5-19-2012

A Modular Approach to Adaptive Reactive Streaming Systems

Christopher E. Neely
Santa Clara University

SANTA CLARA UNIVERSITY
School of Engineering
Department of Computer Engineering

Dissertation Examination Committee:
Prof. Gordon Brebner (industrial co-advisor)
Prof. Silvia Figueira
Prof. JoAnne Holliday
Prof. Tokunbo Ogunfunmi
Prof. Weijia Shang (academic co-advisor)

A Modular Approach to Adaptive Reactive Streaming Systems

by

Christopher E. Neely

A dissertation presented to the School of Engineering
of Santa Clara University in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

May 2012
Santa Clara, California

ABSTRACT

A Modular Approach to Adaptive Reactive Streaming Systems

by

Christopher E. Neely

Doctor of Philosophy in Computer Engineering

Santa Clara University, 2012

Industrial Co-advisor: Professor Gordon Brebner

Academic Co-advisor: Professor Weijia Shang

The latest generations of FPGA devices offer large resource counts that provide the headroom to implement large-scale and complex systems. However, there are increasing challenges for the designer, not just because of pure size and complexity, but also in harnessing effectively the flexibility and programmability of the FPGA. A central issue is the need to integrate modules from diverse sources to promote modular design and reuse. Further, the capability to perform dynamic partial reconfiguration (DPR) of FPGA devices means that implemented systems can be made reconfigurable, allowing components to be changed during operation. However, use of DPR typically requires low-level planning of the system implementation, adding to the design challenge. This dissertation presents ReShape: a high-level approach for designing systems by interconnecting modules, which gives a 'plug and play' look and feel to the designer, is supported by tools that carry out implementation and verification functions, and is carried through to support

system reconfiguration during operation. The emphasis is on the inter-module connections and abstracting the communication patterns that are typical between modules – for example, the streaming of data that is common in many FPGA-based systems, or the reading and writing of data to and from memory modules. ShapeUp is also presented as the static precursor to ReShape. In both, the details of wiring and signaling are hidden from view, via metadata associated with individual modules. ReShape allows system reconfiguration at the module level, by supporting type checking of replacement modules and by managing the overall system implementation, via metadata associated with its FPGA floorplan. The methodology and tools have been implemented in a prototype for a broad domain-specific setting – networking systems – and have been validated on real telecommunications design projects.

Acknowledgements

I am grateful to Xilinx for sponsoring my research and my participation in this Ph.D. program at Santa Clara. At Xilinx, I have had the enjoyable experience to work together with colleagues who I also consider to be great friends. They have also given me much encouragement along the way to finish my Ph.D. and to take pride in my work.

Thank you to several colleagues for interesting discussions and for various contributions to the tools: Robert Esser, Padmini Gopalakrishnan, Phil James-Roxby, Eric Keller, Chidamber Kulkarni, Nathan Lindop, Andy Norton, Soren Pedersen, Martin Sinclair, and Henry Styles.

Thank you very much to Jack Lo for his close collaboration on various projects related to ShapeUp, in particular to refining the interface types and on auto-bridging for linking modules, and help supporting the OAM project. Jack, thanks for putting up with me, you are always a pleasure to work with. Mike Attig, thank you for always being there: whether for technical advice or for your friendship. Special thanks to my dad for all his encouragement, love, and support, and for inspiring me to pursue computers and technology.

Lastly and most importantly, I am grateful for the mentorship, opportunities, and support given to me by my advisors. Thank you to Prof. Gordon Brebner and Prof. Weijia Shang for their much appreciated help and preparation.

Chris Neely

Santa Clara University
May 2012

Dedicated to:
Carol,
Mom, Dad,
Cathe, Andrea,
and Mammaw.

Contents

List of Tables

List of Figures

Preface

This dissertation presents research performed towards completing the requirements of the part-time, industrial track Ph.D. program at Santa Clara University, which was conducted at Xilinx Research Labs. The main contributions of this dissertation are derived from the following publications:

- C. Neely, G. Brebner, W. Shang. "ShapeUp: A High-Level Design Approach to Simplify Module Interconnection on FPGAs", In Proceedings of the 18th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, May 2010.

- C. Neely, G. Brebner, W. Shang. "Flexible and Modular Support for Timing Functions in High Performance Networking Acceleration", In Proceedings of the 20th International Conference on Field-Programmable Logic and Applications, August/September 2010.

- C. Neely, G. Brebner, W. Shang. "Reshape: Towards a High-Level Design Approach to Simplify Module Interconnection on FPGAs", ACM Transactions on Reconfigurable Technology and Systems (TRETS), Accepted and pending publication.

Chapter 1
Introduction

Personal networked devices are ubiquitous. People carrying small electronic devices post updates to social networks, download the latest popular videos, or connect to each other through video chat. Smart phone users communicate instructions for processing on remote cloud servers. The public is placing growing dependence on the networking infrastructure that transparently enables these personalized services. In the modern Internet, application processing is moving away from host-end systems and into network clouds. This trend is due, in part, to limitations of mobile and portable devices, which are constrained by limited processing capabilities and stringent low-power requirements. Major companies are continuing to invest in large data centers that farm content and services to a vast Internet audience.

The popularity of the Internet is growing worldwide. There is rising demand for video and other services, which is causing increased traffic. Network service providers are constantly upgrading their backbone and core networks, and changing from 40 Gb/sec to much higher bandwidths like 400 Gb/sec. Processing of packets that ship this data requires keeping up with these increasing line rates, so there is need for programmability at hardware speeds. Networks require adequate controls for maintenance, monitoring, and providing quality of service, e.g. by traffic shaping. These are important to network service providers and network carriers, as they continue to build up and enhance their networks.

The challenges that custom networking hardware and system designers generally face are that: (a) costs of producing application-specific integrated circuits are greatly rising, (b) their projects have short development cycles, (c) network standards are rapidly changing, and (d) their designs are increasing in complexity [1]. There is a need for new design methodologies that take advantage of programmability, while providing the required high performance and improving productivity.

These industry trends are the impetus for this research. The vast and growing array of Internet services create an insatiable demand for communications bandwidth. Concurrent with this demand for bandwidth there is a need for improving programmability towards designing systems that run at hardware speeds.

1.1 Research Topic

The key driving forces behind this research are high-speed networking and the need for improved programmability. The focus of this research is thus on the area of programmable streaming systems that are reactive and adaptive.

The choice of streaming systems means an emphasis on processing systems whose main characteristic is that of data flow through the system. Digital signal processing systems and packet processing systems are examples of stream processing systems. This is in contrast to traditional von Neumann style data processing systems. The topic of stream processing is attracting much attention in the parallel computing community at present, particularly as a way of harnessing multi-core processors, through such programming initiatives as Brook [2], CUDA [3], OpenCL [4], and StreamIt [5]. The research presented in this dissertation focuses on targeting and harnessing of *programmable hardware technologies*. To provide further focus to the work, case studies are drawn from *packet processing* as a specific domain of streaming systems.

The reactive characteristic of the systems under study means that the systems react and execute in response to stimulus events from their environment, for example the arrival of packets over a communication channel. These systems have an ongoing interaction with the environment, as opposed to systems that run to completion and produce a final result. The adaptive characteristic refers to the possibility of a system adapting to changes in the environment. Specifically, this refers to the possibility of reprogramming hardware while a system is in operation, to make architectural changes as opposed to just having adaptability within software. An example would be to modify packet processing capabilities in response to changing data traffic patterns in a network.

The goal of this research is to improve the ease of design of networking systems that require hardware-like performance. These network systems have great complexity as static designs and acquire additional complexity when they are required to adapt to changing environments. This dissertation proposes a programming methodology to mitigate design complexity by providing appropriate high-level abstractions that assist a modular design approach. These systems may potentially incorporate the use of time in their specification, and they may also have dynamic behavior to adapt. This dissertation also presents two case studies demonstrating example reactive systems using the above-mentioned approaches: one non-adaptive and one adaptive.

1.2 Outline

This section provides an outline for the remainder of this dissertation, highlighting the contributions.

Chapter 2 provides relevant background, in the form of a literature survey organized into four main categories: (a) FPGA architectures and tools, (b) relevant system design methodologies, (c) dynamic reconfiguration research related to supporting

adaptive systems, and (d) networking research involving reconfigurable hardware such as FPGAs. This research is bridging and synthesizing these diverse areas.

Chapter 3 presents the basic approach using a modular abstraction called *ShapeUp*. A set of interface abstractions and a modular design methodology is described based on abstractions of module interface behavior, from three programming paradigms. This research is novel in that there has been significant past work on abstracting behavior of module functions, but little on the abstraction of the interconnection of modules. ShapeUp addresses this by abstracting the behavior of the interfaces and connections between the interfaces. Several tools were developed that use general data driven mechanisms. A brief introduction to the Click language, from MIT, is provided in Section 3.1 since it is extended by ShapeUp and used to describe systems. The contributions of Chapter 3 include:

- A set of abstractions of module interface behavior, featuring five types of interface that cover both streaming and procedural programming paradigms for modules. These are presented in Section 3.2.
- The use of metadata (and meta-metadata, in fact) to describe a module's interfaces in terms of the defined abstractions, enabling the creation of module repositories. This is described in Section 3.3.
- A type checker that is used by the other tools to indicate the compatibility of two ports when forming a connection. This is described in Section 3.4.
- Tools that process (extended semantics) Click descriptions and module metadata in order to provide a high-level modular design experience. These are described in Section 3.7.

Chapter 4 presents a modular approach for timing functions, which are pervasive in networking. A wide-ranging review leads to the design of modules for timing. These mechanisms are similarly flexible and modular, fitting in with the proposed design methodology, so it is not necessary to re-implement ad hoc timing capabilities

each time some network packet processing function is being accelerated using FPGA technology. The contributions of Chapter 4 include:

- A review of the prevalent timing paradigms observed in network protocols that exposed three basic timing functions requirements. This is summarized in Section 4.1.
- The design and implementation of a set of three highly configurable timing modules that provide a flexible solution for the identified basic requirements. These are described in Section 4.2. Activity diagrams were created to show time requirements and the use of the three timing modules as they relate to individual activities. These are described in Section 4.2.3.
- The embedding of these modules within the ShapeUp methodology, to allow seamless integration with other modules. This is described in Section 4.3.

Chapter 5 presents a case study that incorporates these timing modules into the ShapeUp framework and tool flow. The insatiable demand for bandwidth, which was mentioned earlier as one of the driving forces, is requiring network providers to upgrade their core networks, including a move to *carrier Ethernet*. Ethernet OAM, which stands for Ethernet Operations, Administration, and Maintenance, is an increasingly important standard in modern carrier Ethernet. The main contributions of Chapter 5 are:

- A thorough analysis of the complex timing needs of OAM protocols is presented in Section 5.2. This is demonstrated by productive use of activity model and mapping to timing modules.
- A high-level approach is carried throughout the programming methodology and framework, combining ShapeUp and the programming language G within the methodology. This is described in Section 5.4.
- Non-trivial Click descriptions (Y.1731 and CFM) were entered and processed with ShapeUp tools. The results were flexible and maintainable designs, delivering required hardware performance. These are described in Section 5.6.

Chapter 6 introduces the adaptive systems part of this work, which extends the ShapeUp framework to support dynamic modules in an extended methodology called *ReShape*. This model allows: (a) modules to be *substituted* dynamically when the system is in operation, (b) brings benefits of abstraction and modularity to dynamic reconfiguration based on the latest partial reconfiguration (PR) tools, and (c) extends the ShapeUp framework from purely design-time use to lifetime use. A key topic in this work is floorplanning, which physically constrains design placement. This chapter investigates the automatic floorplanning of modules and describes experiments measuring the performance of partition-based design flows. This chapter also proposes an algorithm to constrain the placement of modules communicating in a linear pipeline. The contributions of Chapter 6 include:

- An investigation of the characteristics of the backend PR tools, revealing 50% less internal fragmentation is achievable, compared with prior expectations. The analysis of these results is presented in section 6.2.
- A domain-specific floorplanning algorithm that provided a reliable basis for abstraction of the underlying dynamic partial reconfiguration mechanisms. The algorithm is presented in Section 6.3.
- Extending the ShapeUp methodology and tools into the more general ReShape methodology, illustrated concretely through a specific set of prototype tools that support dynamic reconfiguration of networking systems defined using *Linear Click*, a Click subset. The methodology and the prototype tools are described in Section 6.4.

Chapter 7 presents a case study from an adaptive high-speed (150 Gb/sec) networking, packet parser example. Overall, the case study demonstrated the benefits of the ReShape approach, in terms of supporting the 'system for life' model and hiding the low-level details of FPGA partial reconfiguration from the user. The main contributions of Chapter 7 are:

- Validation of the productivity gains from use of the ReShape methodology and the prototype tools on a real-life industrial-strength case study. A high-speed real-life system is described in Linear Click. An example of the Programmable Packet Parser supporting dynamic behavior is described in Section 7.2.
- Experiments using four configurations of the Programmable Packet Parser were conducted, comparing hard-coded, microcoded, and ReShape approaches. A comparison of the results is presented in Section 7.3.
- The case study demonstrated the benefits of the ReShape approach, in terms of supporting the 'system for life' model and hiding the low-level details of partial reconfiguration from the user. This is discussed in Section 7.3.

Chapter 8 presents the conclusions of this dissertation and suggests directions for future work.

Overall, this dissertation shows that the ReShape methodology makes a significant contribution to encouraging a high-level modular approach to designing FPGA-based networking systems. This work synthesizes builds upon results from four different areas: FPGAs, system-level design, dynamic reconfiguration, and networking, and some relevant background material introduced in the next chapter.

Chapter 2
Background

The dissertation spans four broad research categories and synthesizes ideas from a number of different directions. This chapter reviews the foundation for this work.

Section 2.1 presents **FPGA architecture and tools** and provides an introduction to the underlying device technology that is being targeted for this research – its characteristics and current design methodologies in use. The basic architecture of FPGAs and the current design flow for programming the technology are discussed. The dissertation extends current FPGA tool flow and programming.

Section 2.2 presents **Modular system design** and provides an overview of tools and methodologies for performing system-level design, particularly targeting the hardware aspects of systems. The dissertation extends existing modular design flows for networking software to target programmable hardware.

Section 2.3 presents **Dynamic reconfiguration** and provides an overview of research into the use of programmable hardware to support systems that have adaptive behavior at run time. The dissertation focuses on harnessing the partial reconfiguration capability of FPGAs and suggests improvements in the programming framework for updating the FPGA's programming at run time.

Section 2.4 presents **Packet processing using FPGAs** and provides a review of past work involving the implementation of packet processing system functions using FPGA technology. The dissertation suggests improvements in the programming of high performance networking applications targeted to FPGAs.

2.1 Field Programmable Gate Array (FPGA) architecture and tools

The most recent general survey of this area, the 2002 paper by Compton and Hauck [6], gives excellent coverage of what the authors term "reconfigurable computing," which is seen as filling a gap between hardware and software. However, this survey is now ten years old in a rapidly evolving field. In the past decade there have been significant advances in FPGA technology and capabilities. For example, there have been great increases in the logic density and in I/O speed. Furthermore, significant architectural improvements have been made for better design scalability.

2.1.1 Basic FPGA Architecture

The Field Programmable Gate Array (FPGA) is a type of Programmable Logic Device (PLD) technology that can be efficiently programmed to implement custom logic and systems on chip, and also has the ability to be reprogrammed repeatedly after it is deployed in the field. The first FPGAs (in 1984) contained only 64 to 100 programmable logic elements. Modern FPGAs have over a million basic programmable logic elements. These FPGAs can be used in implementing complex embedded processor systems on chip, advanced signal processing applications for video or wireless, or high-speed (e.g. 100 Gb/sec) communications applications.

Before the invention of FPGAs, programmable logic arrays (PLAs) were the mainstream PLD technology for providing the ability to implement custom logic functions. PLAs were programmed by expressing logic functions as Boolean algebra expressions, in either sum of products form or product of sums form, depending on the technology. Early PLAs were manually programmed by setting interconnection points between individual logic gates to build larger functions. The programming process involved applying a current to destroy tiny fuses within the interconnection.

The fuses provided a choice of inputs to gates, and so fuses connecting undesired inputs were destroyed until only the desired connections remained. The main disadvantage of these PLAs was that they could only be programmed once, so any changes required starting over with a new device. The Complex Programmable Logic Device (CPLD) is a similar form of technology that improved upon early PLAs. CPLDs also feature product-based programming, but use a non-volatile memory for configuration that can be reprogrammed. The key advantage of FPGAs over PLAs and CPLDs is that they contain a more general programmable structure to implement logic functions that is capable of supporting reprogramming. The fundamental characteristic of FPGAs is that logic programming is implemented using *n*-input lookup tables (LUTs) to implement programmed logic functions. The LUTs can be connected in series to implement larger logic functions. Another essential feature that FPGAs incorporate is a programmable switch box for creating interconnections between LUTs. Memory-based configuration supports reprogramming by being implemented in SRAM.

The LUTs are small memories for implementing bit-level logic functions. An *n*-input LUT contains 2^n single-bit values, indexed by the input. LUTs can either implement any *n*-input logic functions by configuring the set of truth table values, or be used as a small distributed single-bit memory unit with an *n*-bit address space. The chosen value of *n* varies by manufacturer and architecture, and has evolved over time, but is typically in the range of three to six [7] [8] [9] [10]. The earliest FPGAs had three- or four-input LUTs. Modern, high-performance FPGAs, like the Xilinx Virtex-7 and the Altera Stratix V, have six-input LUTs. There has been considerable research into the most beneficial LUT size, looking at tradeoffs between area, performance, power, and mapping efficiency for logic circuitry. A seminal paper by Rose et al. in 1989 [11] made the case for the four-input LUT, which became dominant for over 15 years. In fact, it also made a case for the three-input LUT, but this was not adopted in practice. The question was revisited by Ahmed and Rose in 2000 [12] given advances in FPGA technology, this time with the conclusion that up to six-input LUTs could be beneficial. This research had a

direct impact on practice, first with the Altera Stratix II device, which introduced a six-input LUT architecture [13].

In most FPGA architectures, LUTs are clustered together in larger units. The coupling of flip-flops (FFs) with LUTs, to combine storage with combinatorial logic, was another key recommendation by Rose et al. [11]. In Xilinx architectures, several LUTs and FFs are grouped together into 'slices'. For example, each Virtex-7 slice contains four LUTs and eight FFs. In turn, slices are grouped into larger 'configurable logic blocks' (CLBs). Each Virtex-7 CLB contains two slices. The CLBs then form the basic two-dimensional array architecture – the 'A' in the 'FPGA'.

Aside from the LUTs and FFs for computing and storing logic function results, the other essential aspect of the FPGA is programmable interconnection, to allow logic circuitry to be built. The basic component is the Programmable Interconnection Point (PIP). This is a small programmable switchbox to select between interconnection paths. The actual paths provided are a key feature of any FPGA architecture. In early FPGAs, the paths were just between neighboring LUT clusters, giving limited scope for programmable switching. Nowadays, a range of different paths, spanning different distances over the array and covering different directions, are provided. In fact, typical FPGA silicon area can be 90% for the programmable interconnection and only 10% for functions [14], indicating the relative importance of this feature. Much research has been carried out into the most beneficial styles of interconnection. For example, Lemieux and Lewis [14] discuss this issue in detail in their 2004 book.

A final essential component of the FPGA is programmable input/output blocks for communicating with off-chip devices through the pins of the device. They are programmable to support different I/O signaling standards, such as drive strengths and voltages. These blocks have grown increasingly complex over FPGA generations, retaining support for legacy standards while gaining support for newer

standards. A particular trend now is towards support for high-speed serial input/output channels, operating at rates of up to 28 Gb/sec, with higher rates in prospect.

2.1.2 Advanced architectural elements

The modern FPGA device is no longer a simple two-dimensional array of logic blocks with programmable interconnect and input/output blocks. Additional features have been selectively hardened to improve performance for commonly used functions. An early feature was explicit support for addition-carry chains between logic blocks; this arithmetic support was then broadened to include complete multiplication blocks; and now to fixed-point multiply-accumulate blocks for DSP acceleration.

A key ingredient of FPGA architectures is a collection of embedded SRAM memory blocks, to give a more silicon-optimal storage option than building store out of LUTs and FFs. For example, the largest Xilinx Virtex-7 device has 1,292 dual-port SRAM blocks, each storing 36 Kbits, giving a total of 46 Mb of on-chip storage. Research has been carried out into the best ways of organizing embedded memory and integrating it with the basic logic fabric, notably by Wilton, Rose and Vranesic [15].

Moving beyond hardened arithmetic support, some FPGA architectures feature embedded processors. The Altera Excalibur and Xilinx Virtex-II Pro devices were the first FPGAs to contain a hard embedded processor core (ARM and PowerPC respectively) integrated with the logic fabric. In tandem with these hardened developments though, the size of FPGA logic arrays has advanced so significantly that soft processors [16] [17] can be configured as an alternative solution to processor needs (in fact, up to hundreds with current technology). Thus, hardened processors do not feature in recent Altera generations, and were not offered in the latest Xilinx (Virtex-7) generation. Xilinx has recently released the processor-centric Zynq platform targeted to software developers and to non-hardware experts,

containing dual core ARM Cortex A9 processors coupled to a smaller programmable logic fabric, for implementing custom accelerators [18]. Altera has recently announced a similar ARM-based platform [19].

Other advanced programmable features sometimes available include digital clock managers for implementing programmable clock signals, voltage or temperature sensors, and communication blocks for widely used protocols (e.g. for Ethernet or PCI Express).

2.1.3 Programming the FPGA

The basic tools and methodologies for designing systems based on FPGAs are closely related to those used for ASIC design. Thus, the FPGA programming experience today is very much a hardware-design style of experience, which presents a high entry barrier to those from a software background. On the other hand, FPGA tools differ from ASIC tools by generating information for hardware configuration of an FPGA instead of mask information for a silicon chip.

Verilog and VHDL are the most popular hardware description languages (HDLs) used for describing FPGA designs at the register transfer level (RTL), which is the highest level of abstraction typically used. Design at the lower logic gate level is relatively unusual nowadays except in specialized or critical circumstances, a fact that reflects the maturity and acceptance of RTL design. The basic steps in the standard tool design flow are (a) compilation and synthesis, (b) technology mapping, and (c) placement and routing. Modern computer aided design (CAD) tools and backend synthesis tools are used to compile and synthesize HDL descriptions. These HDL descriptions map the synthesized designs into logic gate level representations that can then be mapped onto FPGA resources: lookup tables, flip-flops, and interconnection between them.

The most time-consuming aspect of using standard FPGA tools is the assignment of the FPGA resource requirements to specific sites on the FPGA. This assignment includes placement of LUTs and FFs and routing of interconnections. For large FPGA designs, this can take many hours, which is a major deterrent to users accustomed to the fast turnaround and hence frequent iteration possible with software compilers. Improvement of placement and routing algorithms, for both run time and storage requirements, is an area of major ongoing research. Chen, Cong, and Pan conducted an extensive survey of this area in 2006 [20]. A common approach to placement involves the use of simulated annealing as a heuristic technique to solve the NP-complete optimization problems involved.

Floorplanning provides the means for mapping system modules to distinct physical FPGA regions. This topic has been extensively researched. Algorithms for traditional ASIC floorplanning based on geometry and wire length were described by Adya and Markov [21] and Adya et al. [22]. Later, specialized algorithms targeting FPGAs with heterogeneous resources, were devised by Cheng and Wong [23], Feng and Mehta [24], and Banerjee [25]. The most recently published floorplanning algorithms further consider device capabilities, including granularity of reconfiguration and also resource distribution, for example the work of Montone et al. [26], Bolchini et al. [27], and Banerjee et al. [28]. This prior work has lent great insight to the domain-specific solution adopted for the work in Chapter 6.

When embedded processors are included in the FPGA design, whether hard or soft, additional tool support is required to facilitate hardware-software co-design. One component is a standard software development kit (SDK) for the embedded processor software. The more challenging component is support for the hardware-software interface. In current practice, this is fairly low-level, in that the user must specify details of buses that connect the processor to peripheral blocks implemented in the logic fabric. Then, the user must adjust details such as address maps for the bus and software device drivers for the peripheral blocks. Much research has been done on higher-level approaches to this hardware-software co-design, including

automated hardware-software partitioning and hiding of bus details, but this has not yet made its way into mainstream products from the FPGA vendors. Aspects of such research are considered in more detail in the next section.

A consequence of the ASIC-like design flow for FPGAs is that standard support for programming or reprogramming the FPGA is rudimentary: the tools generate a "bit stream" containing the programming information, and then this is loaded (via a relatively slow serial interface) into the FPGA. Thus, implementing adaptive FPGA systems is slow – first because of the time needed to generate replacement bit streams via the CAD tool flow, and second because of the time needed to load the new bit stream (which can be of the order of 100s of milliseconds). A notable feature of most Xilinx FPGAs is support for partial reconfiguration, where only selected parts of the FPGA are reprogrammed, thus reducing the loading time significantly when only small changes are being made. However, this requires specialized tool support. First, there is a need to indicate selected parts of the design that are subject to partial reconfiguration, and then there is a need to generate separate partial bit streams for these parts from the remaining background design. JBits [29] was an early gate-level design tool that was supported by Xilinx until 2004. Since then, a Xilinx partial reconfiguration (PR) flow [30] has been made available as an add-on to the standard design tools. However, PR requires non-trivial manual floorplanning of the design layout by the user to define FPGA regions that are to be partially reconfigured.

2.2 Modular System Design

Modular design involves partitioning a system design into modules of smaller complexity or building a system out of smaller preexisting sub-modules. Such designs involve two types of programmed description: for the structural network and for the behavioral modules. The structural description contains a listing of the modules and connections formed by wires or other communication pathways between modules. The behavioral descriptions describe the processing within the

modules, which is typically specified as a function reading a pattern of inputs to generate a programmed pattern of outputs. A modular approach to system design involves system modeling methods, inter-module interfaces, and tools and methodologies for modular system design.

2.2.1 System Modeling

A plethora of hardware and software system models have been proposed for exposing and abstracting different behavioral aspects of concurrent designs. Some of the early models for concurrent systems include formal models such as Hoare's Communicating Sequential Processes (CSP) [31] and Milner's Calculus of Communicating Systems (CCS) [32]. These models are useful for analyzing the behavior of concurrent software for undesirable properties like deadlock and livelock. Petri nets [33] graphically illustrate concurrent interaction and highlight synchronization barriers. The Unified Modeling Language (UML) [34] is a set of graphical models that illustrate separate design concerns, for example: class relationships, state charts, block diagrams, and interaction.

Ptolemy [35] is an influential research project conducted at UC Berkeley that features heterogeneous formal models, reflecting the practical desire to mix different models within one design. In Ptolemy terminology, the type or domain of the model is called the "Model of Computation" (MoC). Examples of MoCs are continuous time (CT), discrete event (DE), synchronous dataflow (SDF), and Hoare's CSP. The Ptolemy system model is hierarchical, like a tree composed of sub-models at each level, in order to constrain interactions between components. Lee et al. use formal models to define interface automata that describe the communication states of interfaces, as well as MoC specific parameters. Ptolemy has a simulation framework that supports building complex applications composed of heterogeneous models. In [35], a Ptolemy simulation was used to describe an SDF application that included DE components targeted to an FPGA.

System modeling is becoming an increasingly important aspect of designing for FPGAs, as modular approaches become mandatory to cope with target system complexity. There are a variety of different models in use, and understanding and verifying the interaction between modules to detect unintended behavior can be extremely difficult. This is exacerbated by the fact that programming applications for FPGAs exposes different levels of programming abstraction and high levels of concurrent execution. Practical models used in designing applications for FPGAs currently fall into three categories: dataflow programming, embedded system-on-chip (SoC) programming, and RTL programming. Dataflow models describe relationships between modules that indicate there is a movement of data between a first processing module and a second processing module, forming a pipeline. Dataflow models are at a higher level of abstraction than the target hardware, abstracting away wire signaling details. SoC models describe architectures in terms of a bus topology where modular components are connected to shared buses. Interactions are typically between a master component like a processor and multiple peripheral components, and a bus represents an abstraction of the wires that implement a data path, and the handshaking signals used for control. RTL system descriptions are at a lower level of abstraction and describe physical ports and individual wires that form connections. RTL descriptions additionally include low-level connection details like clocks and resets.

2.2.2 Module Interfaces

In most research on design at the system level, the focus is on the structural network, rather than the behavioral modules, which are treated as black boxes. However, it should be noted that there is also an extensive body of research on higher-level abstraction for modules. For example, there is a trend for higher-level tools to create behavioral hardware modules, notably electronic system level (ESL) tools that synthesize high-level C-like languages into RTL descriptions. These tools are informally called “C to gates” tools, and take a “C-like” description that usually contain extra pragmas that help to guide the high-level synthesis tools to

automatically extract parallelism from the "C-like" descriptions. AutoESL [36], Bluespec [37], Impulse C [38], and Synphony C [39] are some examples of high-level synthesis tools.

To reduce design complexity and enable productivity at the system level, standard module interfaces that facilitate "design reuse" are important. Traditionally, IP (intellectual property) cores have been tied to a particular technology or vendor because they use proprietary interfaces and metadata describing interfaces. For example, Coral [40] was a pioneering project on automated interface synthesis by IBM Research that involved synthesis of virtual SoC interconnections into physical bus structures, including synthesizing any necessary glue logic. It featured a tree classification for describing the functional, structural, and electrical attributes of module interfaces, and pin constraint matching for type checking compatibility of pins.

More recently, there has been an industry drive for standard ways to describe interfaces so that IP cores can be used interchangeably between different tool vendors. Initial work on describing module interfaces was done by the Virtual Socket Interface Alliance (VSIA) [41], which was an industry initiative to promote IP reuse and standardize terminology and models for SoC integration. IP-XACT [42], by the SPIRIT Consortium, now merged with Accellera, is a current initiative by leading EDA companies to develop a standard specification of design metadata, which will allow IP vendors to more easily exchange IP cores, and system design tools to more easily interoperate with tools from other vendors. IP-XACT was influenced by VSIA's work on Virtual Component Transfer, describing what types of data to include when packaging modules, called virtual components, for use by other companies. The IP-XACT object model supports transaction-level models (TLM), which are at a higher level of abstraction, in addition to RTL models. IP-XACT v1.5 was approved as IEEE standard 1685 in December 2009.

OpenCPI [43] is an open standard under development that is centered on the importance of interface abstraction for interoperability. It describes IP component interface descriptions that are abstracted into five interface-type categories: worker control, worker time, worker stream, worker message, and worker memory. These type abstractions form profiles for the Open Core Protocol (OCP) [44], which is an openly licensed interface standard for SoC integration. OCP is a functional superset of VSIA's Virtual Component Interface, adding configurable protocol options for sideband signaling and test harness signals. OpenCPI adds a thin layer of metadata for patterns of control, memory, data, and time to the Open Core Protocol.

CHREC [45] is a current research project in which an XML schema is used to create a portable IP interface description that can be used with multiple tools, and is intended to enhance IP-XACT's capabilities when FPGAs are being targeted specifically. CHREC XML comprises three layers: the RTL layer, the data type layer, and the interface operation information layer. The RTL layer describes low-level details of the core, the list of parameters for the core, and the related mathematical expressions for parameters. The data type layer describes high-level data types such as string, integer, floating point, fixed point, character, and boolean. The interface operation information layer contains information for high-level interface synthesis, for example to enable a tool to reason about the timing of signals, data dependencies, and latencies of signals. Recently, CHREC XML was further aligned with IP-XACT by describing metadata extensions for different types of parameterization of modules [46]. Some of the CHREC ideas have also been applied to descriptions of interfaces of heterogeneous CPU/FPGA systems for wireless [47].

2.2.3 System-level FPGA design tools

As mentioned earlier, system models for targeting FPGAs fall into three categories: dataflow programming, embedded SoC programming, and RTL programming. Current system-level design tools correspond to these three models. There is also

some support for integrating system designs that are constructed using the different types of tools, although this falls somewhat short of the grand vision of, for example, Ptolemy.

Tools for dataflow design are typically used to build stream-processing applications with spatial or temporal data parallelism that can be pipelined. Such tools often have a Digital Signal Processing (DSP) focus at present. The Mathworks Simulink [48] and National Instruments LabView [49] are examples of tools that can be used for constructing data flow applications that target FPGAs. Xilinx System Generator for DSP [50] is a toolbox for the Simulink based design environment that can be used to build fixed-point signal processing applications like DSP filters. Simulink provides libraries of modular filter blocks that can be simulated at a high level, and System Generator for DSP provides libraries of fixed-point blocks that have efficient FPGA implementations. Examples of System Generator blocks are FFTs blocks, adders, and multipliers. The System Generator blocks can be connected within Simulink to build a dataflow model and then the application can be compiled for FPGA implementation. LabVIEW offers a similar dataflow design environment to Simulink, but with a focus on designing laboratory instruments for data acquisition and diagnostic purposes. Much research on tools for packet processing using FPGAs (e.g. [51]) has used Click [52], which is a dataflow language for describing networking applications that was developed at MIT. Click, which is described in more detail in the next chapter, can be used, for example, to describe programmable routers in terms of simple descriptions that expose the main forwarding paths of packets.

SoC tools tend to be focused on building embedded processor-based designs. One example embedded design environment is Xilinx Platform Studio (XPS) [53], used for integrating embedded processor subsystems into FPGA applications. In XPS, modular components represent, for example, PowerPC and Microblaze processor cores, UART and Ethernet peripheral cores, and bus interconnect generators. The programmer builds the system on chip description by selecting components from a

library, specifying bus connections, and programming the address maps. Altium Designer [54] is another similar SoC design tool that features a schematic entry view, a graphical bus-based entry view, and a printed-circuit-board design view. Altium Designer supports a range of different embedded processors and also features an interactive FPGA diagnostic probe tool.

RTL structural descriptions typically instantiate modular RTL components, while the programmer specifies wire connections between them using RTL. The components are typically either static or parameterizable. Xilinx Core Generator and Altera MegaWizard are example of tools for generating specific RTL components drawn from parameterized library components.

Regardless of the original system model and design description, the analysis and verification of implemented systems must often be carried out at the lower RTL level only. Thus, validating designs typically involves compiling high-level models into RTL-equivalent models and then running RTL level simulations. It is less usual to 'back compile' RTL-level models to higher-level models, and then verify everything at a higher level. Designs are typically simulated to show functional correctness at edge conditions. RTL simulations are discrete event simulations, typically with waveform visualization to inspect signals within the implementation. Examples of RTL simulators are Mentor Graphics Modelsim and Synopsys VCS. Aside from the loss of any higher-level abstraction, it can be a very tedious and time-consuming process if there are many edge conditions. Some tools, like System Generator for DSP, allow simulation at a high-level functional level that respects the original dataflow model for the design.

2.3 Dynamic Reconfiguration

For researchers, a compelling feature of FPGA technology has long been the capability to build systems wherein the hardware can be modified during operation, that is, post design time. The availability of partial reconfiguration, where only

selected parts of the FPGA are updated, allows the option of more delicate, minimally invasive surgery. Research proposals range from fine-grain extremes, where tiny hardware features are updated on a very frequent basis, to coarse-grain extremes, where large hardware features are updated on a more occasional basis. While the former style offers more novel and exciting prospects, the latter style is more often seen in practical examples. This section overviews some significant research into mechanisms to support dynamic reconfiguration.

2.3.1 Hardware

The mainstream commercial hardware vehicles for dynamic reconfiguration are the Xilinx FPGA families. As mentioned earlier, physical programming of the FPGA is performed by writing a complete circuit configuration to SRAM after the device is powered on. Partial reconfiguration (PR), which is unique to Xilinx devices, involves modifying the circuit behavior at run time by writing an updated portion of the circuit configuration. Xilinx introduced partial reconfiguration in 1995 as a feature of the XC6200 FPGA family. This allowed very fine-grain partial reconfiguration support at the level of individual logic gates. When the XC6200 family was discontinued in 1998, the Virtex family became the mainstream FPGA architecture supporting partial reconfiguration. The original Virtex, and then Virtex-II, device architectures supported column-based reconfiguration of frames, in which columns spanning the entire device were grouped into coarse-grain units called frames that could be individually reconfigured. In the later Virtex-4 and Virtex-5, frames are smaller regions that no longer had to span entire device columns. However, a legacy of the earlier Virtex families was that many researchers still focus on reconfigurable designs that have a physical columnar structure.

The Virtex-II, and later, devices have three different physical interfaces for loading partial bitstreams into the FPGA fabric: JTAG, SelectMap, and internal configuration access port (ICAP). JTAG is the slowest programming interface. SelectMAP is an external interface for a coprocessor to write configuration frames.

ICAP is a wider internal port for an embedded controller to write updated configuration frames, and is the fastest programming interface.

Trimberger et al investigated an alternative FPGA architecture in [55]. This research proposed a time-multiplexed FPGA, which increased the configuration memory per logic region in order to multiplex configurations over time, for example eight configurations per region. Then, the time multiplexed FPGA emulated a single large FPGA, using time as a third dimension to the two-dimensional logic gate array. Its logic engine consisted of regions that switched configurations every micro-period, thus giving dynamic reconfiguration every clock cycle. Hence, the same regions could implement successive combinatorial logic before storing the result in flip-flops. Two other modes for logic regions were a time-share mode and a static mode. More recently and in fact 12 years later, Tabula, a startup company, announced a similar style of time-multiplexed architecture [56].

In another notable research project, Nagami proposed the Plastic Cell Architecture (PCA) [57], which was a cross between an FPGA architecture and a coarser-grain architecture, having a homogeneous cell structure. Each cell consisted of two parts: a static built-in part having fixed functionality and a programmable plastic part. Each cell also contained basic routing infrastructure to transmit between each part and to transmit to neighboring cells. The PCA offered the capability of provide highly flexible processing elements through the use of dynamic reconfiguration.

2.3.2 Software

As mentioned earlier, there has always been some basic Xilinx tool support for dynamic reconfiguration. However, these tools required the user to be fully aware of the physical floorplan of a design on the FPGA. The user was also wholly responsible for the organization and management of dynamically reconfigurable parts of the system being implemented. A historical analog of this situation was the

need for early programmers to directly manage memory overlays for programs and data. In general, this was not an attractive situation for the FPGA user.

Consequently, a major research focus over the past 15 years has involved approaching dynamic reconfiguration from an operating system angle, particularly bringing ideas from virtual memory management and translating these into mechanisms for virtual hardware management. This introduces two particular extra complications: first, two-dimensional FPGA regions must be managed, as opposed to one-dimensional pages or segments; and second, it is often necessary to provide physical connectivity between different regions, corresponding to connections between modules.

Some of the earliest research was by Hutchings et al. In [58], Hadley and Hutchings presented a design methodology for partial runtime reconfiguration, specifically for a runtime reconfigurable artificial neural network. Their methodology aimed to maximize static circuitry and minimize dynamic circuitry, and they implemented a feed-forward multiplier as a case study. In [59], Hutchings and Wirthlin compared two types of reconfiguration: compile-time reconfiguration and run-time reconfiguration. They presented different strategies for run-time reconfiguration: global and local. The global strategy involved total reconfiguration, based on a phased partition of the design, whereas the local strategy involved the use of partial reconfiguration, based on a functional partition of the design. This work was also targeted at an artificial neural network.

More general research on managing two-dimensional regions followed this pioneering work, and still continues. Brebner [60] proposed a virtual hardware-programming model consisting of swappable logic units (SLUs). SLUs were considered as virtual hardware components, with the motivation of extending a conventional operating system to manage SLUs in a similar manner to virtual memory. Two models were considered: the sea of accelerators, where SLUs were heterogeneous and not inter-connected, and the parallel harness, where SLUs were

homogeneous and tiled to allow nearest-neighbor connectivity. Diessel and ElGindy [61] also considered the two-dimensional placement problem, and in particular presented a complex two-dimensional compaction algorithm to reclaim free area after fragmentation occurred over time. This improved the utilization of the FPGA, and also the time taken for tasks to be executed using the dynamically configured blocks. A shortcoming of this early research is that it largely ignores the problem of implementing inter-module connectivity.

Partly motivated by the fact that the earlier Virtex family FPGAs only supported whole-column reconfiguration, but also motivated by the complexity of managing two-dimensional regions, many researchers have focused only on the one-dimensional problem of managing regions that span the whole height of the FPGA. For example, Brebner and Diessel [62] presented a scheme for removing fragmentation along a one-dimensional array of blocks, implemented using the FPGA itself. Blocks were arranged in a one-dimensional array of columns having variable width, and could either be allocated to tasks or left free. Unused available blocks were found using a first fit allocation scheme, implemented using a hardware-based string match for a set of n consecutive zeros on a binary string representing used or unused columns. A compaction algorithm was also provided, which shifted used blocks to the left in order to move free blocks together. In this way, the management of the FPGA was offloaded from operating software onto the FPGA itself.

Under the auspices of a national research program on dynamic reconfiguration funded by the German government, two notable examples of complete one-dimensional ("slot-based") reconfigurable architectures emerged. Majer et al introduced an extended FPGA architecture called the Erlangen Slot Machine [63] containing a configurable communications switch, which avoids the problem of feed-through paths. Feed-through paths occur when a module must reserve a circuit path cutting through a module, for example to provide access to external pins. Feed-through paths create a problem because they make reconfigurable modules less

portable. Ullmann, Hübner, and Becker [64] present an on-demand FPGA run time system for flexible and dynamic reconfiguration using a slot-based architecture for inter-module communication. This run-time system is implemented on a MicroBlaze soft processor and uses the ICAP for partial reconfiguration. The run-time system loads modules that have been compressed using LZSS compression and the loader decompresses partial bitstreams.

Wigley et al. presented ReConfigMe [65], the first complete operating system with runtime support for dynamic reconfiguration. The overall framework was partitioned into three levels of abstraction: platform tier, operating system tier, and user tier. The prototype was implemented using a coprocessor host to manage reconfiguration.

More recent research has focused on means of adding more flexibility to the implementation of communication paths between dynamically reconfigured modules. Koh and Diessel [66] proposed merging of communication graphs so that the communication infrastructure reuses wire paths. Using a fixed wiring harness and merged communication paths they reduced reconfiguration time. Suris, Patterson, and Athanas [67] presented WoD, a run-time router that can create routes between modules arranged in a slot array. At run time, new modules are placed in empty slots, and the router calculates new routes to and from the new module. This routing calculation takes four orders of magnitude less computation time than used by traditional design-time routers.

2.3.3 Theoretical models

Theoretical analysis of the computational capabilities and properties of dynamically reconfigurable systems has been a somewhat neglected research activity to date. The practical aspects of the field are still sufficiently immature that identifying underlying principles is difficult. One candidate for a computational model has been the Reconfigurable Mesh (RMESH) model [68] from the parallel computing world, which has some apt properties, such as an underlying two-dimensional grid, but

other less apt properties, such coarse-grain processing elements and per-cycle reconfiguration.

Lange and Middendorf have investigated the notion of 'hyperreconfiguration' [69], where dynamic reconfiguration is layered hierarchically, that is, one can structure reconfigurable regions. Hyperreconfiguration involves reconfiguring a larger area so that it supports smaller reconfiguration within its boundaries. A central problem is to determine when hyperreconfiguration steps should be taken, and how to define the reconfiguration potential in order to minimize the time for hyperreconfiguration of a computation. They showed that the Partition into Hypercontexts (PHC) problem is NP-hard in general, but can be solved for certain practical cases in polynomial time.

Malik and Diessel have defined the notion of the entropy of an FPGA reconfiguration [70]. This measures the entropy of a circuit that is to be configured, which leads to practical bounds on the minimum number of reconfiguration bits that need to be written to the FPGA in order to effect dynamic reconfiguration. This theoretical notion has important practical consequences, given the limited bandwidth available on FPGA configuration ports. They use Golomb encoding, which is a variant of run-length encoding, as a practical compression technique, and showed that the results were within 1 to 10% of the theoretical bound for a wide range of representative circuits.

2.4 Packet Processing using FPGAs

Traditionally, FPGAs have featured in a supporting role in networking systems: for providing physical interface logic or general glue logic. In recent years, this situation has changed significantly, with FPGAs assuming mainstream packet-processing roles, reflecting the need for hardware performance at increasing transmission rates, but coupled with programmability. The Internet commonly requires multi-gigabit data line rates in access networks and multi-terabit switches in the core networks. For example, the Cisco CRS-3 core router has a switching

capacity of 322Tb/sec. Modern, high performance FPGAs can be used to implement packet-processing functions at line rates surpassing 100 Gb/sec and packet switching at rates greater than 1 Tb/sec.

Telecommunication-equipment vendors, such as Alcatel Lucent, Cisco, Huawei and Juniper, perform much of the leading-edge research and development internally. Consequently, the work remains largely unpublished, although visible to FPGA vendors, such as Xilinx. However, there is an increasing quantity of published academic research in this area too. This section overviews some hardware platforms for FPGA-based networking research, packet processing functions that benefit from the use of FPGA technology, and comparison with other processing technologies.

Figure 2.1: Example of a telecommunication line card

2.4.1 FPGA-based platforms

In real-life networking equipment, FPGAs usually feature as a part of telecommunication line cards, an example is shown in Figure 2.1. These plug into a switching backplane via high-speed interfaces, an example backplane being shown in Figure 2.2. Packets arrive on an input connection into one line card, are processed there, then passed to the switch, and emerge onto another (perhaps on the same) line card for further processing before departing on an output connection. In research settings, FPGAs more often feature on standalone cards that include any switching

capability as an integral on-board feature. Such cards often plug into standard computer workstations.

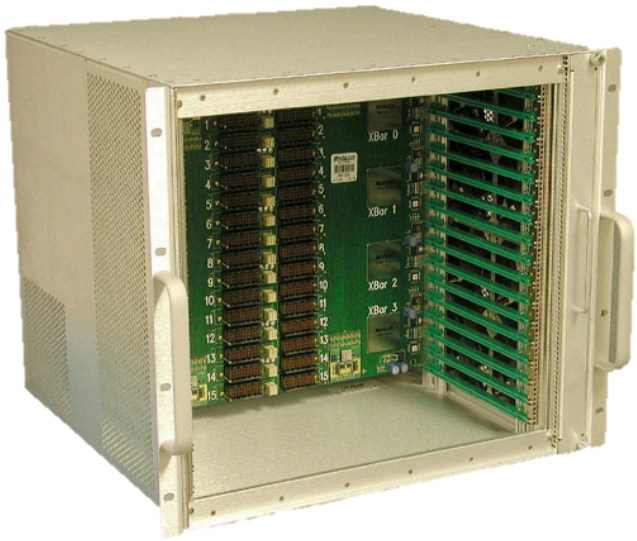

Figure 2.2: Example of a switch backplane, which seats multiple line cards

Here, three major academic research platforms are described, which are all public domain and have found widespread usage in the international networking community for both research and teaching. However, there are other instances of specific boards being built for particular research projects, or the use of standard boards available from FPGA vendors.

The pioneering platform was the Field-programmable Port eXtender (FPX) developed at Washington University in St. Louis [71]. This platform was a plug-in line card that sat between a 2.5 Gb/sec line interface and the switch backplane of a multi-gigabit router. The FPX could be deployed within a core router or an edge network – though line rates have now increased significantly since the FPX's introduction. It has been used by researchers to implement accelerated versions of many networking applications ranging from fast Internet Protocol lookup, to content scanning and replacement, to network intrusion detection, and to streaming video processing.

NetFPGA [72] is an ongoing project at Cambridge University and Stanford University, incorporating ideas from FPX, and currently providing the standard worldwide research platform. The original version was a plug-in PC card with four 1 Gb/sec Ethernet interfaces and one PCI interface. In early 2012, there were around 2200 of these cards in use worldwide. The second-generation version, NetFPGA 10G, was completed in 2010. It improves upon the original platform by using a more modern and much larger FPGA, four 10 Gb/sec Ethernet interfaces and one PCI Express interface.

NetCOPE [73] is a networking platform that was developed by Brno University of Technology in the Czech Republic. Like NetFPGA, it plugs into a standard PC, and one application for it is as a high-speed host-based network interface card (NIC). As well as the hardware board, NetCOPE comes with a firmware abstraction layer that includes common modules required in networking applications.

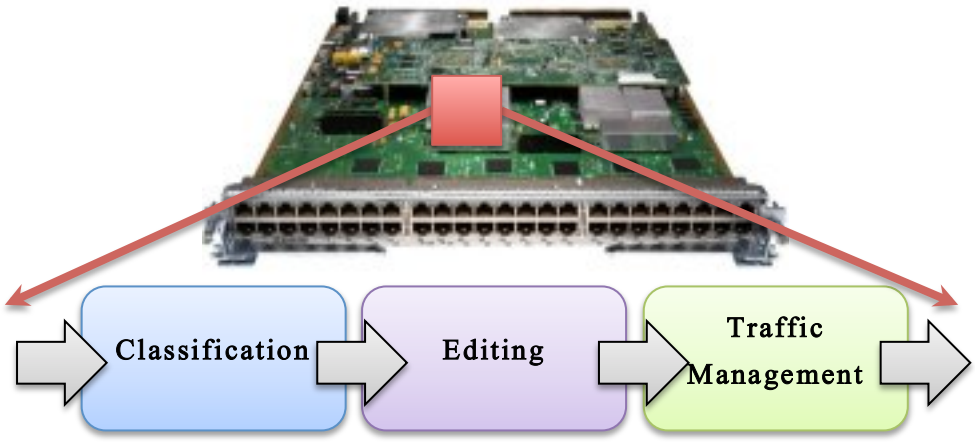

Figure 2.3: Types of packet processing functions of a line card

2.4.2 Packet processing functions

Functions in the main packet processing data plane can be divided into three broad groups: classification, editing, and traffic management, as shown in Figure 2.3. Classification involves some parsing and checking of a packet to ascertain its relevant properties or ownership. Editing involves changing fields in packet headers (or trailers), inserting or removing fields, splitting and joining packets, or completely dropping packets. Traffic management involves queuing and scheduling of packet departures to meet quality of service goals. FPGAs find application for all these functions, particularly when they are on the "fast path" where a function is applied to all packets passing through (at high speed). Associated with these packet processing functions are management activities like gathering statistics and maintenance, which may be orchestrated by embedded processors on an FPGA.

Classification has attracted much attention in the research community, particularly to exploit the ability of FPGAs to perform high-speed table lookups or pattern matching. Basic classification to determine packet forwarding through a switch involves extracting one or more packet header fields, and using these as a key into a lookup table containing forwarding information. These tables are typically implemented as Ternary Content Addressable Memory (TCAM), the ternary aspect allowing wild cards in table entries. Commodity TCAM devices are available, but these consume higher power compared to using FPGAs together with on-chip or off-chip memory. High speed, power-efficient FPGA-based approaches have been demonstrated using pipeline architectures [74] or hash-based techniques [75].

Security applications, such as network intrusion detection, typically require not just inspecting packet headers, but also scanning the contents of packet payloads. The latter is often referred to as deep packet inspection (DPI). Efficient ways to implement content scanning applications involving regular expression matching have been well researched (e.g. [76] [77] [78] [79] [80]). Because they require checking an incoming packet against a large database of rules for content, they are easy to accelerate by checking the rules in parallel using an FPGA. Snort [81] is an

example open source software program for network intrusion detection, and the standard Snort rule database has been a frequently used example rule set for benchmarking FPGA implementations. The basic implementation of regular expression matching normally involves execution of an equivalent deterministic finite automaton (DFA) or non-deterministic finite automaton (NFA). In [76] for example, Hutchings presents a regular expression matching application implemented on an FPGA using an NFA approach, which was tested on regular expression rules derived from the Snort database. Bloom filters have been used as a different approach to implementing DPI [82]. Parallel Bloom filters used for dictionary lookup were chained together to implement string matching for thousands of rules, as a series of probabilistic hash functions. In addition to scanning for content, some researchers have also included packet modification to actively decontaminate packets. For example, Moscola et al. [78] scan packets for Internet viruses and spam and then neutralize infected packets by removing the malicious data.

Other research has focused on high performance protocol handling though packet parsing and editing. Fallside and Smith [83] demonstrated FPGA implementations of networking protocol layers, implementing the standard ARP, IP, TCP and UDP protocols over an Ethernet connection. Schuehler [84] presented a TCP splitter for stateful monitoring of thousands of TCP/IP flows. Marcus et al. [85] described "protocol boosters", a general approach to implementing protocols on FPGAs. Protocol boosters were based on the idea that new protocols can easily be constructed by incrementally adding inline protocol booster components to accelerate a baseline protocol. Brebner [86] presented a high-level language and compiler for automatically building high-performance packet parsing and editing pipelines on an FPGA. Soviani and Hadzic [87] also presented research on high-level synthesis and optimization of packet processing pipelines.

Comparatively little research on traffic management has been published, although this area is of great importance to networking equipment providers and has generated many patents. Zhang et al. [88] presented a programmable traffic manager (TM)

architecture for supporting simultaneous scheduling of uni- and multi-cast traffic in a packet switch. This had a modular architecture, allowing flexible configuration in terms of number of packet queues, scheduling algorithms, etc.

2.4.3 Networked FPGA programming

A notable converged-application area is the dynamic reconfiguration of FPGAs at a distance, over networks. Casselman [89] presented an API for a networked FPGA that can be remotely reconfigured with new bitstreams. Horta and Lockwood [90] partitioned the application FPGA within the FPX platform into two logical halves so that each half could be reprogrammed at runtime using partial bitstreams transmitted over a network. Circlets [91] were proposed as a model for networked FPGA programming, inspired by the runtime portability enabled by Java applets. Circlets involve programmed circuit descriptions made portable by targeting an abstract FPGA two-input LUT. Each such LUT was then dynamically mapped on to an actual FPGA's *n*-input LUT implementation.

2.4.4 Comparison with other technologies

FPGAs are well suited for accelerating packet processing functions because modern networking often requires very fast transmission rates, but also offers a high amount of potential independent data parallelism. Networking applications tend to have a high number of contexts or flows, which can be processed with parallelism across different contexts, as well as between individual packets within flows. Compared to general-purpose microprocessors, FPGAs can be used to program new custom architectures to implement parallel functions with low latency and high throughput. FPGAs can effectively implement thousands of concurrent operations mapped onto the logic gate resources, whereas current generation general-purpose microprocessors have only 2 to 8 cores for implementing thread-level and instruction-level parallelism.

Network processors (network processing units: NPUs) are a high performance alternative for packet processing. They are programmed using C and/or assembly level programming of their functional components, which may be pipeline stages, micro-engines, or specialized function units. There is currently a wide diversity of NPU architectures and a lack of standards for programming NPUs, which is a negative point since it hinders porting and reuse of NPU firmware. Some NPUs support instruction and thread level parallelism since they also contain multiple processor cores. For example, the Intel IXP 2800 [92] has 16 programmable micro-engine cores for performing packet inspection and traffic shaping for line rates up to 10 Gb/sec. Other NPUs feature programmable packet processing pipelines. For example, the Xelerated X10q NPU [93] can perform packet processing at line rates up to 40 Gb/sec. Newer network processors such as EZchip's NP-5 is targeted for 200Gb/sec packet processing, and acceleration of a variety of traffic management and Carrier Ethernet functions [94]. Cavium's Octeon III contains up to 48 2.5 GHz MIPS64 cores for processing multiple lanes of 40Gb/sec traffic [95].

Designing application specific integrated circuits (ASICs) for networking can potentially offer the highest level of performance, but ASICs in general require involve much higher engineering costs and more time to design, fabricate, and test than using FPGAs. ASICs require high initial mask costs that are continuing to rise. FPGAs present a higher unit cost, with no initial cost, and so they are preferred for lower volume production. FPGAs can have the performance advantage that, because of high volume production they can leverage the latest silicon process technologies before they are available to new ASICs [96].

In practice, mixtures of technologies are often used when implementing complete networking or telecommunications systems. FPGAs can be programmed with standard external interfaces to connect directly to other networking devices such as ASICs or NPUs, as well as to general-purpose CPUs. This way, FPGAs can be selectively used to implement the particular functions that are best suited to the

capabilities of technology. The work presented in dissertation could be used to target the FPGA portion of these hybrid systems.

FPGAs have an attractive technological case for networking systems, but the current design methodologies are a barrier to entry. Currently FPGAs operate at a fairly low-level of abstraction. This dissertation presents a design methodology and tools to bring the four main categories of this chapter together in order to raise the level of abstraction.

Chapter 3

ShapeUp: A High-Level Design Approach to Simplify Module Interconnection

This chapter introduces ShapeUp, a high-level approach for designing systems by interconnecting modules, that gives a 'plug and play' look and feel to the designer and is supported by tools that carry out implementation and verification functions. The emphasis is on the inter-module connections and abstracting the communication patterns that are typical between modules – for example, the streaming of data that is common in many FPGA-based DSP or networking systems, or the reading and writing of data to and from memory modules. The details of wiring and signaling are hidden from view, via metadata associated with individual modules. The ShapeUp tool suite includes a module interface type checker and a design environment with a novel visualizer.

Custom computing has come of age with the advent of large Field Programmable Gate Array (FPGA) devices that enable the implementation of complex application-specific configurable systems. For example, the largest Xilinx FPGA device has around two million programmable logic cells, with larger devices to be expected in the future. The double-edged sword is that actually designing these now-feasible systems is an increasingly complex engineering task, and so methodologies above and beyond traditional FPGA design flows are required to improve designer productivity. Learning from other disciplines, modular design and reuse are essential to make progress, along with higher levels of abstraction in design specification. While considerable research energy has been focused on abstraction of functional descriptions of modules, the complementary topic of abstraction of module

interconnection has been somewhat neglected. This chapter describes work that addresses the hitherto neglected area of 'system plumbing'.

Xilinx has adopted the term "plug and play IP" to refer to a vision of modules – *IP* (Intellectual Property) blocks – that can be used together in a plug-and-play manner without the need for significant effort. To make this modular vision real and incorporated into design methodologies (for both static and dynamically reconfigurable systems), three key module interconnection enablers are required. The first is increased standardization of module interfaces, to replace the plethora of legacy ways in which candidate modules have been interfaced to their environment. The second is standardization of metadata formats used to describe the nature of module interfaces (and modules themselves). The third is high-level tools that can interpret such metadata in order to provide assistance with building systems from modules by making connections between module interfaces.

While increased standardization of interfaces is welcome, complete standardization is not seen to be achievable, or indeed desirable, and one aim of the work reported in this chapter is to provide assistance in connecting interfaces that have compatible semantics but may have different 'syntaxes'. A simple FPGA example is where two interfaces have a similar role, but have different data widths, e.g. one is 32-bit wide and the other is 128-bit wide.

For expressing module metadata, the increasingly influential IP-XACT standard [97] from Accellera [98] (which absorbed the work of the Spirit Consortium in 2009) is a most appropriate approach, and is the chosen substrate for this work as it progresses. In earlier research prototypes, custom metadata formats were devised and used but these have now been superseded by alignment with the IP-XACT standard – and indeed they have suggested some possible future extensions to the standard.

The ShapeUp approach to providing higher-level tools that assist in higher-level modular system design has been founded upon the definition of a clean, but

pragmatic, set of abstractions of module interface behavior. This set captures the semantics of standard (or less standard) interfaces, and is associated with a standard metadata format that is used to describe these semantics. Thus, the modules themselves are treated as black boxes, and the focus is entirely upon their interfaces and connections between these interfaces. Not surprisingly then, the work has a networking flavor to it, being concerned with respecting and implementing communication protocols between interconnected modules.

Current tools for assisting FPGA-based module interconnection can be divided into three broad categories, each with very different natures. The first are traditional HDL level tools targeted at the hardware design specialist, for example Mentor HDL Designer [99]. Here, there is no interconnection abstraction above Verilog or VHDL hand wiring between blocks. The second are tools targeted at embedded system design, and strongly influenced by ASIC system-on-chip methodologies, for example Xilinx Platform Studio [100]. Here, there is a setting of processors, buses, and peripherals, in other words a specialized style of module interconnection. The third are tools targeted at the DSP domain, for example Xilinx System Generator for DSP [50]. Here, there is a domain-specific setting of DSP components and streaming dataflow between them. Complete systems are typically constructed using all three types of tools in tandem, perhaps uncomfortably. ShapeUp was designed to provide the basis for a more uniform tool framework, in terms of both level of abstraction and breadth of application.

One requirement for ShapeUp is a notation by which a user can describe the connections made between module interfaces. The Click language [52] was chosen for this purpose, continuing its use from earlier FPGA-targeted research [51]. Click originated at MIT, and is much used in the networking research community for describing software systems built out of modular components that are ‘clicked together’. An overview of Click is given in Section 3.1. Aside from one minor generalization, ShapeUp uses the Click syntax as is; however, a significantly

generalized underlying semantics has been added in order to broaden the applicability of this domain-specific language.

In terms of recent research in this area, the ShapeUp work has closest relationships with the CHREC XML work of Wirthlin et al. [101] [102], discussed in Section 2.2.2. This features an XML data schema that goes beyond current IP-XACT to address module metadata requirements for reconfigurable computing. A simple demonstration module integration and reuse tool based on CHREC XML metadata is presented in [45].

3.1 The Click Language and Extensions

There was nothing deeply profound about the particular language choice in Click. Click is a declarative language for representing a directed graph of connected communicating elements. However, other benefits include: both graphical and textual representations; hierarchical graphs; and parameterization of vertex properties. All of these extras are useful for the practical requirements of ShapeUp descriptions.

This section briefly describes the Click language and provides simple examples written in Click. Routers are devices that guide packets from one network to another by classifying incoming packets and forwarding them towards their destination. Click simplifies the designs of software-based routers: (a) by providing modular abstraction of the software, (b) by exposing the main forwarding path of packets as they are processed and buffered by the router, and (c) by providing an open source library of reusable modules. A full description of Click can be found in [52]. The Click short examples that follow are taken from [103].

In Click terminology, the graph vertices are called elements (which are modules) and the edges are called connections. Elements process packets, and packets flow over

connections. Elements can have an arbitrary set of input and output ports. Connections are made between ports on elements (which are module interfaces). So packets are transmitted at output ports and received at input ports, and this constitutes the complete interconnection semantics of standard Click. Elements have an optional configuration string, which contains parameters for initializing the element.

A simple textual example is shown below. Since Click is a declarative language, a description consists of declarations of: (a) elements and (b) connections between elements.

```
// Declare three elements ...
src :: FromDevice(eth0);
ctr :: Counter;
sink :: Discard
// ... and connect them together
src -> ctr;
ctr -> sink;
```

An example configuration string is "`eth0`", indicating that the `src` will capture packets from the `eth0` device interface.

This Click example could be reduced to just one line using Click syntactic sugar, for example:

```
src :: FromDevice(eth0) -> ctr :: Counter -> sink :: Discard;
```

When elements have multiple input and/or output ports, port numbers are used to identify them, for example:

```
src[2] -> [0]ctr;
```

Figure 3.1 illustrates the graphical Click syntax. The first example (a) shows the graphical syntax of an element where a box represents an instance of an element, and symbols represent the ports, text describes the element class and also the configuration string. The second example (b) shows Click graphical connection syntax, where arrows between boxes represent connections.

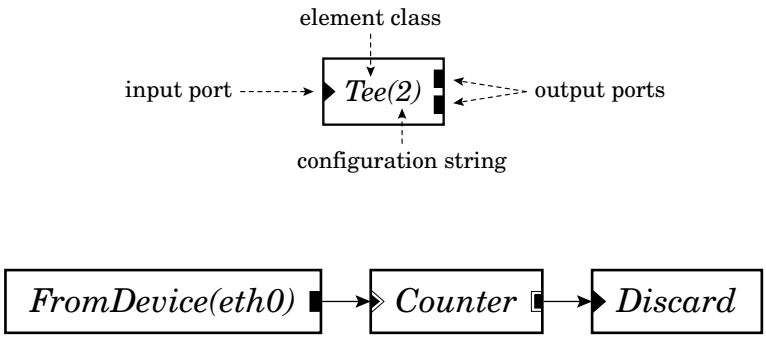

Figure 3.1: Simple Click examples: (a) A sample element. Triangular ports are inputs and rectangular ports are outputs; (b) A simple graphical Click example of a three element pipeline

The standard implementation of Click is in software, and C++ objects represent elements. Elements have one or more method interfaces e.g. for obtaining information about packets and packet transfer between elements. An undesirable feature is that sometimes elements have behind-the-scenes interactions through other method calls, and these interactions are not explicit in the provided Click description, which captures only packet dataflow.

Click's model supports hierarchical designs. Elements can be either simple elements or compound elements, and they are stored in a library. Compound elements consist of an aggregate group of elements, which is treated as a single element. The Click programmer can create new compound elements from existing library elements. A textual example of a compound element is shown on the next page, with one input port and one output port. The corresponding graphical example of the same compound element is shown in Figure 3.2.

```
elementclass SFQ {
  hash :: HashSwitch(…);
  rr :: RoundRobinSched;
  input -> hash;
  hash[0] -> Queue -> [0]rr;
  hash[1] -> Queue -> [1]rr;
  rr-> output;
}
```

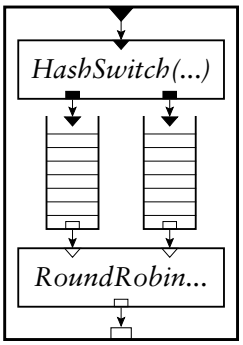

Figure 3.2: Click compound element example of a simple switch

Another aspect of Click connections is the *push* and *pull* abstraction for ports, illustrated in the example in Figure 3.3. A port on an element can be either push or pull, which indicates which side of the connection controls the movement of data. Push ports are indicated as black symbols and pull ports are indicated by white symbols. A third type of port control flow is *agnostic*, which is used when the port matches either push or pull behavior of the port directly connected to it. An agnostic port is shown on the null element in the example, represented by a double outline. This is an important feature of the software implementation because it also directly indicates control compatibility between the ports of two modules. However, this feature is less relevant to hardware implementations, where the control flow is normally agnostic.

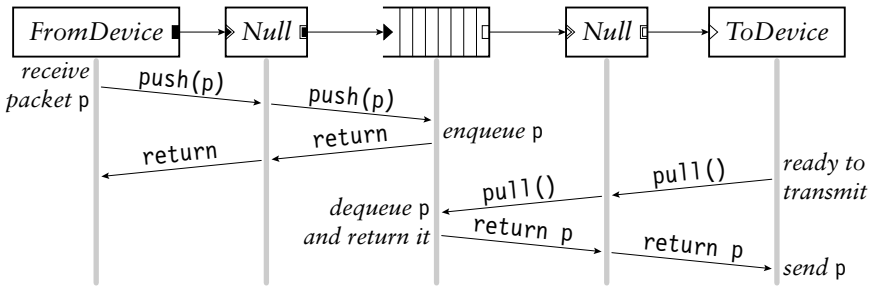

Figure 3.3: Click example featuring both push (black ports), pull (white ports)

For ShapeUp, the implementation of Click might be in either software or hardware, or both. On the hardware side, packets are transmitted over wiring that forms connections between hardware modules. Between hardware and software, interfacing elements are based on software device drivers. Aside from these implementation aspects of connections though, the significant semantic change to Click was in allowing connections to represent a much wider range of element interactions than just the transmission of packets, as will be described in Section 3.2.4. This generality is based on the module interface abstractions presented in the next section. To aid clarity, the only (tiny) syntax change in ShapeUp was to allow alphanumeric port identifiers, not just numerical.

3.2 Abstractions of Module Interface Behavior

To motivate the desire for greater abstraction, Figure 3.4 shows a simple example drawn directly from the world of the hardware designer. It concerns a module, which is a FIFO for packet data with parameterizable depth. This FIFO has a 64-bit data path width, and uses the Xilinx LocalLink [104] standard for its input and output interfaces. Figure 3.4(a) shows the actual schematic for the FIFO, featuring the detailed input and output wiring detail (over which LocalLink standard signaling is carried out). Figure 3.4(b) shows a graphical Click description of this FIFO

abstracted, with just 'stream type' input and output ports. The abstracted version might have either a hardware or software implementation. Automatically bridging the gap between the Click abstraction and the implementation is the goal of this research.

ShapeUp is intended to go far beyond support for just the traditional Click packet type of interactions between modules, however. Figure 3.5 shows a larger example depicting the interface of an Ethernet MAC module. This features three other interface types that will be introduced in the next subsections.

Since modules are treated as black boxes, the range of interface abstractions is based on broad observations about overall module behavior and how this is programmed. Three basic programming paradigms were identified: (i) hardware programming; (ii) communications programming; and (iii) procedural programming. These, and their impacts on interface behavior, are described in turn below, together with one, two, and two, derived interface abstractions respectively. Each of these module interface abstractions has an (open-ended) set of attributes associated with it, used to specify the characteristics of particular instances of the type.

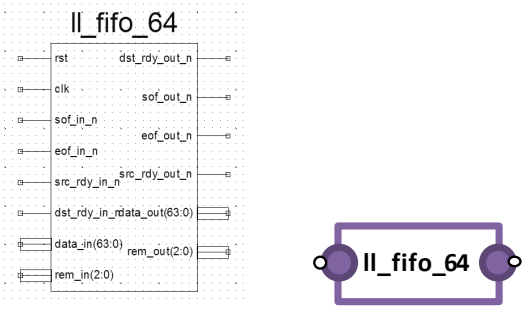

Figure 3.4: (a) Schematic view of FIFO; (b) ShapeUp view of FIFO

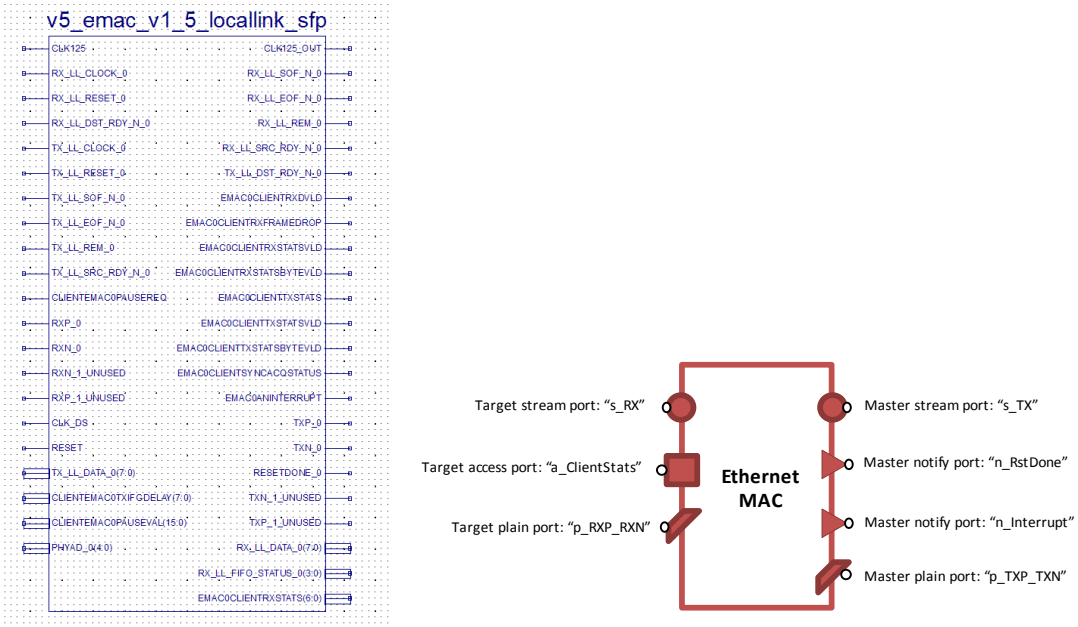

Figure 3.5: (a) Schematic view of Ethernet MAC; (b) ShapeUp view of Ethernet MAC

3.2.1 Hardware-programmed modules

This paradigm is somewhat less deep than the other two, being the placeholder for hardware modules that exhibit some arbitrary pattern of signaling over wires at their interfaces. In short, it leads to a lower-level interface abstraction included as a fallback option for module interfaces that are not conveniently described using the other, more sophisticated, abstractions. This interface abstraction is termed *plain*, and its basic attribute is that a logical vector of bits is transmitted unidirectionally over the interface. An important additional implementation attribute is whether transmission is synchronous or asynchronous. In synchronous mode, communication is by polling at agreed times. In asynchronous mode, changes in bit vector values are explicitly communicated, for example, using an edge-triggered signaling approach at the receiver.

3.2.2 Communications-programmed modules

This paradigm captures modules that process streaming data, for example, DSP samples or network packets. Two interface abstractions are defined, corresponding to the connectionless (datagram) and connection-oriented (virtual circuit) styles that are universal in data communications.

The connectionless interface abstraction is termed *notify*, and its basic attribute is that atomic messages are passed unidirectionally over the interface. A typical usage is that these messages are used to signal the occurrence of events. Communication of messages between two modules connected via notify interfaces is lossless and sequenced. This interface type can be seen as the next higher level of abstraction above the plain interface type, adding a little data structuring above basic bit vector transmission.

The connection-oriented interface abstraction is termed *stream*, and its basic attribute is that a stateful stream of atomic packets (equivalently: samples or tokens) is passed unidirectionally over the interface. An important additional attribute is that there can be a flow control mechanism for the stream, to police the rate of transmission. Typically, flow control is applied by the receiving module to avoid data loss, though explicit flow control by the transmitting module is also possible.

For both the notify and stream types, the atomic data units transmitted have various attributes. These include implementation attributes, such as parallel data widths and start/finish indications, and structural attributes, such as data formats and interpretations.

3.2.3 Procedural-programmed modules

This paradigm captures modules that embody the standard software programming mechanisms of accessing variables and calling functions. Two interface abstractions are defined, corresponding to these mechanisms

The first interface abstraction is termed *access*, and its basic attribute is that a primary module accesses data in a secondary module via read and write requests, for example, processing that interacts with memory, or processing that uses input/output devices. A particular attribute is whether the accesses are addressless (e.g. to a register or a FIFO) or addressed (e.g. to a memory array). Another implementation attribute is whether read or write requests can be grouped together into bursts.

The second interface abstraction is termed *compute*, and its basic attribute is that a primary module calls a function in a secondary module by passing arguments and receiving results back. This is analogous to the networking notion of a remote procedure call. A particular attribute is how the target function is specified.

In both cases, there is a simple two-stage handshake protocol between the primary and secondary modules (though the second stage of the handshake may not be explicitly required for access writes). This is analogous to the flow control capability of the stream interface type.

As for the notify and stream types, the atomic data units transmitted (read or written values, function arguments and results, respectively) have both implementation attributes and structural attributes.

3.2.4 Module interface types and Click semantics

Figure 3.6 shows a summary of the five interface abstractions indicating the basic interactions between two modules connected using each of the five types. This also indicates that there are three basic layers of abstraction: plain; then notify; then stream, access, and compute. Aside from plain, the interface abstractions are applicable to both hardware and software implementations of modules. Note that notify corresponds to the common notion of message passing between modules in software implementations.

In the standard Click semantics, a connection denotes streaming packet dataflow from one module to another and so, in either the textual or graphical representation, an arrow denotes direction of data flow. For ShapeUp, the meaning of a connection arrow is generalized to denote a primary-secondary relationship: a primary element initiates an interaction with a secondary element. In the particular case of the stream interface type, this is equivalent to the standard Click semantics. In the case of the access interface type for example though, the arrow shows the direction in which read and/or write requests are made. So, for a write request, data flows in the direction of the arrow. However, for a read request, data flows against the direction of the arrow. In the case of the compute type, data flows in both directions.

Note that no extension was made to the Click syntax in order to explicitly indicate the interface type associated with element ports. However, given that port name syntax was generalized to allow alphanumeric identifiers, something like Hungarian notation [105] can be systematically used: prefix the port name with an indication of type (e.g. "P_" for plain).

3.3 Interface Metadata

The ShapeUp design environment requires that each module has metadata associated with it, describing its interfaces in terms of the abstract interface types and their attributes. Thus, repositories of modules available for use and reuse store this information alongside other metadata about the modules (for example, descriptions, creation times, etc.). When users create new custom modules, then the appropriate interface metadata must be created at the same time. With maturity of the ShapeUp flow, higher-level tools used for module creation can also be made to generate the required metadata automatically.

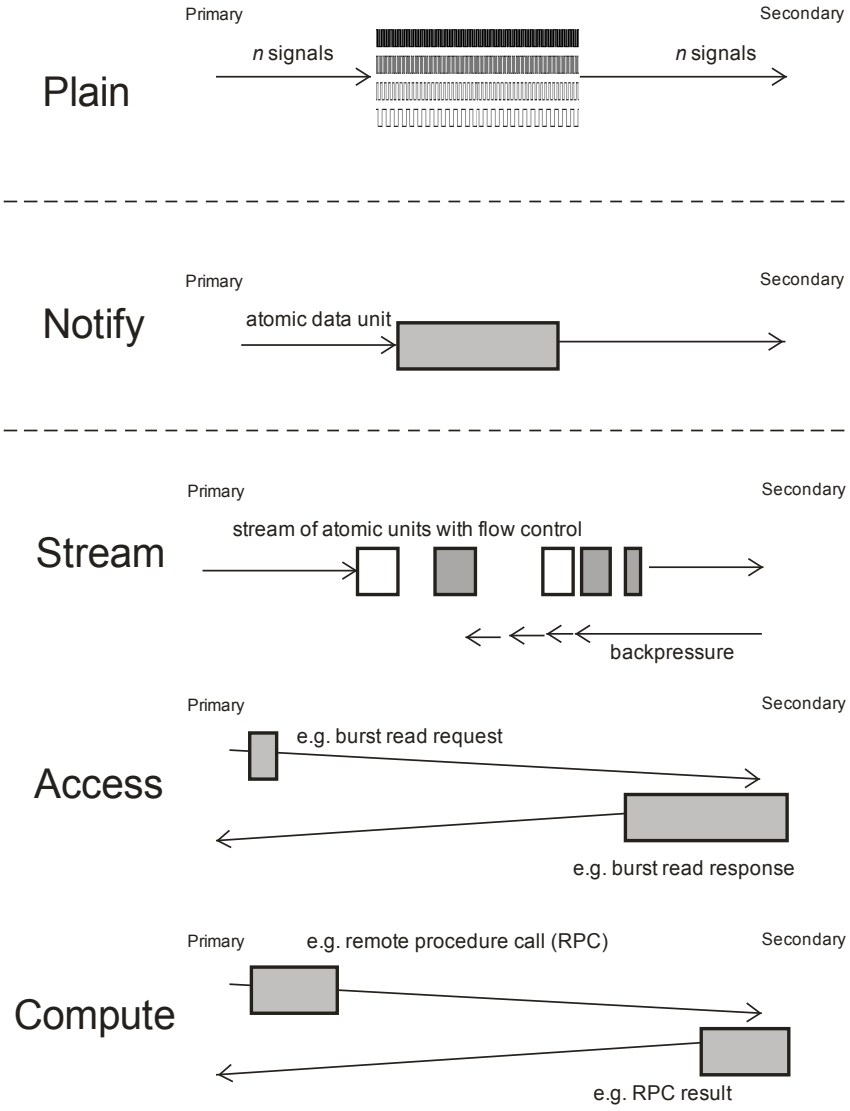

Figure 3.6: Module interaction with five interface abstractions

The interface metadata is described using the ShapeUp Element Description Language (EDL). This follows a data schema giving, for each interface on a module (i.e., for each port on a Click element), the required information. The two essential pieces of information are the interface type, and whether the interface is primary or secondary. These provide the most basic type-checking capability: to determine whether a legal connection can be made between two modules using their respective interfaces. Basically, both have to have the same type, and one has to be primary and the other has to be secondary.

Beyond the basic interface metadata, the EDL description includes more detailed information about the various attribute values that apply for the particular interface type instance. These can be used for more detailed type checking, as well as to guide the operation of the various ShapeUp tools, as described in the next section. To give a feel for the potential descriptive power of the metadata, the current ShapeUp prototype has 24 defined available attributes for the stream interface type, and 38 attributes for the access type. (The larger number for access is a reflection of greater tool experimentation using this interface type.)

To make the mechanism as flexible and table-driven as possible, the EDL data schema is itself described using meta-metadata described in the ShapeUp Interface Description language (IDL). This follows a data schema giving, for each defined abstract interface type, the required metadata information. So, it expresses the basic behavior of an interface type, together with its attributes, giving type and range information, and default information, for each. Thus, the chosen five types need not be seen as being tablets of stone, but as an initial pragmatic selection that can be easily evolved and upgraded based on practical experience and learning. Ultimately, certain knowledge of the interface type behavior is built into the tools that process EDL descriptions, since it is not practicable to make IDL a language that can express all conceivable interface behaviors.

3.3.1 Stream attributes example

Example attributes for the stream interface type are provided in Figure 3.7. Stream is of particular focus since it is necessary for describing packet processing systems. These attributes were based on characterizing the behavior of typical streaming interface protocols of modules, e.g. Xilinx's LocalLink, ATM Aurora, ARM's AXI-stream, etc. The attributes describe both the format of data and the behavior of the transmission. A short example of the IDL for the stream interface type is shown in Figure 3.8. A corresponding example of EDL for an element is shown in Figure 3.9, which has a stream input port.

For each of the interface types there are attributes for describing the behavior of the interaction. Each connection forms a control relationship connecting a *primary* (controller) to a *secondary* (target). As mentioned, the basic functions of a stream interface are to transmit and to receive network packets or tokens. Packets or tokens are transmitted by the primary to the secondary for one-way communication. The control plane of stream allows the secondary to throttle the transmission rate by asserting flow control/backpressure to pause the transmission in order to avoid data loss. The following is a more detailed description of the stream behavioral attributes.

There are attributes for indicating the word length and endianness in the transmission. The packet length, and whether the packet can be segmented into smaller *chunks* or cells is also indicated. The method of specifying the length of the packet is also described.

The method for indicating packet boundaries chunk boundaries is also an attribute. For example, the start of packet can be indicated by (a) a marker that is a signal or by a reserved position in a packet header, (b) the assertion of a ready signal, (c) ready plus a status or control word, (d) determined by a time slice, or (e) determined by some other encoding. The start of a chunk may also be indicated by these encodings. The end of packet can be indicated by (a) a marker that is a signal or by a reserved position in a packet header, (b) the assertion of a ready signal, (c) a length field, (d)

determined by a time slice, or (e) determined by some other encoding. The last word of the packet is indicated either by a specified length or remainder signal that indicates how many bytes are valid in the last word of the packet, called *rem.* The rem may have different encodings, for example the number of bytes can be expressed in binary or as a byte enable.

There is a timing relationship between the primary and the secondary. Flow control, or backpressure, can be asserted by the secondary to indicate to the primary to pause the sending of any more data until the secondary releases the backpressure. The transmitter also signals its readiness, and the primary can pause transmission by deasserting data enable or its ready signal.

In terms of a transaction, the transmission of the packet or token is either all or nothing. The transmitter, or primary, can abort the transmission of the current packet by asserting its abort signal. The receiver, or secondary, can similarly abort the transmission of the current packet by asserting its discontinue signal. A transmission of packets or tokens is order preserving, meaning that the data is received in the same order as it was transmitted.

Stream interfaces may have an associated logical identifier that can be used for different purposes, e.g. identifying channels or unique identifier for each interface. Channels are typically a grouping for packets based on a low-level physical transmission property, e.g. channels are found in wireless transmission based on frequency or optical transmission based on wavelength. Grouping by flow identifier is more common for packets.

For preserving data integrity, whether the transmission interface supports parity bits is described. Alternatively whether the entire packet or chunk contains a checksum, e.g. CRC, is also described.

Relationship of Endpoint (primary | secondary)

Stream Data Format

Intra-word

1.Size of word (specify: bytes | bits; specify min & max)
2.Structure:
a) byte order
b) bit order
c) Method of specifying the completeness of a word (or less than max bytes)
(rem | byte enable | length)

Chunks / Short Inter-word / Horizontal *(Chunk = Cell / Burst / Segment)*

0. Supported by Protocol
1. Size of chunk (specify words | bytes | bits; specify: min & max)
2. Structure:
a) method of specifying start of chunk
(marker { signal | reserved position within hdr } |
assertion of ready signal |
ready plus a status/ctrl word |
determined by time slice | other encoding)
b) method of specifying completion of chunk
(marker { signal | reserved position in trailer } |
deassertion of ready signal |
length field | determined by time slice | other encoding)
3. Uses Data Offset (Y & specify value | N)

Packets / Long Inter-word / Aggregate Horizontal

1. Size of packet (specify chunks | words | bytes | bits; specify: min & max)
2. Structure:
a) method of specifying start of packet
(marker { signal | reserved position within hdr } |
assertion of ready signal | ready plus a status/ctrl word |
determined by time slice | other encoding)
b) method of specifying completion of packet
(marker { signal | reserved position in trailer } |
deassertion of ready signal | length field |
determined by time slice | other encoding)
3. Uses Data Offset (Y & specify value | N)

Transmission Behavior

Intra-word Data Enable (Y & specify signal | N)
Chunks Data Enable (Y & specify signal | N)
Chunks Flow Control (Y & specify signal | N)
Packet Data Enable (Y & specify signal | N)
Packet Flow Control (Y & specify signal | N)
Abort (Y & specify Granularity (F2 | F3) & specify mechanism: signal | encoded msg) | N)
AbortResponseAction(proceed to transmit partial data | discard partial data)
Uses Channels (Y & specify max number | N)
Uses Parity (Y & specify width | N)

Figure 3.7: Example attributes for the Stream interface type

```
/* IDL definition for Stream */
Stream::$Definition1 [Set of experimental behavioral attributes]::Attributes(
 Format[Details of the formats and organization of control information] (
  Words[Format of words] (
   *size:(int|intlist|expr) with units ("bits":1 | "bytes":8)
          with range(0:128),
   byte_order:choice("Big Endian"|"Little Endian"),
   bit_order:choice("high to low"|"low to high"),
   *indicate_completion:choice(
            rem(sz[size of the encoded value]:int,
                signal_valid:choice("active high"|"active low")) |
                byte_enable(sz:int) | length(sz:int)
                              )
   ),
  Chunks[Format of chunks]  (
   *required:bool,
   *supported:bool,
   size:(int|intlist|expr) with units("bits":1|"bytes":8),
   indicate_start:choice("marker" |
                         "assertion of ready signal" |
                         "ready plus a status ctrl word" |
                         "determined by time slice" |
                         "other encoding"
                         ),
   indicate_completion:choice("marker" |
                              "assertion of ready signal" |
                              "ready plus a status ctrl word" |
                              "determined by time slice" |
                              "other encoding"
                              )
   ),
```

Figure 3.8: IDL stream type interface attributes example

```
Element TestElement {

/* Secondary (input) port*/
Input in_port Stream :: $ localLink :: Attributes(

 Format (
  Words ["intrawords"]  (
   size:1 bytes,
   byte_order:"Big Endian",
   bit_order:"high to low",
   indicate_completion:rem/*help?*/(sz:4, signal_valid:"active high")),
  Chunks["this doesnt support chunks"]  (
   required:false,
   supported:false
   ),
  Packets["format of streams"]  (
   size:1500 bytes,
   indicate_start:"assertion of ready signal",
   indicate_completion:"assertion of ready signal"
   )
  ),
```

Figure 3.9: EDL stream type port attributes example

3.3.2 Metadata representation and packaging

The ShapeUp prototype implementation used custom XML representations for both EDL and IDL descriptions, to allow easy experimentation. For the future, EDL is being aligned with the IP-XACT standard for expressing module interface metadata. IP-XACT at present is very bus-centric in terms of inter-module connections, and so a certain amount of artificiality, and some use of 'vendor-specific extensions' is necessary to map the more general model of ShapeUp connections onto the IP-XACT data schema. A small sample of the XML version of the EDL is shown in Figure 3.10.

The interface metadata is bundled with each element to support table-driven tools that assist the designer with making connections between modules. Figure 3.11 shows the flow for adding a new element to the ShapeUp element library. The module source files, e.g. RTL descriptions, are packaged along with the interface metadata and then stored in the element library. The metadata for the interfaces is specified either by the designer of the module or auto-generated by a high-level language compiler. The ShapeUp suite of tools described in the next section uses this element library and metadata for raising the level of abstraction.

```
<interfaces>
    <Stream direction="input" technology="LocalLink" name="streamin">
      <data maximumLength="1024" minimumLength="32" width="32"/>
      <speed units="Gbps" value="20"/>
    </Stream>
    <Stream direction="output" technology="LocalLink" name="streamout">
      <data maximumLength="1024" minimumLength="32" width="32"/>
      <speed units="Gbps" value="20"/>
    </Stream>
    <Access direction="input" technology="fifo" name="stats"
     writeable="true">
      <data width="16"/>
      <speed units="MHz" value="133"/>
    </Access>
    <Access direction="input" technology="register" name="ctrl"
     readable="true">
      <data width="32"/>
      <speed units="MHz" value="133"/>
    </Access>
  </interfaces>
```

Figure 3.10: XML EDL example with two stream ports and two access ports

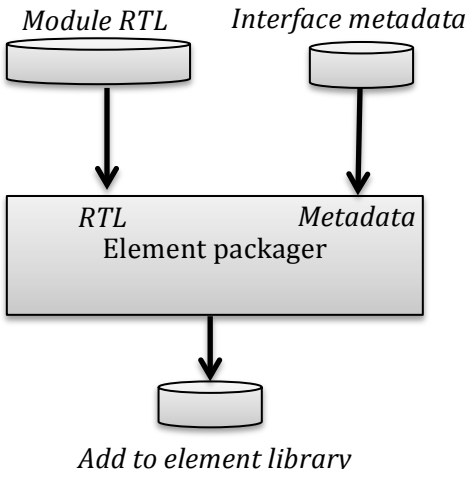

Figure 3.11: Flow for adding a new element to the ShapeUp element library

3.4 Type Checker

Type checking is central to the ShapeUp framework. A fundamental requirement is to check whether two element ports match, and so determine whether it is possible to make a connection between them. This involves testing whether the two ports have the same type, and then that they have matching attributes for that type. The approach is in the same spirit as earlier research of Bergamaschi et al. [40] in checking pin compatibility, but tackles a much more general interface checking problem. The main goal is that by characterizing and capturing the signaling behaviors of these interfaces, system designers will no longer need to check the detailed behavior, and some tool can usefully automate the process and indicate compatibility. An algorithm was developed to carry out this type checking operation. Since ShapeUp uses a data-driven model for the interface types, whereby their characteristics are described using meta-metadata, some details of how both IDL and EDL descriptions are processed are relevant to understanding the type checking algorithm itself.

Interface type attributes are organized hierarchically: at the top level is the type, and then there are main groups of attributes, then sub-groups, etc. The particular structure is defined in the IDL description of each type. Particular attributes have data types, which can include ranges of allowed values or enumerated values. Each attribute is flagged as whether or not it is compulsory. If not, then a default value can be specified in the IDL. All of these aspects are taken account of by the type checker.

The internal data structure used for storing an IDL description is shown in Figure 3.12. If the IDL file parses correctly, the main feature is the interface list, which contains the defined interfaces. For the standard case, there are five on the list: plain, notify, stream, access, and compute. For each interface, its name is stored (e.g. "plain"), together with an optional subtype name to allow derivative interface profiles to be defined (e.g. "stream" "LocalLink"), and an optional description string for documentation. Then there is an attribute list for the interface. Each attribute has a name and an optional description string. As mentioned, there is an indication of whether the attribute is compulsory, and there is also an indication of its sort. One sort is that the attribute is a node in a hierarchy, and then a list of sub-attribute children is stored. The other sort is that the attribute is a leaf, and then a list of type choices for the attribute value is stored. For each such type, information is stored on any characteristics or restrictions on values of that type. Figure 3.13 shows an abstracted example representing the IDL as a tree of attributes.

The internal data structure used for storing an EDL description is shown in Figure 3.14. When a description is processed, it is first parsed to check syntax, and then parsed against the stored IDL description to check interface types. The main feature of the data structure is the port list, which contains the defined ports of the element. For each port, its name and direction are stored, together with its type name and

optional subtype name, and an optional description. Then there is an attribute list for the port.

IDL data structure
- File name (string)
- Parse error count (integer)
- Interfaces (**Interface** list)

Interface data structure
- Type name (string)
- Subtype name (string)
- Description (string)
- Attributes (**Attribute** list)

Attribute data structure
- Name (string)
- Description (string)
- Compulsory (boolean)
- Sort (ATTRIB, LIST)
- *Union of sort characteristics, as relevant:*
  - Node sub-attributes (**Attribute** list)
  - Leaf type choices (**Type** list)

Type data structure
- Type (BOOL, INT, STRING, CHOICE_STRING, CHOICE_ATTRIB)
- *Union of type characteristics, as relevant:*
  - Integer units and allowed range (**Unit** list, integer, integer)
  - Choice strings (string list)
  - Choice attributes (**Attribute** list)

Unit data structure
- Name (string)
- Multiplicative factor (integer)

Figure 3.12: Internal data structure for storing IDL description

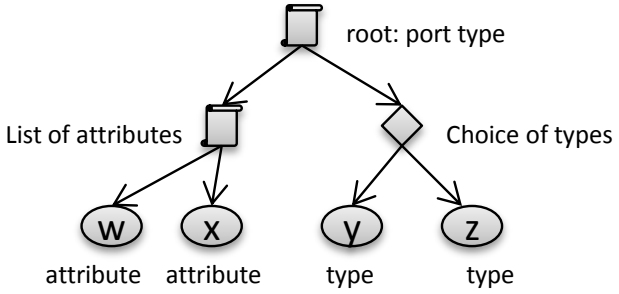

Figure 3.13: Example of IDL attribute tree

Two attribute data structures are shown: one used before the parsing against the IDL, and the other used for the final EDL representation. The port type name and subtype name are used to select the appropriate IDL interface type and subtype to check against. The checking involves ensuring that all compulsory attributes are present, no unknown attributes are present, and the attribute hierarchy matches structurally. Each leaf attribute is checked to ensure it has a legal type and a legal value for that type. The final EDL data structure indicates whether each leaf attribute has an unassigned, wildcard, or assigned value.

The port type checking algorithm involves comparing two stored EDL port data structures. Pseudo-code for the algorithm is given in Figure 3.15. The two ports are first checked to ensure they have the same type and the same subtype if it is used. Then the attribute lists of the two ports are compared, attribute by attribute, to ensure that they match. The comparison of two attributes involves first checking that the attribute names match, and then that the attribute values are consistent. If either attribute has a wildcard value, or both attributes have unassigned values, then they are deemed to match. Note that, as discussed earlier, more elaborate approximate matching tests can be included here, depending on the nature of the attributes. If it is a leaf attribute, the two values are compared directly if they are scalar or are compared recursively if they are themselves attributes. If it is a node attribute, the next level of the hierarchy is compared recursively.

EDL data structure

- File name (string)
- Parse error count (integer)
- Element name (string)
- Ports (**Port** list)

Port data structure

- Port name (string)
- Direction (string)
- Type name (string)
- Subtype name (string)
- Description (string)
- Attributes (**Attribute** list)

Attribute data structure 1 – before parsing against IDL

- Name (string)
- Description (string)
- Type (UNASSIGNED, WILDCARD, INT, STRING, ATTRIB, ATTRIB_LIST)
- *Union of values, as relevant:*
  - Integer value and units (integer)
  - String value (string)
  - Attribute value (**Attribute**)
  - Attribute list value (**Attribute** list)

Attribute data structure 2 – after parsing against IDL

- Name (string)
- Description (string)
- Type (BOOL, INT, STRING, ATTRIB, ATTRIB_LIST)
- Value sort (UNASSIGNED, WILDCARD, ASSIGNED)
- *Union of values, as relevant:*
  - Boolean value (boolean)
  - Integer value (integer)
  - String value (string)
  - Attribute value (**Attribute**)
  - Attribute list value (**Attribute** list)

Figure 3.14: Internal data structure for storing EDL description

```
boolean compare_ports (port1, port2)
{
   return false if type1 != type2 || subtype1 != subtype2;
   return compare_attribute_lists (attributes1, attributes2);
}

boolean compare_attribute_lists (list1, list2)
{
   for (a1=first1,a2=first2; a1 != last1 && a2 != last2; a1++,a2++)
      return false if ! compare_attribues (a1, a2);
   return (a1 == last1 && a2 == last2);
}

boolean compare_attributes (attribute1, attribute2)
{
   return false if name1 != name2;
   return true if sort1 == wildcard || sort2 == wildcard;
   return true if sort1 == unassigned && sort2 == unassigned;
   return false if sort1 == unassigned || sort2 == unassigned;
   switch (type) {
      bool:        return (value1 == value2);
      int:         return (value1 == value2);
      string:      return (value1 == value2);
      attrib:      return compare_attributes (value1, value2);
      attrib_list: return compare_attribute_list (value1, value2);
   }
}
```

Figure 3.15: Pseudo-code for type checking of two ports

Note that the algorithm used to parse an EDL port description against a stored IDL interface description has a very similar organization to this type checking algorithm. It also involves the recursive traversal of the attribute hierarchy tree, but with more complicated operations carried out at each stage: both checking and matching, and also building the final attribute data structures.

The original IDL and EDL parsing algorithms, and the port type checking algorithms, were implemented in Java. Later, when transferred into product development at Xilinx, they were reimplemented in C++.

A simple type checking example is presented in Figure 3.16. Here, the connections almost match, but are not an exact match. The primary stream port, (a), and the secondary stream port, shown in (b) differ in both their width and size of rem, and so do not exactly match. However, they approximately match if tools are available to bridge the differences. Automatically generating shims with bridging logic is discussed in Section 3.8.2. As well as giving a true or false result, the type checker can also supply a list of things that did not match exactly.

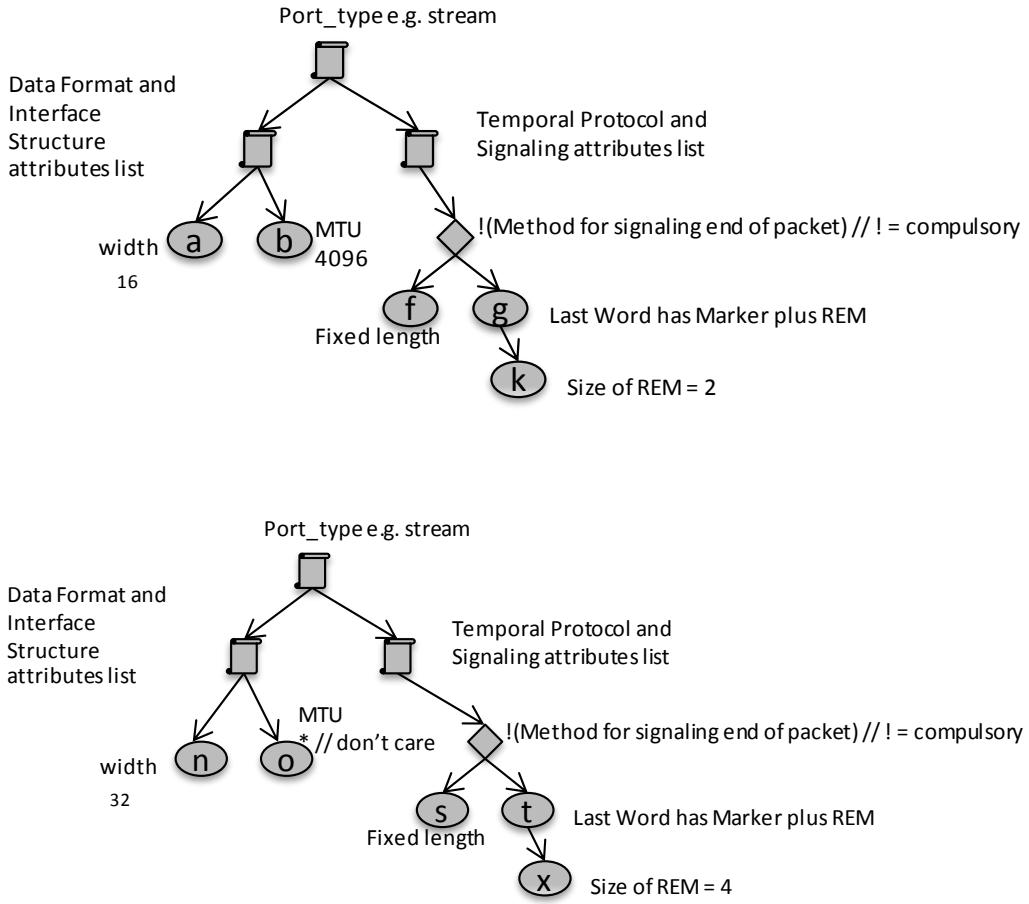

Figure 3.16: Type checking example: (a) primary port; (b) secondary port

Figure 3.17 shows a more in-depth example using example EDL attributes showing compatible interfaces. This example contains a comparison of two implementations of LocalLink interfaces, which are represented as stream ports. The interface metadata for this example is arranged in columns. Green highlighting indicates compatible behavioral attributes. Yellow highlighting indicates attributes that are similar and can be bridged in order to be compatible. The Xilinx XAPP536 ll_temamac_v1_00_c, or ll_temac, is connected to an XAPP691 LocalLink FIFO, or ll_fifo. There are three differences in the EDL attributes that can be bridged in order to make the interfaces compatible. First, they signal the rem, which indicates how many bytes are valid in the last word of the packet, differently. The ll_temac signals rem using a four bit encoding (byte enable encoding) and the ll_fifo uses a two bit encoding (binary encoding). Second, the ll_temac has a minimum size to the frames that it transmits and the ll_fifo has a maximum size to the frames that it accepts. The bridging logic needs to check that the frames from the ll_temac do not exceed the maximum size frames to the ll_fifo. Third, the ll_temac uses a data offset, and the ll_fifo does not, however, this information can be passed through.

Figure 3.18 shows a comparison of two implementations of LocalLink ports, arranged in columns, having incompatible behavioral attributes. Again, the yellow highlighting indicates attributes with similar behavior that can be bridged. The red highlighting indicates attributes that are incompatible and cannot be bridged. The LocalLink GMAC, or ll_gmac, is connected to a LocalLink IPOptionizer, or ipoptionizer. There are four sets of attributes that are similar and can be bridged, and there are two sets that are incompatible and cannot be bridged. The first incompatible attribute is that the ll_gmac does not support chunks, and the IPOptionizer requires segmentation support. The second related incompatible attribute is that the IPOptionizer requires the use of channel identifiers, and the ll_gmac does not have this support, and so the compiler cannot automatically bridge these two interfaces. Therefore, the two versions of these modules are incompatible and cannot be automatically used together, without the user manually creating a custom wrapper to reconcile the differences and make them compatible.

XAPP536 ll_temac_v1_00_c	XAPP691 FIFO
Primary	Secondary
Format	Format
F1 (Word)	F1 (Word)
1. 32 bits	1. 8, 16, 32, 64, 128 bits
2. a) Big Endian b) 31:0	2. a) N/A b) N/A
c) rem (4 bits encoded value, active high)	c) rem (2 bits encoded value, active high)
F2 (Chunk)	F2 (Chunk)
0. Y	0. Doesn't care
1. min(9, C_RX_FIFO_KBYTE) # in k bytes	
2. a) marker(signal, ll_sop_n)	
b) marker(signal, ll_eop_n)	
F3 (Packet)	F3 (Packet)
1. min(9, C_RX_FIFO_KBYTE) # in k bytes	1. max(BRAM_MACRO_NUM * F1.width)
2. a) marker (signal, ll_sof_n)	2. a) marker (signal, sof_in_n)
b) marker (signal, ll_eof_n)	b) maker (signal, eof_in_n)
Behavior	Behavior
F1 Data Enable = Y, (based on rem, only valid with ll_eop_n)	F1 Data Enable = Y, (based on rem, only valid with eof_in_n)
F2 Data Enable = Y, ll_src_rdy_n	F2 Data Enable = Y, src_rdy_in_n
F2 Flow_Control = Y, ll_dst_rdy_n	F2 Flow_Control = Y, dst_rdy_in_n
F3 Data Enable = Y, ll_src_rdy_n	F3 Data Enable = Y, src_rdy_in_n
F3 Flow_Control = Y, ll_dst_rdy_n	F3 Flow_Control = Y, dst_rdy_in_n
Abort(N)	Abort(N)
Uses Channels(N)	Uses Channels(N)
Uses Parity(N)	Uses Parity(N)
Uses Data Offset(Y for F2, protocol specific)	Uses Data Offset(N)

Figure 3.17: Example 1: compatible interfaces

LocalLink GMAC	IPOptionizer
Primary	Secondary
Format	Format
F1 (Word)	F1 (Word)
1. 8 bits	1. 128 bits
2. a) Big Endian b) 7:0	2. a) Big Endian b) 127:0
c) rem (1 bit encoded value, active high)	c) length
F2 (Chunk)	F2 (Chunk)
0. N	0. Y, requires 1. 8 words 2. a) ready plus a status/ctrl word b) ready plus a status/ctrl word
F3 (Packet)	F3 (Packet)
1. 1500 bytes 2. a) marker (signal, ll_sof_n) b) marker (signal, ll_eof_n)	1. Not specified 2. a) ready plus a status/ctrl word b) ready plus a status/ctrl word
Behavior	Behavior
F1 Data Enable = Y, (based on rem, only valid with ll_eof_n) F2 Data Enable = Y, ll_src_rdy_n	F1 Data Enable = N F2 Data Enable = N
F2 Flow_Control = Y, ll_dst_rdy_n	F2 Flow_Control = Y, in0_backpressure or in0_status_backpressure
F3 Data Enable = Y, ll_src_rdy_n	F3 Data Enable = N
F3 Flow_Control = Y, ll_dst_rdy_n	F3 Flow_Control = Y, in0_backpressure or in0_status_backpressure
Abort(Y, ll_src_dsc_n), Response(discard, sender retransmit)	Abort(Y, encoded in status/ctl word), Response(discard, sender retransmit)
Uses Channels(N)	Uses Channels(Y requires, 6)
Uses Parity(N)	Uses Parity(N)
Uses Data Offset(N)	Uses Data Offset(N)

Figure 3.18: Example 2: incompatible interfaces

This work on type checking was patented as part of US Patent #7,852,117: "Hierarchical Interface for IC system" [106].

3.5 ShapeUp Design Tools

In ShapeUp, systems are specified using a (possibly hierarchical) Click description of the component modules and their interconnections. Then, module interface metadata based on the defined behavioral abstractions can be used by a variety of design tools that aid in system implementation and testing. Three initial ShapeUp tools are described here: a design entry tool and visualizer, a linker, and a validator. The tool flow is shown in Figure 3.19. Each of the three ShapeUp tools makes use of the type checker, which is abbreviated as "TC" in the figure.

Section 3.6 will describe a novel Click entry environment and visualizer, another key topic in this chapter, which uses the type checking in order to suggest possible connections. The visualizer shows connection possibilities and which ports have already been connected. The interface for this design environment was patented as US Patent #8,121,826: "Graphical interface for system design" [107].

Section 3.7 will describe two additional tools for producing the output structural description and for performing system-level validation. The validator runs a system level simulation, using a simple protocol to pass data between multiple module level simulators. The linker is used to create a top level RTL structural description of the design.

The resulting system RTL design is fed as input to the standard FPGA tool flow, discussed in Section 2.1.3, consisting of synthesis (e.g. XST), mapping the synthesized netlist onto FPGA primitives (e.g. MAP), and placement and routing (e.g. PAR). Lastly, the Xilinx ISE tools produce the final FPGA configuration bitstream, which is used to program the FPGA device.

These prototype tools have been used together practically on some real FPGA-based product designs in the telecommunications industry, forming the system design level of an experimental packet processing design tool suite developed by Xilinx Research Labs [86]. The point tools were embedded in an Eclipse-based Integrated Development Environment (IDE).

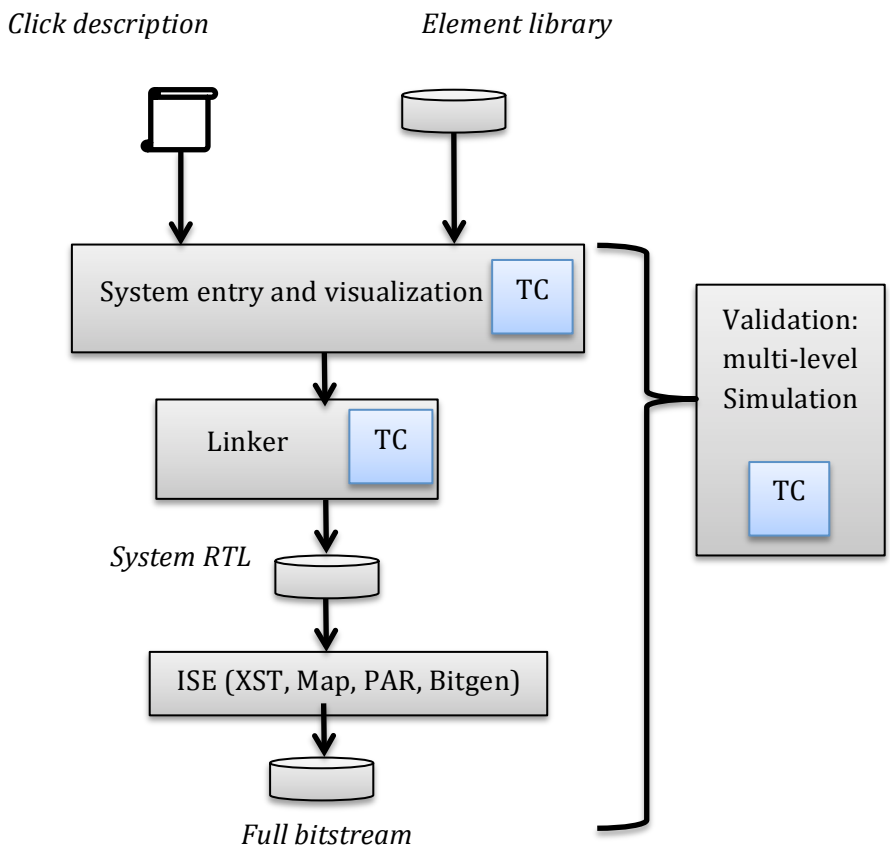

Figure 3.19: ShapeUp tool flow diagram

3.6 Design Entry Environment and Visualizer

The entry environment and visualizer, called Pop, is an interactive tool that provides visualization of emergent systems as the user enters their Click description. It is somewhat the reverse of a traditional schematic editor, which would have graphical input and textual output. Rather, it has textual Click input and graphical visualization output. This is because a Click description is often most conveniently entered in a textual form. However, since Click is a declarative language, i.e. declarations and connections can be written in any order, an ongoing visualization aids the user in ensuring that all connections are entered and that only valid connections are made. Because of this, the visualizer is connection-centric rather than module-centric in terms of its operation.

Figure 3.20 shows a screenshot from the visualizer. In the lower part of the screen is the textual Click description being entered by the user. The upper part of the screen shows the visualization of the system in progress. Individual element ports, and connections between them, are shown separately. That is, the ports of a particular element may be distributed over the visualization, rather than clumped together in a display of that element. This is the key connection-centric feature of the visualizer.

Ports are represented by shapes that correspond to the different interface types (plain, notify, stream, access, and compute). Each element is assigned a different color, and this is used for all the port shapes for that element, which allows the user to see each element in its distributed form. Connections in the Click description are shown as arrows between the port shapes. Real-time type checking is carried out during Click entry, to check that only valid connections are made between elements. Beneath this display, currently unconnected ports for declared elements are shown, as an aide memoire to the user.

The visualizer deploys various heuristics to determine the placement of port shapes and arrows on the screen, and these were evolved after experimentation with test users. For example, for communication-style systems dominated by stream type

ports, left-to-right placement of connections on a single line is used to highlight streaming dataflow. This can be seen in the first line of the display in Figure 3.20.

The concise syntax of Click makes it easy to read, however, the concise nature of port identifiers can potentially lead to connecting mismatched ports. The user must be familiar enough with components to specify the names of ports and match their types against that of other ports when forming connections. This is unsurprisingly more difficult than working with the original Click, which had integer-only identifiers for ports and only packet interfaces. Exposing the additional port types that were once hidden, due to cross-element method calls not shown in the Click graph, can lead to the user having to specify more connections in their design overall, but at the same time they also need to keep track of what is unconnected. The first problem is that the Click programmer writes statements to form connections, however, the ports that are not connected do not appear in the description—it is up to the programmer to keep track of them. The second problem is that Click's independence of connection statements allows connections to be arbitrarily arranged within the description, and their arrangement impacts the readability of the code. The third problem, which is related to the first, is that in both complex designs and even simple ones, unless the programmer is familiar with each of the elements, to the point that they can keep track of all the ports, it is possible to forget to specify one (or more) connections without noticing this from their Click description.

Pop automatically solves each of these problems by providing real time assistance, as the user types in their description. The compiler reads the metadata for ports and assists the user when forming connections. Pop identifies possibilities for forming connections by suggesting names according to the interface type. Pop also checks the interface types across connections to make sure that detailed attributes of the interfaces are correctly matched. Pop graphically shows what has been connected

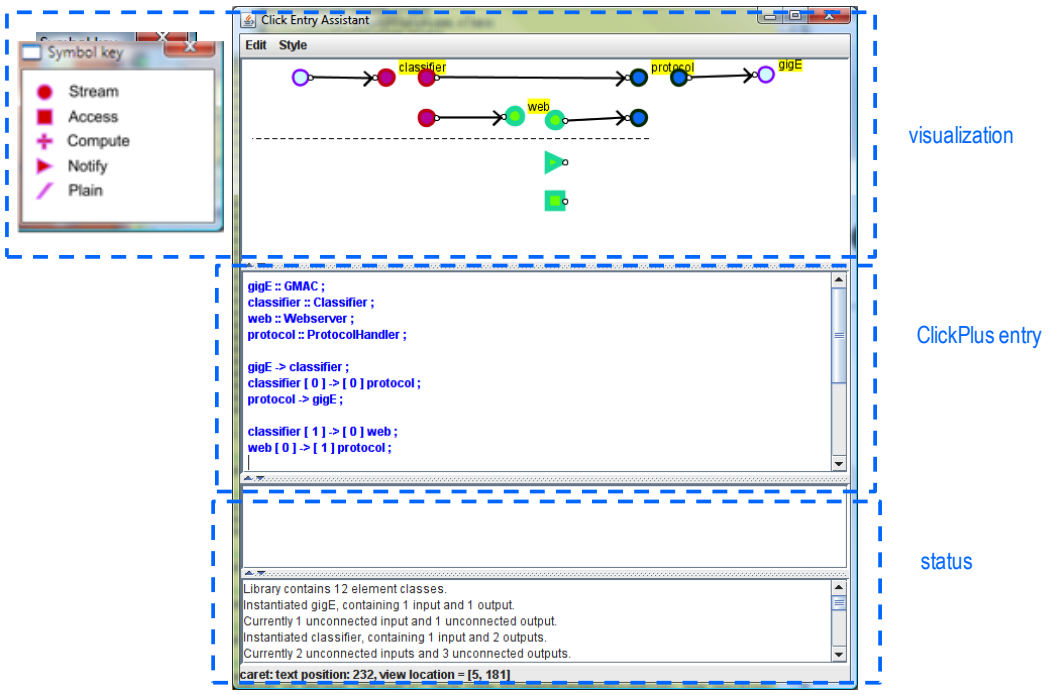

Figure 3.20: Real-time visualization of Click design entry

and what remains to be connected, and it shows this information arranged by port type. This is intended to help the user gauge the level of completeness of their description. Pop also performs auto-completion of port-name and element-name statements.

The most novel feature of Pop is the kind of visualization that it provides. The visualization format was developed from examining traditional block diagrams for some example systems and also the textual Click descriptions for the same systems. In a block diagram view like the one shown in the top half of Figure 3.21, boxes are color-coded and represent elements. Ports are drawn on the sides of the boxes. Depending on convention, the input ports are indicated by symbols (or labels) on the left and the outputs are similarly shown on the right. The boxes importantly provide association between the ports from the same element as well as giving an abstract representation of some embedded functionality.

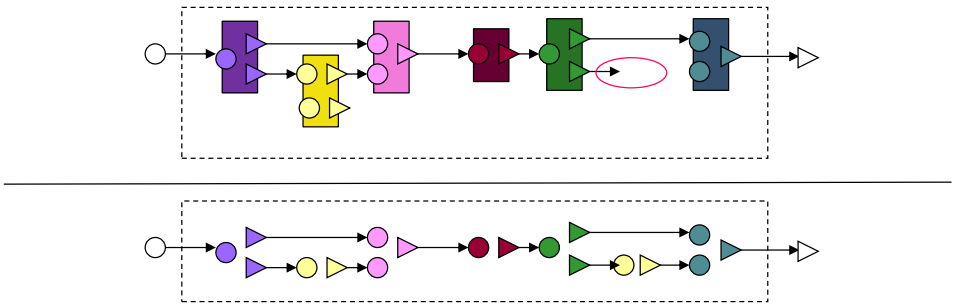

Figure 3.21: Top half shows block diagram view of system; bottom half shows directed graph view of the same system with boxes removed and second connection formed to the yellow element

Two properties of Click that are relevant to mention here are: (a) connections are just between ports on elements and (b) connection statements are independent. Because of the independence of statements in Click, the ordering of connections is not necessarily based on the adjacency of ports. Some elements may even be artificial in terms of their dataflow, depending on how they are used within a system.

For example, the yellow element in the top half of Figure 3.21 is shown connected between the purple and pink elements. Its remaining ports would be connected in the area circled in red, between the green and blue elements. However, if these two new connections were to be drawn, a very cluttered layout would result. In fact, because the yellow element features at both the beginning and end of the main pipeline, there is no ideal place to position it. The key insight embodied into Pop was that the ports and connections matter more to the user than the actual elements. In particular, the user needs to be able to easily determine what ports remain to be connected in the design. Therefore, the visualization style shown in the lower half of Figure 3.21 was adopted instead. Here, ports have independent existences from their elements. However, all ports of the same element have the same color to allow easy identification. Where it is convenient to collocate the all the ports of an element, it is done, as can be seen for most of the elements in the figure. But, the yellow element is now seen in a distributed visualization: two ports are to the left, and two ports are to the right. The overall effect is to focus on the system dataflow.

Experiments were conducted to see what effect color selection had to the readability of the diagrams, when color is used as a primary means to associate ports by their parent element. Figure 3.22(a,b) show some of the pseudo-random test patterns that were generated. Figure 3.22(a) shows a large test pattern to check readability of randomly generated colors for port symbols. Figure 3.22(b) shows a smaller test pattern to see if reducing number of ports improved readability. The test patterns are considered pseudo-random because a hash-based function was used to ensure that colors were somewhat dissimilar, in that they did not produce the same hash value as any of the other randomly selected colors, when comparing respective luminance and chrominance values. To help make elements easier to identify, a second color is assigned to the symbol outline and also the thickness of the outline is varied.

In Figure 3.22(a,b,d), triangles represent output ports and circles represent input ports, which was done only for early stages of testing color choice. Later, specific symbol shapes were selected to represent the port types instead of direction, as seen in Figure 3.22(c). This image shows the symbols that were assigned to each of the five interface types in our experiments. To keep things simple, the same shape is used for both inputs and outputs. The symbol choice like the plus symbol for compute, for example, helps to make the description somewhat easier to read through symbol association. However, color alone might not be sufficient to associate ports that belong to the same element, particularly if the programmer is colorblind or has difficulty with matching colors. As a result, an option to label the symbols with the element name and port name makes association easier for the programmer.

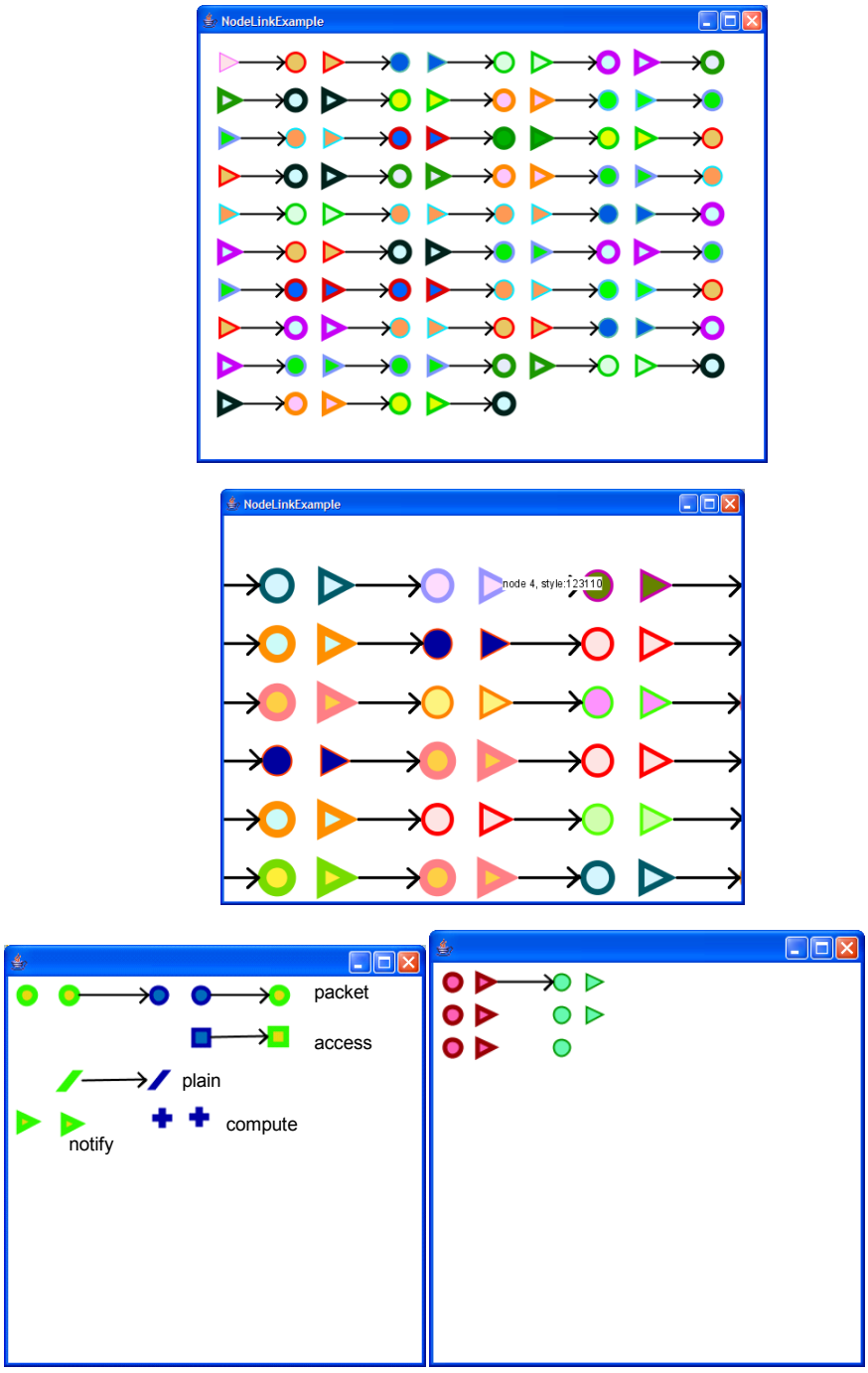

Figure 3.22: Visualization color experiments and symbol choices: Counter-clockwise from upper-left: (a) large test pattern; (b) small test pattern; (c) chosen port symbols; (d) column layout for ports

The port symbols are drawn in two columns, at the time the programmer types the instantiation of an element, as shown in Figure 3.22(d). The ports per element are arranged with inputs to the left and outputs to the right.

The visualization pane in Figure 3.23 shows both connected and unconnected ports to help user gauge progress while entering the design. This indicates the overall completeness of the description. A dotted horizontal line separates the ports that have been connected in the upper portion from the ports that have not yet been connected in the lower portion. This way, the programmer can quickly see what remains by scanning below the line. Checking a schematic diagram for an unconnected port requires a scan over the diagram, whereas in this visualization there is a partition to directly show what is not connected. In this view, the relative number of unconnected ports can be seen right away. A small circle is added to each symbol to the left or right indicate whether it is an input or output port. The set of ports for a single element are drawn in columns going left to right, as they are instantiated.

Each row above the line corresponds to a line of Click that the user typed. At the time the user forms a connection, port symbols are moved down into new rows if they are not used in the connection, so that anything unconnected rests below the horizontal line.

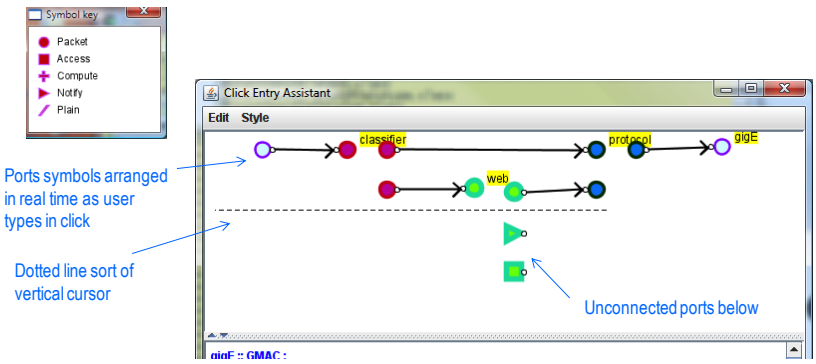

Figure 3.23: Visualization pane shows both connected and unconnected ports to help user gauge progress

Arrows are drawn between port symbols used in the connection and an animation takes place to rearrange the drawing, so that the order of ports and connections matches the textual Click description. The Piccolo Zoomable User Interface (ZUI) construction kit [108], developed by the University of Maryland, was used in the implementation of the visualization pane to draw and animate the symbols.

The Click entry pane, shown in Figure 3.24, is the text editor area where the user types in their description. The entry pane also helps to suggest the names of ports, with a pop-up context menu, as the user is typing their connection. When the user selects a port from the context menu, the text in the description is automatically generated and inserted at the text cursor position. Only unconnected ports are suggested, and when the user is prompted for the input port on the right hand side of an arrow it will suggest only ports of the appropriate type.

The status pane, shown in Figure 3.25, provides an activity log that displays an incremental record of how the user has constructed the system model. The activity log has statements for each action; with the number of unconnected ports remaining in the design after each action is performed. This is another place to indicate if there are errors detected in the design with an error or warning message. This is also a place to print summary messages about the library of elements and the also to provide any resource estimates for the design.

To summarize this section, the Pop development environment is different from existing approaches in that the Pop environment interactively draws and checks the system as the user types in their description. Pop addresses a difficulty introduced by the extension to Click. The real-time, line-by-line visualization is made possible because Click is a declarative language. The fact that descriptions can be conveniently checked on a line-by-line basis helps to reinforce our view that Click is an appropriate language to use for the stitching task of designing modular

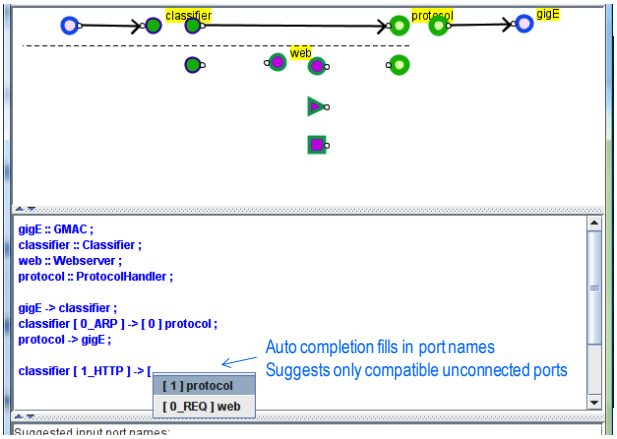

Figure 3.24: Click entry pane helpfully prompts as the user types in their description

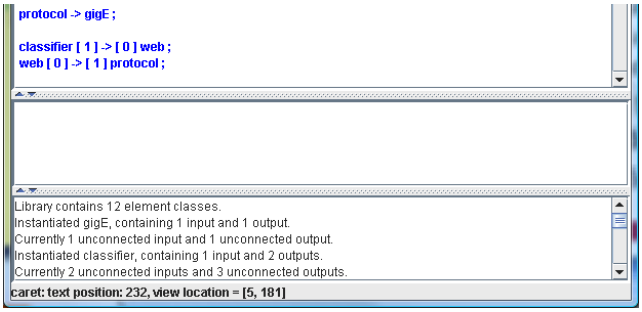

Figure 3.25: Status pane provides a textual description of the actions performed to the system model and other status

architectures. The graphical enhancement of textual focus provided by Pop conveys useful information about progress and the remaining ports. The visualization, overall, very closely resembles graphical Click syntax, and the port symbols provide symbolic help to interpret the ShapeUp design.

3.7 Additional ShapeUp Tools

The following two tools were implemented by others, fitting within the overall ShapeUp framework. Both of these tools incorporated the ShapeUp type checker.

3.7.1 ShapeUp validator

The validator is the main testing tool for systems built from component modules. It encompasses multi-level (including mixed-level) simulation and on-FPGA operation. This allows validation prior to any FPGA implementation steps, after ShapeUp linking, after synthesis, after place-and-route, or running on a real FPGA.
The ShapeUp validator takes a Click description of the target system, and the data and metadata for the constituent modules. It then has a variety of options, depending on the validation level required.

The highest level of validation is prior to any explicit assembly of the system from its constituent modules. A distributed simulation framework corresponding to the Click interconnection graph is constructed. Each node in the framework is responsible for the simulation of one module, and these nodes communicate to simulate interactions between modules. Standard Unix TCP/IP sockets are used as the communication mechanism. A master process is responsible for creating the simulation nodes, and then making TCP/IP connections between them. The use of TCP/IP means that nodes need not all be running on the same computer, allowing genuine parallel simulation.

The simulation at each node requires some model for the module. This might be simple, for example a Perl script, or more accurate, for example a SystemC model or the actual RTL. The validator uses the module interface metadata and module model metadata to ensure accurate emulation of the interactions between modules.

Figure 3.26 shows how the validator can generate a framework that allows validation across different implementation levels, in this case for a system with a streaming packet input and a streaming packet output. The same packet data source is used for each level, and the same format of output packet data is produced for each level. This allows automated comparison of results between the levels to ensure correctness of implementation steps. There are three levels of RTL simulation using ModelSim, corresponding to HDL generated by the ShapeUp linker, by the synthesized netlist

from this HDL, and by the placed-and-routed layout from the netlist. Finally, there is 'hardware in the loop' operation, where the system is exercised on the target FPGA with real input and output.

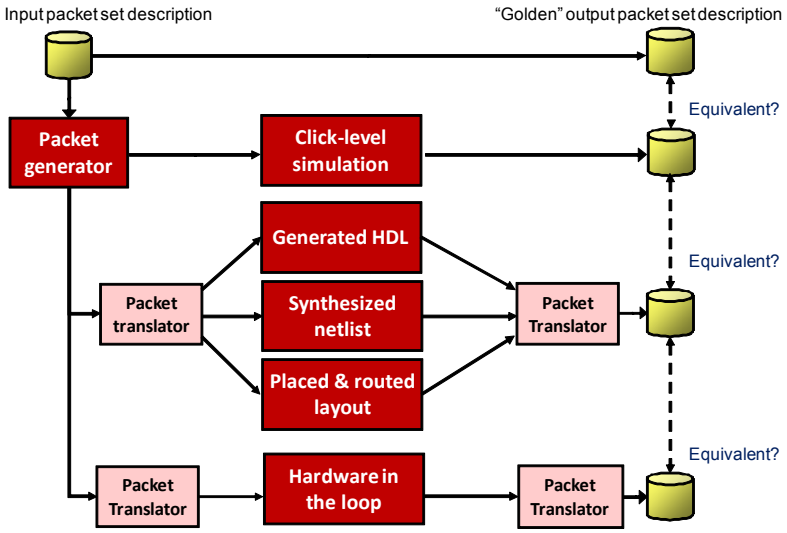

Figure 3.26: Multi-level validation environment for streaming systems

3.7.2 ShapeUp linker

The linker is the main implementation tool for system assembly from component modules. The term 'linker' is drawn from the analogous software design flow, envisaging that compilers are used to generate the modules, and then the linker is used to connect them together into a whole.

The ShapeUp linker takes a Click description of the target system, and the data and metadata for the constituent modules, and generates the structural RTL design (in Verilog or VHDL) of the complete description. Standard synthesis and place-and-route tools can then process the design.

The main action of the linker is to type-check the Click description and then to create wiring that implements the required connections between the hardware modules. Type checking is as described in Section 3.4. The linker can also generate wiring for specific module requirements, such as clocks and resets. Finally, it can ensure that wiring is consistent with board-level constraints on FPGA input/output pin placement. All of this is guided by module interface metadata, other module metadata, and board-level metadata that is supplied to the linker.

The basic benefit of this wiring activity is to relieve the user of the tedious and sometimes intricate task of gathering all necessary module interface information and then writing HDL code to connect interface pins together. It also simplifies maintainability and evolvability, by allowing simple changes to be made at a high level without the need to rewrite and recheck low-level HDL code.

A significant additional function of the linker is 'auto-bridging'. When the type checker indicates that two interfaces approximately match, rather than strictly match, the linker is able to bridge certain differences by inserting one or more additional conversion blocks between two modules. This is illustrated in Figure 3.27. Here, one module has a LocalLink packet interface with a 32-bit data path that is to be connected to another module with a LocalLink interface with a 128-bit data path. The linker inserts a block that accumulates four successive 32-bit words and then forwards them as a single 128-bit word. If the first module is not clocked at four times the rate of the second module, then the block must also assert LocalLink flow control as appropriate.

In the current prototype version of the linker, there is a pre-canned repository of available parameterized conversion blocks and metadata for them. A future aspiration is to support the automated generation of conversion blocks based on the exact needs identified by the attribute mismatch(es) between the connected module interfaces. One approach is to build upon earlier work of Passerone et al. on automated synthesis of interfaces between incompatible protocols [109].

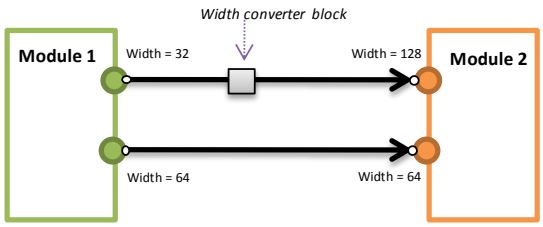

Figure 3.27: Insertion of width converter block between two modules

An aspect of improving the overall modular design experience is to import the ShapeUp interface abstraction into the tools that are used to generate modules themselves. This has been done for the experimental G packet processing language [86], so that its typing of input and output ports matches the ShapeUp typing. The impact is to produce modules that are 'ShapeUp ready', and thence pose fewer bridging problems for the linker and validator tools to overcome. Chapter 5 will discuss an integrated example, with programming in G and Click, and founded on ShapeUp.

3.8 Summary

Chapter 3 presented the basic approach using a modular abstraction called *ShapeUp*. A set of interface abstractions and a modular design methodology was described based on abstractions of module interface behavior, from three programming paradigms. This research is novel in that there has been significant past work on abstracting behavior of module functions, but little on the abstraction of the interconnection of modules. ShapeUp addressed this by abstracting the behavior of the interfaces and connections between the interfaces. Several tools were developed that use general data driven mechanisms.

The main contributions of the ShapeUp work described in this chapter are the following:

- Section 3.2 presents a set of abstractions of module interface behavior, featuring five types of interface that cover both streaming and procedural programming paradigms for modules.
- Section 3.3 presents interface metadata (and meta-metadata, in fact) to describe a module's interfaces in terms of the defined abstractions, enabling the creation of module repositories.
- Section 3.4 presents the type checker that is used by the other tools to indicate the compatibility of two ports when forming a connection.
- Section 3.5 provides an overview of the ShapeUp tool flow.
- Section 3.6 presents the Pop Click entry and visualization environment
- Section 3.7 presents additional tools that were enabled by the (extended semantics) Click descriptions and module metadata and completed the high-level modular design experience.

Chapter 4 describes a practical addition to the ShapeUp library, modules for performing timing functions. These modules enable time-triggered behavior important to many networking systems. Chapter 5 describes the validation of the 'plug-and-play IP' productivity gains from use of the ShapeUp methodology and the prototype tools on a real-life industrial-strength case study involving building real high performance networking systems.

Chapter 4

Flexible and Modular Support for Timing Functions in High-performance Systems

Field programmable logic is increasingly used to provide the high performance and flexible acceleration needed for network processing functions at multi-gigabit rates. Almost all such functions feature the use of clocks and timers in control and/or data roles, and these are typically implemented in an ad hoc manner. This chapter introduces a set of three configurable timing modules that are based on abstractions of the prevalent timing paradigms observed in network protocols. The modules fit within the experimental ShapeUp methodology for modular FPGA-based system design, and so can be easily integrated with other modules that are tailored for specific networking functions. The use and benefits of the new modular approach are demonstrated in Chapter 5 by an example of a flexible FPGA reference design that has been made available for real-life use by telecommunication equipment providers.

A characteristic of numerous computing and networking functions is the use of clocks and timers. A broad survey was conducted showing that time is used extensively in computing, for example: to schedule processing to start or to meet deadlines; to schedule the sharing of resources; for synchronization; to keep track of events; to model performance, realistic delay, or phenomena; and for security. Similarly, time is used extensively within networking. Figure 4.1 depicts a collage representing the diverse areas of the conducted survey.

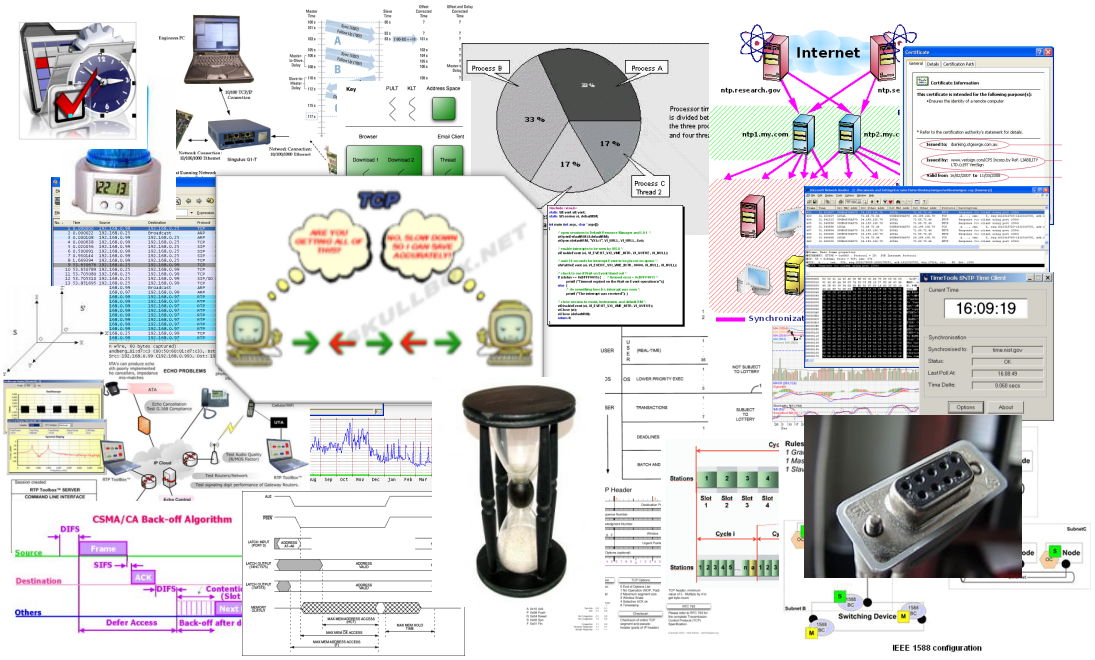

Figure 4.1: Survey of time in networking and computing

In networking, time is used at the physical interface level; hardware clocking is directly used for signaling functions. Above this level though, less direct timing is used. For example, many network protocols involve timeout mechanisms, which specify actions to be taken if a time period has elapsed without some communication event taking place. This requires an alarm clock style of timer to be implemented. Other protocols require explicit timestamps to be placed in packets to guarantee properties such as freshness or uniqueness. This requires a real time clock to be implemented.

In these early days of FPGA acceleration of sophisticated networking functions, the various required clocks and timers are usually implemented on an ad hoc basis, closely integrated with the rest of the system design. This is not desirable in terms of providing maintainable and extensible systems that can evolve with changing requirements. Aside from the drawbacks of monolithic designs, this is counter to any attempts at higher-level design specification techniques.

The aim of this work was to demonstrate that it is not necessary to incur the overhead of re-implementing ad hoc timing capabilities each time some network packet processing function is being accelerated using FPGA technology.

Although this research has focused initially on the particular needs of the important domain of network processing, it has potential application much more widely for other types of real time embedded systems implemented on FPGAs. In essence, it can be seen as a higher level of timing abstraction above the standard digital clock manager blocks that feature in FPGA architectures.

4.1 Timing Paradigms in Networking

At first sight, there is a plethora of ways in which clocks and timers are used in networking. However, if one adopts a time-centric viewpoint of what is happening, as opposed to a protocol-centric viewpoint, the situation becomes dramatically simplified. Indeed, one fairly obvious observation, noted in the past (e.g., in [110]), explains almost the whole picture. This is that communication between two or more parties can be seen as an activity over time with a start point and a finish point. There may be structuring of activities, into sub-activities, sub-sub-activities, etc. conducted over time. Ultimately, an atomic leaf-node activity (in the digital world) could be the communication of a single bit of data between two parties.

4.1.1 Timers and activities

Considering the start points of activities, two main use cases can be identified:

- Activities scheduled at some specific time.
- Missing events recognized after some time period.

The first case includes activities that are deliberately delayed for some time or those that are periodic in nature.

A few standard examples will make the above general description more tangible. The well-known CSMA/CD approach used in Ethernet [111] involves checking for the transmission medium to become idle, and then waiting for a random amount of time before transmitting. In this case, a start point is scheduled for the transmission ready time plus this random time period.

Many control or management protocols, for example the RIP routing protocol [112], involve sending messages at fixed time intervals to provide status information to another entity; in this case, a start point is scheduled for the previous sending time plus this fixed time interval.

The widely used technique of polling deals with expected, but missing, events. When an entity has seen no communication from another entity for some period of time, it starts a polling communication to check on the status of this entity; in this case, the start point is at some fixed time after the last seen communication. The Internet Transmission Control Protocol (TCP) [113] keepalive is an example of polling behavior.

For the finish points of activities, the two main use cases are:

- Lack of activity recognized after some time period.
- Activities terminating at some specific time.

Many communication protocols, notably TCP for example, embody the notion of timeouts used by one entity to recognize when another entity has not responded within some period of time chosen to be longer than the maximum possible response time. In each of these cases, a finish time is scheduled for the start time plus the timeout period. Note that this time-related finish point is nullified whenever an activity finishes naturally through a communication event.

Many security protocols, for example the SIP session protocol [114], embody the notion of an expiry time which limits the duration of activities in order to bound the

time for which an authorization lasts; in this case, a finish point is scheduled corresponding to the expiry time.

When considering the implementation of some specific protocol, it is just necessary to observe where these use cases arise in order to situate timing functions correctly. Then the goal of this work is to provide generic configurable FPGA-based timing modules that can be correspondingly situated as part of modular protocol implementations. The benefit of such hardware modules in general is to provide accuracy and responsiveness that may not be possible with software timing implementations. In some applications, for example the case study presented in Chapter 5, just acceleration of the timing functions is motivation for an FPGA-based implementation.

4.1.2 Clocks and timestamps

The only other significant timing paradigm is the use of clocks is to provide timestamp values, which are included as data within communication activities. These serve a number of purposes in network protocols, including:

- Indicating the time when a message was sent.
- Indicating the time when a message expires.
- Differentiating cases when exactly the same message has been sent more than once.
- Measuring communication times

This use case points to the need for a generic FPGA-based timing module to supply absolute timestamps. These may be absolute times-of-day or relative internal clock values.

A prime example of timestamp use is the Real Time Protocol (RTP) [115], which is concerned with sending real time data, such as audio or video, over the standard Internet best-effort service. RTP packets carry monotonically increasing timestamps

with application-specific time granularity, so that the receiver can deal with packet delay variation. The associated RTCP control protocol uses packets with timestamps in seconds since 1 January 1900.

4.1.3 Time protocols

A special category of protocols is concerned with communicating information about time itself. The principal examples are the Network Time Protocol (NTP) [116] and the IEEE 1588 Precision Time Protocol (PTP) [117]. As its name suggests, the latter is a higher accuracy (potentially sub-microsecond) protocol than the former. These protocols are further examples of those whose packets carry timestamps. Importantly though, these protocols can form part of the implementation mechanism for an FPGA-based module that provides absolute real timestamps.

4.1.4 Time Summary

This walk through the world of timing paradigms in networking (based on an underlying survey and review of networking protocols) motivated the provision of just three necessary and sufficient types of FPGA-based abstract timing modules: for activity start timing; for activity finish timing; and for providing timestamps.

4.2 Configurable Timing Modules

4.2.1 Starting and finishing activities

A characteristic of many protocols is that there can be many simultaneous activities at one time, corresponding to different contexts within the protocol. For example, in the case of the TCP protocol, there is a collection of active connections between TCP ports on the node being implemented and TCP ports elsewhere on the Internet, and there are separate timers for each. Depending on the setting, there might be tens, hundreds, or even thousands of concurrent activities. For this reason, the timing

modules for starting and finishing activities support multiple contexts, as it is not efficient to use separate modules for each activity.

Figure 4.2 shows the interfaces and configurable features of the activity start timing module that was designed. There is a request input interface and an event signaling output interface. The basic timer request includes a start time offset value, meaning that there should be an event signal output at the current time plus the offset value. A repetitive timer request also contains a non-zero period value, meaning that there should be periodic event signal outputs at times separated by the period value. There is also a cancel type of request, used to cancel a currently scheduled timer request.

Each request and event signal includes an identifier, which is used to differentiate between activities. An event signal has the identifier from the corresponding timer request; a cancel request has the identifier of the timer request to be cancelled. There are three configuration parameters for the module: the maximum number of concurrent activities (a); the maximum time horizon (h); and the minimum time quantum (q), which is the unit for the time values in requests and for the time horizon.

Figure 4.3 shows the interfaces and configurable features of the activity finish timing module that was designed. These are broadly similar to those of the activity start timing module. The timer request includes a finish time offset value, meaning that there should be an event signal output at the current time plus the offset value. There is also a done type of request, used to indicate a (non timer caused) activity finish, which has the effect of aborting a currently scheduled timer request. The three configuration parameters are the same as those of the activity start timing module.

The structural similarity between the activity start and finish modules makes a common implementation possible. In fact, the start module has a strict superset of the capabilities of the finish module: the repetitive timer request is its (optional) extra feature; and its cancel request is equivalent to the finish module's done request.

Figure 4.4 shows a logical implementation of the activity start module as a set of alarms, which can then have a physical calendar wheel implementation. Figure 4.5 shows the internal architecture of the timing module calendar wheel implementation.

A stored table contains the future time commitments for the timer requests in progress: a completion time, and optionally a repetition period, for each activity. It has a rows, each with width $r\lceil \log_2 h \rceil$, where r=2 if repetitive requests are allowed and r=1 otherwise. On Xilinx FPGAs, this can be stored in Block RAM (BRAM) memory or in distributed LUT RAM memory. For a Virtex-5 FPGA, a single BRAM can store 36K bits and a single LUT can store 64 bits, with the table requiring a total $ar\lceil \log_2 h \rceil$ bits. The timer request arbiter writes to the table to schedule events based on incoming requests.

Figure 4.2: Activity start module

Figure 4.3: Activity finish module

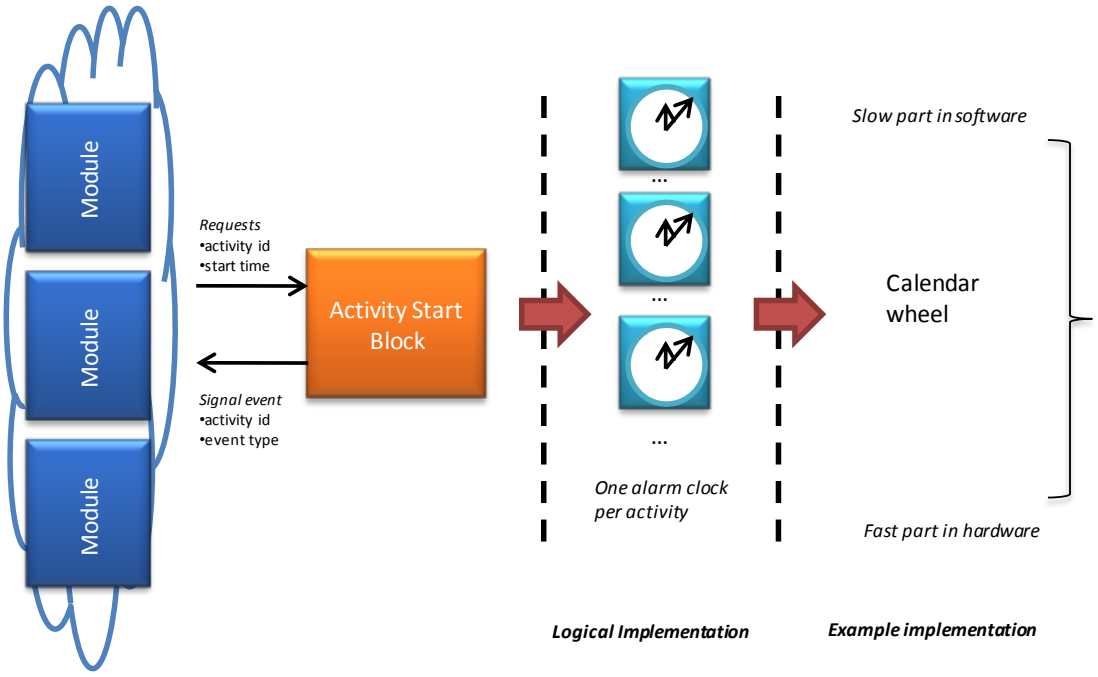

Figure 4.4: Logical implementation of activity start module

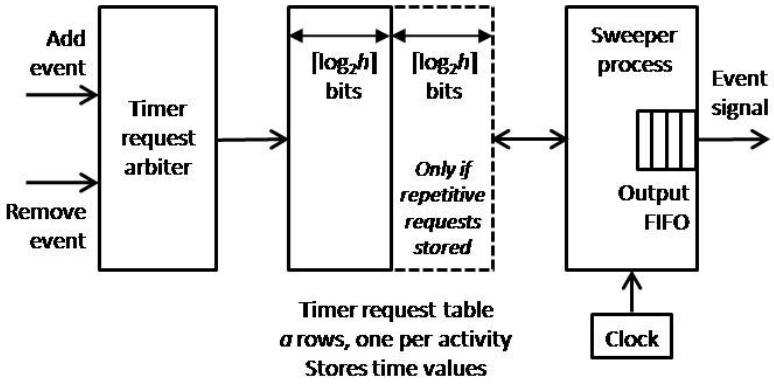

Figure 4.5: Calendar wheel implementation of activity start and finish timing modules

A sweeper process scans through the table on a regular basis, checking for any timer requests that have completed, and generating event-signaling outputs in such cases. The sweeper spends a (deterministic) five cycles per table row on the check and any follow-up. Therefore, if the maximum module hardware clock rate is c MHz, the maximum scan frequency is $c/5a$ million sweeps per second. This, in turn, imposes a lower bound of $5a/c$ μs on the minimum time quantum q. So, for example, a single

module with a clock rate of just 125 MHz could support 25,000 activities using a 1 ms time granularity, which is more than ample for most networking protocol needs. Note that a typical software implementation would use a more subtle data structure, e.g. a sorted event list, but the method used here is well suited for hardware implementation because it minimizes memory use.

Table 4.1 shows Xilinx Virtex-5 LXT implementation data for nine representative configurations with repetitive requests allowed (r=2): time horizon width $\lceil \log 2h \rceil$ = 16, 24, and 32 bits, and activity maximum a = 128, 1024, and 8192. Block RAM was used for the table storage and for the signal output FIFO. It can be seen that the LUT, FF, and slice counts increase with the time horizon width, because of the need to store time values and to compare them to check for completion, and (less so) with the number of activities, because of the need to use counters of $\lceil \log_2 a \rceil$ width. The BRAM counts increase in line with the $2a\lceil \log_2 h \rceil$ formula for table size; the number of BRAMs used in fact has the most impact on clock frequency because of fan-in considerations.

Table 4.1: Xilinx Virtex-5 data for activity timing modules

Time width (bits)	Max. activities	Lookup tables (LUTs)	Flip- flops (FFs)	Virtex-5 slices	BRAM (36Kb) count	Clock freq. (MHz)
16	128	322	329	185	2	299
	1024	330	335	192	2	280
	8192	375	364	224	9	236
24	128	412	435	247	3	281
	1024	418	439	244	3	278
	8192	466	438	271	13	201
32	128	502	504	259	3	263
	1024	507	507	294	3	266
	8192	571	473	299	17	195

4.2.2 Providing timestamps

Figure 4.6 shows the interface and configurable features of the timestamp-providing module that was designed. Compared to the other modules, it has a simple interface. This supports a simple register read request that returns a current timestamp. An alternative would have been for the module to output a timestamp continuously. Note that this module's interface could support the Worker Time Interface (WTI) profile of the OpenCPI open component portability infrastructure initiative [43]. The key configuration parameter for this module is whether it supplies its own localized timestamp sequence, initialized at reset, or whether it supplies a real time-of-day timestamp. The latter potentially involves a significantly more complex implementation. For each case, derived parameters are then the maximum time horizon, which determines the size of the timestamp, and the minimum time quantum, which determines the accuracy of the timestamp. A final configuration parameter is the number of read request interfaces that are supported. This multi-port memory option is provided to relieve the module user of having to multiplex read requests from several different client modules.

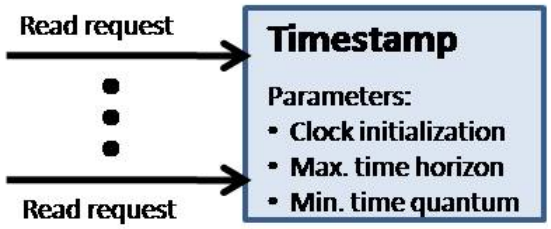

Figure 4.6: Timestamp providing module

In the case where the module supplies a localized timestamp sequence, the FPGA implementation is trivial, since it just requires a simple counter of the appropriate size that is incremented at the appropriate frequency, plus one or more standard register read interfaces. With a module clock rate of 200 MHz, the lower bound on the minimum time quantum is 5 ns, much smaller than needed in practice.

In the case where the module supplies a real time-of-day timestamp, there are various different options. The simplest approach is to use a simple counter as just described, initialized to a current time-of-day value. For example, it can be a 64-bit counter of seconds since 1 January 1970 (as used in modern Unix), with an initial value supplied as part of system configuration via a control register interface. Where there is no in-system way of supplying the current time, a more elaborate approach would be to embody a complete IEEE 1588 client within the module, for example the IPClock IPC50000 networked slave clock block [118].

4.2.3 Activity diagrams

Activity diagrams are a novel graphical way to indicate which of the time modules are needed for a particular activity and the actions within the activity. An example activity diagram, illustrating the graphical notation, is shown in Figure 4.7. The gray box represents the activity and inside it is a listing of the actions performed during the activity. The list of actions is ordered according to their sequence. Time is depicted as flowing left-to-right and so the activities and actions are displayed in order left-to-right.

Labels above the gray box are used to indicate there is a requirement for a timing module and that the action directly below it is dependent on using time. In the example shown, action 1 requires the activity start module (START) and action *n* requires the activity finish module (FINISH).

Figure 4.7: Activity diagram notation

An asterisk (e.g. START*) is used to denote that the activity repeats, and that start module will be used to periodically signal the start of this activity. An example is

shown in Figure 4.8, which periodically transmits a packet (1DM frame), containing a timestamp (TxTimestampf). The activity start module notifies the activity to begin forming the 1DM frame. Next, a timestamp is obtained from the timestamp module (STAMP). The TxTimestampf is a field of the 1DM frame. Finally, the 1DM frame is transmitted, and the activity is complete.

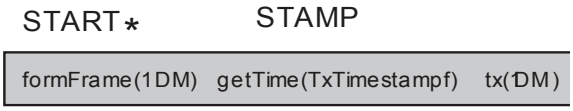

Figure 4.8: Activity diagram with an asterisk

An activity does not necessarily need to use timing modules for beginning or ending its actions. For example, the activity in Figure 4.8 is short, and it ends naturally after transmitting the packet. As mentioned, the periodic nature of the activity means that the start module will restart that activity after its period has elapsed. The example activity shown in Figure 4.9 begins naturally, when a packet (LBM frame) is received). The activity finish module is used to implement a timer to notify the activity when the random wait period has expired. The notification from the finish module triggers the transmission of a packet (LBR frame).

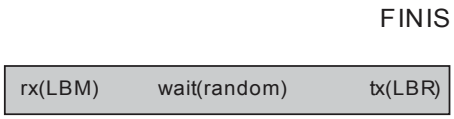

Figure 4.9: Activity diagram of an activity that begins naturally, without the use of a timing module

Activity diagrams will be used in Chapter 5 to illustrate the timing requirements of the main functions of the case study.

4.3 ShapeUp Context for Timing Modules

The three configurable timing modules were designed to fit within the ShapeUp framework, to maximize their usability and reusability within modular networking

system (or other embedded system) designs. In fact, software implementations of these modules could also be used in this setting. The specifications of the module interfaces involve two of the five defined ShapeUp interface types: access, where a primary module accesses data in a secondary module via read and write requests; and notify, where a primary module passes messages to a secondary module.

The modules for starting and finishing activities have access type request input interfaces, the module being the secondary and the interface having address-less and write-only (writing an activity identifier and one or two time values) attributes. Figure 4.10 shows the ShapeUp interface for the activity start module and the activity finish module. They have notify type event signaling output interfaces, the module being the primary and the messages carrying an activity identifier and an event type indication. The module for providing timestamps has an access type request interface, the module being the secondary and the interface having address-less and read-only (reading a timestamp value) attributes. Figure 4.11 shows the ShapeUp interface for the timestamp-providing module.

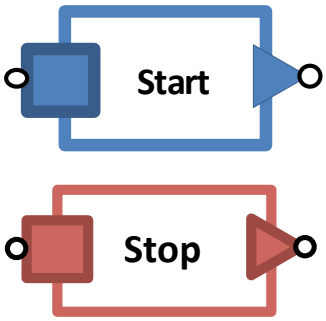

Figure 4.10: ShapeUp activity start and finish timing modules

Figure 4.11: ShapeUp timestamp providing module

4.4 Summary

This work is a contribution to encouraging a higher-level approach to designing FPGA-based networking systems. Timing is a feature of almost all communication protocols but, as a review of networking showed, there are just a small number of basic timing paradigms in use. This motivated the design of the collection of configurable networking timing modules introduced in this chapter. These components might have either software or hardware implementations, the latter being necessary for an increasing number of applications as networking speeds grow from gigabit rates towards terabit rates. Resource-efficient FPGA implementations of the modules have been embedded within the new ShapeUp modular design methodology. The fact that Click is used as a description language in ShapeUp assists accessibility for networking researchers who are already familiar with Click for modular software implementations. Although motivated by the needs of networking, the new configurable timing modules have potential applications in many types of real time embedded systems where there are events and activities that are influenced by the passage of time. Thus, they represent one of a core set of generic module libraries that contribute to the overall ShapeUp methodology.

To enable progress towards more flexible and modular design of networking systems, the main contributions of the work described in this chapter are threefold:

- A wide-ranging review of the prevalent timing paradigms observed in network protocols, which exposed and abstracted three basic timing functions requirements. This is summarized in Section 4.1.
- The design and implementation of a set of three highly configurable timing modules that provide a flexible solution for the identified basic requirements. These are described in Section 4.2. Activity diagrams were created to show time requirements and the use of the three timing modules as they relate to individual activities. These are described in Section 4.2.3.

- The embedding of these modules within the experimental ShapeUp methodology for modular system design, to allow seamless integration with other modules. This is described in Section 4.3.

The next chapter describes the validation of the timing modules (and ShapeUp) through use in real-life industrial-strength case studies of network processing acceleration.

Chapter 5

Case study 1: A Scalable Modular System Design for Ethernet OAM

In this chapter, the ShapeUp methodology and tools are validated on a Xilinx customer design project. This case study concerns a modular reference design that has been shared with a number of FPGA users in the telecommunications industry. A key benefit of ShapeUp was the capability to have a set of modules, and then easily assemble these in different configurations corresponding to specific system requirements. The application is hardware acceleration of Ethernet Operations, Administration and Maintenance (OAM) functions, an area of rapidly increasing importance in modern carrier Ethernet. It was selected because of both its importance and also its numerous and subtle uses of time.

Section 5.1 provides an overview of Ethernet OAM, including a description of the network entities and the protocol functions. Section 5.2 provides an analysis of the timing requirements in Ethernet OAM functions. Section 5.3 describes the overall system architecture. Section 5.4 gives relevant background on the G language (used for implementing this case study), and describes the library of G elements. Section 5.5 discusses the integration of the timing modules within the example designs. Section 5.6 provides a Click description of one major part of the system: the connectivity fault management (CFM). Section 5.7 describes how the ShapeUp framework and tools and methodology enabled this CFM design. Section 5.8 summarizes the contributions of this chapter.

5.1 Ethernet OAM in a nutshell

Ethernet OAM is specified in the ITU-T Y.1731 [119] and IEEE 802.1ag [120] standards. These standards address the scaling of service and maintenance across different network domains. While Ethernet service networks scale to increasing number of services and customers, it remains important for service providers to guarantee their services with monitoring and maintenance. Ethernet OAM provides a set of management services for administering Ethernet services across multiple network domains. This helps to provide an organized environment for detecting and reporting errors that occur across service levels. Example service levels are shown in Figure 5.1. A maintenance entity (ME) is simply a network entity that requires management. Functions are performed between ME groups (MEGs) that are peers (i.e. they have the same service level), represented by colored rectangles and colored circles in Figure 5.1. A MEG end point is abbreviated as MEP (represented by a colored square), and a MEG intermediate point is abbreviated as MIP (represented by a colored circle).

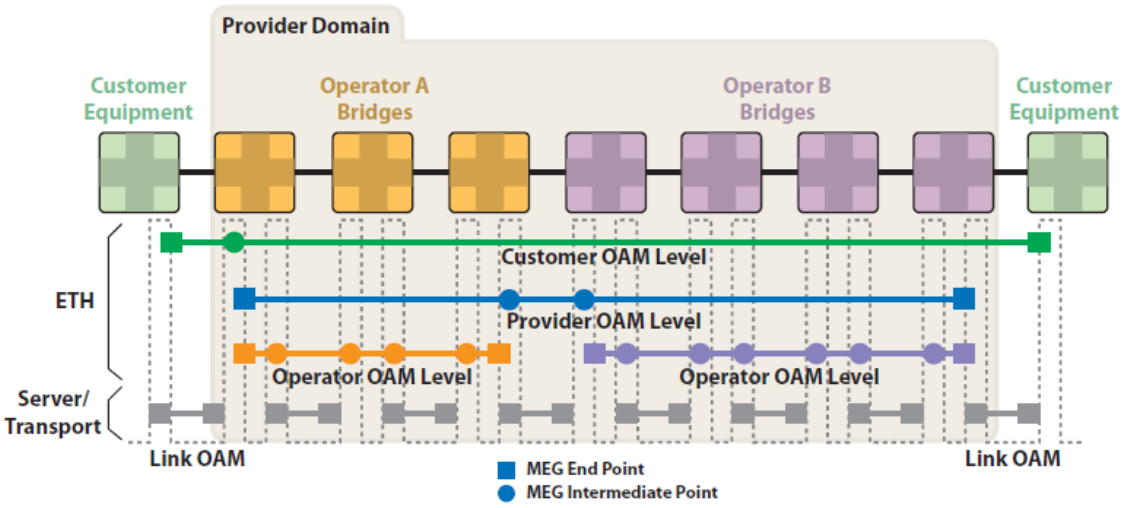

Figure 5.1: Ethernet OAM service levels, taken from [119]

This section will describe the functions and protocols that are described in ITU-T recommendation Y.1731. There are two basic management aspects to Ethernet

OAM, consisting of: (a) fault management and (b) performance monitoring. The fault management functions generally include: (a1) checking for lost continuity and other defect conditions, (a2) configuring diagnostic testing modes, and (a3) signaling alarms. The performance monitoring functions generally include: (b1) measuring frame loss, (b2) measuring delay, and (b3) measuring throughput.

Fault Management consists of the functions for detecting various kinds of defect conditions as well as functions for setting up modes in order to perform diagnostic testing:

Continuity Check (ETH-CC)	Used to check connection continuity by periodically transmitting test frames and also used to measure frame loss.
Loopback (ETH-LB)	Used to verify bidirectional connection, with ping-like request/reply function.
Link Trace (ETH-LT)	Used to trace the path to a peer and to isolate faults.
Alarm Indication Signal (ETH-AIS)	Used to signal connection failures to next level service.
Remote Defect Indication (ETH-RDI)	Used to signal defect conditions from a remote peer in the upstream direction.
Locked Signal (ETH-LCK)	Used to suppress alarms and for differentiating an administrative mode used for performing diagnostic testing.
Test Signal (ETH-Test)	Used to send a test message for testing throughput, to measure bit errors, or to detect out of sequence delivery.
Automatic Protection Switching (ETH-APS)	Used to control protection switching operations to enhance reliability.
Maintenance Communication Channel (ETH-MCC)	Used as a maintenance channel to request maintenance functions from a remote peer.

Experimental OAM (ETH-EXP)	Used to allow administrative functionality on a temporary basis.
Vender Specific OAM (ETH-VSP)	Used to allow vender specific Ethernet OAM extensions.

Performance monitoring consists of the following basic performance measurement functions:

Frame Loss Ratio (ETH-LM)	Report of frames not delivered vs. frames delivered
Frame Delay (ETH-DM)	Both unidirectional (with IEEE 1588) and bidirectional/round-trip measurement functions for computing frame delay.

Modern carrier class networks require systems supporting high aggregated throughput, e.g. 25 Gb/sec. Parts of the OAM functions require hardware acceleration due to these scaling line rates, including maintaining the ability to count frames and also because many of these functions require highly accurate timestamps. For example, ETH-CC is a key function that requires hardware acceleration because the measurement rate has increased from every few seconds to more recently a polling interval of every 3.3 ms. Furthermore, this polling may be required for up to one thousand simultaneous contexts. Figure 5.2 shows an illustration of the ETH-CC function, with a MEP transmitting continuity check (CC) frames to a peer MEP. In this example, MEPs are represented using colored triangles. The ETH-CC function is depicted, and the red arrow shows a flow of CC frames that are transmitted from the blue MEP on the left to the blue MEP on the right.

Although the examples in Figure 5.1 and Figure 5.2 show the same simple network topology, containing a few MEG peers, real deployments have more complex topologies. For example, the topology might consist of multipoint-to-multipoint networks, as shown in Figure 5.3. In this example, the ETH-CC function is shown, and the red arrow shows three different flows of CC frames that are transmitted from

the blue MEP on the top left to the other peer MEPs. The OAM systems described in this chapter are designed to support up to one thousand different flows.

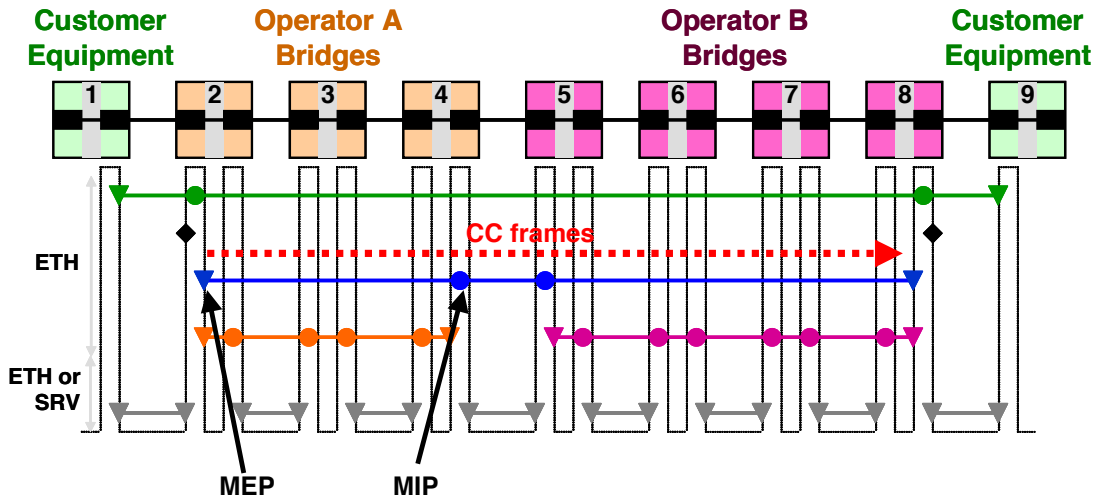

Figure 5.2: Continuity check (CC) function tests the connection status between peer MEPs, shown as triangles, taken from [121]

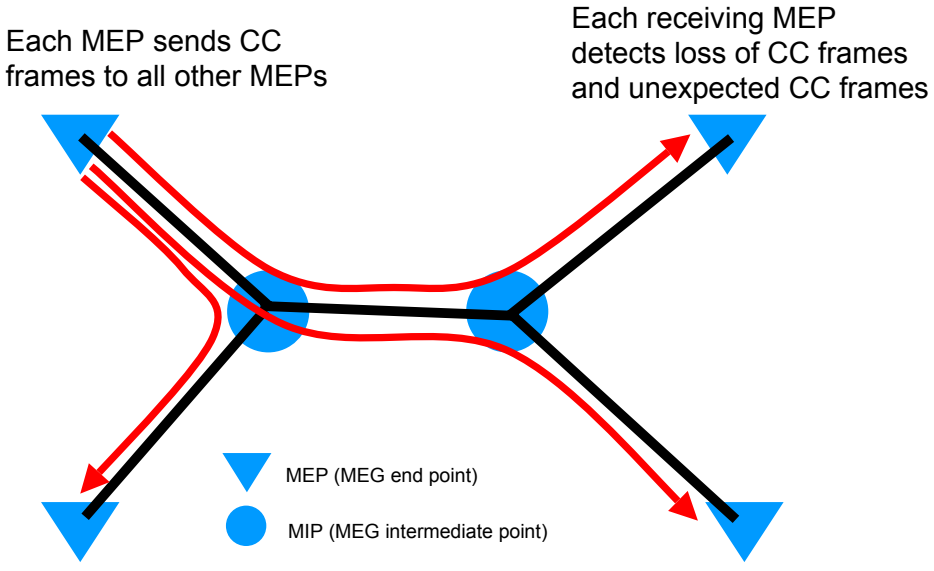

Figure 5.3: Continuity Check in a multipoint-to-multipoint network, taken from [121]

5.2 Analysis of Timing Requirements

This section presents an analysis of the timing requirements, expressed in terms of the novel activity diagram notation, introduced in Section 4.2.3. As discussed earlier, activity diagrams illustrate a summary of behavior over time. This is required for each of the functions described in Section 5.1. The left column shows the activity of the MEP sender. The right column shows the activity of the MEP receiver. The formats for Ethernet OAM frames are organized by function and they are referred to in the standard as protocol data units (PDUs). In general, there is approximately one PDU format for each of these functions.

ETH-CC: The sender initiates the periodic transmission of continuity check measurement (CCM) frames, requiring the START module, at one of the defined rates, e.g. the CC_period equals every 3.3 ms. On the receiver side, ETH-CC, waits 3.5 times the CC_period to receive the expected incoming CCM frame from its peer (the sender), before timing out, requiring the FINISH module. If a timeout occurs the receiver signals a defect condition and initiates an alarm.

ETH-LB: The sender initiates the periodic transmission of loopback message (LBM) frames, requiring the START module, and the sender completes when the sender receives the loopback response (LBR) frame from the receiver or it times out, requiring the FINISH module. The receiver receives the loopback response frame and then waits for a random amount of time before transmitting the LBR to the sender, requiring the FINISH module.

ETH-LT: The sender transmits a link trace measurement (LTM) frame and then waits to receive a link trace response (LTR) frame. The sender times out if the LTR is not received, requiring the FINISH module. When the receiver receives a LTM frame it waits for a random amount of time before transmitting the LTR to the sender, requiring the FINISH module.

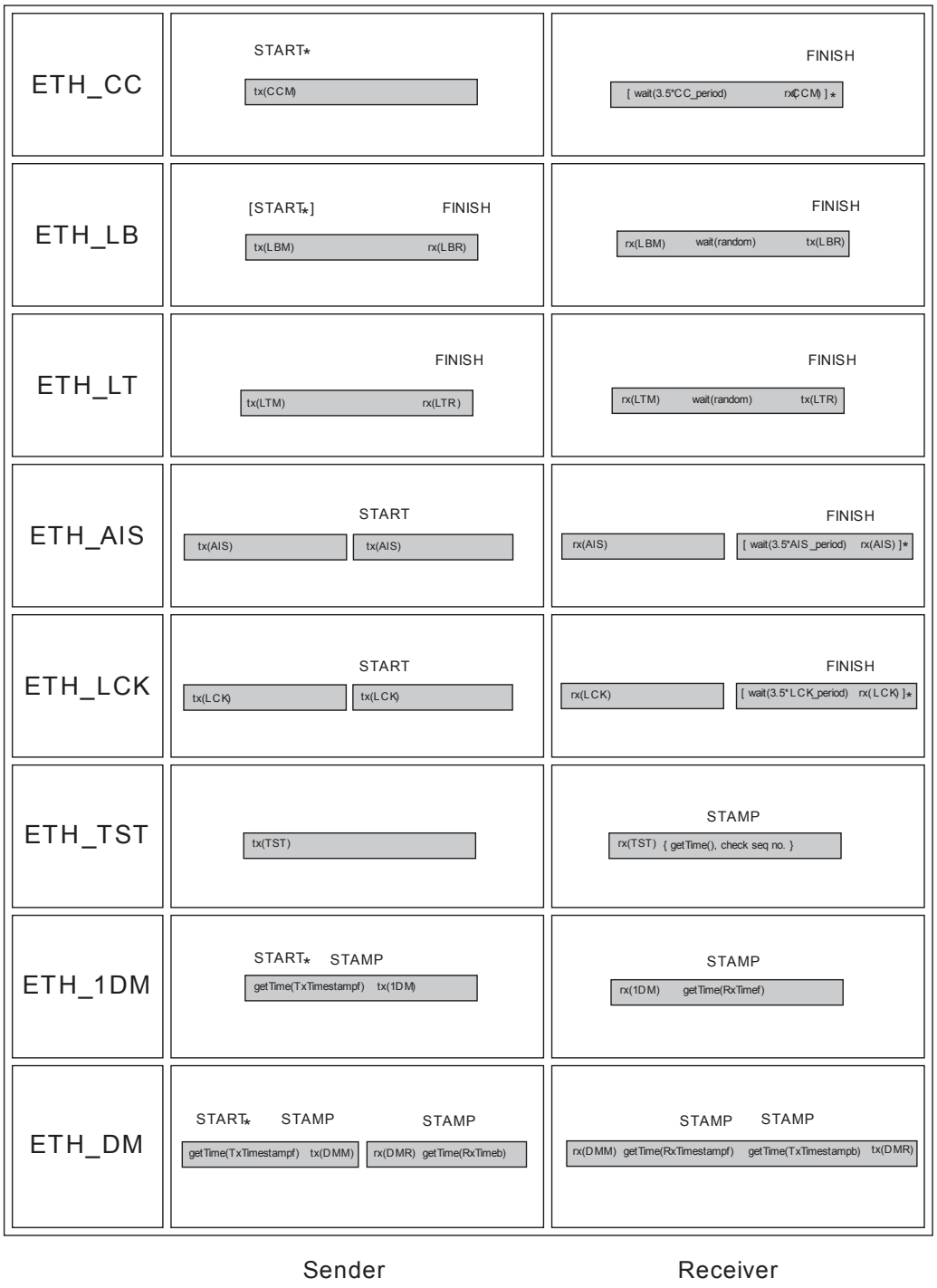

Figure 5.4: Activity diagram for Ethernet OAM functions

ETH-AIS: The sender side during an alarm condition transmits an alarm indication signal (AIS) frame to its relevant peers. Then the sender waits for the AIS_period and then transmits another AIS frame, requiring the START module. After the receiver receives an AIS, it waits for 3.5 * AIS_period expecting to receive another AIS before timing out. If the timeout condition occurs, requiring the FINISH module, then the alarm is canceled.

ETH-LCK: The sender side transmits an Ethernet administrative lock signal (LCK) frame to its relevant peers. Then the sender waits for the LCK_period and then transmits another LCK frame, requiring the START module. After the receiver receives an LCK, it waits for 3.5 * LCK_period expecting to receive another LCK before timing out. If the timeout condition occurs, requiring the FINISH module, then the lock is canceled.

ETH-TST: The sender transmits the Ethernet test (TST) frame to the receiver. When the receiver receives the TST, it uses the STAMP module to get the current time, and then it checks the sequence number.

ETH-1DM: Initiated by the START module, he sender periodically gets the current time using the STAMP module, inserts the timestamp into the one-way delay measurement (1DM) test frame, and then transmits the 1DM to the receiver. When the receiver receives the 1DM, it gets the current time, using the STAMP module and calculates the one-way delay time.

ETH-DM: Initiated by the START module, the sender periodically gets the current time using the STAMP module, inserts the timestamp into the round-trip delay measurement (DMM) test frame, and then transmits the DMM to the receiver. When the receiver receives the DMM, it gets the current time using the STAMP module, and calculates the one-way delay time. Then the receiver gets the current time using the STAMP module, and inserts it into the delay measurement response (DMR)

frame. The receiver transmits the DMR to the sender. When the sender receives the DMR it gets the current time using the STAMP module, and calculates the round-trip time.

5.3 System Architecture

A simplified view of the architecture is shown in Figure 5.5. The OAM capability is independent of the line-side and system-side interfaces. A more detailed schematic of the overall OAM subsystem framework (including the CPU interface) is shown in Figure 5.6. The block in the center shows how the OAM subsystem framework connects to the line-side and system-side interfaces. The line-side interface contains an Ethernet MAC interface. The system-side interface includes a CPU interface, a loopback interface, and a data plane interface. The OAM subsystem framework is flexible in the functions that are accelerated, depending on the design. OAM subsystems can be assembled from the OAM elements described in the next section.

Figure 5.5: Setting for the OAM design

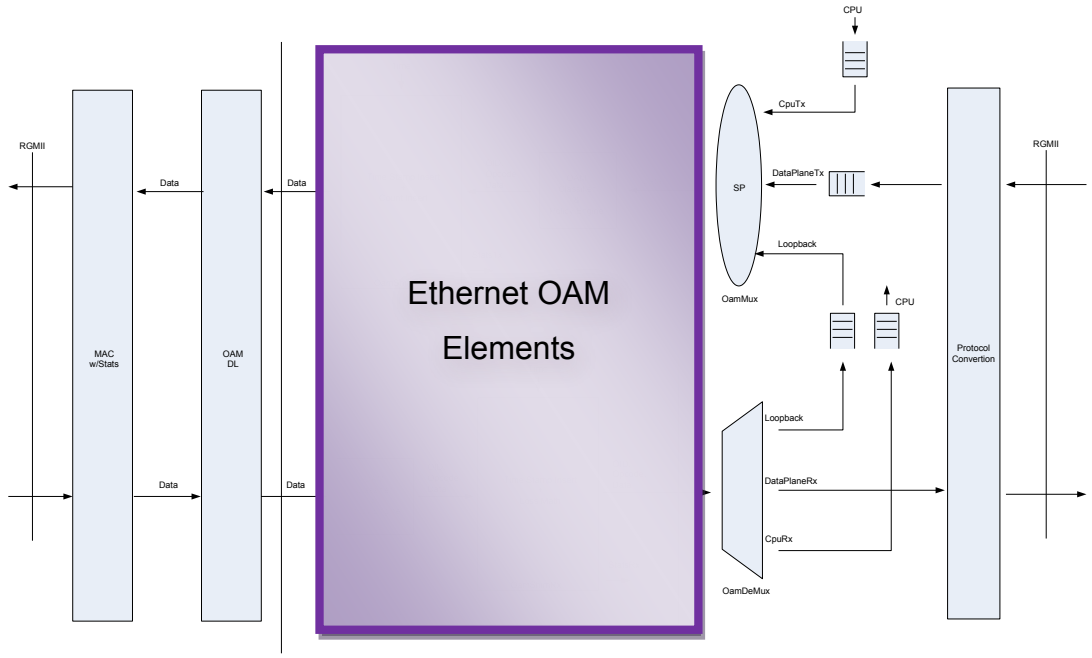

Figure 5.6: Detailed schematic of the overall OAM framework

5.4 OAM Elements, and the G Language

The individual modules (Click elements) in the OAM case study were implemented using the G 2007 language [86]. G is a high-level, domain-specific language for describing modules that perform packet processing functions. G is complementary to the overall ShapeUp framework because it raises the level of abstraction for designing individual modules, which are then used to build networking systems targeting programmable hardware. G was designed to share the same abstractions for module interfaces from ShapeUp. G descriptions can be compiled into RTL code that has efficient FPGA implementations. G can be used to create new elements for expanding the ShapeUp library. Together, the ShapeUp framework and tools, along with G, provide an efficient framework for designing, integrating, and validating packet processing functions, all at a high level of abstraction.

5.4.1 Overview of the G language

G 2007 descriptions consist of two main parts: (a) packet data formats and (b) a handler. The handler performs a set of operations on incoming packets. The format of incoming packets is declared at the top of the handler. A G 2007 module may contain one incoming stream port and multiple output stream ports. G modules are reactive in that the handler is triggered by the arrival of packets on the input packet port. The handler typically performs packet surgery and then forwards the packet on a selected output stream port. The set of modification operations for performing packet surgery generally consists of: insert, edit, and remove operations applied to the fields of the packet header.

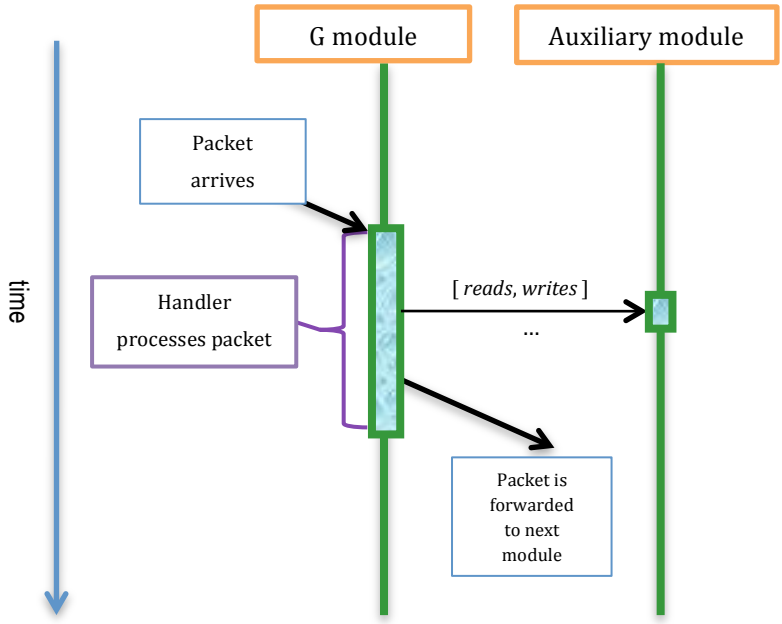

Figure 5.7: G module UML interaction diagram

Figure 5.7 shows a UML-style interaction diagram of a G module and an auxiliary module. The example shows the G module receiving a packet, which triggers the module's handler to begin processing the packet header. As shown, the processing may additionally include interaction between a G module and auxiliary modules that have an access interface, which allows the G module to e.g. read from lookup tables

or to perform stateful processing like updating frame counters. After processing, the packet is forwarded on its output stream interface to the next module in the design, e.g. to the next stage within a packet processing pipeline.

The reactive behavior of G modules is sufficient for many protocols, however, as discussed in Chapter 4, network protocols often require the use of time. Some high-level protocols may involve timeouts, e.g. TCP, where a timeout is an event that triggers data retransmission. Additionally, some networking systems may require proactive behavior, e.g. polling to monitor the state of a network peer. However, G 2007 did not include syntax or built in mechanisms to function according to time. This case study shows how the G modules were used, in conjunction with the timing modules from Chapter 4, to enable the necessary proactive behavior. A ShapeUp library of elements was created for Ethernet OAM, with modules implemented in G, facilitating the creation of two OAM reference designs.

5.4.2 Ethernet OAM reference designs

Two reference designs were implemented as example ShapeUp systems that use the OAM library elements. The first system is the Y.1731 reference design, which implements functionality that does not require the start and finish modules. To summarize, the Y.1731 OAM reference design performs the following functions:

a) parses incoming packets
b) classifies between the OAM frames and the user frames
c) counts the "in-profile" OAM frames
d) inserts a timestamp and sequence number information in OAM frames
e) inserts loss math into overloaded fields for upstream collection of the CCM PDU
f) delivers OAM frames to an upstream function

The second system is the CFM reference design that extends the Y.1731 functionality by adding functions that require the start and finish modules. The CFM design additionally:

g) generates CCM continuity check frames for transmission
h) checks incoming CCM frames against a table of expected values
i) detects defect conditions associated with CCM reception and informs the software controller
j) supports programmable time intervals, for example: 3.3ms, 10ms, 100ms, 1s

The focus in this chapter is on the second system because it has the timing module interest.

Appendix A contains a description for each of the OAM library elements that were used in the example reference designs. Appendix B contains the G source code for an example implementation of one of these elements.

5.5 Integration of the timing modules

This section describes how the timing modules are integrated into the CFM system. ActivityStart and ActivityFinish are instances of the timing modules from Chapter 4. Figure 5.8 shows the interaction between the timing modules and the CCM modules. The start module activates the CCM generator to read a partial frame from memory and to transmit the new CCM frame. When a CCM frame arrives at the CCM checker, it resets the corresponding Activity Finish module's timer. If one of the Activity Finish module's timer expires, then it signals the CCM checker to report the defect condition.

Figure 5.9 shows a view of the interaction between the Activity Start module, the CCM generator, and the control processor. The Activity Start module manages alarms for each of the contexts. The alarms are programmable to 100 μs accuracy (for a 3.3 ms interval). Either the CCM Generator or the control processor

configures the timers, for example, in the current implementation the control processor configures the timers. The Activity Start module signals an event message, containing: an activity ID, a 16-bit timestamp, an event type, and a reserved field. The arrival of the event message triggers the CCM generator to produce a new CCM frame. The contents of the CCM frame are based on values read from local on-chip memory tables.

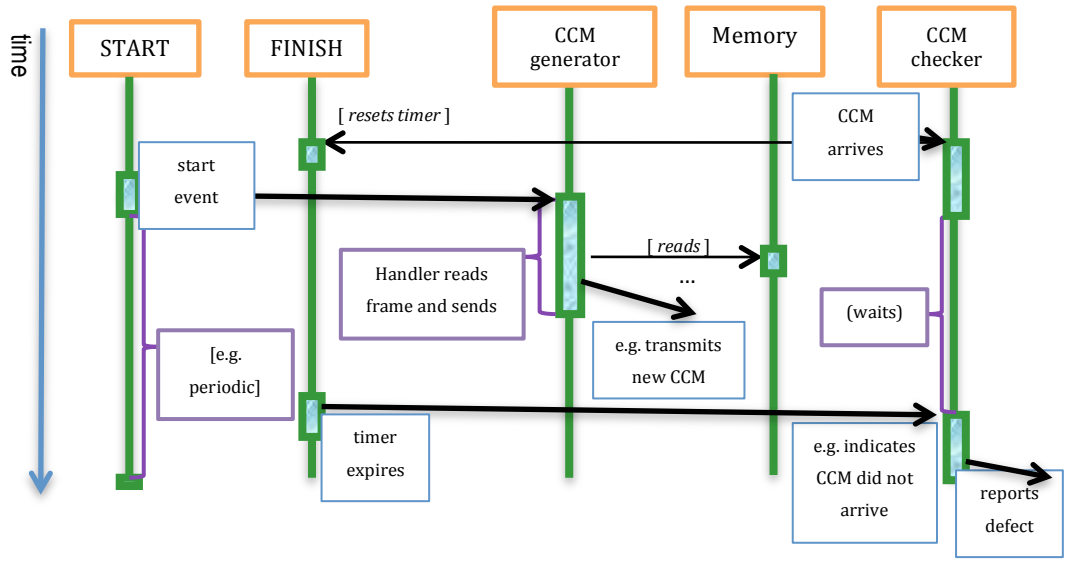

Figure 5.8: Interaction diagram for CFM design showing the system interaction between the timing modules and the OAM modules

Figure 5.10 shows a view of the interaction between the Activity Finish module, the CCM checker, and the control processor. The ActivityFinish module manages timers for each of the contexts. Similarly, the timers are programmable to 100 µs accuracy and either the CCM checker or the control processor configures the timers. The activity finish module signals an event message, containing: an activity ID, a 16-bit timestamp, an event type, and a reserved field. The event message is sent to the control processor. The CCM checker resets the timer, when the corresponding CCM frame is received.

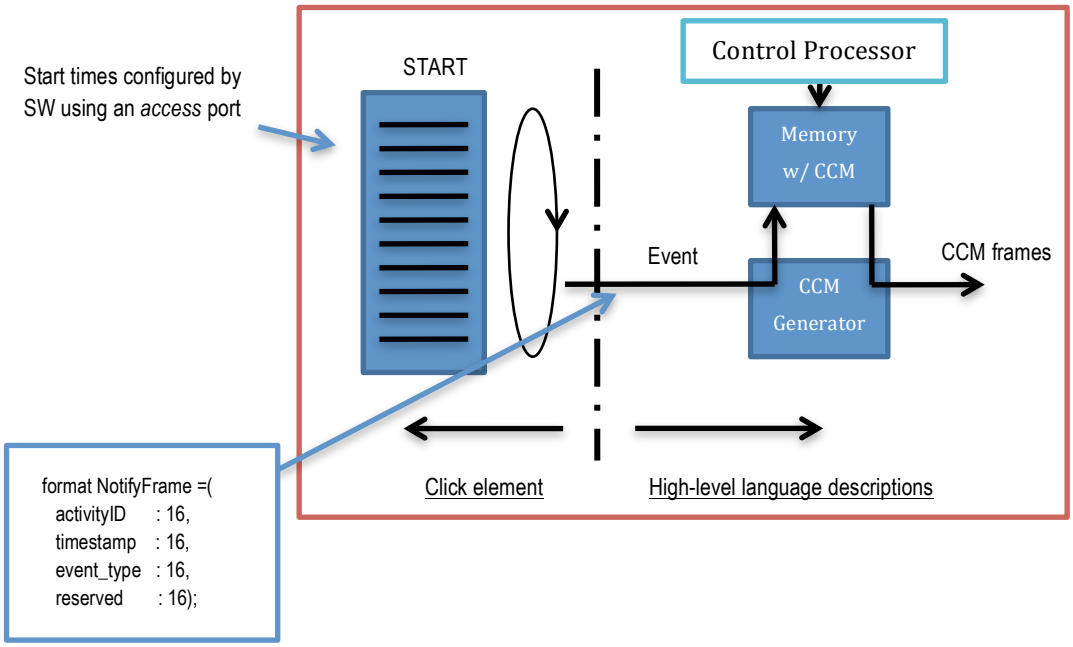

Figure 5.9: Start module activates the CCM generator to periodically transmit CCM frames to peer MEP

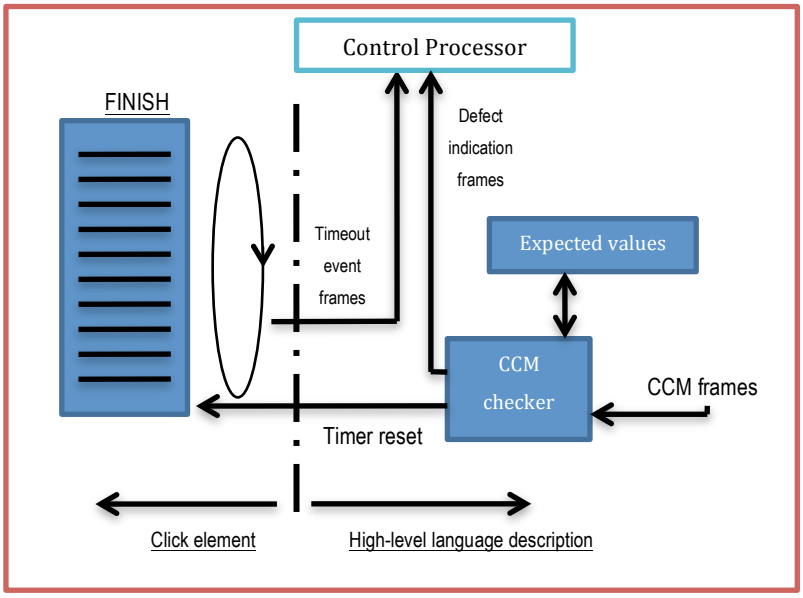

Figure 5.10: Finish module polices the reception of CCM frames from peer MEPs and times out if no CCM frame is received

5.6 Click description

The Click for the CFM design is shown in Figure 5.11, continued in Figure 5.12. In this example, OAM frames are received from line side, processed, then forwarded to system side; when expected OAM CCM frames are not received, timeouts are used to inform the system side. In the opposite direction, stimulated by a periodic timer, OAM CCM frames are constructed and transmitted to line side. These activities are steered by consulting various lookup tables.

Certain features of the Click description are worthy of attention. In lines 1 to 17, instances of Click elements are declared. Each element class name has a configuration string (Click terminology) denoting its source language, e.g. "TYPE G" means written in the G language [86]. The implementation of each of the modules written in G is described in Appendix A. The example also shows types "C" and "VHDL". For example this mixed-language information is also useful for the validator tool to guide it in how high-level simulation models should be used.

Following the declarations, this example includes four of the five interface types, the plain type not being required. Hungarian-notation port names indicate the type. Lines 20 to 34 show the two main streaming data paths in the system, "FromDevice" and "ToDevice" being Click conventional names for receivers and transmitters respectively. The remaining connections involves access and compute type interfaces, the final line showing a connection that allows the cfm_in module to call a function in the controller module, which is actually a software module written in C.

At line 11 of the Click description, an activity start timing module is declared with the name "start". This is used to cause the periodic generation of outgoing OAM CCM frames. In the reference design, there could be up to 1024 OAM flows at any time, and so the start timing module was configured for 1024 activities. The time between frames could vary from flow to flow, being one of 3.3 ms, 10 ms, 100 ms, or 1 s. To support this, the module was configured with a 100 μs time quantum and 14-bit time horizon width. The repetitive timer requests originate from an embedded

controller (declared at line 17), and line 55 shows the connection made between this module and the timing module. Here, "A_request" is the name of the request input interface, with the "A_" being Hungarian notation to indicate that it is of the access interface type. The timing module sends event signals to a packet generation module (declared at line 9), and line 29 shows the connection made between the modules, "N_signal" being the name of the (notify type) event signaling output interface.

At line 12 of the Click design, an activity finish timing module is declared with the name "finish". This is used to generate a timeout signal when no incoming OAM CCM frame is received on a flow for a time period of 3.5 times the flow's inter-CCM time. The configuration of this module was the same as for the start module, except for having an increased 16-bit time horizon width. Line 40 shows the connection between a packet reception module and the timing module. A new timeout request is made each time a frame is received; note that a new request automatically aborts any existing scheduled request for the same activity. Line 56 shows the connection between the timing module and the embedded controller, to signal any timeout events for software handling.

A timestamp-providing module is declared at line 13, and lines 43 and 44 show connections to it from packet reception and transmission modules respectively. This module provides 64-bit localized timestamp values. In the former case, this value is used for checking a timestamp in an incoming frame; in the latter, it placed as a timestamp in an outgoing frame. The module was configured with two request interfaces (named "A_time1" and "A_time2" here).

```
1.  /* Declare element instances */
3.  y1731_cl_in :: VlanClassifier(TYPE G);
4.  y1731_cl_out :: VlanClassifier(TYPE G);
5.  y1731_in :: OAM_Y1731_In(TYPE G);
6.  y1731_out :: OAM_Y1731_Out(TYPE G);
7.  cfm_in :: CheckCcm(TYPE G);
8.  cfm_out :: GenerateCcm(TYPE G);
9.  preread :: CalcAddress(TYPE G);
10. ccm_reader :: FrameReader(TYPE VHDL);
11. start :: StartActivity(TYPE VHDL);
12. finish :: FinishActivity(TYPE VHDL);
13. timeref :: TimeStamp(TYPE VHDL);
14. contextIDs :: ContextsIdTable(TYPE VHDL);
15. vlanProfiles :: VlanProfileTable(TYPE VHDL);
16. melContexts :: MelContextsMem(TYPE VHDL);
17. controller :: EmbeddedController(TYPE C);
18.
20. /* Inbound frame handling path */
21. FromDevice(LineSide)
22. -> [S_in]y1731_cl_in[S_out]
23. -> [S_in]y1731_in[S_out]
24. -> [S_in]cfm_in[S_out]
25. -> ToDevice(SystemSide);
26.
28. /* Generates outbound CCM frames */
29. start[N_signal] -> [N_in]preread[N_out]
30. -> [N_in]ccm_reader1[S_out]
31. -> [S_in]cfm_out[S_out]
32. -> [S_in]y1731_cl_out[S_out]
33. -> [S_in]y1731_out[S_out]
34. -> ToDevice(LineSide);
```

Figure 5.11: Click description of the connectivity fault management (CFM) design

```
35.
37. /* Auxiliary connections*/
39. /* Reset timer when CCM arrives */
40. cfm_in[A_reset_timer] -> [A_request]finish;
41.
42. /* Connections to timestamp reference */
43. y1731_in[A_timestamp] -> [A_time1]timeref;
44. y1731_out[A_timestamp] -> [A_time2]timeref;
45.
46. /* Connections to shared lookup tables */
47. y1731_cl_in[A_pTbl] -> [A_pTbl1]vlanProfiles;
48. y1731_cl_out[A_pTbl] -> [A_pTbl2]vlanProfiles;
49. y1731_cl_in[A_cTbl] -> [A_cTbl1]contextIDs;
50. y1731_cl_out[A_cTbl] -> [A_cTbl2]contextIDs;
51. y1731_in[A_mTbl] -> [A_mTbl1]melContexts;
52. y1731_out[A_mTbl] -> [A_mTbl2]melContexts;
53.
54. /* Connections to embedded controller */
55. controller[A_CCM_req] -> [A_request]start;
56. finish[N_signal] -> [N_CCM_timout]controller;
57. cfm_in[C_defects] -> [C_report]controller;
```

Figure 5.12: Continued Click description of the CFM design

5.7 Results

The ShapeUp tool suite was used to implement both the Y.1731 design and the described CFM design. These reference designs both contained numerous modules, side connections, and a software control interface. The Pop design environment and visualizer were used to enter the description.

Interface type matching was straightforward since G uses the same ShapeUp interface type abstractions. When the modules were compiled from G, the compiler generated interfaces that matched supplied interface metadata. For the modules written in G (lines 3 to 9), EDL descriptions of the interface metadata were already available, being part of the standard G development flow. For example, all the G modules used a 64-bit data width at their streaming type interfaces. For the other modules, EDL descriptions were created separately prior to making the module collection available in a repository. Note that the modules declared at lines 14 to 16 were on-FPGA memory blocks containing lookup information, and so just had standard Xilinx BlockRAM read and write interfaces.

The ShapeUp linker was used to generate complete VHDL system descriptions for this example (and any other desired system configurations), including wiring to implement the connections specified in the Click descriptions. In this case, 1570 additional lines of VHDL were generated automatically. This code was exactly as would have been written in an efficient hand implementation – most of the wiring was not subtle, just detailed and tedious for a human to undertake. Targeted at a Xilinx Virtex-5 LXT device, the resulting system occupied 4126 slices, though this area stems from the chosen modules, the linker just adding necessary wiring area. Of these, 348 slices were used for the three timing modules, which is 8% of the total. This version of the reference design supported Ethernet OAM operating at up to a 25 Gb/sec line rate, providing hardware acceleration that allowed 1024 flows in both directions, each one with a 3.3 ms inter-CCM rate. The final system clock frequency was 125 MHz, which was in line with the minimum individual module frequency, indicating that no time overhead was introduced by the automated linking.

The ShapeUp validator was extensively used during the validation of this example, using exactly the same test data files for each implementation level. For the highest level of simulation, a G-specific simulator was used to model the modules written in G, and simple Perl scripts were adequate to model all of the other modules. For the lowest level of validation, this particular example was run on a Xilinx ML505

development board, handling real traffic received and transmitted over a standard external Ethernet interface.

5.8 Summary

Overall, use of the ShapeUp methodology and tool suite proved beneficial in terms of developing and evolving a very modular reference design that tracked rapidly emerging standardization in the telecommunications industry. Ethernet OAM is an increasingly important standard in carrier Ethernet. This case study lead to a single ShapeUp description of the system, which previously would have been described using separate tools.

This chapter described the validation of the overall ShapeUp framework and the timing modules. The main contributions of the case study described in this chapter are the following:

- A demanding real-life example is presented that is relevant to network service providers for monitoring network functionality and performance. This is described in Section 5.1.
- A thorough analysis of complex timing needs of OAM protocols is presented in Section 5.2. This is demonstrated by productive use of activity model and mapping to timing modules.
- A high-level approach is carried throughout the programming methodology and framework, combining ShapeUp and G within the methodology. This is described in Section 5.4.
- Non-trivial Click descriptions (Y.1731 and CFM) were entered and processed with ShapeUp tools. The results were flexible and maintainable designs, delivering required hardware performance. These are described in Section 5.5.

Chapter 6

Dynamic Modular Systems with Adaptable Behavior

This chapter introduces ReShape, which builds on the design approach of ShapeUp and carries through to support system reconfiguration during operation. This setting allows system reconfiguration at the module level, by supporting type checking of replacement modules and by managing the overall system implementation, via metadata associated with its FPGA floorplan. The methodology and tools have been implemented in a prototype for a broad domain-specific setting – networking systems – and have been validated on real telecommunications design projects. The development of ReShape required fundamental extensions to ShapeUp in order to allow fluidity of modules within adaptive and reactive systems. Support for system modification was focused on to allowing the *substitution* of modules within the system during its operation.

The main underpinning for implementing this capability is using dynamic partial reconfiguration of an FPGA to effect the substitution of a module. In essence, the hardware module is 'hot swapped' in a working system. This use model for partial reconfiguration is somewhat different from the widespread use model over the past 15 years or more. Historically, small FPGA devices meant that programmable logic was a scarce resource. Thus research has largely focused on a task-based use model, where an operating system or other run-time system manages a collection of tasks that are sharing the resource over time, for example the work of: Hadley and Hutchings [58], Brebner [60], Diessel and ElGindy [61], and Walder and Platzner [122]. In many cases, these tasks are assumed independent; in some cases,

infrastructure for communication between tasks is provided, for example Majer et al. [63].

Nowadays, FPGAs are significantly larger, so very substantial systems can be implemented without the need to conceive of resource sharing. This has tracked the earlier evolution of memory in computer systems, from shared scarce resource to abundant resource. So, analogously, the headline issue for FPGAs today is more *management* of the resource rather than sharing of it. The ReShape approach is pioneering this view, adopting a system-based use model for partial reconfiguration.

The central technical challenge to enabling the system-based use model is allowing the user to work in terms of a system and its inter-connected modules, while the implementation involves partial reconfiguration of an FPGA that works in terms of modifying physical regions of the device. That is, it is necessary to bridge between a higher-level logical description view and a lower-level physical implementation view, and – importantly – without blighting system performance and/or resource use.

Given the characteristics of current physical design tools for FPGAs and, in particular, the tools supporting partial reconfiguration, the basic solution to the technical challenge is the use of *floorplanning*. Floorplanning provides the means for mapping system modules to distinct physical FPGA regions. Traditionally, doing manual floorplanning is the realm of the FPGA expert, whereas achieving the ReShape user experience goal requires automation of floorplanning.

Section 6.1 presents an introduction to the new Xilinx partial reconfiguration design flow, which is based on using partitions. Section 6.2 presents experiments on internal fragmentation and the floorplanned PR methodology. Section 6.3 presents the ReShape floorplanning algorithm for networking systems, which performs low-level device specific floorplanning, introduced earlier in Section 2.3.4. Section 6.4 presents the design flow for ReShape, which extends the ShapeUp design flow by adding support for dynamic changes to Click descriptions.

The approach is to use a *domain-specific* system floorplanning component in the tool flow, so that system-level module interconnection characteristics can be taken into account. In the ReShape prototype, this floorplanning was tailored towards the system architectures typically used for high-speed networking: pipelines connected by wide data paths. The prior floorplanning research provided great insight to the general capabilities of the domain-specific floorplanning component, in terms of targeting contemporary FPGA architectures with heterogeneous resources and with certain obstacles to regularity.

6.1 Partial Reconfiguration Design Flow

This work was targeted at the Xilinx partial reconfiguration (PR) mechanisms, using the latest tool support for PR in the ISE design suite version 13.1 [123]. The general front-end modular system concepts though would be applicable to other open tools for PR, for example the work of Suris et al. [67] [124] and by Koch et al. [125] [126]. In order to explain fully how the overall ReShape methodology works, some background information on the underlying Xilinx PR methodology is given in this section.

The PR methodology centers on identifying and listing *dynamic* regions, which are the potentially reconfigurable parts of an overall design. The surrounding circuitry, which includes interfaces to dynamic regions and any through wiring across dynamic regions, is treated as the *static* part of the design. Floorplanning is a necessary step in this PR design flow, and this involves using the PlanAhead tool. Some basic knowledge of the FPGA architecture is needed because the floorplanning involves setting up area constraints for where dynamic modules are to reside.

Starting in PlanAhead 12.1, Xilinx introduced a new hierarchical methodology for design preservation. This was enhanced in version 13.1 to support incremental design and team-based design, by introducing a new notion of *partitions*. In terms of the overall tool flow, partitions provide layers to the design, and are used as an

ordering to the design tools. Partitions are created in PlanAhead by selecting one or more netlists and defining a new partition that includes these netlists. A netlist is the low-level output description from the synthesis tool, describing the design as a collection of FPGA primitives connected by wires. During the initial phase of the design tools, placement and routing is performed for a first partition. Then, for the second phase, the first partition results (placement and, optionally, routing results) are imported into the design tools, and then placement and routing for a second partition is run. In general, for the *k*-th phase, the previous (*k*-1) results are imported before placing and routing the *k*-th partition. The benefit of this approach is seen when performing incremental design updates that affect only a single partition, because there is a resulting saving in runtime through not recomputing the placement and routing of the other partitions.

The PR methodology is based on the partition-based approach, and adds a special type of partition called a *reconfigurable partition* (RP). The interconnect for the design includes *partition pins* at the boundary of each RP. These are similar to the 'bus macros' that featured at boundaries in the earlier Xilinx partial reconfiguration flow, but are hidden from the user. The PR flow requires that each RP has an AREA_GROUP constraint to specify the physical resources that belong to the RP. This is where knowledge of the device architecture is necessary. Moreover, the physical floorplanning mandates a different floorplan for each target FPGA, since the available resources and the reconfiguration arrangements vary significantly.

In PlanAhead, an indication of whether or not a project involves partial reconfiguration is specified at project creation. Currently, this requires that the design be a netlist-based project design, created by imported pre-synthesized netlist files. When creating a partition within a partial reconfiguration project, the user indicates whether the partition is reconfigurable or not. The PR flow then involves specifying lists of *reconfigurable modules* (RMs) that are assigned to each of the reconfigurable partitions. This is done individually by selecting an RP and then adding an RM to it. The different RMs are the different choices for populating the

partition. The netlist files for RMs can be omitted at project creation if desired, in which case they are then just represented as unpopulated black boxes.

A key goal of the ReShape methodology is to hide the cumbersome low-level details of the underlying PR flow by providing a system-level front end focused on automatic floorplanning, and then the automatic generation of the required RP and RM data for PlanAhead and the PR flow. It is of course important that this approach does not introduce undue inefficiency either in resource usage or in system performance.

6.2 Internal Fragmentation and the Floorplanned PR Methodology

While the partition-based methodology has user benefits compared with a traditional 'flatten the whole design' approach, a concern is the possible adverse impact of explicit floorplanning of partitions that introduces internal fragmentation within bounding boxes. Therefore, experiments were carried out to assess the interplay between internal fragmentation, performance, and tool run time.

Three sets of experiments investigated the effects of using the partition-based flow for a representative pipelined dataflow design containing equally sized pipeline stages, with each stage being implemented as a separate partition. The interconnection between the pipeline stages had a 512-bit wide data path, as is found in high speed (100 Gb/sec) packet processing applications. The experiments targeted a Xilinx Virtex-6 HX380T FPGA, and used the Xilinx ISE version 13.1 tools running under Windows XP on a 2.8 GHz Intel Core2 Duo T9600 processor with 4GB of RAM.

Existing guidelines for floorplanning with the Xilinx PlanAhead tool recommended only approximately 60% slice utilization within bounding boxes for best results

[127]. Motivated by this advice, the purpose of the first experiment was to investigate the effect of changing the size of the bounding box for each floorplanned stage partition on both the quality of results and the tool run time. The size was varied so that the slice percent utilization, abbreviated here as the PUT, within each bounding box varied between 50% and 80%. Earlier experience reports had indicated that utilizations above 80% generally lead to unsuccessful implementation results. For this experiment, no timing constraints on the implementation were specified.

To introduce additional controlled variability into the experiment, three data points were taken for each PUT (except at 50%), by slightly varying the area constraints for the set of pipeline registers located along the interconnect between each stage. These three variations are shown in Figure 6.1. In the stretched variant, the registers are spaced over the entire inter-stage height; in the centered variant, they are placed more tightly in the inter-stage height; and in the alternating variant, they are placed more tightly higher, then lower, between alternating stages. The choices were practical, being made to consider the possible impact of different positioning of the data path inputs and outputs in each stage.

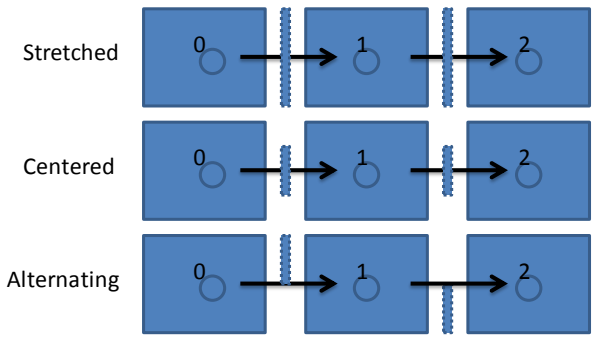

Figure 6.1: Variations in positioning registers on interconnect between stages

The PUT was determined to be an important factor on both quality of results and implementation tool run time. Figure 6.2 shows the clock frequencies achieved, showing an increase as the PUT increases. At first sight, this trend may seem

counter-intuitive because available routing may be more limited when the partition is more densely packed. However, the observation indeed makes sense because the paths within each partition are constrained to become shorter, with consequent benefit for timing. It can be seen that PUT at 75% represented a transition point, with discrimination in timing that favored the stretched register variant.

Figure 6.3 shows the impact of the PUT on tool runtime, showing a trend that run time gets longer for tighter area constraints. A clear threshold between shorter and longer run time appeared at a PUT of 72%. Additionally, the experiments showed that position of the registers on the interconnect between stages caused relatively minor differences in performance, except for PUT at 75%, which straddled the threshold for a more than doubled increase in run time. There, the lowest run time was for the stretched interconnect, which resulted because its placement was less constrained than for the centered or alternating variations. It should be noted that the reported run times were for implementing the complete pipeline system as an initial run. Because of the use of partitions, run time savings are to be expected during incremental design updates affecting single partitions.

To explore these observations further, the experiments were repeated for four-stage and five-stage pipelines, which provided larger and differently shaped designs, and similar results and trends were observed for both performance and tool run time.

The second set of experiments investigated the impact on resource use (in slices) of meeting this timing constraint across all PUT choices. Furthermore, these experiments checked on the effect of different floorplanning choices, by varying the orientation and location of the pipeline design. The two orientations were horizontal and vertical, and the locations were top, middle and bottom (for horizontal) and left and right (for vertical). The middle case for horizontal orientation was the version that was used in the first experiments. The middle case for vertical orientation was omitted due to an obstruction (the configuration block) in the FPGA architecture.

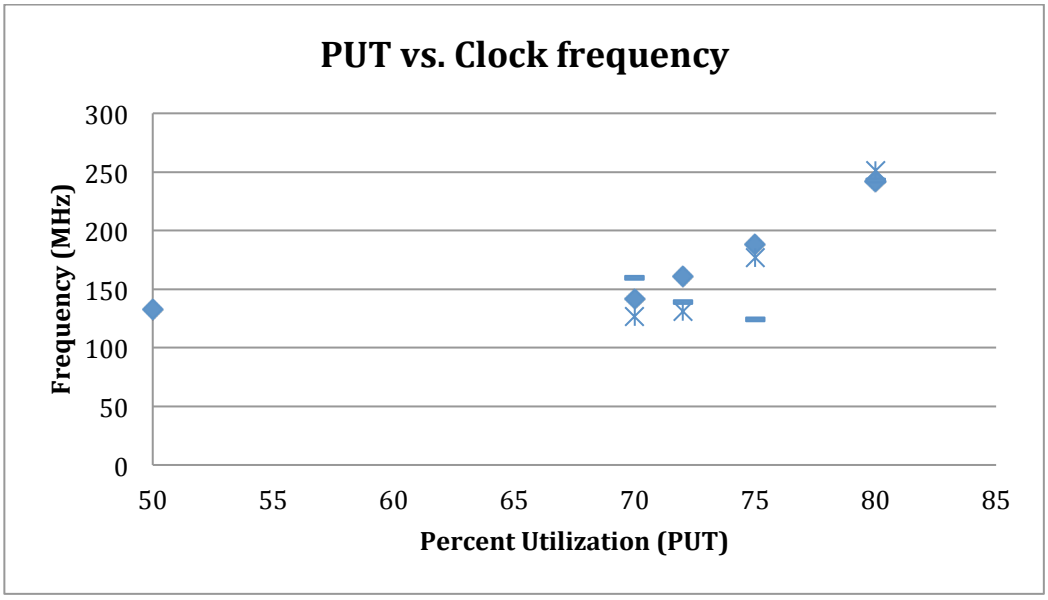

Figure 6.2: Effect of using partitions on clock frequency of implementation

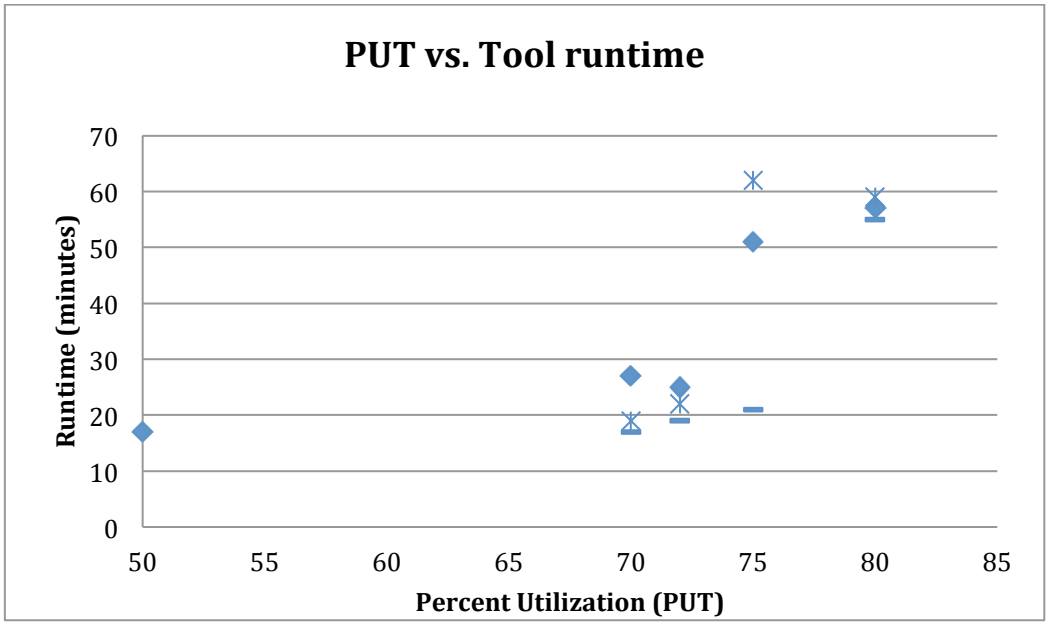

Figure 6.3: Effect of using partitions on implementation tool run time

Figure 6.4 shows the resulting area of the designs for the different floorplans of the three-stage pipeline design at different PUT settings. The variation in slices used was relatively small, which indicated that the total resource use is largely insensitive to position or orientation in the floorplan, and also to the particular choice of PUT at a fixed performance point. Tool run time was not affected by the choice of orientation or location of the pipeline.

The experiments were repeated for four-stage and five-stage pipelines. The total area increased proportionally to the size of the extra stage(s) added, but again showed little variation across the different floorplans and PUT choices.

Taken together, the results of the first two experiments gave insight into the effects of some basic floorplanning choices when working within the partition-based methodology. In particular, a PUT of around 80% emerged as a sweet spot delivering the best clock frequency and total area with acceptable tool run time. This is much more positive than the generic PlanAhead guidance, and indeed halves internal fragmentation. This choice of PUT was incorporated into the ReShape tools.

The third set of experiments was conducted to quantify the actual impact of using both partitions and floorplanning, compared to a more traditional 'flat logic' approach. The figures of merit continued to be clock rate, tool run time, and total area in slices.

Table 6.1 shows the results of these experiments. The three scenarios were: no partitions and no floorplanning; partitions but no floorplanning; and (as in the previous experiments) partitions and floorplanning. Each scenario was tested with and without the specification of an explicit timing constraint. Given that use of partitions and floorplanning is necessary for the PR methodology, the reassuring outcome is that there was no dramatic performance or resource hit. The noticeable impact though is on tool run time. However, as noted earlier, this is the run time for

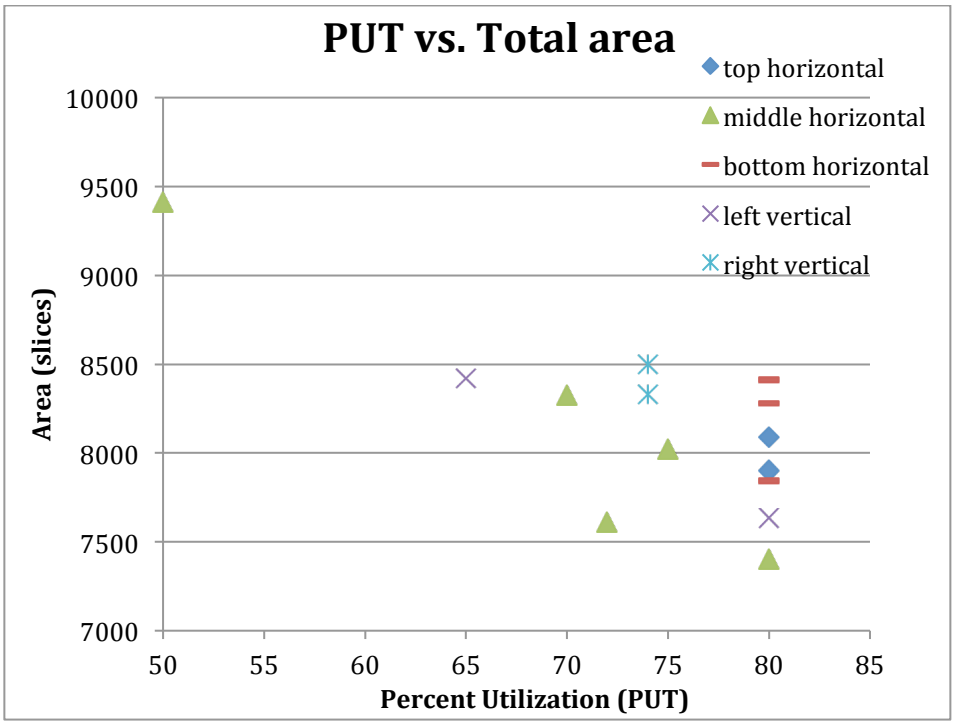

Figure 6.4: Effect of using partitions on total area in slices

the initial implementation of the whole design, and subsequent updates to particular partitions will be faster because there is no need to re-implement other partitions that are unchanged.

Table 6.1: Quality of results, with and without partitions and floorplanning

Partitions	**Floorplanning**	**Timing constraint**	**Frequency (MHz)**	**Run time (mins)**	**Total area (slices)**
No	No	No	271	21	7988
Yes	No	No	246	19	7913
Yes	Yes (80% PUT)	No	245	55	7301
No	No	Yes	240	18	7991
Yes	No	Yes	241	18	7767
Yes	Yes (80% PUT)	Yes	241	42	8164

6.3 ReShape Floorplanning Algorithm for High-speed Networking Systems

In general, automatic floorplanning is a well-known NP-complete problem, which is of course why tools such as Xilinx PlanAhead act primarily as assistants to human users who are determining the actual floorplan choices. However, in order to achieve the ReShape goal of hiding floorplanning details, the automation of the floorplanning task is a key pre-requisite. The solution adopted for the prototype was to adopt a domain-specific approach that takes into account the typical nature of Click descriptions targeted at FPGA implementations. The future plan for ReShape is to allow different domain-specific floorplanners to be included as plug-ins to the overall framework. This is in contrast to adopting a more general-purpose approach of seeking to devise heuristics that tackle the unconstrained floorplanning problem.

Specifically, the ReShape prototype involves constraining the Click system description to be in a form called *Linear Click*. In Linear Click, the structure of any directed sub-graph containing multiple dynamic elements must be a linear chain of elements. The ReShape floorplanning algorithm was then based upon tackling the problem of floorplanning a linear chain of dynamic Click elements. Linear Click proves to be general enough to represent a wide class of pipelined processing systems or subsystems. Notably, in networking and telecommunications applications for FPGAs, there are two main data flow pipelines, for ingress from line side to system side, and for egress from system side to line side. With ReShape, each of the pipelines within a system's architecture can be made dynamically configurable, with other associated system infrastructure, such as memory controllers, being static. The overall networking system, containing a small number (typically between six and eight) of top-level components, is given a crude overall floorplan by the designer in the normal manual way, and then the locations allocated for the individual Linear Click subsystems are input as bounding boxes to the ReShape floorplanner.

The context for the floorplanning algorithm is a model of the target physical FPGA architecture. Detailed models of device architectures are used behind the scenes by the standard FPGA design tools. Rather than seeking access to such internal models, a simplified device architecture model based on openly available Xilinx data was devised. The basic model consisted of a two-dimensional array of Configurable Logic Blocks (CLBs), with specific embedded resource types, such as Block RAM, DSP blocks, and input/output blocks incorporated at their physical positions within the array. Certain other features, such as the reconfiguration controller and PCI Express blocks, were incorporated as anonymous obstacles at their positions.

The floorplanning thus takes into account the heterogeneous FPGA architecture. One subtlety – often overlooked – is that the Xilinx Virtex-6 architecture features two different types of logic slices within CLBs: the SLICEL and the SLICEM. The former are logic-only, while the latter also have memory. There are few SLICEMs in the center of the architecture, and so the treats this area as a further obstruction, so that a balanced density of SLICELs and SLICEMs can be assumed.

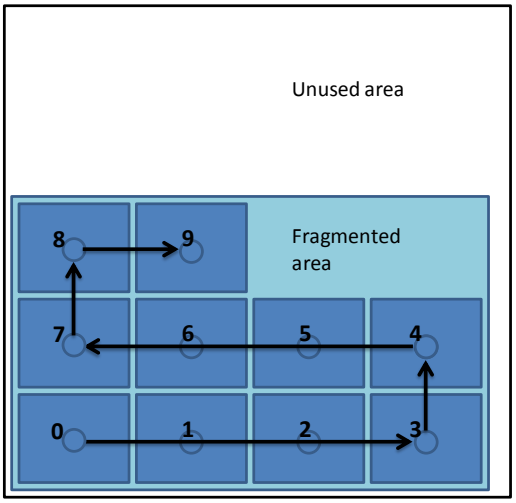

Figure 6.5: Example horizontal zig-zag layout of ten-stage linear pipeline

The floorplanning algorithm places the linearly-connected modules as a set of rectangles on this two-dimensional device model. As its output, it generates area constraints based on the coordinates of the rectangles, and these are then directly used by the standard partition-based partial reconfiguration flow. Specifically, the algorithm places the modules as adjacent blocks within a rectangular area that fits within a specified rectangular region. To do this, the algorithm imposes a zig-zag layout, as illustrated in Figure 6.5. The zig-zag approach first places stages in order along a row (alternatively: along a column, depending on an overall orientation choice). When a module reaches the boundary of the region, there is a reversal of direction, and that stage begins a new row (alternatively: column), running in the opposite direction. In cases where a module requires Block RAM or other specialist resources, or placement in a separate reconfigurable area, the algorithm places the module at the next available specialist region. An overall goal is to minimize *external fragmentation* in the rectangular pipeline layout, as shown in Figure 6.5. The unused area outside this rectangle is not wasted, being made available for other system uses.

The overall floorplanning algorithm is best explained in two steps. The inner step, PlacePipeline, takes a list of pipeline stages, with a rectangular bounding box given for each stage, and then applies the zig-zag placement algorithm. The other inputs are an orientation – whether the layout is to be horizontal or vertical – and the initial pipeline direction along that orientation – right or left for horizontal, up or down for vertical. The algorithm returns the list of pipeline stages, with the placed coordinates of the bounding box for each stage. It also returns the percentage of fragmented area within the enclosing rectangle for the placed pipeline, as a measure of goodness of the layout. Pseudo-code for this algorithm is given in Algorithm 6.1.

One particular concern is the handling of obstacles. Two approaches are used: either stretching one module over the obstacle to obtain sufficient resource density, or adding wiring between two modules to span the obstacle, the choice depending on the current pipeline layout status and the extent of the obstacle. In the latter case, a

concern is that there may be routing difficulties when non-trivial inter-stage wiring is needed. Given the nature of the FPGA architecture, this problem can be more acute for vertical pipelines because most obstacles are higher than they are wide.

Algorithm 6.1 PlacePipeline

Input: List of bounding boxes of pipeline stages, orientation, initial direction
Outputs: List of coordinates of placed pipeline stages, fragmentation percentage of layout

```
if empty list return SUCCESS;

// Based on orientation and direction, determine proposed coordinates for first stage in list
// If stage has specific resource requirements or configuration requirements, adjust coordinates
// Check for overlap of draft coordinates with obstacles and skip over them if necessary
if bounding box exceeds boundary of orientation in current direction
    // This stage is too wide for a whole row (or too high for a whole column), so fail
    if no stages yet placed in current row or column return FAILURE;
    // Otherwise, reverse direction in zig-zag
    PlacePipeline (list, orientation, reverse (initial direction));
else
    // Save coordinates of this stage, and update overall bounding box for pipeline
    // Place rest of pipeline recursively, continuing in same direction
    PlacePipeline (tail (list), orientation, initial direction);
```

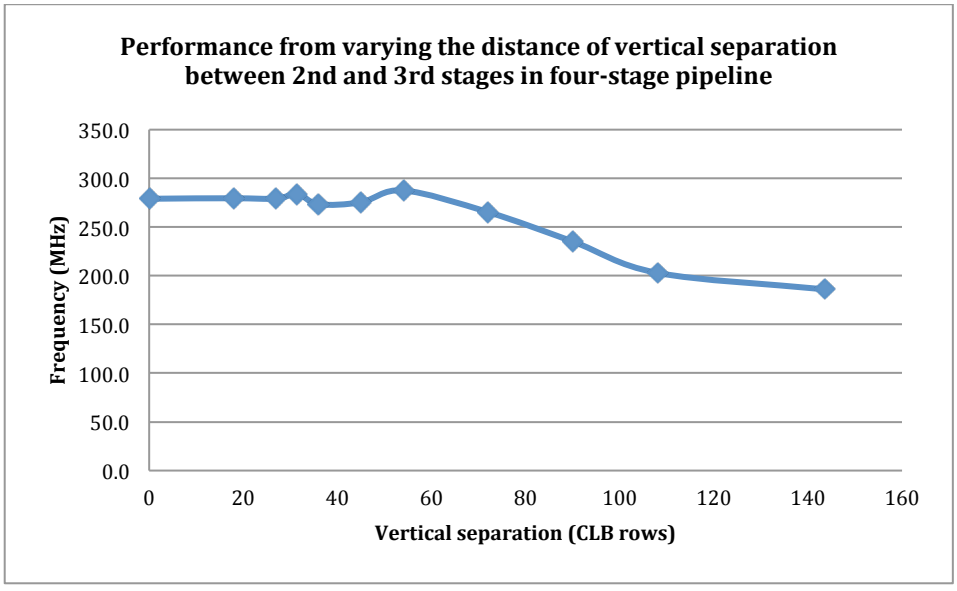

Figure 6.6: Performance vs. vertical separation between stages

To assess this concern, an experiment was carried out to ascertain the performance impact of vertical separation to avoid obstacles. For this experiment, a vertical 512-bit wide data pipeline consisting of four stages, each with a square bounding box, was created. Then the distance of separation between the bottom two abutting blocks and the upper two abutting blocks was varied, in order to investigate the impact on the overall performance. The target FPGA was a Xilinx Virtex-6 HX380T device, which has 360 CLB rows in its architecture. The results of the experiment are shown in Figure 6.6. It can be seen that performance was unaffected up to 50 rows of separation. Thereafter, there was a steady decline in the overall performance. To calibrate the significance of the number of rows, note that a clock region, and a minimum-height reconfigurable partition, is 40 rows high, and so within the unaffected range. The highest obstruction on this FPGA is in fact a configuration memory center block that is 80 rows high. The results of this experiment assisted in implementing additional heuristics in the pipeline layout algorithm. These concerned making a good compromise between skipping over obstacles without performance penalty and deciding that a particular obstacle rendered a particular layout unviable by forming too large an obstruction.

The overall floorplanning algorithm involves calling Algorithm 6.1 repeatedly with different combinations of pipeline stage bounding boxes and different pipeline orientations. The pseudo-code for this enclosing algorithm is given in Algorithm 6.2.

The goal of Algorithm 6.2 is to find a pipeline floorplan that involves the smallest percentage of area lost to external fragmentation within the rectangular bounding box for the floorplan (as illustrated in Figure 6.5). Note that all of the candidate pipeline layouts have the same internal fragmentation within stages: a PUT of 80% was used for defining stage bounding boxes in line with the discussion in Section 6.2.

The outer loop of Algorithm 6.2 tests both horizontal and vertical orientations for the pipeline. The experiments of Section 6.2 had indicated that there was little total area

Algorithm 6.2 Overall floorplanning of Linear Click pipeline

Input: Set of sizes (slices) of pipeline stages, and coordinates of bounding rectangle for layout
Output: List of area constraints for placed pipeline stages, with minimally fragmented layout

```
// Form histogram of sizes of the stages and sort into bins
// Determine candidate set of aspect ratios for pipeline stages
for each pipeline orientation: horizontal or vertical
    for each combination of pipeline stage aspect ratios drawn from candidate set
        // Place pipeline using zig-zag approach, by calling PlacePipeline (Algorithm 1)
        PlacePipeline (list of stage bounding boxes, orientation, direction forward);
        // If successful, insert results into list sorted by percentage of fragmented area

// Select the result with minimum fragmented area (if none, then fail completely)
// Generate area constraints based on selected pipeline placement
```

and performance difference between different positions and orientations of pipeline floorplans in this domain, and so these candidate orientations provided differentiation based on their respective external fragmentation scores. The inner loop involves choosing different combinations of different layouts for the individual stages.

Potentially, there are a huge number of possible candidates for the inner loop to explore, and so this search space was constrained in two ways. First, a limited range of different bounding box aspect ratios was considered for each pipeline stage. Second, a limited number of combinations of stage layouts were considered for the overall pipeline. The approach to choosing the range of aspect ratios and for bounding the number of combinations was based upon two experiments customized to the particular domain-specific setting of this pipeline floorplanning algorithm.

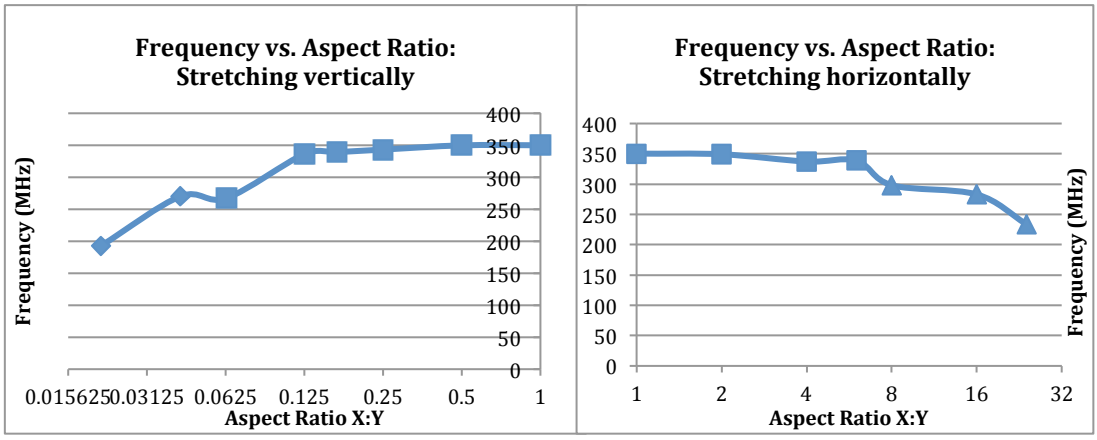

Figure 6.7: Performance vs. aspect ratio, stretching vertically and horizontally

Figure 6.7 shows a summary of the results of this experiment. The left-hand side shows the impact on performance of vertical stretching, where the aspect ratio has width less than or equal to the height. At the extremities of this stretching, the data points marked by a diamond indicate that a purely horizontal layout resulted because of the large stage heights. At the other points, the normal two-dimensional zig-zag layout could be used. The right-hand side shows the impact of horizontal stretching, where the aspect ratio has width greater than or equal to the height. At the extremities of this stretching, the points marked by a triangle indicate that a purely vertical layout was necessary because of the large stage widths. It can be seen that, stretching vertically, there was a significance performance decrease beginning after the 1:8 aspect ratio. Stretching horizontally, there was a performance decrease at the 6:1 aspect ratio. The reason for the decrease in performance was further investigated in PlanAhead, and the CLB metrics showed that routing congestion significantly increased in each dimension as stretching increased along that dimension. As an example, the highlighted regions in Figure 6.8 show the great difference in vertical routing congestion between 1:4 and 1:48 aspect ratios for vertical stretching.

Since the experiments indicated that high performance was maintained for the mid range of aspect ratios, the following set of six aspect ratios was chosen as a configuration for the automatic floorplanning exploration: {0.125, 0.25, 0.5, 1, 2, 4}.

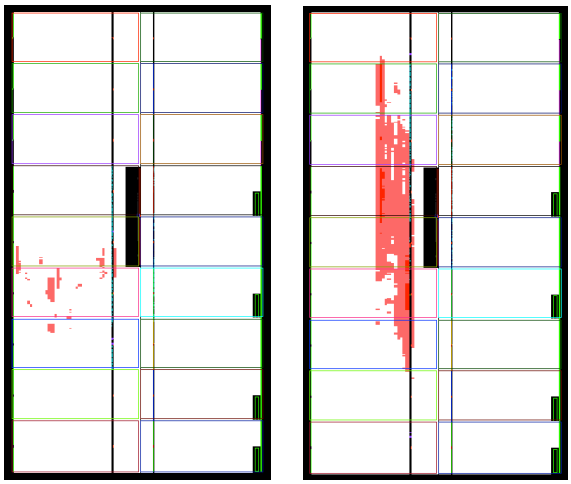

Figure 6.8: Vertical routing congestion: (a) ratio 1:4, (b) ratio 1:48

It is not feasible to try all combinations of aspect ratios for each stage, because this approach does not scale well as the number of stages increases. Specifically, the number of combinations is exponential given by r^n, where r is the number of aspect ratios and n is the number of stages, i.e., 6^n for the chosen set of six aspect ratios. Instead, Algorithm 6.2 groups the stages by similarity of size into a smaller number of size bins. The number of aspect ratio combinations is still exponential, but reduced to r^b, where b is the number of size bins.

In practice, pipelines have a relatively small number of stages and the sizes of the stages are relatively similar, so a small number of bins, for example $b = 2$ or 3, seemed reasonable. However, to check for asymptotic trends under more extreme and synthetic conditions, the second experiment investigated the impact of the number of bins on four example 21-stage pipelines with randomly generated stage sizes between 1 and 500 slices. This experiment was solely concerned with relative floorplan quality, not the routability of the resulting floorplan. Figure 6.9 shows how the amount of external fragmentation, as measured by the relative size of the fragmented area in the best floorplan generated by Algorithm 6.2, decreased as the number of size bins increases. Based on this experiment, seven bins were used as a

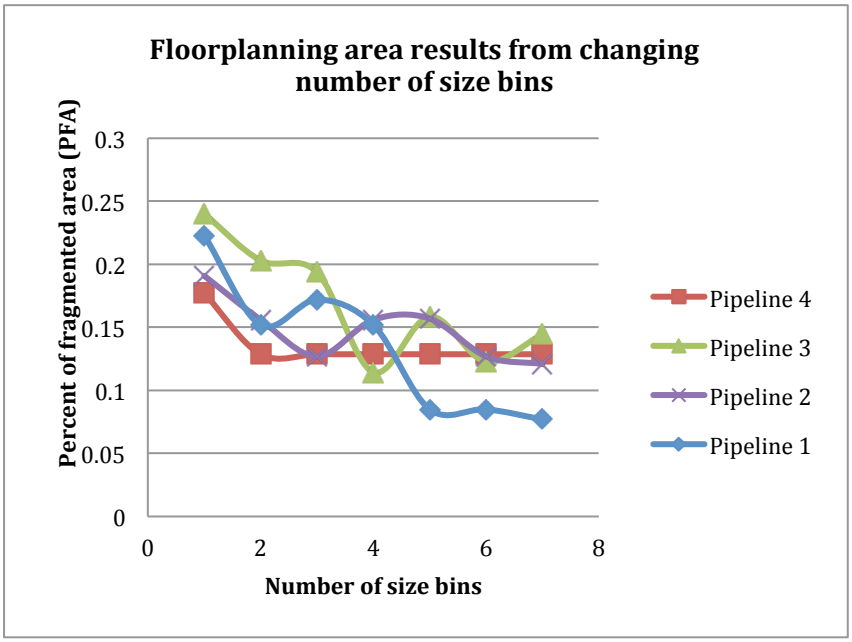

Figure 6.9: Minimizing the area of pipeline designs by adding size bins

more conservative choice for the automated floorplanning exploration, which meant that $6^7 = 279{,}936$ combinations of aspect ratios are checked by Algorithm 6.2. The run time for the algorithm with this setting was under one minute for a 21-stage pipeline.

It is important to note that these two experiments were conducted in order to guide heuristic choices that bound the search space of Algorithm 6.2, and the first experiment in particular does not necessarily give guidance on desirable aspect ratios for blocks in general. A less domain-specific study of the impact of aspect ratio was carried out by Kalte et al. [128], although this was confined to stretching in a single dimension. In general, this indicated that, for smaller designs, extreme vertical stretching (in fact, extreme horizontal compression) had a negative impact on performance and power consumption, but had a negligible impact on larger designs.

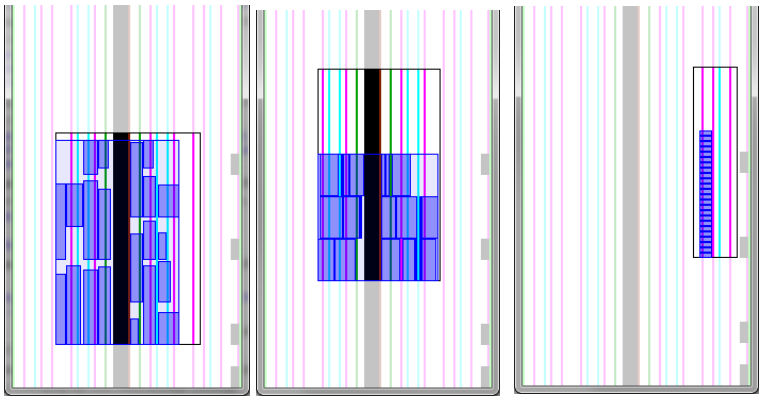

Figure 6.10: Three example floorplanned designs targeting Virtex-6

There are of course also other factors that could feed into floorplan exploration, further complicating the search space. For example, Carver et al. [129] showed that the algorithmic placement of the bus macros used in the old Xilinx PR flow had significant impact on performance, whereas this work relied on the inbuilt quality of placement of the hidden partition pins in the new Xilinx PR flow.

Figure 6.10 shows three examples of generated floorplans, targeting the Xilinx Virtex-6 380 HXT FPGA. The first two designs were 21-stage pipelines, and were given vertical and horizontal orientations respectively, each following a zig-zag pattern. The third design is a 24-stage pipeline that required memory at every stage, and so it was given a vertical orientation and placed along a BRAM column.

Figure 6.11 shows the floorplanning process in action for a five-stage 512-bit wide packet parsing pipeline example that will be introduced in detail in the case study of Chapter 7. The pipeline has a horizontal zig-zag orientation, as follows: first stage at the bottom left, second stage at the bottom right, third stage at the middle right, fourth stage at the middle left, and fifth stage at the top left. First, Figure 6.11(a) shows a visualization generated by the floorplanner after Algorithm 6.2 had been applied. Notably, this shows a central obstruction, in black. This region corresponded to a combination of a configuration block real obstruction and a sparse

SLICEM virtual obstruction (as discussed at the beginning of this section). Because of this, the first and second stages, and the third and fourth stages, have interconnections that span this obstruction. The floorplanner generates area constraints in a UCF file for PlanAhead, and Figure 6.11(b) shows the imported floorplan in a PlanAhead view. Finally, Figure 6.11(c) shows the placed and routed pipeline in an FPGA Editor view. The stages are shown in alternating shades for clarity, and have 80% slice utilization. Note the inter-stage interconnection wiring, shown in the darkest shade. The floorplanning algorithm positions the stages only, since they are dynamic regions for partial reconfiguration and so have to be within known bounding boxes. The interconnection is part of the static region and does not have to be floorplanned explicitly, though of course its good placement and routing by the standard tools is important to the performance of the pipeline.

One final detail, reflected in this example, is that Xilinx partial reconfiguration is performed in units of 'frames' which are of 40x1 CLB size on the Virtex-6 FPGA, and that reconfigurable partitions must not share frames. Therefore, Algorithm 6.2 takes these frame boundaries into account as an additional factor. This can be seen in Figure 6.11(b), where the bounding boxes for the stages are aligned with the horizontal lines that denote 40x1 CLB clock regions on the FPGA.

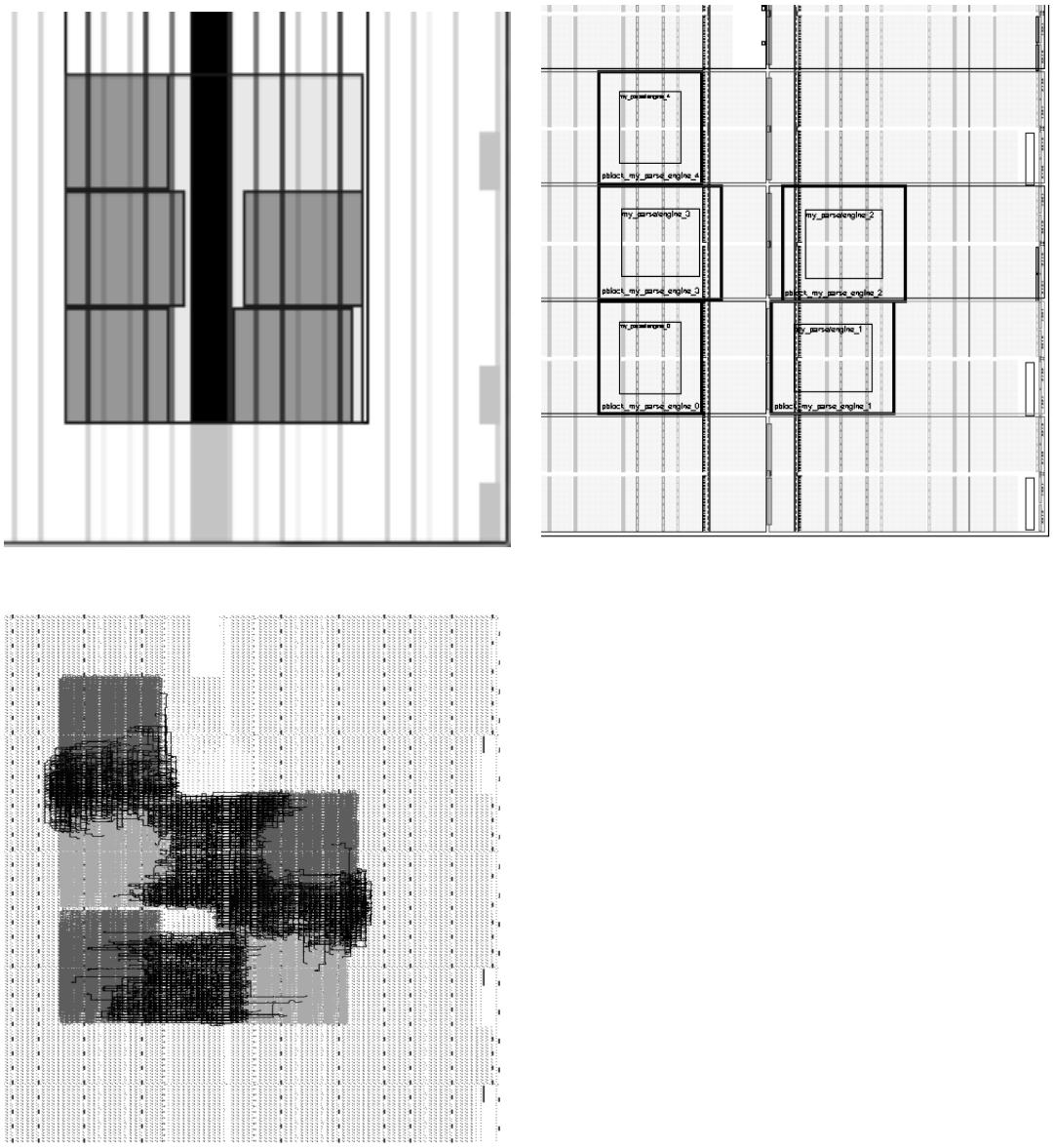

Figure 6.11: Five-stage pipeline layout: (a) Floorplanner, (b) PlanAhead, (c) FPGA Editor

6.4 ReShape design tools

The original ShapeUp tools support a modular design time methodology based on high-level Click descriptions. The central purpose of the ReShape extensions is to support the *minimally intrusive* updating of systems in operation. Clearly, with only the ShapeUp tools described in Chapter 3, it is possible to update systems over time, by just creating a new implementation of the complete system and then loading it by completely reconfiguring an FPGA. The contribution of ReShape is to enable selective change, through partial reconfiguration of the FPGA, reducing the time needed for updates and also allowing uninterrupted operation of the overall system during updating of particular components.

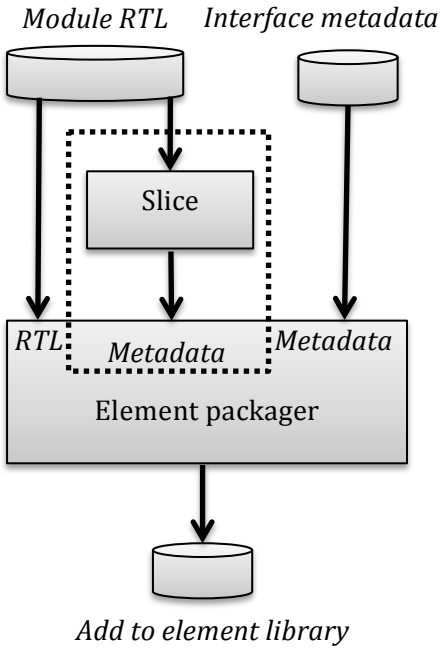

Figure 6.12: Click element packaging

Figure 6.12 and Figure 6.13 show the two parts of the ShapeUp design flow, described in Chapter 3, with specific additions for ReShape shown enclosed within

dotted boxes. Figure 6.12 shows the process of transforming an RTL (e.g. Verilog or VHDL) description of a module into a Click element within the library used by the Click-based system implementation flow shown in Figure 6.13. An element packager is used to associate metadata with the module. In the original ShapeUp flow, this metadata is supplied by the user, and describes the characteristics of the module's interfaces, as outlined in Section 3.3. In the extended ReShape flow, additional metadata is included to describe the resource use (in slices) of the module. This information is obtained by synthesizing the module and then processing it with the Xilinx Map tool, for one or more target devices. An estimate of slice use can be obtained with lighter weight tool use, either through PlanAhead resource estimation based on the RTL or through synthesis-only estimation of LUT/FF use, but Map gives a more accurate result.

Figure 6.13 shows the ShapeUp design methodology, including tools for entering, checking, and validating Click system descriptions, and the stitching tool for generating RTL descriptions of the wiring for connecting elements together. The new feature for ReShape is a domain-specific floorplanner, incorporating the floorplanning algorithm discussed in Section 6.3. This inclusion of a floorplanning step for partial reconfiguration is similar to that seen in the ReCoBus design flow [125] for example. The floorplanner reads in a Click description and element metadata, and it outputs a set of physical placement constraints for the modules, e.g. in UCF format.

The other new feature is the retention of system information for later use when updating the system over time. There are two types of new metadata:

- Results from the PlanAhead/ISE tool flow: information about partitions and their implementation.
- Results from the floorplanner: locations and size of bounding boxes for each module.

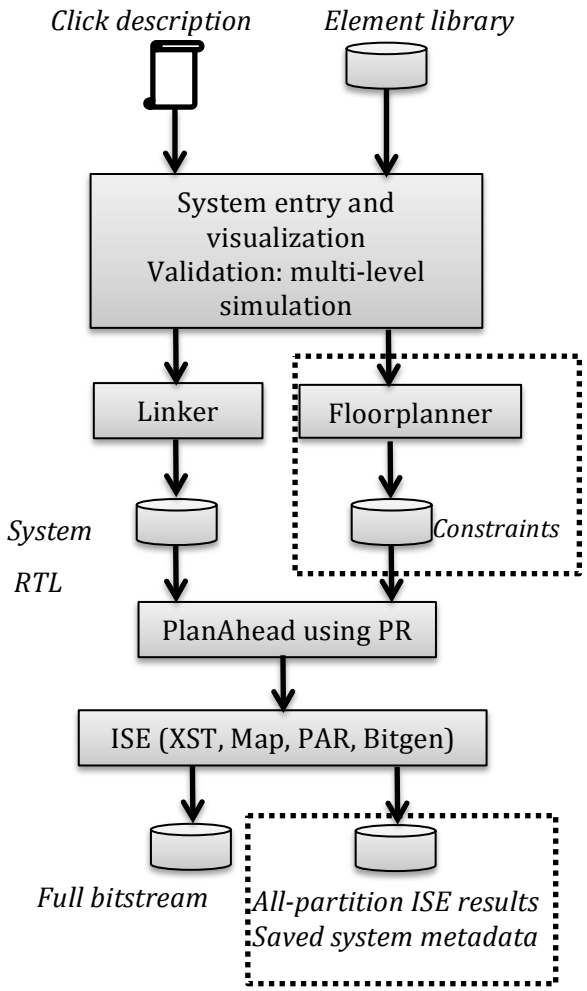

Figure 6.13: Full system implementation flow

Note that there are in fact some potential benefits of introducing floorplanning for the ShapeUp methodology alone, through introducing predictability into the implementation. These include the ability to provide system performance guarantees directly derived from the performance of individual elements, and also to allow higher-level debugging in terms of individual elements.

One important question concerned whether extensions to the Click syntax or semantics would be necessary in order to support the desired ReShape methodology. Here, there was no inspiration from the traditional software version of Click, which

does not have the notion of dynamic updating of only selected parts of the system. Dynamic changes to Click descriptions are realized by hot swapping the entire system. This includes preserving state, by moving any in-transit packets from the old version of the system to the new version. In short, the slower-speed and less time-critical Click systems implemented in software had not provided motivation for considering partial updating of systems in operation.

In fact, no extensions to Click are strictly necessary in order to enable the basic ReShape methodology. The use model is that there are successive versions of a Click description as a system evolves over time. The ReShape tools can analyze the differences between two versions in order to ascertain whether partial updating is feasible, or whether complete system reimplementation and full reconfiguration is required. The details can be entirely hidden from the user, except as reflected by differences in the observed implementation and configuration time. In the ReShape prototype, there are two requirements for partial updating to be feasible:

- The structure of the Click graph – elements and connections – is unchanged;
- Any substitute elements are both interface and floorplan compatible with their predecessor elements.

Slackening of the first requirement is a topic of future research, and centers around supporting dynamic system structures: adding or removing elements, and adding or removing connections. One approach could be to harness past research on task-based reconfiguration, treating elements as a collection of resident tasks. This must ensure that the direct connections of Click system architectures are efficiently mapped to any generic inter-task communication harness. Policing, and then acting upon, the second requirement has been the main focus of the initial ReShape tool effort.

To support arbitrary substitution of Click elements, each element must be mapped to a floorplanned module. For cases where it is known that complete flexibility is not

required, ReShape allows the user to add information to the Click description in two different ways, to reflect two varieties of less dynamic systems:

- *Specific element substitution.* Here, the default is reversed: elements are not dynamic, and the user indicates explicitly which elements are. This can be done without any change to the Click syntax. Any element declaration can include a configuration string that specifies parameters for that particular element instance. The semantic addition is to support the interpretation of an extra keyword `DYNAMIC` in configuration strings of dynamic elements, for example:

  ```
  ccm_reader :: FrameReader(TYPE VHDL, DYNAMIC);
  ```

- *Static element selection.* Here, a fixed selection of choices can be specified for a particular element at design time. This is a common approach taken for partially reconfigurable systems with a static collection of module choices. A syntax extension to Click was the cleaner way to express this case, generalizing an element declaration to list the multiple element types allowed, for example:

  ```
  proc :: Proc1 | Proc2 | Proc3 ;
  ```

The practical benefit of these schemes is to facilitate a more optimized implementation, with the non-dynamic parts of the system being considered as a single unchanging static region with a flattened implementation.

Figure 6.13 shows the ReShape implementation flow for updates to the 'system for life'. The main notable features are three tests to ascertain whether an update to the Click system description is amenable to partial reconfiguration, or whether full system re-implementation and reconfiguration is required. The first test checks whether the input Click system graph is isomorphic to the previous version of the graph that is retained as saved system metadata. If the elements or the connections between elements (including the types of the connections), has changed, then the full re-implementation flow is triggered. In ShapeUp, the type checker was concerned

just with checking that two interfaces are compatible, so that a connection can be made between them. For ReShape, this function is extended to provide a check that two elements are compatible, that is, one element can replace another. To do this, it is necessary to check that both elements have completely compatible sets of interfaces, which can be done using the original per-interface type checker.

The second test checks whether substitute elements fit into the existing floorplan of the system. The first test has already covered interface compatibility, so this test just compares the slice count of the substitute element(s) with the slice counts of the existing element(s). The resource use of the substitute element(s) is given by the metadata attached to these elements in the library, and the resource use of the existing element(s) is held as saved system metadata. If the first two tests are passed, the PlanAhead-based PR flow is used to update the reconfigurable partitions for the element(s) being replaced.

After this, the third test checks that the resulting system still meets the performance of the existing system. This performance information is held as saved system metadata. All of the floorplanning metadata, and the ShapeUp module interface metadata, are retained throughout a system's lifetime to allow for updates.

No major changes were made to the existing validator and visualizer. When there is a system update, the validator operates with the distinct versions of the system separately. At present, the prototype does not include any validation of system behavior during reconfiguration, although it is hoped that a later version can benefit from recent research into this topic, for example [130]. A visualizer extension is to indicate whether or not the system under construction is compatible with a previous version of the system, in other words, whether the update will be simple (with partial reconfiguration) or complex (with full implementation).

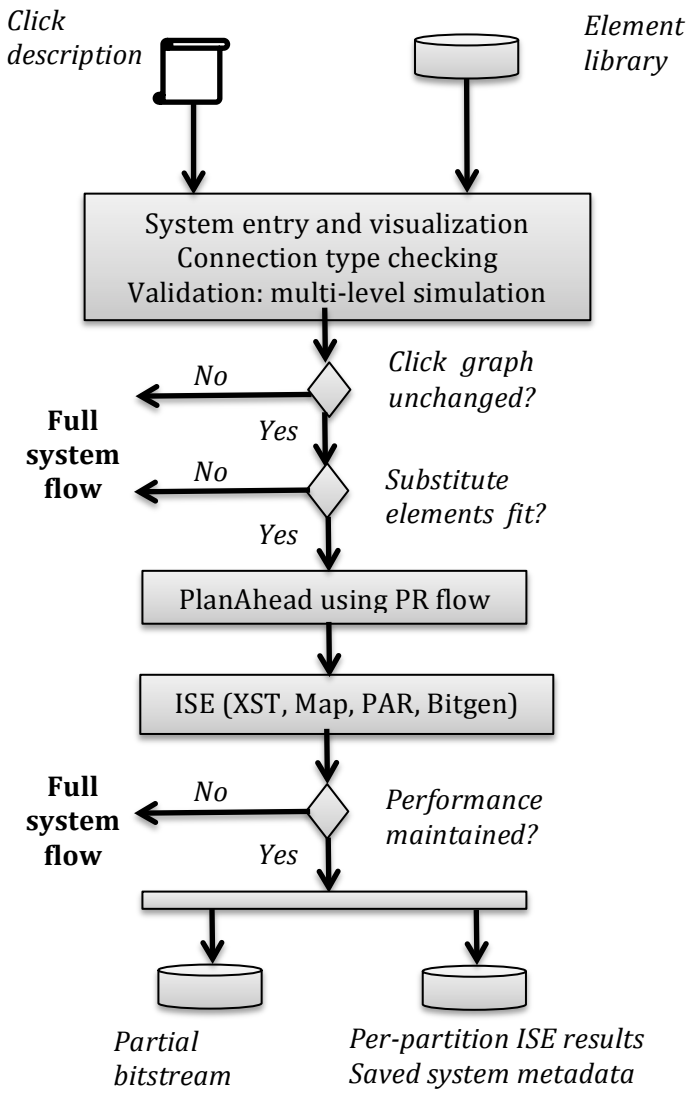

Figure 6.14: ReShape system update implementation flow

The final piece of the ReShape methodology is a tool to perform the partial reconfiguration of the system during operation. This uses the standard Xilinx PR mechanism, rather than a custom runtime system, e.g. [64].

6.5 Summary

Chapter 6 introduced the adaptive systems part of this work, extending the ShapeUp framework to support dynamic modules in an extended methodology called *ReShape*. This model allows: (a) modules to be *substituted* dynamically when the system is in operation, (b) brings benefits of abstraction and modularity to dynamic reconfiguration based on the latest partial reconfiguration (PR) tools, and (c) extends the ShapeUp framework from purely design-time use to lifetime use. A key topic in this work was floorplanning, which physically constrains design placement. This chapter investigated the automatic floorplanning of modules and described experiments measuring the performance of partition-based design flows. This chapter also proposed an algorithm to constrain the placement of modules communicating in a linear pipeline.

The main contributions of this chapter are:

- Section 6.1 presented a concise summary of the new Xilinx partial reconfiguration design flow, which is based on using partitions.
- Section 6.2 presented experiments on internal fragmentation and the floorplanned PR methodology, which guided tactics for the new floorplanner.
- Section 6.3 presented the ReShape floorplanning algorithm for networking designs described in Linear Click, using a zig-zag layout. Experiments on varying the aspect ratio and floorplanning examples provided heuristics for the new floorplanner.
- Section 6.4 presented the design flow for ReShape that supports dynamic changes to the design. This illustrated how the new floorplanner is integrated into the ShapeUp design flow. Additional metadata is stored about the resource use of the modules. Notably, the type checking process is extended to check whether two elements are compatible, and whether a new element can replace the existing one.

The next chapter describes the validation of the ReShape through use in a real-life industrial-strength case study of network processing acceleration.

Chapter 7

Case Study 2: An Adaptive High Performance Network System

The original ShapeUp methodology and now the extended ReShape methodology have been evaluated on a number of real-life, industrial-strength case studies. These have been drawn from the networking and telecommunications area, the application domain within which FPGAs find the greatest application. The aim was to demonstrate that the user productivity gains seen using a higher-level system design approach did not introduce unacceptable losses in quality of results. In particular, the implemented systems had to meet the performance targets for networking functions at data rates ranging between 1 and 200 Gb/sec.

A main case study for the full ReShape methodology involved a programmable packet parsing (PPP) system, required to operate at a data rate of up to 150 Gb/sec on a Xilinx Virtex-6 FPGA. The full specification of the PPP system, and the detailed discussion of a prototype version that did not involve the use of ShapeUp or ReShape, have appeared in an earlier publication by Attig and Brebner [131]. In fact, the results demonstrated a data rate of up to 400 Gb/sec on the most recent PPP version that was targeted at a Xilinx Virtex-7 FPGA. This case study was based on an earlier version, although it is anticipated that it can scale up to the faster version without issues.

Section 7.1 presents the background on the programmable packet parser that was used for this case study. This background section is adapted from [131], with the permission of the authors. Section 7.2 presents ReShape Linear Click examples

describing the PPP. Section 7.3 presents the results of this case study. Section 7.4 summarizes the contributions of this chapter.

7.1 High-speed Programmable Packet Parser

As the Internet evolves, there is a growing need for non-trivial packet parsing at all points in the networking infrastructure, including the core carrier networks. Parsing is central to packet classification in order to identify flows and implement quality of service goals. Increasingly, it is also important to guide deeper packet inspection in order to implement security policies. Of course, packet parsing also continues to have a central role in the implementation of end-to-end communication protocols. With core networks increasing towards 400 Gb/sec rates, packet parsing at line rate poses a major problem. A further complication is that parsing requirements can change frequently as network traffic patterns evolve and protocols are introduced, modified or replaced. This demands dynamic flexibility within networking equipment.

A packet in transit consists of a stack of headers, a data payload, and – optionally – a stack of trailers. At an end system, a packet might begin with a stack of Ethernet, IP and TCP headers, for example. In a core network, a packet might begin with a stack of various Carrier Ethernet or MPLS headers, reflecting en-route encapsulation, for example. The basic parsing problem can be formulated as traversing a stack of headers in order to:

- Extract a key from the stack (e.g., a 16-bit packet type field or a TCP/IP five-tuple); and/or
- Ascertain the position of the data payload (e.g. to enable deeper packet inspection).

The traversal is guided by a parsing algorithm consisting of rules for interpreting different types of header format. Note that, without loss of generality, this approach

can be extended to the parsing of packet trailers, if required. The parsing process must also smoothly handle failures of parsing, indicating unsupported packet forms. The results of parsing feed into other network processing components. These can include key lookup engines for packet classification, and regular expression matching engines for deep packet inspection.

Traditional approaches to providing the required flexibility in packet parsing involve using general purpose servers as a basis for network nodes. However, these may not be capable of providing the required performance. To address this, the combination of general purpose processors and specialized high-performance network processors is possible. However, the increasing specialization of network processors can thwart goals of flexibility and scalability. The Field Programmable Gate Array (FPGA) is an alternative technology that can fulfill the necessary requirements for high-speed concurrent packet processing, and which can be harnessed in tandem with complementary general-purpose processors.

The main goal of the FPGA-based Programmable Packet Parser (PPP) was to achieve packet throughput in the 100s of Gb/sec range, employing a scalable approach that would not require substantial re-engineering with each new step in required throughput. The physical constraints were the amount of programmable logic available on target FPGA devices, and the achievable clock rates for such logic.

The setting involves the streaming of packet data through the PPP system, using a very wide data path, for example, 512 bits wide to achieve a 150 Gb/sec data rate. In some cases, this packet data might just consist of the relevant header part, following payload offload to temporary memory; in other cases, notably initial packet classification, this data is the entire packet. The packet parsing is performed on the fly as the packets stream through. In other words, the module has cut-through operation, rather than store and forward, which would introduce higher packet processing latency. In order to achieve clock frequencies in the desired range, pipelining is deployed extensively.

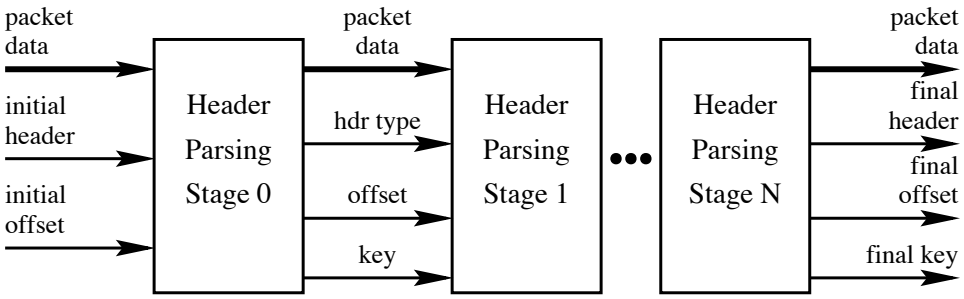

Figure 7.1: Packet parsing pipeline architecture

Figure 7.1 shows the top-level pipelined architecture. There is a natural mapping between the parsing algorithm and the pipeline: one pipeline stage for each level in a packet header stack. As a packet advances through the pipeline, one header is parsed at each stage. In steady state operation, multiple packets are being parsed simultaneously in the pipeline. Each stage has a fixed internal microarchitecture, which has microcoding to provide a degree of programmability when the system is in operation.

When a packet starts to arrive at the input of a header parsing stage, it comes in tandem with the header type identifier, the offset in the data stream, and a key being constructed. The stage microarchitecture has five components. A header type lookup component uses the input header type identifier to fetch customized microcode that programs the remaining components in the stage to be able to handle the particular header type. Meanwhile, the input header offset within the packet stream is forwarded to a locate component that finds the header within the input packet stream. The locate component works in tandem with an extract component that contains customized shifting and masking logic to isolate header fields for use in parsing computations, and key building. A compute component contains customized, heavily pipelined, logic to perform operations associated with the parsing algorithm, including computing the next header and the header size. Results of the compute component can also be forwarded to an optional key builder component that constructs a revised parsing key.

Microcode instructions are used to control the behavior of the five components within each parsing pipeline stage. This allows the same set of resources to be shared for each of the different header types being processed by a stage. The exact microcode format is specific to the set of components contained in a particular stage. The size of the stored microcode depends on the complexity of the components.

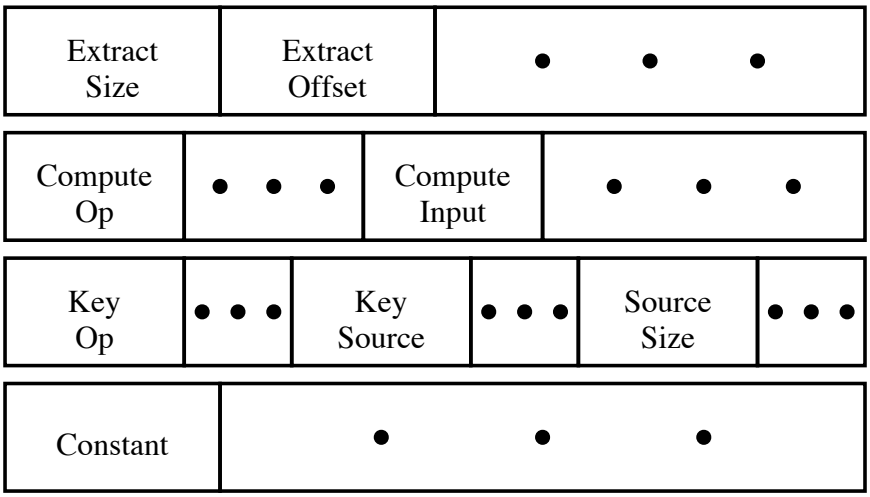

Figure 7.2: Pipeline stage microcode organization

The general format of the microcode is shown in Figure 7.2. It consists of four sections. The first section consists of zero or more extract size-offset pairs. These correspond to different fields that may be extracted from the packet in order to parse a header. The size indicates the bit width of the field, and the offset indicates its bit position from the start of the header segment. The second section consists of compute operations and input selectors. One compute operation entry exists for each stage in a compute unit pipeline. The supported operations are encoded as unique integer identifiers. The compute input selectors program a multiplexer to enable the appropriate inputs to reach a compute unit. Multiplexer inputs could be the different extracted fields or constants from the microcode. The third section consists of zero or more sources for data to be appended to the packet's context key. The final

section consists of constants, occurring in the header object description and then used directly in computations. Constants can be of variable size.

The microcoded PPP system was evaluated using a benchmark suite drawn from examples required in practical networking situations. These fell into two broad categories: carrier (wide area and metro area networks), and end system (access and enterprise settings). In turn, these categories correspond to layer-two and below, and layer-three and above, protocol settings respectively. Experimental results for the FPGA implementation confirmed that packets could be parsed at very high line rates, of 100 Gb/s and higher.

7.2 ReShape Linear Click Descriptions

A key feature of the PPP is that its parsing algorithms must be modifiable at run time, in other words the header parsing stages must be programmable. This is so that a network administrator can make changes to support different types of network traffic as requirements evolve. The existing version of the PPP accommodated this need by including specialized microcoding within the parsing stages, and an interface to update the control stores containing the microcode.

The PPP thus offered a valuable case study for the ReShape methodology. The architecture was well suited for the Linear Click setting, being representative of the packet processing pipeline with wide data path style that is very common in high speed networking implementations on FPGAs. Moreover, it had an essential requirement for modifying the system at run time, and presented the opportunity to compare an approach based on partial reconfiguration with the existing approach based on microcoding.

In ReShape terms, the case study involved the most general use case for dynamic elements. There was no fixed set of pre-defined Click elements because new

elements needed to be created and then incorporated in order to satisfy evolving requirements in the field. So the PPP presented a genuine 'system for life' use case. Three versions of the PPP were used in experiments, to assess not just the benefits of ReShape, but also the advantages or disadvantages of using partial reconfiguration.

The three versions had different degrees of programmability. The first was a hardcoded reference version that did not allow reprogramming after FPGA implementation. The second was the standard microcoded version, which allowed reprogramming within the constraints of the microcode and the internal architecture driven by the microcode. The third was the ReShape version, which allowed the most general reprogramming through the complete change of the parsing stage logic.

The first two versions were modeled with the static ShapeUp methodology. An example Click description of a three-stage instance of the hardcoded version is:

```
FromDevice(MAC) ->
stage0 :: ParseEthernet ->
stage1 :: ParseIPv4orIPv6 ->
stage2 :: ParseTCPorUDP->
ToDevice(MAC);
```

Here the three fixed elements are for parsing an Ethernet header, either an IP version 4 or IP version 6 header, and either a TCP or UDP header, respectively.

An example Click description of a three-stage instance of the microcoded version is:

```
FromDevice(MAC) ->
[S_in] stage0 :: MicrocodedStageSize1 [S_out] ->
[S_in] stage1 :: MicrocodedStageSize2 [S_out] ->
[S_in] stage2 :: MicrocodedStageSize2 [S_out] ->
ToDevice(MAC);
```

```
Controller [A_update0] -> [A_control_store] stage0;
Controller [A_update1] -> [A_control_store] stage1;
Controller [A_update2] -> [A_control_store] stage2;
```

Here, there are two different microcoded elements in the main pipeline, one used at the first stage, the other used at the second and third stages. These reflect different provisioning in terms of the complexity of functions carried out, and the corresponding microcode. The final three connections represent the interfaces used for updating the microcode in the stages, by a controller that writes to the internal control stores in the elements. As in earlier examples, Hungarian notation is used to denote the type of the element ports: "S" for stream and "A" for access.

An example Click description of a three-stage instance of the ReShape version would be exactly the same as that shown for the hardcoded version. This could represent an initial instance. Then, an updated version might be presented to ReShape:

```
FromDevice(MAC) ->
stage0 :: ParseEthernet ->
stage1 :: ParseVLANorIPv4orIPv6 ->
stage2 :: ParseIPv4orIPv6orNull ->
ToDevice(MAC);
```

Here, the parser is being changed to handle an optional Ethernet VLAN header, and to discontinue TCP/UDP header handling. The first stage is unchanged, and the second and third stages can be reconfigured (assuming that they fit of course).

7.3 Experiments and Results

Experiments were carried out using four PPP instances. Designs 1 and 2 contained a three-stage pipeline, and Designs 3 and 4 contained a five-stage pipeline. These instances handled parsing of different combinations of Ethernet, VLAN, IP, and TCP

protocols, exact details of these not being relevant here. The hardcoded versions had separate implementations for the four designs. The microcoded and ReShape versions both had one (three-stage) implementation for Designs 1 and 2 and another (five-stage) implementation for Designs 3 and 4. The implementations were created using the Xilinx ISE tools version 13.2, and targeted a XilinxVirtex-6 HX380T FPGA.

Table 7.1 shows the implementation results from these experiments. The first row for each design shows the overall system resource use and the performance. In the hardcoded versions, the system implementations were flattened, that is, no logical structure was preserved in the physical layout. This allowed global optimization over the whole pipeline, in the traditional manner of placement and routing tools. These results form a baseline in terms of the best possible results for each design. Note however that these hardcoded versions fail the requirement for run-time programmability – unless of course one allows complete re-implementation as a form of programmability, which was not the case in the driving application. All of the hardcoded designs achieved clock rates in excess of 400 MHz, which was considerably in excess of the actual requirement of 300 MHz to satisfy a 150 Gb/sec data rate. It can be seen that the resource use, in terms of both lookup tables (LUTs) and flip-flops (FFs), was directly proportional to the pipeline length.

The overall system implementation results for the microcoded and ReShape versions of the four designs indicate the cost of adding programmability, in terms of resources and performance. The resource use for the ReShape versions takes into account the entire bounding box for the reconfigurable partition for each stage, not just the actual resources used within it. The total bounding box resources are then added to the resources used for the static region to give the numbers in the table.

It can be seen that both versions of each design met the required performance target of 300 MHz. As might be expected, the ReShape versions, with the highest degree of programmability, had the lowest clock rates. The microcoded versions, with a

lesser degree of programmability, were intermediate in clock rate between the ReShape versions and the hardcoded versions. A similar continuum can be seen when considering the use of LUTs, showing the logic price paid for programmability. When considering the use of FFs, the ReShape version is penalized because of flip-flops 'trapped' within bounding boxes, as opposed to being used actively in the implementation. In general the three versions use fairly similar numbers of FFs.

Table 7.1: PPP instances: hardcoded (HC), microcoded (uC), and ReShape (RS) versions

Module	HC LUTs	HC FFs	HC Freq.	uC LUTs	uC FFs	uC Freq.	RS LUTs	RS FFs	RS Freq.
Design1	11,832	14,000	404	14,304	16,101	364	26,165	36,496	354
Stage0	3,833	3,253		4,665	4,281		6,080	12,160	
Stage1	3,167	3,697		6,267	6,740		5,928	11,856	
Stage2	3,810	3,304		4,701	4,387		6,240	12,480	
Design2	11,921	14,023	435	14,304	16,101	364	26,165	36,496	333
Stage0	3,833	3,253		4,665	4,281		6,080	12,160	
Stage1	4,280	3,652		6,267	6,740		5,928	11,856	
Stage2	3,810	3,304		4,701	4,387		6,240	12,480	
Design3	19,166	23,172	411	25,530	28,158	361	38,645	66,531	336
Stage0	3,833	3,253		4,665	4,281		6,080	12,160	
Stage1	4,246	3,718		6,292	6,837		5,928	11,856	
Stage2	4,256	3,725		7,004	7,748		6,240	12,480	
Stage3	4,243	3,707		6,435	7,066		6,552	13,104	
Stage4	3,810	3,304		4,701	4,387		5,928	11,856	
Design4	19,185	23,269	422	25,530	28,158	361	38,645	66,531	329
Stage0	3,833	3,253		4,665	4,281		6,080	12,160	
Stage1	4,261	3,735		6,292	6,837		5,928	11,856	
Stage2	4,272	3,741		7,004	7,748		6,240	12,480	
Stage3	4,257	3,725		6,435	7,066		6,552	13,104	
Stage4	3,810	3,304		4,701	4,387		5,928	11,856	

Table 7.1 also shows the resources used for the individual stages. The stages do not have independent existences in the hardcoded and microcoded versions, but the data allows comparison with the stages in the ReShape versions, which are the independent modules used for reconfiguration. It is interesting to note that the microcoded versions show wide variability between stages compared to the other two versions. This is because each stage was provisioned for a different worst-case programming possibility. In particular, some stages required more resource in the microcoded version than in the ReShape version. The ReShape resource use was compared with the hardcoded resource use in more detail. While there was a large headline increase in LUTs and FFs due to the bounding box effect, the actual increase in utilized resource was ~25% in LUTs and zero in FFs. The LUT increase was because each partition pin is implemented by a LUT1 and there were ~1100 nets crossing in or out of each reconfigurable partition.

The reprogramming times of the microcoded and ReShape approaches were compared for a five-stage PPP instance, and are shown in Table 7.2. In the microcoded version, an update can be done by writing 64-bit words into the control store for a stage. For the ReShape version, the number of reconfiguration frames for each stage was computed, and hence the size of the partial bitstream needed for the update. The ICAP and SelectMAP interfaces for partial reconfiguration are 32-bit and have a maximum frequency of 100 MHz for the Xilinx Virtex-6 architecture, which severely limits the speed of reconfiguration [30]. Note that researchers have successfully run these interfaces at higher rates (for example [132]), but a conservative quantification is used here.

As can be seen from Table 7.2, there is a significant difference between the two approaches in terms of reprogramming time: nanosecond time versus millisecond time. However, this reflects the wide difference in programmability, between the modest tweaks possible with microprogramming and the complete architecture change possible with ReShape. In fact, given that parser updates are likely to be very infrequent, it is not unacceptable that a reconfiguration takes around 1 ms.

Table 7.2: Reprogramming time for microcode and ReShape approaches

Stage	Microcode update data (bits)	Microcode update time (ns)	No. config frames	Partial bitstream size (bytes)	ReShape update time (us)
0	39	3	26	355,104	888
1	57	3	27	366,768	917
2	84	6	29	417,312	1,043
3	82	6	28	378,432	946
4	62	3	26	413,424	1,034

Module RTL source files and metadata were generated using a high-level language compiler for the PP language, described in [131]. Additional metadata, including the area estimates for the FPGA, was determined by running synthesis (XST) and resource mapping (MAP). This metadata was also packaged with the element RTL source, as illustrated in Figure 6.12.

The ReShape design tools were used to create the case study FPGA implementation, as shown in Figure 6.13. The floorplanning tool was used to apply the floorplanning algorithm, described in Section 6.3, to both the three-stage Design 1 and five-stage Design 3 example pipelines. The floorplanner processed the initial Click description and determined the physical placement constraints for each of the modules. The linker was used to create the top-level structural description of the system. The structural description and the placement constraints were loaded into PlanAhead to create a PR implementation of the design. The ISE tools were used to create the full, static bitstream, for configuring the FPGA. The validator was used to simulate the behavior.

The modified Click descriptions for Design 2 and Design 4 were used to perform system updates, as shown in Figure 6.14. The updated Click was processed by the floorplanner, and the Click graph remained unchanged between versions, and the

substitute elements were checked to confirm that they fit. The PlanAhead PR flow was used along with ISE to produce a partial bitstream. (As an alternative example, updating from Design 1 to Design 3 would have had a different number of elements in the Click graph, which would have required using the full system flow.) PlanAhead used the partition-based design flow to implement only the updated partitions in the design, which meant that the previous implementation results for stage 0 and stage 4 were imported.

The ReShape design tools raised the level of abstraction, so that the implementation of the examples did not require the tedious and error prone task of manual floorplanning.

7.4 Summary

Overall, the case study demonstrated the benefits of the ReShape approach, in terms of supporting the 'system for life' model and hiding the low-level details of partial reconfiguration from the user. It was possible to work from a single high-level Click description, using the various tools in the ReShape suite. The implementation results showed that the target performance for a 150 Gb/sec data rate could be achieved using a 512-bit data path at over 300 MHz, how the resource use compared with a non-floorplanned and flattened implementation, and how the reprogramming time compared with a microcoded implementation.

The main contributions of the case study described in this chapter are the following:

- A high-speed real-life system is described in Linear Click. An example of the Programmable Packet Parser supporting dynamic behavior is described in Section 7.2.
- Experiments using four configurations of the Programmable Packet Parser were conducted, comparing the hard-coded, microcoded, and ReShape approaches. A comparison of the results is presented in Section 7.3.

- The case study demonstrated the benefits of the ReShape approach, in terms of supporting the ‘system for life’ model and hiding the low-level details of partial reconfiguration from the user. This is discussed in Section 7.3.

Chapter 8
Conclusions

The ShapeUp methodology is a significant contribution to encouraging a high-level modular approach to designing FPGA-based systems. This is very necessary, given the increasing complexity of such systems. The ShapeUp methodology is founded upon a small set of abstractions of module interface behavior, chosen to be comprehensive yet compact, and principled yet pragmatic. To make these abstractions practical, a metadata format was developed to describe instances of the interface data schemas. This is aligned with the emergent IP-XACT standard. The metadata is then used by a variety of tools, which contribute to a high-level modular design flow. The Click language, with a fundamental generalization of the semantics of connections, is employed for system description. The new configurable timing modules represent one of a core set of generic module libraries that contribute to the overall ShapeUp methodology. Although motivated by the needs of networking, the new configurable timing modules have potential applications in many types of real time embedded systems where there are events and activities that are influenced by the passage of time.

The ReShape methodology extends the modular approach to apply throughout the lifetime of a system. It provides a consistent high-level view that hides the intricacies of using dynamic partial reconfiguration of FPGAs to perform system updates. This is underpinned by automated floorplanning to act as the bridge between the logical system description and the physical FPGA implementation. This automation has been prototyped for Linear Click, a variant that is well suited to the broad domain of networking applications. The ReShape framework is designed to allow other domain-specific floorplanners to be incorporated in the future. The

overall methodology has been successfully applied to a number of real-life case studies involving networking at very high data rates. This confirms that ReShape is not a 'toy' approach, but gives a practical way of hiding low-level detail while not compromising the quality of results unduly. This helps to reinforce the notion of the FPGA as a mainstream programmable technology.

8.1 Main Contributions and Impact

The main contributions of this dissertation are:

- Synthesis of background research results from four different areas: FPGAs, system-level design, dynamic reconfiguration, and networking

- **ShapeUp research:**
  - Contributions:
    - Analysis of inter-module communication, and abstraction of interface behaviors;
    - Definition of data-driven approach using metadata and meta-metadata;
    - Creation of innovative interface type checking algorithm;
    - Building of methodology and tool flow for module-based system design
  - Impacts:
    - New Xilinx product under development for high-level modular design, featuring "plug-n-play" interfaces
    - Two patents:
      - Interface type checking for integrated circuit design
      - Novel graphical interface for the Pop design environment

        - Conference paper at 18th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2010) [133]

- **Timing research:**
    - Contributions:
        - Analysis of time-related functions in computer and networking systems, with particular focus on communication protocols, and abstraction of timing behaviors;
        - Definition of a small set of standard timing modules;
        - Incorporation of modules into ShapeUp methodology;
        - Demonstration on high-speed telecommunications examples
    - Impacts:
        - Customer reference designs for Ethernet OAM, resulting in design wins for Xilinx
        - Conference paper at 20th International Conference on Field Programmable Logic and Applications (FPL 2010) [134]

- **ReShape research:**
    - Contributions:
        - Analysis of behavior of existing partial reconfiguration techniques, and halving of expected internal fragmentation;
        - Creation of innovative floorplanning algorithm tuned for high-speed networking;
        - Building of methodology and tool flow for high-level adaptive system design;
        - Demonstration on high-speed telecommunication examples
    - Impacts:
        - Xilinx customer engagements
        - Patent pending

- Journal paper accepted for the ACM Transactions on Reconfigurable Technology and Systems (TRETS), pending publication
- Floorplanning extension to the IP-XACT standard

In summary, this dissertation has presented several important contributions that encourage a modular, high-level approach to designing FPGA-based networking/streaming systems that also simplifies reconfiguration in order to facilitate adaptive behavior.

8.2 Future work

There are three main directions suggested for future work, which are expanded upon in the discussion following:

1. Incorporating ShapeUp interface abstractions and timing abstractions into tools for module creation from high-level descriptions.
2. Supporting adaptive systems beyond pipelines, and dynamic graph structures.
3. Generalization of mappings between Click elements/connections and modules interconnect.

One step in further raising the level of abstraction is building upon the experimental work of (a) incorporating ShapeUp interface type abstractions into the G language and (b) coupling ShapeUp timing modules with G modules, in order to build the ShapeUp system view into languages for describing system modules, and their compilers. This will lessen the system integration challenges posed to tools like the linker and validator, by providing harmonious modules that are ShapeUp ready.

Another step is generalizing the prototype ReShape capabilities. One aspect is providing alternative plug-in floorplanners that are specialized for important domains. The other aspect is extending the tools to deal with dynamic Click graphs:

where vertices and edges can be added or deleted. This would allow for more radical Click description changes to be handled using partial reconfiguration.

A third step would be to loosen up the strict one-one mapping of system modules and connections to hardware blocks and wiring, which is the setting that this dissertation has focused on. In general, more complex mappings are possible. Individual modules may be mapped to multiple implementation entities, maybe hardware or software, or several modules might be mapped to the same entity. Here, much of prior research has been done, in the general area of hardware-software partitioning. Also, connections might be mapped to more complex underlying entities, for example, networks-on-chip [135]. Here, one can learn much from conventional networking, where the norm is to implement point-to-point connections using complex underlying networking like the Internet.

Appendix A

Ethernet OAM Element Library

This appendix consists of concise descriptions for each of the G library elements contained in the case study design in Chapter 5.

For reference, only partial CCM frames are stored in this implementation, which helps compress CCM data to fit within the available on-chip memory (BRAM). This is important because by using on-chip memory, the OAM designs are able to consume less power than they would by using off-chip memory.

- VlanClassifier.g -

VlanClassifier is an element for classifying incoming frames as either OAM frames or data frames, based on their frame type and whether the frame is determined to be in-profile. The downstream OAM handler requires this classification.

The OAM handling supports the existence of zero or more VLAN tags. The VLAN tag presence is ascertained by inspecting the TPID. The classifiers support two configurable TPID values for VLAN tag identification. The OAM block uses the type/length fields to identify the existence of an OAM frame. It supports two configurable values to identify OAM frames.

The OAM handler maintains contexts for different VLANs. The VLAN ID, contained in the VLAN header is used to lookup the assigned context ID for input frames. For each context, there is a set of values that specify how to perform the in-profile determination step. In-profile determination is used in separating OAM frames and data frames.

The in-profile determination is made based on either the discard eligible (DE) bit or the priority code point (PCP) bits. The selection between DE and PCP bits is configurable on a per MEG basis. The PCP option supports the four standard profiles as defined by Provider Bridging (PB). The profile selection is configurable on a per-MEG basis.

The context information is added to the beginning of the output frame as a Context_t header shim, described in “MyContextFormats.g”.

- Oam_y1731_in.g -

OAM_Y1731_IN is an element for manipulating OAM frames, with Context_t header shims, from the line-side interface to the system-side interface, for the following performance monitoring functions: CCM, LMM, LMR, 1DM, DMM, DMR.

The output CCM frame is manipulated (to make results available for later software handling) by:

• The far-end frame loss result is inserted in the RxFCb field

• The near-end frame loss result is inserted in the TxFCf field

The output LMM frame is manipulated by inserting the following values:

• RxFCf

• The near-end frame loss result is inserted in the TxFCb field

The output LMR frame is manipulated by inserting the following values:

• The near-end frame loss result is inserted in the TxFCb field

• The far-end frame loss result is inserted in the RxFCf field

The output 1DM frame is manipulated by inserting the following value:

• RxTimeStampf

The output DMM frame is manipulated by inserting the following value:

• RxTimeStampf

The output DMR frame is manipulated by inserting the following value:

• RxTimeStampb

All other OAM frames are transparently passed through the service level OAM manipulation components.

- Oam_y1731_out.g -

OAM_Y1731_OUT is an element for manipulating OAM frames, with Context_t header shims, from the system-side interface to the line-side interface, for the following performance monitoring functions: CCM, LMM, LMR, 1DM, DMM, DMR.

The output CCM frame is manipulated by inserting the following values in their appropriate fields:

• TxFCf

• RxFCb

• TxFCb

The output LMM frame is manipulated by inserting the following value:

• TxFCf

The output LMR frame is manipulated by inserting the following values:

• TxFCb

The output 1DM frame is manipulated by inserting the following value:

• TxTimeStampf

The output DMM frame is manipulated by inserting the following value:

• TxTimeStampf

The output DMR frame is manipulated by inserting the following value:
• TxTimeStampb

All other OAM frames are transparently passed through the service level OAM manipulation components.

- GenerateCcm.g -

GenerateCCM is an element for completing CCM frames, after they are read from memory. The partial CCM frame data stored in memory has certain zero-valued fields omitted in order to compress the frames to fit within available BRAM. The incoming frames to this module are partial CCM frames. This element inserts the missing fields and sets the CCM period to be 3.3 ms. The output frames are entire CCM frames.

- CheckCcm.g -

CheckCCM is an element for checking incoming CCM frames for several defect conditions, wherein a defective frame is forwarded to the control processor for further inspection. The potential defect conditions are:

- A. Loss of Continuity
- B. Unexpected MEG level
- C. Unexpected MEG ID
- D. Mismerge (inconsistent MEG ID and MEP ID)
- E. Unexpected Period

A. Loss of continuity is detected and signaled separately by the Finish module
B. If the incoming MEG level is lower than the configured MEG level, then it is marked with "Unexpected MEG Level defect" tag in the defect field of the context shim.
C. Similarly to B, if the MEG ID is different from the stored MEG ID, then the

frame is marked with a "Mismerge defect".
D. Similarly to B, if the MEP ID is different from the stored MEP ID, then the frame is marked with an "Unexpected MEP defect" tag.
E. Similarly to B, if the CCM Period is different from the stored Period, then the frame is marked with an "Unexpected Period defect" tag.

Each of the defect conditions are described in more detail in the Y.1731 specification.

The incoming CCM frames contain a Context_t header shim, added by the local classifier element. The Context_t header shim provides the context ID for the incoming CCM. This element should connect to a reference memory with a table of expected CCM values. Certain fields in the incoming CCM are checked against the expected values, when checking for defects.

The output frames contain an updated Context_t header shim, wherein a CCM containing a defect is marked to indicate the defect type in the updated Context_t header shim.

The defect checking functions described here are implemented in hardware, in order to support a large number of contexts at the 3.3 ms service interval.

- RemoveShim.g -

RemoveShim is an element for removing the Context_t header shim, inserted by the classifier element. The input frame is an OAM or data frame prepended with the Context_t header shim, containing local control information. The Context_t header shim is removed, and the output frames return to their original length.

- CalcAddress.g -

CalcAddress is an element for calculating the address necessary for reading partial CCM frames from memory. The input is a 'notify' frame from the Start element. This contains the activityID (in this design the activityID = contextID). The partial CCM frames for this
version are packed in chunks of 18, 32-bit words.

The outgoing frame is formatted for the downstream FrameReader. The first word of the output frame contains: (a) the address value for the start of the frame and (b) the length of the partial CCM frame to be read.

Appendix B

Example G element description

- CheckCcm.g -

```
/*=========================================*/
/* Ethernet OAM Functions - OAM CFM (CCM.Rx) */
/* cneely                                  */
/*-----------------------------------------*/

element CheckCcm {

  /* ShapeUp interface types */
  input  framein       : stream;
  output frameout      : stream;
  output request        : access;
  output ccmRefs        : access;

#define DEF_UNEXPECTED_MEG_LEV 1
#define DEF_MISMERGE 2
#define DEF_UNEXPECTED_MEP 3
#define DEF_UNEXPECTED_PERIOD 4

  format ContextShim_t =(
    contextValid : bool,
    contextID    : 9,
    isOAMframe   : bool,
    isDefective  : bool,
    defectType   : 3,
    inProfile    : bool
  ); // totals 2-octets

  format EthernetHeader = (
    destAddr : 48,
    srcAddr  : 48,
    tpid     : 16);

  format VlanHeader =(
    userPriority : 3,
    cfi          : 1,
    vlanID       : 12);

  format CcmFlags_t =(
    rdi       : 1,
    reserved  : 4,
    period    : 3);
```

```
format OamHeader =(
  megLevel  : 3,
  version   : 5,
  opcode    : 8,
  flags     : CcmFlags_t,
  tlvOffset : 8);

 format CcmPdu=(
  seqNum  : nat,
  mepID   : 16,
  megID   : 384,
  TxFCf   : nat,
  RxFCb   : nat,
  TxFCb   : nat,
  Reserved : nat,
  endTlv  : 8);

format NotifyFrame =(
  activityID : 16,
  timestamp  : 16,
  event_type : 16,
  reserved   : 16);

format FinishReq = (
  active         : bool,
  relativeFinish : bool,
  activityID     : 12,
  finishTime     : 16);

format CcmFrame =(
  shim    : ContextShim_t,
  ethHdr  : EthernetHeader,
  vlanHdr : VlanHeader,
  etype   : bit[16],
  oamHdr  : OamHeader,
  ccm     : CcmPdu);

format CcmReference =(
  megLevel : 3,
  megID    : 384,
  mepID    : 16,
  period   : 3);


handle CcmFrame on framein {

  /* general vars */
   var myFinish : FinishReq;
   var ref : CcmReference;

  /* Check for the following defect conditions (from ITU-T Y.1731): */
       /*  - If no CCM frames from a peer MEP are received within
        *  the interval equal to 3.5 times the receiving MEP's
        *  CCM transmission period, "loss of continuity" with peer
        *  MEP is detected.
        */

          /* Handle this using timers for every MEG level context.
```

```
         * The defect message (on timeout) is sent directly
         * to the Control processor, bypassing this module.
         */

        /* Our action: on receiving a CCM frame, reset the timer
         * for this context */
          set myFinish.active = true;
          set myFinish.relativeFinish = true;
          set myFinish.activityID = shim.contextID;
          set myFinish.finishTime = 0x70; // 3.5 * CCM period from now
          write myFinish to request;

      set shim.isDefective = false;
      read ref from ccmRefs[shim.contextID];
      /*  - If a CCM frame with a MEG Level lower than the receiving
       *  MEP's MEG Level is received, "Unexpected MEG Level"
       *  is detected.
       */

       [oamHdr.megLevel < ref.megLevel] {
         set shim.isDefective = true;
         set shim.defectType = DEF_UNEXPECTED_MEG_LEV;
       }

       | [oamHdr.megLevel == ref.megLevel] {


          /*  - If a CCM frame with same MEG Level but with a
           *  MEG ID different than the receiving MEP's own MEG ID
           *  is received, "Mismerge" is detected.
           */
           [ccm.megID != ref.megID] {
             set shim.isDefective = true;
             set shim.defectType = DEF_MISMERGE;

           } | {
              /*  - If a CCM frame with the same MEG Level and a correct
               *  MEG ID but with an incorrect MEP ID, including
               *  receiving MEP's own MEP ID, is received,
               *  "Unexpected MEP" is detected.
               */
              [ccm.mepID != ref.mepID] {
                set shim.isDefective = true;
                set shim.defectType = DEF_UNEXPECTED_MEP;
              }
           }
        }
        /*  - If a CCM frame is received with a correct MEG Level,
         * a correct MEG ID, a correct MEP ID, but with a period field
         * value different than the receiving MEP's own CCM transmission
         * period, "Unexpected Period" is detected.
         */

       [oamHdr.flags.period != ref.period] {
                set shim.isDefective = true;
                set shim.defectType = DEF_UNEXPECTED_PERIOD;
       }
/* A receiving MEP must notify the equipment fault management process
```

```
      * when it detects the above defect conditions.
      */

    [shim.isDefective] forward on frameout;
  }
}
```

References

[1] T. Erjavec, "Introducing the Xilinx Targeted Design Platform: Fulfilling the Programmable Imperative," Xilinx, Inc., Whitepaper WP306, 2009.

[2] I. Buck et al., "Brook for GPUs: Stream Computing on Graphics Hardware," *ACM Transactions on Graphics*, vol. 23, pp. 777-786, 2004.

[3] NVIDIA. (2008, June) NVIDIA CUDA Compute Unified Device Architecture Programming Guide. [Online]. http://www.nvidia.com/object/cuda_develop.html

[4] Khronos. (2010, September) OpenCL 1.1 Specification. [Online]. http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf

[5] W. Thies, M. Karczmarek, and S. P. Amarasignhe, "StreamIT: A Language for Streaming Applications," in *Proceedings of the 11th International Conference on Compiler Construction*, vol. 2304, London, 2002, pp. 179-196.

[6] K. Compton and S. Hauck, "Reconfigurable Computing: A Survey of Systems and Software," *ACM Computing Surveys*, pp. 171-210, June 2002.

[7] Xilinx, Inc. (2012, January) Virtex-6 Family Overview. [Online]. http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf

[8] Altera. Altera Stratix 4 device handbook. [Online]. http://www.altera.com/literature/hb/stratix-iv/stratix4_handbook.pdf

[9] Actel. Actel ProASIC3 data sheet. [Online]. http://www.actel.com

[10] Lattice. Lattice ECP3. [Online]. http://www.lattice.com

[11] J. S. Rose, R. J. Francis, P. Chow, and D. Lewis, "The Effect of Logic Block Complexity on Area of Programmable Gate Arrays," in *Proceedings of the IEEE Custom Integrated Circuits Conference (CICC)*, San Diego, 1989, pp. 5.3.1-5.3.5.

[12] E. Ahmed and J. Rose, "The Effect of LUT and Cluster Size on Deep-submicron FPGA Performance and Density," in *Proceedings of the 2000 ACM/SIGDA eighth International Symposium on Field Programmable Gate Arrays (FPGA)*, 2000, pp. 3-12.

[13] Lewis et al., "The Stratix II Logic and Routing Architecture," in *Proceedings of the ACM/SIGDA 13th International Symposium on Field Programmable Gate Arrays (FPGA)*, 2005, pp. 14-20.

[14] G. Lemieux and D. Lewis, *Design of Interconnection Networks for Programmable Logic.*: Springer (formerly Kluwer Academic Publishers), 2004.

[15] S. J. E. Wilton, J. Rose, and Z. G. Vranesic, "Memory/Logic Interconnect Flexibility in FPGAs with Large Embedded Memory Arrays," in *Proceedings of the IEEE Custom Integrated Circuits Conference*, 1996.

[16] Xilinx, Inc. Microblaze Soft Processor Core. [Online]. http://www.xilinx.com/tools/microblaze.htm

[17] Altera. NIOS II Processor. [Online]. http://www.altera.com/products/ip/processors/nios2/ni2-index.html

[18] Xilinx, Inc. Zynq-7000 Extensible Processing Platform. [Online]. http://www.xilinx.com/products/silicon-devices/epp/zynq-7000/index.htm

[19] Altera. Dual-Core ARM Cortex-A9 MPCore Processor. [Online]. http://www.altera.com/devices/processor/arm/cortex-a9/m-arm-cortex-a9.html

[20] D. Chen, J. Cong, and P. Pan, "FPGA Design Automation: A Survey," *Foundations and Trends in Electronic Design Automation*, vol. 1, no. 3, pp.

195-330, November 2006.

[21] S. N. Adya and I. L. Markov, "Fixed-Outline Flooplanning: Enabling Hierarchical Design," *IEEE Transactions on Very Large Scale Integrated Systems*, vol. 11, no. 6, pp. 1120-1135, December 2003.

[22] S. N. Adya, S. Chaturvedi, J. A. Roy, D. A. Papa, and I. L. Markov, "Unification of Partitioning, Placement and Floorplanning," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2004, pp. 550-557.

[23] L. Cheng and M. D. F. Wong, "Floorplan Design for Multimillion Gate FPGAs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 12, pp. 2795-2805, December 2006.

[24] Y. Feng and D. Mehta, "Heterogeneous Floorplanning for FPGAs," in *Proceedings of the Conference on VLSI Design*, 2006, pp. 257-262.

[25] P. Banerjee, S. Sur-Kolay, and A. Bishnu, "Fast Unified Floorplan Topology Generation and Sizing on Heterogeneous FPGAs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 5, pp. 651-661, May 2009.

[26] A. Montone, M. D. Santambrogio, D. Sciuto, and S. O. Memik, "Placement and Floorplanning in Dynamically Reconfigurable FPGAs," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 3, no. 4, November 2010.

[27] C. Bolchini, A. Miele, and C. Sandionigi, "Automated Resource-Aware Floorplanning of Reconfigurable Areas in Partially-Reconfigurable FPGA Systems," in *Proceedings of the 21st International Conference on Field-Programmable Logic and Applications (FPL)*, 2011, pp. 532-538.

[28] P. Banerjee, M. Sangtani, and S. Sur-Kolay, "Floorplanning for Partially Reconfigurable FPGAs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 1, pp. 8-17, January 2011.

[29] S. Guccione, D. Levi, and P. Sundararajan, "JBits: Java-based Interface for Reconfigurable Computing," in *Proceedings of the 2nd Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD)*, 1999.

[30] Xilinx, Inc. (2011, November) Partial Reconfiguration User Guide. [Online]. http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_3/ug702.pdf

[31] C. A. Hoare, "Communicating Sequential Processes," *Communications of the ACM*, vol. 21, no. 8, August 1978.

[32] R. Milner, *A Calculus of Communicating Systems*. New York: Springer-Verlag, 1982.

[33] T. Murata, "Petri Nets: Properties, Analysis and Applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541-580, 1989.

[34] Unified Modeling Language. [Online]. http://www.uml.org

[35] J. Eker et al., "Taming Heterogeneity--The Ptolemy Approach," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127-144, January 2003.

[36] Xilinx, Inc. AutoESL. [Online]. http://www.xilinx.com/tools/autoesl.htm

[37] Bluespec. Bluespec. [Online]. http://www.bluespec.com/index.htm

[38] Impulse. Impulse C. [Online]. http://www.impulseaccelerated.com/products_universal.htm

[39] Synopsys. http://www.synopsys.com/Tools/SLD/HLS/Pages/SynphonyC-Compiler.aspx.

[40] A. Bergamaschi et al., "Automating the Design of SOCs Using Cores," *IEEE Design Test*, vol. 18, no. 5, pp. 32-45, September 2001.

[41] Virtual Socket Interface Alliance (VSIA). [Online]. http://www.vsi.org

[42] SPIRIT Consortium, "IP-XACT User Guide: IP-XACT Draft

Specifications version 1.4 (beta one)," 2007.

[43] J. Kulp and S. Siegel, "Worker Interface Profiles (WIP) Functional Specification: Profiles for OCP," 2010.

[44] Open Core Protocol. [Online]. http://www.ocpip.org

[45] A. Arnesen, N. Rollins, and M. Wirthlin, "A Multi-layered XML Schema and Design Tool for Reusing and Integrating FPGA IP," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, 2009, pp. 472-475.

[46] A. Arnesen et al., "Increasing Design Productivity Through Core Reuse, Meta-data Encapsulation, and Synthesis," in *Proceedings of the International Conference on Field-Programmable Logic and Applications (FPL)*, 2010, pp. 538-543.

[47] T. Perry, R. Walke, and K. Benkrid, "An Extensible Code Generation Framework for Heterogeneous Architectures Based on IP-XACT," in *Proceedings of the Southern Conference on Programmable Logic (SPL)*, 2011, pp. 81-86.

[48] Mathworks. Simulink. [Online]. http://www.mathworks.com/products/simulink

[49] National Instruments. LabVIEW. [Online]. http://www.ni.com/labview

[50] Xilinx, Inc. System Generator. [Online]. http://www.xilinx.com/tools/sysgen.htm

[51] C. Kulkarni, G. Brebner, and G. Schelle, "Mapping a Domain-specific Language to a Platform FPGA," in *Proceedings of the 41st ACM/SIGDA Design Automation Conference* , 2004, pp. 924-927.

[52] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. Kaashoek, "The Click Modular Router," *ACM Transactions on Computer Systems*, vol. 18, no. 3, pp. 263-297, August 2000.

[53] Xilinx, Inc. EDK. [Online]. http://www.xilinx.com/ise/embedded/edk_docs.htm

[54] Altium. Altium Designer. [Online]. http://www.altium.com/products/altiumdesigner/

[55] S. Trimberger, D. Carberry, A. Johnson, and J. Wong, "A Time-multiplexed FPGA," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, 1997, pp. 22-29.

[56] Tabula. [Online]. http://www.tabula.com

[57] K. Nagami, K. Oguri, T. Shiozawa, H. Ito, and R. Konishi, "Plastic Cell Architecture: Towards Reconfigurable Computing for General-Purpose," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, 1998, p. 68.

[58] J. D. Hadley and B. L. Hutchings, "Design Methodologies for Partially Reconfigured Systems," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, 1995, pp. 78-84.

[59] B. Hutchings and M. Wirthlin, "Implementation Approaches for Reconfigurable Logic Applications," in *Proceedings of the International Conference on Field-Programmable Logic and Applications (FPL)*, 1995, pp. 419-428.

[60] G. Brebner, "The Swappable Logic Unit: a Paradigm for Virtual Hardware," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, 1997, pp. 77-86.

[61] O. Diessel and H. ElGindy, "Run-time Compaction of FPGA Designs," in *Proceedings of the International Conference on Field-Programmable Logic and Applications (FPL)*, 1997, pp. 131-140.

[62] G. Brebner and O. Diessel, "Chip-Based Reconfigurable Task Management," in *Proceedings of the International Conference on Field-Programmable Logic and Applications (FPL)*, 2001, pp. 182-191.

[63] M. Majer, J. Teich, A. Ahmadinia, and C. Bobda, "The Erlangen Slot Machine: A Dynamically Reconfigurable FPGA-based Computer," *VLSI*

Signal Processing, vol. 47, no. 1, pp. 15-31, 2007.

[64] M. Ullmann, M. Hübner, and J. Becker, "On-demand FPGA Run-time System for Flexible and Dynamical Reconfiguration," *IJES*, vol. 1, no. 3/4, pp. 193-204, 2005.

[65] G. B. Wigley, D. A. Kearney, and M. Jasiunas, "ReConfigME: A Detailed Implementation of an Operating System for Reconfigurable Computing," *IPDPS*, 2006.

[66] S. Koh and O. Diessel, "Module Graph Merging and Placement to Reduce Reconfiguration Overheads in Paged FPGA Devices," in *Proceedings of the International Conference on Field-Programmable Logic and Applications (FPL)*, 2007, pp. 293-298.

[67] J. Suris, C. Patterson, and P. Athanas, "An Efficient Run-time Router for Connecting Modules in FPGAs," in *Proceedings of the International Conference on Field-Programmable Logic and Applications (FPL)*, 2008, pp. 125-130.

[68] Miller, Prasanna, Reises, and Stout, "Meshes with Reconfigurable Buses," in *Proceedings of the 5th MIT Conference on Advanced Research in VLSI*, 1988, pp. 163-178.

[69] S. Lange and M. Middendorf, "The Partition into Hypercontexts Problem for Hyperreconfigurable Architectures," in *Proceedings of the International Conference on Field-Programmable Logic and Applications (FPL)*, 2004, pp. 251-260.

[70] U. Malik and O. Diessel, "The Entropy of FPGA Reconfiguration," in *Proceedings of the International Conference on Field-Programmable Logic and Applications (FPL)*, 2006, pp. 1-6.

[71] J. W. Lockwood, J. S. Turner, and D. E. Taylor, "Field Programmable Port Extender (FPX) for Distributed Routing and Queuing," in *Proceedings of the ACM International Symposium on Field Programmable Gate Arrays (FPGA)*, 2000, pp. 137-144.

[72] G. Gibb, J. W. Lockwood, J. Naous, P. Hartke, and N. McKeown, "NetFPGA---An Open Platform for Teaching How to Build Gigabit-Rate Network Switches and Routers," *IEEE Transactions on Education*, vol. 51, no. 3, pp. 364-369, August 2008.

[73] M. Zadnik, J. Korenek, P. Kobiersky, and O. Lengal, "Network Probe for Flexible Flow Monitoring," in *11th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems*, 2008, pp. 1-6.

[74] H. Le, W. Jiang, and V. K. Prasanna, "A SRAM-based Architecture for Trie-based IP Lookup Using FPGA," in *Proceedings of the 16th IEEE Symposium on Field Programmable Custom Computing Machines (FCCM)*, 2008.

[75] H. Song, F. Hao, M. Kodialam, and T. V. Lakshman, "IPv6 Lookups using Distributed and Load Balanced Bloom Filters for 100Gbps Core Router Line Cards," in *Proceedings of the IEEE INFOCOM*, Rio De Janeiro, 2009.

[76] B. L. Hutchings, R. Franklin, and D. Carver, "Assisting Network Intrusion Detection with Reconfigurable Hardware," in *Proceedings of the 10th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2002, p. 111.

[77] R. Sidhu and V. K. Prasanna, "Fast Regular Expression Matching Using FPGAs," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2001.

[78] J. Moscola, M. Pachos, J. W. Lockwood, and R. P. Loui, "FPsed: A Streaming Content Search-and-Replace Module for an Internet Firewall," in *11th Symposium on High Performance Interconnects*, 2003, p. 122.

[79] Y. Cho, S. Navab, and W. Mangione-Smith, "Specialized Hardware for Deep Network Packet Filtering: Reconfigurable Computing is Going Mainstream," in *Proceedings of the 12th International Conference on Filed-Programmable Logic and Applications*, 2002, pp. 452-461.

[80] Z. K. Baker, H. J. Jung, and V. K. Prasanna, "Regular Expression Software Deceleration for Intrusion Detection System," in *Proceedings of the 16th International Conference on Field-Programmable Logic and Applications (FPL)*, 2006.

[81] Sourcefire. Snort Open Source Network Intrusion Prevention and Detection System. [Online]. http://www.snort.org

[82] S. Dharmapurikar, P. Krishnamurthy, T. S. Sproull, and J. W. Lockwood, "Deep Packet Inspection Using Parallel Bloom Filters," *IEEE Micro*, pp. 52-61, January/February 2004.

[83] H. Fallside and M. J. S. Smith, "Internet Connected FPL," in *Proceedings of the 10th International Conference on Field-Programmable Logic and Applications*, 2000, pp. 48-57.

[84] D. V. Schuehler and J. W. Lockwood, "TCP Splitter: A TCP/IP Flow Monitor in Reconfigurable Hardware," *IEEE Micro*, pp. 54-59, January/February 2003.

[85] W. S. Marcus, I. Hadzic, A. J. McAuley, and J. M. Smith, "Protocol Boosters: Applying Programmability to Network Infrastructures," *IEEE Communications Magazine*, vol. 36, no. 10, pp. 79-83, October 1998.

[86] G. Brebner, "Packets Everywhere: The Great Opportunity for Field Programmable Technology," in *Proceedings of the International Conference on Field-Programmable Technology (FPT)*, 2009, pp. 1-10.

[87] C. Soviani, I. Hadzic, and S. A. Edwards, "Synthesis and Optimization of Pipelined Packet Processors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 2, pp. 231-244, February 2009.

[88] Q. Zhang, A. Marshall, and R. Woods, "A Traffic Manager for Integrated Queuing and Scheduling of Unicast and Multicast IP Traffic," in *Proceedings of the 16th International Conference on Telecommunications*,

Marrakech, Morocco, 2009, pp. 65-70.

[89] S. Casselman and J. Schewel, "Net Aware BitStreams that Upgrade FPGA Hardware Remotely Over the Internet," in *Proceedings of SPIE*, vol. 4867, Boston, MA, 2002.

[90] E. L. Horta, J. W. Lockwood, D. E. Taylor, and D. Parlour, "Dynamic Hardware Plugins in an FPGA with Partial Run-time Reconfiguration," in *Proceedings of the ACM/SIGDA Design Automation Conference (DAC)*, New Orleans, LA, 2002, p. 24.2.

[91] G. Brebner, "Circlets: Circuits as Applets," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, 1998, p. 300.

[92] Intel. IXP 2800 Programming. [Online]. http://www.intel.com/intelpress/ixp2800/

[93] Xcelerated. [Online]. http://www.xelerated.com/en/core-x10

[94] EZchip. NP-5 Network processor. [Online]. http://www.ezchip.com

[95] Cavium. Octeon Plus MIPS64 Processors. [Online]. http://www.cavium.com/OCTEON_MIPS64.html

[96] J. W. Lockwood, C. Neely, C. Zuver, and D. Lim, "Automated Tools to Implement and Test Internet Systems in Reconfigurable Hardware," *ACM SIGCOMM Computer Communications Review (CCR)*, vol. 33, no. 3, pp. 103-110, July 2003.

[97] V. Berman, "Standards: the P1685 IP-XACT IP Metadata Standard," *IEEE Design & Test of Computers*, vol. 23, no. 4, pp. 316-317, April 2006.

[98] Accellera. [Online]. http://www.accellera.org

[99] Mentor Graphics. HDL Designer. [Online]. http://www.mentor.com/products/fpga/hdl_design/hdl_designer_series/

[100] Xilinx, Inc. (2011, November) Platform Studio. [Online]. http://www.xilinx.com/platform.htm

[101] M. Wirthlin et al., "OpenFPGA CoreLib Core Library Interoperability Effort," *Journal of Parallel Computing*, vol. 34, no. 4-5, pp. 231-244, May

2008.

[102] N. Rollins, M. Wirthlin, and A. Arnesen, "An XML Approach to Facilitating IP Core Reuse," in *Proceedings of the National Aerospace and Defense Conference*, 2008.

[103] E. Kohler, *The Click Modular Router.*: Ph.D. thesis, MIT, 2000.

[104] Xilinx, Inc., "LocalLink Interface Specification," Xilinx Specification Paper SP006, 2005.

[105] C. Simonyi, "Hungarian Notation," Report 1999.

[106] J. S. Lo, C. E. Neely, and G. J. Brebner, "Hierarchical Interface for IC system," 7,852,117.

[107] C. E. Neely, G. J. Brebner, and J. S. Lo, "Graphical user interface for system design," 8,121,826.

[108] Piccolo Zoomable User Interface (ZUI) Construction Kit website. [Online]. http://www.cs.umd.edu/hcil/jazz/

[109] R. Passerone, J. Rowson, and A. Sangiovanni-Vincentelli, "Automatic Synthesis of Interfaces Between Incompatible Protocols," in *Proceedings of the ACM/SIGDA 35th Design Automation Conference (DAC)*, 1998, pp. 8-13.

[110] G. Brebner, *Computers in Communication.*: McGraw-Hill International, 1997.

[111] R. Metcalfe and D. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks," *Communications of the ACM*, vol. 19, no. 7, pp. 395-404, July 1976.

[112] G. Malkin, "RIP version 2," RFC 2453, 1998.

[113] J. Postel, "Transmission Control Protocol," RFC 793, 1981.

[114] J. Arkko, V. Torvinen, G. Camarillo, A. Niei, and T. Haukka, "Security Mechanism Agreement for the Session Initiation Protocol (SIP)," RFC 3329, 2003.

[115] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "RTP: A Transport Protocol for Real-time Applications," RFC 3550, 2003.

[116] D. Mills, "Network Time Protocol version 4," Technical Report 06-6-1, 2006.

[117] Institute of Electrical and Electronic Engineers (IEEE), "1588-2008 Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems," IEEE, Standard 1588-2008, 2008.

[118] IPClock, "IPC 50000 IEEE1588v2 Slave Ordinary Clock," Product brief R3.01, 2010.

[119] International Telecommunications Union (ITU-T), "Y.1731: OAM Functions and Mechanisms for Ethernet-based Networks," Standard 2008.

[120] Institute of Electrical and Electronic Engineers (IEEE), "802.1ag Standard for Local and Metropolitan Area Networks Virtual Bridged Local Area Networks, Amendment 5: Connectivity Fault Management," Standard 802.1ag, 2007.

[121] Hiroshi Ohta. (2006, April) Ethernet OAM and Protection Switching. [Online]. http://www.itu.int/ITU-T/worksem/ngn/200604/presentation/s7_ohta.pdf

[122] H. Walder and M. Platzner, "Non-preemptive Multitasking on FPGAs: Task Placement and Footprint Transform," in *In Proceedings of the 2nd International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, 2002, pp. 24-30.

[123] Xilinx, Inc. (2011, November) ISE Design Suite. [Online]. http://www.xilinx.com/products/design-tools/ise-design-suite/

[124] J. Suris, A. Recio, and P. Athanas, "Enhancing the Productivity of Radio Designers with RapidRadio," in *Proceedings International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2009.

[125] D. Koch, C. Beckhoff, and J. Teich, "ReCoBus-Builder: A Novel Tool and Technique to Build Statically and Dynamically Reconfigurable Systems for

FPGAs," in *Proceedings of the 18th International Conference on Field Programmable Logic and Applications (FPL)*, 2008, pp. 119-124.

[126] D. Koch, C. Beckhoff, and J. Torrison, "Advanced Partial Run-Time Reconfiguration on Spartan-6 FPGAs," in *Proceedings of the International Conference on Field-Programmable Technology (FPT)*, 2010, pp. 361-364.

[127] Xilinx, Inc. (2011, November) PlanAhead Software Design Analysis and Floorplanning Tutorial. [Online]. http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_3/PlanAhead_Tutorial_Design_Analysis_Flooplan.pdf

[128] H. Kalte, G. Lee, M. Porrmann, and U. Ruckert, "Study on Wise Design Compaction for Reconfigurable Systems," in *Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT)*, 2004, pp. 413-416.

[129] J. Carver, R. Pittman, and A. Forin, "Automatic Bus Macro Placement for Partially Reconfigurable FPGA Designs," in *Proceedings of the 17th ACM SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2009, pp. 269-272.

[130] L. Gong and O. Diessel, "Modeling Dynamically Reconfigurable Systems for Simulation-Based Functional Verification," in *Proceedings of the IEEE 19th International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2011, pp. 9-16.

[131] M. Attig and G. Brebner, "400 Gb/s Programmable Packet Parsing on a Single FPGA," in *Proceedings of the ACM/IEEE Seventh Symposium on Architectures for Networking and Communication Systems (ANCS)*, 2011, pp. 12-23.

[132] D. Koch, C. Beckhoff, and J. Torrison, "Fine-Grained Partial Runtime Reconfiguration on Virtex-5 FPGAs," in *Proceedings of the 18th IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2010, pp. 69-72.

[133] L. Benini and G. De Micheli, "Networks on Chips: A New SoC Paradigm," *IEEE Computer*, vol. 35, no. 1, pp. 70-78, January 2002.

[134] C. E. Neely, G. Brebner, and W. Shang, "ShapeUp: A High-Level Design Approach to Simplify Module Interconnection on FPGAs," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2010.

[135] I. Hadzic and J. Smith, "P4: A Platform for FPGA Implementation of Protocol Boosters," in *Proceedings of the International Workshop on Field-Programmable Logic and Applications*, London, England, 1997, pp. 438-447.

[136] IPBlaze, "High speed 10G Ethernet and TCP/IP Offload Engine (TOE)," Product brief PR004, 2008.

[137] J. Halak, "Multigigabit Network Traffic Processing," in *Proceedings of the International Conference on Field-Programmable Logic and Applications (FPL)*, 2009, pp. 521-524.

[138] W. Jiang and V. K. Prasanna, "Large-scale Wire-speed Packet Classification on FPGAs," in *Proceedings of the ACM Symposium on Field-Programmable Gate Arrays (FPGA)*, 2009, pp. 219-228.

[139] G. Watson, N. McKeown, and M. Casado, "NetFPGA: A Tool For Network Research and Education," in *Proceedings of Workshop on Architecture Research using FPGA Platforms*, Austin, TX, 2006.

[140] T. Becker, W. Luk, and P. Y. K. Cheung, "Enhancing Relocatability of Partial Bitstreams for Run-Time Reconfiguration," in *Proceedings of the 15th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2007, pp. 35-44.

[141] G. Brebner, "Live In-Service Modification of Optical Network Elements Implemented with Xilinx FPGAs," in *National Fiber Optic Engineers Conference, OSA Technical Digest (CD)*, 2011.

[142] Y. Lu, T. Marconi, K. Bertels, and G. Gaydadjiev, "A Communication Aware Online Task Scheduling Algorithm for FPGA-Based Partially Reconfigurable Systems," in *Proceedings of the 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2010, pp. 65-68.

[143] C. E. Neely, G. Brebner, and W. Shang, "Flexible and Modular Support for Timing Functions in High Performance Networking Acceleration," in *Proceedings of the International Conference on Field-Programmable Logic and Applications (FPL)*, 2010, pp. 513-518.

[144] Xilinx, Inc. (2011, November) PlanAhead Software Partial Reconfiguration Tutorial. [Online]. http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_3/PlanAhead_Tutorial_Partial_Reconfiguration.pdf

[145] W. Kruijtzer et al., "Industrial IP Integration Flows Based on IP-XACT Standards," in *Proceedings of the Conference on Design, Automation, and Test in Europe (DATE)*, 2008, pp. 32-37.

Vita

Christopher E. Neely

Date of Birth November 18, 1979

Place of Birth Saint Louis, Missouri

Degrees B.S. Computer Engineering, May 2002
M.S. Computer Engineering, May 2004
Ph.D. Computer Engineering, June 2012 (anticipated)

Publications

C. Neely, G. Brebner, W. Shang. "Reshape: Towards a High-Level Design Approach to Simplify Module Interconnection on FPGAs", ACM Transactions on Reconfigurable Technology and Systems (TRETS), Accepted and pending publication.

C. Neely, G. Brebner, W. Shang. "Flexible and Modular Support for Timing Functions in High Performance Networking Acceleration", Field-Programmable Logic and Applications (FPL'10), August/September 2010, pp.513-518.

C. Neely, G. Brebner, W. Shang. "ShapeUp: A High-Level Design Approach to Simplify Module Interconnection on FPGAs", IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'10), May 2010, pp.141-148.

T. Sproull, G. Brebner, and C. Neely. "Mutable Codesign for Embedded Protocol Processing", Field-Programmable Logic and Applications (FPL'05), Tampere, Finland, Paper 1C, Aug 24-26, 2005.

J. W. Lockwood, C. Neely, C. Zuver, J. Moscola, S. Dharmapurikar, and D. Lim. "An Extensible, System-On-Programmable-Chip, Content-Aware Internet Firewall", Field-Programmable Logic and Applications (FPL'03), Lisbon, Portugal, Paper 14B, Sep 1-3, 2003.

J. W. Lockwood, C. Neely, C. Zuver, and D. Lim. “Automated Tools to Implement and Test Internet Systems in Reconfigurable Hardware”, ACM SIGCOMM Computer Communication Review (CCR), vol 33, no 3, July 2003, pp 103-110.

D. Lim, C. E. Neely, C. K. Zuver, J. W. Lockwood. “Internet-based Tool for System-on-Chip Integration”, IEEE Computer Society International Conference on Microelectronic Systems Education (MSE'03), Anaheim, CA, June 2003

C. K. Zuver, C. E. Neely, J. W. Lockwood. “Internet-based Tool for System-On-Chip Project Testing and Grading”, IEEE Computer Society International Conference on Microelectronic Systems Education (MSE'03), Anaheim, CA, June 2003

Patents

US Patent #8,121,826
Neely; Christopher E. (San Jose, CA), Brebner; Gordon J. (San Jose, CA), Lo; Jack S. (Santa Clara, CA) “Graphical user interface for system design”

US Patent #7,852,117.
Lo; Jack S. (Santa Clara, CA), Neely; Christopher E. (San Jose, CA), Brebner; Gordon J. (San Jose, CA) "Hierarchical Interface for IC system"

US Patent #7,784,014.
Brebner; Gordon J. (Los Gatos, CA), Neely; Christopher E. (San Jose, CA), James-Roxby; Philip B. (Longmont, CO), Keller; Eric R. (Boulder, CO), Kulkarni; Chidamber R. (Santa Clara, CA), Baxter; Michael A. (Sunnyvale, CA), Styles; Henry E. (San Jose, CA), Schelle; Graham F. (Boulder, CO) "Generation of a specification of a network packet processor "

Service

Program Committee Member for 23rd IEEE International Conference on Application-specific Systems, Architectures, and Processors (ASAP’2012)

Program Committee Member for 2011 International Conference on ReConFigurable Computing and FPGAs (ReConFig’2011)

June 2012