drainage of all data values injected into the network.

Lanes provide a degree of parallelism in the network architecture, but the SMF buffer is only able to service a single word per cycle. To resolve the SMF bottleneck, each lane is provisioned with an independent SMF for its channels. Thus, lanes can operate independently and are fully parallel.

Counter-intuitively, multiple-lane routers, even routers with unrealistically large numbers of lanes, are actually similar in size to single-lane routers. The chief costs, in terms of FPGA area, of SMF router implementation are the combinational structures needed to manage the channel flow control, to fan-in the channel data paths, and to arbitrate the external interconnect. These structures scale log-linearly in size and delay with the number of channels tied to the the router. Figure 5-11 shows an example of a router arbiter: the channel state is linear in the number of channels, but the priority selection network scales logarithmically with in area and delay with the number of channels. By implementing multiple lanes to route a fixed number of channels, the logarithmic term in area and delay can be reduced, while the linear term is distributed among all the lanes. Table 5.2 shows the implementation area of various router configurations. Introducing many lanes increases the area utilization of the router infrastructure by only 10%, or less than 1% of the total FPGA area.

Much of the motivation for implementing routers with multiple lanes involves improving the performance of narrow channels, since they are the common case in many programs. However, narrow lanes do not necessarily reduce the performance of wide channels: rather than serialize on a single lane, wide latency-insensitive channels may be partitioned into a set of narrow, but parallel channels and allocated separately across several narrow lanes. This grants wide channels dynamic access to

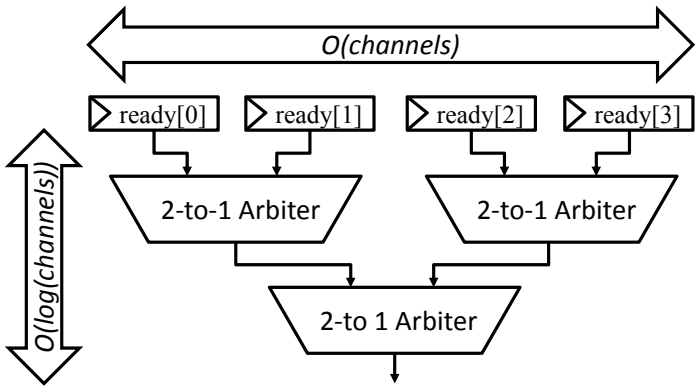


Figure 5-11: An example of log-linear scaling in the synthesized routers. Some logic is required for each channel, but some logic scales logarithmically with the number of channels.

	LUTS	**Registers**	**BRAM**
1 lane	20509	20607	41
2 lanes	21162	20792	30
3 lanes	21240	20957	20
4 lanes	21785	22151	21
8 lanes	22421	23587	20

Table 5.2: Synthesis metrics for various router configurations. Each router services 40 channels, which are evenly divided among the router lanes. Xilinx 14.1 was used to produce bit-files. Only the number of lanes and the channel-allocation were varied.

the full bandwidth of the interconnect, while allowing narrow channels to share the interconnect.

The routers produced by the LIM compiler may have many parallel lanes. However, the number of lanes and the static allocation of channels to lanes are important and program-specific synthesis problems. Chapter 6 will explore algorithms for automatically selecting good router implementations based on the behavior and properties of specific programs.

5.5 Physical Interconnect

The physical interconnect layer consists of specially annotated physical devices which carry data between platforms. The description of these interconnects is included in

the syntax of the environment description file described in Section 4.2.2.

The requirements placed on the physical interconnect for interoperability with the synthesized network are minimal. They must provide guaranteed, in-order delivery of messages, channel-level flow control signals, and a self-initialization mechanism. Given the satisfaction of these requirements, the backing implementation of the interconnect is irrelevant from the network's perspective and could range from LVDS to Ethernet.

An important consequence of the guaranteed delivery requirement placed on the platform interconnect is the creation of an automatic, distributed initialization for the entire multiple FPGA program: a message injected into the inter-platform interconnect must be delivered, and so interconnects cannot admit messages until they are themselves initialized. As platforms come online, their interconnects self-initialize and their modules begin executing. Modules continue executing until they stall waiting for communications to or from an uninitialized interconnect. Because distributed initialization is implicit in the local initialization of the physical interconnect, routers need not provide global agreement on when it is safe to begin transmitting data.

5.6 Router Instrumentation

For instrumentation purposes, the synthesized routers can be automatically augmented with statistics collection facilities. These statistics are automatically connected to the statistics service provided by the LEAP FPGA operating system. Statistics instrument both the individual channels and the router infrastructure, including bandwidth and occupancy information, providing valuable insight into both program behavior and the operation of the synthesized network. Indeed, some of the feedback-driven algorithms described in Chapter 6 make use of these statics.

Chapter 6

Compiler Optimizations

Chapter 4 described a flow for compiling designs described using latency-insensitive channels. One of the chief activities of the LIM compiler is the automatic synthesis of a network capable of carrying inter-module latency-insensitive channels between FPGAs. This chapter will explore how the *properties* of latency-insensitive programs can be leveraged to improve the performance of the synthesized inter-FPGA network. The techniques presented in this chapter focus on local optimizations: point-to-point channels and single inter-FPGA links. Global optimizations, such as optimal link routing between FPGAs, are important avenues of research, but are left for future work.

The goal of the optimizations presented in this chapter is to maximize the goodput, the total amount of useful data, carried between a pair of FPGAs. The preceding chapters discussed the synthesis of the inter-FPGA network at a high level, but several details of the parameterization of this network were omitted. For example, Chapter 5 described the architecture of a router with multiple, parallel lanes. However, neither how the number or the width of the lanes are chosen nor how latency-insensitive channels are assigned to the lanes of the router was explained. Section 6.2 will examine how the compiler maximizes goodput by making intelligent choices for these parameters. Section 6.3 will examine a different tactic for maximizing goodput: type-specific compression. Compression improves overall system goodput by removing data from the network, decreasing overall network utilization.

The optimizations presented in this chapter are static, that is, they are determined at compile time. Some optimization-enabling properties of channels are also static, for example the type and width of the channel. However, the dynamic behavior of latency-insensitive channels can also be leveraged to improve network synthesis. Dynamic properties, such as channel utilization can be obtained by inserting statistic collectors into the synthesized inter-FPGA network and observing its behavior at runtime. Dynamic information sources imply *feedback-driven* compilation, since they must be collected from a functional, pre-existing implementation of the same design.

HAsim-style processor models, which will be described in detail in Chapter 9, illustrate the need for feedback-driven optimization. HAsim is characterized by tight feedback loops between modules, making its performance very sensitive to inter-module communications latencies and thus an excellent target for network optimization. Consider the case of two channels crossing between a pair FPGAs. One channel carries information from the simulated processor's decode stage. It has a narrow bit-width, but high utilization. The other channel is the memory interface between the last-level cache and backing main memory. This channel is hundreds of bits wide. However, because programs generally have good locality the utilization of this channel is very low. Clearly, any network carrying these channels should be optimized to ensure that the high-utilization channel is given priority over the low-utilization channel. However, the behavior of both of these channels is highly input-dependent. Thus, for the LIM compiler to automatically discover this property and build an optimized network, it must build an instance of HAsim with enough instrumentation to discover channel utilization, run this instance, and then compile a new, optimized version of HAsim using the feedback derived from the earlier implementation.

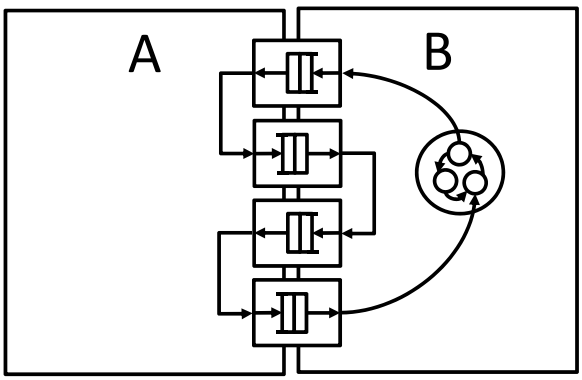


Figure 6-1: Inter-FPGA loop test. A and B are two latency-insensitive modules.

6.1 Loops: A Micro-kernel for Optimization Evaluation

The behavior of optimizations when applied to real programs can be difficult to predict. Thus, a good test case is needed to verify that optimizations function as expected. The transformations presented in this chapter largely focus on the optimization of the inter-FPGA network synthesized by the LIM compiler. The loops test, shown in Figure 6-1, is a simple design for testing such optimizations: latency-insensitive channels go back and forth between a pair of modules on two FPGAs, creating a long, data-dependent pipeline.

Despite its simplicity, the loops test exhibits a number of useful features. First, it scales easily, by parameter, to large numbers of inter-FPGA crossings. This scaling gives a knob by which to generate varying degrees of pressure on the various components of the inter-FPGA communications complex. Second, the highly predictable behavior of the loops program make performance bugs easier to detect and fix. Third, the loops test should be relatively easy to optimize - the connections are plentiful, independent, and largely uniform. Finally, because it is so simple, its performance under optimization is easy to predict and to understand. As such, it is helpful in the early evaluation and validation of optimizations to the inter-FPGA network. In the following discussion of optimizations, the loops test will be used as both a means to gather intuition about the behavior of the optimizations and as a limit study for evalu-

ating the maximum practical performance improvement for each of the optimizations presented.

6.2 Lane-Sizing and Channel-Allocation

The preceding chapters discussed the construction of the inter-FPGA network. The most important element of this network was an inter-FPGA router with multiple, statically parameterized lanes. Details of the parameterization of this router were omitted in the discussion of network synthesis. In particular, two decision problems, the number and the width of the lanes and the assignment of latency-insensitive channels to the lanes of the router, were deferred. The former problem, lane-sizing, and the latter problem, channel-allocation, will be considered jointly in this section as they are deeply related. Indeed, lane-sizing ultimately determines channel-allocation, since the channels themselves have have both variable width and variable levels of traffic. However, there are many choices of lane sizes, and the ultimate measure of goodness for each lane-sizing is the quality of the channel-allocation obtained. Thus, a good solution to either problem must consider both problems simultaneously. This section will evaluate a number

6.2.1 Complexity of Lane-Sizing and Channel-Allocation

The goal of lane-sizing and channel-allocation, in the abstract, is to minimize program runtime by maximizing inter-FPGA network throughput. However, it is difficult to reason statically about the performance of specific lane-sizings and channel-allocations due to complex timing dependencies between channels that may exist in a user program. For example, two channels may always be simultaneously active. Assigning these channels to the same lane is clearly bad for throughput, but the relationship of the channels is difficult to derive, if only coarse-grained feedback measurements are available.

The sizing and allocation problems can be simplified by ignoring all correlation between channels and simply considering the relative loadings of channels. The goal

of this simplified problem is to minimize the load of the maximally-loaded lane, the *minimax* lane. Though semantically precise, even this problem is still computationally difficult. Under this formulation, the channel assignment is similar to the well known multiprocessor scheduling problem, while the lane-sizing problem resembles integer bin-packing. Taken together, the two problems strongly resemble to the stock-cutting problem [60]. Unfortunately, all of these problems are NP-complete, as are lane-sizing and channel-allocation.

If the lane sizes are known, the channel-allocation problem is NP-complete by reduction from the multiprocessor scheduling problem. The multiprocessor scheduling problem involves running a set of processes, each with a statically known runtime, across m symmetric processors. The objective is to minimize the time until the last process finishes.

The multiprocessor scheduling problem can be solved using a channel assignment algorithm, by creating a lane allocation with M lanes of single-bit width, representing the M machines to be scheduled. The processes to be scheduled are represented as single-bit channels with loads corresponding to the run-time of the process. An algorithm solving this channel-allocation problem will produce a solution that represents a run-time minimizing assignment of processes to processors. Thus, through this constant-time transform, a processor-scheduling problem can be converted into a channel-allocation problem and solved using any algorithm for channel-allocation. Therefore, because a polynomial time solution of channel-allocation would imply a polynomial solution to the known NP-complete process scheduling problem, the channel-allocation problem must itself be NP-complete.

The combined lane-sizing and channel-allocation problem is also NP complete, again by reduction to the processor-scheduling problem. As in the previous proof, processes are converted into single-bit wide channels with loading factor equal to their runtime and the interconnect size is set to M. This formulation is fed into an algorithm for solving the combined lane-sizing and channel-allocation. The key difference in the solution returned by the combined algorithm and that returned by the channel-allocation solver alone is that some lanes may have width larger than one.

These supernumerary bits correspond to unused processors in the process assignment.

When considered together, the lane-sizing and channel-allocation problem form a joint optimization problem related to the NP-complete one-dimensional stock-cutting problem [60]. The typical formulation of the cutting-stock problem involves cutting a number of variously sized smaller rolls of paper from uniformly sized larger rolls of paper while minimizing waste in the form of scrap paper. In the general formulation of the cutting stock problem, the points at which the large roll is cut, the "knife-positions," can change as often as is necessary to obtain an optimal solution. However, in the one-dimension version of the problem, only one choice of knife allocation is allowed. This problem has a clear relationship to the static formulation of the lane sizing and channel-allocation. The wide inter-FPGA lane corresponds to the large roll of paper, while wasted bandwidth is the analog of wasted paper in the cutting stock problem. In router construction, lane sizes, that is, the knife-edge positions, are fixed at compile time.

Optimal solutions to static lane-sizing and channel-allocation are difficult to determine. Worse, optimal solutions to the static problem do not guarantee an optimal solution to the dynamic channel-allocation and lane-sizing problem. In particular, the figure of merit in the static model, *minimax* lane loading, does not capture dynamic timing and dependencies between channels, instead treating channels as independent, interchangeable entities. In cases where channels do have dependence on one another, it is possible to achieve suboptimal performance if dependent channels are assigned to the same lane. However, the results presented in Section 6.2.3 will show that *minimax* optimization does a reasonable job of optimizing real program performance.

6.2.2 Channel-Allocation and Lane-Sizing in the LIM Compiler

To facilitate experimentation with different algorithms, lane-sizing and channel-allocation are implemented as a discrete phase of the compilation process. These two phases occur together, after the channel routing step discussed in Section 4.3.3, and are

invoked per pair-wise inter-FPGA interconnect. In the current LIM compiler, lane-sizing and channel-allocation are iterative. The LIM compiler does not currently consider simultaneous solutions to the two problems, even though the relationship between lane-sizing and channel-allocation and the stock-cutting problem suggests that these problems should be considered jointly.

The lane-sizing algorithm first proposes a lane-sizing, and then that lane-sizing is used to produce a provisional channel-allocation using some channel-allocation algorithm. This provisional channel-allocation is then evaluated by comparing its *minimax* with the previous best known lane-sizing and channel-allocation, and the better solution is retained. After some non-zero number of iterations, the best solution, in terms of *minimax* is reported to the the network synthesis code for physical implementation. Although some algorithm combinations may perform many iterations in search of the best *minimax* allocation, all schemes will minimally test a few different fixed numbers of lanes.

Some of the algorithms presented in this section are *feedback driven*, in that they rely on channel loading information collected from some previous run of the program under consideration. Prior to attempting lane-sizing and channel-allocation, the compiler takes a statistics file containing run-time-generated channel loads as an optional input and uses the relative amounts of traffic on each channel to assist in determining the optimal lane-sizing and channel-allocation. Channels in the compilation are matched to the statistics file based on name. If no statistics file is provided or if the statistics file is incomplete, channels without loading information are assigned the average of all the other loads. If no loading information is given all channels are assigned the same default weight, which has the effect of generating a uniform, random assignment of channels.

There are several tactics for producing solutions to NP-complete problems. The most obvious algorithm is a brute-force search across the space of potential solutions. This approach is generally infeasible, but may be feasible if certain dimensions of the problem space are constrained. Randomized solutions are another possibility. Yet a third possibility are heuristic algorithms, in which some structure of the underlying

problem is exploited to produce a problem solution in polynomial time. The solution produced by the heuristic may be suboptimal, but generally heuristic algorithms have good performance bounds. The remainder of the section explores algorithms from each of these categories as they pertain to lane-sizing and channel-allocation.

When solving an NP-complete problem, an optimal solution generally requires a brute-force search among all possible solutions. If the cardinality of the solution space is large, then this approach is infeasible. However, for some NP-complete problems, brute force solutions are feasible if certain dimensions of the problem are suitably constrained. In the case of channel-allocation, brute force involves testing all possible mappings of channels to lanes, an exponential problem with complexity $O(l^c)$, while lane-sizing involves testing $O(w^l)$[1] lane-sizings, where c is the number of channels, w is the lane width, and l is the target number of lanes. The search-space of channel-allocation precludes brute-force solutions in all but toy programs, since real programs have, minimally, a dozen channels. However, if the inter-FPGA interconnect is narrow or if the number of lanes is small, brute-force methods can be applied to lane-sizing. Thus, the first algorithm for lane-sizing is Brute-Force. Brute-Force lane-sizing evaluates all possible combinations of lane widths for a given interconnect width and a fixed number of lanes, choosing the lane-sizing with the best *minimax* score.

Since brute-force may be computationally infeasible for some problem sizes, suboptimal tactics are necessary to achieve a solution to the sizing and allocation problems. The most basic scheme for solving the channel-allocation problem is randomized uniform allocation. In this scheme, the set of channels are randomly permuted and then assigned to lanes sequentially, balancing the number of channels assigned to each lane.

[1]The cardinality of set of possible lane sizes is deeply related to the binomial theorem. The recurrence for an upper-bound on the number of sizings is $S_{w,l} = \sum_{i=1}^{w} S_{i,l-1}$. Intuitively, this means selecting the width of one lane and then recursing to partition the remaining bits into lanes. For a fixed number of lanes, the recurrence forms a diagonal of Pascal's triangle. For three lanes and variable width, the number of sizings are the triangular numbers 1,3,6,10,...; for four lanes the number of sizings follows the tetrahedral numbers. This recurrence is, unfortunately, exponential in the number of lanes, with closed-form formula: $\frac{(w+l-1)!}{(l-1)!(w-1)!}$. If l is small, then the number of lanes-sizings may be practically tractable.

No consideration is given to static or dynamic properties of the channels in this allocation scheme. For lane-sizing, the simplest scheme is Uniform partitioning, in which the inter-FPGA interconnect is divided into an equal number of equally-sized lanes. Although these two techniques are simple and certainly suboptimal, they have one important property: they require only statically available information about the program. If, however, feedback about the dynamic behavior of the program is available, more complex approximations become available.

The reductions in the previous section demonstrated the relationship between channel-allocation and processor scheduling, a well-studied problem which has good approximate solutions. Processor scheduling has an approximate algorithm known as longest processing time (LPT) [22], which sorts the processes to be scheduled according to length and then assigns the sequentially to the processor with the least amount of assigned work. This algorithm has an optimality bound of $(\frac{4}{3} - \frac{1}{3m})OPT$, where m is the number of processors. For small m of the kind that are typically seen in router allocations, the LPT algorithm gives a very good bound on an optimal allocation. The application of LPT heuristic to channel allocation is straightforward, but requires information about the loading of each channel which must be obtained from a previous run or simulation of the target design. To implement LPT channel-allocation, the channels are sorted according to their traffic levels and then assigned, in sequence, to the least loaded lane. The LPT heuristic can also be applied to lane-sizing: the channels are sorted according to load factor and the widths of the most heavily trafficked channels are chosen as the lane sizes, with the left over bits forming an extra lane. Since the heavily loaded lanes are often narrow, this approach typically results in many small lanes.

The lane-sizing and channel-allocation policies are independent of one another, permitting the full cross-product of policies to be studied. However, once load information becomes available, combinations like Random/Longest-Job-First do not make sense – sizing a lane for a specific channel and then assigning that channel to a different lane is a poor policy.

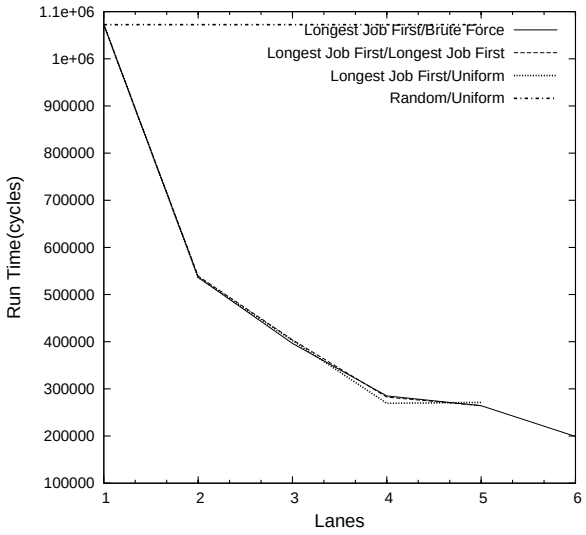


Figure 6-2: Loops test performance with various lane allocation and sizing algorithms.

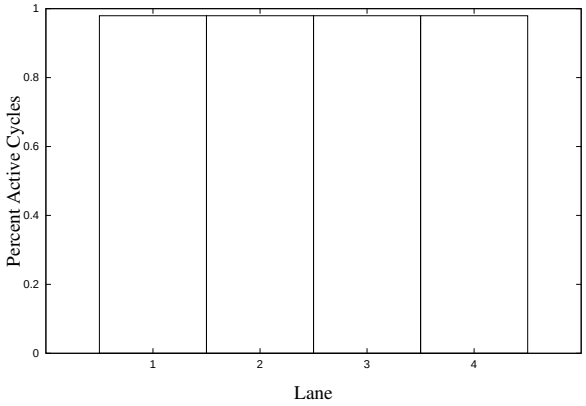


Figure 6-3: Lane loads for a Uniform lane-sizing of the loops kernel. Longest-Job-First allocation does an excellent job of load balancing.

6.2.3 Performance Results

Figure 6-2 shows the performance of the loops test when different allocation and sizing algorithms are applied. As expected, Longest-Job-First channel-allocation does a good job, even in the case of uniform lane-sizing, due to the simplicity of the test case. The kink in the graph between 4 and 5 lanes occurs because the performance of the loops test is determined by the lane with the most traffic, but because channel-allocation is discrete, one lane happens to have an extra channel assigned. Figures 6-4 and 6-3 show that the heuristic channel-allocation and partitioning schemes are able to utilizes all available inter-channel bandwidth for the simple loops kernel.

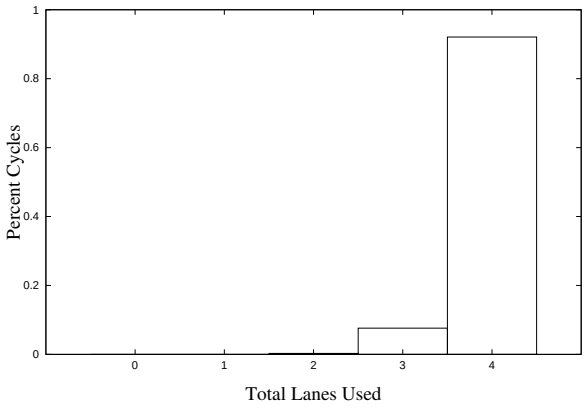


Figure 6-4: Lane parallel utilization for a Uniform lane-sizing of the loops kernel. This kernel saturates the inter-FPGA bandwidth.

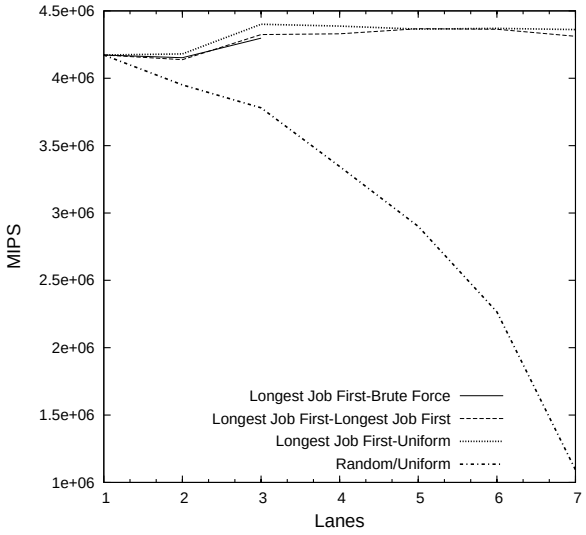


Figure 6-5: HAsim 16 core model performance with various lane allocation and sizing algorithms.

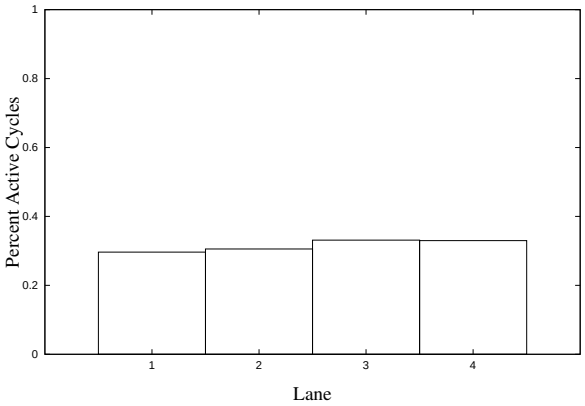


Figure 6-6: Lane loads for a Uniform lane-sizing. Longest-Job-First allocation does an excellent job of load balancing.

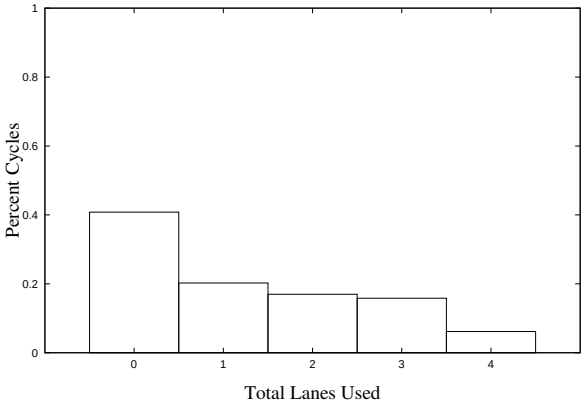


Figure 6-7: Lane parallel utilization for a Uniform lane-sizing. HAsim has limited parallelism and many dead cycles.

Programs more complicated than the loops benchmark can also benefit from multiple lane routers. Figure 6-5 shows the performance of a 16 core HAsim model when different allocation and sizing algorithms are applied. The performance improvement for HAsim is much smaller than the performance improvement for the loops test, in part because HAsim has much less traffic. Adding more lanes improves HAsim performance until the total number of lanes is larger than four, with the largest gap occurring when moving from one lane to two. As with loops, random allocation fares poorly even for a small number of lanes, since high-utilization channels may be assigned to the same lanes, inducing a performance bottleneck.

To understand why HAsim experiences limited performance improvements when provided with more parallel lanes, it is necessary to dig a little deeper into the behavior

of HAsim. In contrast to the simple loops test, which is effectively a straight pipeline capable of consuming all offered network bandwidth, HAsim has internal performance bottlenecks that serve to limit the performance of the system. Indeed, Figures 6-7 and 6-6 show that a significant fraction of cycles in HAsim have no inter-FPGA communication. This results helps explain why HAsim performance improves when scaling to a small number of lanes: there are a limited number of opportunities for parallelism in the network, and a handful of lanes are sufficient to capture most of them.

In theory, LPT lane-sizing should outperform Uniform lane-sizing, because LPT can choose-program specific lane sizes. Indeed, LPT does outperform Uniform in some cases, but only by a small margin. The reason for the lack of performance differentiation between the LPT and the Uniform lane-sizing scheme for HAsim is twofold. First, HAsim has limited opportunities for performance improvement, and these are easy to obtain. Second, in practice, the routers the two algorithms generate have similar lane sizes. LPT typically chooses lanes sizes that are somewhat narrow, since the heavily trafficked links are also narrow. However, beyond a two or three lanes, uniform lanes are also narrow.

One of the chief concerns when facing a brute-force search is time complexity. Run-times for the Brute-Force lane-sizing algorithm on HAsim are tolerable only for very small problems. For HAsim (20 channels) and a wide interconnect (256-bits), it takes tens of hours to test all possible lane-sizings for 4 lanes, far longer than the FPGA implementation tool chain. Beyond 4 lanes, the brute-force search does not terminate in a reasonable time frame. For evaluation, this is an acceptable run time, but for deployment it is unacceptable. Moreover, as FPGA technology improves, inter-chip bandwidths will scale and larger numbers of lanes will be required to obtain full inter-chip bandwidth. Thus, Brute-Force is not a particularly good choice for production deployments of the compiler since Brute-Force produces results that are similar to the simpler algorithms.

Intelligent router parameterization by way of channel-allocation and lane-sizing can produce significant performance gains in some applications.As designs continue

to scale, both in complexity and number of FPGAs, improving network throughput will become increasingly important in producing high performance multiple FPGA implementations.

6.3 Channel Compression

The previous section discussed optimizations that improve the parallelism and, thereby, throughput of the inter-FPGA routers generated by the LIM compiler. However, these improvements in throughput are intrinsically limited: at some point, the inter-FPGA interconnects have been so finely divided that adding additional lanes actually results in performance degradation. Once the routers have been fully parallelized, one of the only options remaining to improve performance is to reduce loading in the inter-FPGA network. One approach to reducing this loading is to compress the data transported between FPGAs.

At first glance, data compression seems both costly and potentially ineffective: usual applications of compression in software involve either IP network communication or storage: slow, long-latency operations that hide the computational overhead of compression. Otherwise, in software, even the simplest compression scheme will degrade performance. For example, a FIFO communicating between two threads will typically not be compressed, even if the potential for compression is high, since the performance overhead of compression is far greater than the overhead of transporting a few extra bytes.

Unlike software, where compression can be significantly more expensive than communication, hardware compression is almost always cheaper than communication. Typically, compression algorithms involve fine-grained bit manipulations and difficult-to-predict branch behavior, both of which are expensive and slow on general purpose computers. In contrast, the fine-grained, parallel fabric of FPGAs is ideal for compression. In the FPGA, these kinds of manipulations reduce to wiring and small numbers of gates which can operate in parallel, and can often be completed in less than a single cycle. The simplicity and speed of compression in FPGAs also

implies that even schemes which obtain very poor compression, schemes which would completely destroy software performance, can result in application-level throughput improvement when applied to latency-insensitive channels. Experimental results presented in this section will demonstrate than even a few percent compression can result in a measurable throughput gain.

This section examines the automatic application of compression to inter-FPGA channels. Two general schemes for compression are presented. The first scheme leverages static type analysis to automatically synthesize channel compression schemes. The second scheme enables programmers to specify their own type-specific compression algorithms and integrate them into the LIM compiler.

Compiler assistance is necessary in the application of compression schemes, as compression should only be applied to those latency-insensitive channels that cross between FPGAs. For channels between modules located on the same FPGA, RTL FIFOs are both the cheapest and highest performance implementation of latency-insensitive channels. Instantiating a compression scheme on such a channel not only wastes resources, but also degrades performance. The problem, then, with programmer instantiation of compression is that the programmer cannot necessarily predict which channels cross FPGA boundaries. In the case that a compressed channel does not cross an FPGA boundary, the compiler cannot remove programmer-instantiated compression scheme, resulting in a performance penalty.

The schemes presented in this section are fully automated. If the compiler determines that a channel is compressible and that channel crosses an FPGA boundary, then the compiler will automatically instantiate compression hardware for that channel as an augmentation to the inter-FPGA router. Otherwise, the compiler will default to the normal implementation hardware for the channel.

Some of the optimizations and experiments in this section were conducted in collaboration with Michael Adler.

```
typedef union tagged {
    void   Invalid;
    data_t Valid;
} Maybe#(type t);
```

Figure 6-8: Syntax and bit representation for `Maybe` type.

6.3.1 Tagged Unions

In general, the LIM compiler treats channel types as opaque and always transmits the all bits of the type, even if some bits are semantically invalid. If many bits of particular data word are not semantically valid, then transmitting the whole word is wasteful. Modern HDLs [4], [37] support `tagged Union`s, which extend the classical C `union` construct with a tag field so that the type of the value stored in the payload may be determined. Because the `tagged Union` construct explicitly carries information about the type its payload, compression schemes for `tagged Unions` can be synthesized automatically.

In hardware designs, `tagged Unions` often have differently sized members. For example, the commonly used `Maybe` type, shown in Figure 6-8 has two members: `Valid` can be quite wide, while `Invalid` is always a single bit. If the `Valid` leg is sufficiently wide, then an inter-FPGA channel carrying the type must use either multiple lanes to transport the data or serialize the transmission. In either case, `Invalid` values needlessly occupy network resources that could be used to transport useful data. If `Invalid` values are more common than `Valid` values, resource waste can be significant.

Fortunately, `tagged Union` types can be compressed automatically by the compiler. There are two possible transformations for `tagged Union` depending on whether the `tagged Union` channel is split across multiple lanes. In the case of a single lane, the compiler can adjust the length of the marshalled packet depending on the type of the `tagged Union` member being sent in the packet. For narrow `tagged Union`

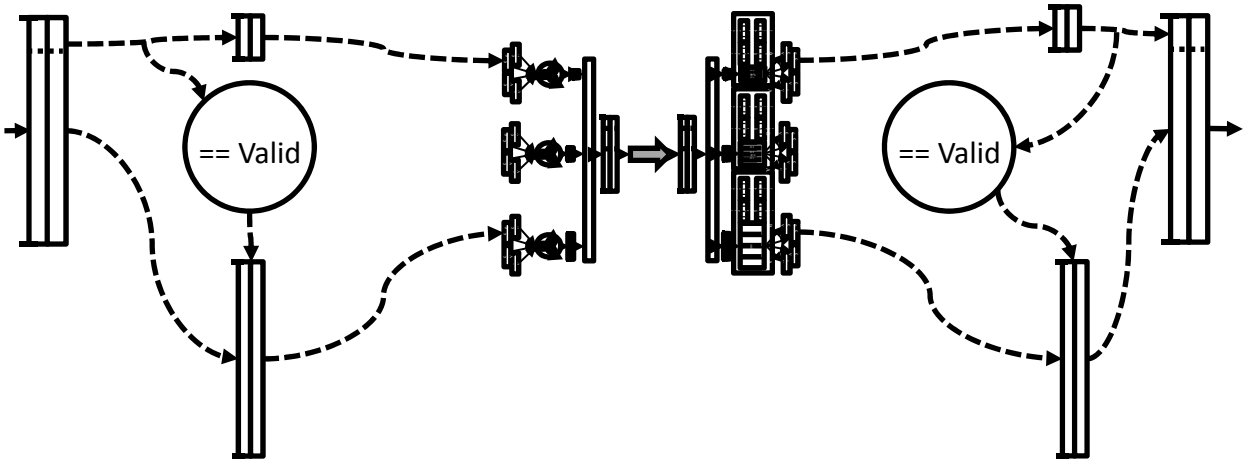


Figure 6-9: An automatically generated compression scheme for the `Maybe` type. The `Maybe` is transformed into two separate latency-insensitive channels, the tag and body. The body channel is only enqueued if the tag is `Valid`. The two channels tie in to the inter-FPGA router just like any other channels.

members, fewer packet chunks must be sent. Alternatively, the compiler may split the `tagged Union` into physically separate channels. The advantage of this splitting is that the generated channels may be mapped onto distinct lanes, preserving parallelism in the case of wide `tagged Union` members. Narrow members, on the other hand, may inject messages on fewer of the generated channels, thereby saving bandwidth. To effect this kind of type compression, the LIM compiler directly modifies the post-placement LIM graph: the compressed latency-insensitive channel edge is replaced by a hyper-edge representing the compiler-generated compression channels. These generated channels are no different than the channels expressed in the user program, and thus do not affect the existing network synthesis phase of the LIM compiler. An example of a generated compression scheme for the `Maybe` type is shown in Figure 6-9. Here, the wide payload type is transmitted only in the case of a `Valid`.

Programmers may not always express the types communicated on latency insensitive channels using `tagged Union`s as a top-level type, even if `tagged Unions` appear within the type communicated. Figure 6-10 shows an example of this case, drawn from HAsim. The `BTB_Resp` type is a structure consisting of a wide `Maybe` type and a small CPU tag. Since branches are relatively uncommon compared to other instruc-

```
typedef struct {
    CPU_ID Id;
    Maybe#(Bit#(32)) Target;
} BTB_Resp#(type t);
```

Figure 6-10: User defined types may be automatically modified by the compiler to expose opportunities for compression.

tion types, the typical value for this structure is `Invalid`, and compression would be extremely useful in this case. To capture this type usage, the compiler supports automatic hoisting of `tagged Unions` over `struct` types. In this case, the compiler simply converts `BTB_Resp` to a `Maybe` of two `struct`s, each of which has a CPU tag at the router. This new type can be compressed using either of the `tagged Union` compression schemes described above. `Tagged Union` hoisting is applied only within the synthesized network, so as not to interfere with user program behavior.

In terms of area and performance, `tagged Union` compression is extremely simple: it involves only a few gates and gate delays over the uncompressed implementation. Thus, it not only composes well with other forms of compression, but requires no extra utility analysis to apply: all `tagged Unions` of sufficient width are so compressed.

6.3.2 User-specified Compression

Static, type-based compression is useful for `tagged Union`s, but for other types the compiler cannot generate compression schemes because the compiler has no way of reasoning about the semantic meaning of types. To optimize non-`tagged Union` types, the compiler requires programmer assistance. Consider, for example, the `Maybe` type discussed in the previous section. In many cases, the `Invalid` leg of the `Maybe` type will be more common than the `Valid` leg. For example, the HAsim A-Ports of Chapter 9, which are used to convey timing information in processor models are

effectively a `Maybe` type. However, because most parts of the timing model, i.e. the caches and network, are inactive for most model cycles, `tagged Invalid` is extremely common. One possible optimization for this channel, then, is to apply run-length compression to the `Invalid` leg. Rather than sending each `Invalid` separately, back-to-back `Invalid` messages may be aggregated and transmitted en-block, dramatically reducing the amount of network traffic at the cost of of slightly increasing channel latency. In the case of the `Maybe` type for A-Ports, the programmer has specific knowledge of the behavior of A-Ports latency-insensitive channels: the channels have imbalanced token traffic and are latency-tolerant, due to the time-multiplexed simulation in HAsim. Compression is possible for many channel types, but the compiler is not able to leverage this high-level behavior automatically.

To enable type-specific channel compression schemes, the programmer needs an interface into the LIM compiler by which these schemes may be specified. To provide this interface, the LIM compiler leverages Bluespec typeclasses, which allow the user to specify a set functions and modules associated with a particular type. To leverage the compression capabilities of the LIM compiler, programmers must provide an instance of the `Compress` typeclass in their source code for each type that they wish to compress. The `Compress` typeclass consists of a pair of straightforward modules: `mkCompressor` and `mkDecompressor`, with interfaces shown in Figure 6-11.

The LIM compiler is able to test for the existence of `Compress` typeclass instances for specific data types at compile time and will automatically instantiate compressors for those inter-FPGA channel types that implement the typeclass. Compression is implemented solely at the discretion of the compiler, and channels with types that do not implement the `Compress` typeclass or that connect modules co-located on the same FPGA are not compressed. The compiler supports full type polymorphism on definitions of the `Compress` class. For example, the `Compress` instance for the `Maybe` can support any payload type. At compile time, the types of the latency-insensitive channels, which must be concrete, are unified by the LIM compiler against an aggregation of potentially-polymorphic instances of the `Compress` typeclass that have been collected from the user source. If the channel type is successfully unified against

```
interface COMPRESSOR#(type t_DATA, type t_ENC_DATA);
    method Action enq(t_DATA val);
    method Bool notFull();

    method t_ENC_DATA first();
    method Action deq();
    method Bool notEmpty();
endinterface


interface DECOMPRESSOR#(type t_DATA, type t_ENC_DATA);
    method Action enq(t_ENC_DATA cval);
    method Bool notFull();

    method t_DATA first();
    method Action deq();
    method Bool notEmpty();
endinterface
```

Figure 6-11: Interfaces for compression modules.

some instance of the `Compress` typeclass, the compiler will instantiate a compressor.

User-specified channel compression schemes are composable, both with other user-specified compression algorithms and with the `Tagged Union` compression scheme above. The `Maybe` run-length compression scheme itself produces a `Tagged Union` type, which, for some widths, can be compressed using the scheme of the previous section. To achieve composable compression, the compiler could iterate over the compressed channels, applying any newly enabled compression schemes, until a fixed-point is reached.

6.3.3 Performance Results

Figure 6-12 shows an application of run-length compression to several instances of the loops micro-kernel with varying numbers of inter-FPGA channels. In this test, the channels crossing between FPGAs have been converted to the `Maybe` type. The compiler automatically discovers and applies a simple run-length compression scheme

on the `Maybe` channels. In this scheme, adjacent `tagged Invalid` values are collapsed into a single data token. The loops test is bandwidth dominated, as shown in Figure 6-3, resulting in dramatic performance increases as the probability of `tagged Invalid` tokens increases. In the case of heavy inter-FPGA loads, compression can result in as much as a 425% performance improvement.

Channel compression is most useful when the inter-FPGA channels are heavily loaded. Under extremely light loads, compression may even degrade performance, since it adds latency to communication. In the loops test, when there are relatively few lanes crossing the inter-FPGA boundary, the performance gains due to channel compression are much less pronounced, but still measurable. In the case of four inter-FPGA loops, there is enough bandwidth between the FPGAs to service all channels in parallel. However, even in this case, channel compression provides some benefit.

Since compressors must be implemented in hardware, they must necessarily be finite in size. The `Maybe` compressor in this example can collapse at most seven consecutive `tagged Invalid` tokens, limiting the maximum throughput gains. Compressors may also add latency to some tokens: the `Maybe` compressor waits for several cycles before issuing a compressed token in the hope that more `tagged Invalid` tokens will arrive. In the case that the inter-FPGA network is heavily loaded, this latency is irrelevant, since tokens spend large amounts of time stalled at the network routers. In the heavily loaded case, `Maybe` compression shows a small performance improvement, even if `tagged Invalid` values are extremely rare. Figure 6-13 shows the performance of a compressed implementation of the loops benchmark relative to a non-compressed implementation for a light load. If `tagged Invalid` are uncommon, compression results in a small performance degradation.

Real applications also benefit from channel compression. HAsim channels carrying A-Ports data are the most heavily loaded connections. They must transmit data for each processor model cycle, and have around three times more traffic that the channels with the next heaviest load. Indeed, in the performance results presented in Chapter 9, the entire timing model was intentionally co-located on the same FPGA to avoid having these channels cross an FPGA boundary. Figure 6-14, which considers

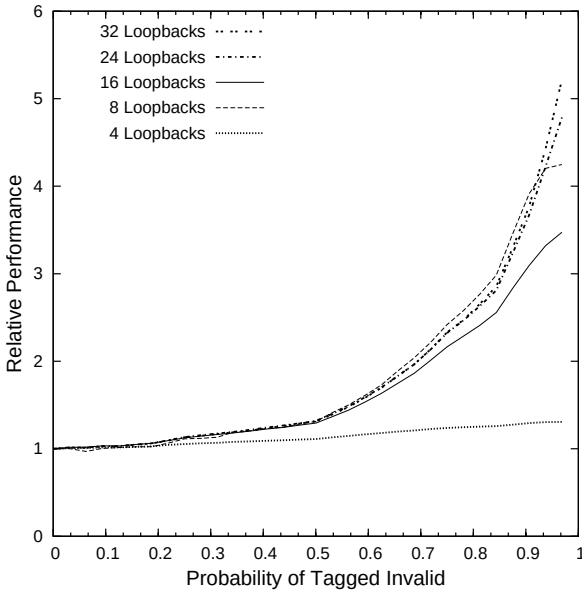


Figure 6-12: Loops test performance improves with automatic compression. Performance has been normalized to using 0% `tagged Invalid` as a baseline. Network loading scales linearly with the number of loops.

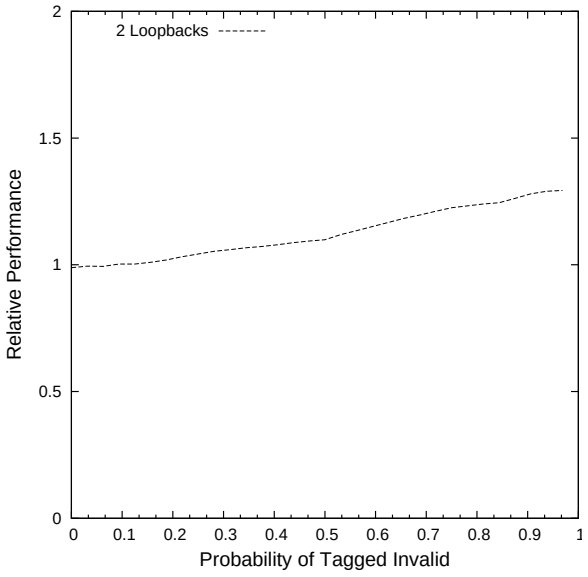


Figure 6-13: Compression schemes may reduce performance under some limited circumstances. Performance has been normalized to a non-compressed implementation of the Loops benchmarks.

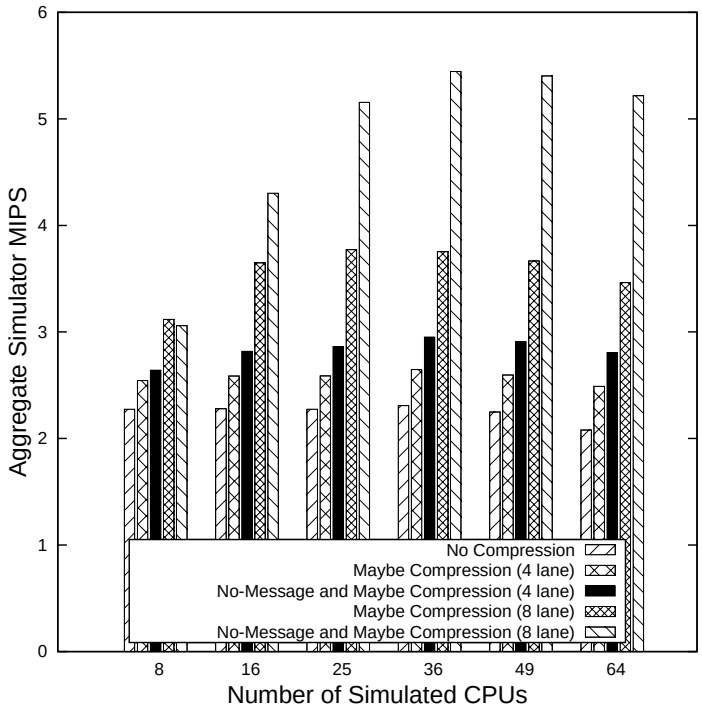


Figure 6-14: A 64-core HAsim model with A-Port compression. MIPS is a measure of the performance of the simulator, and higher MIPS means faster simulation times. Channel-allocation was performed using LJF.

HAsim implementations with timing models partitioned across FPGAs, confirms this partitioning choice: without compression, the performance of this partitioning is one-third of that obtained by placing the entire timing model on a single FPGA.

Applying channel-based compression schemes to the HAsim timing partition recovers nearly all of the performance lost due to the poor partitioning. The composition of `tagged Union` compression with run-length encoding of the `Maybe` type more than doubles the baseline performance. The broadly applicable `tagged Union` compression scheme results in a performance gains of up to 50%.

The compression schemes presented in the section are largely compatible with the lane allocation schemes of Section 6.2. Indeed, all benchmarks in this section made use of multiple lanes and the Longest-Job-First channel allocation scheme. Good channel-allocation and lane-sizing are essential in the context of automatic compression: the `tagged Union` compression scheme creates many extra, very narrow channels to save inter-FPGA bandwidth. Figure 6-15 shows the same HAsim partitioning, but with an

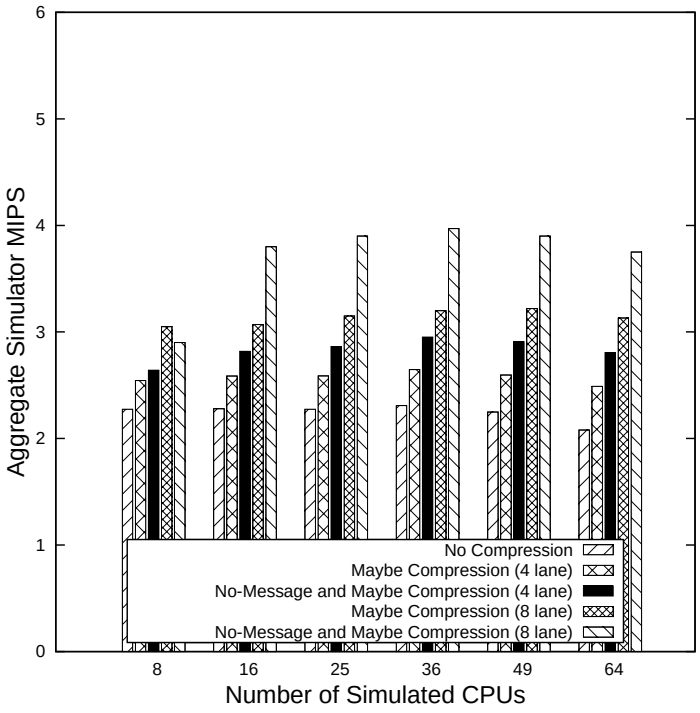


Figure 6-15: A 64-core HAsim model with A-Port compression. Channel-allocation was performed with random lane allocation.

inter-FPGA network synthesized using the suboptimal random lane allocation algorithm. As compared to the optimized implementation in Figure 6-14, the performance of the unoptimized implementation is up to 33% lower.

Channel compression is highly effective in improving the performance of latency-insensitive programs mapped to multiple FPGAs. However,the concept of channel-specific compression is not limited to the formulation of multiple FPGA compilation presented in this thesis. The compression schemes presented in this section exploit general properties common to all hardware implementations, particularly those targeting FPGAs. Thus, the approach to compression presented in this section will also have application to any simulation infrastructure for hardware designs in which data is transported over a multiplexed channel, for example SCE-MI [27] and some of the other formulations of multiple-FPGA compilation that were presented in Section 3.1. Although these infrastructures will always be performance-limited by the need to maintain cycle accuracy, compression will still be beneficial when transferring data between timing domains or between chips.

6.4 Conclusion

The optimizations presented in this chapter focused on improving the throughput of the inter-FPGA network synthesized by the LIM compiler by increasing router parallelism and reducing network traffic through compression. Many programs can benefit from these optimizations, although those programs that use the most network bandwidth to begin with will obtain the most improvement. For example, a well-placed implementation of HAsim has internal performance bottlenecks that limit its network bandwidth utilization. Thus, no matter how sophisticated the optimization, the current implementation HAsim will enjoy marginally improved performance. On the other hand, for programs with high network utilization, like loops, or the poorly-placed HAsim models considered in Figure 6-14 and Figure 6-15, the network optimizations presented in this chapter can improve performance by several hundred percent.

The implementations presented in this thesis consider only two-FPGA environments. As implementations scale to larger numbers of FPGAs, network traffic is likely to increase in aggregate, both because designs are larger and because portions of the designs are physically distant. The optimizations presented in this chapter will be extremely beneficial in such scenarios.

Chapter 7

Platform Resources

The previous group of chapters discussed the implementation of programs described in terms of latency-insensitive channels across multiple FPGAs. Most of this discussion focused on the handling of the latency-insensitive channels themselves, from their source-level syntax to the synthesis and optimization of a network for inter-module communication. The treatment of latency insensitive channels is central to the thesis; however, programs expressed in terms of latency insensitive channels still require access to external resources. For example, some of the Airblue (Chapter 8) designs need access to a radio front-end, while HAsim (Chapter 9) and H.264 (Chapter 10) both benefit from access to a large, fast memory store.

Increased resources are one of the major advantages of multiple FPGA implementation: multiple FPGAs have multiple resources. For example, each new FPGA board will have its own on-board memory, and designs using memory should benefit from this extra resource. Moreover, a good multiple FPGA compiler should automatically incorporate these new resources wherever possible, allowing latency-insensitive modules to directly use the local resources of the platform to which they are mapped.

The reality of physical devices frustrates this goal. The chief difficulty in interfacing with physical devices is that they are highly non-uniform. Each physical device has its own size, timing characteristics, and overall behavior. In a typical FPGA implementation, programmers tightly couple their design RTL to the RTL interface of a nuanced, latency-sensitive physical device, such that it becomes difficult to separate

the user program from the physical device. Such RTLs are unable to take advantage of new FPGA resources, since the consumers of the resource are not differentiated from the resource itself. Worse, any portion of the design touching devices directly becomes unportable, limiting the benefit of multiple FPGA compilation.

This chapter explores a solution to the device interface problem: abstract latency-insensitive interfaces. Abstract latency-insensitive interfaces are both general and platform independent, and it is the responsibility of the platform to provide a library supporting the interface. Programs written using these interfaces are shielded from the details of the platform upon which they are executed, enabling programs or modules of programs to move to any platform providing the abstract interface. The chapter begins with a discussion of interfacing to unportable wired interfaces in Section 7.1. The remainder of the chapter focuses on memory interfaces, since the performance of the physical memory subsystem is often critical in determining the performance of the overall program. The discussion begins with a simple, scalable abstraction for memory in Section 7.2 and gradually builds up from a single FPGA memory implementation in Section 7.3 to a description of a multiple FPGA memory hierarchy in Section 7.4. Although the chapter will focus primarily on memory systems, the techniques presented are applicable to other kinds of physical devices. The chapter concludes in Section 7.5 with a brief description and characterization of the various multiple FPGA platforms targeted in this thesis.

7.1 Latency Insensitive Platform Interfaces

Before outlining a scalable memory system for multiple FPGAs, it is first necessary to reconcile physical, board-level resources to latency-insensitive design. The latency-insensitive model of computation consists of modules communicating solely by way of latency-insensitive channels. However, external devices have wired interfaces which are likely to be latency sensitive and are not accommodated by this model.

To bridge the gap between the wired latency-sensitive world and the latency-insensitive domain of the compiler, Parashar et al. [42] introduce the concept of plat-

form services: a virtualized latency-insensitive abstraction layer. Services essentially give each kind of device a latency-insensitive interface described in terms of the Soft Connections of Chapter 4. The implementation of the wired device driver behind the latency-insensitive service interface may be complex and latency-sensitive, however the interface shields the user-program from these details.

Service interfaces are intentionally generic, so that the interface may capture many different physical instances of the device – the memory interface of Figure 7-1 makes no indication of whether it is backed by a DRAM or an SRAM chip, or what the size of the backing chip might be. The program simply demands a memory space, and it is the responsibility of the backing implementation to provide this abstraction. This generality enables the movement of device-using modules between FPGA platforms *without modifying module source.*

From the perspective of a program, services behave like any other latency-insensitive module, a property that the LIM compiler exploits in scaling services to multiple FPGAs. However, services are literally tethered to specific a FPGA and are not part of the user program, in much the same way that UNIX device drivers are not a part of the an application space program. Rather, services are supplied as interface libraries associated with each FPGA platform, and are included as part of the execution environment description of Section 4.2.2. Common services provided by most platforms include memory and inter-FPGA communications, but application-specific platforms may include network, wireless, or video interfaces. Most service are shared among several clients, and their implementations provide an automatically multiplexed interface. However, services need not be shared: radio-frequency (RF) front-ends are presented as a simple point-to-point link.

Services bear strong resemblance to kernel modules in a Unix system, and Parashar et. al. intended platform services to be analogs to the traditional monolithic OS. However, in the context of multiple FPGA systems, a distributed approach to service availability is necessary, in part because some platforms may have different sets of available services. To ensure program correctness in this case, it is the responsibility of the compiler to make a particular service on any platform on which it is needed.

Providing access to remote resources requires the synthesis of inter-chip networks. However, multiple FPGA services will be constructed in such a way that the LIM compiler will automatically synthesize the needed network infrastructure as part of its normal compilation flow.

7.2 An Abstraction for Memory

Memory is fundamental to programs, hardware programs included. However, the general state of memory systems built on modern FPGAs is primitive. The first task that FPGA designers typically take on when mapping a design to a new board is building a memory subsystem. This process involves instantiating some complicated vendor-provided IP, writing device-specific test benches, and debugging, often with limited visibility into the FPGA. There are two problems in this approach. First, each memory has a slightly different interface behavior, and modules touching these memories often absorb these behaviors. Second, because the user program and device interface are tightly coupled, it becomes difficult for any automated tool to differentiate the two and to assist in high-level implementation. Both of these difficulties make typical program-device interfaces unportable. Multiple clients of memory and multiple FPGAs compound these issues.

Inter-FPGA portability is essential to multiple FPGA implementation, but traditional memory interfaces and design paradigms limit portability. What, then, is a better memory interface? A better memory interface meets the following three requirements: first, the interface must be stable across all platforms; second, clients must be able to access arbitrary amounts of memory, even if the target platform does not have enough on-board memory; third, an arbitrary number of memory clients must be supported. A memory interface meeting these requirements allows seamlessly porting memory-using designs or modules across multiple platforms.

The first question in designing a memory abstraction is the general interface needed by the clients of the memory. In software, this general interface is the familiar virtual memory: a large contiguous memory with apparently sequential, blocking

reads and writes. Indeed, virtual memory as it exists in modern processors satisfies the first, second, and third interface requirements. First, virtual memory is a portable abstraction, and ample proof of its portability is evident in modern software. Second, virtual memory provides the illusion of a large, contiguous address space to user processes, making these processes somewhat easier to program. This illusion also provides portability between platforms with different amounts of physical memory. Third, virtual memory provides protection, allowing multiple processes to share a single physical memory without interfering with one another.

FPGA programs are similar to multiprocessing with respect to memory. FPGA programs are inherently parallel, and each memory client in a design can be viewed as an analog to a process in a traditional time-shared general purpose computer. Thus, virtual memory is a good starting point for a hardware memory interface, since it satisfies all the requirements put forth for portable memory interfaces and resembles existing hardware programs. However, some modifications to software virtual memory are needed in the context of hardware programs. Hardware is intrinsically parallel and pipelined, and the blocking interface prescribed by software virtual memory would limit the performance of many designs. Therefore, a hardware interface to memory must expose non-blocking memory operations to the programmer, so that the programmer has the opportunity to hide memory latency through pipelining. Thus, rather than the blocking read and write interface presented by general purpose ISAs, a better interface consists of three operations:

```
interface MEMORY_IFC#(type address, type data);
    Action readRequst(address addr);
    ActionValue#(data) readResponse();
    Action write(address addr, data newData);
endinterface
```

Figure 7-1: A general memory interface for hardware designs [17]. Actions in the memory system occur in-order, with writes taking precedent over reads occurring in the same cycle. The interface is described in the syntax of Bluespec SystemVerilog.

An important property of this non-blocking memory interface is that it does not explicitly state how many operations can be in flight, nor how quickly in-flight op-

erations will be retired. Requiring the programmer to handle this ambiguity in the program provides significant freedom in the implementation backing the interface. For example, a small memory could be implemented as a local SRAM, while a larger memory could be backed by a cache hierarchy. This freedom of implementation admits of automatic tools capable of producing program-specific memory implementations.

Although the state of device and memory portability in FPGA is poor, there is some recognition in industry of the need to support more general memory interfaces. For example, the Xilinx MPMC interface [69] allows designers to interface to a DRAM by way of several symmetric read-write ports. However, this interface still lacks an abstraction of size, and is limited to the physical size of whatever DRAM memory is attached to the board. Moreover, the interface has somewhat complex behavior in terms of the order in which responses can be received from memory. These weaknesses make the MPMC a poor candidate for a generic memory interface.

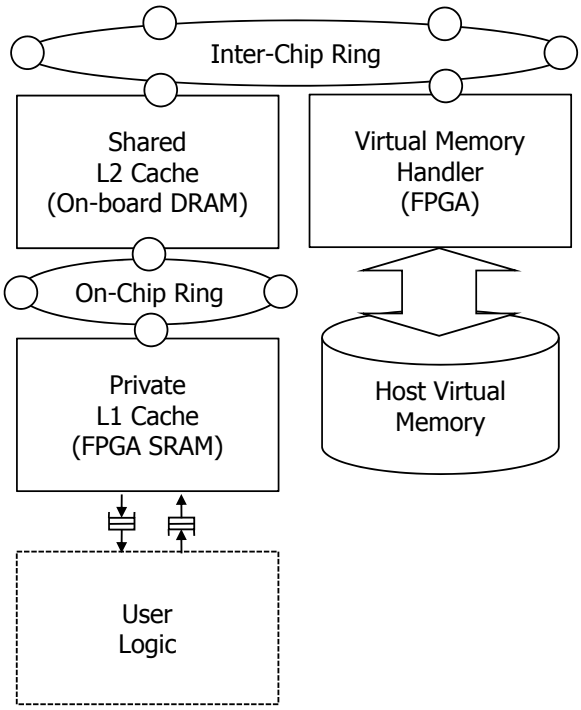


Figure 7-2: A view of a scalable multiple FPGA memory hierarchy. Fast, private local caches are backed by a shared last-level cache, which in turn is backed by global virtual memory. The structure of the hierarchy is automatically inferred at compilation time.

7.3 Services on a Single FPGA: the Scratchpad Memory Hierarchy

The previous section defined a portable and extensible interface to memory. This section will examine the use of that interface in providing the memory needs of a program running on a single FPGA: the scratchpads memory hierarchy. Not only does the scratchpads hierarchy provide an implementation of the interface described in the preceding section, it also provides the automatic synthesis of a program-and-platform specific multi-level memory hierarchy. The original work on the scratchpads memory hierarchy was conducted primarily by Michael Adler in [1], though the evaluations presented here are novel.

The description of the scratchpads hierarchy, begins with a discussion of their programming interface in Section 7.3.1. A basic implementation of the scratchpads hierarchy is described in Section 7.3.2. This basic implementation is augmented with an automatically synthesized memory hierarchy in Section 7.3.3. Finally, Section 7.3.4 presents a performance evaluation of the scratchpad memory hierarchy, based on the methodology of Stricker and Cross [56].

7.3.1 Programming Interface

The scratchpads hierarchy generally conforms to the memory interface presented in Section 7.2. Semantically, each scratchpad represents an independent memory space of arbitrary size. Programs instantiate a scratchpad memory interface using the syntax shown in Figure 7.3.1. Programs may instantiate an arbitrary number of scratchpad interfaces, however, separate scratchpad memory spaces are not coherent. Should the program require coherence between clients, the program must orchestrate the coherence manually.

The syntax shown in Figure 7.3.1 instantiates a single memory space with $2^{\texttt{ADDR\_T}}$ words of type `DATA_T`. The scratchpad constructor takes two arguments. The first argument is a unique numeric identifier for the scratchpad, an artifact of interfacing with

```
MEMORY_IFC#(ADDR_T,DATA_T) mem <-
    mkScratchpad(VDEV.SCRATCH.FBUF_Y, SCRATCHPAD_CACHED);
```

Figure 7-3: An instantiation of a local scratchpad. This declaration creates a single, unshared memory space of size `ADDR_T` with data type `DATA_T`. The system library will automatically generate a backing memory hierarchy for this address space.

the Bluespec compiler. This identification can be thought of as filling the function of a process ID in a conventional memory hierarchy, although multiple scratchpads in the same module will have different ids. The ID is used to identify and route requests from the particular scratchpad instance, and the user must ensure that these identifiers are unique, for example by making use of an enumeration. The second argument, which can be ignored until later in this discussion, directs the backing, synthesized cache hierarchy to instantiate a local L1 cache.

Externally, the scratchpad interface can handle data of any size. However, to simplify hardware implementation, particularly in the portions of the scratchpad hierarchy shared among several scratchpads, the memory hierarchy uses a fixed-width implementation. As a result, some marshalling or de-marshalling logic must be inserted at the scratchpad interface if the data type of the scratchpad is different from the internal data type, nominally 64-bits. For large data types, requests to the scratchpad are broken into multiple requests for the backing memory, while smaller data types are packed into the 64-bit word size. If it is needed, Marshalling logic is automatically inserted at compile time.

7.3.2 A Basic Scratchpad Implementation

Given a program instantiating scratchpads, the most basic implementation of the scratchpad hierarchy is to connect each scratchpad directly to host memory on an attached general purpose processor, ignoring both on-die memory and on-chip memory in favor of a memory with better semantic properties. Connecting user scratchpads directly to the host memory seems counterintuitive: the host memory is far away and the point of an FPGA memory system is, generally speaking, performance. However,

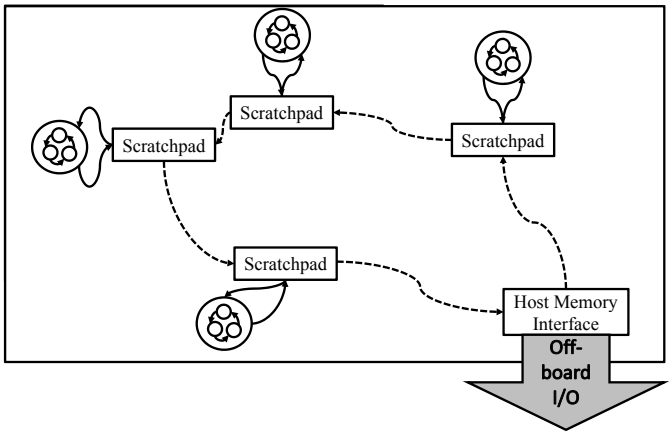


Figure 7-4: A degenerate memory hierarchy with no central cache.

the host memory provides pre-existing support for arbitrarily large virtual memory spaces, permitting any number of scratchpads to have any size, up to the maximum size supported by the host. Leveraging the existing memory functionality of the host greatly simplifies the hardware required to support scratchpads: the hardware simply produces requests and drains responses. Complex mechanisms like paging and address translation are handled by software and hardware on the host machine.

In the basic scratchpad implementation, each declared scratchpad is logically tied directly to a unique virtual memory space, with all memory requests going back and forth to that memory. However, in practical systems, there is a single host memory which is multiplexed among all user scratchpads. Because there may be many scratchpads distributed throughout a program, a direct connection between each scratchpad and the host memory interface is inefficient. Instead, as shown in Figure 7-4, scratchpads are connected using a pair of latency-insensitive rings, wherein each scratchpad and the host memory are ring stops. Memory requests flow around one ring from the scratchpads to the host interface, while responses flow back from

the host to the client scratchpads on the other ring. Two rings are needed in order to avoid deadlock, due to request-response dependence.

Using the host memory system to provide virtual memory requires a measure of software support. At FPGA startup time, software on the host side allocates large buffers for each scratchpad on the FPGA. When requests arrive at the host, these pre-allocated buffers are used to service them. These memory buffers are managed by the operating system just like any other memory.

7.3.3 Synthesizing a Cache Hierarchy

Utilizing host virtual memory gives a correct, but extremely low-performance, implementation of an FPGA-based memory system. Borrowing from general-purpose processor architecture, the scratchpad memory hierarchy uses multiple levels of caching to close the memory performance gap between user logic and the relatively slow host memory. Just as processor caches capture data locality, so too do caches in FPGA designs. To effect this caching, the scratchpad memory hierarchy incorporates both on-chip and off-chip memory resources.

The first level of caching in the scratchpad hierarchy occurs local to each scratchpad. Each scratchpad has the option of instantiating a private cache. These non-blocking, in-order caches are implemented in fabric SRAM resources, and can be sized independently by the designer. The L1 cache is inserted between the local memory interface and the host memory rings. In addition to providing fast, low-latency access to local data, this cache also serves as a filter for requests to the host memory, which improves both the performance and scalability of the scratchpads hierarchy.

Local caches improve the performance of the memory system, but further gains are possible. Most FPGA platforms have significant on-board memory capacity, typically in the form of an external bank of DRAM or SRAM chips. Though this memory has low bandwidth and high latency relative the on-die SRAM resources, accessing this on-board memory is still orders of magnitude faster than accessing host memory. On-board memory is usually discrete and therefore must be shared among all scratchpad clients. Considered in this light, the on-board memory bears a striking resemblance

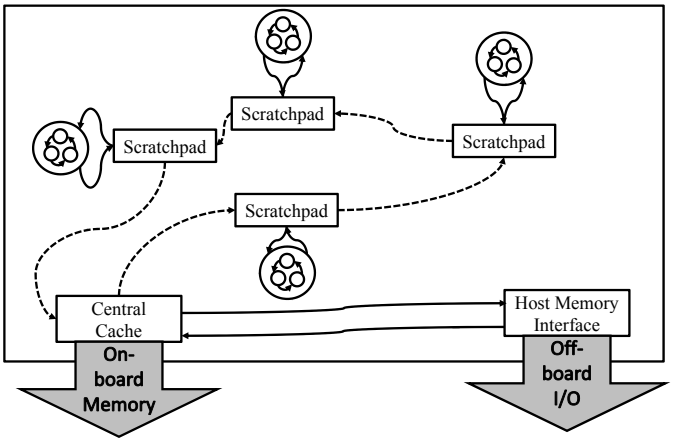


Figure 7-5: A memory hierarchy with central cache.

to a shared L2 or L3 cache in general-purpose processor systems.

The cache built on top of the on-board memory resource is called the central cache. As a shared entity, the central cache architecture differs substantially from the private L1 caches, and more closely resembles the multi-cycle, multi-ported, set-associative, non-blocking caches used at higher layers of general purpose processors. The central cache is introduced as a ring stop on the scratchpad rings, replacing the host memory interface, as shown in Figure 7-5. The central cache communicates with the host virtual memory interface, its backing store.

Of course, some platforms may not have an on-board cache, or the designer may choose not to expose this resource for area considerations. In this case the central cache module is replaced by a null implementation, and the memory hierarchy effectively degenerates to the scenario shown in Figure 7-4, where all the scratchpad memories are connected directly to the host memory interface.

7.3.4 Performance

Stricker and Cross studied the performance of processor memory systems during a period of growth in the level of integration of workstation-level machines, in an effort to determine which platforms would run various scientific workloads the fastest. They recognized that many scientific workloads have, at their core, some kernel that iterates over a working set with some stride length, for example looping over some array of structures. To decide which machine would be best for a particular kernel set, they ran simple stride-length, working set size combinations across a variety of machines to measure cache throughput.

This methodology is attractive in evaluating FPGA-based memory hierarchies because many FPGA applications have stride-based access patterns that resemble the kernels proposed by Stricker and Cross. Not only do block algorithms like matrix-matrix multiplication exhibit this pattern, but also many sliding-window algorithms like pixel-interpolation in H.264 are essentially strided, but with slight boundary irregularities. The Stricker and Cross method is also useful because of its highly predictable output – memory systems should generally have monotonically decreasing performance as stride and working set scale. Moreover, cache levels should have clearly identifiable plateaus. If these characteristics are not present, it suggests that the memory system may have a performance bug.

Overall, empirical evidence confirms that the behavior of scratchpads is similar to the behavior of processor caches originally studied by Stricker and Cross. Figure 7-6 shows the performance of a scratchpad hierarchy implemented on one of the ACP (Section 7.5.1) FPGAs. This graph depicts the three very clear plateaus expected of the scratchpad architecture, that is, the high-performance private L1 cache, the intermediate-performance central cache, and finally the low-performance host memory. Figure 7-7 shows the latency of read operations. As with bandwidth, there are three regions. However, the latency of the L1 and L2 caches is dwarfed by the latency of access to host memory. The large latency of host accesses is due to the lack of FPGA-demand-driven DMA between the FPGA and host memory on the ACP

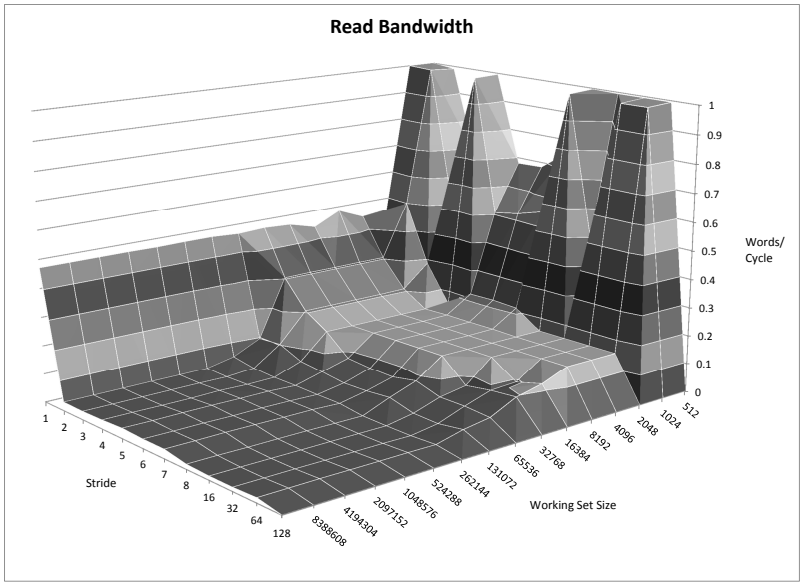


Figure 7-6: A example of scratchpad read bandwidth.

platform.

Figure 7-8 shows the average write bandwidth of the scratchpad memory. Exhibiting three clear plateaus, the write performance is qualitatively similar to the read performance. However, because writes are well-pipelined, the difference in performance across the three levels is much smoother. The test case depicted in the figure uses a word size smaller than the word size used in the rest of the memory system. As a result, read-modify-writes, which are not fully pipelined at the L1, cause the write bandwidth to saturate at one-half word per cycle.

7.4 Scaling Services to Multiple FPGAs

In a single FPGA implementation, there are two kinds of resources, shared and unshared. Unshared resources make use of point-to-point links, connecting a single client to a single device. On the other hand, shared resources use rings, which can scale automatically to handle any number of clients. However, in the case of shared devices, even thought there are many clients, there is still only one physical device

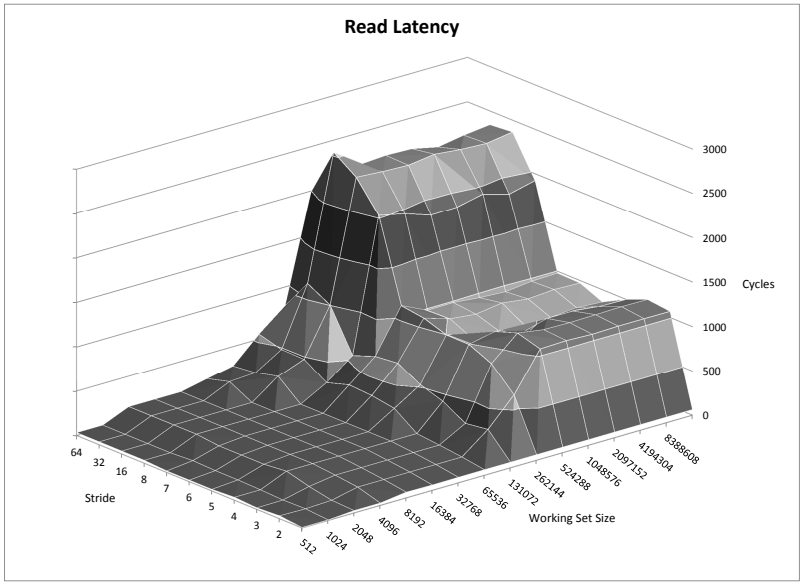


Figure 7-7: A example of scratchpad read latency.

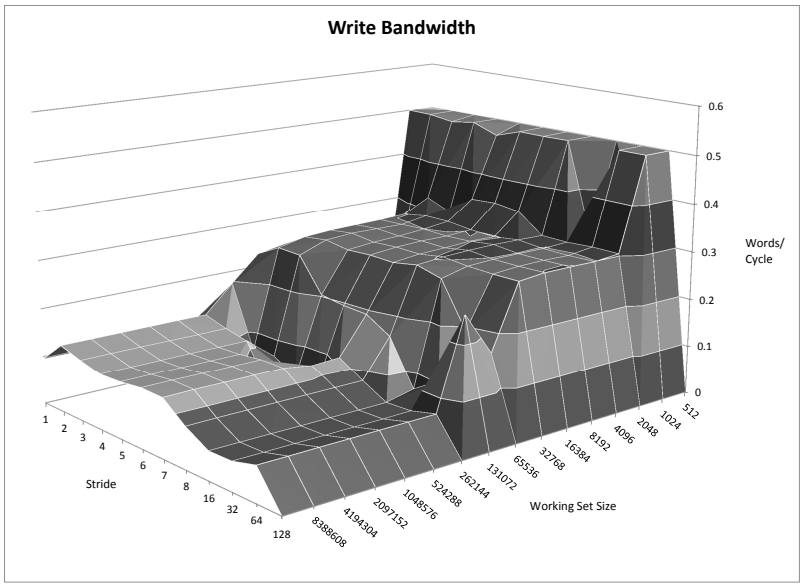


Figure 7-8: A example of scratchpad write bandwidth.

serving them.

In multiple FPGA implementations, the situation with respect to devices is more complicated, since there can be several devices located across the FPGAs in the execution environment. There are three possible configurations of devices in a multiple FPGA environment: unique, symmetric, and asymmetric.

Unique devices, as the name implies, are unique within the environment. Their handling is the same as a device in a single FPGA implementation, with the exception that some inter-ring hops may cross FPGA boundaries. An example of this kind of this kind of service is shown in Figure 7-9. Although the clients in this example are spread across two FPGAs, because latency-insensitive rings were in the client interfaces, the multiple-FPGA compiler can automatically extend the service to a second FPGA, making it available to remote modules. This synthesis is a direct byproduct of using rings to implement shared services: the service itself is unaware that multiple FPGAs exist in the system. However, multiple devices in the system require special handling on the part of the service writer. Symmetric devices are those in which the device occurs on every FPGA platform in the system, while asymmetric devices do not appear on all the FPGAs. In reality, these two cases are almost identical.

Handling the multiple resource case is well is an extremely important source of performance in multiple FPGA implementations. For example, many problems end up being memory bound on a single FPGA. A well-partitioned implementation can potentially make use of all available memory resources in a multiple FPGA system, leading to improved whole-system performance. There are many design issues related to optimizing program performance in the context of distributed resources. For example, devices across platforms may vary in quality, for example, a small, fast SRAM versus a large, slow DRAM. A good placement tool would take advantage of these asymmetries and attempt to match different memory consumer to memories matching their specific performance needs. This thesis provides a few programmer tunable interfaces for device optimization, but makes no attempt at automated optimization in this space.

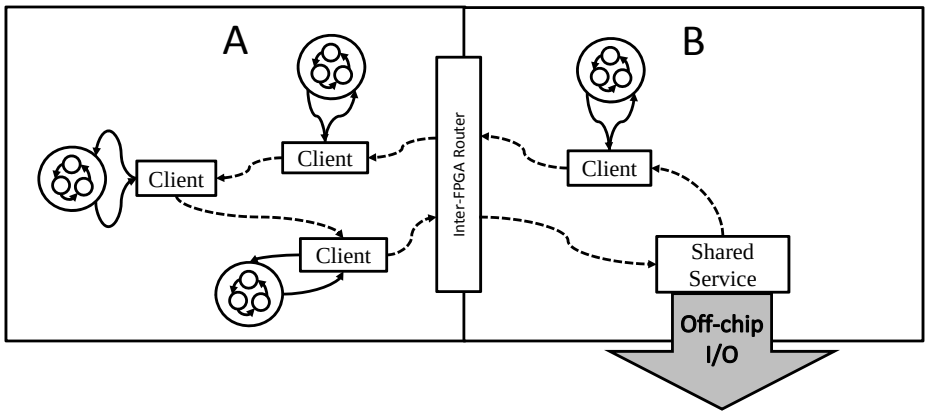


Figure 7-9: An example of a service distributed across two FPGAs. Here, the compiler will automatically insert the dotted links and all necessary logic to transit the inter-FPGA link.

Section 7.4.1 discusses the physical implementation of multiple FPGA services, and Section 7.4.2 gives a brief performance characterization of the scratchpads memory hierarchy in the context of multiple FPGAs.

7.4.1 Multiple Distributed Services

On a single FPGA and on multiple FPGAs in the unique device configuration, implementing shared devices is made somewhat straightforward by the use of rings, which the LIM compiler handles automatically. However, in the case of multiple FPGAs with multiple resources, whether symmetric or asymmetric, simply using a single ring breaks down, since in the multiple resource case, servers must know which client requests they are to service. Moreover, extending the single-service ring model to multiple resources forces the ring to cross the high-latency inter-FPGA interconnect, a needless performance penalty if clients only interact with their local memory resource.

In the current compilation flow, the issue of communicating with multiple physical resources is solved by creating a uniquely named ring for each resource. Unique names are determined by the name of the platform physically hosting the device. In some sense, this is a generalization of the unique resource case: each resource has a uniquely named ring for communicating with its clients. Creating unique rings per device

is straightforward in the current compilation flow. Each device library supporting multiple device instances is parameterized with a variable defining platform name, which is provided by the LIM compiler at compile time. The device library then instantiates named rings specific to the platform.

Clients now have a means of specifying exactly which resource they should use. However, there are still a problems. Resources are now identified, but now clients face the problem of choosing which resource to use. Ideally, this resource allocation decision would be taken by the LIM compiler after analyzing the memory needs of the particular program, but the LIM compiler does not currently have this functionality. Minimally, the module source code using a resource should not be required to be aware of which resource it is using, only that it is indeed using a resource. If the client module is required to know exactly which platform's resource to use, portability between environments is immediately lost. For example, a client may demand the memory resource of platform "FPGA1", when no such platform exists.

To abstract device usage from the client modules, the current LIM compiler implementation makes a strong use of the static knowledge of module placement. Because module placement is statically known, when device interfaces are instantiated, the device interface code can obtain the placement of the instantiating module and assign the new client interface to the device interface rings of the platform on which it is to be placed. This achieves the goal of abstracting resource usage: the ring choice is determined by the mapping file and device interface libraries, which vary between execution environments, while the client module only instantiates a device interface.

In the symmetric resource case, each platform has a resource, and making the client-resource allocation choice simple: all clients use their local resources. The asymmetric resource case, in which some platforms do not have a particular resource, is slightly more complex. In this case, the asymmetric platform provides a device interface, but this device interface is specifically parameterized to use the physical device of another platform instead of the local platform. The LIM compiler will aggregate these ring stops into a single ring spanning multiple FPGAs.

Figure 7-10 shows an example of an FPGA environment with several symmetric

services spread across an asymmetric set of platforms. In this case, the clients of C, which does not have a local device, are routed to platform B. The design is correct in the sense that all client have access to the service, although clients will each experience different and perhaps, in the case of the clients on B and C, degraded performance.

In the current LIM compiler implementation, distributed access to multiple FPGA resources makes use of static knowledge about module placement. This knowledge drastically reduces the amount of work that the compiler needs to do in order to connect device-using modules to the appropriate physical device. However, the implementation of automatic module placement will remove this source of knowledge, and requires the introduction of new genus of ring primitive.

The current device-sharing scheme make use of static, platform-based naming to allocate clients to device-specific rings. Thus device-specific rings could be implemented using the existing ring primitives described in Section 4.2.1 because the name of the device-specific ring was available at LIM graph construction. In the case of compile-time placement, device-specific rings are still necessary, but they cannot be described using normal rings. A client can only be bound to a device-specific ring once placement is known, but placement occurs after LIM graph construction. To solve this problem, a new kind of platform-specific ring can be introduced. These new ring stops are tied to a ring which is prohibited from crossing FPGA boundaries.

7.4.2 Performance

Figures 7-11 and 7-12 show the performance of a scratchpad memory in the context of a multiple FPGA system. In this system, scratchpads on both FPGAs share a single connection to the host memory in a configuration similar to the one shown in Figure 7-9. For this test, although there are multiple scratchpads in the system, only one is active at a time. The scratchpads in this case exhibit remarkably similar performance, even though the host memory link is asymmetric: the system tested has a relatively long latency between host and FPGA and this obscures the effect of the remote scratchpad requests to host memory transiting FPGAs. For those applications that have good locality of reference, sharing a single host connection, even if it is

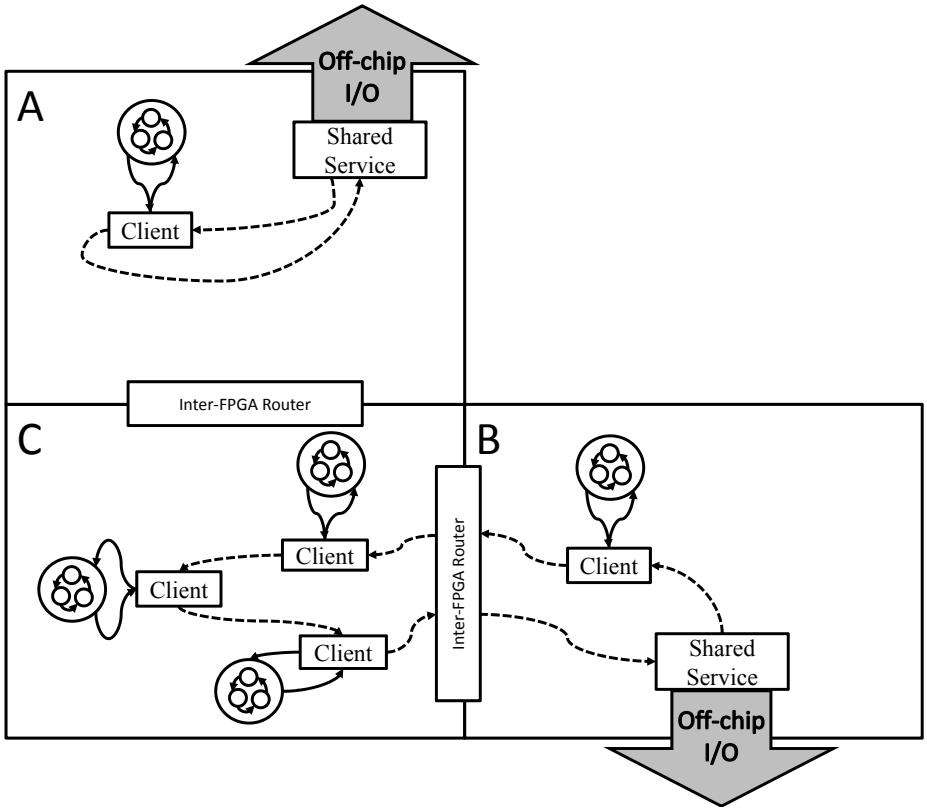


Figure 7-10: An example of a asymmetric services available in a single environment. Here, clients on platform B are routed to platform C for service.

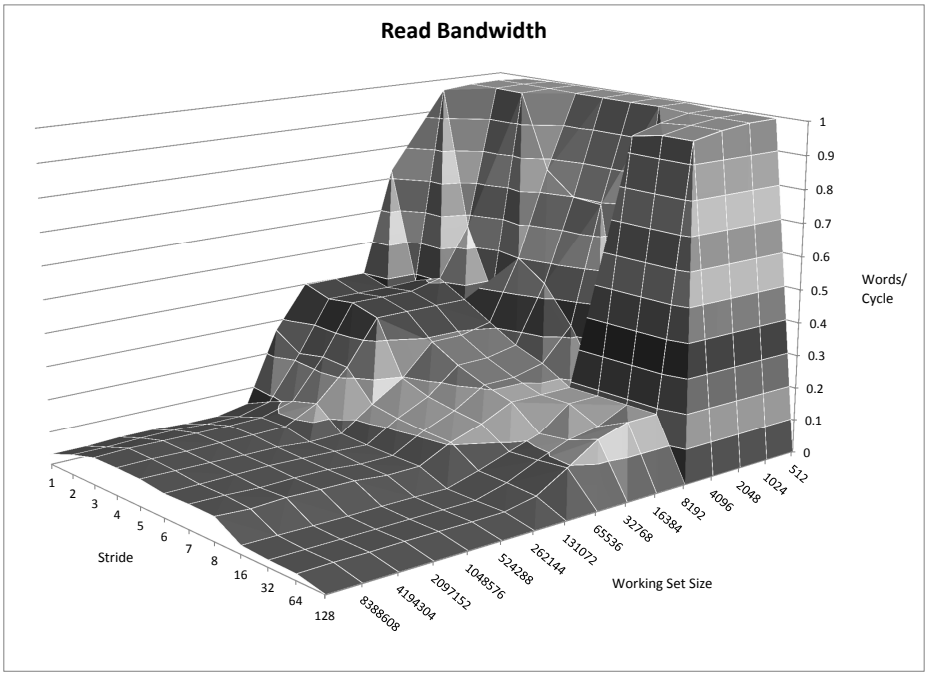


Figure 7-11: Scratchpad performance in the context of a multiple FPGA system. This scratchpad is local to the host memory connection. Notice performance degradation as compared to the single FPGA performance in Figure 7-6.

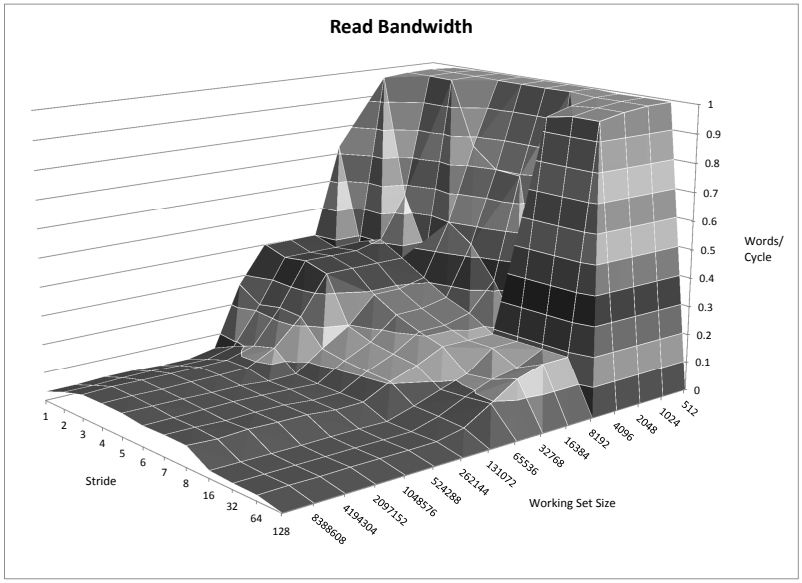


Figure 7-12: Scratchpad performance in the context of a multiple FPGA system. In this case the scratchpad is remote from the host memory link.

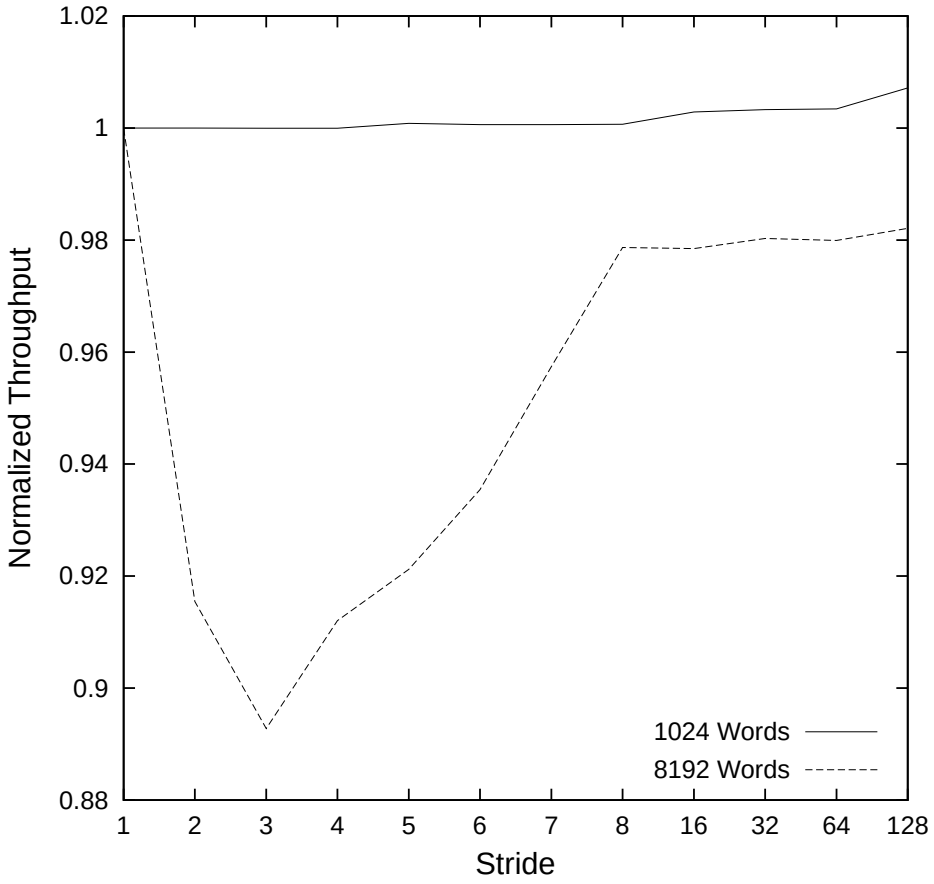


Figure 7-13: Simultaneous scratchpad performance. In this test, two scratchpads operated simultaneous. The performance of a placement in which the two scratchpads are on different FPGAs is normalized to the performance of a placement in which the two scratchpads are on the same FPGA.

several FPGAs away does not represent a significant performance bottleneck.

Figure 7-13 examines the case in which scratchpad memories are active at the same time. The test examines two configurations: one in which the scratchpads are co-located on the same FPGA and one in which they are located on two different FPGAs. These configurations differ only in that, in the single FPGA case, the L2 cache is shared between the scratchpads, while in the two FPGA case, each scratchpad has a dedicated, local L2 cache.

In the case that the working set fits in the L1 cache, the two configurations have nearly identical performance. If the working set fits in L2, but not in L1, the two FPGAs configuration outperforms the single FPGA configuration by up to 11%. In the single FPGA case, the L2 request and response rings transit both scratchpads,

increasing access latency from each by a single cycle. Additionally, scratchpad L2 requests may occasionally collide, causing further latency.

7.5 Multiple FPGA Platforms

Although the LIM compiler is fully general and the applications discussed in Chapters 8, 9, and 10 can be mapped to any FPGA environment, the evaluations presented in the those chapters were limited to two multiple FPGA platforms: the Nallatech ACP [28], described in Section 7.5.1, and the Xilinx XUPv5 [70], described in Section 7.5.2. These platforms represent nearly the cutting edge in FPGA technology at the time of the development of the implementations presented in subsequent chapters. Although both physical FPGA environments targeted in this thesis use Xilinx FPGAs, it should be noted that the LIM compiler is general and can target FPGAs provided by any vendor.

Choice of platform influences the performance of an FPGA application in the same way as choice of processor architecture influences the performance of a computer program. For example, different FPGA platforms have different memory hierarchies and different I/O bandwidths, and different applications prefer different amounts of these resources. Table 7.1 lists some salient, performance-influencing characteristics of the various FPGA environments.

Section 7.5.3 describes a generic simulation framework for simulating arbitrary configurations of FPGAs. Ultimately, programs must be mapped to real FPGA environments for execution. However, real programs have bugs, especially during early development, and debugging on the FPGA is, at best, a difficult proposition. Thus, at least some software simulation and validation is required before deploying a program on the FPGA. Unlike physical FPGAs, which are intrinsically finite in size, hardware simulators offer an unlimited supply of "implementation resources". Thus, even very large designs can be emulated in a single simulator. Because of this, it may at first seem that modelling multiple FPGAs is unnecessary. However, the multiple FPGA simulator is useful in three ways. First, from the perspective of the compiler

	Total Slices	Host Bandwidth	Inter-FPGA Bandwidth	Inter-FPGA Latency
Simulator	Infinite	300 MB/s	314 MB/s	6400 ns
Nallatech ACP	51840	55 MB/s	1157 MB/s	346 ns
XUPv5	17280	1.3 MB/s	180 MB/s	159 ns

Table 7.1: Structural and performance metrics for various FPGA Platforms. Simulator performance metrics depend on the host CPU, and the numbers shown are representative of a an Intel Core i7 Nehalem machine.

writer, partitioned simulation serves as the primary means to test the LIM compiler and the various components of the synthesized multiple FPGA communications infrastructure. Second, partitioned simulation permits some analysis of the behavior of partitioned programs, which is a useful source of feedback for the compiler. Finally, partitioned simulation improves the parallelism of the software simulator, potentially decreasing simulation time.

7.5.1 Nallatech ACP Module

The Nallatech ACP module consists of a pair of Virtex-5 LX330T that plug directly into an Intel Front Side Bus (FSB) socket. Since it is tightly coupled to both processor and memory, the ACP enjoys low-latency and high-bandwidth communication for its silicon generation. Unfortunately, the ACP I/O does not support DMA to the host memory, and host-FPGA interaction is slower than otherwise might be expected from a direct memory-bus connection.

The two FPGAs on the ACP are connected by way of a high-speed LVDS bi-directional interconnect. This interconnect has a fairly high bandwidth, but also has a latency on the order of 3 μs. The Virtex-5 generation represents the last FPGA generation in which LVDS-style communications is competitive with high-speed in-silicon SERDES links. The ACP platform provides a two 8MB SRAM, one attached to each of the FPGAs. Although this memory has high-bandwidth and low-latency, it is extremely small, and, in practice, presents problems for designs with large working sets, such as HAsim (Chapter 9). The ACP provides no other useful peripheral

devices.

The ACP is a very large FPGA environment for its generation, and so it can be used for the implementation of very large algorithms. Most designs presented in this thesis were implemented on the ACP, with the exception of the on-air wireless experiments, which require an interface to an RF-front-end.

7.5.2 XUPV5

The XUPV5 [70] is a widely deployed educational platform built around the Virtex-5 LX110t chip. Its chief features are its low cost and its wide deployment in the academic community. Counterbalancing its cost and popularity are its small size and lack of a high-performance host I/O interface. However, the XUPV5 features a wide variety of peripherals including a 2GB DDR2 memory. Additionally, the XUPV5 has many SERDES transceivers, with operating bandwidths up to 3.75 Gbps and sub-microsecond latencies. These links may be used to construct many different multiple-FPGA topologies.

The XUPv5 is of primary interest in the context of Airblue, the wireless platform presented in Chapter 8, since the XUPv5 can be integrated with the USRP2 [15] RF-front-end. This interface goes over one of the XUPv5's high-speed SERDES links, and permits the XUPV5 to host a variety of fully-functional, commercial-standard-compatible wireless transceivers.

7.5.3 Multiple FPGA Simulator

In much the same way that they can be partitioned among multiple FPGAs, designs can also be partitioned among multiple simulator instances, and, given the compilation framework of Chapter 4, partitioned simulation is straightforward. To model multiple FPGAs in simulation, the LIM compiler is simply presented with an environment containing as many simulation platforms as needed. The compilation flow for the simulator is the same as the flow for physical FPGAs. However, instead of invoking the FPGA-implementation tool backend to produce FPGA programming files,

a set of independent simulation executables are generated, one for each simulated FPGA.

The LIM compiler requires that the platforms of an execution environment be strongly connected by way of an inter-platform interconnect. However, RTL simulators, which are generally intended for whole system simulation, have no notion of external pins or communication with other, independent simulators. However, modern RTL simulators generally support some interface to C or C++ as a convenience in modelling. These interfaces give access to any host system functionality available in general programming environments, including inter-process communications. Using this software interface, partitioned simulators communicate with one another via Unix file-system FIFOs. Each pair of communicating simulators has a single, private FIFO, forming a fully-connected graph of simulators. These FIFOs are created at the start of simulation, and begin transmitting as soon as both simulator endpoints are initialized.

Any topology of FPGAs can be implemented using this simulator, giving the capability to model any program on any execution environment. Although each simulator platform can physically communicate with all other simulators in the environment, only those inter-connects explicitly list in the environment description file will carry traffic. To model non-fully-connected topologies, the programmer can simply remove these links from the description file.

Partitioned simulation leverages the latency-insensitive channels to avoid the need to synchronize the behavior of the multiple simulators, and thus makes no attempt to accurately model the timing of any particular physical environment. As a result, it may be of limited use in many traditional simulation roles, such as verification. However, in exchange for this loss of fidelity to a particular physical environment, simulation speeds of a single design can be improved by partitioning it among many parallel simulators.

The performance of the software simulator is generally quite good, although it is usually a little slower than the equivalent single FPGA simulation for small designs. Intuitively, this should not be so - parallelizing the simulation of even a small hardware

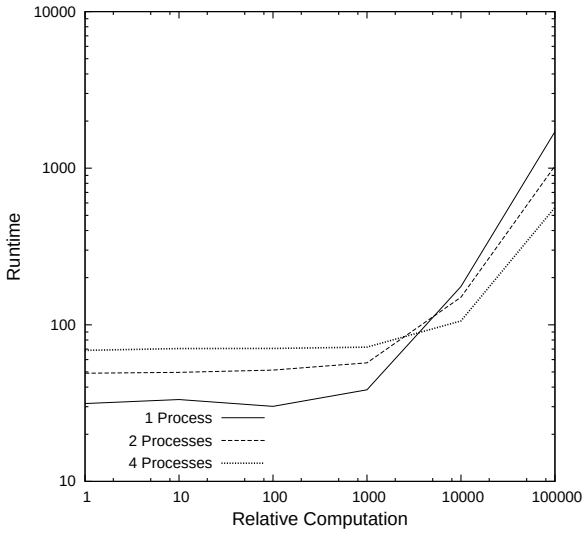


Figure 7-14: A variant of the loops test, run on multiple parallel simulators. Initially the overhead of communication dominates, but as compute-to-communication ratio grows, parallel simulation becomes advantageous.

design should improve the speed of what was intrinsically a serial process. The cause of this slow down is communication between the simulation processes.

Figure 7-14 shows the results of a limit study of parallel simulation. In the test, a busy delay loop is introduced into each link of the loops micro-kernel of Section 6.1, modelling progressively more complex,and consequently slower, simulation. For light simulation, the multi-process simulators are slower than the single process: the loops test is simple forwarding data between the processes and incurring some communication overhead not present in the single process simulator. As the busy delay increases and the ratio of compute to communication commensurately increases, multiple threads of execution eventually yields improved wall clock run times. Although the graph presented implements a simple kernel, the partitioned simulation speed of real programs may also improve. Figure 7-15 shows the simulation speeds of several HAsim models with different numbers of cores. Parallel simulation with two processes is as much as 35% faster than a single-process simulation for the larger HAsim simulators. As in the case of the loops kernel, if the amount of simulation work is small, that is, if few cores are simulated, then single process simulation is faster than parallel simulation due to communication overhead.

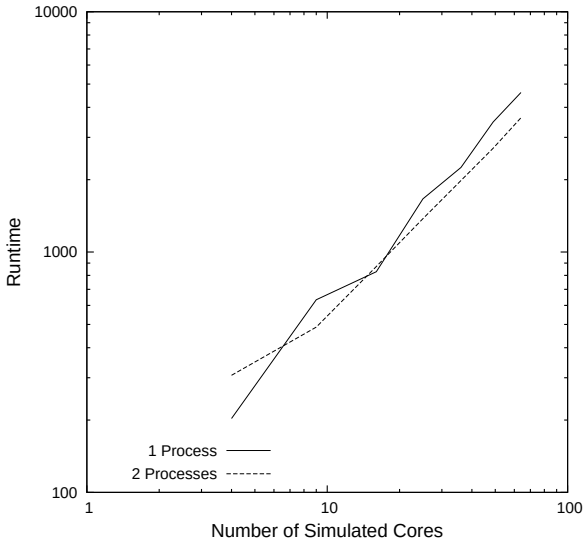


Figure 7-15: Parallel simulation improves the simulation speed of real programs, in this case, HAsim. As with loops, the overhead of communication dominates for small models, but as compute-to-communication ratio grows, parallel simulation becomes advantageous.

The simulation infrastructure presented in this section is a preliminary implementation. A better simulator architecture would implement simulation platforms as threads in a single process to reduce communications cost, perhaps with some better scheduling control and shared-memory-based inter-process communication. The current simulation environment is likely paying enormous overhead due both the cost of separate processes and the overhead of communication through file-system FIFOs, which results in an operating system call each time a data is transmitted. If refinements are made to reduce these overheads, it is possible that partitioned simulation could eventually outperform most, if not all unpartitioned simulation.

Chapter 8

Airblue

In recent years, the wireless research community has developed a large and growing set of protocols and algorithms to improve the throughput and capacity of wireless networks. These schemes span the physical (PHY), MAC, and network layers of the protocol stack. Some examples include interference cancellation [24], ZigZag decoding [21], Conflict Maps (CMAP) [65], the SoftPHY interface [32, 31, 64], and SWIFT [51]. A common theme in all these schemes is that they embody some form of *cross-layer design*, in which additional information is passed from lower to higher layers, with higher layers using this information to exercise some improved control over lower-layer decisions. For example, the SoftPHY interface [31] extends the receiver PHY to send to higher layers confidence information about each bit's decoding, so that those layers can perform better error recovery [32], bit rate adaptation [64], diversity routing [34], and so on. In fact, even the simple example of using the receiver's signal-to-noise ratio (SNR) to determine a transmit bit-rate is a PHY-MAC cross-layer protocol.

The problem with on-going research in cross-layer protocols is in their evaluation. Hardware is attractive for evaluation because of its good throughput and latency characteristics. However, commercially available wireless chip-sets offer limited control flexibility: some simple MAC modifications are possible, but serious protocol modification is generally infeasible, and, of course, no new hardware can be added to the chip-set. FPGAs offer the performance of hardware and full flexibility. Unfor-

tunately, the effort required to implement a high-performance wireless protocol from scratch in FPGA is enormous, and many existing FPGA-based solutions [66, 58] are difficult to modify. As a result, researchers turn to software-defined radios (SDR), like GNUradio [20], for their evaluation needs. GNURadio offers the flexibility and familiarity of software and the wide-band capabilities of the USRP [15]. Since the USRP can transmit and receive any waveform and can operate in a number of spectrum bands, many interesting protocol experiments are possible. However, in light of the limited processing capability of general purpose CPUs and the microsecond-deadline real-time requirements of modern wireless protocols, GNURadio does not offer sufficient performance for cross-layer experiments at or near the levels of throughput and latency required for commercial wireless protocols. The lack of realism in typical wireless protocol experiments may flaw experimental results in serious ways.

Airblue solves the experimental issues in the wireless domain by coupling the performance of the FPGA with the programmability of software. Airblue is a highly parametric library of OFDM components intended to assist wireless domain-experts in implementing novel baseband processors, enabling them to achieve high-performance OFDM implementations with relatively little engineering effort. Airblue was designed in the latency-insensitive style [39] [40], with modular refinement in mind. This property allows Airblue modules to support many protocol configurations, since blocks in an OFDM pipeline may be directly substituted for one another, and greatly reduces programming overhead: new protocols typically require that only one or two blocks be modified. In addition to programmer productivity, Airblue adds a measure of credibility to wireless experiments: in software programmers are free to explore unrealizable algorithms and implementations, while Airblue implementations validated on FPGAs can be used to fabricate high-performance ASICs with relatively minor modification. A typical baseband pipeline implemented in Airblue, shown in Figure 8-1 has relatively little feedback, although the main data path has high bandwidth and low latency requirements.

Airblue implementations of many current wireless protocols, such as 802.11g, fit on a single FPGA, but implementing future wireless protocols will likely require

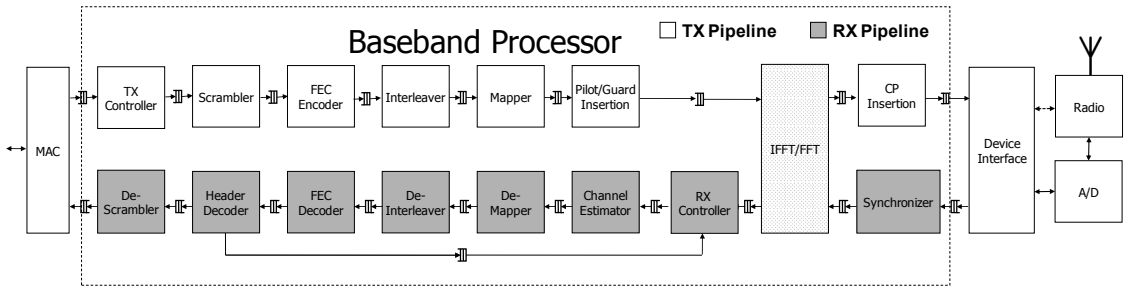


Figure 8-1: An Airblue 802.11g-compatible transceiver. In the SoftPHY experiment, only the forward error correction (FEC) decoder block is modified.

multiple FPGAs. The computational complexity of protocols is growing, and can overwhelm the resources of a single FPGA. Some implementations of the two protocols examples examined in this chapter, Spinal Codes in Section 8.2 and SoftPHY in Section 8.3, do not fit on the original Airblue platform and require multiple FPGAs. Additionally, the *physical* requirements of upcoming wireless protocols may mandate implementation across multiple FPGAs. Most modern wireless protocols, including 802.11n and 802.11ac, are multi-in, multi-out (MIMO), and require multiple radio front-ends to operate at full bandwidth. In single FPGAs, this is problematic, because each FPGA platform can intrinsically support a limited number radio front-ends, and board-level redesign is both expensive and time-consuming. On the other had, multiple FPGA platforms can be aggregated, using the LIM compiler, to provide access to many front-ends simultaneously, permitting the development of large MIMO systems using existing equipment.

A first order concern in multiple-FPGA wireless implementation is meeting protocol-level timing. These timings govern when protocol messages must be transmitted and violating them results in a loss of throughput due to, for example, spurious retransmission. Typical wireless protocols implemented using Airblue have protocol-level latency requirements on the order of tens of microseconds. However, the microsecond time scale is easily satisfied by modern inter-FPGA interconnects which typically sub-microsecond latencies and giga-bit bandwidths. For example, a two-FPGA partitioning of the 802.11g pipeline shown in Figure 8-1 can communicate with commodity wireless hardware.

This chapter will first describe the operational characteristics of a typical Airblue system, in Section 8.1. This general discussion is followed by two different applications of the multiple FPGA compiler proposed in this thesis to Airblue. Section 8.2 describes the implementation of a complex transceiver that does not fit in a single FPGA, while section 8.3 examines leveraging multiple FPGAs to accelerate wireless simulation.

8.1 Anatomy of an OFDM Pipeline

Figure 8-1 shows a block diagram of an OFDM physical-layer (PHY) transceiver. The transceiver is divided in to two halves, the transmitter and receiver, with a shared FFT. The PHY layer receives its digital input and output from the Medium Access Control (MAC), which controls protocol-level timings and handshake packets. At the other end of the pipeline, the PHY feeds digital representations of analog signals into the radio front-end, which eventually puts a signal on the air, and receives digital representations of incoming signals. The following section gives a brief overview of the major blocks in a typical OFDM protocol. To assist in this discussion, Figure 8-2 shows the power spectrum of an 802.11g packet.

8.1.1 Transmitter Pipeline

TX Controller: Receives information from the MAC and generates the control and data for all the subsequent blocks.

Scrambler: Whitens, or randomizes, the data stream to improve peak-to-average power ratio and Forward Error Correction (FEC) behavior. Scramblers are usually implemented with linear-feedback shift registers (LFSR).

FEC Encoder: Adds redundancy to the data stream to improve transmission throughput in the face of a noisy communications channels. Wireless transceivers use a variety of schemes for error correction, including block codes (Reed-Solomon), convolutional codes, and, more recently, rateless codes (the spinal codes discussed in Section 8.2).

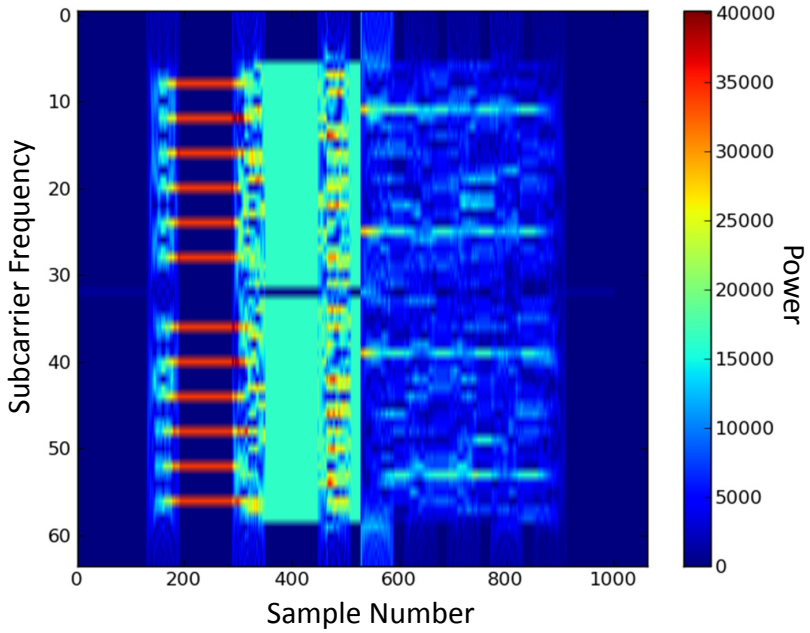


Figure 8-2: The power spectrum of an 802.11g packet. The packet opens with two different preambles one starting at sample 160 and the other starting at sample 320. The body of the packet consists of a sequence of six symbols, starting at 480. Each symbol is 64 samples long, with 16 extra prefix samples for a total of 80 samples. The symbols each have four pilot tones, the continuous bands at frequencies 12, 26, 39, and 53. Note that the guard bands, 0 through 5 and 59 through 63 actually do not have power. When power is present in these bands, the symbol FFT frames are misaligned, for example at sample 800. Such a misalignment represents a failure in synchronization and results in a failure to decode the packet.

Protocols use a technique known as *puncturing* to reduce the transmitted number of bits. For higher transmission rates on low-noise channels, the encoded data is *punctured* by deleting bits before transmission and replacing them with fixed values on reception. This reduces the number of bits to be carried over the channel and depends on the decoder to correctly reconstruct the data.

Interleaver: Rearranges blocks of data bits by mapping adjacent coded bits into non-adjacent sub-carriers to protect against burst errors and to improve the performance of the error correction code. Protocols typically use block size equal to either the number of bits in an OFDM symbol or the number of bits in a packet.

Mapper: The mapper converts digital data into an analog representation. This analog representation is a complex number representing the phase and magnitude of a sinusoid. This sinusoid will eventually be transmitted over the air. For high data rates, multiple bits may be mapped to a single sinusoid, with higher precision variations in its phase and magnitude. In Airblue, the mapper also does a serial-to-parallel conversion in anticipation of the IFFT.

IFFT: The IFFT mixes the frequency domain sinusoid representations produced by the mapper into a time domain representation of those sinusoids which can be transmitted. Because FFT and IFFT are symmetric operations and because they are computationally expensive, this block is shared between the transmit and receive pipelines.

Pilot/Guard Insertion: Adds the values for pilot and guard sub-carriers. Pilots are known, protocol-specific sinusoids that can be used for calibration at the receiver. Guard bands help to mitigate interference from adjacent frequency channels.

Cyclic Prefix Insertion: Inserts a cyclic prefix, a set of repeated samples, for each symbol. These samples have two effects. One is to protect against inter-symbol interference, a self-interference effect due to multi-path signal propagation. The other is to provide some margin of error for the receiver synchronization.

This block also adds a preamble before the first transmitted symbol. The preamble is a set of known symbols used by the receiver to detect the start of a new transmission and to learn channel parameters, including frequency-specific attenuation.

8.1.2 Receiver Pipeline

The receiver is the logical inverse of the transmitter pipeline. However, since the receiver must operate on a noisy, corrupted signal, it must employ more complex algorithms to recover the original transmitted bits. Consequently, the receiver occupies a much greater area than the transmitter.

Synchronizer: The synchronizer's main function is to detect the start of a transmission and to discover the alignment of OFDM symbols in the transmission using the packet preamble. If the symbol alignment is incorrect, then OFDM symbols will be corrupted and reception will fail. The synchronizer also uses the known preamble to compensate for several non-idealities in transmission, including carrier frequency offset and oscillator offset.

Serial to Parallel : This module converts the serial sample stream into parallel FFT frames and removes the cyclic prefix. This module also coordinates some control between the synchronizer and the main receiver pipeline.

FFT: The FFT converts the time-domain signal received by the RF front-end into a frequency-domain representation.

Channel Estimator: This module attempts to compensate for non-idealities in the transmission environment, such as non-uniform attenuation across frequencies. The channel estimator uses known values in the packet, such as the preamble and symbol pilots to tune its internal filters and modify the received samples.

Demapper: This module is the inverse of the mapper module. The Demapper converts the sinusoid magnitude and phase descriptions produced by the FFT back into bits. Because these translations are imprecise due to noise in the transmission, the demapper will also associate confidence values with the output bits. These confidence values improve the decoding power of certain error correction algorithms.

Deinterleaver: The deinterleaver inverts the interleaving performed at the transmitter and restores the original bit ordering.

Error Correction Decoder: This block exploits the structured redundancy information placed in the bit stream at the transmitter to correct errors in the data

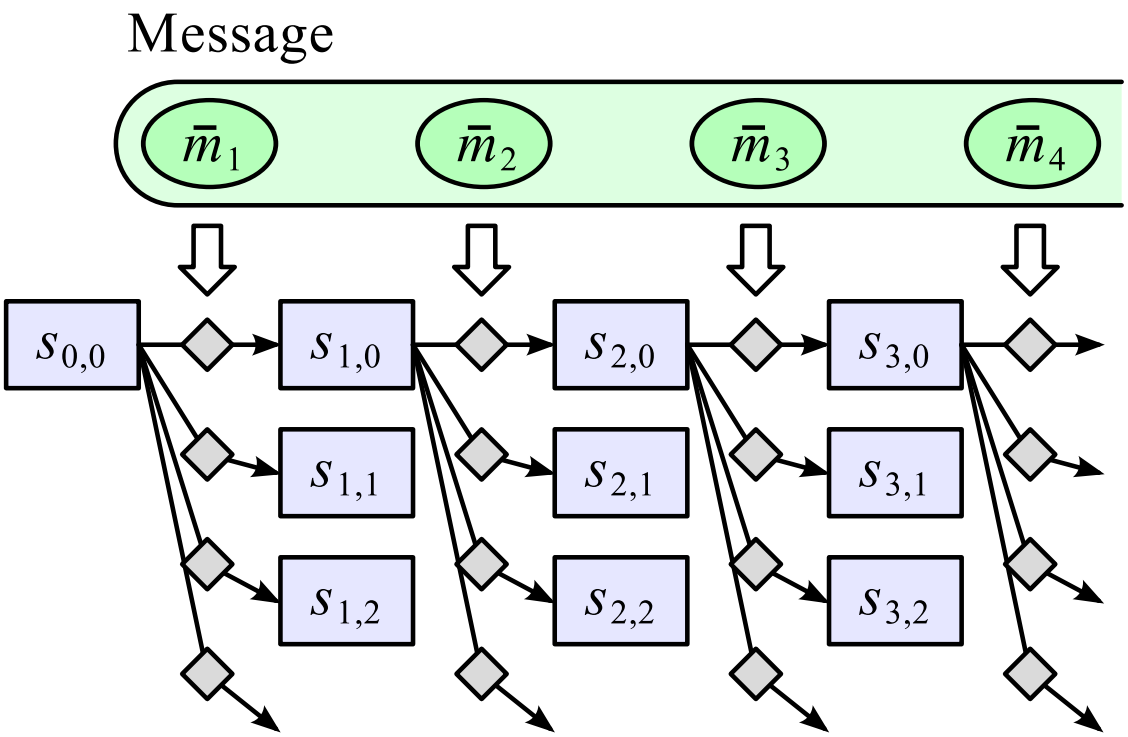


Figure 8-3: Symbols produced by spinal encoder. Chaining the hash function adds structure to the code.

transmission. Typically, the error correction decoder is the most complex block to implement in the FPGA, both because is involves fine-grained bit manipulation and short feedback paths and because and because it must provide high throughput at the granularity of single corrected bits.

Descrambler: This module inverts the whitening operation performed at the transmitter.

RX Controller: This module distributes control and configuration among the modules of the receiver pipeline. Its chief operation is parsing the protocol header to obtain this control information.

8.2 Spinal Codes

Spinal codes [50] are a relatively recent, Shannon-limit approaching rateless error correction code. Spinal code are structurally simple: spinal codes randomize messages by

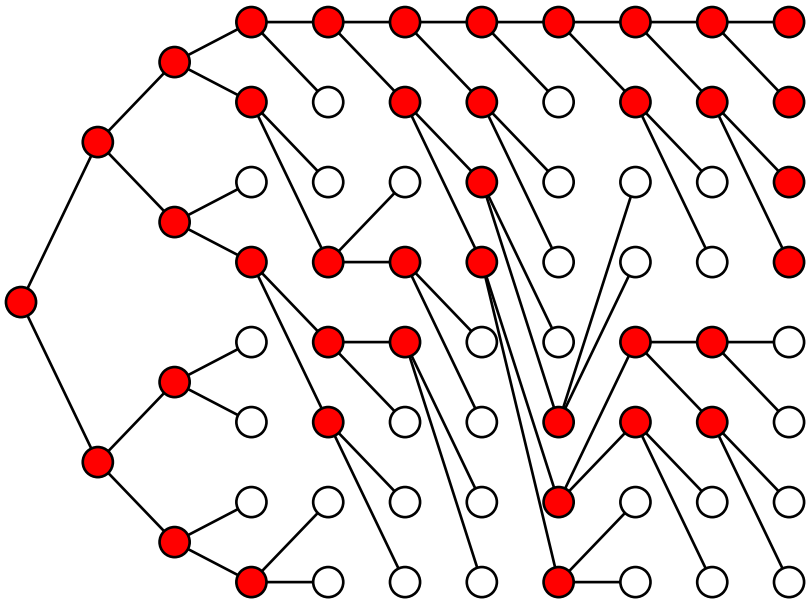


Figure 8-4: Spinal code beam search with beam width = 4. Pruning prevents a state explosion.

way of a chained hash function, shown in Figure 8-3. The message to be transmitted it divided in to small pieces $\bar{m}_i$. These $\bar{m}_i$ are hashed using a known hash function with a known seed $S_{0,0}$, producing a symbol for transmission. Spinal codes exploit the structured randomization provided by symbol hashing: each sequence of hash values corresponds to a single original message prefix with very high probability. Symbol hashing can be repeated indefinitely to produce a sequence of symbols that can be jointly decoded at the receiver. This permits the dynamic construction of arbitrary code rate: the transmitter can simply construct and transmit more symbols, lowering the bit rate and increasing the information at the receiver, until the receiver is finally able to decode the original message. The ratelessness of spinal codes permit them to outperform, in practice, most known coding schemes.

Although the spinal transmitter is simple, consisting only of a set of simple hash functions, the spinal receiver faces the challenging problem of extracting the original transmitted message given the sequence of noisy received symbols. Because the hash chaining of Figure 8-3 creates a dependence between messages $\bar{m}_i$, the space of all possible message can be viewed as an exponentially growing tree, a portion of which is depicted in Figure 8-4. Each path from a leaf of the tree to the root represents a unique message. Given a set of received symbols the receiver must choose the correct path out of the tree.

In contrast to existing convolutional and turbo codes, which operate on a fixed-sized trellis, spinal codes operate on an exponentially expanding tree, making choosing the correct path in the spinal code tree a challenging problem. The decoding problem can be broken down into two parts: deciding the most likely message for a given $\bar{m}_i$ chunk and deciding the most likely sequence of $\bar{m}_i$ message chunks in a complete message. The hash function plays a pivotal role in both decision problems.

At a single node, the hash functions helps to distinguish potential message chunks. Figure 8-5 shows an example of these sequences of hashes for two possible decodings of an $\bar{m}_i$ message chunk: p and q, versus a set of received symbols representing $\bar{m}_i$. For the first hash in the sequence of hashes, p and q have the same expected value. In the case of a high transmission rate in which only one symbol is sent,

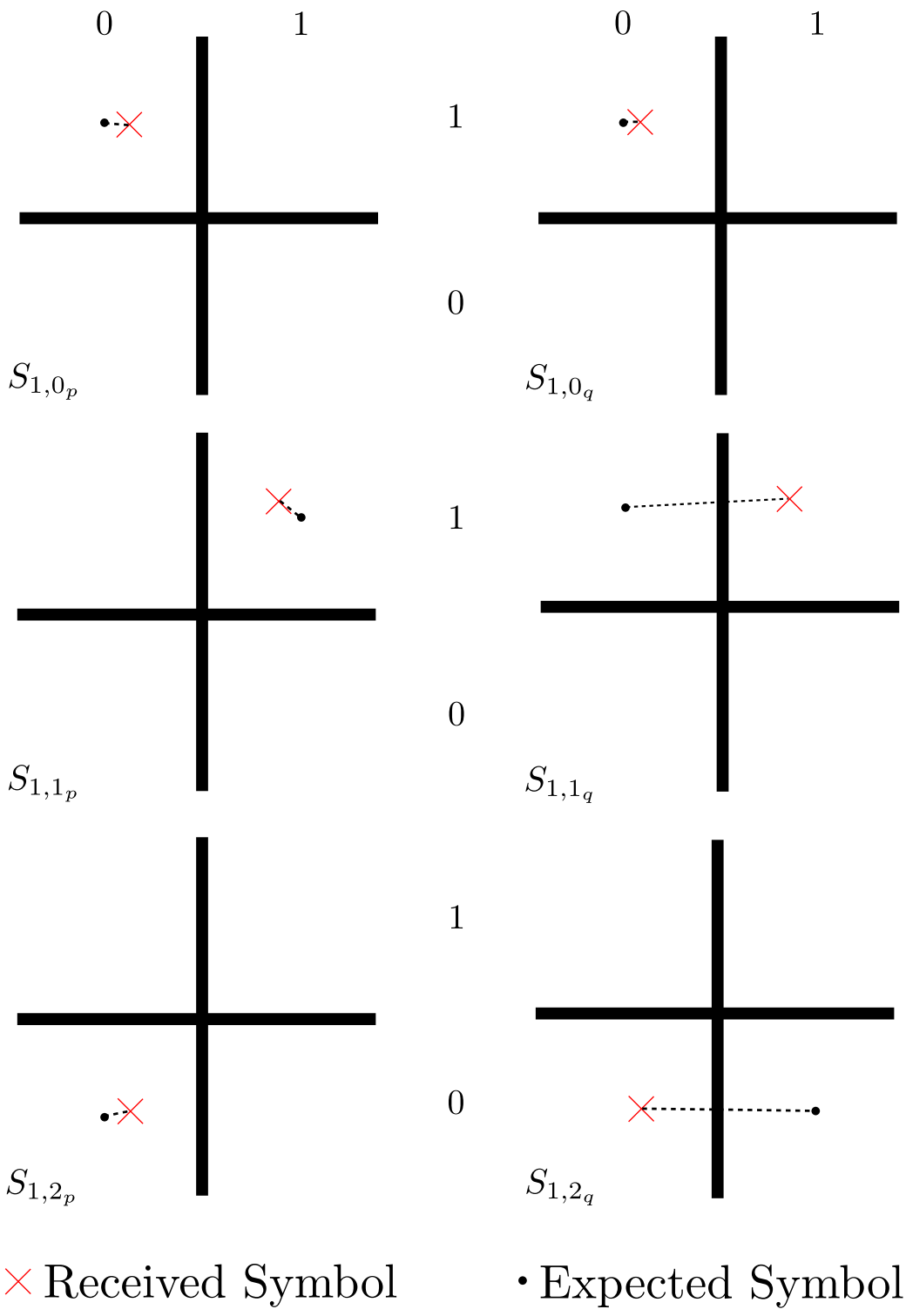


Figure 8-5: Symbol sequences for two messages m_1 and n_1, an alternative message decoding. The chained hashes of Figure 8-3 are used to determine the symbols. Although the first symbols coincidentally collide, the hash ensures a rapid divergence and decodability.

the two messages would be equally likely at the decoder. However, because hashing randomizes the sequence of symbols for each potential message, the likelihood of the messages rapidly diverges as more symbols are accumulated at the receiver, and p becomes far more likely.

Hash chaining between message chunks helps the decoder choose the correct message in the case that symbols collide between potential messages, as in Figure 8-5. The chained hash function guarantees independence between the symbols generated at nodes of the the same generation in the tree. Thus even if some symbols for two paths in the tree match (or nearly match), subsequent $\bar{m}_i$ relative to p and q will have different symbols with high probability. Since only one descendent path from p or q is correct, if the decoder examines all the descendents of p and q, only the descendents of one message will be likely. Another way to think of the operation of the hash chaining is spreading errors and collisions among adjacent message nodes. When SNR is high and noise is limited, this property permits the reception of punctured packets, in which some message chunks are inferred solely from the symbols of adjacent message chunks.

As noted above, decoding the a transmitted, spinal-encoded message amounts to simply picking the most likely path in the tree. A naive decoder calculates, for each node in the tree, the difference between the received set of symbols and the expected set of symbols for that node. Then, for each leaf-to-root path in the tree, the decoder sums summing these differences scores along the path. Finally the decoder selects the leaf node with the smallest path sum. The message represented by this path in the tree is the most likely transmitted message.

The problem with the naive decoder is that it is exponential in the length of the transmitted message. The key observation in reducing the complexity of the decoder is that only one path in the tree is ultimately correct, and only a few paths, typically sibling of the correct path, are even likely at each message step. Only these paths need be examined by the decoder. Aggressively pruning the message space, as shown in Figure 8-4, still results in a reasonable decoder implementation, as shown in Figure 8-6. This pruning reduces the complexity of the exponential tree search to that of the

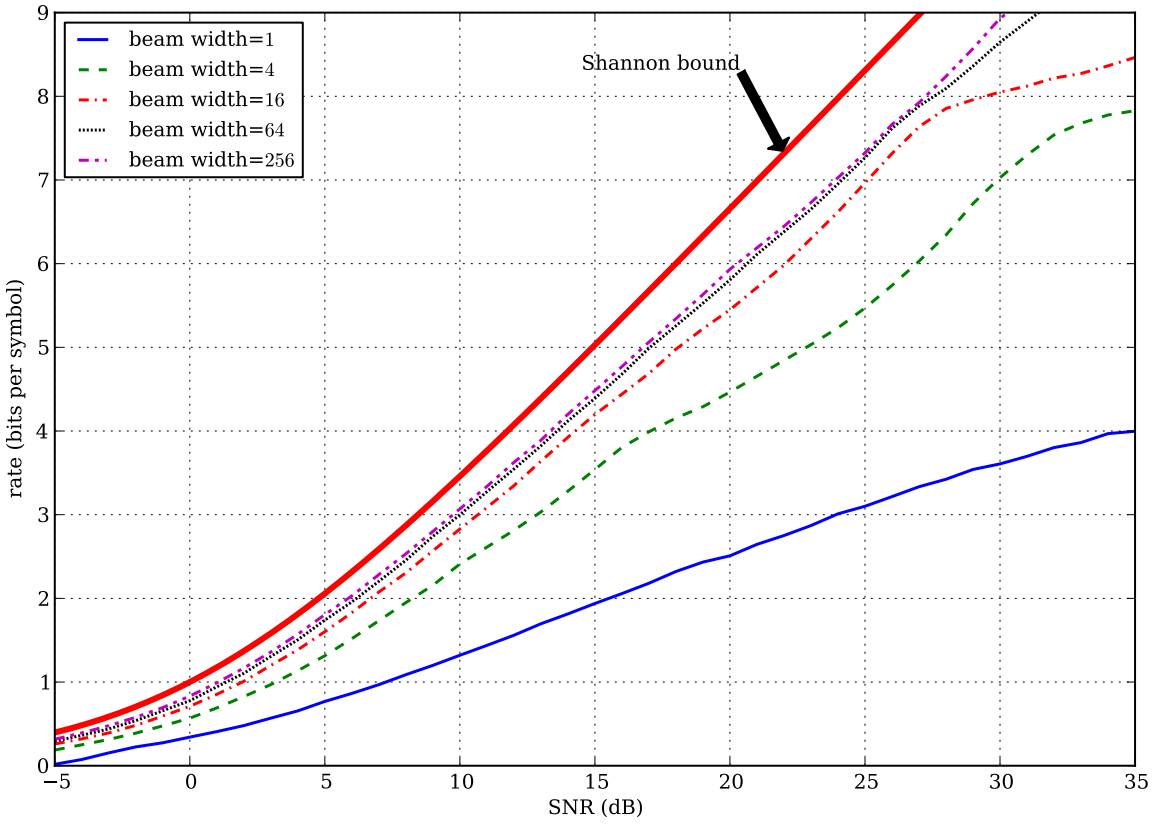


Figure 8-6: Higher beam width indicates less pruning. Spinal decoders achieve greater spectral efficiency when they search a larger portion of the decoding tree.

fixed-sized trellis. Although highly pruned decoders are functional, it is clear from the software simulation of Figure 8-6 that propagating many paths per step results in better channel efficiency. Therefore, a practical evaluation spinal codes requires the implementation, in hardware, of spinal decoders that can handle large numbers of paths.

Integrating spinal codes with Airblue is straightforward, due to the latency-insensitive discipline used by Airblue. From the perspective of Airblue, a spinal decoder or decoder is simply a replacement for the mapping and error correction blocks in the baseline transceiver of Figure 8-1. Interfacing to the remainder of the OFDM stack requires nothing more than a few lines of marshalling code to match up the types of the spinal hardware to the stack. Once integrated into the Airblue framework, the spinal decoder can be targeted at the USRP platform for on-air testing.

With small, low-quality spinal decoders, the experimental setup is complete: the basic Airblue system with a single FPGA is sufficient. However, small spinal decoders are uninteresting because they are highly inefficient; on the other hand, large spinal decoders do not fit in a single FPGA. For larger decoders, the spinal transceiver must be partitioned between two FPGAs for on-air testing. This pipeline partitioning is shown in Figure 8-7. The pipeline is cut in half between the spinal decoder and the remainder of the OFDM stack and placed on two FPGAs. Creating this multiple FPGA implementation is easy using the LIM compiler: the spinal transceiver is simply targeted to a two FPGA environment instead of a single FPGA environment and recompiled. This transformation requires no editing of the transceiver source, and less than ten lines of configuration code to achieve the re-targeting.

Figure 8-8 shows the on-air operation of a partitioned implementation of the spinal transceiver. This experiment was conducted using a pair of Airblue systems communicating over a 10MHz channel in a controlled laboratory setting. The operation shows that the software simulation of the spinal algorithm is actually quite close to the behavior of the actual hardware implementation on the FPGA, validating the software simulator, which used a much simpler channel model and an idealized OFDM stack.

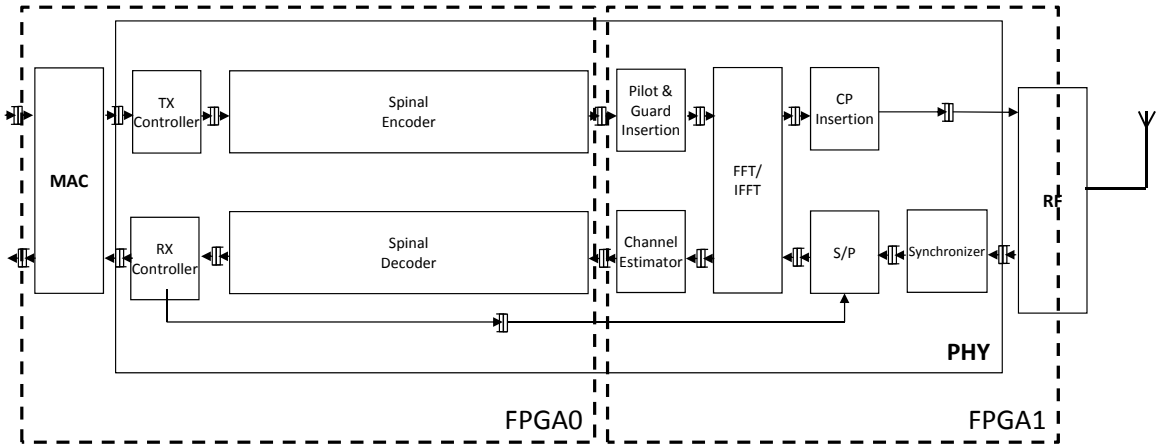


Figure 8-7: A spinal transceiver partitioned among two FPGAs.

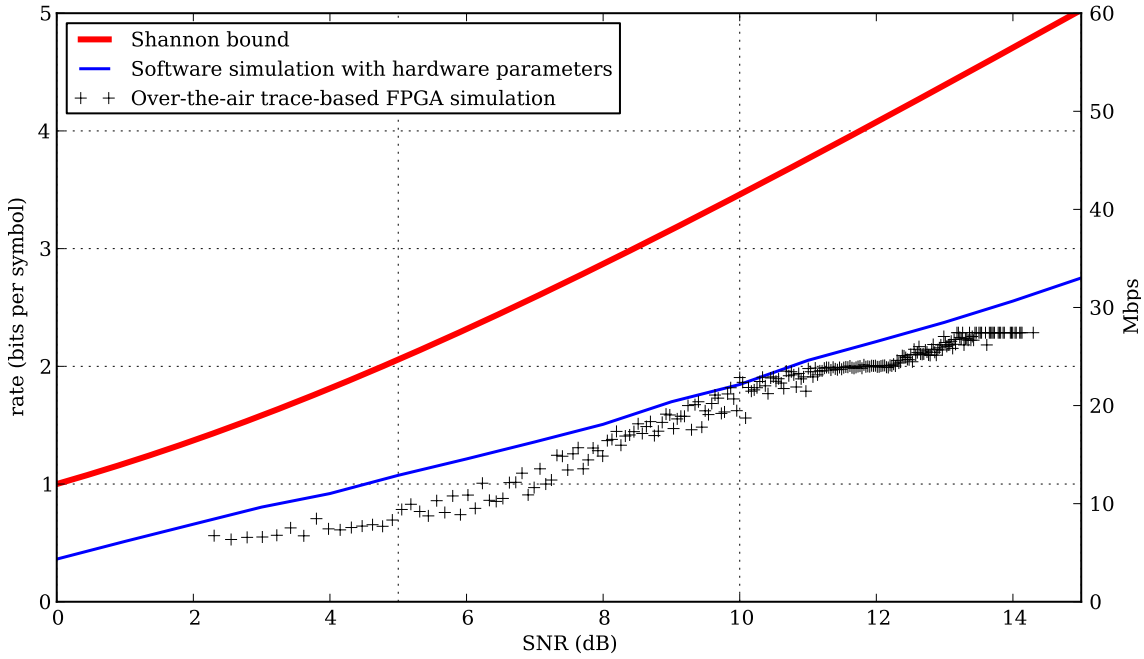


Figure 8-8: Over-the-air tests validate the hardware spinal decoder and its software model. Spinal over-the-air testing was enabled by multiple FPGAs.

8.3 WiLIS: High-speed Simulation of Wireless Pipelines

Although Airblue was intended to assist protocol designers in evaluating protocols, it is also a useful tool for hardware designers. As described in Section 8.1, a wireless transmitter sends data to a receiver by modulating some carrier signal with a signal that represents the digital data being transmitted. The receiver recovers the data from the on-air signal through a reverse process called demodulation. Unfortunately, the carrier signal observed by the receiver is perturbed by various physical phenomena such as noise, interference, and fading. In order to permit reliable transmission, both modulation and demodulation involve applying various types of algorithms in series to minimize the impact of these physical phenomena.

Most wireless protocols are intended to be able to recover data even when the signal is severely corrupted, and in some sense, protocol behavior under this regime is of the greatest interest to protocol designers. On the other hand, for hardware engineers, validating the behavior of a particular piece of hardware requires ensuring correct behavior under a broad range of operating conditions, including conditions under which errors may be uncommon. From the perspective of validation, the "good" operating point is the most challenging case because it requires the greatest amount of simulation. For example, evaluating a new error correction block requires verifying expected operation at bit-error rates (BER) as low as 10^{-9}, an operating point at which the vast majority of bits are received correctly. To achieve reliable measures for an algorithm, it is necessary to produce a statistically significant number of these very uncommon events, a computationally intensive task.

One approach to validation is to simply operate on the air. This sort of simulation is fast, but it has the serious drawback that it is nearly impossible to control the broadcast environment, rendering experiments irreproducible and coverage difficult to guarantee. As a result validation engineers go through great effort to develop controllable channel models in software. These models typical involve the heavy use of complex floating point routines, for example to produce noise.

Unlike the on-air implementations, which only need to meet protocol timings, a

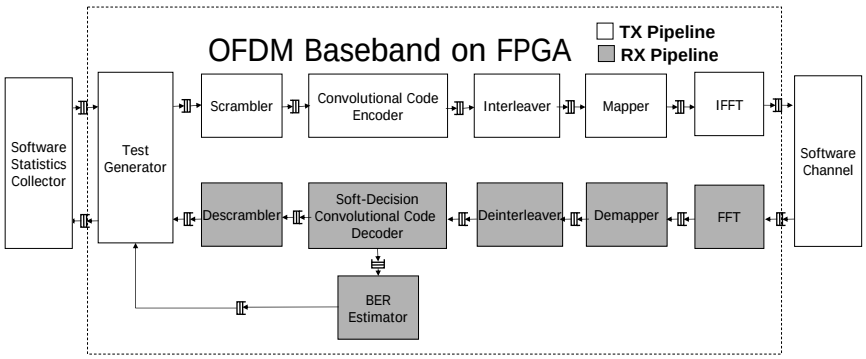


Figure 8-9: Components required to validate a BER estimator in a co-simulation environment.

simulator needs to provide maximum throughput to speed architectural evaluation. The need to model billions of bits suggests that FPGA-based implementation is necessary: software-based simulations of hardware process only a few kilobits per second. Parallel simulation across dozens of machines is not sufficient to produce enough of the rarer events for accurate characterization. However, accelerating the whole test-bench on an FPGA is also problematic because the channel model is not amenable to hardware implementation. Because the original Airblue modules were designed in a latency-insensitive manner, it is straightforward to tie the hardware simulator of a wireless transceiver to a software channel model: the Airblue hardware automatically stalls while waiting for long-latency software operations. Thus, a wireless latency-insensitive simulator (WiLIS) can leverage hardware-software co-simulation, which accelerates the simulation of the hardware pipeline using FPGAs but keeps the complex channel model implementation in software. This architecture is shown in Figure 8-9.

Many WiLIS test-beds can be accelerated by leveraging multiple FPGAs. SoftPHY, which extends commercially deployed error correction schemes, presents one such example. The SoftPHY abstraction [31, 63] offers a solution to the problem of fine-grained bit-error rate (BER) estimation. BER estimations are valuable in optimizing the throughput of a wireless network, but have been traditionally difficult to obtain because of rapid time variations in transmission conditions. SoftPHY makes use of a soft-decision convolutional-code decoder to export a confidence metric, the

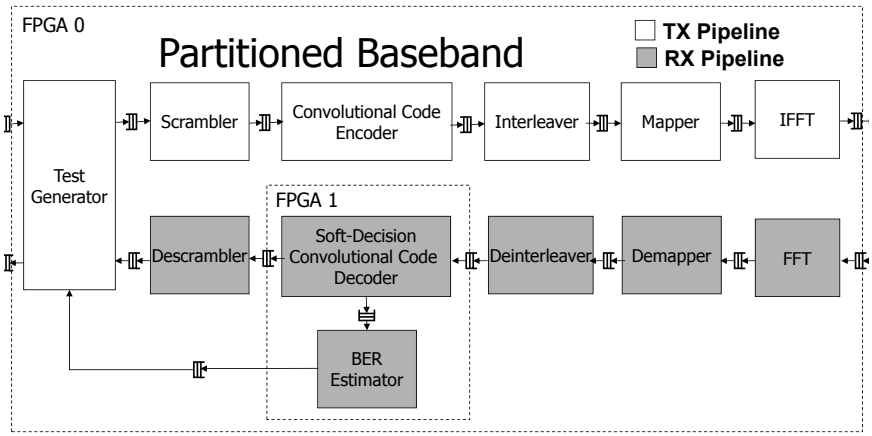


Figure 8-10: BER estimation simulator, partitioned across two FPGAs.

log-likelihood ratio (LLR) of a bit being one or zero, up the networking stack. SoftPHY has many applications in a wireless pipeline, including rate adaptation: if bit confidences are too low, the SoftPHY MAC will lower the transmission data rate. Conversely, it will raise the transmission data rates if bit confidences are too high.

Figure 8-10 shows a good multi-FPGA partitioning of the SoftPHY evaluation system. In this partitioning, only the device-under-test, that is the SoftPHY convolutional decoder, is placed on FPGA 1. The remainder of the system is placed on FPGA 0. Multiple-FPGA partitioning benefits the SoftPHY micro-architectural simulator in two ways. First, because only the microarchitecture of the error correction algorithm varies, by partitioning the simulator at the error correction algorithm, the bulk of the hardware simulator needs to be compiled only once. To test a different algorithm, a relatively small logical change, only the implementation of the second FPGA needs to be rebuilt. Second, because the clock frequencies of the FPGAs can be scaled independently, the wall-clock performance of the simulator improves. This frequency scaling occurs due to the availability of more resources on multiple FPGAs, as the synthesis tools have greater freedom to choose area-intensive but timing-optimized hardware implementations.

Figure 8-11 shows the normalized performance of two experiments: one using a complex software channel model and the other using a simpler hardware channel model. In the first experiment, the software channel model is the performance bot-

Modulation	Simulation Speed (Mb/s)
BPSK 1/2 (6 Mbps)	2.033 (33.9%)
BPSK 3/4 (9 Mbps)	2.953 (32.8%)
QPSK 1/2 (12 Mbps)	4.040 (33.7%)
QPSK 3/4 (18 Mbps)	6.036 (35.3%)
QAM-16 1/2 (24 Mbps)	8.483 (35.3%)
QAM-16 3/4 (36 Mbps)	12.725 (35.2%)
QAM-64 2/3 (48 Mbps)	15.960 (33.2%)
QAM-64 3/4 (54 Mbps)	22.244 (41.3%)

Table 8.1: WiLIS simulation speeds of different rates. Numbers in parentheses are the ratios of the simulation speeds to the line-rate speeds of corresponding 802.11g rates

tleneck and limits the throughput of both the single and multiple FPGA implementations. In this case the multiple FPGA implementation achieves near performance parity with the single FPGA implementation, even though it has a much higher clock frequency. For one data point, QAM-64, the multiple FPGA implementation slightly outperforms the single FPGA implementation. This is because QAM-64 produces more bits per software communication and begins to overwhelm the serial portions of the slower single FPGA implementation.

When a simpler channel model is implemented in hardware, the multiple FPGA implementation outperforms the single FPGA implementation by a wide margin. In this case, the normalized performance is tied to the clock frequencies, listed in Table 8.2, of the two designs. For BPSK, which stresses the FFT, the ratio is highest, since the FFT is located on FPGA 0 in the partitioned implementation and this FPGA enjoys the largest frequency improvement. For higher bit-rate modulation schemes, the bit-wise forward error correction, located on FPGA 1, is the bottleneck. Since the ratio of the clocks of the single FPGA implementation and FPGA 1 is smaller, the performance gap narrows.

In addition to improving experiment throughput, multiple FPGA compilation has another, pragmatic advantage in the context of WiLIS: compilation time. Compilation time can be a major headache in large FPGA-based systems, to the point of

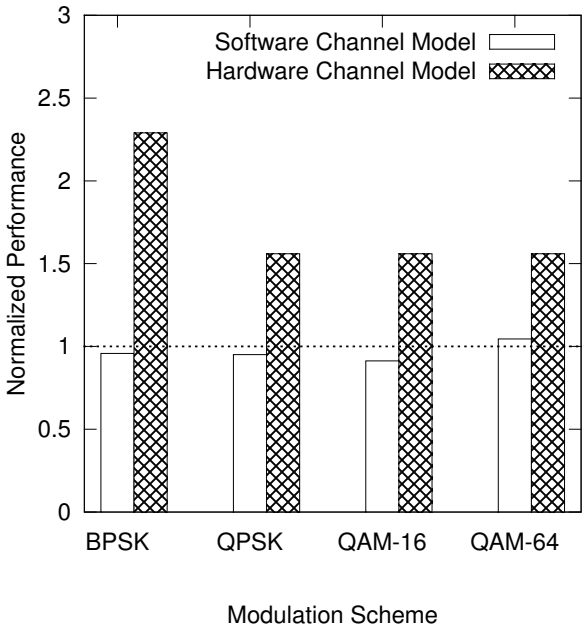


Figure 8-11: Performance results for SoftPHY partitioned simulation. Performance is normalized to a single FPGA implementation.

	LUTS	**Registers**	**BRAM**	**fMax(MHz)**
WiLIS, SOVA, Single	115780	67975	46	25
WiLIS, SOVA, FPGA 0	77982	56499	34	65
WiLIS, SOVA, FPGA 1	46852	21707	39	45

Table 8.2: Synthesis metrics for single and multiple FPGA implementations of WiLIS. Xilinx 12.1 was used to produce bit-files. To limit compile times, fMax steps at increments of 5MHz.

being a rate limiting step in hardware experimentation. WiLIS experiments typically run for a few hours, but in some cases single-FPGA compilation can take nearly a day, especially when aggressive frequencies are targeted. Because only a small part of a WiLIS test-bench is modified for any given microarchitectural experiment, and that portion is partitioned on the second FPGA, only the small design resident on the second FPGA needs to be recompiled for each experiment. In the case of the SoftPHY experiment, partitioned compilation is nearly four times faster than compilation of the same source for a single FPGA.

8.3.1 Airblue on Multiple FPGAs

To meet the ever-growing demand for bandwidth from mobile devices, researchers have developed increasingly complicated wireless protocols. Good evaluation of these protocols requires FPGA implementation, but implementations of complex protocols, such as spinal codes, may not fit onto a single FPGA. Thus, the chief benefit of applying multiple FPGA compilation to Airblue is enabling the construction of these complex transceivers.

In addition to providing a means of implementing complex transceivers, multiple FPGAs are also helpful in evaluating individual transceiver blocks. Testing a wireless device requires modelling a large portion of the wireless pipeline, even if the device-under-test itself is small. FPGAs can help mitigate the computational complexity of this simulation, but if there are many different devices parameterizations to test, FPGA compilation times can be problematic. By placing the device-under-test on an FPGA separate from the remainder of the test harness, rapid recompilations are enabled. In addition, in the case that small blocks are being tested, multiple FPGAs can significantly accelerate simulation throughput due to increases FPGA resource availability.

Chapter 9

HAsim

Simulation is critical to the design of modern processors. Before building a new processor, architects must choose among a set of possible architectural and microarchitectural features with the goal of improving system performance across important programs. The difficulty in this decision making process is twofold. First, architects must explore a broad parameter space. Second, because mistakes can be costly in terms of performance and, ultimately, commercial viability, architects typically demand experiments that faithfully model the proposed architecture in detail. As a result, architects typically turn to detailed software-based simulators [14, 16] to evaluate new architectural ideas. Unfortunately, detailed software simulation is extremely slow: in some cases it can be less than one KIPS in aggregate. The slow speed of simulation coupled with the need to run whole programs, billions to trillions of instructions in length, constrain the exploration of new architectural features to the point of limiting technological advance.

Recently, several relaxed software simulators [38, 19, 68, 49] have become available. These simulators trade varying degrees of accuracy for greatly decreased simulation times. For example, the interval model [19] seeks to model only the occurrence of important processor events, such as cache misses, TLB misses, and branch mispredictions, which have significant impact on core performance. When these high-penalty events do not occur, interval simulation assumes sustained core IPC equal to the width of the core. Upon the predicted occurrence of such an event, the interval

model reverts to detailed simulation for the duration of the event. Because performance impacting events are rare, interval based simulation can result in simulation performance increases of around an order of magnitude as compared to conventional full-detail simulation. Another example, SMARTs [68, 49], attempts to exploit homogeneity in programs and statistical sampling to reduce the amount of detailed simulation required to infer whole program results. SMARTs runs the entire program using a low-detail simulation, and then for each microarchitecture under test, runs a collection of short but detailed simulations using the state collected during the low-detail simulation as a starting point. This approach achieves performance gains proportional to the amount of down-sampling, which, in some cases, can be orders of magnitude.

Relaxation comes at the cost of simulation fidelity. Recent studies have found that interval simulation produces errors on the order of 10% in multi-core simulation, while sampling-based methodologies inherently report a confidence interval in which the parameter of interest may lie. Since performance gains due to microarchitectural optimization are frequently less than 10%, inaccurate simulators are treated with a healthy dose of skepticism. This is particularly the case in industry because simulator errors can lead to wrong design decisions and subsequent revenue loss. Pellauer et. al. [44] and Khan et. al. [35] have shown that lack of fidelity in the core model can lead to substantial variation in experimental results, even if only network processor network performance is considered.

Another option for architectural exploration is to simulate full-system RTL on an FPGA. Full RTL simulation [2], while fast, is only available late in the design cycle, often too late to be of practical use in making design decisions. Full RTL is also costly to develop and to change, even in the context of modifying high-level architectural parameters. In spite of these shortcomings, full-system RTL is still frequently mapped to FPGA, where it serves primarily as a verification tool. Processor RTL usually targets next-generation ASIC technology and, as a result, may not fit on a single FPGA. Indeed, this sort of verification simulation is the primary use case of the legacy multiple-FPGA tools discussed in Chapter 1.

Recently, several researches have been conducted into FPGA-based performance modelling [35, 44, 9], which seeks to combine the performance of the FPGA with some of the ease of use and programmability of the software performance models. FPGA-based performance models are similar to software performance models in design. Rather than implement a given microarchitecture in RTL, a requirement for verification and for ASIC production, FPGA performance models emulate only the cycle-accurate behavior of the original microarchitecture. As a result performance models avoid many of the the challenging and time-consuming issues associated with production circuit design, including high-frequency timing closure and extreme area optimization. For example, a multi-ported register file may be implemented serially as a sequence of accesses to an on-chip SRAM. Because performance models admit FPGA-efficient micro-architectures, full chip-multiprocessor (CMP) models can be realized on a single FPGA. On Virtex-5 family chips, Khan et al. [35] report two processor cores per FPGA, while Pellauer et al. can model up to a sixteen-core CMP on a single FPGA. In contrast, full-system emulation CMP RTLs [2, 53] may require dozens of FPGAs.

Multi-cycle micro-architectures, though FPGA-efficient and easy to code, come at a cost: FPGA clock cycles no longer correspond precisely to model clock cycles. To help FPGA-based performance models keep track of model time, several time-dilation techniques have been proposed. A-Ports [45] and latency-insensitive bounded dataflow networks (LI-BDNs) [61] both permit the differentiation of model cycles and FPGA cycles, allowing FPGA-optimized performance models to maintain cycle-accuracy with respect to the target processor architecture.

In general, FPGA-based processor performance models have excellent performance: full-system throughputs of more than 5 MIPS are common, even for large multi-core models, and some models [35] have reported throughputs in excess of 50 MIPS. From a programmability perspective, most performance models are well-parameterized, and producing a range of experiments can be as simple as testing a range of dynamic parameters.

FPGA-based performance models are quite promising, but they still face scalabil-

ity issues. Although single FPGA implementations of 16-core CMPs are impressive, academic and industrial computer architects are now considering the design of processors with hundreds or thousands of cores. Producing performance models on that scale requires multiple FPGA implementation. The remainder of this chapter will consider the HAsim family of FPGA-based performance models [44], concluding with the demonstration of a multiple-FPGA implementation of a 128-core CMP.

9.1 Anatomy of HAsim

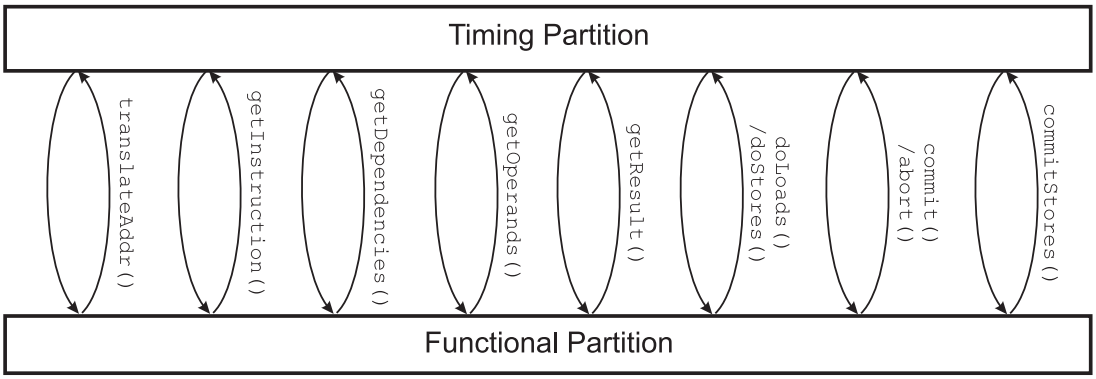


Figure 9-1: HAsim partitioned processor simulator. The timing partition relies on the functional partition for all computation related tasks, for example, instruction decoding. All inter-partition communication occurs over latency-insensitive channels.

HAsim is a framework for constructing high-speed, cycle-accurate, FPGA-based CMP simulators. By utilizing the FPGA, which has good support for the fine-grained parallelism common in processors, HAsim can simulate processors at speeds several orders of magnitude faster than detailed software models. This section provides a brief overview of the architecture of a HAsim simulator. Although this description will cover most major blocks, a full description of the operation of HAsim is beyond the scope of this thesis. A complete description and characterization of HAsim can be found in [48].

HAsim is written in a highly parameterized fashion, both in terms of the structure and the number of the cores modeled. HAsim models can scale to hundreds or thousands of cores by changing a handful of parameters, an important feature

for modeling future processors. The difficulty in modeling such large processors is that, even though describing the models using HAsim is straightforward, the models themselves do not fit in a single FPGA.

Structurally, HAsim is divided into a functional partition and a timing partition, which separates the calculation of architectural state updates from the amount of time that those updates take to calculate in the modeled processor [46]. The functional model serves as an idealized architecture - it takes PC values as inputs and executes instructions, maintaining both the architectural state and memory. The goal of the functional model is to execute instructions as fast as possible.

In contrast to the functional partition, which manages the architectural state of the model and the flow of instructions, the timing model manages the control flow of the processor itself. Based on projected cycle timings, the timing model makes decisions about when (in model time) to issue instructions, manages branch prediction, and handles speculative operations. The timing model also manages the state of the functional model. For example, the timing model tells the functional model when to make speculative memory operations visible (`commit`) and when to roll back execution upon mis-speculation (`abort`). Only the timing model must manage the target processor timing.

Like many FPGA-based processor models, HAsim timing models use multiple FPGA cycles to simulate one model cycle of the target processor. To resolve the timing disparity between FPGA and model, the FPGA-cycle-to-Model-cycle Ratio (FMR), HAsim uses a technique called A-Ports [45]. A-Ports are kind of dynamic dataflow calculus in which processor components communicate solely by means of A-Ports. Each component of the processor must emit a token on each of its output A-Ports in each model cycle. Components do not continue to the next model cycle until all input tokens have been received. By tracking the total number of A-Ports messages received, HAsim can resolve the precise timing behavior of the target processor. Because A-Ports are fully distributed, model components can execute as soon as their inputs are ready, allowing different modules in the processor model to simulate at different and dynamically-variable FMRs. Because it makes use of

dataflow-like, latency-insensitive A-Ports, HAsim is highly amenable to the multiple FPGA implementation using the LIM compiler. Indeed, the original A-Ports protocol implementation makes use of latency-insensitive channels, in particular, Soft Connections, internally.

To improve FPGA implementation efficiency, many operations in HAsim take multiple cycles to complete. If HAsim modelled a single processor in this way, its performance would be terrible, with each model cycle taking scores of FPGA cycles to complete. Worse, most of the simulation pipeline would be guaranteed to be idle in all FPGA cycles. Fortunately, most interesting modern architectures have multiple cores. HAsim exploits this design trend by time-multiplexing its processor simulation infrastructure among many cores, in a manner similar to simultaneous multi-threading (SMT). In HAsim models with multiple cores, different components of the processor model may operate on different model cores simultaneously. Thus, hardware utilization is high and the entire model is continuously active. Time-multiplexing also helps to hide the latency of costly, but infrequent operations such as floating point operations, because many other processors must execute before the result of the long-latency operation is required. The time-multiplexed approach also has the advantage of conserving FPGA implementation area. HAsim can implement a 16-core CMP model, while cycle-accurate modelling infrastructures that opt for direct instantiation of cores, such as Khan et al. can model many fewer cores per FPGA.

Figure 9-1 shows the interface between the timing and functional partitions. This interface permits the timing model to both control the behavior of the functional model, e.g. choosing instructions to execute, and to query the functional model about the status of the instructions, e.g. obtaining instruction dependencies. This general interface permits a single functional model to service many different timing models: when modelling a new processor microarchitecture, only the timing model must be modified.

As compared to OFDM and H.264, which will be discussed in the next chapter, HAsim has relatively tight feedback loops, and these loops occur throughout the simulation pipeline. For example, the timing partition must query the functional

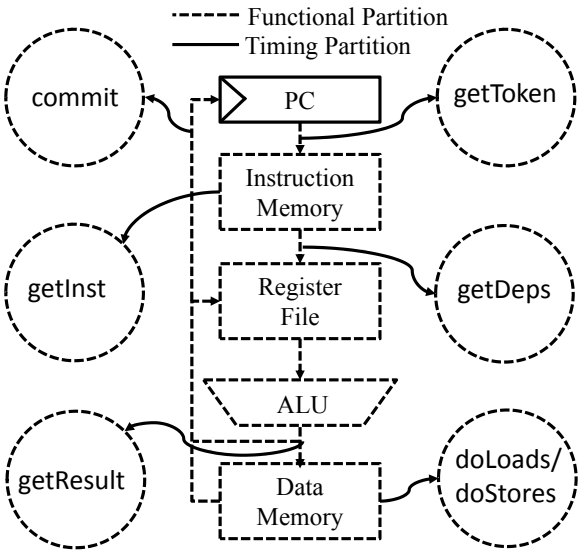


Figure 9-2: HAsim un-pipelined processor timing model. Some control paths and multiplexors have been omitted to simplify the diagram.

partition to decode an instruction and wait for a response before proceeding. Similar feedback loops arise in the other processor stages and in the cache model. Despite this level of feedback, a natural mapping of HAsim to two FPGAs is placing the timing and functional partitions on separate FPGAs.

The following sections offer a brief discussion of major blocks in a HAsim simulator. Because HAsim is a time-multiplexed simulator, many of the interesting microarchitectural features of HAsim center around supporting large numbers of multiplexed cores.

9.1.1 Timing Partition

Figure 9-2 shows the timing model of a single cycle processor. Only the PC is part of the timing model: all other functionality is managed by the functional partition. The timing model calls the interface methods of the functional model to execute instructions. In this model, the HAsim functional partition interface methods are called, one at a time, to model a single processor cycle, culminating in an update to the timing model PC. All interfaces to the functional partition are request-response and may take several cycles to complete. If a method is not required for an instruction

then it need not be called. For example, not all instructions touch the data memory, so `doLoads` and `doStores` may not be called in every model cycle.

Although the example of Figure 9-2 shows an extremely simple uniprocessor, HAsim also provides libraries for modeling modern CMPs, several of which have interesting micro-architectures. HAsim models networks-on-chip by multiplexing a multi-ported router among all simulated cores. Router state is stored in a single, on-FPGA SRAM, and accessed via a stored permutation. Different network topologies can be modelled by changing the permutation function. HAsim also supports the modeling of a full cache hierarchy. Because the timing partition does not maintain state, the cache model needs only to maintain tag information allowing it to determine whether a request hits or misses.

9.1.2 Functional Partition

The HAsim functional partition resembles a large, out-of-order SMT machine, wherein state elements are replicated in FPGA on-die SRAM and the processor data path and memory interface are multiplexed among all modelled processors. These SRAM tables are managed by passing references to in-flight instructions between the timing and functional models. Since HAsim admits out-of-order and speculative execution, the functional model must also maintain and roll back uncommitted speculative state. The functional model uses a map table and free-list based out-of-order approach to achieve this capability.

Although the functional model implements an idealized processor architecture, it differs from a real processor in several fundamental ways. Unlike a real processor, instruction execution in the HAsim functional partition is demand-driven, based on requests from the timing partition. Although the functional partition is deeply pipelined, it is not bypassed, and it relies on independent instruction streams from independent cores to achieve a high degree of parallelism.

9.2 Scaling HAsim: 128 Core Models

On a single FPGA, HAsim scales to 16 cores before the largest available Xilinx Virtex-5 FPGA runs out of resources. By mapping HAsim to two FPGAs, HAsim is able to model CMPs with up to 128 cores. HAsim achieves super linear scaling in problem size because many structures in HAsim are either time-multiplexed among all cores or scale logarithmically with the number of cores.

When mapping HAsim to two FPGAs, the best performing partitioning places the timing and functional partitions on separate FPGAs. This timing-functional partitioning works well for two reasons. First, the timing-functional interface is intrinsically pipelined due time-multiplexing among cores. Individual logical cores can tolerate inter-chip communication latencies because they must already wait for other cores to execute. Better, this tolerance scales as the number of simulated cores increases. Second, the timing model does not actuate all interfaces provided by the functional model on every model cycle. Thus, the timing-functional channels generally have lower loading than either timing-timing or functional-functional channels.

The primary advantage in mapping HAsim to multiple FPGAs is in scaling to larger designs. However, HAsim also benefits from the increased memory resources available on multiple FPGA platforms. HAsim fundamentally runs software programs, and these programs require high-bandwidth, low-latency memory caches to execute quickly. Thus, large HAsim models need large amounts of fast memory. Partitioning HAsim designs among multiple FPGAs automatically introduces new chip-level resources, like DRAM, into the synthesized implementation, increasing both cache capacity and memory bandwidth.

To evaluate the LIM compiler, HAsim models for the family of CMPs described by the parameters in Table 9.1 were generated and tested by running a mix of SPEC2000 integer and floating point applications in parallel on the modelled cores. Although these models are similar to a realistic CMP, two details are missing from the models. First, the model implements only a single memory controller, while modern CMPs typically have at least two. Second, while HAsim supports detailed cache-hierarchy

HAsim Parameter	Value
Core	
ISA	64-bit Alpha
Pipeline Stages	9
Branch Policy	Predict/Rollback
Outstanding Memory Requests	16
Address Translation	Full TLB
L1 Cache	
Associativity	Direct Mapped
Size	16KB
Outstanding Misses	16
L2 Cache	
Size	256KB
Associativity	4
Outstanding Misses	16
On-chip Network	
Topology	2-D Mesh
Routing Policy	X-Y DO Wormhole
Virtual Channels	2
Buffer per Channel	4

Table 9.1: HAsim Model Configuration.

modelling, it does not currently support cache coherence. In spite of these omissions, HAsim remains a useful tool for architectural research.

The performance of various HAsim implementations is shown in Figure 9-3 and Figure 9-4. For small numbers of cores, the gap between the single FPGA and multiple FPGA simulator is large, due to the request-response latency between the timing and functional partitions. However, as the number of simulated processors scales, models become more latency tolerant, and the performance gap closes. For the 64-core simulator modelling both 25 and 36 core processor designs, the partitioned simulation produces more aggregate MIPS than a single FPGA implementation.

HAsim's performance can be measured in two ways: the speed a which it runs programs (instructions per second (IPS), Figure 9-3) and the speed at which it models the target architecture (FPGA cycle to model cycle ratio (FMR), Figure 9-4). The main difference in these two measures is how they are affected by the modelled architecture itself. If the modelled target architecture is slow or poorly designed, even a fast simulator will have low IPS. On the other hand, poor architectures usually

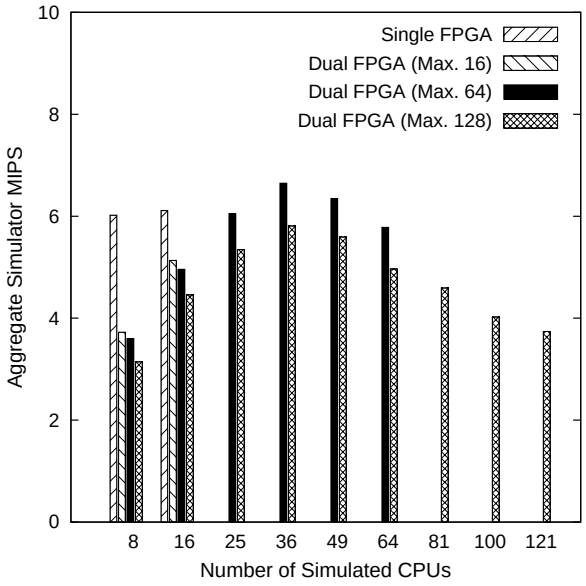


Figure 9-3: Simulator program run-times in MIPS various HAsim simulation configurations. Simulated cores run a combination of wupwise, applu, gcc, mesa, mcf, parser,perlbmk, and ammp from the SPEC2000 suite.

spend large amounts of time idling, which takes little time to simulate. IPS is considered the more relevant measure, since architects ultimately require full-program simulation, but both are included to demonstrate that at least some of the performance degradation IPS seen in the larger FPGA models is caused by a poor memory architecture in the target CMP.

Although partitioned implementations of HAsim have lower throughput than single FPGA implementations, the performance gap is quite narrow. The partitioned 16-core processor model achieves about 80% of the aggregate IPS of the 16-core single FPGA implementation, largely due to the latency of communication between chips. The inter-chip latency is also reflected in the FMR of the partitioned implementation, which is twice as large as the single FPGA implementation. As the number of cores scales in the partitioned models, IPS throughput increases until 36 cores. The reason for this throughput improvement is that larger numbers of cores in a time-multiplexed model are more resilient to inter-link latency. The FMR of the partitioned models flat-lines after 25 cores, suggesting that the latency of the inter-FPGA link is completely hidden. After 36 cores, the IPS throughput of HAsim begins to decline. This

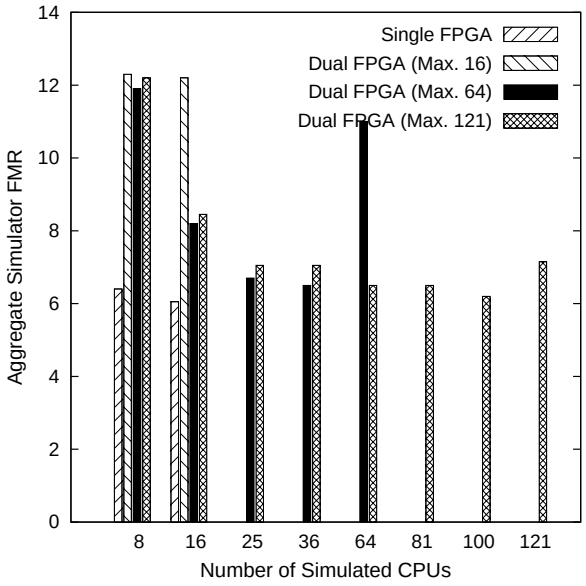


Figure 9-4: HAsim simulator FMR tend to saturate for large numbers of cores, suggesting a performance bottleneck in HAsim. Cores run the same program mix as in Figure 9-3.

decline in IPS is due to flaws in the underlying architecture: network and memory contention at the chip level begin to dominate, reducing the performance of the target architecture. As a result models take more cycles to execute. On the other hand, FMR remains relatively stable, even in in the case of large numbers of cores. This FMR stability suggests that some component of HAsim is saturated and that HAsim itself, as opposed to the multiple FPGA implementation generated by the LIM compiler, is bottlenecking simulation.

Exposing more FPGA resources to HAsim, in addition to increasing the maximum number of cores that can be modelled, has a positive effect on the operating frequency of HAsim because each FPGA has excess area available for placement. The multiple FPGA partitioning of the 16-core simulator can be clocked at 7% and 12% faster than the single FPGA implementation. However, as the maximum number of simulated cores increases, the FPGA becomes more crowded, reducing operating frequency. As a result, partitioned models supporting more cores have lower performance when simulating the same number of cores as a smaller model.

Section 9.1 discussed the division of HAsim-based simulators into timing and

	LUTS	Registers	BRAM	fMax(MHz)
HAsim, 16 cores, Single	156351	153906	127	70
HAsim, 16 cores, FPGA 0	99331	94387	75	75
HAsim, 16 cores, FPGA 1	114403	107288	60	80
HAsim, 64 cores, FPGA 0	112980	98382	111	65
HAsim, 64 cores, FPGA 1	132670	108958	133	70
HAsim, 128 cores, FPGA 0	127911	104391	245	60
HAsim, 128 cores, FPGA 1	153721	116421	174	65

Table 9.2: Synthesis metrics for single and multiple FPGA implementations of HAsim. Xilinx 12.1 was used to produce bit-files. To limit compile times, fMax is stepped at increments of 5MHz.

functional partitions. This partitioning permits a single, highly optimized functional partition with a fixed interface to support many different timing models. If, in a multiple FPGA partitioning, the timing and functional models are placed on different FPGAs, the generality of the functional model implies that the functional partition need be compiled for FPGA only once, since its interface is constant across all timing models. Once the functional model is compiled, all subsequent timing models can reuse it. Since the functional model is larger than most timing models, this represents a significant compilation time savings. Indeed, compiling the timing model alone represents up to a 40% savings in overall compilation time.

9.2.1 HAsim on Multiple FPGAs

As in the case of Airblue, the chief advantage of implementing HAsim on multiple FPGAs is problem size scaling. FPGAs cannot match the area-efficiency of silicon implementation. Even if FPGAs did match the area efficiency of ASIC, computer architects model machines that are generations ahead of current silicon technology. Therefore most interesting processor models will not fit into a single FPGA. Multiple FPGAs solve this problem by permitting the scaling of models to very large problem sizes. Indeed, scaling the HAsim processor model to two FPGAs permits an order-of-magnitude increase in the number of processors that can be modelled simultaneously.

In addition to problem scaling, mapping HAsim to multiple FPGAs has other

benefits. Increased access to resources, such as memory and FPGA fabric, across chips can accelerate HAsim models. In some configurations, a two-FPGA partitioning of HAsim has higher throughput than the peak throughput of a single FPGA implementation. Additionally, because HAsim timing partitions can share a common functional partition, multiple FPGA compilation may reduce compile times in some cases.

Chapter 10

H.264

The H.264 Advanced Video CODEC is an ITU standard for encoding and decoding video with a target coding efficiency twice that of H.263 and with comparable quality to H.262 (MPEG2) [23, 52]. H.264 enables PAL (720×576) resolution video to be transmitted at 1Mbit/sec. Like other video coding standards, H.264 specifies how to reconstruct video from a bit stream but does not specify how to encode video. H.264 shares many of the techniques used in other video CODECs, including VP8 [62], and is a good representative of the video compression family.

Although recent processors are capable of handling full-frame rate HD video decoding, H.264 remains computationally intensive and most deployments of H.264 include at least partial hardware acceleration, particularly in the power-sensitive mobile space. The computational requirements of decoding H.264 video vary depending on video resolution, frame rate, and level of compression used. Low-end mobile phone applications favor videos encoded in the QCIF format (176×144) at 15 frames per second. At the high end of the spectrum, HD-DVD videos are encoded at 1080p (1920×1080) at 60 frames per second. Complicating these performance requirements are the H.264 standard *profiles*, which use different combinations of compression features. The combination of different performance levels and feature sets suggest that different hardware architectures are needed for different H.264 deployments. Given the importance of video as a medium of entertainment and information transfer, the practical need for hardware acceleration in commercial deployments, and the variabil-

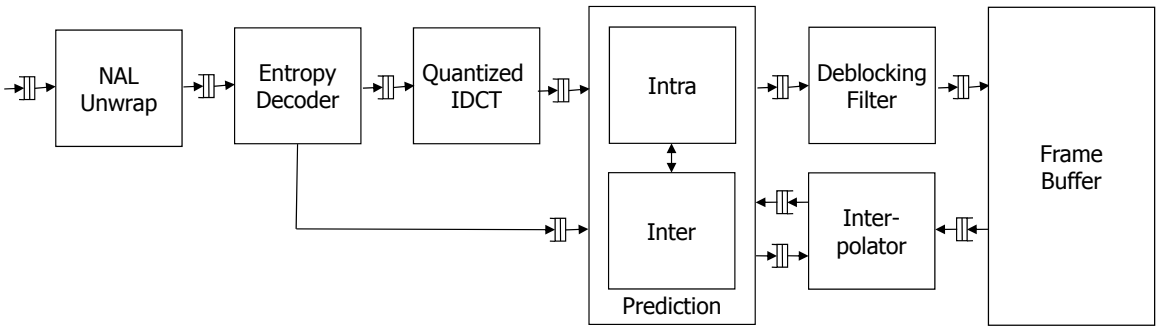


Figure 10-1: An H.264 decoder.

ity of hardware requirements, a platform for research into the architecture of H.264 decoders is valuable.

To facilitate architectural research into H.264, we implemented an FPGA-based decoder [17]. Although H.264 does not have the latency requirements and throughput requirements that force OFDM systems like Airblue to utilize the FPGA for functional correctness, simulating hardware systems the size of H.264 is extremely slow in software. When conducting design-space explorations across a significant parameter space, for example, when tuning cache parameters, FPGA-acceleration is help in quickly testing parameters. This implementation effort led to the development of several important aspects of the design of latency insensitive systems for architectural research and for the FPGA, including the concept of a decoupled memory hierarchy with a uniform, split-phase interface. This interface will form the foundation of the portable, automatically generated memory system described in Chapter 7. H.264 also serves as an excellent study for memory systems for both embedded and reconfigurable implementation substrates.

This chapter will first describe the operational characteristics and behavior of H.264 in Section 10.1. Section 10.2 examines some experiments in to the architecture of a frame buffer cache hierarchy. Section 10.3 describes a multiple FPGA partitioning of H.264 and a handful of microarchitectural results.

10.1 Anatomy of H.264

H.264 reconstructs video at the granularity of 16×16 pixel macroblocks, which may be further subdivided into smaller sub-blocks in some decoding steps. H.264 uses two main techniques to reduce the number of bits necessary to encode video. Intra-prediction predicts macroblocks in a frame from other previously-decoded spatially-local macroblocks in the same frame. Inter-prediction predicts macroblocks from indexed macroblocks in previously decoded frames. Within a coded frame, *slices*, or groups of macroblocks, may be intra-predicted, inter-predicted from the previous frame, or inter-predicted from multiple reference frames. Figure 10-1 shows a block diagram of an H.264 decoder.

The macroblock processing style of yields an interesting pipelined behavior: hardware blocks typically operate for a few dozen cycles on a single macroblock and then pass the macroblock to the next pipeline stage for subsequent processing. The natural latency arising from these sequential block-level operations coupled with non-deterministic processing times in the hardware blocks, requires a latency-insensitive implementation. The size of the latencies between hardware blocks also make H.264 suitable for multiple FPGA implementation.

H.264 is between OFDM and HAsim in terms of feedback dependence. Feedback in H.264 occurs at two levels: the frame and the macroblock. Feedback at the frame level occurs through the frame buffer, as pixels from previously frames are used to decode subsequent frames. Because of the temporal length of this feedback path, it is of limited concern for implementations partitioned across FPGAs. Feedback at the macroblock level occurs between the inter-prediction and intra- prediction blocks. In this case, the inter-prediction module feeds pixel information to the intra-prediction block. Beyond these two feedback paths, the H.264 pipeline is feed-forward. Thus, because feedback is limited, partitioning among FPGAs at most points in the design will not incur a major performance penalty.

The following paragraphs describe the basic behavior of blocks in H.264.

NAL Unwrap: H.264 streams are encapsulated within a transportation packet for-

mat, which permits the mixing of different media types, for example audio. The Network Adaptation Layer (NAL) interprets these packets and extracts the H.264 stream from them, including high-level control information.

Entropy Decoder: In addition to video-specific compression, H.264 makes use of general entropy encoding to further compress video streams. Depending on the performance profile, H.264 applies one of two entropy encoding schemes: CAVLC(Context Adaptive Variable Length Coding) and CABAC(Context Adaptive Binary Arithmetic Coding). Both techniques feature context-aware bit-mappings that vary during decoding. CABAC typically achieves better compression, but is much more computationally intensive.

Inverse Transformation and Quantization: H.264 produces new pixel data via a set predictions based on previously decoded image data. Although these predictions do a good job of recovering pixel information, they do contain small errors. To improve the overall quality of the decoded video, H.264 corrects these predictions with a block of residual-error correcting values representing the difference between the fixed prediction and the original image. This greatly enhances compression, since the prediction modes and residual errors can be concisely expressed.

However, residual terms can be further compressed. Since residual terms exhibit high spatial entropy, H.264 employs a lossy, low-pass discrete cosine transform to develop a compact representation of the residual values. The DCT coefficients of the residual errors are have small magnitudes, allowing the residuals to be expressed in terms of a very small number of frequency components. H.264 also allows variable quantization of DCT coefficients to enhance coding density.

Intra-prediction: Video frames have a high amount of spatial similarity. Intra-prediction use previously decoded, spatially-local macroblocks to predict the next macroblock. Blocks are predicted from these pixels using a set of simple pixel transforms, mostly two-dimensional cosines. For additional compression, H.264 will also predict the intra-prediction transform to use for each macroblock, based on the transforms used by spatially local blocks. Errors in this prediction are also encoded in the data stream.

Inter-prediction: In video, temporally adjacent frames usually have only small differences. Inter-prediction attempts to capitalize on this inter-frame similarity by encoding macroblocks in the current frame using a reference to a macroblock in a previous frame and a vector representing the movement that macroblock took between the two frames. The decoder will attempt to predict both which block to use and how far that the block moved, based on the behavior of previously decoded, spatially-local blocks.

H.264 permits macroblock motion to occur at sub-pixel granularity. Macroblocks with fractional motion vectors are interpolated from multiple previous macroblocks, using several computationally intensive spline calculations.

Deblocking Filter: Since the various compression modes of H.264 operate at the granularity of pixel blocks, significant visual discontinuities can appear at block boundaries. To remove these visual artifacts, H.264 incorporates a smoothing filter into its decoding loop. The deblocking filter applies a series of smooth operations to block boundaries, depending on the kind of prediction mechanism used to produce the particular pixel blocks. However, not all inter-block discontinuities are undesirable; edges in the original image may naturally occur on block boundaries. H.264 incorporates fine-grained filter control to preserve these edges.

Buffer Control: H.264 does not require inter-predicted images to depend on temporally-local or temporally-ordered images. Rather, frames can be predicted from previously decoded frames corresponding to frames far in the past or future of the video. Buffer control maintains this set of previously decoded frames and is responsible for handling the in-stream requests to access the stored frames (e.g. delete, reads from inter-predict, writes from deblocking). Writes to the buffer control are typically in raster order at the granularity of macroblocks. Reads, which occur in inter-prediction, typically touch a few adjacent rows of macroblocks, and exhibit strong data locality.

Stream	QCIF	720p
Input	0.01	2.75
Output	0.50	26.5
Inter-prediction	1.00	67.0

Table 10.1: H.264 bandwidth requirements for streams with variable resolutions(MB/s)

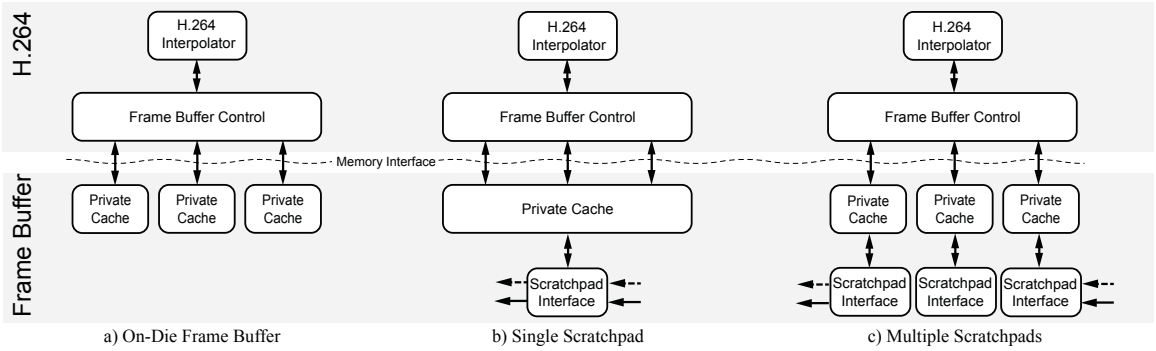


Figure 10-2: H.264 memory architectures. (a) Stores all interpolation data on the FPGA. (b) Stores the same data in a single scratchpad, allowing the decoder to work on larger frame resolutions. (c) Stores the data in multiple scratchpads, increasing memory I/O parallelism and reducing cache conflict misses and head-of-line blocking.

10.2 H.264 Memory Architecture

H.264 can source pixels for prediction in two ways: Intra-prediction predicts macroblocks in a frame from other previously decoded, spatially local, macroblocks in the same frame. Inter-prediction predicts macroblocks from motion-indexed pixels in previously decoded frames. Intra-prediction takes advantage of spatial locality within a frame – colors and textures are similar across a frame. Inter-prediction takes advantage of temporal locality across a frames – adjacent frames tend to depict similar images. In the context of memory management, this second kind of prediction is of the greatest interest, since it necessarily involves retrieving data from previously decoded, stored frames.

H.264 has three distinct memory streams: input, output, and inter-prediction. As shown in Figure 10.1, the inter-prediction, somewhat counter-intuitively, stream dominates memory bandwidth usage. This dominance arises from the method by which blocks are inter-predicted in H.264. Previous standards permitted only whole

pixel motion vectors: inter-prediction of a macro block only involved copying some other macroblock from an existing frame and then applying a residual correction. The bandwidth requirements of this style of inter-prediction must be strictly less than the output bandwidth, since not all blocks are inter-predicted. However, H.264, in addition to legacy inter-prediction schemes, permits sub-pixel motion vectors. To provide better image reconstruction in this case, H.264 applies blurring spline filters to pixels from as many as five different adjacent macroblocks to produce a single predicted pixel. For quarter-sample motion vectors, thirty-six pixels must be accessed to compute a single output pixel. Inter-prediction not only requires high memory bandwidth, but also its access pattern is dynamic. Whole, half, and quarter pixel motion vectors may occur at any point in the frame.

Fortunately, H.264 exhibits a number of properties that can be exploited to develop an efficient memory system. First, the memory streams for the intensity (luminance) and coloration (chrominance) fields are independent. Second, motion vectors tend to be spatially homogeneous, yielding a high degree of regional locality in pixel accesses. Third, inter-prediction feedback occurs at the granularity of a frame, thus simplifying management of coherence across multiple caches.

The original H.264 implementation was designed to target several different performance levels with the idea of producing a minimum-cost implementation for each design point. One key difference in the architectures was the memory subsystem. High-end systems might target a dedicated SRAM frame buffer, while low-end systems might use a shared DRAM. These memory architectures have radically different timings, and accommodating these timing in a latency-sensitive design can precipitate large code changes. To avoid this issue, the original H.264 implementation abstracted away the timing behavior of the various system memories through the use of a uniform, latency-insensitive, request-response interface consisting of three operations: read request, read response, and write. Any memory hierarchy can built under this abstraction layer without impacting the functional correctness of the client hardware, including automatically generated memory hierarchies. This interface, and its implications are discussed at length in Chapter 7.

Because the memory interfaces of the base design were written in this latency-insensitive, request-response manner, augmenting the base implementation with caches requires no code modification to the H.264 core: the cache is simply a memory hierarchy with a dynamically variable latency. Figure 10.3 presents the results of the basic cache experimentation. A direct-mapped, blocking cache was designed and manually inserted between the frame buffer controller and the frame buffer store, with separate caches for the luma and chroma samples. The memory access patterns tested were taken from a low-bit-rate QCIF video stream and a high-definition 720p clip.

It is evident from Figure 10.3 that even a small cache is highly effective in capturing inter-prediction memory locality. With a four-byte line size and two one-kilobyte cache, 46% of luma memory and 30% of chroma memory requests hit in the cache. Using larger cache line sizes more than double the hit rates. The largest gain in caching performance occurs when making the cache large enough to contain the image data of three adjacent macroblocks at the same time, that is, large enough to capture the data of the macroblocks to the immediate left and right of the current macroblock, since these data values are likely to have been recently used (left) or to be used again soon (right). Further hit-rate improvements require caching the rows of macroblocks above and below the current macroblock; some of this benefit is seen by the larger caches in the QCIF experiment.

Caches are clearly beneficial for H.264: with relatively little extra area, the bandwidth of the memory hierarchy is amplified and, perhaps more importantly, power is reduced. However, there are two difficulties in baseline caching experiment. First, while the manual creation and integration of simple cache modules is not difficult, more complicated modules, like a second-level cache, start to present implementation difficulties. Second, from the perspective of multiple FPGA implementation, interfacing to a memory module directly limits the portability of a design. An interesting solution to both of these problems is automatic cache synthesis. Rather than manually developing a cache hierarchy, the request-response interfaces in H.264 can be used to *infer* a hierarchy. Scratchpads [1] are such an automatic memory hierarchy synthesis tool, the details of which are presented in Chapter 7.

Cache Size (8-bit Pixels)	Chroma Hit Rate Line Size (Pixels)				Luma Hit Rate Line Size (Pixels)			
	4	8	16	32	4	8	16	32
128	.177	.346	.431	.363	.032	.394	.511	.573
256	.336	.530	.636	.453	.033	.398	.516	.576
512	.369	.554	.657	.466	.242	.609	.640	.629
1024	.463	.729	.814	.823	.305	.650	.719	.720
2048	.464	.729	.862	.925	.306	.650	.720	.721
4196	.564	.780	.888	.944	.558	.778	.888	.944

Table 10.2: H.264 Inter-prediction Cache Parameter Exploration (QCIF)

Cache Size (8-bit Pixels)	Chroma Hit Rate Line Size (Pixels)				Luma Hit Rate Line Size (Pixels)			
	4	8	16	32	4	8	16	32
128	.082	.373	.509	.578	.013	.247	.330	.371
256	.138	.429	.565	.635	.067	.275	.348	.384
512	.376	.666	.810	.883	.217	.432	.510	.552
1024	.393	.684	.829	.903	.468	.719	.837	.898
2048	.428	.712	.854	.926	.516	.756	.876	.937
4196	.428	.712	.854	.926	.516	.756	.876	.937

Table 10.3: H.264 Inter-prediction Cache Parameter Exploration (720p)

The chief issue in the evaluation of automatically synthesized systems is whether they have performance commensurate to a hand-coded system. The baseline H.264 decoder uses the same interface as the the Scratchpad-based hierarchy, allowing the application of Scratchpads without modifying the core code. Figure 10-2 shows three different memory hierarchies of varying complexity for H.264. The first hierarchy is a simple, shared, on-die, RAM-based frame buffer. This buffer is constrained by both the size of on-die RAM available on the FPGA and offered bandwidth, which must be multiplexed among the three field streams. It is worth noting that this memory architecture is chosen by many published H.264 implementations. The second hierarchy is a single Scratchpad memory hierarchy backed by the central cache. While the architecture makes use of the automatic mid-level cache provided by Scratchpads, the lowest-level cache is still multiplexed among the three pixel streams. The third hierarchy partitions the Scratchpads to implement separate storage for the luma and chroma components. Although this architecture requires a larger number of caches, the bandwidth offered by the hierarchy is substantially higher due to the distribution of requests across caches. Additionally, individual caches experience better perfor-

mance due to the removal of conflict misses. To provide a uniform basis for comparison, the three hierarchies are configured to use similar amounts of on-die memory.

The performance results for the various memory hierarchy implementations are shown in Figure 10-3. Results are not shown for higher resolutions using on-die RAM blocks because they are not synthesizable on the FPGA, due to the size of the on-die RAM frame buffer. Unlike direct implementations employing on-die memory, the Scratchpad versions require only caches on the FPGA and remain synthesizable as the problem size grows. As expected, the memory hierarchy in which only a single Scratchpad is used offers less performance than the hierarchy in which the field memory streams are split across multiple platforms. However, the performance increase is much larger than a factor of three. This larger difference is a result of head-of-line blocking in the single Scratchpad hierarchy. High-latency misses prevent faster, unrelated hits from exiting the in-order memory response queue, thereby stalling the processor pipeline. Figure 10.4 shows the implementation areas of the different memory hierarchies combined with the H.264 memory controller. Because of the complex synthesis process used to build Scratchpads-based hierarchies, it is difficult to separate individual components of the FPGA platform from one another, so an FPGA platform without any memory hierarchy is provided as a reference.

Memory Hierarchy	**Registers**	**LUTs**	**On-Die RAM KB**
Platform Components	9599	11239	8
On-die Frame Buffer	22834	31880	244
Single Scratchpad	53941	66184	208
Multiple Scratchpads	37815	52748	208

Table 10.4: H.264 synthesis results with various cache hierarchies, targeting Xilinx Virtex 5.

10.3 H.264 on Multiple FPGAs

The chief advantage in using multiple FPGAs in evaluating H.264, beyond evaluating the robustness of the multiple FPGA compiler, is to speed compile time. H.264 has

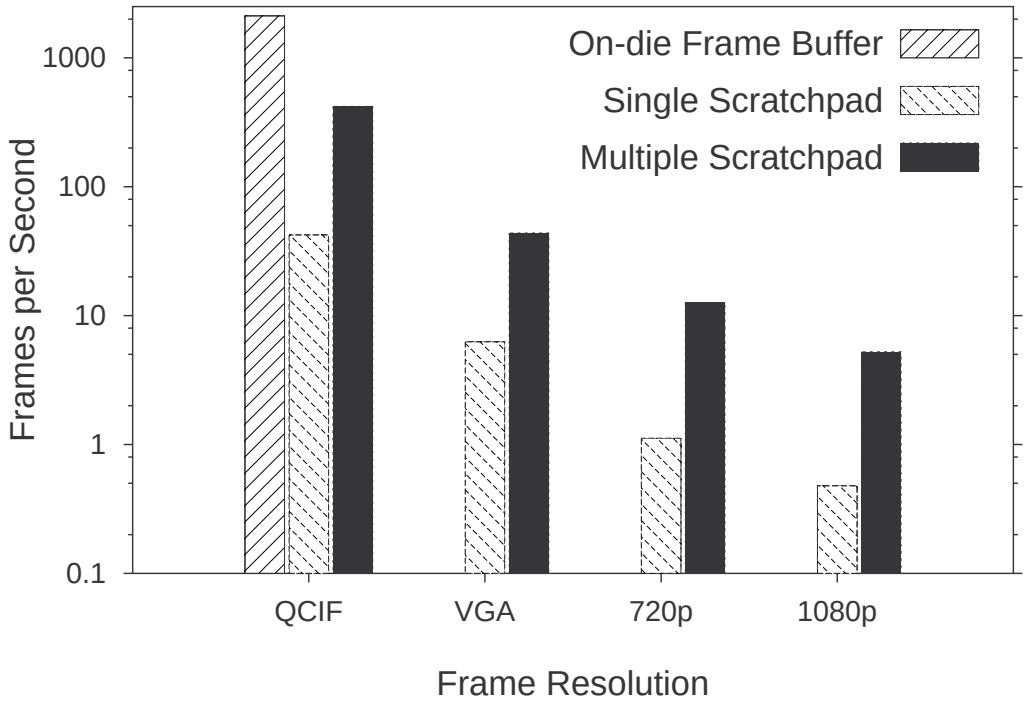


Figure 10-3: H.264 frame-rates for different memory architectures and resolutions, plotted on a log scale.

several potential levels of implementation with widely varying feature sets and performance requirements. When implementing these various feature sets, it is useful to have a platform for rapidly evaluating the performance of different micro-architectures and memory organizations. The lower compile times offered by our compiler are useful in this kind of architectural exploration.

H.264 is naturally decomposed into a bit-serial front-end and a data parallel back-end. The front-end handles decompression and packet decoding, while the back-end applies a series of pixel-parallel transformations and filters to reconstruct the video. H.264 has limited feedback between blocks in the main pipeline. The pipeline synchronizes only at frame boundaries, which occur at the granularity of millions of cycles. Intra-prediction does require some feedback from inter-prediction, but this feedback is somewhat coarse-grained, occurring on blocks of sixty-four pixels.

Because H.264 generally lacks tight coupling among processor modules, many high performance partitionings are possible. As an example, I choose to partition the bit-serial front-end because the front-end computation does not parallelize efficiently.

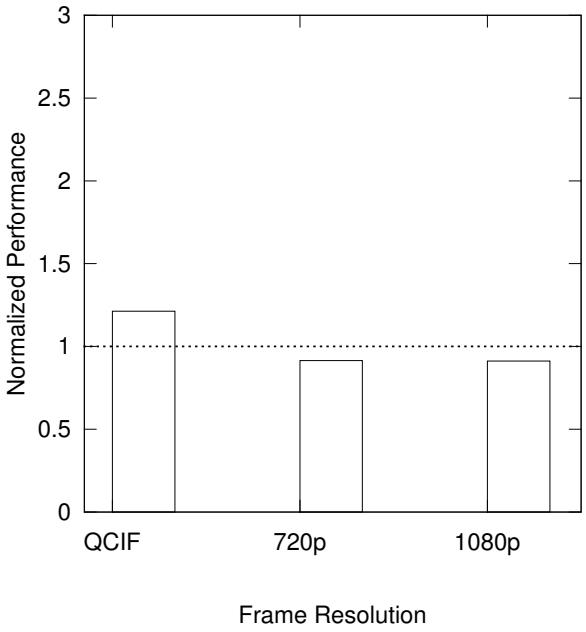


Figure 10-4: Performance results for H.264 implementations mapped across multiple FPGAs. Results are normalized to a single FPGA implementation.

As such, its performance can only be increased by raising operating frequency. The front-end also contains a number of difficult feedback paths, which end up limiting frequency in a single FPGA implementation.

Figure 10-4 shows the performance of a partitioned implementation of H.264 relative to a single FPGA implementation. In the case of the low resolution, the multiple FPGA implementation outperforms the single FPGA implementation by 20%. This performance gain comes from increasing the clock frequency of the partitioned implementation relative to the single FPGA implementation. However, at higher resolution, inter-prediction memory traffic becomes more significant which has the effect of frequently stalling the processing pipeline. As a result some part of the latency of inter-chip communications is exposed and the multiple FPGA performance degrades slightly.

	LUTS	Registers	BRAM	fMax(MHz)
H.264, Single	79839	59212	63	55
H.264 FPGA 0	66893	52860	65	65
H.264 FPGA 1	13998	9493	19	85

Table 10.5: Synthesis metrics for single and multiple FPGA implementations of our sample designs. Xilinx 12.1 was used to produce bit-files. To limit compile times, we stepped fMax at increments of 5MHz.

Chapter 11

Conclusion

FPGAs are growing in importance as substrates for algorithmic computing. As programmers implement larger and larger algorithms, these algorithms may cease to fit in a single FPGA. In the past, when programs have spilled across FPGA boundaries, programmers have faced either the painful task of manually partitioning their design across multiple FPGAs or the painful loss of performance through the use of existing multiple FPGA implementation tools. By exploiting high-level properties already present in many hardware designs, this thesis demonstrated that high-performance multiple FPGA implementations can be produced automatically.

Past approaches to multiple FPGA partitioning have been limited by the descriptive power of traditional RTLs, which obfuscate the high-level behavior and properties of original hardware. This obfuscation forces compilers to maintain cycle-accuracy as a fundamental level of abstraction, which causes a loss of performance in the case that clock synchronization must be distributed across chips. To raise the level of abstraction available to the programmer, this thesis presented the latency-insensitive channel, a new primitive for communication in hardware designs. The latency-insensitive channel permits designs to describe to the compiler points in the design where it is safe to change the cycle behavior of the design.

Because they admit cycle behavior changes, latency-insensitive channels may have many different physical implementations, ranging from fixed-size hardware FIFOs to complex networks spanning multiple FPGAs. Thus, it is possible to automatically

compile designs so described across an arbitrary network of FPGAs. This thesis presents the LIM compiler, a tool capable of automatically synthesizing efficient inter-FPGA networks from user source. Unlike traditional automatic partitioning tools, implementations produced by the LIM compiler generally enjoy high levels of performance, in many cases surpassing single FPGA implementations of the same program. Latency-insensitive programs implemented using the LIM compiler compute as data flows, rather than when some globally synchronized clock ticks. Thus, rather than paying a fixed overhead for inter-FPGA communication, latency-insensitive programs naturally tolerate the latency of communication.

The programs presented in the latter part of the thesis benefitted substantially from multiple FPGA implementation. All programs compiled faster, and experienced reduced recompilation times. Many program throughputs also improved, and, in the case of some Airblue derivatives, super linear performance gains were observed. Additionally, HAsim and Airblue were able to scale to capture much larger designs.

In providing a language syntax, model of computation, and optimizing compiler for multiple FPGAs, the thesis made the following specific contributions:

- The formulation and application of a model of computation for designs with explicit latency-insensitive channels (Chapter 2.1).
- A compiler mapping designs with explicit latency-insensitive channels to multiple FPGAs (Chapter 4).
- An area-efficient, high-performance network architecture for latency insensitive channels (Chapter 5).
- Optimizations for inter-FPGA networks (Chapter 6).
- Techniques for providing automatic and scalable access to resources, such as memory, across multiple FPGAs (Chapter 7).
- Numerous large hardware designs obtained through application of the compiler (Chapters 8, 9, and 10).

11.1 Future Work

The compiler presented in this thesis is already capable of producing high-quality hardware implementations for multiple FPGA environments, which was amply demonstrated in Chapters 8, 9, and 10. However, the study of the latency-insensitive model of computation and multiple FPGA compilation presented in this thesis suggest that there are many exciting researches left to be conducted. There are three possible avenues of exploration suggested by the contributions of this thesis: building very large system prototypes, improving the multiple FPGA compiler, and applying the latency-insensitive model of computation to other applications in the space of parallel system design.

Large Prototypes

The LIM compiler presented in this thesis permits the efficient implementation of very large hardware programs. One example of a scalable design space capable of producing such large programs is multiple-input, multiple-output (MIMO) RF transceivers. In MIMO, the transmitter sends multiple distinct data streams simultaneously on physically separate antennas (MO). The corresponding receiver then receives a mixing of these streams on physically separate antennas (MI). Due to the spatial diversity among the antennas, the receiver is able to solve a series of linear equations to decode each original data stream. Thus, adding more antennas fundamentally improves the throughput of a wireless network. MIMO is a developing trend in high-throughput wireless communications standards: 802.11n [30] introduced MIMO, and the follow-on 802.11ac doubles the maximum number of MIMO streams.

The LIM compiler is an enabling implementation technology for experimenting with large MIMO systems. MIMO baseband requires much greater computational capacity than the wireless systems presented in Chapter 8, and multiple FPGAs do enable the construction of these complex basebands. However, the real difficulty in build large MIMO systems is not in the computational complexity of the baseband, it is in physically interfacing to multiple antennas. Prototyping MIMO with eight or

sixteen antennas requires building an FPGA board with that many antennas, which is a difficult task for the academic research community. However, because the LIM compiler offers a scalable way to aggregate boards with multiple physical resources, the LIM compiler could be used to build a large MIMO system by simply aggregating existing single antenna solutions.

Improvements to Multiple FPGA Compilation

Some designs presented in this thesis, such as HAsim and OFDM, can scale to very large numbers of FPGAs. There are several issues related to scaling to large numbers of FPGAs that have been left unaddressed in the LIM compiler. Foremost among these is automatic placement of latency-insensitive modules on to FPGA platforms. The examples considered in Chapters 8, 9, and 10 were all comprised of a small number of modules, making hand-placement feasible. However, as designs scale high-quality hand-placement will quickly become difficult. A basic algorithm for automatic placement requires two pieces of information: the area of each latency-insensitive module and the traffic on the inter-module channels. Once this information is known, for example by synthesizing user latency-insensitive modules to obtain their area and using the methods of Chapter 6 to determine inter-module communications, then successive applications of the Kernighan and Lin heuristic [5] could be used to optimize the placement.

Applications of Latency-Insensitive Channels

The LIM compiler maps programs described using latency-insensitive channels on to environments comprised of multiple FPGAs, with the primary intention of making the implementation of large hardware programs feasible. However, latency-insensitive channels enable many other interesting kinds of compilation. Within the domain of FPGA design, latency-insensitive channels may provide a means of reducing the ever-increasing run-times of the FPGA synthesis tool chain. Furthermore, latency-insensitive design may be extended to solve several important problems in more general field of digital design.

Currently, compilation is a significant bottleneck in FPGA-based design. Even small designs can require hours of compilation time to produce an FPGA implementation. Worse, changing any portion of the design can result in a complete recompilation, even if the change is isolated to a small portion of the design. These issues represent a serious impediment to the adoption of FPGAs as algorithmic compute platforms. This compilation problem represents one serious barrier to the adoption of FPGAs as algorithmic compute platforms and severely impinges upon programmer productivity.

The chief reason that small RTL changes precipitate global is the difficulty of meeting global timing closure. Small changes may only perturb a small portion of the design, but these perturbations can ripple across the design necessitating extra work. The latency-insensitive model of computation proposed in this thesis can be leveraged to solve this problem by providing an insulating layer between RTL modules. A single FPGA can be partitioned in to smaller "virtual" FPGAs, with some reserved space between the "virtual" FPGAs for inter-"virtual" FPGA routing. The "virtual" FPGAs themselves can be targeted using a compilation flow similar to the flow presented in Chapter 4. "Virtual" FPGAs can be independently placed and routed, with their inter-FPGA interconnects tied to specific locations on die. Once these RTL islands have been placed-and-routed, the inter-"virtual" FPGA links can be routed through the reserved interconnect region, in a process similar to buffer-box routing in classical ASIC flows. Since the inter-"virtual" FPGA interconnect is latency-insensitive, additional delay stages can be added as needed between virtual FPGAs to achieve global timing closure. During subsequent recompilations only those "virtual"-FPGAs that have changed and the global interconnect must be recompiled.

This approach improves tool run-times in two significant ways. First, the primary compilation step is fundamentally parallel, and maybe quite fast in the context of well-partitioned designs. Second, on recompilation, substantial portions of previous compilations may be re-used. It may also be the case that platform interface libraries of the kind discussed in Chapter 7 may be reused, without recompilation, across all designs, in much the same way that fundamental C libraries are almost never

recompiled by modern software programmers.

The scope of latency-insensitive design is not limited to FPGAs alone. Indeed, it has broad applicability to digital design, especially in the increasingly important space of System-On-Chip (SoC) design. Current SoC design consist of heterogeneous mixes of accelerator cores and various kinds of general purpose processors. Currently, the interconnection network between these devices is manually designed, using a traditional bus hierarchy. This approach is somewhat fragile, in the sense that new system-level requirements and additional accelerator blocks can necessitate a complete system redesign. Such efforts are also error prone: giving the wrong priority to a channel, even at one router, can result in a loss of functionality [71].

As the complexity of SoC systems grows beyond human ability to reason about their behavior, increased design automation will be required to assist chip developers in producing functional designs. Latency-insensitive channels represent on possible solution: the kind of communications that occur between blocks in an SoC are qualitatively similar to kinds of communication that occur in multiple FPGA designs. In a typical SoC design, hardware blocks are aggregated into processing pipelines, for example to handle a phone call or to play back compressed audio formats. Since these hardware blocks conform to the latency-insensitive model of computation, it should be possible to synthesize automatically the interconnect between the hardware and to provide at least some performance guarantees about the aggregate behavior of the system.

SoCs designs also involve complex interactions between software and hardware.In most systems, the hardware acts as an offload engine for software: when software encounters an accelerated operation, it pushes some data to the hardware and awaits a response. In the context of FPGA design, the roles of software and hardware can sometimes be reversed: software acts as an assistant to the FPGA, performing complex operations that would be difficult to implement in hardware. However, in both cases, the interaction between hardware and software can be naturally framed in terms of latency-insensitive channels. Couching hardware-software communication in terms of latency-insensitive channels simplifies the co-design problem as many of

the tedious and error-prone portions of the hardware-software communication process can be automatically synthesized.

Bibliography

[1] Michael Adler, Kermin Fleming, Angshuman Parashar, Michael Pellauer, and Joel S. Emer. LEAP Scratchpads: Automatic Memory and Cache Management For Reconfigurable Logic. In *FPGA*, pages 25–28, 2011.

[2] Sameh W. Asaad, Ralph Bellofatto, Bernard Brezzo, Chuck Haymes, Mohit Kapur, Benjamin D. Parker, Thomas Roewer, Proshanta Saha, Todd Takken, and José A. Tierno. A cycle-accurate, cycle-reproducible multi-fpga system for accelerating multi-core processor simulation. In *FPGA*, pages 153–162, 2012.

[3] Jonathan Babb, Russell Tessier, Matthew Dahl, Silvina Hanono, David M. Hoki, and Anant Agarwal. Logic Emulation With Virtual Wires. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 16(6):609–626, 1997.

[4] Bluespec Inc. http://www.bluespec.com.

[5] B.W. Kernighan and Shen Lin. An Efficient Hueristic Procedure for Partitioning Graphs. In *Bell Systems Technical Journal*, pages 291–307, 1970.

[6] Luca P. Carloni, Kenneth L. McMillan, Alexander Saldanha, and Alberto L. Sangiovanni-Vincentelli. A methodology for correct-by-construction latency insensitive design. In *ICCAD*, pages 309–315, 1999.

[7] Luca P. Carloni, Kenneth L. McMillan, and Alberto L. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 20(9):1059–1076, 2001.

[8] Josep Carmona, Jordi Cortadella, Michael Kishinevsky, and Alexander Taubin. Elastic circuits. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 28(10):1437–1455, 2009.

[9] Eric S. Chung, Michael Papamichael, Eriko Nurvitadhi, James C. Hoe, Ken Mai, and Babak Falsafi. Protoflex: Towards scalable, full-system multiprocessor simulations using fpgas. *TRETS*, 2(2), 2009.

[10] Jordi Cortadella, Michael Kishinevsky, and Bill Grundmann. Synthesis of synchronous elastic architectures. In *DAC*, pages 657–662, 2006.

[11] Jordi Cortadella, Marc Galceran Oms, and Michael Kishinevsky. Elastic systems. In *MEMOCODE*, pages 149–158, 2010.

[12] W. J. Dally and C. L. Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Trans. Comput.*, 36:547–553, May 1987.

[13] Nirav Dave, Man Cheuk Ng, Michael Pellauer, and Arvind. A design flow based on modular refinement. In *Formal Methods and Models for Codesign (MEMOCODE), 2010 8th IEEE/ACM International Conference on*, pages 11 –20, Jul. 2010.

[14] J. Emer, P. Ahuja, E. Borch, A. Klauser, C.K. Luk, S. Manne, S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa, and T. Juan. Asim: A performance model framework. *IEEE Computer*, 2002.

[15] Ettus Research USRP2. `http://www.ettus.com/products`.

[16] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[17] K. Fleming, Chun-Chieh Lin, N. Dave, Arvind, G. Raghavan, and J. Hicks. H.264 Decoder: A Case Study in Multiple Design Points. In *Formal Methods and Models for Co-Design, 2008. MEMOCODE 2008. 6th ACM/IEEE International Conference on*, pages 165 –174, Jun. 2008.

[18] Kermin Fleming, Myron King, Man Cheuk Ng, Asif Khan, and Muralidaran Vijayaraghavan. High-throughput Pipelined Mergesort. In *MEMOCODE*, pages 155–158, 2008.

[19] Davy Genbrugge, Stijn Eyerman, and Lieven Eeckhout. Interval simulation: Raising the level of abstraction in architectural simulation. In *HPCA*, pages 1–12, 2010.

[20] The GNURadio Software Radio. `http://gnuradio.org/trac`.

[21] Shyamnath Gollakota and Dina Katabi. ZigZag decoding: Combating hidden terminals in wireless networks. In *SIGCOMM'08*, Seattle, WA, 2008.

[22] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM JOURNAL ON APPLIED MATHEMATICS*, 17(2):416–429, 1969.

[23] ITU-T Video Coding Experts Group. Draft ITU-T Recommendation and Final Draft International Standard of Joint Video Specification, May, 2003.

[24] Daniel Halperin, Thomas Anderson, and David Wetherall. Taking the Sting out of Carrier Sense: Interference Cancellation for Wireless LANs. In *MobiCom'08*, San Francisco, CA, 2008.

[25] Tyler S. Harris, Zhuo Ruan, and David A. Penry. Techniques for li-bdn synthesis for hybrid microarchitectural simulation. In *ICCD*, pages 253–260, 2011.

[26] `http://www.cadence.com/products/sd/palladium_series/pages/default.aspx`. "Cadence Palladium".

[27] http://www.eda.org/itc/scemi.pdf. Standard Co-Emulation Modelling Interface (SCE-MI): Reference Manual.

[28] http://www.nallatech.com. Nallatech ACP module.

[29] `"http://www.synopsys.com/Systems/FPGABasedPrototyping/pages/certify.aspx"`. "Synopsys Certify".

[30] IEEE Standard 802.11: Wireless LAN Medium Access Control and Physical Layer Specifications, 1999.

[31] Kyle Jamieson. *The SoftPHY Abstraction: from Packets to Symbols in Wireless Network Design*. PhD thesis, MIT, Cambridge, MA, 2008.

[32] Kyle Jamieson and Hari Balakrishnan. PPR: Partial Packet Recovery for Wireless Networks. In *SIGCOMM'07*.

[33] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing*, pages 471–475, 1974.

[34] Sachin Katti, Dina Katabi, Hari Balakrishnan, and Muriel Medard. Symbol-Level Network Coding for Wireless Mesh Networks. In *SIGCOMM'08*, Seattle, WA, 2008.

[35] Asif Khan, Muralidaran Vijayaraghavan, Silas Boyd-Wickizer, and Arvind. Fast and cycle-accurate modeling of a multicore processor. In *ISPASS*, pages 178–187, 2012.

[36] Michel A. Kinsy, Myong Hyon Cho, Tina Wen, G. Edward Suh, Marten van Dijk, and Srinivas Devadas. Application-aware Deadlock-free Oblivious Routing. In *ISCA*, pages 208–219, 2009.

[37] S. Y. Liao. Towards a new standard for system level design. In *Proceedings of the Eighth International Workshop on Hardware/Software Codesign*, pages 2–7, San Diego, CA, May 2000.

[38] Jason E. Miller, Harshad Kasture, George Kurian, Charles Gruenwald III, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. Graphite: A Distributed Parallel Simulator for Multicores. In *HPCA*, pages 1–12, 2010.

[39] M. C. Ng, K. Fleming, M. Vutukuru, S. Gross, Arvind, and H. Balakrishnan. Airblue: A System for Cross-Layer Wireless Protocol Development. In *ANCS'10*, San Diego, CA, 2010.

[40] Man Cheuk Ng, Muralidaran Vijayaraghavan, Gopal Raghavan, Nirav Dave, Jamey Hicks, and Arvind. From WiFI to WiMAX: Techniques for IP Reuse Across Different OFDM Protocols. In *MEMOCODE'07*.

[41] A. Parashar, M. Adler, M. Pellauer, and J. Emer. Hybrid CPU/FPGA Performance Models. In *WARP '08: The 3rd Workshop on Architectural Research Prototyping*, 2008.

[42] Angshuman Parashar, Michael Adler, Kermin Fleming, Michael Pellauer, and Joel Emer. LEAP: A Virtual Platform Architecture for FPGAs. In *CARL '10: The 1st Workshop on the Intersections of Computer Architecture and Reconfigurable Logic*, 2010.

[43] M. Pellauer, M. Adler, D. Chiou, and J. Emer. Soft Connections: Addressing the Hardware-Design Modularity Problem. In *DAC '09: Proceedings of the 46th Annual Design Automation Conference*, pages 276–281. ACM, 2009.

[44] M. Pellauer, M. Adler, M. Kinsy, A. Parashar, and J. Emer. HAsim: FPGA-Based High-Detail Multicore Simulation Using Time-Division Multiplexing. In *The 17th International Symposium on High-Performance Computer Architecture (HPCA)*, February 2011.

[45] M. Pellauer, M. Vijayaraghavan, M. Adler, Arvind, and J. Emer. A-Ports: An Efficient Abstraction for Cycle-Accurate Performance Models on FPGAs. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA)*, February 2008.

[46] M. Pellauer, M. Vijayaraghavan, M. Adler, Arvind, and J. Emer. Quick Performance Models Quickly: Closely-Coupled Timing-Directed Simulation on FPGAs. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2008.

[47] Michael Pellauer, Michael Adler, Michel A. Kinsy, Angshuman Parashar, and Joel S. Emer. HAsim: FPGA-based high-detail multicore simulation using time-division multiplexing. In *HPCA*, 2011.

[48] Michael I. Pellauer. *HAsim: Cycle-accurate Multicore Performance Models on FPGAs*. PhD thesis, MIT, Cambridge, MA, 2010.

[49] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. Using simpoint for accurate and efficient simulation. In *SIGMETRICS*, pages 318–319, 2003.

[50] J. Perry, H. Balakrishnan, and D. Shah. Rateless spinal codes. In *HotNets-X*, October 2011.

[51] Hariharan Rahul, Nate Kushman, Dina Katabi, Charles Sodini, and Farinaz Edalat. Learning to share: narrowband-friendly wideband networks. In *SIGCOMM'08*, Seattle, WA, USA, 2008.

[52] Iain E.G. Richardson. *H.264 and MPEG-4 Video Compression*. John Willey & Sons, 2003.

[53] Graham Schelle, Jamison D. Collins, Ethan Schuchman, Perry H. Wang, Xiang Zou, Gautham N. Chinya, Ralf Plate, Thorsten Mattner, Franz Olbrich, Per Hammarlund, Ronak Singhal, Jim Brayton, Sebastian Steibl, and Hong Wang. Intel Nehalem Processor Core Made FPGA Synthesizable. In *FPGA*, pages 3–12, 2010.

[54] Charles Selvidge, Anant Agarwal, Matthew Dahl, and Jonathan Babb. TIERS: Topology Independent Pipelined Routing and Scheduling for Virtual Wire Compilation. In *FPGA*, pages 25–31, 1995.

[55] Todd Snyder. Multiple FPGA Partitioning Tools and Their Performance. Private communication, 2011.

[56] T. Stricker and T. Cross. Global Address Space, Non-Uniform Bandwidth: A Memory System Perform ance Characterization of Parallel Systems. In *Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture*, HPCA '97, 1997.

[57] Syed Suhaib and Deepak Mathaikutty and Sandeep Shukla. Dataflow Architectures for GALS. *Electron. Notes Theor. Comput. Sci.*, 200(1), February 2008.

[58] Kun Tan, Jiansong Zhang, Ji Fang, He Liu, Yusheng Ye, Shen Wang, Yongguang Zhang, Haitao Wu, Wei Wang, and Geoffrey M. Voelker. Sora: High Performance Software Radio Using General Purpose Multi-core Processors. In *NSDI'09*, Boston, MA, 2009.

[59] Russel Tessier. Multi-FPGA Systems: Logic Emulation. *Reconfigurable Computing*, pages 637–669, 2008.

[60] Shunji Umetani, Mutsunori Yagiura, and Toshihide Ibaraki. One-dimensional cutting stock problem to minimize the number of different patterns. *European Journal of Operational Research*, 146(2):388–402, April 2003.

[61] Muralidran Vijayaraghavan and Arvind. Bounded Dataflow Networks and Latency-Insensitive Circuits. In *MEMOCODE'09*, Cambridge, MA, 2009.

[62] VP8 Open Video Codec. `http://www.webmproject.org/`.

[63] Mythili Vutukuru. *Physical Layer-Aware Wireless Link Layer Protocols*. PhD thesis, MIT, Cambridge, MA, 2010.

[64] Mythili Vutukuru, Hari Balakrishnan, and Kyle Jamieson. Cross-Layer Wireless Bit Rate Adaptation. In *SIGCOMM'09*.

[65] Mythili Vutukuru, Kyle Jamieson, and Hari Balakrishnan. Harnessing Exposed Terminals in Wireless Networks. In *NSDI'08*.

[66] Rice university wireless open-access research platform (WARP). `http://warp.rice.edu`.

[67] Nam Sung Woo and Jaeseok Kim. An Efficient Method of Partitioning Circuits for Multiple-FPGA Implementation. In *Proceedings of the 30th international Design Automation Conference*, DAC '93, pages 202–207, New York, NY, USA, 1993. ACM.

[68] Roland E. Wunderlich, Thomas F. Wenisch, Babak Falsafi, and James C. Hoe. Statistical Sampling of Microarchitecture Simulation. *ACM Trans. Model. Comput. Simul.*, 16(3):197–224, 2006.

[69] Xilinx, Inc. Multi-Port Memory Controller. 2011.

[70] Xilinx University Program XUPV5-LX110T Development System. `http://www.xilinx.com/univ/xupv5-lx110t.htm`.

[71] Rumi Zahir. Design Issues in Telecommunications SoCs. Private communication, 2012.