Validation and Verification of the Interconnection of Hardware Intellectual Property Blocks for FPGA-based Packet Processing Systems

Paul Edward McKechnie BEng (Hons)

A thesis submitted to

The Universities of

Edinburgh

Glasgow

Heriot Watt

Strathclyde

for the Degree of

Doctor of Engineering in System Level Integration

To Alexis, Elijah and Elyzabel

Abstract

As networks become more versatile, the computational requirement for supporting additional functionality increases. The increasing demands of these networks can be met by Field Programmable Gate Arrays (FPGA), which are an increasingly popular technology for implementing packet processing systems. The fine-grained parallelism and density of these devices can be exploited to meet the computational requirements and implement complex systems on a single chip. However, the increasing complexity of FPGA-based systems makes them susceptible to errors and difficult to test and debug.

To tackle the complexity of modern designs, system-level languages have been developed to provide abstractions suited to the domain of the target system. Unfortunately, the lack of formality in these languages can give rise to errors that are not caught until late in the design cycle. This thesis presents three techniques for verifying and validating FPGA-based packet processing systems described in a system-level description language. First, a type system is applied to the system description language to detect errors before implementation. Second, system-level transaction monitoring is used to observe high-level events on-chip following implementation. Third, the high-level information embodied in the system description language is exploited to allow the system to be automatically instrumented for on-chip monitoring.

This thesis demonstrates that these techniques catch errors which are undetected by traditional verification and validation tools. The locations of faults are specified and errors are caught earlier in the design flow, which saves time by reducing synthesis iterations.

Acknowledgements

I have been supported by many people during this project and I would like to thank everyone who has offered me encouragement during these last four years. Several people have played a pivotal role in my studies and I would like to thank them specifically by name as I would not have achieved as much without their help.

First, I would like to thank the software team at Xilinx, who taught me the art of software engineering, instructed me in the ways of FPGA design and patiently endured my technical talks. Those patient people are Scott Leishman, Martin Sinclair, Brian Cotter, Estelle Deperetis, Alex Shaw, Ann Duncan and David O'Callahan.

I am also grateful for the support that I received from the Xilinx Research Labs. In particular, I would like to thank Michaela Blott, who moulded my ideas and gave freely of her time, and Gordon Brebner, who suggested many avenues of investigation.

I am indebted to my industrial supervisors, who have been excellent sources of knowledge and advice. I would like to thank Ben Curry, who supervised me during the latter stages of my research and Nathan Lindop, who was particularly patient and supportive during the critical stages.

My academic supervisor, Wim Vanderbauwhede, has been a great support and always willing to provide a different perspective. Without his guidance, I would never have explored the variety of avenues that I have.

I would like to thank my parents for their support over these last four years and my two children, Elijah and Elyzabel, for the times that we have missed together while I have worked on this thesis.

Finally, I am thankful for the support that my wife, Alexis, has shown me over these last four years. She has been critical to my success. Without your sacrifice and understanding I might never have finished.

Declaration of Originality

I declare that this thesis was composed entirely by myself and that the work contained herein is my own except where acknowledged in the text. A list of references has been given in the bibliography. I also declare that this thesis has not been submitted for any other degrees or professional qualifications at any university. Parts of the work have appeared in other publications and these are listed in Appendix A. I am the sole author of this thesis and any errors contained herein are my own.

(Paul Edward McKechnie)

Contents

List of Figures

List of Tables

Acronyms and Abbreviations

ADSL	Asymmetric Digital Subscriber Line
ASIC	Application Specific Integrated Circuit
BEE2	Berkely Emulation Engine 2
BORPH	Berkeley OS for ReProgrammable Hardware
BRAM	Block Random Access Memory
BUFG	Global Buffer
CAM	Content Addressable Memory
CLB	Complex Logic Block
CPU	Central Processing Unit
DCM	Digital Clock Module
DMA	Direct Memory Access
DRAM	Distributed Random Access Memory
DRC	Design Rule Check
DSLAM	Digital Subscriber Line Access Multiplexer
DSP	Digital Signal Processor
EDA	Electronic Design Automation
EDIF	Electronic Design Interchange Format

EDK	Embedded Development Kit
ESL	Electronic System Level
FPGA	Field Programmable Gate Array
FTP	File Transfer Protocol
GMII	Gigabit Media Independent Interface
GPU	Graphics Processing Unit
HDL	Hardware Description Language
HTTP	Hyper Text Transport Protocol
IP	Intellectual Property
IP	Internet Protocol
JHDL	Java Hardware Description Language
LUT	Lookup Table
MAC	Medium Access Control
MII	Media Independent Interface
NCD	Native Circuit Description
NCF	Netlist Constraints File
NGC	Native Generic database and Constraints
NGD	Native Generic Database
NGO	Native Generic Object
NoC	Network on Chip
NPU	Network Processing Unit
PAR	Place And Route
PCU	Packet Control Unit

PHY	PHYsical layer device
PLB	Processor Local Bus
PLD	Programmable Logic Device
PSF	Platform Specification Format
RED	Random Early Discard
RPM	Relationally Placed Macro
RTL	Register Transfer Level
SoC	System-on-Chip
SSH	Secure Shell
TCAM	Ternary Content Addressable Memory
TCP	Transmission Control Protocol
TEMAC	Tri-mode Ethernet Medium Access Controller
TLM	Transaction Level Modelling
TRACE	Timing Reporter And Circuit Evaluator
UCF	User Constraints File
UDP	User Datagram Protocol
UML	Unified Modelling Language
XCF	Xilinx Constraints File
XST	Xilinx Synthesis Tool

Chapter 1

Introduction

Networks are pervasive in modern society. High bandwidth applications, such as audio and video streaming, are frequently used on both fixed and mobile networks. Cloud computing is commonplace with many traditionally local applications being hosted remotely as services. This trend towards software services means that more computation is performed within the network.

In order to support these applications, the network needs to provide complementary functions such as access control, encryption, load balancing, packet classification and packet forwarding. These functions are collectively known as packet processing and they are provided transparently by nodes within the network. The demands placed on these nodes are ever-increasing as bandwidth and computational capabilities improve to meet the insatiable demand for more functionality.

Early packet processing systems were implemented using general purpose processors. This permitted the use of cheap standard components, which could be customised through software. However, these architectures failed to meet the increasing computational requirements associated with the increase in bandwidth. In response to this, a class of domain-specific system-on-chip devices was developed. Known as Network Processing Units (NPUs), these devices provide dedicated hardware components and frequently include specialised microengines. The dedicated hardware units typically perform functions such as framing and checksum calculation. The microengines support specialised instructions for packet processing such as bit field manipulation. These features permit

performance improvements over general purpose processors while retaining the flexibility and customisation of software. Software customisation allows a NPU device to be used in a variety of applications, which leverages economies of scale. Although these architectures are more efficient than general purpose processors, the software programming model exposes complex detail, making programming difficult and time consuming. As a result, there has been a significant research effort devoted to improving the programming model.

The concept of device customisation can be extended further through the use of Field Programmable Gate Arrays (FPGAs). FPGAs are regular structures composed of programmable logic cells and their interconnections [1]. They might also include other components such as multipliers, memories and processors. The main advantage of FPGAs is that they allow hardware units to be customised to the needs of a specific application and they can be reconfigured after device fabrication. Some devices even permit the hardware units to be reconfigured as the system is executing. FPGAs can also provide improved performance and lower power consumption compared to CPU and NPU implementations. The available resources in modern FPGAs permits complex designs to be implemented on a single device. Again, the precise customisation afforded by FPGAs permits their use in a variety of applications, which can also leverage economies of scale. However, the increasing complexity of FPGA systems makes them difficult to program and susceptible to functional errors. Consequently, FPGAs can be difficult to test and debug. As FPGAs provide many benefits over CPUs and NPUs, improving the programming model of FPGA-based systems would allow the capabilities of these devices to be exploited in a wider range of applications.

1.1 Motivations

As process geometries shrink following the trend of Moore's law, more transistors can be fabricated on a single silicon device. On a FPGA this means that the number of programmable logic cells can be increased and indirectly the functionality of the device increases. However, the ability of the designer to utilise this increased functionality is improving at a much slower rate, giving rise to what is known as the designer productivity gap, as shown in Figure 1.1.

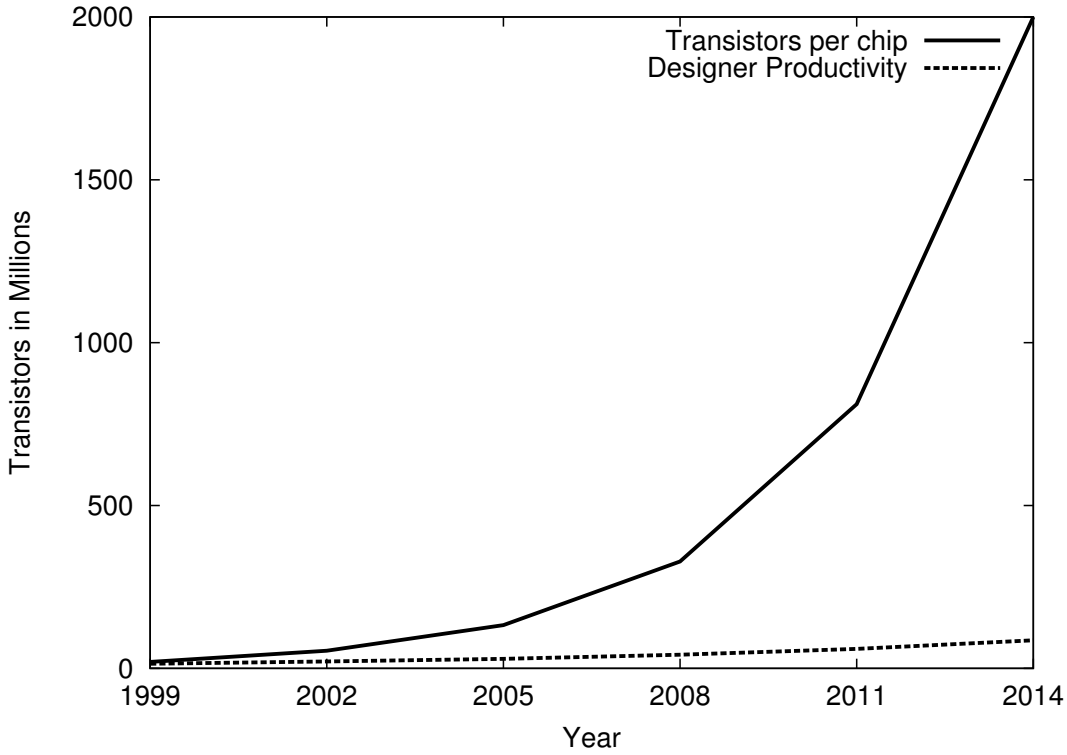

Figure 1.1: The designer productivity gap with numbers obtained from the International Technology Roadmap for Semiconductors[2].

One solution to the productivity gap is to use abstraction through libraries of reusable functions. These libraries of functions, commonly known as Intellectual Property (IP) blocks, are customisable to suit the needs of a particular design. These libraries have well-defined interfaces and functionality. They are frequently used to reduce complexity as perceived by the designer and improve the time to market. The IP blocks are also well-tested, validated and used in a variety of designs providing confidence that the component is functionally correct. Systems can be built by assembling IP blocks to create the desired functionality[3], which parallels the software engineering technique of using software libraries.

In order to further reduce the productivity gap, high-level design tools have been developed that allow the designer to connect IP blocks to form a complete system. Such high-level design environments define the types and semantics of IP block interfaces and provide IP block interconnection specification languages, which may be textual or graphical. Using domain-specific languages removes the need for the designer to connect each

wire of an IP block individually and reduces the time needed to describe the system. It is also less error prone. Furthermore, the designer does not need a detailed understanding of each interface type used in the system.

Within Xilinx, several IP interconnection tools have been developed including the Embedded Development Kit [4], System Generator [5], Brace and System Stitcher. The Embedded Development Kit and System Generator are commercial applications, which target processor-based systems and DSP applications respectively. Brace and System Stitcher are research tools that support packet processing systems. However, Brace can also be used in a range of application domains beyond packet processing. The common feature between these tools is that each one is capable of creating systems from interconnections of IP blocks.

1.2 Aims and Objectives

The aim of this research is to provide verification and validation of FPGA-based packet processing systems described in a high-level design environment without requiring a detailed understanding of the low-level signalling. Existing techniques and methods require a detailed knowledge of low-level signalling but high-level system descriptions present a different class of errors that might not be caught through traditional validation and verification techniques.

For example, an IP block might function correctly in isolation but errors can be introduced when it is integrated with other components. The area requirements, spatial layout and timing constraints of an IP block might cause system-level integration errors. Typical integration errors range from mismatched timing constraints to undesired component interactions. For example, two incompatible interfaces might exhibit unintended interactions when connected together. As a result, integrating IP blocks might require a significant effort as the designer needs to understand the function of the block, the operation of its interface and the protocol used for data transmission. Furthermore, the interfaces on IP components might be incompatible, requiring wrappers or collars to communicate with the rest of the system. These issues are compounded in the traditional HDL design flow as the designer must manually ensure that all components involved use

the same standard.

The aim of this research is complemented by six specific objectives, which state how verification and validation can be applied to FPGA-based packet processing systems described in a high-level design environment.

1. Use static verification to detect interconnection errors that are not presently detected by the implementation flow.
2. Reduce the time required to detect interconnection errors compared with the traditional implementation flow.
3. Use dynamic monitoring to detect errors not observed by existing on-chip monitoring tools.
4. Design a mechanism to interpret low-level signalling as high-level events.
5. Temporally relate distributed events observed on-chip.
6. Eliminate observation errors by automating the insertion of a dynamic on-chip monitoring system into designs.

1.3 Contributions

This work has focused on researching techniques for the validation and verification of IP block interconnections used in FPGA-based packet processing systems. Within this thesis, validation is defined as a demonstration of conformance to a set of properties or tests. Validation gives confidence that an artifact is correct but does not guarantee that its properties hold under all conditions. Conversely, verification is a formal proof which states that the given properties are true under all specified conditions. This thesis makes three main contributions:

1. Type checking of an IP interconnection specification language, allowing errors to be caught before the system is synthesised.

2. System-level transaction monitoring of packet processing systems described by an IP interconnection specification language, permitting errors to be observed at run-time.

3. Automatic instrumentation of packet processing systems described by an IP interconnection specification language for on-chip monitoring, eliminating errors due to incorrect instrumentation.

Type checking an IP interconnection specification language is a form of static verification. This technique allows errors to be detected in the high-level design environment as the system is compiled and before it is synthesised. Type checking saves a significant amount of time by catching errors earlier in the design flow and reducing the number of synthesis iterations. This technique is frequently used in software design to prove certain properties of a program.

This thesis makes two specific contributions with regards to type checking an IP interconnection language. First, a type system for static verification of IP block interfaces is specified. Second, an implementation of a type checker, which verifies connections according to the rules of the type system, is presented. The type checker has been implemented as a component of the Brace research tool.

System-level transaction monitoring addresses some of the limitations of traditional on-chip monitoring tools. Traditional run-time monitoring tools typically record low-level information and tend to focus on monitoring a single location. Low-level monitoring also produces vast amounts of information, which can be difficult to comprehend. To configure and use these tools effectively, the designer needs a detailed understanding of the low-level signalling within the system, which they might not necessarily have. System-level transaction monitoring addresses these limitations by using transaction-level observations, which relates events to the design environment and exposes a different class of errors compared to traditional tools. The amount of data transmitted off-chip is significantly reduced and fewer external pins will be required. Furthermore, system-level transaction monitoring has a small resource requirement, which minimises the impact on placement, routing and system timing. The small resource requirement also allows more probes to be inserted throughout the design to provide a system-level perspective.

The architecture of a system-level transaction monitoring tool is a major contribution of this thesis. This architecture consists of probes, collection circuitry and external host software. This thesis presents six variants of the probe architecture and two variants of the collection circuitry. Each probe and collector architecture has been implemented on a variety of FPGA devices and has been used to monitor several packet processing systems. Finally, this thesis has also contributed monitoring software, which interprets the results transmitted by the monitoring system on the target FPGA.

The insertion of on-chip monitoring circuitry is typically a manual process. In traditional design flows the probe insertion process is time consuming and error prone as signals need to be connected individually. Typical designs require the designer to connect many signals in order to correctly insert a probe and its supporting circuitry. The configuration of the probe is also a separate step that must be manually performed. The transaction-level semantics of component interfaces can be exploited to permit automatic instrumentation of the design, which reduces the potential for error in connecting probes and reduces the time required to instrument a design. The transaction-level semantics can also be used to configure the monitoring system. This thesis specifies an algorithm for automatically instrumenting a design and has contributed an implementation of that algorithm within the System Stitcher research tool.

1.4 Thesis Layout

The format of the remainder of this thesis is as follows. Chapter 2 provides a background to the validation and verification of packet processing systems. It also provides a wider understanding of network monitoring and the problems associated with monitoring packet processors.

Chapter 3 describes the FPGA design flow, while highlighting existing validation and verification techniques. The main contributions of this thesis are discussed in relation to their applicability to the FPGA design flow and high-level design environments.

Chapter 4 discusses the verification of IP block interconnections. It presents a type system for static verification of interfaces and their connections. The formal representation of

the type system is presented and the error detection rate is discussed in comparison to techniques used commercially.

The architecture of an on-chip monitoring mechanism that observes packet processing systems implemented in a FPGA is presented in Chapter 5. The system raises the abstraction level of the monitored signals and is designed to observe the interfaces of components, which has the advantage of leaving the IP blocks unaltered. The architecture of the components comprising the monitoring system are explored and their resource requirements are presented.

Chapter 6 describes the application of the monitoring system to three case studies. This chapter demonstrates the observations that can be made with a system-level monitoring tool and highlights the errors detected. The errors that could not be detected with traditional tools are also presented.

An algorithm for automatically instrumenting a design is presented in Chapter 7. It uses a variation of the type system as a method of instrumenting a system described in a domain specific language such as Click. The technique used to create a system description with monitoring circuitry is also described.

Chapter 8 examines future directions for the work. This thesis has presented work on validating and verifying packet processing systems, which has improved designer productivity. However, the techniques presented can be researched further. Additionally, there are other complementary techniques that can be explored to further improve design validation and verification.

The conclusions are presented in Chapter 9. Although, there are avenues yet to be explored, this thesis has presented a significant body of work related to the validation and verification of FPGA-based packet processing systems. The techniques presented in this thesis have improved designer productivity and caught a class of errors not detected by traditional tools.

Finally, Appendix A contains a list of publications, which have resulted from the research.

Chapter 2

Background

Networks are employed in many diverse applications that vary dramatically in scale. The most common perception of networks is the interconnection of multiple desktop computers which can span the globe, such as the Internet, or occupy a single room. Networks can also be much smaller. They can be found within a single computer or contained entirely within a single silicon device. In each case, the user of the system might be unaware that a network is being employed.

Due to the pervasive nature of networks in modern electronic systems, packet processing is frequently employed to provide the functionality which permits components to communicate. This chapter presents an overview of packet processing applications. It examines packet processing systems as components within the network and as complex systems in their own right. An overview of existing FPGA-based packet processing systems is presented and descriptions of functions implemented in FPGAs are given. The chapter also discusses existing design environments for packet processing applications and presents an overview of type checking applied to existing languages. Furthermore, tools and techniques for monitoring and observing FPGA-based systems are presented. These tools tend to focus on capturing low-level information, which is not directly related to the abstractions used in high-level design environments. Consequently, the monitoring tools discussed in this chapter are not restricted to packet processing applications. Existing automated instrumentation techniques are presented and the requirements for on-chip monitoring are discussed. Finally, this chapter presents a review of two methodologies for

debugging applications.

2.1 Introduction

The design and implementation of packet processing systems is a topic of active research, which can be examined from two different perspectives. First, packet processors form components within a network. The properties and responses of an individual packet processor can affect other components in the network, which can alter the performance and response of the network as a whole. Second, a packet processor is itself a complex system, which performs a variety of functions. The architecture of the packet processor can impact the power consumption, latency, throughput and memory requirements of the silicon device.

Due to the complexity of packet processing systems, languages and tools have been developed that describe, implement, validate and verify such systems. These languages and tools tend to increase the level of abstraction used to describe the operation of such systems and frequently enforce a separation between the control and data paths in the system. Modern design environments remove the need for the designer to perform mundane error prone tasks and can provide tools for validating and verifying packet processing systems. These design environments frequently include tools for monitoring the implementation of the system, as not all errors can be caught beforehand.

While packet processors are implemented in a variety of technologies, FPGAs have become a popular implementation choice. The customisation and fine-grained parallelism inherent in FPGA devices allows them to be used in a variety of applications and can provide improved performance compared to software implementations. The cost and reconfigurability of FPGAs provides several advantages over ASIC implementations, which makes them suitable for a range of packet processing applications. Due to the popularity of FPGA-based packet processing systems, research has been carried out on the architecture of such implementations and the boards that support these devices.

2.2 Packet Processor as a Network Component

Packet processors are not isolated artifacts. Each packet processor is one component of a larger system which provides services to other components. The provision of services to other components allows the network to operate as a single complex system by supporting network protocols and providing quality of service guarantees. In order to examine and determine the properties of these services, the packet processor cannot be observed in isolation. The network must be examined as a complete system.

Within networks, protocols are used to define the rules of communication and are an area of active development. Peer to peer protocols for file sharing and instant messaging are becoming increasingly popular, while client-server protocols are widely used for database applications and delivery of web pages. Ideally, these protocols would be tested using the target hardware but this is impractical due to the expense of creating large physical networks, the complexity of repeatedly reconfiguring the network topology and the difficulty of creating repeatable experiments on hardware. As a result, network simulators have been developed to observe and examine network operations.

Network simulators are used to develop and examine a variety of protocols, which range from router queueing protocols, such as Random Early Discard (RED), to TCP behaviour, such as selective acknowledgement. The abstraction mechanisms provided by network simulators allow the designer to focus on details of interest in both abstract and detailed models. Network simulation also provides a high degree of control over scenario generation and can usually permit multiple protocols to be simulated simultaneously, which can highlight unexpected protocol interactions. Two frequently used network simulators are OMNeT++ and ns-2.

OMNeT++ [6] is a discrete event simulator, which uses a split programming model. Components of the network and their operations are described in C++, whereas the architecture of the network is described in a declarative language called NED. The NED language separates component implementation from network topology, which encourages components to be reused between various simulations.

Ns-2 [7] is also a discrete event simulator with a split programming model. The event-level

packet processing operations are described in C++ but the simulation orchestration is described in OTcl. The OTcl scripts express the definition, configuration and control of the simulation. As OTcl is an imperative scripting language the distinction between the component functionality and the network topology can become blurred.

Although abstraction is a powerful tool, the operation of the network can be affected significantly by the architecture and implementation of individual components. The details of component architecture and implementation are lost as additional layers of abstraction are applied. To address the loss of accuracy many network simulators allow real implementations of network components to be included in simulation models. For example, ns-2 has been extended to include SystemC models with the aim of providing more accurate simulation results [8]. It has also been argued that hardware and software trade-offs in networked embedded systems cannot be made without reference to the impact on the network [9]. To further improve the accuracy of network simulators, real software network stacks can be included to eliminate the inaccuracies of the default models [10]. For example, ns-2 has been extended to incorporate Click descriptions as part of the simulation [11]. Click is a domain specific packet processing description language that will be presented in detail later in this thesis. The inclusion of Click within the ns-2 simulator allows the code that will be deployed in the target packet processor system to be simulated, giving greater accuracy in terms of event timing and system response.

As networks are complex systems, it is difficult to predict the actual operation of protocols on various topologies. Network simulators allow designers to understand the operation of a network and highlight the interaction between various protocols, whether the interaction is intended or not. As the architecture and implementation of individual packet processors can affect the operation of the network, the design of packet processing systems cannot be validated or verified by solely using network simulators. In order to validate and verify packet processors, these components must be examined as complex systems themselves.

2.3 Packet Processor as a System

Although packet processors are not isolated artifacts, they are complex systems themselves. As a system, packet processors are subject to a variety of constraints including

packet throughput, packet loss, quality of service guarantees, power dissipation, memory utilisation and silicon area requirements. These constraints limit the choices for implementing such systems but the exact nature of the requirements depend on the target application.

Packet processors are used in a variety of applications such as multiplexers, routers, switches, firewalls and intrusion detection systems. They are implemented in a variety of technologies and provide a set of services to the network. For example, Asymmetric Digital Subscriber Lines (ADSL) are frequently multiplexed to provide connections from the local telephone exchange to multiple subscribers. The multiplexing function is performed by a Digital Subscriber Line Access Multiplexer (DSLAM), which parses the packet headers and provides a scheduling policy for multiplexing the connections. The DSLAM system presented by Neogi et al. [12] is based on a network processor but the system could be implemented in other technologies.

Mobile phones increasingly use packet-based transmissions for data communications, which require packet processing systems within the mobile phone basestation. Packet Control Units (PCUs) are frequently used in mobile phone basestations to coordinate packet transfer between the mobile phone and the networking subsystem. The PCU parses packet headers and performs packet encapsulation and decapsulation. Other services provided by the PCU include scheduling packets and controlling the link power algorithms. Again, the PCU system described by Yu-Jie et al. [13] is implemented using a network processor but other technologies could be used.

The Smart Port Card [14] is a processor-based packet processing system, which consists of an embedded Pentium processor running the NetBSD UNIX kernel. The card processes ATM cell streams on a per connection basis and supports active network applications, which allows end systems to alter the behaviour of the network. Code fragments or references to code fragments are received by the processor and are executed as required to alter the behaviour of the network. The ability to change the behaviour of the network permits network management applications to be applied and can alter the behaviour of the network to support other functions as required. Although this system could be implemented in other technologies, the general purpose processor provides the most flexibility for executing code on demand.

The Smart Port Card can be used in conjunction with the Washington University Gigabit Switch [15], which is an ATM switch implemented in an ASIC. This system provides an efficient mechanism for switching packets but the functionality of the device cannot be altered. Conversely, the restriction on reconfiguration and the exploitation of parallelism allows the system to exhibit high data transfer rates and remain power efficient.

Finally, active networking can also be applied to networks-on-chip as described by Vanderbauwhede [16]. Described as a service-based architecture, the system controls the dataflow in a heterogeneous multi-core SoC device and uses a task graph description language to define the operations required for a specific application.

Packet processing systems need to meet a variety of constraints, which can affect the choice of implementation. Each technology has unique strengths that encourage their use in specific applications. However, the diverse range of packet processing applications means that the best implementation technology for a system is not always immediately obvious.

2.4 FPGA-based Packet Processing Systems

While packet processing systems are frequently implemented in ASIC devices or in software, the use of FPGA devices is becoming more popular. FPGAs exhibit the programmability of software and the parallelism of ASIC designs, which make them an interesting option for implementing packet processing systems. The architecture of FPGA-based packet processing systems has been explored for a variety of applications, which include routers, switches, firewalls and intrusion detection appliances.

Routers and switches are essential to the operation of packet-based networks as they forward packets to their eventual destination. Typically, these devices work independently but Ethane [17] and OpenFlow [18] have been proposed as centralised solutions, which creates consistency between appliances. The approach taken by Ethane is to simplify the switches in a network such that they only contain a flow table, which determines where packets should be forwarded. Unknown flows are sent to the central controller for identification and to update the flow tables, which forward packets matching that

particular flow. The central controller makes all forwarding decisions, which are then sent to the switches for implementation.

OpenFlow is based on Ethane but all forwarding decisions are made once for each flow as opposed to each switch requesting independent flow updates. In this simplified system the switches still need to perform packet header parsing and flow identification, which determines whether the packet can be scheduled for transmission or whether a flow update request is made to the controller. In both cases, the switches are implemented using FPGAs on the Field Programmable Port Extender board [19], which allows the functionality of the switch to be tailored to the network application.

Firewalls prevent unwanted access to network resources by blocking specific flows, which requires the firewall to parse packet headers, perform packet classification and potentially scan the packet payloads. A firewall can be created on a single FPGA, as demonstrated by Lockwood et al. [20], which can perform these operations. The firewall is flexible enough to permit exact matching of IP headers and use regular expressions for scanning the packet payload. This particular system also permits the use of HDL plugins to provide additional functionality. The use of a FPGA permits a higher throughput compared to software implementations, while maintaining the flexibility to alter the rules and functionality of the device following implementation.

While firewalls that block flows using packet classification can stop certain attacks on networks, they cannot prevent the transmission of malicious software as the payloads of the packets need to be scanned. Internet worms and viruses can be blocked by scanning the payloads of packets for signatures of malicious intent [21]. Lockwood et al. [22] have proposed a distributed monitoring system that consists of a data enabling device, regional transaction processor and content matching server. The data enabling device is configured by the content matching server and is responsible for searching packet payloads for strings that match the regular expressions provided by the content matching server. There may be multiple data enabling devices in a network and each reports positive payload matches to the regional transaction processor. The regional transaction processor receives reports from the data enabling devices and provides the network administrator with information as to which packet was blocked and why. The data enabling device is implemented on a FPGA, as these devices implement regular expressions efficiently and can cope with

the data rates employed in the network. These devices are also reprogrammable, which allows updated signatures to be received from the content matching server.

While specific packet processing systems have been implemented in FPGA devices, there are also several general purpose packet processing boards that utilise FPGAs. Two such boards are the Field Programmable Port Extender and the NetFPGA board.

The Field Programmable Port Extender (FPX) [19] supports packet processing at the edge of a network switch and consists of two FPGAs, five banks of memory and two network interfaces. One FPGA controls the packet flows and performs routing to and from the various modules on the board. The second FPGA implements dynamically loadable modules which contain specific functionality. The board can be used for active networks in a similar manner as the Smart Port Card but a FPGA bitstream is required as opposed to code fragments. The use of a FPGA allows custom pipelines to be created and the potential for parallel computation to be exploited.

The NetFPGA [23] board uses the logic of a FPGA to implement core data processing functions, and an embedded or external processor to perform the control functions. The board contains two FPGA devices, a PCI interface, banks of memory, a quad-port PHY and two SATA connectors. The board supports communication between the host PC and the user-defined logic in the FPGA through the use of a software driver and the PCI interface.

While most of the research into FPGA-based packet processing systems has focused on using the fabric of the FPGA device, the use of processors within the fabric has not been precluded. Processors implemented within the fabric of the FPGA are commonly referred to as soft processors. The architecture of soft processors differs from ASIC implementations as the relative speed of memory and logic is different between FPGA and ASIC devices [24]. The area cost of implementing a processor is also higher in a FPGA device. However, complex processors can be implemented efficiently using the FPGA fabric as demonstrated by Buciak et al. [25] and Munteanu et al. [26], who implemented a multi-threaded network processor and a network processor with IP compression support respectively.

Many networks require testing to determine whether their performance is satisfactory.

One method of achieving this is by injecting packets into the network and monitoring the response. Packet generators [27] allow different shapes of traffic to be injected into the network with customised packet headers to exercise a specific component or protocol of the network. For example, the firewalls can be tested for the correct application of access control policies to specific ports, and prioritisation protocols can be tested for correctness. The packet generator can also repeat captured traffic so that tests are repeatable.

FPGAs provide the reconfigurability of processors combined with the fine-grained parallelism of ASICs. However, they tend to be less power efficient and require more area for the same functionality implemented in ASIC devices. Even with those limitations, FPGA implementations of packet processing systems provide many advantages for research and commercial systems, as they retain the flexibility of software with a comparative performance improvement.

2.5 System Functions

In order to provide services to the network, packet processing systems need to perform operations on the packets which they receive. These operations frequently include packet classification, forwarding, access control, encryption and framing. The implementation of these operations affects the architecture of the system and the performance of the packet processing system.

Packet classification associates a packet with a particular flow, which defines the source, destination and protocol of the packet. This operation must be performed before forwarding or access control decisions can be made, and the implementation can affect the performance of the system. Due to the critical nature of packet classification to the system's performance, many algorithms have been developed and implemented in various technologies. These algorithms can be classed into exhaustive search, decision trees, decomposition and tuple space [28, 29].

For example, Nikitakis and Papaefstathiou [30] have proposed a decomposition algorithm, which is implemented in a dual stage Bloom filter. The decomposition algorithm decomposes multi-field searches into single field rules, which are combined using multi-level

Bloom filters.

Exhaustive search can be implemented using a specialised hardware circuit called a Ternary Content Addressable Memory (TCAM). Although this circuit provides optimal performance, it is very expensive and power hungry. The circuit is frequently combined with other less expensive alternatives to balance the cost and performance of the device. For example, a TCAM circuit has been combined with a bit vector algorithm [31] to support intrusion detection.

Following packet classification, forwarding operations are frequently performed in many packet processing systems. Forwarding is implemented by performing lookup operations for the flow once the packet has been classified. As with packet classification, there are several algorithms which can be used to obtain forwarding information. For example, a Content Addressable Memory (CAM) can be used. This circuit performs an exhaustive search of the set of flows and returns a result in linear time. However, this circuit does not scale well, is very expensive to implement and might not meet the timing requirements of many applications. Alternatively, a tree bitmap algorithm can be used, which employs a trie data structure for performing IP lookup operations as suggested by Taylor et al. [32]. Tree based algorithms also lend themselves to performance improvements through pipelining as demonstrated by Le et al. [33], who proposed a linear pipeline architecture that uses longest prefix matching.

While packet classification and forwarding are employed in many systems, access control has become an equally important subject in recent years. Access control requires the payload of a packet to be inspected, which is achieved by parsing the payload with a string matching engine. Due to the importance of access control within modern networks, there are several approaches which have been suggested as described by Lin et al. [34]. These approaches are heuristic matching, filtering and the use of automatons. For example, Moscola et al. [35] scan the contents of IP packets by generating a custom finite state machine or automaton that searches for matches to regular expressions. This state machine can be extended to perform search and replace operations in linear time [36]. Schuehler et al. [37] also perform regular expression matching using deterministic finite automata. Both approaches can generate custom state machines from a higher-level description as presented by Mackenzie and Johnson [38].

A filtering approach, proposed by Johnson et al. [39], uses binary decision diagrams to classify patterns. Binary decision diagrams are acyclic graphs, which can be implemented as a series of multiplexers. The multiplexers filter the payload as it is being parsed.

SIFT [40] is another intrusion detection filter, which supports the Snort filtering database. It performs regular expression scanning, header processing and TCP flow reconstruction [41] and uses Bloom filters to perform string matching [42].

Packet processing systems perform a variety of operations on packets, which are required to provide services to the network. Each operation has been the subject of much research and various implementations have been proposed. The implementations have varying throughput, memory requirements and area costs associated with them. However, the architecture of FPGA devices can support a class of algorithms that are infeasible in other implementation technologies.

2.6 Design Environments

Due to the complexity of modern packet processing systems, specialised design environments have been devised to ease the burden of designing, programming and configuring the system. These design environments can be either general purpose or domain specific. Two general purpose design environments that have been created are the Ptolemy project [43] and Hume [44]. Both design environments model the constraints placed on embedded systems and support various representations of time and space independently of the technology used to implement the system.

The Ptolemy project aims to address the design of reactive systems, which includes modelling signal processing, communications and control. These systems are subject to real-time constraints with various components executing concurrently. The Ptolemy project composes systems using multiple domain specific models of computation as no single general purpose model can capture all of the relevant properties. The domain specific models include data flow networks, discrete-event systems, finite state machines and communicating sequential processes. The strength of the Ptolemy project is the separation of the data path and the control path, which allows such a variety of systems to be described.

The Hume [45, 46, 47] project is designed for developing, proving and assessing concurrent, safety-critical systems. While not directly designed for packet processing applications, it can capture some of the properties of network communications. Systems are described using three distinct strongly type languages, which are the declarative language, expression language and coordination language. The declarative and expression languages describe the functions performed by a component and the constraints placed on its operation, while the coordination language describes the communication between components.

While general purpose design environments can provide better representations of systems, further improvements can be made by employing domain specific design environments. Packet processing systems have traditionally been implemented as monolithic systems, which have extended general purpose operating systems. Several researchers have proposed alternative strategies that address modularity, extensibility, flexibility and performance of the system [48]. Three of these design environments are Scout, Router Plugins and Click. Each of these systems separate the datapath description from the control description.

Scout [49] is a modular, communication-centric operating system. Packet processing is performed on paths, which are composed of multiple modules that operate on packets. Modules implement functions such as quality of service or IPv6 processing.

Router plugins [50] is an extension to the NetBSD operating system kernel, which allows modules to be dynamically loaded and configured at run-time. The system supports the notion of flows, which are similar to the paths used by Scout. Each flow is able to load modules independently. The system incurs a performance penalty over the standard kernel but can forward packets up to three times faster than a standard kernel. The standard kernel operates on a best effort basis, which makes no guarantees as to the throughput or latency of packets.

Click [51, 52] is a programming language specifically designed to create modular routers. The language describes the interconnection of elements, which form directed graphs representing the data path. Elements are processing blocks, which perform functions such as Random Early Discard (RED) or traffic prioritisation. The runtime component of Click bypasses the traditional kernel and can sustain a throughput twice that of a Cisco 7200

series dedicated router. Click provides limited support for run-time configuration and does not support run-time module loading as Scout and Router Plugins do.

The Click language is not restricted to describing packet processing applications in software. The language has been extended for use in FPGAs, where the elements are written in VHDL, Verilog or a domain specific networking language [53]. Using Click in this way simplifies the design of hardware components, and allows Click systems to be implemented either in hardware or software providing that the appropriate elements exist.

2.7 Type Checking of IP Interconnection Languages

There are many languages that permit descriptions of IP block interconnections. The trend towards separation of functionality and communication permit systems to be described and implemented more quickly than traditional HDL languages. The languages that describe these interconnections do not always exploit the available information to prevent errors in the target design. High-level design environments can provide more information related to the interfaces of IP blocks and this information can be used to detect errors before compilation and synthesis are executed.

As these languages are focused on describing the interconnection of IP blocks, the errors exhibited are due to incorrect connections. Incorrect connections may be the result of misunderstanding the composition, operation or functionality of the IP block interfaces. Consequently, the aim of verifying the description is to expose incorrect connections before the system is implemented.

One verification technique frequently used in languages is type checking, which requires the creation of a type system [54] and the implementation of a type checker. A form of semantic analysis, type checking is a lightweight formal method which can prove that a class of errors is absent in a given program [55]. This technique will also explicitly state which part of the description is incorrect, allowing the designer to correct the error quickly. Finally, type checking is an automated phase in the compiler that will catch errors earlier in the design flow and save time.

Type checking of IP block interconnections is a property of several design languages.

VHDL, Verilog, SystemC, SystemVerilog, Lava, Coral, IP-XACT and PSF each provide methods for statically verifying connections. The verification methods used in these languages include type checking but other techniques are also applied.

Register Transfer Level (RTL) languages such as VHDL [56] and Verilog [57] verify simple properties of designs through type checking. For example, Verilog verifies that 'wires' are not used to store the results of operations as the data type does not permit blocking assignments. Verification of operands is supported through data types representing the direction of the 'wires' on module interfaces. VHDL provides a comprehensive representation of interfaces using record types. Collections of related wires can be represented as a single complex interface where the direction of each wire and the total number of wires are verified. Neither language contains functional information regarding the wires and consequently cannot detect whether a clock signal has been inadvertently connected to a reset interface.

Transaction Level Modelling (TLM) languages such as SystemC [58] and SystemVerilog [59] support a greater range of data types. SystemVerilog is an extension of Verilog that supports verification of interface types. Interfaces are similar to VHDL records but also include definitions of the transactions supported on the interface. The SystemVerilog type system verifies the structural and transactional properties separately as modules are either transactional or structural descriptions but never both. SystemC supports a similar set of types compared with SystemVerilog. However, the emphasis of SystemC is on transactional descriptions and uses the C++ type system to verify connections.

Lava [60] is a component interconnection specification language, which is embedded in Haskell and based on Hydra[61]. The language describes regular netlists compactly while maintaining relational placement information. As Lava is embedded in Haskell, it uses the Haskell type system to verify connections and permits abstract component representations through the use of type variables. System level extensions [62] have been created that support complex interfaces but these extensions do not verify the functional properties of wires or the payloads transferred.

Coral [63] is a graphical design language that does not use a type system. Instead, it verifies connections through binary decision diagrams. Coral manipulates components with

abstract interfaces and elaborates the properties of those interfaces through descriptions associated with each IP block. The properties defined by the descriptions are encoded as boolean functions, which are then matched using binary decision diagrams. The binary decision diagrams also allow basic glue logic to be inserted, if necessary.

IP-XACT is a XML-based component interface and interconnection description language defined by the SPIRIT consortium [64]. The language supports the definition of complex interfaces and has a defined set of rules for validating connections. Custom scripts may also be used to validate connections and these were the sole source of validation in earlier versions of the standard. Using scripts to validate connections places the burden of ensuring correctness on the designer of the IP block. Unfortunately, the IP block designer is unlikely to anticipate every possible use case for their component, which may result in inconsistent checks and rejection of valid connections.

The Platform Specification Format (PSF) is a collection of proprietary descriptions used by the EDK [4]. The PSF format defines the roles of interfaces and the roles of wires within an interface. The descriptions use nominative typing to verify connections, which allows structural errors to be propagated to the low-level tools. The EDK is also tightly coupled to a predefined set of interface types, which restricts the verification of custom interface descriptions.

2.8 Dynamic On-chip Monitoring

FPGA-based systems are commonly validated through simulation, which exercises models of the final implementation. Unfortunately, simulation cannot guarantee designs to be free of defects and is a time consuming process. In order to provide greater confidence in the system formal verification techniques, such as property checking, equivalence checking and static timing analysis, are applied. These formal techniques complement simulation as they mathematically prove design properties but they frequently rely on the designer to specify the properties of interest. Consequently, even with the plethora of tools available to support the designer, errors can still occur in the final implementation. These errors could be the result of an undetected functional inconsistency or they might be due to variations between silicon implementations of the system. In either case, these errors can

only be detected by observing the execution of the implemented system.

Historically, FPGA designs were comparatively small and on-chip monitoring tools focused on observing low-level information as systems were created using the primitives present in their target device. Even today, designers still create critical sections of their design using device primitives and manually place specific components to meet timing, area and power constraints. Consequently, on-chip monitoring tools have evolved to address the need for low-level information such as signal timing, signal toggle rates and signal propagation. The majority of on-chip monitoring tools are therefore related to tracing, capturing and recording signals in FPGA-based systems.

The available resources in modern FPGA devices permit complex designs, which are more susceptible to errors. While signal tracing is still the most dominant method of monitoring a FPGA-based system, other monitoring techniques have evolved to observe a wider range of errors. On-chip monitoring tools now include support for assertion monitoring, transaction observations, combined monitoring with software and system profiling. Assertion monitoring uses custom circuits to monitor the sequence of signal transitions and reports any illegal sequences. Transaction monitoring is becoming popular in network-on-chip designs where the signalling between components is abstracted by packet switches present in the device. The information from the switches can then be recorded to provide a representation of the communication on a link. Combined monitoring with software allows the designer to observe hardware events in tandem with software executing on a processor, which observes errors that result from operations in either domain. Finally, system profiling is used to understand the communication in a system and to identify any bottlenecks that might prevent real-time constraints from being met.

2.8.1 Signal Tracing

Signal tracing samples the values of signals over a period of time in relation to a clock. It is particularly useful for observing signal transitions in relation to the other signals that form a computation or communication event. Signal tracing infrastructure is now frequently included in ASIC systems and can reuse existing test infrastructure to debug functional errors as demonstrated by Ludewig et al. [65]. This form of signal tracing

requires the system clock to be halted as the system state is recorded externally through an IEEE 1149.1 standard interface, which is commonly known as JTAG. Stopping the clock limits the class of errors that can be detected and is not suitable for applications where real-time constraints are imposed.

Trace buffers can be used to capture samples without stopping the reference clock as samples are stored on-chip. However, the size of the buffer limits the number of samples that can be stored. Hsieh and Huang [66] have proposed a compression mechanism which aims to increase the number of samples that can be stored by employing on-chip compression techniques. This approach has been extended by Kao et al. [67, 68], who employ both compression and abstraction techniques to improve the real-time observations of an AMBA bus in SoC devices.

While these approaches improve the visibility of the system under observation, the points of interest are fixed following synthesis. Quinton and Wilton [69] have demonstrated a programmable debugging module that can alter which signals are monitored following fabrication of an ASIC device. While the debugging module can process information tailored to the needs of the designer, the range of signals that can be observed at run-time needs to be defined before fabrication. As with the tools mentioned previously, the programmable debugging module uses the JTAG interface to communicate with a host computer.

As the JTAG interface is commonly found on FPGA devices for validation by the manufacturer and configuration of the device, the interface has been exploited for use in on-chip monitoring systems. The unused external pins or internal JTAG access ports can be used to form custom scan chains which can be used to stimulate the device [70]. The JTAG interface is also commonly used by commercial monitoring tools such as Xilinx ChipScope [71], which creates monitoring probes within the FPGA fabric. This tool samples signal values in relation to a predefined reference clock and records low-level data. However, ChipScope can monitor any signal in the FPGA fabric and provides a sophisticated triggering mechanism, which allows data capture to be controlled by complex sequences of low-level signal transitions [72]. As a result of its architecture, ChipScope requires a significant number of on-chip memory blocks, which limits the number of samples that can be captured [73]. As packet processing applications tend to use the majority of BRAM

components on a FPGA, it might not be possible to instrument systems using low-level signal trace tools such as ChipScope.

One approach, proposed by Penttinen et al. [74], aims to overcome the buffering limitations of ChipScope by using a microprocessor to sample signals. In their system, the signals of interest are sampled by a microprocessor which processes the data and sends the results over an Ethernet interface. While this approach provides continuous monitoring, the use of a microprocessor limits the sampling rate of the system and requires BRAM resources to store the sampling code.

While readback techniques and trace mechanisms are normally inserted independently of the design environment, several research tools have been developed which relate the hardware observations to the design environment. For example, the Java Hardware Description Language (JHDL) is a structural description language which instantiates the primitives on a FPGA [75]. As JHDL operates at a low-level, it can exploit the readback facility present in some devices to present the sampled values in relation to the components specified by the designer [76, 77]. Another approach demonstrated by Graham et al. [78] instantiates logical probes in the FPGA fabric and uses a low-level library to connect the probes to signals of interest by altering the bitstreams. Finally, JHDL has been extended to include design-level scan chains that allow the state of various IP blocks to be captured using a JTAG scan chain [79].

While JHDL can map data observations to the design specified by the user, the Xilinx Virtual File System (XVFS) creates a representation of the FPGA device primitives as a file system that can be explored on a host computer [80]. As the file system is mapped directly to the device primitives, it does not process the data and requires the system clock to be halted while the state is readback. BoardScope [81] performs a similar set of operations but it displays captured data as waveforms or within a schematic.

Signal tracing mechanisms vary in functionality and resource requirements. Tools tradeoff some features to provide other benefits to the end user. For example, tools sacrifice real-time signal capture for the ability to spatially observe the entire FPGA. Alternatively, resource requirements are increased in order to provide a greater temporal sampling period. Consequently there is no single tool, which can provide appropriate information in

all circumstances.

2.8.2 Assertion Monitoring

Assertions are frequently used to terminate the execution of simulations when an error condition is detected. As simulation can be time consuming, it is possible to implement assertions directly in the hardware. For example, Bartzoudis and McDonald-Maier [82] have implemented an assertion monitor in a FPGA-based daughter card, which checks for PCI protocol and application errors. Assertion circuitry can be synthesised from a property description language and automatically inserted into a design as demonstrated by Gharehbahi et al. [83]. Finally, Straka et al. [84] have implemented an assertion checker that validates the LocalLink protocol used in Xilinx FPGAs. As with the tool by Gharehbahi et al., this tool takes a description of the protocol and automatically synthesises the circuit to perform the analysis.

Assertion monitoring in FPGA designs tends to focus on monitoring low-level signals. Existing tools do not allow assertions to be described in relation to high-level events at disparate locations in the system. While assertion monitoring is very useful for catching errors in a design, the technique needs to be combined with a form of data capture in order to observe and understand what is happening in the system. Thus, an assertion monitor could form the trigger of a signal capture tool.

2.8.3 Transaction Observations

Signal tracing is a useful monitoring technique for observing low-level details with cycle accuracy but the sheer volume of data can overwhelm the designer. Transaction-level monitoring addresses the volume of data by processing it to obtain higher-level data and is increasingly being used to monitor the buses in SoC devices and NoC applications.

Goossens et al. [85, 86, 87, 88, 89, 90, 91] have proposed a transaction monitoring infrastructure for the Aethereal Network-on-Chip, which monitors the communication between packet switches in the network and abstracts the low-level details of the communication between components. Their approach reuses the network fabric to transmit the processed

information, which can affect the performance of the system under observation. The transactions are obtained by reusing the interpretation circuitry already present in the switches.

The architecture of network-on-chip monitoring tools permits observations to be controlled by triggers at varying locations on-chip as suggested by Tang et al. [92]. Cross-triggers allow patterns of communication in one location of the system to initiate observations in other parts of the design. The system-level communication can also be combined with detailed observations of a single IP block, which can improve the designers understanding of the operation of a component and its interaction with the system [93].

Transaction monitoring can aid the designer to understand the system under observation. A high-level perspective of the operation of the system can be obtained but the implications of transaction level monitoring are dependent on the target technology. Systems composed of indirect interconnects, such as a network-on-chip, exhibit communication patterns that may change over time as the system executes. However, direct interconnects, used frequently in FPGA-based designs, cannot change the target of their communication so monitoring circuitry cannot reuse the existing infrastructure in this instance.

2.8.4 Combined Monitoring of Hardware and Software

The tools presented thus far in this chapter have focused solely on monitoring the hardware signals in the system. As many applications require the use of processors, debugging techniques need to support combined observations of hardware and software operations in tandem.

Roesler et al. [94] have developed a tool, which integrates software debuggers with gate-level debug tools in the JHDL environment. However, this tool captures low-level information that is difficult to relate to the symbols used by the software debugger. The tool uses the readback capability of the FPGA to observe the hardware components and requires the operation of the circuit to be halted while reading back the current values. The combination of monitoring hardware and software can be extended to include emulation as suggested by Yu and Zou [95]. Again using JHDL, the design is decomposed

into submodules which can be emulated independently while the remainder of the design is simulated. This provides a performance improvement while maintaining visibility and accuracy for key locations of interest.

The BEE2 platform [96, 97] is a multi-FPGA system, which has been created for reconfigurable computing applications. It is designed to be modular, scalable and uses a software design methodology. BEE2 abstracts FPGA designs as user processes via the BORPH operating system. The debugging environment allows assertions and breakpoints to be created and variables to be recorded through pipes. The debugging environment is the same regardless of whether the application is implemented in hardware, software or both. This system abstracts low-level details for debugging purposes but is focused on recording system state, which also hides the communication events from the user.

Combined hardware and software monitoring can produce significant amounts of data. This data rate can be reduced by only recording the events that can alter the execution of the system as presented by Hochberger and Weiss [98]. The data requirements for combined monitoring are reduced by restricting observations to peripheral components and interrupts as only events in these locations can change the program flow unpredictably. The software and other hardware properties are recorded using a separate in-circuit emulator, which can be used to deduce the operation of components that are not monitored directly.

Although there are tools for combining the observations of software and hardware, most tools focus on capturing low-level hardware observations. These observations are difficult to relate to the functions provided by the software and require high data rates. In order to assist the user further, the hardware events need to be presented to the user in a form which is more suited to software debugging. Transaction-level events might be the most appropriate form to represent hardware operations in conjunction with software functions.

2.8.5 Profiling

Profiling is the dynamic analysis of a system's behaviour, which highlights bottlenecks and determines the computationally intensive functions in the design. The information obtained through profiling can aid the designer in deciding whether a function should

be implemented in software or hardware. There is a wide range of profiling tools, which collect diverse sets of information. A classification of profiling tools has been proposed by Tong and Khalid [99], which groups profiling tools into software-based, hardware-based and FPGA-based implementations.

Software-based profilers, such as Gprof [100] and Valgrind [101], develop profiles of function call graphs and memory accesses. Gprof instruments the software application, which alters the target being observed and reduces the accuracy of the results obtained. Valgrind simulates the processor, which is very slow compared to other profiling techniques. The accuracy of Valgrind is also dependent on the accuracy of the simulation model, which tends to be architecture agnostic.

Hardware-based profilers, such as those found in Intel processors [102], provide similar information to that obtained by software profilers. The inclusion of counters in hardware improves the accuracy of the results obtained and negates the need for the target application to be instrumented. As with the software-based tools, only the software application can be profiled, which prohibits comparisons of performance with the microarchitecture of the system.

FPGA-based profiling tools tend to be more flexible by permitting the profiling information to be customised to the needs of the designer and target application. Tools, such as SnoopP [103] and Airwolf [104], provide counters in the FPGA fabric to profile software functions. The results obtained are cycle accurate and do not require the software to be altered. Analysis of software loops can also be performed using the Frequent Loop Analysis Tool (FLAT), which identifies the most frequent short backwards branches in the software application.

FPGA-based profiling tools also allow accurate profiles of hardware signals in conjunction with the software application [105]. Both Hough et al. [106] and Padmanabhan et al. [107] describe methods of monitoring software functions and recording the value of hardware signals, which provide an overview of the hardware performance in relation to the software applications. Nunes et al. [108] describe a profiler which observes the message passing between components in the BEE2 platform regardless of whether the component is implemented in hardware or software.

Watches Over Data STreaming on Computing element linKs (WOoDSToCK) [109, 110, 111, 112] is an FPGA-based profiling tool that records the buffer utilisation between computing processor elements. Assuming a single interface type with buffering, the system adds monitors to detect cycles when links are stalled or starved. This approach uses communication-centric monitoring and abstracts the low-level signalling from the designer but does not record individual transactions. It provides useful information for identifying bottlenecks but permits only limited observability of transactions between components.

Using the same interface type as WOoDSToCK, it is possible to stimulate a system using an on-chip testbed to reduce verification time [113]. It has been shown that run-time traffic can be emulated using a hardware testbed, which is infeasible with the software equivalent. Such a hardware testbed can be used in conjunction with a monitoring system to provide insight into the operation of a system in a controlled environment.

There are many profilers available to analyse software applications and the supporting hardware. These tools can provide useful information to optimise applications and reduce performance penalties. However, these tools do not focus on the interconnection between IP blocks. WOoDSToCK monitors the connection between components but it is restricted to monitoring the buffer utilisation and does not suggest the frequency of specific operations.

2.9 Automated Instrumentation Techniques

With the range of on-chip monitoring tools available to observe systems, there has been little work to explore automatic instrument of those systems for observation. For example, design environments, such as Coral [63], Platform Express [114], Xilinx Embedded Development Kit [4] and Cliff [53], provide unique abstractions for representing components but none provide support for automated instrumentation. The EDK is the only tool with limited support for on-chip monitoring systems. However, it only permits components to be manually inserted using abstract interfaces and does not provide any configuration information to the monitoring software.

Systems can also be instrumented using software provided by the on-chip monitoring

tools. The majority of these tools focus on capturing low-level information, which implies that the instrumentation is performed on low-level signals. On-chip monitoring tools with software instrumentation include JHDL [75], ChipScope [73] and Identify [115].

The JHDL implementation flow provides a mechanism for monitoring the design on-chip but it requires significant guidance from the designer to determine which signals to monitor. The insertion of monitoring circuitry is performed on netlist descriptions where the semantics of signals are unknown and low-level data is recorded. However, the circuitry is inserted as part of the design flow once the signals of interest have been specified.

Xilinx ChipScope can be manually inserted in HDL descriptions as a component in designs. Alternatively, it can be inserted by a specialised program, which instruments netlists. In both cases ChipScope requires the designer to specify the locations of interest and to manually connect the probes to the controller IP block. Furthermore, configuration is performed through a separate software tool, which does not automatically contain the information used to instrument the design.

Synplicity's Identify inserts monitoring circuitry that provides similar functionality to JHDL. However, it allows the designer to specify locations of interest as breakpoints in the HDL descriptions. Consequently, it is able to partially automatically insert monitors. This tool does relate events to the RTL design environment but the design environment does not contain the information found in high-level design environments.

2.10 Requirements for Monitoring Packet Processing Systems

As designs become more complex, the range of errors that might be present in the implementation and the effort required to correct those defects increases. On-chip monitoring tools have been designed to observe defects in the implementation of systems but they have traditionally focused on recording low-level information and monitoring a single location. There are two disadvantages to these low-level monitoring tools. First, low-level monitoring produces vast amounts of information, which can be difficult to comprehend.

Secondly, to configure and use these tools effectively, the designer needs a detailed understanding of the low-level signalling within the system. The designer might not necessarily have a detailed understanding of the low-level signalling as modern designs increasingly rely on libraries of IP blocks. Frequently, these IP blocks are inserted into designs as black boxes, although the function of the component is known. The internal signals and their required behaviour are normally unknown to the user. Thus, a designer integrating IP blocks is unlikely to be familiar with the details of the interface standards and associated low-level signalling of every IP block forming the system.

Furthermore, designers do not need to be familiar with the internal signals of IP blocks as they are well tested and validated. IP blocks are also used in a variety of systems, giving the designer greater confidence that the component is functionally correct. Although the IP block may function correctly in isolation, it might introduce errors when integrated with other components. Typical integration errors range from mismatched timing constraints to undesired component interactions. The method of monitoring systems created in high-level design environments should therefore focus on observing component interactions and abstract low-level signalling.

In addition to these requirements, a monitoring mechanism needs to be unintrusive. This means that the monitoring facilities must not interfere with the computation or communication of the system; They must be passive. For example, many embedded systems interact with external stimuli, where the validity of computations is dependent on the time that the result is calculated. Within the packet processing domain, many applications need to maintain connections, which requires responses to be generated within a given time constraint. If the system is halted then no response can be generated and the connections are lost, which can alter the behaviour of the system and prevent the observation of any errors.

Finally, many packet processing systems employ processors to perform non-critical functions. While software and hardware are normally considered distinct branches of engineering, the interaction between the two disciplines is becoming critical to delivering systems on time. Consequently, a monitoring system designed to observe packet processing systems should be able to observe hardware and software operations simultaneously.

Passive system-level transaction monitoring is one technique that fulfils these criteria. It provides four main benefits, which address some of the limitations of low-level tools.

First, transaction-based monitoring abstracts low-level details into high-level events, which addresses the need for understanding low-level information. Consequently, the low-level signalling does not need to be recorded as the high-level event embodies the same information.

Second, monitoring low-level signal transitions requires high data rates for transferring data off-chip. This may also require a significant number of external pins, which might not be available for debugging purposes. By using a transactional representation, the low-level details can be abstracted from the designer and may reduce the data rate and the number of external pins required.

Third, communication-centric monitoring only records the interactions between IP blocks, which can highlight misconceptions concerning component intercommunication. It also reduces the amount of data recorded by limiting the monitoring points and avoiding changes to the components. This information can be used to recommend a location of interest for further low-level monitoring, if required. It can also be used to monitor software by observing the communication between the processor and memory.

Fourth, transaction level monitoring requires fewer resources. Smaller resource requirements will generally reduce the impact on system timing and ease the burden of routing the design. Consequently the instrumentation is less intrusive and less likely to impact the system's behaviour. Furthermore, the debug infrastructure must also remain within the resource limitations of the monitored device, as otherwise it cannot be observed on-chip.

Thus, the requirements for debugging a complex FPGA-based packet processing application are communication-centric monitoring, abstraction of low-level details and a small resource footprint. The observations of the system should also be passive to allow the correct operation of the system to be monitored.

2.11 Debugging Methodology

Validation of systems through on-chip monitoring is a difficult task that is normally performed in an adhoc manner. The methods employed tend to vary depending on the experience of the designer and the time available. Due to the difficulty and cost of correcting errors at this stage in the design flow, several attempts have been made to formalise the technique for debugging systems through on-chip monitoring. Araki et al. [116] have suggested a methodology for debugging concurrent software applications which can be extended to FPGA-based systems. Concurrent software applications are composed of functional units that exchange data amongst themselves through channels. This separation of communication and computation is similar to the high-level design environments being employed to described FPGA-based packet processing systems. Josephson [117] has suggested a methodology for debugging silicon systems, which refers to low-level implementation defects but can also be applied to functional errors.

Araki et al. proposed two phases of debugging applications: localisation and correction. Localisation requires the designer to determine the location of a defect within a system and correction requires the cause of the error to be determined. Both phases are iterative and tend towards a solution. Localisation begins with a set of hypotheses, which defines the causes that could lead to an error. The set of hypotheses is formed from error reports and any monitoring tools available to the designer. An attempt is made to reproduce the error by selecting a hypothesis and testing whether it holds true. If the hypothesis holds true then the defect has been localised, otherwise the hypothesis is eliminated and another is selected for testing. Due to the concurrent nature of the application, it can be difficult to reproduce the events that caused the defect as each functional unit is executing independently. Araki et al. also warn that care should be taken to ensure that the application behaviour is not affected by monitoring its execution.

Correction of the defect involves an iterative process that refines the hypothesis. As the defect has been localised, the cause of the defect needs to be determined. The hypothesis is tested by modifying the application to reflect the proposed solution. If the solution rectifies the problem then the defect has been corrected, otherwise the hypothesis is refined and tested again.

Josephson has outlined three stages, which detect and correct defects in silicon systems. These stages are characterisation, triage and debug. Characterisation is the process of testing the device and obtaining information on the operation of the system in a range of environments. Triage determines the modes of failure, which can be classified into separate bins. For example, the results of varying the voltage and frequency of a device can characterise the performance of the system. Detected failures can be placed into bins to group related defects of the same type. Following characterisation and failure classification Josephson suggests that debugging can then be performed. The debugging process consists of five distinct stages which are control, isolation, root cause, expansion and correction. The failure is controlled by determining the variables that might contribute to the cause of the error and isolated by determining when and where the failure occurs in the system. Following these stages, there should be sufficient information available to determine the root cause of the failure, which can be expanded to eliminate other errors in the class. Finally, the error is corrected using a technique appropriate to the system in question.

2.12 Summary

Networks are employed in a variety of applications, which vary dramatically in scale. Independent of their size, networks are complex systems that are composed of nodes which perform packet processing operations. The network topology and the protocols employed determine the performance of the network but the implementation of the nodes can also have a direct impact.

The nodes within the network are complex systems themselves and can be implemented in a variety of technologies. The architecture of the nodes affects their throughput, memory requirements, power consumption and cost. The balance between these requirements is dependent on the target application and the implementation technology. The operations performed by packet processing applications also impact the characteristics of the system. Several algorithms have been suggested for packet classification, forwarding and access control mechanisms. Furthermore, this chapter has also discussed several design environments that can be employed to create packet processing systems.

Specifically, this chapter has presented existing type systems used to statically verify designs created in other languages. It has also presented on-chip monitoring systems that a used for monitoring systems in FPGAs and other technologies. Several existing techniques for automatically instrumenting designs have been presented and the requirements for on-chip monitoring have been discussed. Finally, two debugging methodologies have been discussed.

Chapter 3

Implementation Flow for High-level Design Environments

High-level design environments extend the traditional FPGA design flow to provide domain-specific abstractions. In this thesis a high-level design environment refers to a tool or set of tools that use descriptions of the communication between functional components or IP blocks to generate designs that are used by the traditional implementation flow. The functional components can be written in abstract languages that are compiled to traditional RTL descriptions or they can be written directly as traditional RTL descriptions or netlists. Extending the traditional FPGA design flow reuses the sophisticated functionality provided by the relatively low-level implementation tool chain, while providing features applicable to the target domain. Initially, the tools comprising the low-level implementation flow will be discussed and complementary low-level validation and verification tools will be presented. Following this, the implementation flow of two high-level design environments for packet processing applications will be explained. Finally, the validation and verification of high-level design environments will be examined and the contribution of this thesis will be highlighted.

3.1 Introduction

The design flow for implementing FPGA-based systems is complex but well-established [118, 119, 120]. The flow has been used extensively in numerous designs and is supported by a variety of tool vendors. The implementation tool chain consists of several steps, which transform designs from device independent descriptions through to bitstreams that are used to configure the FPGA. Validation and verification is well-supported by the design flow, and each step provides an opportunity to prove specific properties of the system being implemented.

Due to the complexity of implementing FPGA-based systems, high-level design environments extend the existing design flow. Extending the implementation tool chain reuses the functions of the design flow and exploits the existing techniques for verifying systems during implementation. Furthermore, it provides an interface for implementing systems that can be shared between different domain-specific high-level design environments. Sharing the low-level implementation flow allows designers to retain control of low-level aspects of the system, if they so desire.

The techniques for validating and verifying high-level systems are immature as high-level design environments are new extensions to the existing design flow. The information obtained from traditional validation tools is not related to the abstractions and representations used in the high-level design environments. In order to verify a system effectively, the techniques need to be related to the design environment. High-level tools also present more opportunities for detecting errors before the traditional FPGA design flow is executed. These opportunities arise because high-level design environments contain additional information about the intended system that is discarded by the low-level implementation flow. This thesis contributes one technique for verifying systems before synthesis is executed and a method of relating on-chip monitoring to the high-level design environment. It also contributes a technique for automatically instrumenting a design by exploiting the additional information found in high-level design environments. These techniques catch more errors than are presently caught by the low-level implementation flow and will save time by catching errors earlier than the traditional implementation flow.

3.2 Low-level Implementation Tool Chain

Register Transfer Level (RTL) descriptions are typically used as inputs to the traditional FPGA design flow. These descriptions specify the transfer of data between registers and the logical operations performed on that data. Transformations need to be applied to these RTL descriptions to configure the individual configuration memory elements found in a FPGA. The algorithms for performing these transformations are complex and consequently form separate stages of the design flow. As shown in Figure 3.1, these phases are synthesis, translation, mapping, placement, routing and bitstream generation. Each phase translates the design into a more specialised description, which eventually targets a specific FPGA device.

Synthesis transforms a RTL description into a structural netlist. Such RTL descriptions are normally written in Hardware Description Languages (HDLs), such as VHDL [56] and Verilog [57]. These languages permit behavioural descriptions of systems to be written and require the synthesis tool to infer the structural implementation. For example, a state machine can be described as a sequence of state transitions using RTL descriptions. The synthesis tool would then infer the structural composition of the state machine, which would allow the system to be implemented. As already mentioned, the resultant netlist is a structural representation of the system, which describes the set of components and their interconnections. At this stage of the design flow the specified components are device independent logic elements, such as decoders, flip-flops and logic gates. An HDL description does not need to use behavioural descriptions and may explicitly instantiate these generic components or device specific primitives, which removes the need for inference.

Within the Xilinx FPGA design flow, two netlist formats are frequently used. These are the Electronic Data Interchange Format (EDIF) and the Native Generic database and Constraints (NGC) file. Both formats describe component interconnections but the proprietary NGC format also includes constraint information. Although netlists are designed to describe component interconnections, the descriptions are explicitly stated and do not support many of the features found in high-level IP block interconnection languages such as conditional instantiation, replication and complex interfaces.

Following synthesis, translation merges multiple netlists into a single description, called

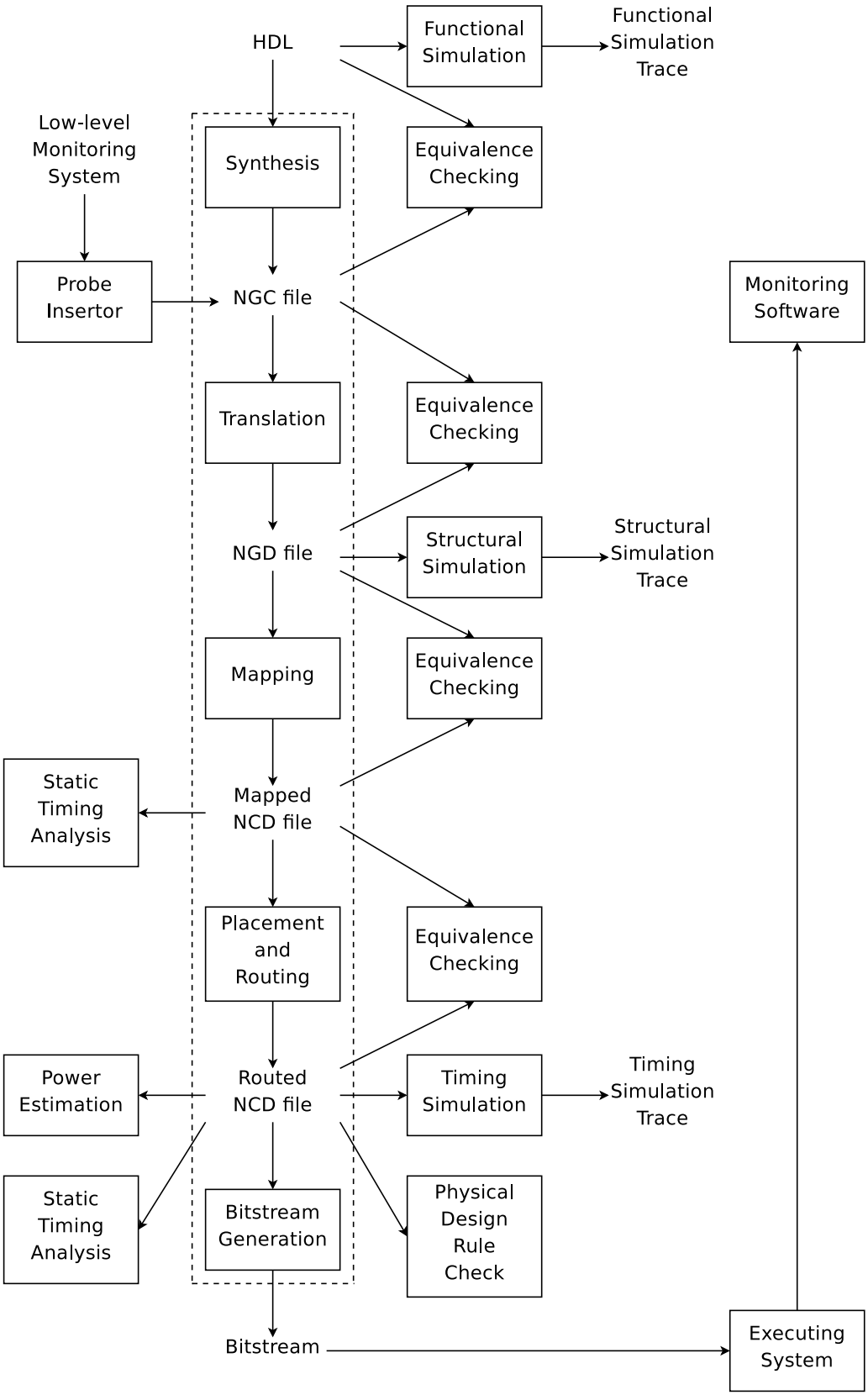

Figure 3.1: The Xilinx FPGA design flow including validation and verification points.

the Native Generic Database (NGD). It also includes any additional constraints from Netlist Constraints Files (NCFs) while maintaining the system hierarchy. After translation, the netlist must be mapped to the primitives supported by the target device. Mapping produces a netlist that refers to logic cells, I/O cells and other hard IP blocks found on the target device. Within the Xilinx tool chain the result is a Native Circuit Description (NCD) file.

At this stage in the design flow the netlist refers to specific device primitives. However, it does not specify the location of those primitives on the device, nor does it describe the paths for interconnecting those primitives. Placement and routing converts the mapped netlist to a placed and routed description. Placement assigns primitives to specific locations on the target device, while routing allocates resources for interconnecting the placed primitives. Due to the finite resources on the FPGA the placement and routing algorithms are heavily interdependent. Consequently, both functions are executed as part of the same tool in the Xilinx design flow.

Having obtained a placed and routed netlist the final step is to convert it to a bitstream that configures the target FPGA. The bitstream is a binary file that contains configuration information for all of the programmable logic elements and additional data, such as the initial values of components.

3.3 Validation and Verification of the Low-level Tool Chain

As the low-level design flow is well-established, several tools for validation and verification have been developed that complement the existing flow. The techniques that are commonly applied include simulation, power analysis, timing verification, equivalence checking, design rule checking and on-chip monitoring, as shown in Figure 3.1.

Simulation can be performed at most stages within the design flow with an increase in accuracy as implementation progresses. Figure 3.1 shows the three main classes of simulation, which are functional, structural and timing simulation. Functional simulation, which does not contain any timing information, is used to exercise HDL descriptions and validate the operation of the design. Structural simulation is executed on translated

netlists, which simulate the system using the inferred generic components. Again, as the netlist is not placed, no timing information is present. However, the increased accuracy of structural simulation incurs a computation penalty as more processing is required to simulate the system. Frequently this means that structural simulation takes more time to execute than functional simulation. Finally, timing simulation uses a placed and routed netlist to simulate device primitives. This form of simulation contains accurate timing information from the netlist but incurs a further computation penalty.

Simulation is a validation technique that cannot prove the absence of defects. However, it can detect errors within execution traces and is a popular technique as it provides complete spatial and temporal visibility of the system. Usually, systems are simulated at multiple levels of abstraction with the same stimulus to give confidence that the implementation and subsequent transformations are correct.

Power analysis is an important validation tool as power consumption has become a critical constraint in some FPGA-based designs. Power analysis is performed using a routed netlist, which defines the length of wires and the connection impedance. It also verifies that junction temperature limits will not be exceeded and ensures that the device's thermal limits will not be surpassed.

The satisfaction of timing constraints in synchronous designs can be formally verified by static timing analysis tools. Within the Xilinx tool flow, TRACE formally proves that all connections in a design meet timing constraints. It determines the maximum frequency for a given circuit and calculates the critical path. Timing estimates can be generated at various stages in the design flow but formal verification is performed on a placed and routed netlist.

As the design flow is comprised of separate tools, formal equivalence checkers can be used to verify the output of each tool. Formal equivalence checkers statically compare HDL descriptions, netlists and other files to detect design inconsistencies. Equivalence checking does not verify the functional properties of the design. Instead, it ensures that the outputs generated by the implementation tools do not deviate from the input descriptions. Formal equivalence checkers can verify that the final bitstream matches the initial HDL description, detecting errors introduced by the implementation tool chain.

The Xilinx design flow also includes a set of Design Rule Checks (DRCs), which are a series of tests for validating the system. These checks form part of the various tools in the flow but they can also be executed independently. The DRCs can be categorised into logical and physical checks. Logical checks are device independent and are executed as part of translation. These checks validate structural properties such as blocks, nets, pads, clock buffers, names and primitive pins. Physical checks uncover electrical design errors and are performed as part of mapping, placement and routing, and bitstream generation. The checks include the examination of signals for floating segments, antennae and checking the placement of signals on external pins.

Finally, on-chip monitoring is used to observe the system as it is executing on the target device. On-chip monitoring is used to record functional, logical and physical errors that were undetected during implementation. Traditional monitoring tools observe individual signals by placing monitoring circuitry on-chip or by redirecting connections to external pins. Additionally, on-chip monitoring allows data sequences to be captured and signal timing to be observed. This technique also provides greater accuracy and executes more quickly than simulation but has comparatively limited visibility. Furthermore, these tools may require external buffering and sampling equipment.

3.4 High-level Design Environments

As the available resources of FPGA devices increases, the abstractions provided by RTL languages become less suited for describing complex systems. As this trend continues, the productivity of designers diminishes and the time required to create designs increases. High-level design environments aim to increase designer productivity by raising the abstractions provided and emphasising the reuse of pre-existing components. These environments also encourage components to be described using specialised domain-specific languages but this does not preclude the use of traditional RTL and netlist descriptions.

To encourage component reuse and foster the adoption of domain-specific languages, high-level design environments tend to separate the specification of functionality and communication. Unlike netlist descriptions, the interconnection of each wire does not need to be stated explicitly. High-level interconnection specification languages typically support fea-

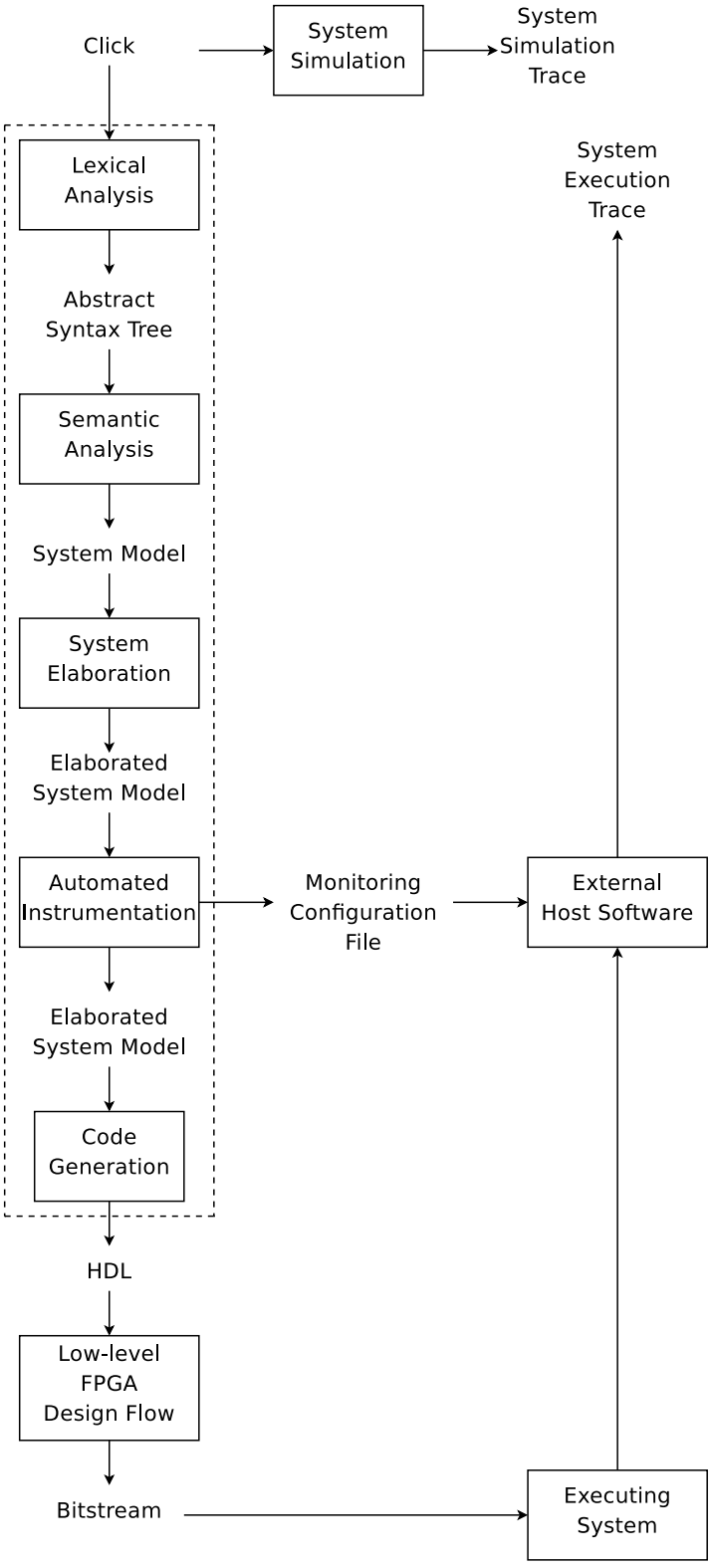

Figure 3.2: An overview of the design flow of System Stitcher including validation and verification points.

tures such as parametrised regular structures, hierarchy and conditional instantiation but do not permit the description of component functionality. The design environments may complement the language capabilities through support for features such as component inference and datapath optimisation.

This thesis is concerned with two specific high-level design environments that target the packet processing domain. These tools are called Brace and System Stitcher. Both tools are proprietary to Xilinx and have been developed as part of the research into high-level design environments. Both tools use the language Click [51, 52] to instantiate, configure and connect IP blocks. Click is intended solely for specifying the interconnection of components and has no support for describing component functionality. Although this may seem restrictive, it encourages separation between the description of component functionality and the system interconnection. The separation of component functionality and system interconnection encourages reusable components to be created, which can be tested in isolation. Systems can then be formed using well-tested components suggesting that any errors are the result of integrating components as opposed to isolated functional errors.

Brace and System Stitcher share a similar architecture for creating high-level designs. The architecture of System Stitcher is shown in Figure 3.2, which is also representative of the architecture of Brace. These tools are composed of several stages, which include lexical analysis, parsing, system elaboration and code generation. The output generated in both cases is a set of files that describe the individual IP blocks and an HDL description of the system interconnections. The outputs can then be used as an interface to the traditional design flow, which generates the final bitstream.

3.4.1 Brace

Brace was initially designed for assembling packet processing systems but it has been extended to act as a generic IP block interconnection tool. Brace extends the capabilities of the Xilinx Embedded Development Kit (EDK) and supports automated implementation from Click descriptions through to bitstreams. The EDK provides a library of verified IP blocks that can be customised and tailored to meet specific requirements. The complexity

of the IP blocks in the library ranges from simple logic gates to complex processors. Brace utilises this library and also supports the inclusion of custom IP blocks, which might have been developed in other tools.

Brace also provides support for inferring components. For example, it can automatically determine the clock requirements for designs and configure the appropriate Digital Clock Modules (DCM). Brace is also capable of automatically inferring reset controllers and correctly matches logic levels using knowledge of the reset signals provided to the FPGA. It also connects output reset signals from the controller to each IP block using the correct logic levels. Finally, Brace also exploits board-level information, which reduces the burden of connecting designs to external components.

3.4.2 System Stitcher

System Stitcher was designed exclusively for assembling packet processing systems. It restricts the interface types that can be used on IP blocks and supports packet processing components written in other high-level languages. System Stitcher interfaces directly with the low-level implementation tools and co-ordinates the generation of IP blocks written in other high-level languages. This tool also incorporates a facility for co-ordinating system simulation where each component can be described at varying levels of abstraction.

3.5 Validation and Verification of High-level Designs

As systems become more complex the need for validating and verifying correctness becomes critical. High-level design environments provide representations that lend themselves to alternative validation and verification techniques. Two existing techniques that are commonly applied to modern high-level designs are transaction level modelling and model checking.

Transaction Level Modelling (TLM) is frequently used to model systems using representations more abstract than RTL [58, 59]. These representations provide a clearer understanding of the functionality of the system and permit quicker simulations compared to RTL descriptions. The simulation techniques have also evolved to provide comprehensive

functional coverage of the system under test [121, 122]. As mentioned in section 3.3, high-level simulations can be compared to simulation traces of low-level implementations providing confidence that the implementation is correct. However, without formal verification, there is no guarantee that a low-level implementation matches the transaction level model. To complicate matters further, the translation from TLM descriptions to RTL implementations is a manual process. However, tools are being developed to automate this process [123, 124].

Model checking verifies the functional properties of an abstract design by exhaustively exploring the state space of the system. This technique can be applied to HDL descriptions but is more commonly applied to more abstract representations. Abstract models are commonly used to reduce the number of states that need to be explored, which reduces the execution time of the tool. Furthermore, the required abstractions of model checkers may ignore important properties of the system. Model checking requires the designer to specify the properties of the system but if the designer omits a property then the tool cannot catch any errors relating to it. Finally, a model of each component in the system is required. However, these might not be available from third party IP vendors.

This thesis is concerned with the validation and verification of high-level design environments. It presents three concepts that extend the validation and verification capabilities of the research tools, Brace and System Stitcher. The concepts proposed are type checking, system-level transaction monitoring and automated instrumentation.

3.5.1 Type Checking

High-level design environments contain more information related to the interconnection of components compared to the low-level implementation flow. Unfortunately, this information is discarded as the implementation flow progresses. Type checking performs semantic analysis of IP block interconnections statically, which saves time by eliminating the need to execute the low-level implementation flow when errors are detected. Additionally, a class of errors permitted by the low-level flow is detected by the type system.

Brace and System Stitcher both use semantic analysis and type checking to prevent errors. System Stitcher uses a constrained set of types that target packet processing systems. This

thesis proposes a type system for Brace that uses the structural and payload properties of interfaces to verify connections. The type system has been implemented as part of the compilation flow of Brace and is a contribution of this thesis.

3.5.2 System-level Transaction Monitoring

The reconfigurable nature of FPGAs encourages rapid prototyping in the target hardware, which allows systems can be exercised more quickly than in a simulation environment. Furthermore, the final implementation is observed, not an abstract model. Unfortunately, traditional monitoring tools focus on signal transitions and require significant on-chip resources.

A system-level transaction monitoring system for observing packet processing systems has been designed and implemented. The monitoring system interprets the sequences of signal transitions on component interfaces as events that can be related to the abstractions used by the high-level design tool. In particular, the events reported by the monitoring system can be related to a separate system simulation environment developed as part of System Stitcher. The monitoring system also minimises the extra resource penalty of monitoring, while providing flexibility for observing a variety of component interconnections. The low resource penalty allows the monitoring system to be used in a wider range of designs than low-level monitoring tools.

3.5.3 Automated Instrumentation

On-chip monitoring systems can be inserted into designs at various stages of the implementation flow. However, the insertion of monitoring circuitry is a manual process as low-level tools cannot determine which signals to monitor. The lack of semantic information in low-level tools also precludes the configuration of monitoring software to interpret the signals being monitored. High-level design environments contain semantic information regarding component interfaces that can be exploited to correctly instrument a design for system-level transaction monitoring. Additionally, the high-level environment contains sufficient information to configure the monitoring tool, allowing signals to be correctly interpreted.

The semantic information in high-level design environments can be exploited to perform automated instrumentation of systems, which does not require user intervention and complements the semantic information of the design tool. The instrumentation algorithm has been implemented as part of System Stitcher and uses the type system to correctly insert the monitoring circuitry.

3.6 Summary

High-level design environments tend to extend the existing low-level FPGA design flow, as it is complex but well-understood. The low-level design flow is also complemented by several techniques for validating and verifying systems. Consequently, high-level environments exploit the features already present and provide additional functionality. This chapter has presented the architecture of two design environments developed internally within Xilinx. Finally, the contributions of this thesis have been explained and their applicability to the design flow has been shown. In particular, an overview of type checking, system-level transaction monitoring and automated instrumentation has been given.

Chapter 4

Static Verification of IP Block Interconnections

High-level design environments contain information that is either not available or used in low-level tools. This information is used during compilation to create low-level representations of the design and it is subsequently discarded once the low-level design flow has been invoked. Properties of the design can be verified before execution of the low-level flow by exploiting the information available in high-level design environments. Detecting errors before implementation reduces synthesis iterations, saves time and increases designer productivity.

This chapter presents a type system that statically verifies the interconnection of IP blocks by exploiting the high-level information available in Brace. The type system itself is formally presented and an implementation of a type checker is explained. The chapter also demonstrates that type checking in high-level design environments catches more errors and catches errors earlier than the traditional design flow. Throughout this chapter, an IP block is also interchangably referred to as a component, referring to its use as a component within a larger system.

4.1 Click: A Domain Specific Language

Click was originally created to improve the performance of software-based routers. By separating the communication and computation of modules, Click improved the performance of software routers compared to traditional monolithic implementations [51, 52]. Cliff [53] aimed to further improve the performance of routers by exploiting the parallelism inherent in FPGA devices and by using Click to maintain the separation between computation and communication. Brace is an extension of Cliff, which provides further abstractions of the hardware. The intention is to provide a means of describing the system irrespective of whether the implementation is in software or directly in hardware. Although Brace was originally intended to assemble packet processing applications, it is capable of assembling systems comprised of IP blocks from multiple application domains. The hardware abstractions used in Brace are also not limited to the packet processing domain.

Click is an untyped declarative language that describes the communication between modules. The computation of Click systems is defined by the functionality of the constituent modules, known as elements, which are written in other languages. Within Brace, the modules or elements are discrete IP blocks that are instantiated in the FPGA and chosen from a library provided by the EDK. The library can be extended with user defined IP blocks, which can be described in a language suited for the target application. The communication described by Click is implemented as a physical connection between IP blocks. There is a direct mapping from the Click description to the hardware implementation. Click also supports hierarchical descriptions through element classes. The element class is a set of connected elements and is treated like any other element in the system. The interfaces of an element class consists of the unconnected interfaces of the elements comprising the element class.

The following code shows the instantiation of an IP block that represents a BRAM on a FPGA. The syntax consists of an identifier, *inst0*, and the type of the IP block, which is *bram_block*. This identifier is used to refer to the instance of the IP block throughout the remainder of the design. The type refers to an IP block that is available in the library. The instantiation does not define the interfaces of the IP block as these are not explicitly

defined in the Click syntax. Rather, the interfaces are specified as part of the IP block library using an extended version of the PSF description.

```
inst0 :: bram_block;
```

Brace includes some extensions that are not present in the the traditional Click syntax. The original Click syntax represents each interface on a module by a unique number. In Brace, an unique alphanumeric identifier is used, which is defined by the component definition in the IP library. The following code shows the syntax for connecting two IP blocks that have already been instantiated. The code specifies that the *LLTX* interface on element *inst1* should be connected to the *LLRX* interface on element *inst2*. As previously mentioned, the composition of the interface is not stated explicitly, as it is defined by the IP library.

```
inst1 [LLTX] -> [LLRX] inst2;
```

The Click syntax has also been extended to allow individual wires within an interface to be accessed. The following code shows individual wires and vectors of wires being accessed and connected. The first line states that wire *0* from the port *data* on element *inst1* is connected to wire *0* from the port *data* on *inst2*. The second statement specifies that the range of wires *(4:1)* from port *data* on *inst1* are connected to the range of wires *(7:4)* on port *data* on element *inst2*.

```
inst1 [data(0)] -> [data(0)] inst2;
inst1 [data(4:1)] -> [data(7:4)] inst2;
```

The code shown in Figure 4.1 describes a packet processing system, which consists of three IP blocks. These IP blocks are a bidirectional LocalLink buffer, Ethernet MAC and a physical layer device (PHY). The IP blocks have already been defined and are included in the IP library. The resultant packet processing system is shown graphically in Figure 4.2.

The statement in line 1 of the code in Figure 4.1 declares the instantiation of the LocalLink buffer. The buffer is of type *ll_xgmac_buffer* and has been given an instance name of

```
 1: frame_buffer :: ll_xgmac_buffer (C_TX_FIFO_SIZE 512,
 2:                                  C_RX_FIFO_SIZE 512);
 3:
 4: mac          :: xgmac           (MANAGEMENT_INTERFACE true,
 5:                                  STATISTICS_GATHERING false);
 6:
 7: phy          :: xaui            (C_MODE         ETHERNET,
 8:                                  C_CLK_FREQ     156250000,
 9:                                  MDIO_INTERFACE true);
10:
11: input [CONFIG] -> [MDIO_CONFIG] phy;
12: mac   [HOST]   -> [HOST]        output;
13: mac   [MDIO]   -> [MDIO]        phy;
14:
15: input [LLRX]  -> [LLRX]      frame_buffer [CLIENT_RX]
16:               -> [CLIENT_RX] mac          [CLIENT_TX]
17:               -> [CLIENT_TX] frame_buffer [LLTX] -> [LLTX] output;
18:
19: mac [ XGMII ] -> [ XGMII ] phy [MGT] -> [MGT] output;
```

Figure 4.1: Code describing example Click system comprised of three IP blocks.

frame_buffer. The key and value pairs in parentheses are parameters used by synthesis tool to define the depth of the buffers. The subsequent two statements on lines 4 and 7 instantiate and configure the Ethernet MAC and PHY. The Ethernet MAC is of type *xgmac* with an instance name of *mac*, while the PHY is of type *xaui* with an instance name of *phy*. Following the instantiation of the three IP blocks, the connections comprising the system are described in lines 11 through 19.

The statement on line 11 specifies that the *MDIO_CONFIG* interface on the instance *phy* is an input to the system, which is called *CONFIG*. Similarly, line 12 specifies that the interface *HOST* on instance *mac* is an output from the system. The system inputs and outputs are connections to the external pins of the FPGA. Line 13 specifies that a

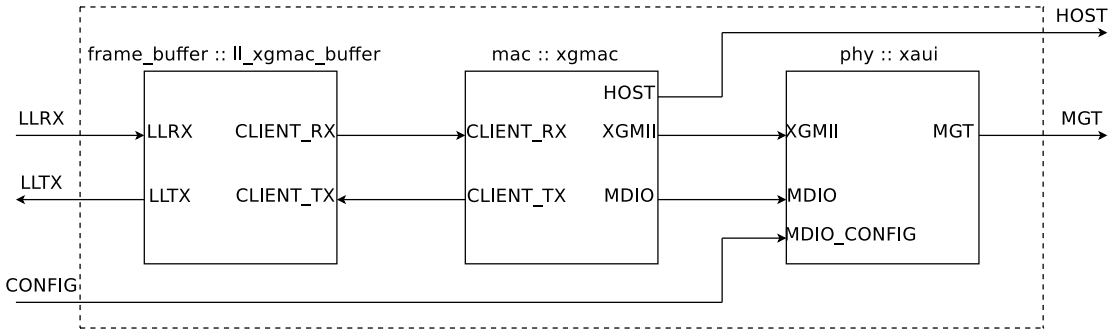

Figure 4.2: Graphical representation of Click system comprised of three IP blocks.

connection exists between the *MDIO* interface of the instance *mac* and the *MDIO* interface of instance *phy*.

As demonstrated by the previous examples, the connections specified by Click are direct. Indirect connections and shared-media are supported by creating direct connections to IP blocks representing those media. For example, buses are instantiated as IP blocks with separate interfaces for masters, slaves and monitors. The EDK also represents shared-media as IP blocks but it does not define separate interfaces for different classes of connection. Rather, the EDK makes a connection to the bus and attempts to resolve connections using internal definitions of the bus. Consequently, the EDK cannot verify the connection of IP blocks to shared-media and places the verification burden on Brace.

Brace is a high-level design environment that supports the design and implementation of packet processing systems. Its aim is to improve designer productivity by abstracting low-level implementation details and by providing a clear representation of the datapath. Brace is a continuation of the work on Cliff[53], which uses the domain-specific language Click[51, 52] to assemble FPGA-based packet processing systems. Brace is not limited to packet processing applications and can be used to assemble systems comprised of IP blocks from multiple application domains. It also provides a complete implementation flow from Click descriptions to bitstreams.

Brace is an extension to the Xilinx EDK and reuses the implementation flow provided by the EDK. Brace itself consists of several compilation stages, which include lexical analysis, parsing, semantic analysis, elaboration and code generation, as shown in Figure 4.3. Lexical analysis and parsing create a system model from the Click description. The system

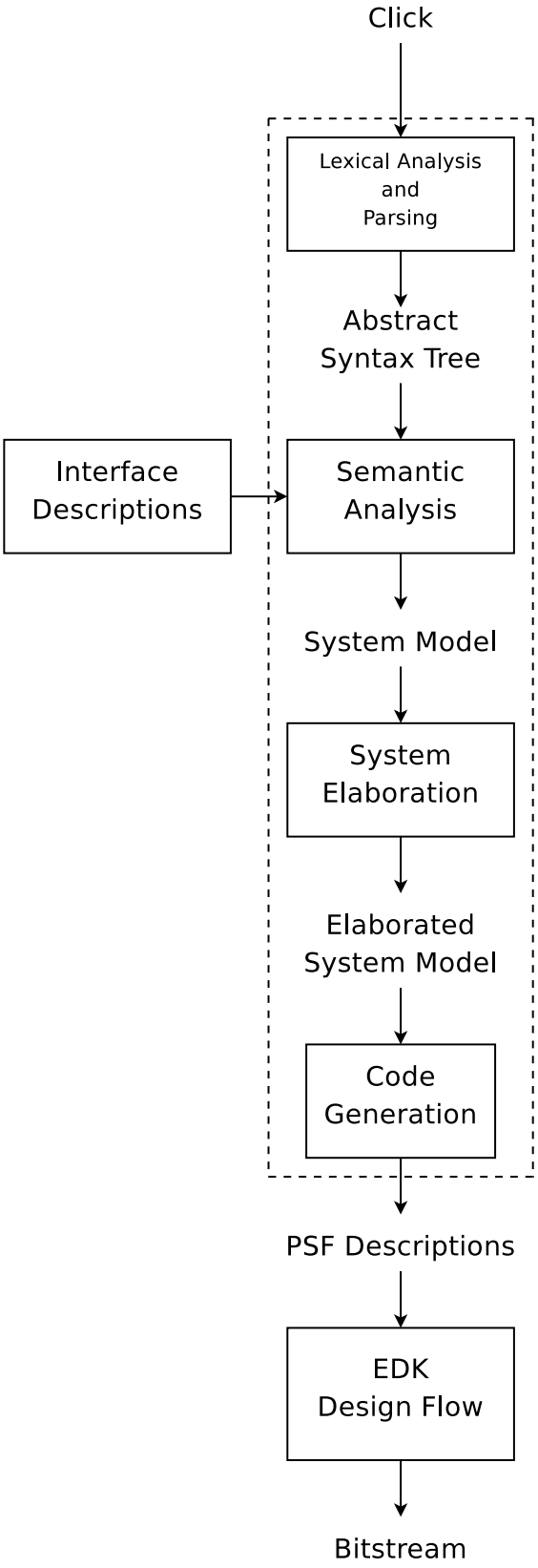

Figure 4.3: The Brace implementation flow.

model is an abstract representation of the components in the design and their interconnections. Elaboration performs a variety of functions that can be optionally disabled by the user. These elaboration functions include component inference and interface resolution, which resolve clock and reset requirements and create structural representations of interfaces respectively. Following elaboration, code generation creates a description of the system in the PSF format, which is then used by the EDK and the low-level FPGA design flow as shown in Figure 4.3.

4.2 Type System

Due to its software heritage, Click does not define types for IP block interfaces. In the original Click system, elements communicate through the use of function calls, which pass pointers on the stack. Each element is implemented as a subclass of the element class, which uses the C++ compiler to ensure that data is passed correctly. The lack of type information in Click can pose a problem for hardware designs as interfaces might not be compatible. The untyped Click syntax permits incompatible interfaces to be connected, which results in unconnected signals being tied to V_{GND} or V_{CC} as appropriate. However, the system is unlikely to operate correctly when interfaces are erroneously connected.

As part of the Brace implementation flow the EDK performs limited validation of connections. The EDK uses a simple nominative type system to validate connections but this type system does not verify the structural properties of the connected interfaces. The EDK also relies on custom validation scripts to ensure that connections are correct, in a similar manner to IP-XACT. However, it is infeasible to expect an IP block designer to anticipate every potential connection that might be made to their component. The verification of connections should be supported by the system description language and detected by the compiler.

A static structural type system has been implemented in Brace to address the weak nominative type checking in the EDK and the untyped nature of the original Click syntax. Type checking is a lightweight formal method, which proves the absence of a class of errors. In this case, the type system proves the absence of specific errors in connected interfaces. The type information is obtained from the extended PSF descriptions in the IP library

and made available to the compiler during semantic analysis. Structural typing ensures that the composition of an interface determines its compatibility with other interfaces. The type system must be static as Click is a declarative language that is not executed. Finally, the type system has been implemented as a type checker, which is executed as part of semantic analysis in the Brace design flow.

Click has a single function, which is the connection function $\rightarrow$. The inputs to the function are typed identifiers that refer to component interfaces. Applying type checking to this function will verify that connections are valid and will explicitly state which parts of the system description are incorrect. Stating incorrect connections will allow the designer to correct errors quickly. However, the system designer might decide to intentionally connect incompatible interfaces as defined by the type system. Intentional incompatible connections should be declared explicitly in the system description to show that the designer has deliberately intended to make the connection. An annotation to the Click syntax could be used to permit a form of type casting or to disable type checking for that connection.

The type system is presented by describing the grammar of the language and presenting the type rules using the inference rule format as described by Pierce[55] and Cardelli[54]. The grammar defines the set of acceptable statements that comprise all possible system descriptions and the type rules determine the preconditions that must be met for a statement to hold true.

Type rules are comprised of premises and a conclusion, which state the preconditions that must be satisfied before the conclusion can hold. The preconditions and conclusion are stated as judgements, which consist of the static typing environment, Γ, and the free variables declared in that environment. The empty environment is represented by ϕ. The judgement used most frequently in this thesis is the typing judgement, which has the following form:

$$\Gamma \vdash M : A$$

The judgement above states that the term M has a type A with respect to the static typing

environment Γ. Another frequently used judgement simply states that an environment is well formed. This judgement is demonstrated as follows:

$$\Gamma \vdash \diamond$$

A type rule asserts the validity of a conclusion judgement based on the precondition judgements that are already known to be valid. The precondition judgements are specified above a horizontal line with a single conclusion judgement below the line. Only when all of the preconditions are satisfied can the conclusion hold. For example, a multiplication function requires two inputs, which must be of the same type or converted to the same type before the function can be performed. The result of the function is a value which has the same type as the inputs. The types required by the function are demonstrated in the following type rule example. Note that the type rule does not specify the semantics of the function.

$$\frac{\begin{array}{l}(\textsc{Val}\ \times) \\ \Gamma \vdash M : A \qquad \Gamma \vdash N : A\end{array}}{\Gamma \vdash M \times N : A}$$

The example states that the function $M \times N$, which returns a result of type A, requires two preconditions to be satisfied. First, the environment must contain a variable M of type A. Second, the environment must contain a variable N, which is also of type A. Only when both conditions are satisfied can the conclusion hold. In other words the function $\times$ can only be performed when both variables exist with the same type that the function requires. For a more detailed explanation, please refer to Cardelli[54] or Pierce[55].

A simplified grammar of Click is shown in Table 4.1. The grammar follows the conventions used by Pierce [55], which defines the terms and types. The key statements include component declaration, interface projection and interface connection. The declaration of a component creates a component type, which is a product of the interface types for each interface on the component. Interface projection selects an interface from the set of interfaces on the component and interface connection is the connection function, which

Table 4.1: Simplified Click syntax of Brace

A::=		Types
	$PhysicalType \times Payload$	Interface type
	$\{l_i : A_i \ ^{i \in 1..n}\}$	Component type
D::=		Declarations
	$I :: A_1, \ldots, A_n$	Component declaration
C::=		Commands
	$I_1 \rightarrow I_2$	Interface connection
	$C_1;C_2$	Subsequent commands
	I[l]	Interface projection
I		Identifier

requires two interfaces as inputs. The interface type is itself a product type and the structure of this type will be discussed in detail. The primitive types used in the type system are listed in Table 4.2.

The type system is presented graphically in Figure 4.4 and the fundamental rules for the type system are shown in Table 4.3. In Figure 4.4 boxed items represent composite types or record types, unboxed items represent primitive types and items enclosed in ellipses group primitive types to highlight the logical relationships.

Rule *Env* ϕ, shown in Table 4.3, states that the empty environment is well-typed and rule *Env* I specifies that an identifier, I, with type A is well-typed, provided that the identifier does not already exist in the environment. Rule *Decl Component* declares an identifier with a component type and rule *Comm Subsequent* states that two commands are valid in the environment. As Click is not a sequential language, the rule means that multiple connections can be stated in a single Click description. Finally, rule *Proj Interface* permits the type of one interface in a component to be projected for use in connecting the components.

The rule for connecting interfaces is not shown explicitly as each interface type is a product of multiple primitive types and would need to be presented as a whole. Due to the large number of permutations that would result, the connection rules for each primitive type are presented individually to maintain clarity.

Table 4.2: Primitive Types

PhysicalType::=		Physical type
	Input	Direction types
	Output	
	Bidirectional	
	Role	Role types
	AnyRole	
	Clock	Signal types
	Reset	
	Interrupt	
	Signal	
	High	Sensitivity types
	Low	
	Insensitive	
	Falling	
	Rising	
	Big	Endian types
	Little	
	Initiator	Bus role types
	Target	
	Master_Slave	
	Master	
	Slave	
	Monitor	
Payload::=		Payload types
	Routable	Payload kind
	Window	
	Control	
	Stream	
	AnyPayloadKind	
	PayloadType	Payload type
	AnyPayloadType	

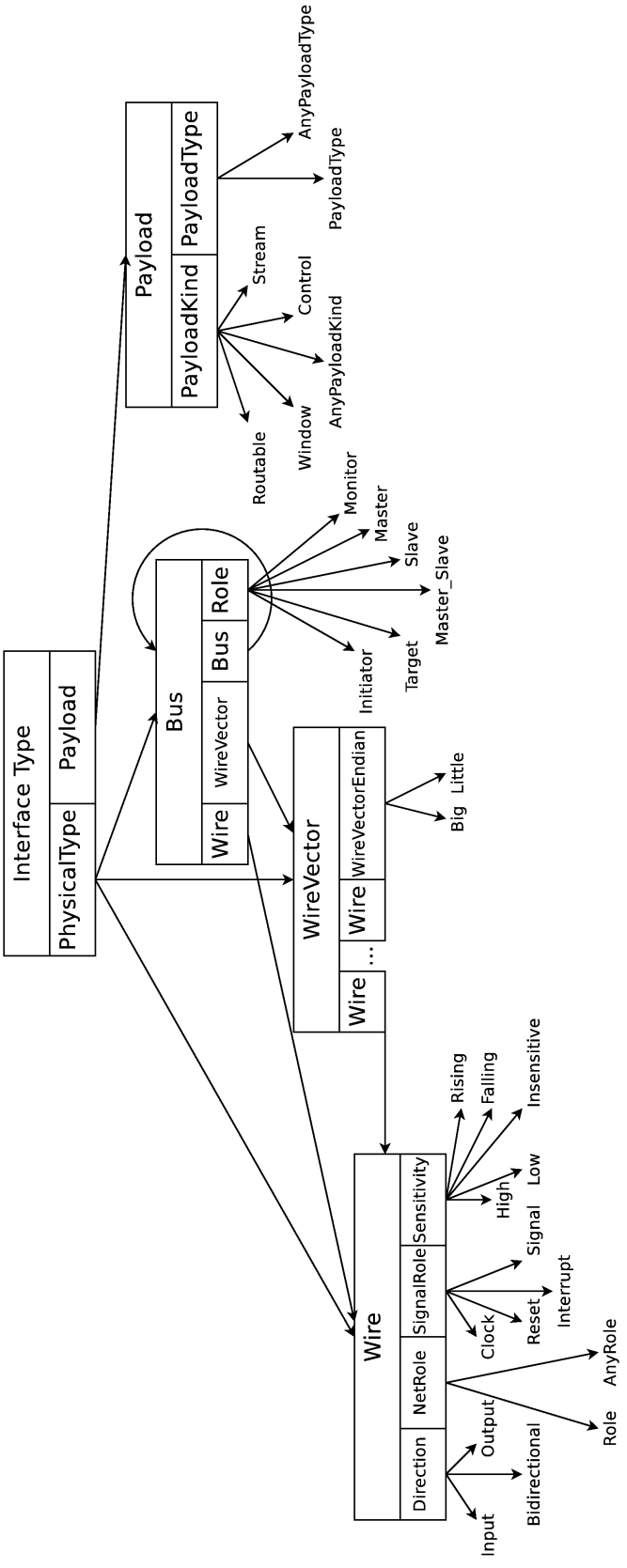

Figure 4.4: Graphical representation of the Click type system

Table 4.3: Overall type rules

$$\frac{}{\phi \vdash \diamond}\ (\textsc{Env}\ \phi) \qquad \frac{\Gamma \vdash A \qquad I \notin dom(\Gamma)}{\Gamma, I : A \vdash \diamond}\ (\textsc{Env I})$$

$$\frac{\Gamma \vdash \{l_i : A_i^{i \in 1..n}\}}{\Gamma \vdash (I :: A_1, \ldots, A_n) : (l_1 : A_1, \ldots, l_n : A_n)}\ (\textsc{Decl Component})$$

$$\frac{\Gamma \vdash C_1 \qquad \Gamma \vdash C_2}{\Gamma \vdash C_1; C_2}\ (\textsc{Comm Subsequent})$$

$$\frac{\Gamma \vdash I_1 : \{l_i : A_i^{i \in 1..n}\} \qquad j \in 1..n}{\Gamma \vdash I_1[l_j] : A_j}\ (\textsc{Proj Interface})$$

Table 4.4: Interface type rules

$$\frac{\Gamma \vdash PhysicalType \qquad \Gamma \vdash Payload}{\Gamma \vdash PhysicalType \times Payload} \text{(Type Interface)}$$

4.2.1 Interface Typing

An *Interface* type is a product of the *PhysicalType* and *Payload* type, as shown in Table 4.4. The *PhysicalType* captures the structural properties of the interface such as the composition of wires, it also embodies the properties of each wire within the interface. The *Payload* represents the data transferred over the interface. A valid connection is determined by the product of the *PhysicalType* and *Payload* type, which verify orthogonal aspects of the interface. Both types are labels for other complex data types and these will be discussed individually.

As the IP blocks may be implemented from varying levels of abstraction, the *Interface* type is designed to capture both high-level properties and structural details. The *Payload* has an emphasis on packet processing as Brace was designed for this domain but it also supports other application domains. For implementation, the interfaces need to be represented structurally regardless of whether the component was initially described in an high-level language or low-level representation. The type system uses all of the available information to verify connections but separates concerns in order to ease the verification problem.

4.2.2 Physical Type

The *PhysicalType* represents the physical structure of the interface and is an algebraic data type or variant type. As an algebraic data type it may be either a *Wire*, *WireVector* or *Bus*. The *Wire* type represents a single wire, the *WireVector* type is a set of identical wires and the *Bus* type captures complex interfaces. The rule *Variant As* in Table 4.5 states that a *PhysicalType* is either a *Wire*, *WireVector* or *Bus*, which prevents errors

Table 4.5: Physical type rules

(Type PhysicalType)

$$\frac{\Gamma \vdash Wire \qquad \Gamma \vdash WireVector \qquad \Gamma \vdash Bus}{\Gamma \vdash PhysicalType(Wire, WireVector, Bus)}$$

(Variant PhysicalType As)

$$\frac{\Gamma \vdash PhysicalType(Wire, WireVector, Bus) \qquad \Gamma \vdash j \in \{Wire, WireVector, Bus\}}{\Gamma \vdash PhysicalType_j}$$

such as connecting a wire to a wire vector or a bus. Without this rule, the subsequent implementation flow may silently remove wires from interfaces. Again, *Wire*, *WireVector* and *Bus* are labels for other data types.

4.2.3 Bus Type

Bus interfaces typically consist of wire vectors for data transfer and additional wires to perform control functions. A bus may also be composed of other buses. For example, two unidirectional interfaces may be concatenated to form a bidirectional interface. The *Bus* type is a recursive product type that represents complex interfaces composed of wires, wire vectors and other buses.

Specifically, a *Bus* is composed of a set of *Wire* types, a set of *WireVector* types, a set of *Bus* types and a *Role*. For two interfaces to be compatible, the sets of wires, vectors and buses must be related by a bijection, which is defined as a one-to-one correspondence. For every *Wire* in one interface there must be a compatible *Wire* in the other interface. This implies that the sets of *Wire* types, *WireVector* types and *Bus* types on both sides must be the same size.

Traditionally, hardware designers omit wires that are not needed in an interface to minimise logic. Omitting wires suggests that the interface does not require the full set of transactions defined by the interface standard. For example, a processor may have two master connections to a bus in order to access memory. One connection may be used for

Table 4.6: Bus type rules

(Type Bus)

$$\frac{\Gamma \vdash Wire_1 \ldots \Gamma \vdash Wire_n \quad \Gamma \vdash WireVector_1 \ldots \Gamma \vdash WireVector_m \quad \Gamma \vdash Bus_1 \ldots \Gamma \vdash Bus_o}{\Gamma \vdash \{l_i : Wire^{i \in 1..n}\} \times \{l_j : WireVector^{j \in 1..m}\} \times \{l_k : Bus^{k \in 1..o}\} \times BusRole}$$

data accesses and the other may be used for instruction fetches. In order to save logic the designer might not include the write signals for the instruction interface as they are not used during system operation. However, synthesis tools are very adept at removing redundant logic and should identify unused functionality. Designers should include all wires to demonstrate that the interface is compatible with the bus although write operations are not performed. For slave interfaces omitted signals indicate that not all transactions that might be requested are supported. Using the type system, the designer would be alerted to the incompatibility and appropriate remedial action could be taken.

The *BusRole* defines the purpose of an interface and is divided into two incompatible categories, *Direct* and *Shared*. A *Direct* interface can either be an *Initiator* or a *Target*. A *Shared* interface can be a *Master*, *Slave*, a *Master_Slave* (both master and slave) or a *Monitor*. *Direct* interfaces match the opposite type whereas *Shared* interfaces match the same type, as shown in Table 4.7. Rule *Comm Direct Connect* states that initiators must be connected to targets for point to point connections. Bidirectional point to point connections are usually comprised of two unidirectional buses and the constituent buses will have *Initiator* and *Target* roles. However, the enclosing bus type will be *Master_Slave* as it encompasses both directions and the types should match to demonstrate compatibility. *Shared* interfaces are intended to connect to instantiations of buses, which is reflected in rule *Comm Shared Connect*. This rule states that a shared-medium must provide different interfaces for masters, slaves and monitors. Separating the interfaces allows the signals for each interface type to be clearly defined. The interfaces of components that support both master and slave operations can be represented by using the *Master_Slave* type, which will contain concatenated *Master* and *Slave* bus types.

Table 4.7: BusRole type rules

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash Initiator}\ (\textsc{Type Initiator}) \qquad \frac{\Gamma \vdash \diamond}{\Gamma \vdash Target}\ (\textsc{Type Target}) \qquad \frac{\Gamma \vdash \diamond}{\Gamma \vdash Master}\ (\textsc{Type Master}) \qquad \frac{\Gamma \vdash \diamond}{\Gamma \vdash Slave}\ (\textsc{Type Slave})$$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash Master\_Slave}\ (\textsc{Type Master\_Slave}) \qquad \frac{\Gamma \vdash \diamond}{\Gamma \vdash Monitor}\ (\textsc{Type Monitor})$$

$$\frac{\Gamma \vdash I_1 : Initiator \qquad \Gamma \vdash I_2 : Target}{\Gamma \vdash I_1 \rightarrow I_2}\ (\textsc{Comm Direct Connect})$$

$$\frac{\Gamma \vdash I_1 : A \qquad \Gamma \vdash I_2 : A \qquad A \notin \{Initiator, Target\}}{\Gamma \vdash I_1 \rightarrow I_2}\ (\textsc{Comm Shared Connect})$$

Table 4.8: WireVector type rules

$$\frac{\Gamma \vdash Wire \qquad \Gamma \vdash WireVectorEndian}{\Gamma \vdash \{l_i : Wire^{i \in 1..n}\} \times WireVectorEndian}$$ (Type WireVector)

4.2.4 WireVector Type

The *WireVector* type is composed of a set of identical *Wire* types and a *WireVectorEndian*, as shown in Table 4.8. As with the *Bus* type, sets of *Wire* types are related by a bijection and must be of the same size. Matching the size of the sets eliminates unintentional connection of subranges, which is permitted by other languages such as Verilog. Mismatched vector sizes are undesirable as most vectors are parameterisable and are usually determined by the result of an expression. If the expressions for the vectors do not yield the same size then errors can go unnoticed without static verification. The *Wire* type is identical for all terms of the product type.

The *WireVectorEndian* type refers to the location of the most significant bit within the *Wire* vector, which can be *Big* or *Little*. It does not refer to the word order of data being transferred. As shown in Table 4.9, the endian of vectors must match. The *Wire* type is a label for another data type which will be explained in the following section. Again, in order to capture the size of the vector, the *Wire* types must be identical within the vector as shown in Table 4.8.

4.2.5 Wire Type

The *Wire* type is the product of the *Direction*, *SignalRole*, *NetRole* and *Sensitivity*, as depicted in Table 4.10. The *Direction* can be an *Input*, *Output* or *Bidirectional*. Rule *Comm Direct Connect* in Table 4.11 shows that an *Input* must be connected to an *Output* type and vice versa. A *Bidirectional* type can be connected to any other *Direction* type including another *Bidirectional* type as demonstrated by the *Comm Bidirectional Connect* rules. During synthesis, bidirectional wires are expanded into separate input and output

Table 4.9: WireVectorEndian type rules

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathit{Little}}\ (\textsc{Type Little Endian}) \qquad \frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathit{Big}}\ (\textsc{Type Big Endian})$$

$$\frac{\Gamma \vdash I_1 : A \qquad \Gamma \vdash I_2 : A}{\Gamma \vdash I_1 \rightarrow I_2}\ (\textsc{Comm Connect})$$

Table 4.10: Wire type rules

$$\frac{\Gamma \vdash \mathit{Direction} \qquad \Gamma \vdash \mathit{SignalRole} \qquad \Gamma \vdash \mathit{NetRole} \qquad \Gamma \vdash \mathit{Sensitivity}}{\Gamma \vdash \mathit{Direction} \times \mathit{SignalRole} \times \mathit{NetRole} \times \mathit{Sensitivity}}\ (\textsc{Type Wire})$$

wires controlled by multiplexers. Consequently, for FPGA designs bidirectional wires are infrequently used.

The *SignalRole* describes system-level functions such as clocks, resets and interrupts. The primitive types embodied by the *SignalRole* are *Clock*, *Reset*, *Interrupt* and *Signal*. As demonstrated in Table 4.12, each primitive type must match against an identical type. The primitive types are self-explanatory as they describe common system-level functions but they may also be included within an interface. The *Signal* type embodies all other signals, which do not have a system-level function. For example, an interrupt controller may define an *Interface* type consisting of a *WireVector* that represents each interrupt priority available in the system. As described previously, the *WireVector* will have a *Wire* type composed of an *Input* and an *Interrupt*. The *SignalRole* allows common functions to be defined and verified across a range of IP blocks.

The *NetRole* type defines the use of a signal within an interface. As shown in Table 4.13, it can be either a specific *Role* or *AnyRole*. A specific *Role* defines a signal with a specific

Table 4.11: Wire Direction type rules

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash Input}\ (\textsc{Type Input}) \qquad \frac{\Gamma \vdash \diamond}{\Gamma \vdash Output}\ (\textsc{Type Output}) \qquad \frac{\Gamma \vdash \diamond}{\Gamma \vdash Bidirectional}\ (\textsc{Type Bi-Directional})$$

$$\frac{\Gamma \vdash I_1 : Input \qquad \Gamma \vdash I_2 : Output}{\Gamma \vdash I_1 \rightarrow I_2}\ (\textsc{Comm Direct Connect}) \qquad \frac{\Gamma \vdash I_1 : Input \qquad \Gamma \vdash I_2 : Bidirectional}{\Gamma \vdash I_1 \rightarrow I_2}\ (\textsc{Comm Bidirectional Connect 1})$$

$$\frac{\Gamma \vdash I_1 : Output \qquad \Gamma \vdash I_2 : Bidirectional}{\Gamma \vdash I_1 \rightarrow I_2}\ (\textsc{Comm Bidirectional Connect 2})$$

$$\frac{\Gamma \vdash I_1 : Bidirectional \qquad \Gamma \vdash I_2 : Bidirectional}{\Gamma \vdash I_1 \rightarrow I_2}\ (\textsc{Comm Bidirectional Connect 3})$$

Table 4.12: SignalRole type rules

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash Interrupt}\ (\textsc{Type Interrupt}) \qquad \frac{\Gamma \vdash \diamond}{\Gamma \vdash Output}\ (\textsc{Type Signal}) \qquad \frac{\Gamma \vdash \diamond}{\Gamma \vdash Clock}\ (\textsc{Type Clock}) \qquad \frac{\Gamma \vdash \diamond}{\Gamma \vdash Reset}\ (\textsc{Type Reset})$$

$$\frac{\Gamma \vdash I_1 : A \qquad \Gamma \vdash I_2 : A}{\Gamma \vdash I_1 \rightarrow I_2}\ (\textsc{Comm Connect})$$

Table 4.13: NetRole type rules

$$\frac{\Gamma \vdash \diamond \qquad NetRole \in Roles}{\Gamma \vdash NetRole}\ (\textsc{Type NetRole}) \qquad \frac{\Gamma \vdash \diamond}{\Gamma \vdash AnyRole}\ (\textsc{Type AnyRole})$$

$$\frac{\Gamma \vdash I_1 : NetRole \qquad \Gamma \vdash I_2 : NetRole \qquad NetRole \notin \{AnyRole\}}{\Gamma \vdash I_1 \rightarrow I_2}\ (\textsc{Comm Specific Connect})$$

$$\frac{\Gamma \vdash I_1 : NetRole \qquad \Gamma \vdash I_2 : AnyRole}{\Gamma \vdash I_1 \rightarrow I_2}\ (\textsc{Comm Any Connect})$$

purpose within an interface. The set of *Role* types is defined by the IP library, which allows the set to be tailored to the application domain. The type system defines the set of rules for connecting interfaces and does not limit the range of roles that can be used. As shown by rule *Comm Specific Connect* in Table 4.13, *Role* types must match. The *AnyRole* type can be matched against any other *NetRole* type as shown by rule *Comm Any Connect*. This capability is useful for generic IP blocks such as "not" gates where the signal may be used in a variety of roles. It will also allow a clock operating at a specific frequency to connect to an IP block with a generic clock requirement, where the *SignalRole* is *Clock* in both cases.

The *NetRole* type prevents misconnections of incompatible wires. For example, the type system would detect an erroneous connection between a wire indicating the start of a frame and a wire indicating the end of a frame. It will also prevent two clock signals operating at different frequencies from being connected. However, it is flexible enough to permit generic components to be connected without excluding valid connections.

The *Sensitivity* states on which value an event is triggered and whether the wire is synchronous to the clock. As shown in Table 4.14, the *Sensitivity* may be *High*, *Low*, *Rising*, *Falling* or *Insensitive*. *High* and *Low* types are synchronous to the clock whereas *Rising*

Table 4.14: Wire Sensitivity type rules

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash High}\ (\textsc{Type Level High}) \qquad \frac{\Gamma \vdash \diamond}{\Gamma \vdash Low}\ (\textsc{Type Level Low}) \qquad \frac{\Gamma \vdash \diamond}{\Gamma \vdash Rising}\ (\textsc{Type Rising Edge})$$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash Falling}\ (\textsc{Type Falling Edge}) \qquad \frac{\Gamma \vdash \diamond}{\Gamma \vdash Insensitive}\ (\textsc{Type Insensitive})$$

$$\frac{\Gamma \vdash I_1 : A \qquad \Gamma \vdash I_2 : A \qquad A \notin \{Insensitive\}}{\Gamma \vdash I_1 \rightarrow I_2}\ (\textsc{Comm Sensitive Connect})$$

$$\frac{\Gamma \vdash I_1 : A \qquad \Gamma \vdash I_2 : Insensitive}{\Gamma \vdash I_1 \rightarrow I_2}\ (\textsc{Comm Insensitive Connect})$$

Table 4.15: Payload type rules

$$\frac{\Gamma \vdash PayloadType \qquad \Gamma \vdash PayloadKind}{\Gamma \vdash PayloadType \times PayloadKind}\ (\textsc{Type Payload})$$

and *Falling* are edge sensitive. An *Insensitive* type does not define whether a wire is synchronous to the clock or which value it is triggered on. The *Insensitive* type can connect to any other *Sensitivity* as defined by rule *Comm Insensitive Connect* in Table 4.14. An *Insensitive* type may be used for automatically generated interfaces, which depend on the structure of the other interface. The same type may also be suited for describing differential signal pairs, where the clock is extracted from a data and strobe pair of signals. Traditionally, the sensitivity of a wire is labelled as part of the wire name but this technique is not normally used for verification.

Table 4.16: Payload Kind rules

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash Routable}\ (\textsc{Type Routable}) \qquad \frac{\Gamma \vdash \diamond}{\Gamma \vdash Window}\ (\textsc{Type Window}) \qquad \frac{\Gamma \vdash \diamond}{\Gamma \vdash Stream}\ (\textsc{Type Stream}) \qquad \frac{\Gamma \vdash \diamond}{\Gamma \vdash Control}\ (\textsc{Type Control})$$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash AnyPayloadKind}\ (\textsc{Type AnyPayloadKind})$$

$$\frac{\Gamma \vdash I_1 : A \qquad \Gamma \vdash I_2 : A \qquad A \notin \{AnyPayloadKind\}}{\Gamma \vdash I_1 \rightarrow I_2}\ (\textsc{Comm Specific Connect})$$

$$\frac{\Gamma \vdash I_1 : A \qquad \Gamma \vdash I_2 : AnyPayloadKind}{\Gamma \vdash I_1 \rightarrow I_2}\ (\textsc{Comm Any Connect})$$

4.2.6 Payload

The *Payload* describes the unit of data transferred over an interface, which complements the structural information captured by the *PhysicalType*. The separation from structural information allows the payload to be verified independently from the method of transfer. The *Payload* is the product of the *PayloadKind* and the *PayloadType*, as shown in Table 4.15. This payload information is not captured in the traditional PSF descriptions and is a novel extension to the format.

4.2.7 Payload Kind

The *PayloadKind* represents general groups of data units that have distinct processing characteristics. These groups are *Routable*, *Window*, *Stream*, *Control* or *AnyPayloadKind* as depicted in Table 4.16. Each group is processed in a distinct manner that is not suitable for the other data units.

A *Routable* kind refers to packets, frames or cells. Each of these data units can be routed through a system or network as they contain header information to direct the payload. Routable kinds are the most common *PayloadKind* in packet processing systems.

A *Window* kind is a unit of data with no routing information. *Window* kinds are frequently used in memory transfers and Digital Signal Processing (DSP) applications. For example, the fast Fourier transform is a DSP algorithm that operates on fixed-size windows of data and Direct Memory Access (DMA) controllers transfer blocks of data between components in a computer system. Both examples use windows as the destination of data is either predefined or specified by sideband signalling.

A *Stream* kind is a continuous set of data. This kind is frequently encountered in signal processing systems where analogue signals are sampled and converted to digital representations. Modern mobile telephone basestations are likely to include *Routable*, *Window* and *Stream* kinds as the basestation consists of signal processing and packet processing subsystems.

The *Control* kind specifies that an interface communicates control information as not all interfaces transfer data. For example, a single interrupt wire does not transfer any data. Instead, it informs the recipient that a component needs to be serviced or that an event has occurred.

Finally, the *AnyPayloadKind* is used to define interfaces that support a range of payloads. For example, passive components such as a FIFO might have interfaces with an *AnyPayloadKind*. The operation of a FIFO is not dependent on the payload and the component can be used in a variety of systems. As defined by rule *Comm Any Connect* in Table 4.16, *AnyPayloadKind* can match any other payload kind. In contrast *Routable*, *Window*, *Stream* and *Control* kinds must match an identical payload kind as shown by rule *Comm Specific Connect* in the Table 4.16.

4.2.8 Payload Type

The *PayloadType* describes a specific instance of a *PayloadKind*, which differentiates between the payloads that an interface can process. For example, Ethernet frames and

Table 4.17: PayloadType type rules

$$\frac{\Gamma \vdash \diamond \qquad PayloadType \in Payloads}{\Gamma \vdash PayloadType} \text{ (Type PayloadType)}$$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash AnyPayloadType} \text{ (Type AnyPayloadType)}$$

$$\frac{\Gamma \vdash I_1 : PayloadType \qquad \Gamma \vdash I_2 : PayloadType \qquad PayloadType \notin \{AnyPayloadType\}}{\Gamma \vdash I_1 \rightarrow I_2} \text{ (Comm Specific Connect)}$$

$$\frac{\Gamma \vdash I_1 : PayloadType \qquad \Gamma \vdash I_2 : AnyPayloadType}{\Gamma \vdash I_1 \rightarrow I_2} \text{ (Comm Any Connect)}$$

Internet Protocol packets are both *Routable* kinds. However, the formats of these protocol data units differ significantly, which means that they need to be processed differently. As with other types, the set of *PayloadTypes* is defined by the IP block library and each *PayloadType* must match as shown in Table 4.17.

The *PayloadType* may also be an *AnyPayloadType*, which can match any *PayloadType* as defined by rule *Comm Any Connect* in Table 4.17. As with the *AnyPayloadKind*, the *AnyPayloadType* can be used with generic components such as a FIFO. The *AnyPayloadType* can also be used with a specific *PayloadKind* to restrict the range of supported payloads. For example, the *AnyPayloadType* could be used with *Routable* kind, which states that an interface can work with any packet, frame or cell type. This could be used with a high-level compilation tool that defines the interface depending on the component that it is connected to.

4.3 Type Checker

A type checker has been implemented as part of the semantic analysis phase of Brace. The type checker uses the proposed type system to verify the connections within a design and verifies the design without user intervention. As a result, the connections within a design are verified during each compilation of the system.

The type checker has been implemented in Haskell and is called by Brace during the traversal of the system model before elaboration. The system model contains information from the IP library, which can be used to perform type checking of connections. As each connection is direct and independent, type checking is performed on each connection separately before resolution to nets. The type checker is invoked by executing a separate process, which returns a code stating whether type checking was successful. Brace communicates with the type checker through an intermediate file that describes the connection being verified. No other communication is required to perform type checking.

The main benefit of using the type checker in Brace is that each connection which fails type checking is identified. The identification of erroneous connections allows Brace to specify the offending line to the user and highlight which property of the type system has failed. Following a compilation with no type errors, the Click description is compiled into a PSF description and the subsequent low-level implementation flow is executed.

4.3.1 Interface Examples

The type checker uses a set of data structures to represent types internally. These data structures follow the architecture of the type system and can be understood by examining three examples. The three examples that will be examined are an interrupt wire, a data bus and an implementation of the LocalLink standard.

Interrupt

An interrupt is a single wire which indicates that an event has occurred and that action needs to be taken by the recipient of the signal. Generally an interrupt is connected to a

Figure 4.5: Graphical representation of Interrupt wire

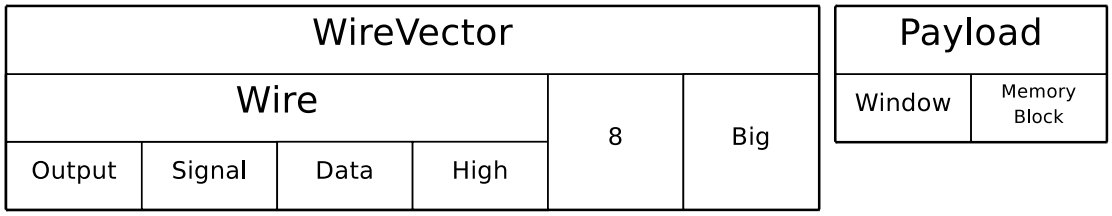

Figure 4.6: Graphical representation of Data bus

processor and requests the execution of a service routine. The data structure representing an interrupt is a record consisting of two fields for the physical type and payload information. Assuming that the interrupt signal is to be connected from an Ethernet MAC to an interrupt controller then the physical type is a wire. The wire data structure is also a record that contains an output direction, interrupt signal type and "packetreceived" net role. For the purposes of this example, the sensitivity is rising edge. An interrupt does not transfer any data so the payload field is a record of a control kind and "Ethernet interrupt" type.

The type checker uses this data structure and the relevant data structure on the opposite interface to perform type checking. Each field in the data structure is compared individually to the relevant field in the opposite interface. If the values of the fields match the opposite field as defined by the type system then the interfaces are compatible. Otherwise an error code is returned to Brace.

Data bus

The data bus is a set of wires that operate collectively to transfer data. Data buses are commonly found on the interfaces from processors to memory and are represented by a wire vector type. The data type is a tuple type composed of the combination of the endian and several identical wire types that define the size of the vector. For implementation,

the data structure is a record consisting of physical and payload fields as described in the interrupt example. The physical data type is also a record that contains an endian value and a reference to the wire data structure, which is unchanged from the interrupt example. Additionally, the size of the product can be represented directly by a natural number, which simplifies processing. Although the data structure for describing the wire component of the vector is unchanged, the values of the fields are different. For a data bus being connected to memory the values would be an output direction and level high sensitivity. Role of the bus would be *Signal* and the net role of type *Data* assuming that it is defined by the IP library. For the purpose of this example the size of the vector is eight and the endian is big.

The payload relates to the operation of the IP block, which in this case is to transfer blocks of memory. The payload kind is *Window* as the data bus transfers fixed size blocks of data. The destination of the data is not specified as part of the protocol data unit but it is specified by sideband signals. An appropriate type for the *PayloadType* can be chosen to reflect the variety of memory accesses supported by the interfaces.

LocalLink

LocalLink is a complex interface that is designed for packet transfers over a direct connection [125]. There are several variants of the standard which are incompatible with each other. For example, one extension includes virtual channels, which is incompatible with variants without this feature. Using a nominative type system these inconsistencies are not detected but they are caught by a structural type system.

The structure of the core LocalLink interface is shown in Figure 4.7. As LocalLink is a complex interface, the *PhysicalType* is *BusType* and the *BusRole* is *Initiator*. As shown in Figure 4.7, the LocalLink standard is comprised of four *Wires* and one *WireVector*. LocalLink is an unidirectional interface but bidirectional interfaces can be composed of multiple unidirectional interfaces.

The *WireVector* is DATA, which has a size of eight and is *Big* endian. As with the previous example, the *WireVector* is synchronous to the clock and it is level *High*. The *Wires* are SOF, EOF, SRC_RDY and DST_RDY which indicate the start of a packet,

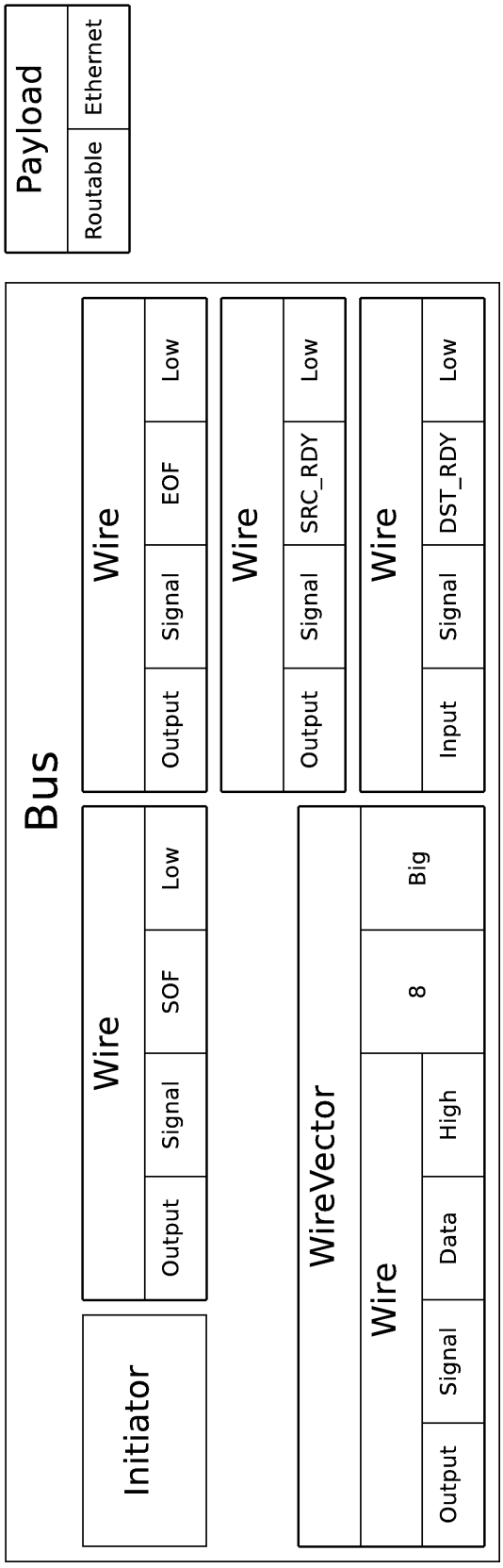

Figure 4.7: Graphical representation of a LocalLink interface

end of a packet, readiness of the initiator and readiness of the target respectively. The *Sensitivity* of these wires is level *Low*. Each *Wire* is an *Output*, except for DST_RDY, which is an *Input*.

Within the LocalLink interface, the *NetRole* prevents the connection of incorrect roles. For example, SOF cannot be connected to EOF in the opposite interface, although the Click syntax allows this to be described. As the *NetRole* differs for each wire in the interface, the type will not match unless it is connected to a wire with the equivalent *NetRole*. The *SignalRole* is *Signal* in all cases.

The *PayLoad* of a LocalLink interface is not specified by the standard but it is loosely designed for packet transfers. The *PayloadKind* is therefore *Routable* and the *PayloadType* is *AnyPayloadType*. However, an IP block implementing the LocalLink standard will use a specific *PayloadType* as the block is unlikely to support every possible payload. For example, a LocalLink interface on an Ethernet MAC will have a *PayloadType* of *Ethernet*. Generic components such as a FIFO with a LocalLink interface will be of *AnyPayloadType* as the operation of the IP block is not dependent on the payload.

4.3.2 Results

The type checker has been evaluated by application to two reference designs. The reference designs are the "Ethernet Cores Hardware Demonstration Platform" [126] and the "Ethernet-to-MFRD Traffic Groomer" [127]. The "Ethernet Cores Hardware Demonstration Platform" consists of a variety of systems that demonstrate the functionality of the Ethernet MACs available for a range of FPGA devices. For the purpose of evaluating the type checker, the Tri-mode Ethernet MAC (TEMAC) demonstration for the Xilinx ML403 board was used. The TEMAC demonstration connects a Microblaze subsystem to the TEMAC, which is connected to the external PHY off-chip. The system also includes an Ethernet statistics IP block and a packet generator. The "Ethernet-to-MFRD Traffic Groomer" is an extension of the Mesh Fabric Reference Design (MFRD), which routes packets between end points. The Traffic Groomer prioritises incoming traffic and segments received Ethernet frames into fixed sized cells. The cells are then processed by the MFRD and reassembled by the Traffic Groomer before being scheduled for transmission

Table 4.18: Lines of code and the number of connections required to describe "Ethernet Cores Hardware Demonstration Platform" in Verilog and Click

	Verilog	Click
Lines of Code	2740	246
Connections	188	73

on the outputs. Both reference designs required approximately 20 minutes to synthesise a bitstream on a 2GHz Intel Core Duo Processor with 2GB of RAM.

Each reference design has been described in Click and implemented using Brace. The appropriate IP blocks were imported into the IP library and each design was tested in hardware to ensure accuracy and correctness. For comparison, the original Verilog interconnection description of the "Ethernet Cores Hardware Demonstration Platform" contained approximately 2740 lines of code and 188 connections, as shown in Table 4.18. The equivalent system in Click contained approximately 246 lines of code and 73 connections. Of the 73 connections, 31 were used for connecting the clock and reset signals.

In order to evaluate the type system, the designs were systematically altered with incorrect connections, where each erroneous connection exercised a different aspect of the type system. Unfortunately, the frequency of each error in typical designs is not known and weights for the probability of the mistake could not be calculated. This method demonstrates the variety of errors caught but it does not necessarily represent the frequency of these errors.

After the set of errors was created, the system was synthesised to determine the point at which the implementation flow failed. The reference designs were synthesised with and without type checking in order to compare the point at which errors were detected. The Brace compiler always performs some basic checks, which could not be disabled and these are included in the comparison. For example, the compiler always checks that a wire is not connected to a bus.

The test suite consists of 31 different errors that were inserted into the reference designs. The number of errors detected and the stage at which they were detected are shown in Figure 4.8. In the absence of type checking, 58% of the errors were caught by the

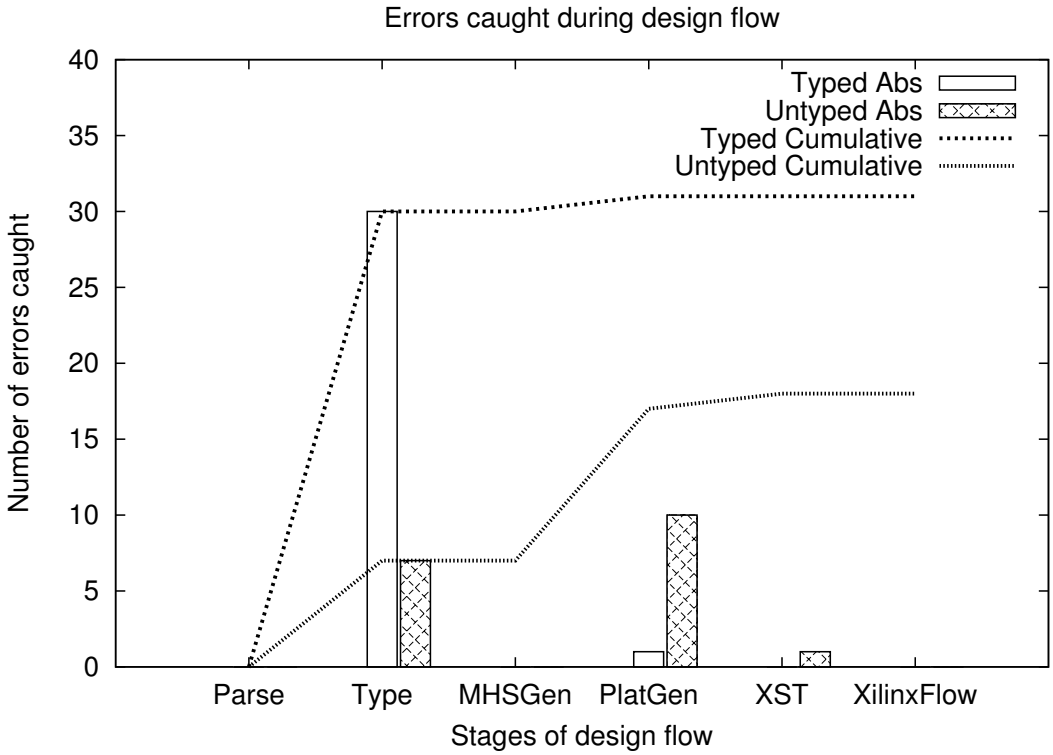

Figure 4.8: Results of evaluating the type system

implementation flow. The remaining errors were undetected but the majority of those were related to mismatched payloads. However, some physical errors were also undetected during implementation. In comparison, the type checker caught 96% of the errors during implementation.

Only 96% of errors were caught as the type checker could not detect a problem related to multiple drivers. The direct connections supported by Click prevent interpretation of the resultant net without elaborating the entire system. For example, the following code shows two components *inst1* and *inst3* that are connected to the same interface on a third component *inst2*.

```
inst1 [data] -> [data] inst2;
inst3 [data] -> [data] inst2;
```

Assuming *data* is a single wire then the previous Click statements are syntactically correct. If *data* on *inst1* and *inst3* were both outputs and *data* on *inst2* was an input then both

statements would pass type checking provided the other criteria of the type system were met. However, the resultant net would consist of two drivers, which is not permitted for FPGA-based systems. In order to detect multiple drivers, a design rule check could be added to code generation. Alternatively, the type system could be enhanced with dependent typing but care would be needed to avoid excluding correct systems. For example, if *data* on *inst1* and *inst3* were both inputs and *data* on *inst2* was an output then the intention of the connection would be to connect a single driver to multiple receivers, which is valid.

In addition to catching more errors, the type system also detects errors earlier in the design flow as shown in Figure 4.8. Without type checking, the caught errors are detected at various stages in the design flow. Errors caught by the EDK tool known as Platform Generator (PlatGen) require 30 seconds for detection. Errors detected by XST required between 5 and 10 minutes depending on the particular IP block that was erroneously connected. Designs with undetected errors execute the complete synthesis flow, which requires approximately 20 minutes to complete on a 2GHz Intel Core Duo Processor with 2GB of RAM. For undetected errors, the time taken to identify and resolve the mistake is indeterminate as it requires manual inspection and testing of the system. In contrast, type checking using the proposed type system takes approximately 10 seconds to execute and explicitly states which connection failed type checking.

Finally, the type system has also caught an error in the EDK library that was previously unknown. The description of an interface of the Microblaze soft processor was incomplete in a released version of the EDK IP library. The Microblaze processor supports connections to the CoreConnect PLB bus with separate interfaces for instruction and data fetches. However, one wire on the instruction interface of the Microblaze processor was not included for connections to the PLB. The EDK allowed this to pass synthesis as it did not perform any structural checks. The type system on the other hand identified the error immediately. The wire concerned did not prevent the operation of the bus but this may also explain why the error was not detected earlier.

4.4 Summary

High-level design environments contain information that can be exploited for verification. Type systems provide one method of verifying designs to prove the absence of a class of errors. This chapter has presented a type system for verifying the interconnections of IP blocks. A type checker has been implemented as part of the semantic analysis phase of Brace and has been evaluated using two reference designs. The type system detects errors that are not found by the existing design flow and also detects errors earlier than the traditional design flow. These features reduce the number of synthesis iterations and save time debugging designs.

Chapter 5

System-level Transaction Monitoring

Using a type system to statically verify the connections between IP blocks guarantees that those connections are valid but the type system does not verify the dynamic execution of the intended system. Dynamic validation allows the designer to observe the operation of the final implementation, which can highlight errors that are undetected by other validation and verification techniques.

This chapter presents the architecture of a system-level transaction monitoring system and a methodology that can be used to debug errors by using the monitoring tool. The aim of this system is to provide observations of distributed high-level events, which occur throughout the design under test. The monitoring tool operates passively while maintaining the abstractions of the high-level design environment. The monitoring system consists of probes, a collection module and software for executing on an external host computer.

This chapter also explores the implementation of the components forming the system-level transaction monitoring system. The architecture of the probes are explored and the trade-off between functionality and resource utilisation will be discussed. Two collector architectures are presented, which provide event capture and profiling capabilities. Finally, the operation of the host software will be explained.

5.1 Debugging Methodology

When creating tools for debugging systems, a methodology describing the techniques used to employ the debugging tool needs to be specified. Based on the proposals of Araki et al. [116] and Josephson [117], a validation methodology for FPGA-based packet processing systems can be composed as shown in Figure 5.1. The methodology consists of three key stages which are testing, isolation and rectification. Testing is similar to the characterisation phase proposed by Josephson, which identifies defects in the system. The term testing reflects the nature of the activity better than characterisation as the functionality of the system is being exercised at a high-level. Testing is designed to detect faults and when a fault is found the isolation phase can begin. Isolation determines the location of a fault and specifies the component that has caused the error. This phase might follow the strategy proposed by Araki, where a set of hypotheses are constructed and tested until the correct hypothesis is found. Upon isolation, the error must be corrected which requires the cause of the fault to be determined and subsequently rectified.

Relating the methodology to the proposal for a system-level monitoring tool, the monitoring tool should be suitable for application in the first two phases of the methodology. To be useful, it must be able to detect errors during testing. It must also be designed to support isolation of errors as it is required to observe the interaction between multiple components in the system. Finally, a system-level monitoring tool is not required to correct defects as low-level data monitoring tools might be more suitable in these circumstances. However, the ability to correct problems would be advantageous even for a subset of the errors that can be detected.

5.2 Monitoring System Architecture

System-level transaction monitoring provides several benefits over low-level monitoring tools. The ability to abstract low-level details and observe events distributed throughout the system provides a monitoring solution suited to high-level design environments. In this section, the architecture of such a monitoring system and the associated external host software is presented.

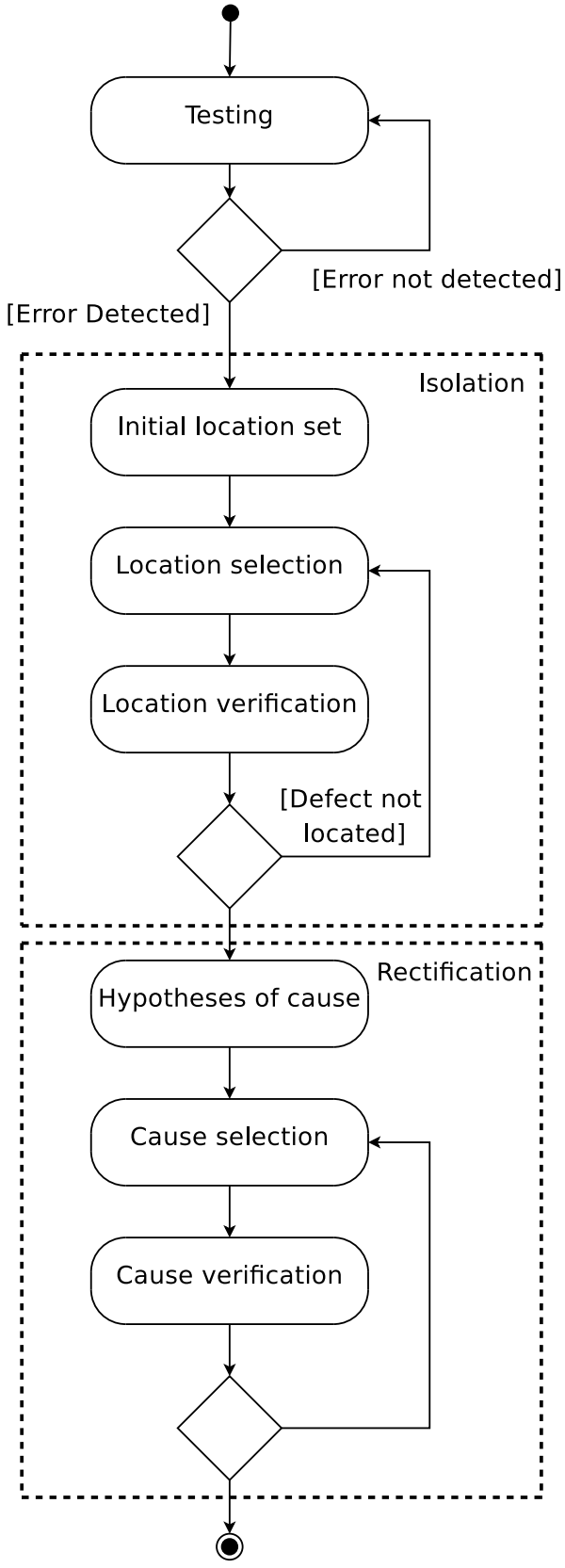

Figure 5.1: Proposed debugging methodology using system-level monitoring

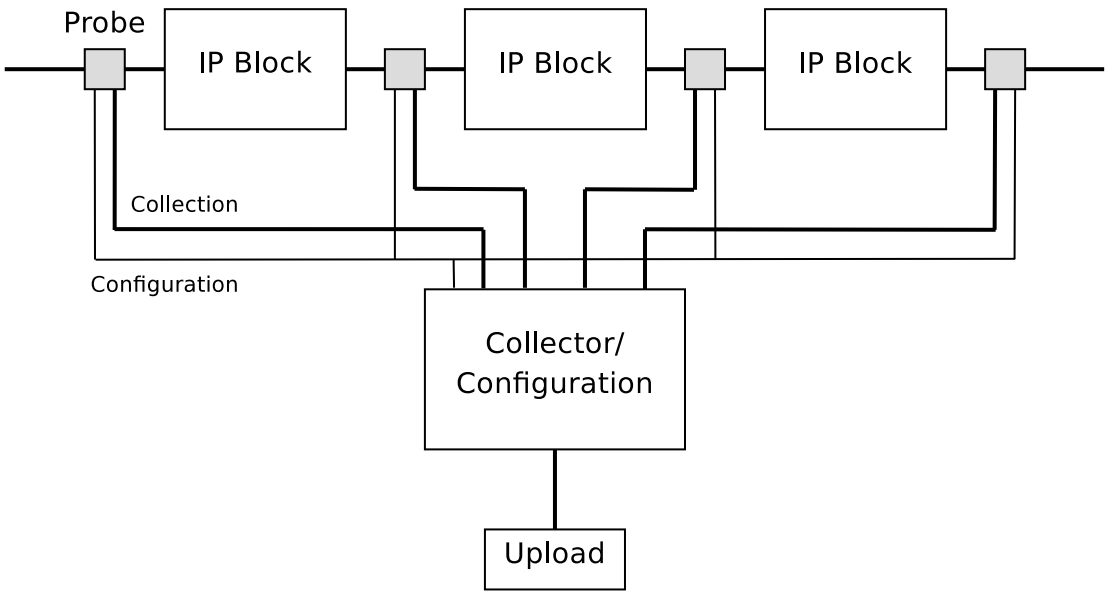

Figure 5.2: Architecture of the system-level monitoring system.

As illustrated in Figure 5.2, the architecture of the monitoring system is comprised of a set of probes, configuration circuitry, a module for collecting and communicating results and an external software application. The probes are distributed throughout the system and observe the interfaces of system components. The probes also communicate their results to the collection module. The configuration circuitry is responsible for configuring the probes at run-time under the direction of the external software application. The collection module receives and processes the results from the probes. It also transmits the processed data from the FPGA to the external software application. The external software application is responsible for recording and interpreting the data transmitted by the collection circuitry. It is also responsible for the presentation of the results to the user.

The probes have two responsibilities. First, each probe observes and records the transactions over a specific interface type. Second, the probe communicates its results to the collection module.

The probe observes transactions by interpreting sequences of low-level signal transitions as high-level events. This requires the probes to be connected directly to the interface being observed. The probes are passive as they do not interfere with the low-level signalling during system execution. However, the interface signal timing may be affected by the

insertion of a probe. The resource requirements of the probe determine the extent to which timing is affected. Therefore, in order to reduce the impact of signal timing, the resource penalty of the probe should be minimised.

The probe communicates its results to the collector module through a direct level-sensitive signal. This method abstracts all transactions to a common representation, which permits multiple probes to be used regardless of the interface type. The location and type of the probe are known prior to synthesis, which means that only the occurrence of a transaction needs to be communicated. The collector module adds location information to events as they are uploaded to the external software application. As the monitors may be located on multiple clock domains, the level-sensitive signal may also need to cross clock domains to be recorded correctly by the collector module.

The collector module receives data from the probes and is responsible for communicating the information to the external host software. The collector module also processes the received data to maintain the spatial and temporal information of each transaction. As the collector is recording high-level information, it requires a lower bandwidth compared to low-level monitoring tools and requires less on-chip buffering.

The probes are connected to the collector by either a single level-sensitive signal or by two level-sensitive signals. For transactions of a fixed duration only a single signal is required. As the duration of the transaction is known a priori, only the time of the completion of the transaction is recorded. For example, a fixed-sized protocol data unit transferred over an interface that forbids stalled clock cycles has a predetermined duration, which makes recording the start time redundant. Many memory interfaces perform operations in a single clock cycle, which also makes recording a start time unnecessary. Again, an interrupt indicated by a rising edge signal transition does not need to record the duration of the event as it is already known. The completion time of the transaction is recorded to allow the probe to filter events of interest.

Where the duration of a transaction is not known a priori, the start and end times of a transaction need to be recorded. In this case two level-sensitive signals are required to connect a probe to the collector. For profiling operations a single signal is sufficient as the duration of the events is lost and only the number of events occurring within a quantum

of time is recorded. For event capture operations, the duration of an event is important. For example, variable-sized protocol data units, such as IP packets, require the start and end time to be recorded. Within a pipelined packet processing system, this allows the user to observe the stages of the pipeline that the packet occupies at any given instant. Without both items of information, it is impossible to deduce which stages are occupied.

The external software application is responsible for receiving the processed data and presenting the results to the user. In order to present the results to the user, the software application needs to be supplied with configuration data. The configuration data is composed of two key elements. First, the location data provided by the collector module needs to be mapped to the location information understood by the user. Second, the software needs to know how to configure the monitoring system to provide meaningful observations.

This monitoring architecture provides advantages over existing on-chip monitoring tools. First, the collected samples are not held in separate probes. This allows all events to be related both temporally and spatially. Second, the probes do not hold data internally which reduces resource requirements and alleviates the effects of component displacement. The size of a probe determines the extent to which an IP block is displaced from its original location. Third, the monitoring architecture allows the location of errors to be isolated quickly. The location of an error can be identified to the associated interface of an IP block.

The probes are designed to be lightweight to minimise the effect on resource utilisation. FPGA designs tend to use the majority of resources on-chip as the smallest chip will generally be chosen to minimise costs. The system designer would most likely use a smaller and cheaper device if the FPGA was insufficiently utilised. Each probe requires the use of Complex Logic Blocks (CLBs) and the interconnections between them. The probes also require a connection to the central collector which will demand the use of routing resources. If the probe and collector are at opposite ends of the chip then a significant proportion of the routing resources might be required. Additionally, as the chip tends to be well utilised it is possible that adding too many probes will make the design too big for the target FPGA device. Clocking resources also tend to be sensitive to routing which might affect the timing delay. Inserting additional logic may cause problems with

timing closure as components may be displaced. Furthermore, if the probes operate at a different clock frequency compared to the component being monitored then the routing of clock signals might be adversely affected. However, the extent of this would depend on the clock resources available in a particular device.

5.3 Probe Architecture

The probes are designed to observe the interfaces of system components and record the transactions detected over those interfaces. The probe detects transactions by following the sequence of signal transitions and inferring high-level events from those transitions. As the probe is designed to observe the constituent signals of an interface, it operates passively while it is connected directly to the interface. A direct connection requires the resource requirements to be minimised in order to reduce the impact on signal timing.

Signal timing is directly related to the impedance of a wire, which is directly related to its length. The length can be minimised by placing probes close to the interfaces being observed, which will reduce the capacitive load on the wire and consequently the propagation delay. As the probe is attached to an existing connection, the monitored signal will be fanned out to an additional point. Increasing fan-out will increase the impedance of a wire but the increased impedance can be compensated by register duplication. Component displacement is also directly affected by the size of the probe, which also affects the length of the wires. A larger probe will require more resources to be placed close to the interface being monitored, which will require other components to be placed further away. The placement of these components will have a subsequent effect on the placement of their adjacent system components. These effects are compounded by the nature of system-level monitoring, which requires multiple probes to monitor multiple interfaces.

The probes communicate with the collector module to indicate when a transaction has been detected. A single direct level-sensitive signal or a pair of direct level-sensitive signals are used to indicate the occurrence of a transaction. No other information is required as the location and type of the interface being monitored is known before synthesis. In order to understand the operation of the system, it is not sufficient to record every transaction that is detected. Properties of the transaction need to be captured in order to provide a

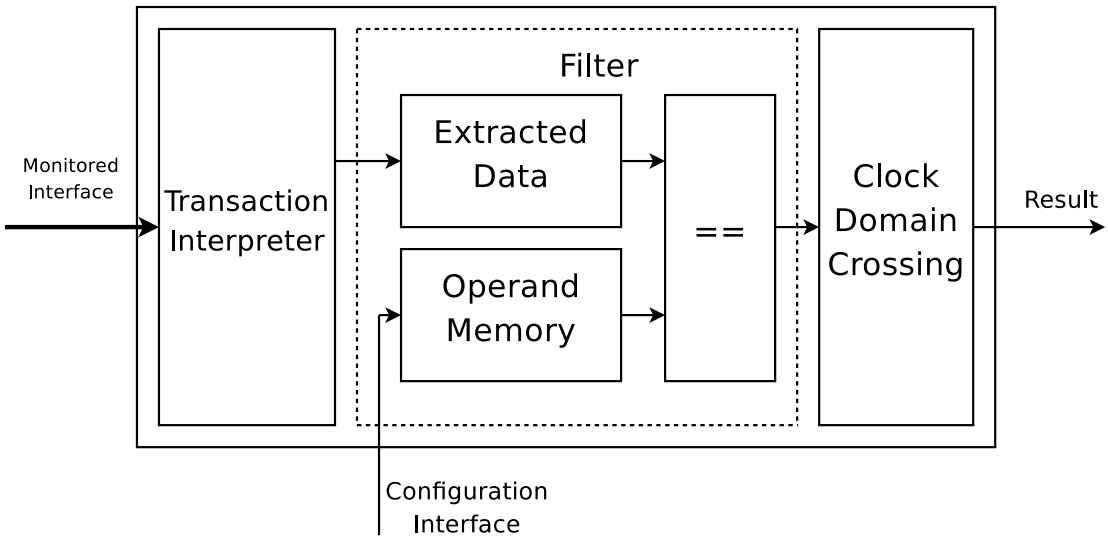

Figure 5.3: Standardised probe architecture.

context for the event. Filtering can be used to only record the events of interest, which will allow the path of events to be identified. Filtering also negates the need to upload the transaction properties as they are already known to the external software and the user. The causal relationship of events can be presented by filtering for the same properties at different points in the system. Finally, most systems require multiple clock domains in order to operate correctly, which means that the probes must be capable of transferring their results over multiple clock domains.

Although the probes are specialised to observe specific interface types, the architecture has been standardised to facilitate rapid development of new monitors. Each probe consists of three main components as shown in Figure 5.3. These components are transaction interpretation, filtering and clock domain crossing. The transaction interpreter provides a common set of control signals regardless of the interface type being monitored. Interpreters have been implemented for a variety of interface types including GMII, LocalLink, on-chip memory and PLB. The resource requirement for transaction interpretation is small but, as stated previously, interpreting transactions is insufficient to understand the operation of the system. In order to observe events of interest, a filtering mechanism must be applied. Filtering allows the functionality of IP blocks to be tested and validated according to a set of properties. Filtering can be applied to the payload of a transaction or to sideband signals such as the address bus in shared media. For example, the headers

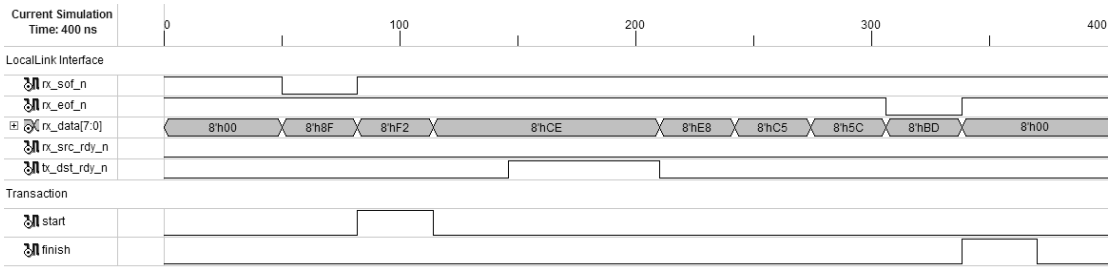

Figure 5.4: An example transaction over a LocalLink interface, which transmits a block of data.

of packets can be filtered to match packet processing operations, which provides a simple form of packet classification. Filtering reduces the debug information to events of interest and reduces the bandwidth required to communicate the results to the host software.

5.3.1 Transaction Interpretation

The transaction interpreter provides a common set of control signals, which are required by the filtering circuitry. The interpreter passively observes the interface to which it is connected and follows the sequence of signal transitions, which identifies events of interest. The transaction interpreters can be implemented as a fixed parser or a run-time configurable parser. A fixed parser is less resource intensive as the circuitry can be optimised for the specific sequence of states. Conversely, the run-time configurable parser requires resources to hold the set of states and the function to compute the following state. For systems designed in a high-level design environment, the transactions presented on an interface are known a priori so run-time configurability is not necessary. Run-time configurable parsers are typically used by low-level monitoring tools where the signal transition sequences are not known a priori and the user is required to specify the interpretation of the low-level signalling.

The interface of an IP block can be represented as a set of wires, which function collectively to perform operations. The signal values of the wires are discrete and are defined to be either 1 or 0. The sequence of transitions of the signal values can be defined as the state of the interface. The data wires are not required to interpret a transaction but are used for filtering purposes. A transaction is therefore a sequence of state transitions,

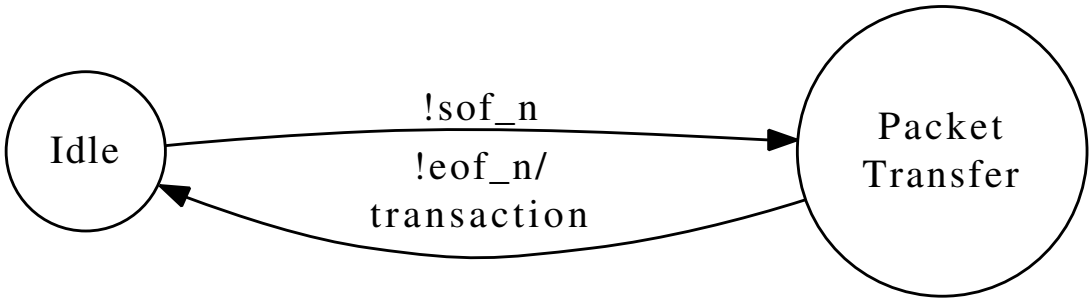

Figure 5.5: The state transitions required to interpret a transaction on a LocalLink interface

which represent an operation. The transaction interpreter is responsible for observing and following the sequence of signal transitions, identifying the start and end of transactions and identifying cycles that contain valid data. As each interface type has a unique set of transition sequences, the transaction interpreter must be customised for the individual interface types. The control signals representing valid cycles are then passed to the filtering circuitry.

For example, LocalLink and PLB are two different types of interconnect, which have different interface compositions. LocalLink is a Xilinx standard for packet interfaces, which contains a number of optional extensions. The core set of wires are *data*, *sof_n*, *eof_n*, *src_rdy_n* and *dst_rdy_n*. As shown in Figure 5.4, the start and end of a frame are indicated by a low value on *sof_n* and *eof_n* respectively. Packets are transferred in the period between these transitions with no limit specified on the amount of data transferred. Consequently, LocalLink allows the sender and receiver to stall transmission in the event that either is unable to complete the request in a given clock cycle. The sender and receiver initiate stalls by raising the *src_rdy_n* and *dst_rdy_n* signals respectively and may assert them indefinitely. The LocalLink signal transition sequence can be interpreted by a deterministic finite automata implemented as a finite state machine as shown in Figure 5.5.

Figure 5.6 shows an implementation of a trigger for observing LocalLink transactions, which uses the *sof_n* and *eof_n* signals to determine the start and end of a transaction repsectively. When the start of a frame is interpreted the circuit generates a signal, *assert start*, to indicate the event to subsequent circuitry in the probe. A separate signal, *assert*

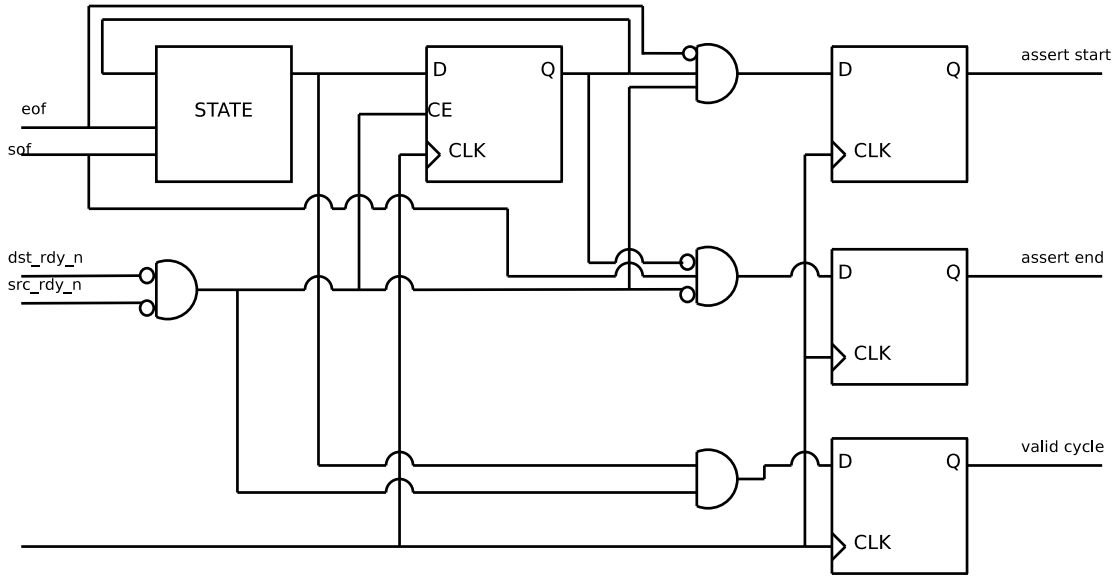

Figure 5.6: Example implementation of LocalLink trigger.

end, is asserted when the end of a transaction is observed. The remaining signal generated by the trigger is *valid cycle*, which instructs subsequent components as to which cycles are valid to match on for filtering purposes. In this example, the *valid cycle* signal is dependent on the values *src_rdy_n* and *dst_rdy_n* as these signals indicate stalled cycles on the LocalLink interface. The signals generated by the trigger are shared between all probes regardless of the interface type that they are connected to.

The CoreConnect PLB bus is an IBM standard for connecting peripherals to a processor. PLB is a fairly complex standard, which permits multiple simultaneous operations through separate read and write buses. A PLB master is able to initiate read and write requests, whereas a slave can only respond to requests. The core signals for FPGA implementations of PLB master interfaces are *M_Abus*, *PLB_MRdDBus*, *PLB_MWrDBus*, *M_request*, *M_RNW*, *PLB_MaddrAck*, *PLB_MrdDAck*, *PLB_MwrDAck* and *PLB_MTimeOut*. Furthermore, there are signals available for burst transfers but these are omitted from this discussion as they do not aid in conveying the basic concepts. A read request is initiated by asserting *M_request*, asserting the address on *M_Abus* and simultaneously driving *M_RNW* high. Alternatively, a write request is initiated by driving *M_RNW* low. In both cases, the request is acknowledged through the signal *PLB_MaddrAck*. The *PLB_MTimeOut* signal is used to abort a request if the slave has not responded in the correct number of clock cycles, which is parameterisable. The slave then replies using the appropriate data

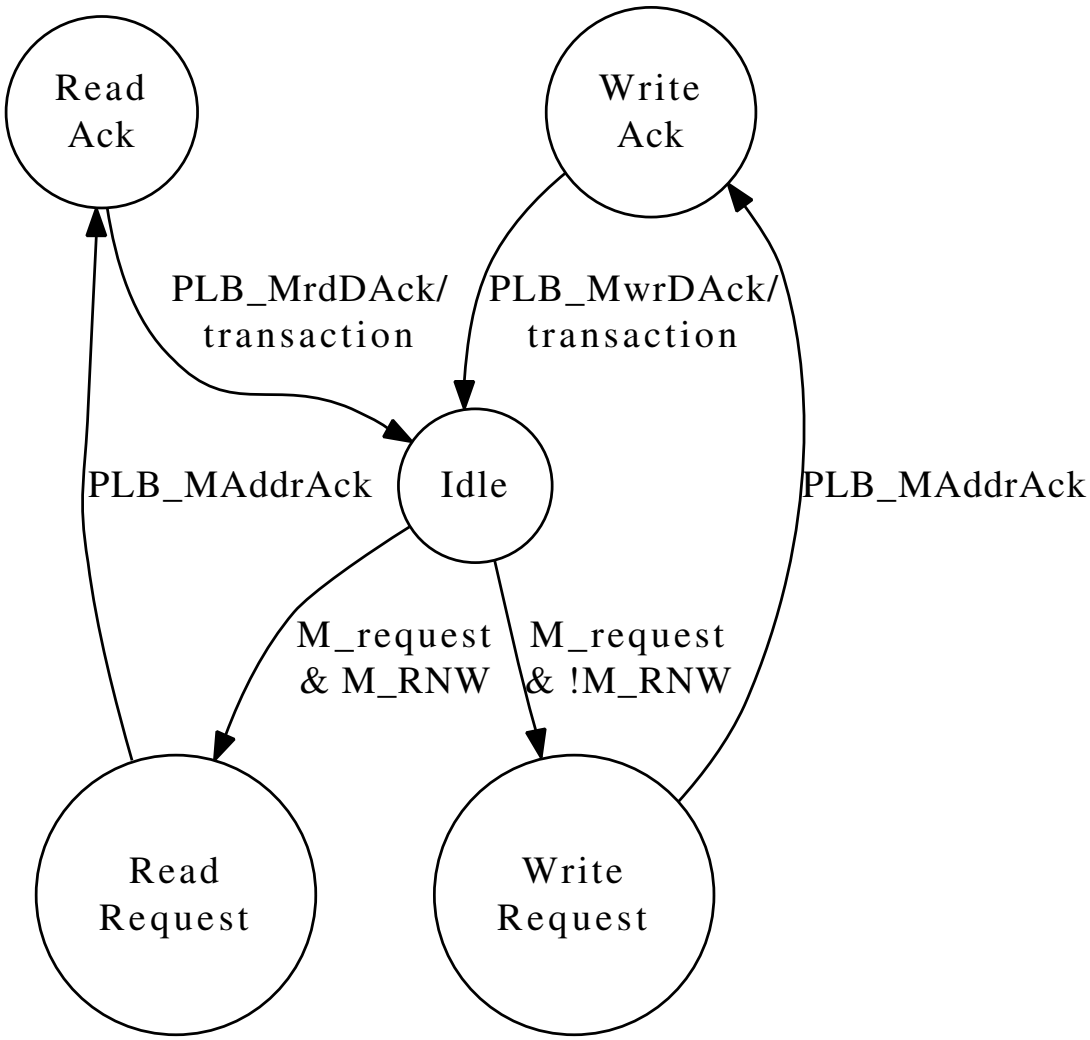

Figure 5.7: The state transitions required to interpret a transaction on a PLB bus master interface.

bus for the request and acknowledges using the appropriate signal. A write request is fulfilled by the slave writing data to *PLB_MwrDBus* and asserting *PLB_MwrDAck*. A read request is fulfilled using *PLB_MrdDBus* and asserting *PLB_MrdDAck*. The separation of read and write data buses allows a read and write operation between different components to occur simultaneously over the bus.

The PLB signal transitions can be interpreted using a finite automata as shown in Figure 5.7. The PLB monitor has two separate output signals as it has separate read and write buses. In order to distinguish between transactions on either bus, separate output signals are used. If multiple transaction types exist on an interface then a different representation

Table 5.1: Transaction interpretation resource requirements with single level-sensitive output

Interface	LUTs	Flip Flops	Slices
LocalLink	3	6	5
PLB	5	7	6
GMII	7	7	8
MII	4	6	6
Interrupt	2	5	5

may be more suitable to minimise resource utilisation.

The transaction interpreters are designed to only record complete transactions. Incorrect transition sequences or aborted transactions are ignored. Missing transactions in the monitoring report can point to problems in the signalling sequences, which can then be further analysed through conventional low-level tools. However, it may be possible to indicate an error while maintaining resource efficiency as the state machine could be extended to produce an additional error signal.

As shown in Table 5.1, the resource requirements for interpreting transactions with a single output signal is small. The direct media, LocalLink, MII and GMII have small resource footprints. Although PLB is a complex standard, the resource requirements are minimal as each end-point is monitored separately, which reduces complexity and provides information related to the interface of a specific IP block.

The transaction interpreters are also designed to support subtypes of an interface through conditional instantiation of signalling components. These signalling components are defined before synthesis as they need to be specified before the system can be instantiated. For example, LocalLink supports various widths of the data bus and allows conditional instantiation of optional wires, which might alter the functionality of the interface. The probes are parameterisable to allow the debugging system to match these variations.

Alternatively, a run-time configurable parser could be implemented to permit flexibility in monitoring transactions. This parser would resemble the triggering mechanism of a low-level tool such as ChipScope, which requires approximately 270 slices, 458 LUTs and

532 FFs for monitoring a GMII interface. The ChipScope parser provides 16 match units and a flexible sequencer for ordering the match units temporally. ChipScope can support sequences of up to 16 matches. As demonstrated, the resource utilisation is greater for a flexible parser and is unnecessary as the operations of the interface and the connection are known a priori.

5.3.2 Filter

The filter is responsible for determining whether a transaction should be recorded. The filtering capabilities of the probes can take one of two forms. The first form filters payloads transferred over an interface and the second form filters addresses through sideband signals.

Payload filtering is supported by matching the data presented on the data bus against a preconfigured register or series of registers. This is a lightweight form of packet classification, which matches packets against a single rule. The transaction interpreter is responsible for indicating when a comparison should be executed. The comparison itself may be a simple exact comparison, a comparison with support for bit-level masking, or the comparison of any expression given in disjunctive normal form.

Address filtering requires the use of a parallel comparator as only a single cycle is available for matching. The parallel comparator uses the spatial nature of the FPGA fabric to implement matching and may implement exact or maskable comparisons. In this instance the destination of a transaction becomes more important than the data being transferred. Address filtering may also be used to monitor software execution within a system. The addresses of software functions can be obtained from most compilers and then used to configure probes that monitor memory or system buses. This technique can also be applied to the program counter of a soft processor, if this signal is available. Filtering transactions according to function addresses allows the designer to observe hardware communications in conjunction with software operations.

There are five variants of the payload filter architecture that will be explored. The first variant provides a point of reference for comparing the other architectures and is a specialised filter for IP packets. The specialised filter extracts the 5-tuple from an IP

packet encapsulated in an Ethernet frame, which is a well-known construct and widely used in packet processing applications. Both parallel and sequential matching circuits will be considered. Following this, a generic filter, which can match up to 512 bits, will be discussed. The generic filter is more flexible as it can filter any packet type, permitting its use in other packet-based media. An IP packet encapsulated in an Ethernet frame requires 304 bits to be transferred before the 5-tuple can be extracted so this should provide a representative comparison of resource utilisation.

Following this, a more flexible version of the generic filter will be presented. This filter exploits locality of interest to allow a fragmented 512-bit match, which provides the ability to observe deeper into the packet. Finally, a scalable architecture will be investigated, which can adapt to the size of the sample being matched and provides more advanced matching capabilities than the other architectures.

The architecture of an address-based filter will also be presented. This filter matches addresses and identifies the source and destination of transactions. Each type of probe has been implemented using a Virtex 4 FX 20 on an ML405 board.

Specialised Filter for IP Packets

As packet processing often targets IP packets, a filter designed to match the 5-tuple will provide a standard which can be compared to other architectures. This will allow a comparison between resource utilisation and functionality of the filter.

The fixed 5-tuple filter has been written specifically to parse a TCP/IP packet encapsulated in an Ethernet frame. The parser implementation varies depending on the width of the datapath as the constituent members of the 5-tuple will be presented on the interface on different clock cycles. Figure 5.8 shows the format of a TCP/IP encapsulated in an Ethernet frame as observed on a 32-bit LocalLink bus. The filter checks the value of the Ethernet type field to ensure that it contains an IPv4 packet and also checks the version number in the IP header. Following this, the filter captures the 5-tuple, which consists of the next protocol value, source address, destination address, source port and destination port, as highlighted in Figure 5.8. The captured values are then passed to the match unit. If the captured data matches the desired values then the transaction is recorded,

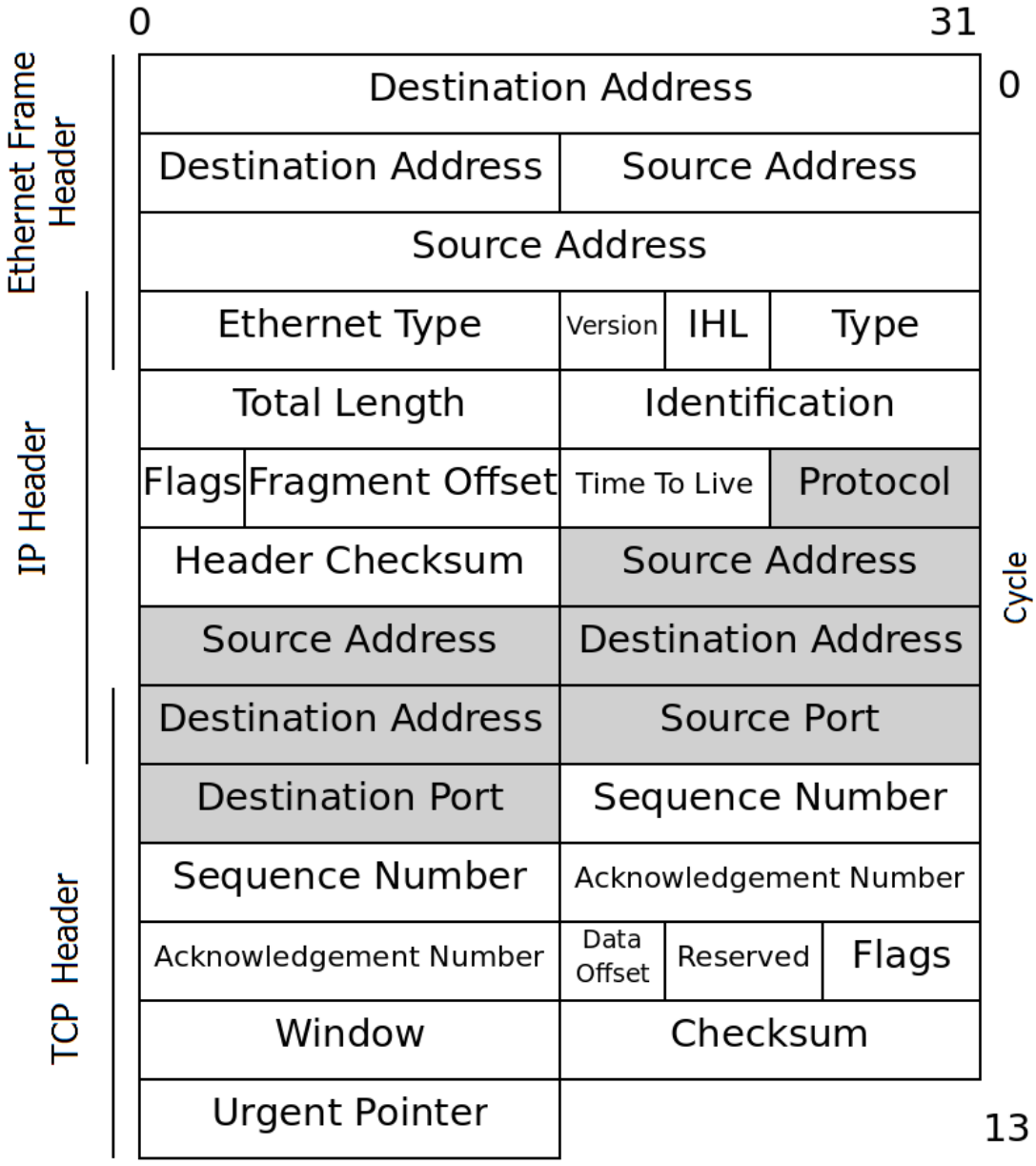

Figure 5.8: Format of TCP/IP packet encapsulated in an Ethernet frame highlighting the 5-tuple as seen on a 32-bit LocalLink interface.

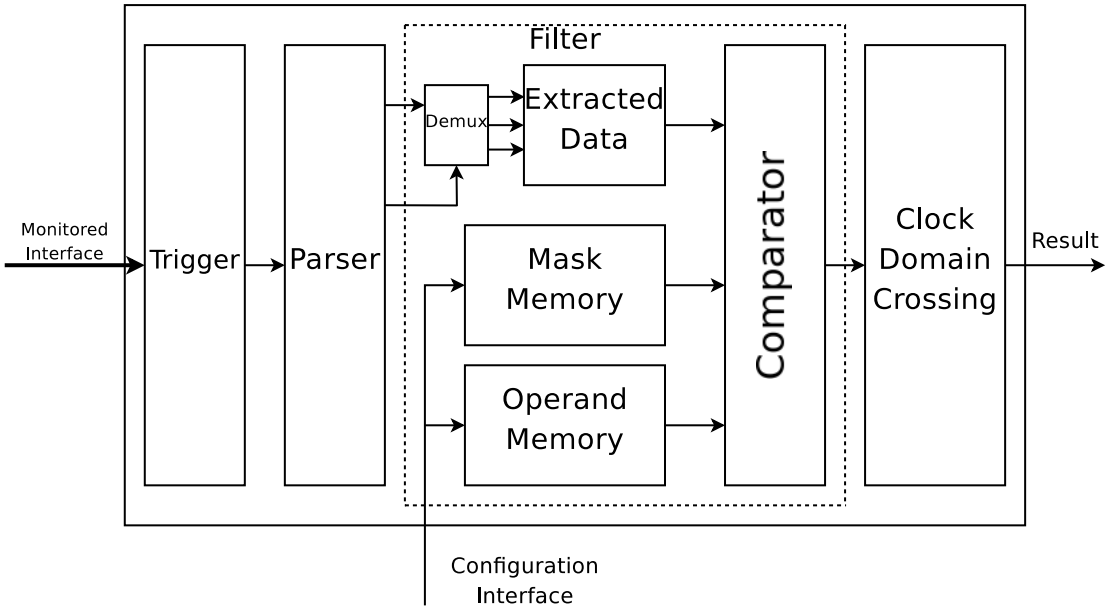

Figure 5.9: High-level architecture of the fixed parser parallel matching probe.

Table 5.2: The resource utilisation of the 5-tuple specialised parser with parallel matching.

Datapath Width	Slices	LUTs	FFs	Period (ns)
8-bit	532	627	154	6.019
16-bit	517	600	135	6.019
32-bit	509	589	119	6.019

otherwise it is discarded.

The match unit applies a mask to the captured data and the operand being compared. This provides the flexibility to monitor simple ranges of addresses and port numbers. The match unit is limited to one mask but the designer can easily add another probe if more masks are required. The fixed filter can match values either in parallel or sequentially.

Parallel Match

Parallel matching uses the spatial nature of the FPGA fabric to create a 104-bit register across multiple slices, which provides the ability to determine the result of the comparison in a single cycle. If multiple clock cycles are required to obtain the 5-tuple then the filter populates the register as it encounters the values in the packet, as shown in Figure 5.9

Table 5.2 shows that as the datapaths are widened the filter uses fewer logic resources. This is a result of the reduced number of clock cycles to obtain the 5-tuple, which also reduces the logic required to implement multiplexing between subsections of the target register. This trend is limited by the spatial nature of parallel matching, which uses three 104-bit registers and a 104-bit comparator. However, this approach has the advantage of abstracting the data-path from the rest of the monitor. Parallel matching is also mandatory for memory interfaces where transactions execute in a single clock cycle. If a match is required on more than one string then the match units need to be replicated.

An interesting property of the parallel match unit, as shown in Table 5.2, is the invariability of the period. The critical path of the probe is in the comparator as it matches 104-bits using a ripple effect. The width of the data bus does not alter the timing of the probe but it does affect resource utilisation. As the data bus width increases, the resource utilisation decreases, which is due to fewer multiplexers being required to place data in the 104-bit registers. These properties are useful for monitoring packet processing systems operating at speeds greater than 10 Gb/s. At 10 Gb/s, the bus width of interfaces on IP blocks might be increased to 512 bits wide to allow the system to operate at lower clock frequencies and to relax the constraints on component placement. Parallel matching is suitable in these instances as it can cope with larger buses without increasing the size of the comparators.

Sequential Match

As packets are transferred over multiple clock cycles, it is possible to reduce the size of the comparator for smaller bus widths. Reducing the size is achieved by matching the 5-tuple as it is presented by the filter over multiple clock cycles. The match unit is then able to exploit the architecture of the FPGA to reduce the spatial area of the registers for the mask and the desired value. As each 4-input LUT contains 16 registers, the registers can be addressed for matching 16 separate clock cycles. Look-up tables can also be combined to form distributed memory, which can be used to extend matching beyond 16 cycles. This method allows the comparator to maintain the same width as the data bus, which minimises resource requirements for smaller bus widths. This method requires the use of a state machine to co-ordinate the matching process.

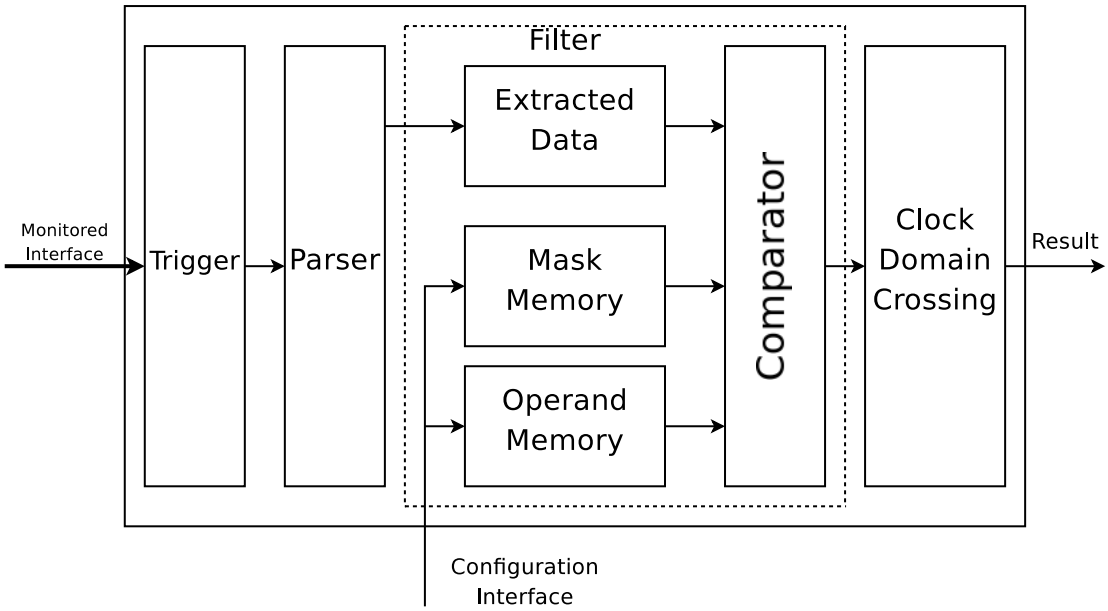

Figure 5.10: High-level architecture of the fixed parser sequential matching probe.

Table 5.3: The resource utilisation of the 5-tuple specialised parser with sequential matching.

Datapath Width	Slices	LUTs	FFs	Period (ns)
8-bit	85	129	67	4.114
16-bit	111	151	45	4.951
32-bit	192	246	60	5.307

As Table 5.3 shows, the resource requirement of the sequential match unit is significantly smaller than the proposed parallel match unit. The table also shows that more resources are required as the datapath increases. This is due to the increasing size of the comparator and the spatial layout of the registers. As the datapath continues to increase, it is expected that the resource requirements will tend towards those of the parallel match unit as more computations execute in parallel. Again, the maximum clock frequency is reduced as the datapath is widened, which is due to the ripple effect of the comparator increasing.

The members of the 5-tuple might not be aligned to the word size of the interface datapath. As the fields are presented on the interface they will appear at various offsets within the current word depending on the width of the datapath. In order to keep the sequential match unit small, the configuration software can be given responsibility for choosing which bits are relevant on a given clock cycle. This removes the need for the designer to ensure that bits external to the 5-tuple are not included. It is possible to include this mechanism in hardware but this would increase the resource requirements.

512-bit Generic Filter

Although IP packets are common, packet processing is applicable in a wider context. The previous method described a fixed parser, which can only parse a single packet type. While this is an effective solution for filtering IP packets, it is restrictive and requires re-synthesis in order to monitor different packet headers. It is also possible that an interface may transfer more than one packet type, such as data packets and internal system control packets. Thus, a more flexible solution might be required to monitor packets at run-time.

One solution is to increase the amount of memory available to the filter and perform a match on each valid clock cycle. This would move the parsing function into the filter, as demonstrated by Figure 5.11. Increasing the memory would allow any packet header up to the size of the allocated memory to be compared. Only sequential matching is considered for this architecture because parallel matching would require a 512-bit register to be placed spatially. It would also require a 512-bit comparator, which might not meet timing constraints.

In this example, the first 512-bits of an IPv4 packet encapsulated in an Ethernet frame are

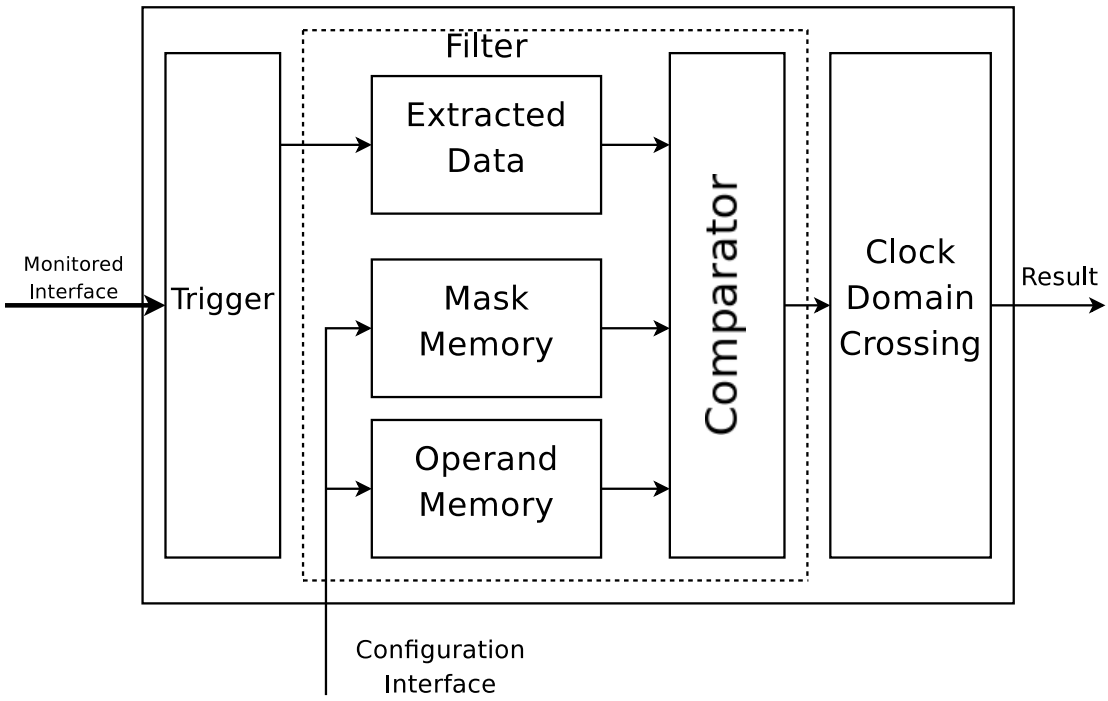

Figure 5.11: High-level architecture of the generic sequential matching probe.

compared. This would allow the probe to classify a datagram according to the Ethernet header, the complete IPv4 header and the source and destination ports of UDP datagrams or TCP packets. This architecture provides greater flexibility compared to the fixed parser as the range of values that can be compared increases in addition to the ability to capture different packet types.

In order to reduce the complexity of configuring a sequential probe, it is possible to arrange the memory addresses such that the configuration is independent of the datapath width. Thus, a single configuration can be used to configure an array of probes with different datapath widths. This is achieved by allowing the memory layout to be decided at synthesis time, when the datapath widths are chosen. Once the datapath width has been decided, the memory addressing scheme redirects the location of values to reflect the clock cycle on which they appear on the interface. The datapath width is then abstracted from the configuration software and the designer. The memory configuration is specified for each permissible datapath width and is then synthesised with the rest of the system.

As 512 bits need to be stored, they can be placed within 32 4-input LUTs. By arranging the addresses in the configuration memory appropriately, the datapath of the IP block's

Table 5.4: The resource utilisation of the generic parser with sequential matching.

Datapath Width	Slices	LUTs	FFs	Period (ns)
8-bit	87	172	29	4.816
16-bit	96	188	37	5.608
32-bit	182	226	57	5.307

interface can be abstracted. For example, an 8-bit interface will require 64 cycles to transfer 512 bits. In this case, 64 cycles can be stored in 4 sequential LUTs. For an 8-bit interface, a total of 32 LUTs are required. For a 32-bit interface, 16 cycles are required to transfer 512 bits utilising a total of 32 LUTs. Finally, for a serial interface with a single data bit, 32 LUTs will be required to match 512 bits. Thus, for up to a 32-bit datapath width, the total memory requirements do not change, although the sequential and spatial layout has been altered appropriately.

Beyond 32-bit datapaths, the memory requirements increase because the LUTs are fixed in size. For a 33-bit datapath, 33 4-input LUTs are required which gives a total of 528 bits for matching. A 64-bit datapath requires 64 LUTs, which provides 1024 bits. The additional memory in the LUTs is not used in the current implementation as a common configuration mechanism is desired. However, it may be appropriate to increase the depth of matching but this will alter the memory map for configuring the probes. The simplest solution is to leave the memory unused to provide a common method of configuring the system regardless of probe type.

For 6-input LUTs found on some FPGA architectures, the first 512 bits can be matched with 16 LUTs. Again, the memory requirement is constant for datapath sizes of 16-bits or less. For datapaths greater than 16 bits, there will be unused memory. The memory requirement can be given by expression 5.1, where α is the number of bits available for matching, β is the width of the interface's datapath and δ is the number of input bits to the LUT in the FPGA architecture.

$$\alpha = \beta.2^{\delta} \tag{5.1}$$

Again, as the size of the datapath increases, the size of the comparator must increase.

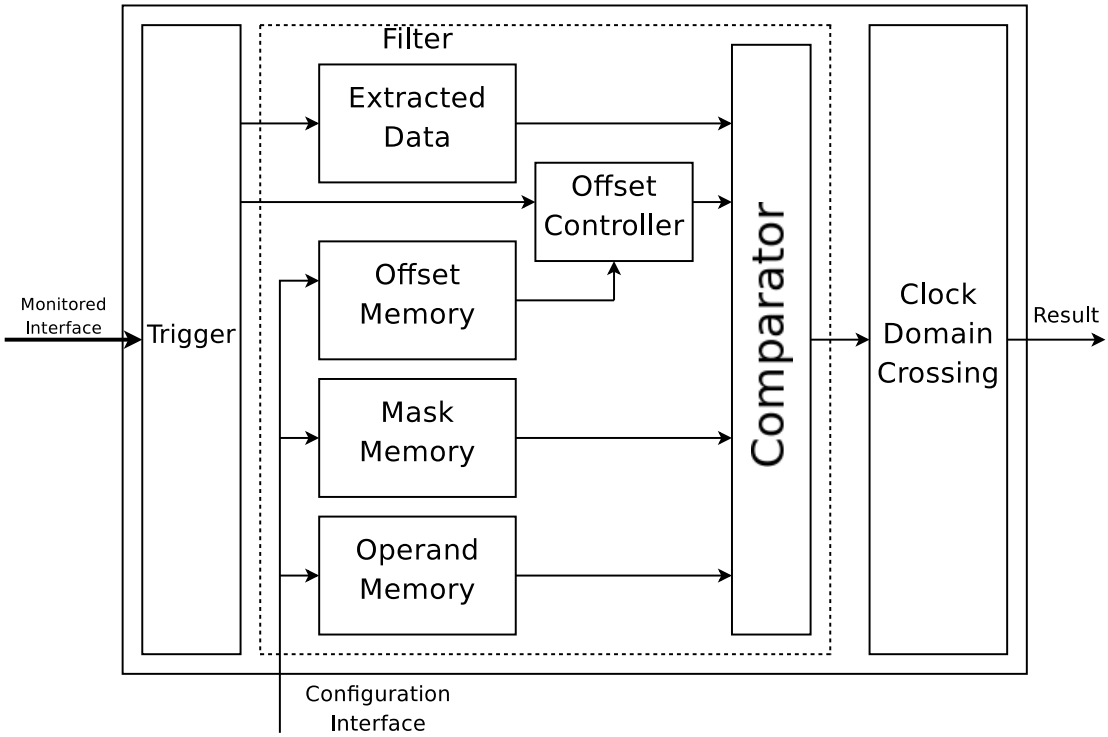

Figure 5.12: High-level architecture of the generic sequential matching probe with offsets.

However, it should be noted that the size of the generic filter is comparable to that of the fixed parser in terms of slices used, as shown in Table 5.4. Although it uses more LUTs overall, the tightly coupled nature of the memory in the CLBs allows the probe to be implemented in a compact form.

512-bit Generic Filter with Offsets

Although the generic filter gives greater flexibility compared to the specialised filter, it is limited to sequential matching. A designer is unlikely to examine the first 512 consecutive bits of a packet header. Instead, there is likely to be locality of interest, with several locations of interest at different points within the packet. For example, the designer is likely to be interested in the encapsulation of a packet. This may involve capturing all Ethernet frames containing IPv4 packets, which would entail matching the Ethernet type field and the IP version number. Subsequently, the designer might match against the 5-tuple and potentially deeper inside the packet.

Table 5.5: The resource utilisation of the generic parser with sequential matching using offsets.

Datapath Width	Slices	LUTs	FFs	Period (ns)
8-bit	137	270	38	5.131
16-bit	139	274	46	5.643
32-bit	230	322	66	5.307

The proposed filter allows an 8-bit offset from each match cycle, giving a maximum of 256 cycles between matches. The amount of memory devoted to matching is still 512 bits so for an 8-bit interface that matches 64 separate cycles the maximum theoretical offset is 16384 cycles. However, this limit will be much smaller in practise due to the locality of interest. The architecture of the filter is shown in Figure 5.12.

Table 5.5 shows that the resource requirements are slightly higher than the generic filter. This is due to the resource requirements for the offset memory and the more complex state machine. However, the offset mechanism employs the same placement structure, which uses the CLB memory efficiently. The resource requirements and timing constraints follow the same trend as those of the generic filter.

Scalable Filter

The last proposed architecture is intended to be more scalable and flexible. First, it allows the user to specify the exact number of match units at synthesis time. Second, it allows more flexible matches at run-time than other filters. For example, it permits the designer to match TCP and UDP packets by checking that the value of the next protocol field is 6 or 17 respectively. However, this flexibility comes at the expense of resources as more comparators are required due to the parallel nature of the scalable probe.

As shown in Figure 5.13, the scalable filter operates by creating multiple filters. Each filter has an independent offset, which can match against any one word within the first 256 clock cycles. This accommodates matching within the first $\alpha.\beta$ bits, where α is the datapath width and β is the number of clock cycles. For example, a probe operating on a 32-bit datapath can match up to the 256^{th} word, which allows matching from the $8{,}161^{st}$

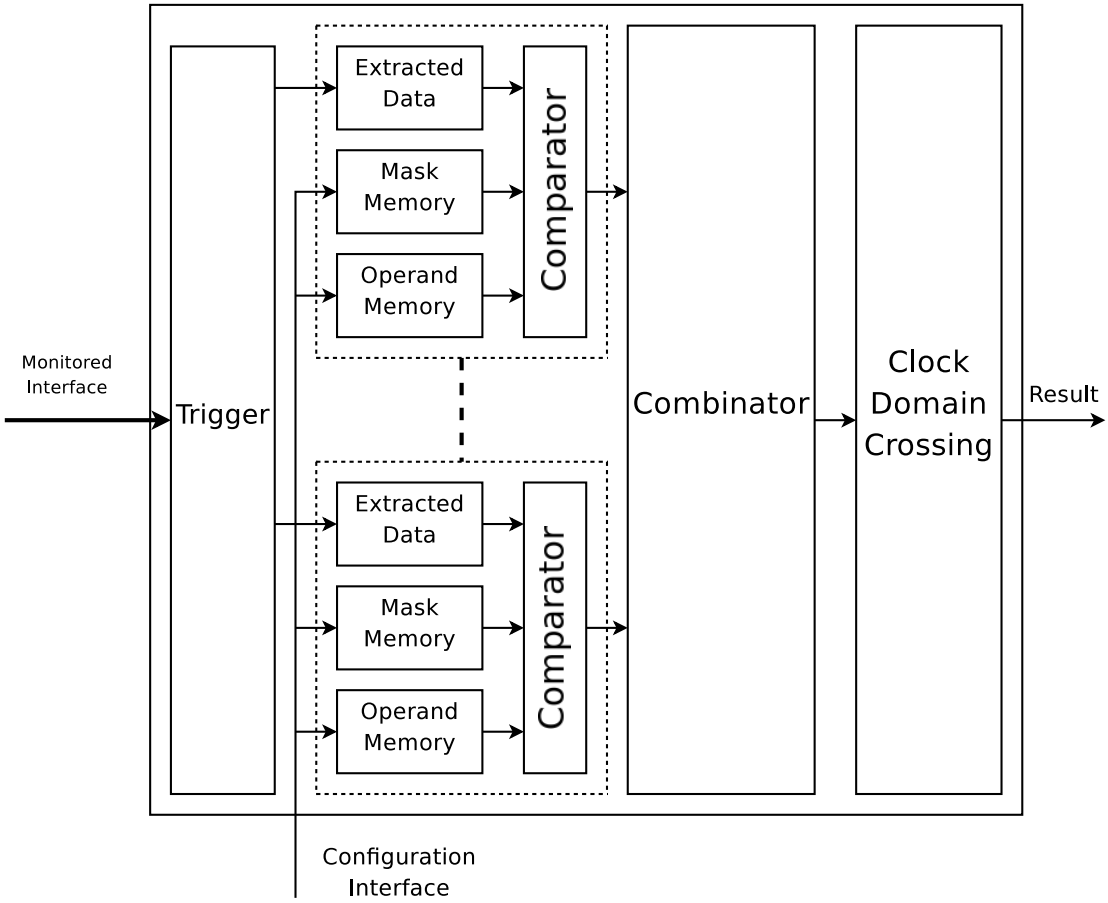

Figure 5.13: High-level architecture of the generic sequential matching probe with offsets and multiple match units.

Table 5.6: The resource utilisation of the scalable probe with 16 match units

Datapath Width	Slices	LUTs	FFs	Period (ns)
8-bit	258	341	56	4.295
16-bit	485	437	56	4.164
32-bit	844	693	56	4.400

Table 5.7: The resource utilisation of the scalable probe with 1 match unit

Datapath Width	Slices	LUTs	FFs	Period (ns)
8-bit	36	52	6	3.952
16-bit	62	74	6	4.128
32-bit	117	122	6	4.400

to the 8,192nd bits. The comparator must be the same size as the datapath width.

The system is controlled by a single transaction trigger, which instructs multiple atomic match units as to which cycles are valid to count or match on. As each atomic match unit is triggered and matches the value assigned to it, the outputs are combined and a result given from the combination. The combination unit implements disjunctive normal form of all trigger outputs from the atomic match units, which allows any possible logical combination to be expressed. To further optimise resource requirements, the number of possible conjunctive clauses could be constrained. For example, a compiler could determine the maximum number of conjunctive clauses needed before run-time. Overall, this approach is more flexible than can be achieved through any of the other proposed architectures.

The results in Table 5.6 show the utilisation for the scalable filter with 16 match units. The resource requirements increase at a greater rate with datapath width than other probe types due to the requirement for multiple match units and associated memory. Table 5.7 shows that the resource requirements for a filter with a single match unit are smaller than those for the generic filter. For a single match unit, the combinator does not need to be present. Furthermore, the memory requirements are smaller as only a single clock cycle can be matched. In this case, the functionality of the scalable probe is more limited than that of the generic filter.

Regular Expression Matching

The proposed probe architectures use masked matching to classify fields within a packet header. The offset capabilities of some architectures allow the probe to observe deeper within a packet. However, the features of the monitors do not lend themselves to deep packet inspection. An alternative technique for classifying packets is regular expression matching, which is more suited to these applications. Regular expression matching engines require significantly more resources. In particular, they require the use of BRAM resources which are unlikely to be available for monitoring the packet processing system. FPgrep [35] and FPsed [36] are two examples of regular expression matching engines. FPgrep uses a textual representation of a regular expression, which is converted to a state machine implemented in an HDL language. The regular expression matching systems also require protocol wrappers to interpret the headers of packet passing through the content scanning module.

The functionality of regular expression matching engines is excessive for packet classification but it is beneficial for deep packet inspection. FPgrep has been implemented in a Xilinx Virtex XCV2000E device and requires 4422 slices, 4547 flip flops and 22 BRAMs for implementing the protocol wrappers and content scanning module with a single search engine [35]. The system also operates at 37MHz in that device. Compared to the proposed architectures, this is a significant increase in the number of slices. The use of BRAM and a significant proportion of the device slices would preclude the use of regular expression matching from a system-level transaction monitoring tool.

Address Filtering

The address filter is designed to monitor transactions over shared-media, where the destination of a transaction is specified by the address of the target. As shown in Figure 5.14, the monitor contains a single trigger that interprets the signalling as perceived by a single bus master. The source of the transaction is then known as each probe only records events for a single interface and not the bus as a whole. The probe uses multiple match units to identify a range of destinations. Each match unit is independent and can filter independent address ranges. Additionally, each match unit has its own clock do-

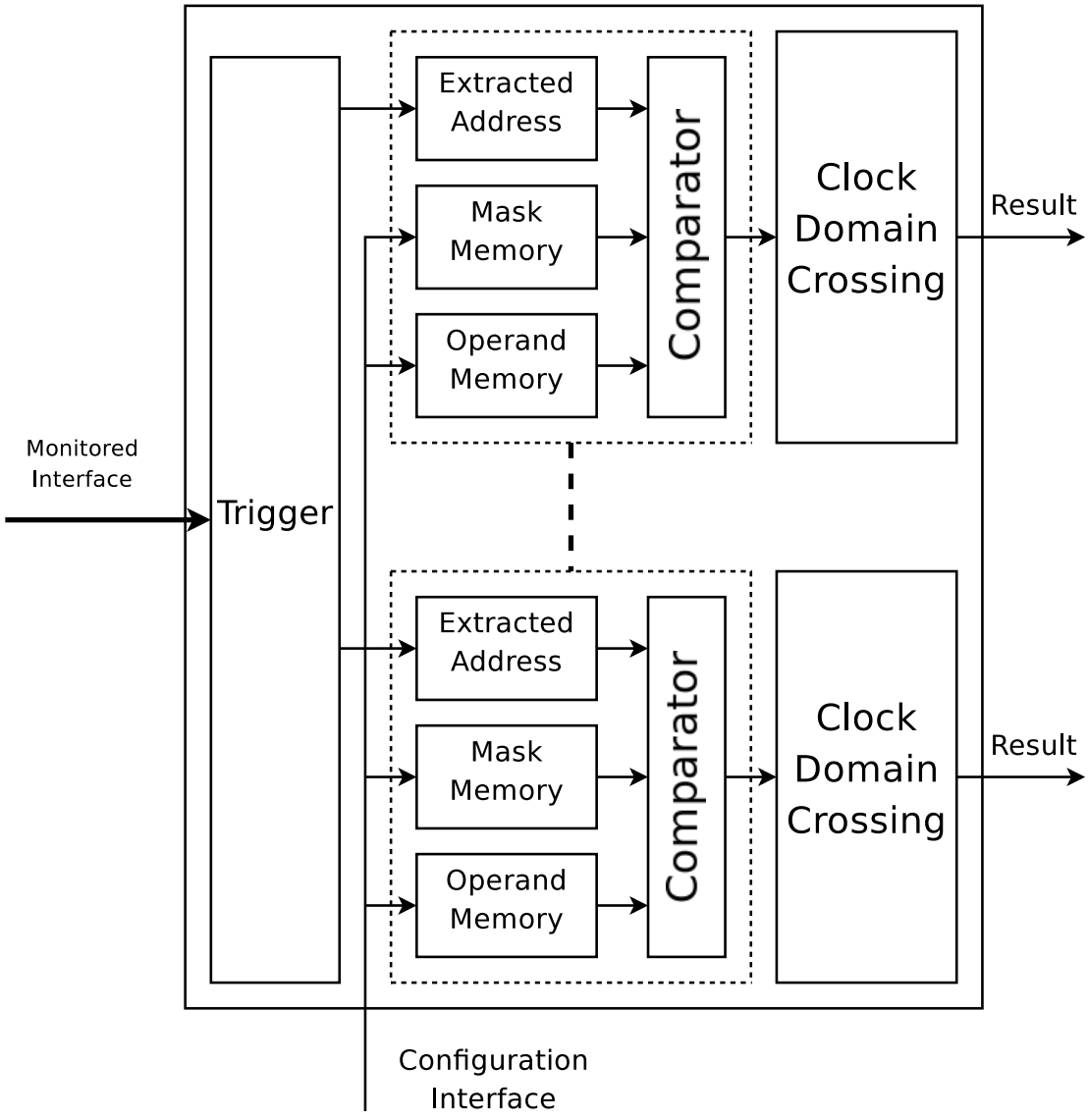

Figure 5.14: High-level architecture of the address matching probe.

Table 5.8: The resource utilisation of an address filtering probe with masked matching for monitoring PLB bus masters.

Match Units	Slices	LUTs	FFs
1	47	44	45
2	91	88	90
3	137	132	135
4	181	174	180
5	228	217	225
6	271	259	270

main crossing circuitry and generates a separate result. The individual results generated from the match unit, allow the source and destination information to be conveyed to the collector circuitry.

The address filter can be configured to monitor software function addresses, the address space of peripheral IP blocks or other parts of the system memory map. As shown in Table 5.8, the resource requirements increase nearly linearly as the number of match units increases. However, each match unit produces a separate output signal, which will affect the resource utilisation of the collector module.

5.3.3 Clock Domain Crossing

Most nontrivial designs require multiple clock domains in order to operate correctly. For example, a system processing Ethernet frames at 1Gbps will require a GMII interface operating at 125MHz. Following frame reception and buffering, the subsequent system components may use wider datapaths and a lower clock frequency in order to reduce the burden of meeting timing constraints. As the debugging system is designed to monitor several interfaces simultaneously it is likely to be used to monitor interfaces operating in different clock domains. Thus, it must be able to cope with recording transactions and transferring the results over different clock domains. The proposed solution transfers all events to a common clock domain, which drives the collector. Transactions are represented as level-sensitive signals, which can be transmitted across clock domains using pulse synchronisation, as shown in Figure 5.15.

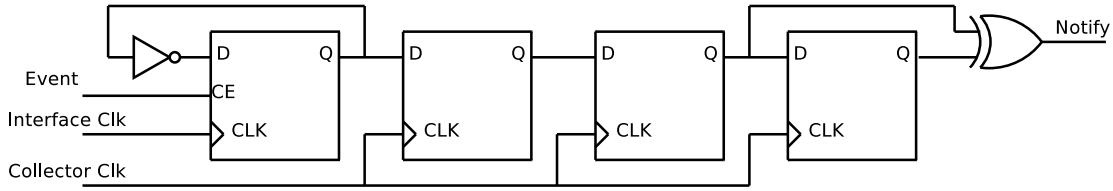

Figure 5.15: Diagram of the clock domain crossing circuitry shared by all probe architectures.

5.4 Collector Module

The probes transmit their results to the collector, which processes the spatial and temporal information received from the probes. The collector module also transmits its results to the external host software. Two variants of the collector module exist. The first architecture provides a profiling capability, which sacrifices timing accuracy for the ability to continuously monitor a design. The second architecture captures individual events. Event capture gives improved accuracy in observing event sequences but cannot guarantee continuous monitoring under all conditions.

The collector module has been designed to use a 115,200bps serial UART connection for communicating with the external software application. The UART connection was used as readily available on the development boards but the collector module could be easily extended to use other relatively higher-speed media, such as JTAG, for communicating with the external host software. This serial link can configure the monitoring system and upload results from the collector. A low-speed link is suitable for transmitting the results as transaction monitoring requires a lower bandwidth compared to low-level monitoring tools. The configuration mechanism is also decoupled from the monitoring circuitry, which negates the requirements for high-speed links.

5.4.1 Configuration Mechanism

The configuration mechanism is common between both collector architectures. In both cases, the serial link sends a sequence of two bytes to configure a single memory address. The first byte represents the 8-bit address to be configured and the second byte is the data to be written to the configuration memory. This provides a total of 256 addresses, which

can be assigned to the monitors in the system. The addressing format can be extended easily, for example, a three byte code could be used. In this instance, the first two bytes could represent the address and the third byte would represent the configuration data. This would provide a total of 65,536 memory locations. Alternatively, a scan chain could be used to configure the various memory elements, which would eliminate the need for an address space. However, these alternatives have not been explored in detail.

As previously stated, the data rate for configuration is unimportant. Configuration is decoupled from the operation of the monitoring system and the system under test. However, the impact of the configuration architecture on resource requirements is significant. Each probe type has a unique address space size and multiple probe types are frequently included within a single system under test. Each probe identifies the most significant bits of its address space, which generates an internal enable signal. However, each address within the probe's address space requires multiplexers to direct data to the correct memory location within the probe. This feature is part of the probe architecture and has been included within the resource comparisons in the previous section. Thus, adding probes at the system level does not require any resources beyond those already specified for the probes. High fan-out of the data bus and address bus can be tolerated as the data rates for configuring the system are low and buffers are automatically inserted as required.

As the probes have their address space specified at synthesis, it is possible for multiple probes to share the same address space. The configuration of the probes is unidirectional so a shared memory address space merely requires the probes to share the same base address. However, the probes need to be of the same type in order to interpret the configuration data correctly.

The configuration mechanism can also be used to control the system under test. For example, a reset control block can be inserted into a design, which can place the system under test in reset mode. The reset block can also control multiple reset domains for system buses, processors, peripherals and other components. Combined with a set of stimuli for the system, this provides the ability to automatically test and monitor the system.

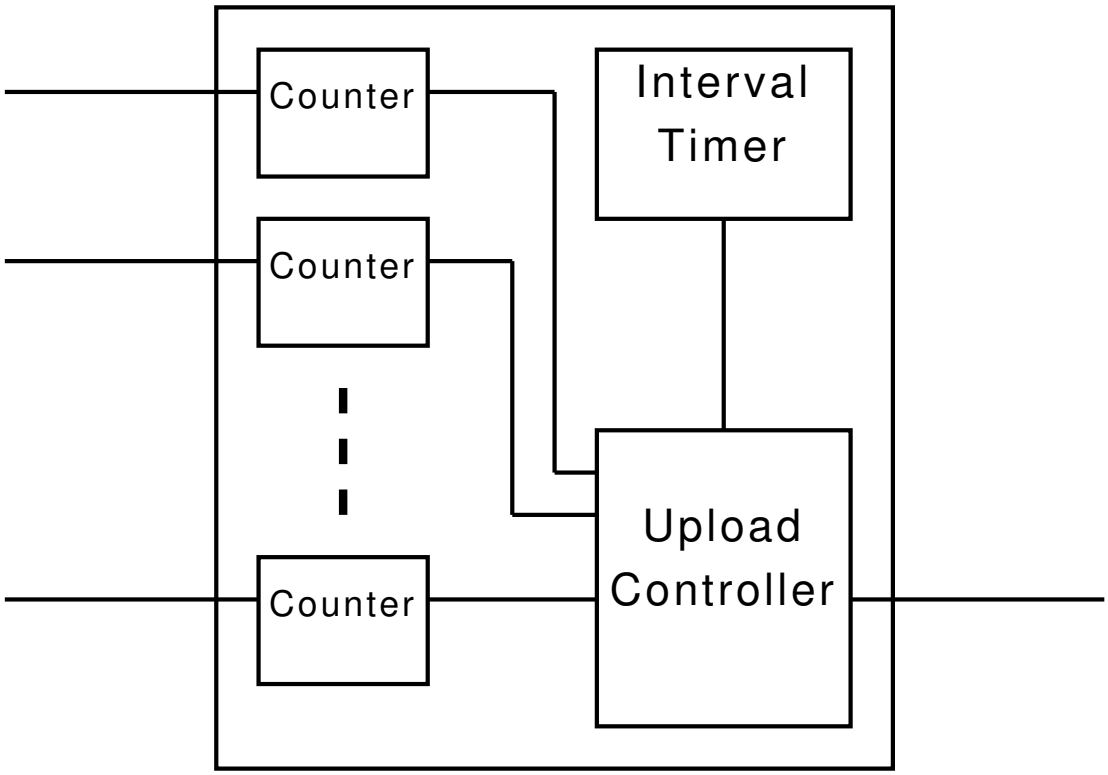

Figure 5.16: Architecture of statistical collector

5.4.2 Profiling Collector

The profiling collector is designed to provide a means of continuously monitoring a system without requiring buffering resources. It counts the number of transactions that occur within a period of time and match the given filtering criteria. This allows a different class of errors to be detected that cannot be observed with traditional low-level monitoring tools. For example, performance bottlenecks and events of long duration can be observed by using the profiling collector.

The profiling collector consists of a timer, set of counters and a controller as shown in Figure 5.16. The timer is used to determine when a sampling period has completed, while the set of counters accumulate the number of completed transactions that occur within the sampling period. Each probe has its own unique counter, which maintains the spatial information provided by the probes. Upon reaching the end of a sampling period, the values of the counters are copied to temporary registers, allowing the counters to be reset and accumulate results for the subsequent time period. Using temporary registers allows

the time period to be measured accurately. The controller is responsible for coordinating the copying process and uploading the results from the temporary registers to the external host software. The system also provides parametrisation to cope with a flexible number of probes in the system under test. The current implementation limits the number of probes to 16 but this can be easily extended.

The profiling collector employs a simple encoding scheme for uploading the results to the external host software. In this scheme, the collector transmits a datagram for each probe. This datagram consists of an identifier for the probe location and the result obtained for that probe. The time interval is recorded by transmitting a tuple, consisting of the datagrams for each probe. The subsequent time interval is indicated by the transmission of the datagram for the initial probe.

As the system uploads results periodically, it is necessary to determine the minimum sampling period for the profiling collector. Intuitively, the minimum permissible sampling period is the time taken to upload the results of all probes for one period to the external host software. If the minimum sampling period was shorter than this then buffering would be required to store results on-chip. Thus, the minimum sampling period for the simple encoding scheme is a function of the number of probes, the number of bits for the probe identifier, the size of the counters for each probe and the speed of the link to the external host software.

Assuming that the counters for each probe are of identical size, then the expression for the minimum interval is given in 5.2, where α is the minimum sampling interval, N is the number of probes, β is the number of bits required for the probe identifier, δ is the number of bits required for the probe counter and γ is the upload speed in bits per second.

$$\alpha = \frac{N\,(\beta + \delta)}{\gamma} \tag{5.2}$$

The simple encoding scheme also permits the external host software to synchronise itself to the uploaded datagrams. The increasing sequence of probe identifiers allows the host software to determine which bytes are the identifiers and which are the results. However, a more efficient encoding scheme could reduce the minimum sampling period, which would reduce the number of transmitted bits. For example, the values of all counters could be

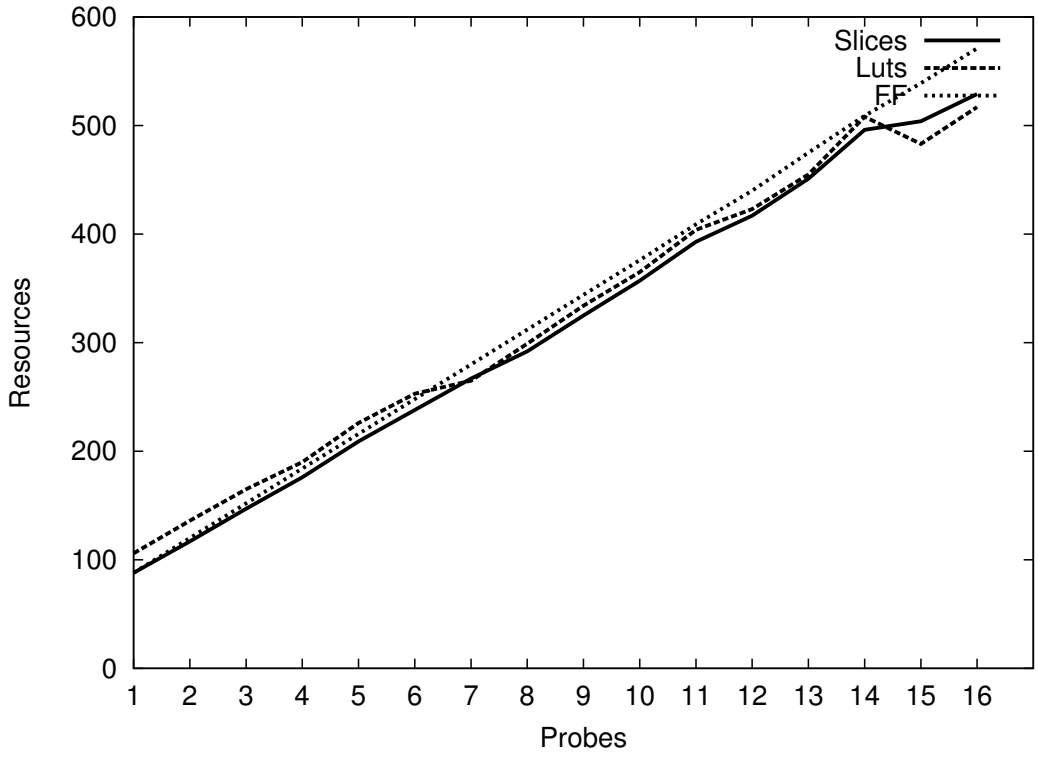

Figure 5.17: Resource utilisation of the profiling collector

grouped together into a single datagram with a short preamble. The preamble is necessary to permit the external host software to synchronise on the start of the datagram. The minimum time interval is given by expression 5.3, where ϵ is the number of bits in the preamble.

$$\alpha = \frac{(\epsilon + N\delta)}{\gamma} \tag{5.3}$$

As the number of probes increases, the encoding scheme described by expression 5.3 gives a smaller minimum sampling period. This statement holds true provided that the preamble is smaller than the overhead of transmitting a probe identifier with each sample.

As shown in Figure 5.17, the resource requirements of the profiling collector increases almost linearly as the number of probes increases. Each additional probe requires a counter, temporary register and a connection to the upload controller. The connection to the upload controller tends to consume the majority of the extra resources as the multiplexer connections become more complex.

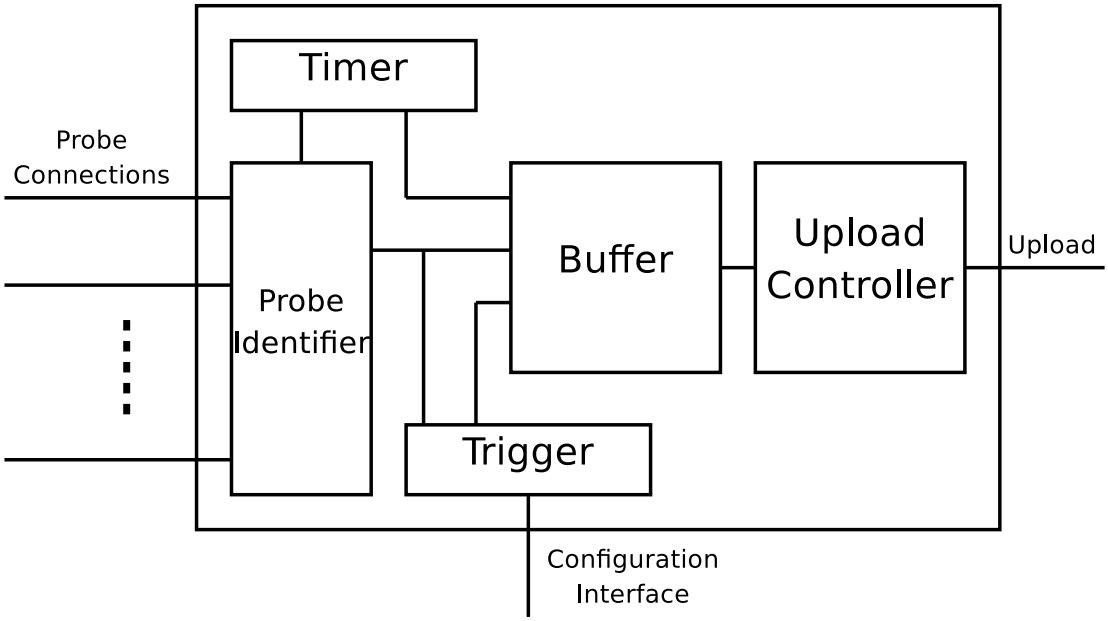

Figure 5.18: Architecture of transactional collector

The profiler provides several benefits over traditional low-level monitoring tools. First, it provides the capability to monitor systems continuously. The temporal representation of events allows monitoring to be continuous. Second, it demonstrates the performance of a given architecture. Third, it can identify system bottlenecks. As locations can be monitored for long periods of time, link utilisation trends can be followed for various interconnects. This provides an overview of system performance and allows potential bottlenecks to be highlighted. By the same virtue, the monitoring system also has the potential to show whether deadlock is present in the system. This would be observed by a lack of events. Finally, the profiling collector can identify the paths of events by exploiting the filtering capabilities of the probes. By filtering various locations for the same properties, the path of events will be shown and the IP blocks which operate on the events will be highlighted.

5.4.3 Event Collector

The event collector is designed to capture individual transactions and report them to the user. This provides a detailed system-level view of the component interactions. The event collector also allows the designer to observe the temporal relationship between individual

transactions. For example, with event capture the user can determine whether a path executes as a pipeline or not. It can also determine the location of an error more quickly than with low-level monitoring tools.

As shown in Figure 5.18, the collector is composed of a probe identifier, timer, trigger, buffer and upload controller. The probe identifier is responsible for giving each probe a unique identifier to be used by the external host software. The start and end of data collection is controlled by the trigger. The timer is used to present the temporal relationship between events and the upload controller is responsible for transmitting the results to the host.

The probe identifier assigns an unique identifier to each probe in the system, which is then appended to each event before it is uploaded to the host. This allows the external host software to determine the location of each event. The identifier is also used by the trigger to determine when to start data capture. As with the profiling collector, the event collector can specify the number of probes as a parameter during synthesis.

As multiple events may be recorded on the same clock cycle, the results from the probes are registered. The probe identifier is implemented as a priority encoder which provides a deterministic method for multiplexing events. The highest priority identifier is presented first and is then followed by the second highest priority identifier on the following clock cycle. Multiple identifiers cannot be recorded simultaneously as the collector must multiplex events for serial transmission to the host. This means that the recorded transactions might not be cycle accurate. However, this level of detail is not required to convey a system-level perspective. For a system being monitored by 16 probes, the critical instant would require 16 clock cycles to record all events into the buffer. However, the buffer might require multiple clock cycles to upload an event over a serial connection.

Using a priority encoder also presents the potential for starvation. It is conceivable that one or more higher priority interfaces continually present events for identification preventing the lowest priority event from being recorded. In practise, this is generally not a problem as transactions usually occur over multiple clock cycles. Furthermore, the impact of this problem can be minimised by developing a profile of each monitored interface using the profiling collector.

Alternatively, a different representation could be used for encoding the probe identifier. Each probe could be assigned a designated bit within a word, where the word is the same size as the number of probes in the system. When an event is recorded, all interfaces that detected an event on the same clock cycle would assert their designated bit within the word. This would permit multiple transactions to be recorded on the same clock cycle and remove the need to register the probe results. However, this is not a scalable solution as the size of the datagram for uploading events to the host would vary according to the number of probes in the system. For large systems, the overhead for each transaction could reduce the temporal visibility of the monitoring system by filling the buffer more quickly. This alternative encoding strategy has not been implemented.

The collector uses a run-time configurable trigger to determine when to start event capture. Once started, event capture continues until the buffer is full. The trigger complements the filtering present in the probes, forming a distributed triggering system. The current implementation limits the number of probes to 16 but this can be easily extended. The triggering mechanism could also be extended to permit the use of transaction sequences to start event capture.

The event collector uses a single timer to relate all events to its clock domain, as a common reference. This maintains the accuracy of clock domains on lower frequencies relative to the collector but exhibits inaccuracies for higher frequency domains. However, this does not require the timer to operate using the highest clock frequency in the design. First, the event collector does not guarantee cycle accuracy. This is also a consequence of the priority encoder for the probe identifier. Second, operating at the fastest clock frequency is often undesirable due to the effect on placement and routing. For a complex design, such as the collector, it can be difficult to obtain high clock rates. Additionally, the fastest clock rate is generally used on a small portion of a design. Within packet processing applications the fastest clock rate is typically used for communicating off-chip. The remainder of the design uses wide datapaths to reduce the clock rate and ease the burden on placement and routing. A lower frequency is therefore more desirable but it should match the clock frequency of the majority of the design to maintain a degree of accuracy. This technique can maintain the relative order of events provided that the probes are connected to the collector using the correct priorities.

The timer is the highest priority event presented to the probe identifier, which occurs when the timer rolls over. The current implementation uses a 24-bit counter, which gives a period of 0.335 seconds at 50MHz. It is possible to use a smaller counter to reduce the size of uploads but this would need to be balanced with an increased frequency of timer rollovers.

The buffer receives events presented by the probe identifier and the timer. It can be implemented using Block RAM or Distributed RAM. Using a BRAM on a Xilinx Spartan 3E device, 512 entries can be scheduled for transmission. When the buffer is full, the trigger is notified and event capture terminates. If the buffer is never filled then it is possible to maintain continuous monitoring. However, this is dependent on the frequency of the recorded events and the data rate of the upload controller.

The upload controller is responsible for uploading recorded data to the external host software. As events are placed into the buffer, they are then scheduled for transmission to the host. The datagram for uploading an event consists of the probe identifier and the time of the event according to the collector timer. The probe identifier requires 8-bits providing a theoretical maximum of 255 probes after subtracting the timer rollover signal. If the alternative encoding was used then only 8 probes could be used for an 8-bit probe identifier. Time is represented using 24-bits, which means that each event requires a total of 32-bits to be transmitted. While there is data present in the buffer, the upload controller will transmit it to the host.

As shown in Figure 5.19, the resource requirements of the profiling collector increase in a non-linear fashion. For small numbers of probes, the resource requirements are fairly linear. However, when 8 or more probes are instantiated the resource requirements increase at a higher order of magnitude, which is due to the resource requirements of the priority encoder. A more efficient probe identifier could reduce the resource requirements for large numbers of probes.

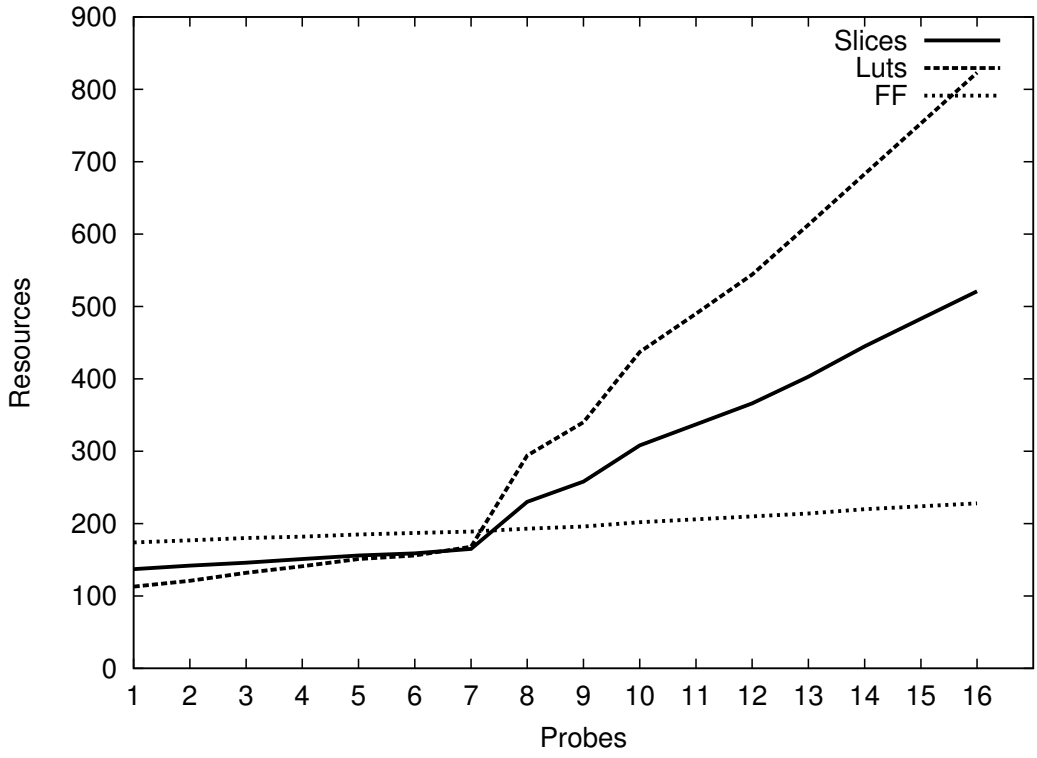

Figure 5.19: Resource utilisation of the event collector with priority encoding

5.5 External Host Software

The on-chip monitoring system is complemented by software executing on an external host. The software has several responsibilities, which include receiving results from the collector, synchronisation of the received data, configuration of the monitoring system, correct interpretation of the results and presentation of the results to the user. The software can be configured to interact with the various probe implementations and collector architectures, which allows the designer to use a single software application regardless of the exact configuration of the monitoring system.

As shown in Figure 5.20, the host software can be represented as a state machine. The first task performed by the software is to parse the configuration file, which specifies the number of probes and the type of the collector used in the system. This file also describes the location of the probes and the translation of that location to a user defined representation. The configuration memory map and the configuration sequence for each filter are also described. After the configuration file has been parsed, the host software

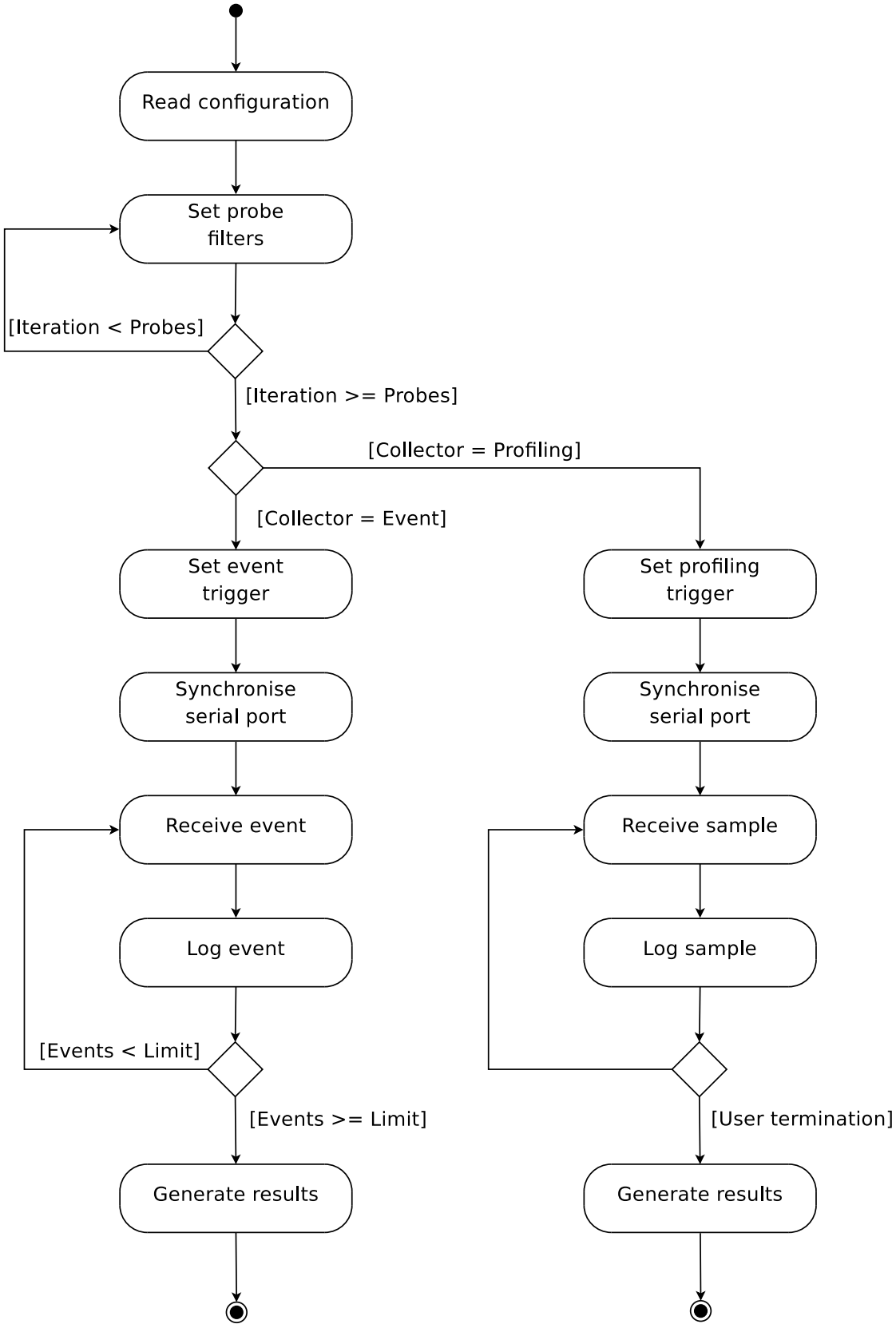

Figure 5.20: UML state diagram representing the operation of the host software.

iteratively configures each probe through the configuration circuitry.

Following the configuration of the probes, the host software configures the collector's trigger. At this point the operation of the host software is dependent on the type of the collector so the software branches into two distinct sequences. Although both sequences set the trigger for the collectors and synchronise the serial ports, the methods for performing these actions are different and require separate functions. The synchronisation of the serial port allows the software to interpret the data sequence correctly by determining the start of the payload being transmitted. The event collector might have periods with no data transmission as the events detected by the system may be sporadic, which simplifies synchronisation. However, the profiling collector transmits data continuously and requires the start of a sample to be correctly identified within a continuous set of data.

As the profiling collector uploads the results of each probe in sequence, the probe identifier can be used to synchronise the serial port. The host software examines the sequence of data being received and determines whether an incremental series of numbers is present. This series does not exceed the maximum number of probes in the system and is present at a fixed offset in the data stream, which is determined by the size of the samples used by the collector. An error is raised if the sequence cannot be detected.

Once the serial port has been synchronised, the host software receives samples from the collector on the FPGA. Each sample consists of the results from every probe. The host software logs the samples and continues to receive samples until it is instructed to terminate execution by the user. Once termination has been invoked the software ensures that all of the received data is written to an external file, which the designer can then use in other applications.

The event collector differs as it only records a finite number of events, which prevents the on-chip buffer from overflowing. The desired number of events is specified in the configuration file but it can be omitted to permit continuous monitoring, if permitted by the system being monitored. Following the specified number of events the host software terminates execution and ensures that the received data is written to the external file.

The textual output generated by the host software can be used in other software tools or scripts. Such tools could compare execution traces and provide graphical representations

of the recorded data. Furthermore, the host software could be used as part of an integrated development environment where the configuration file is automatically generated and the results are presented in a form suited to the development tool.

5.6 Summary

This chapter has presented a monitoring tool that addresses some of the limitations of traditional monitoring solutions. In particular, the tool provides a system-level perspective by monitoring the communications between IP blocks and abstracting the communications as transactions. The resource requirements of the monitoring tool are reduced compared to other alternatives and it uses lower data rates for uploading results as a results of transaction monitoring. System-level transaction monitoring also provides improved localisation of error conditions by monitoring the communication between components and abstracting low-level signalling.

The monitoring system consists of a set of probes and a collector. The probes are designed for the specific interfaces being monitored which reduces the resource requirements and reduces the impact on timing and component displacement. The collector interprets the results from the probes and processes the information before uploading its results to the host software. The host software is responsible for configuring the monitoring system, receiving the results and presenting it to the user.

This chapter has presented the common architecture of the probes, which consists of the transaction interpreters, filtering mechanisms and clock domain crossing circuitry. The filtering mechanisms have been classified into payload and address filters. Several payload filtering variants and an example of address filtering have been explored and the resource requirements for each variant have been presented. The specialised filter allows the designer to match any field determined before synthesis. The generic filter provides greater flexibility by permitting filtering according to a user-specified set of fields at runtime. The offset filter allows the designer to inspect deeper inside a packet up to the limit of the offset. The scalable filter provides the same capabilities but extends the combination of triggers to provide more flexible combinations of fields. The functionality of each probe has been explained and the potential applications have been highlighted.

The chapter has also presented the architecture of two different collector mechanisms. Each mechanism uses the same interface from the probes but interprets the data in a different manner. The profiling collector records the number of events within a period of time and can provide continuous monitoring. It is useful for observing system operations over a long period of time and detecting bottlenecks. The event collector records individual events and relates all events to a common representation of time. Finally, the functionality provided by the host software has been explained.

Chapter 6

Evaluation of Dynamic Monitoring

The system-level transaction monitoring system captures information that can be considered high-level compared to traditional on-chip monitoring tools. As demonstrated in Chapter 5 the resource requirements of various probe architectures are low but these requirements need to be compared with a traditional monitoring tool. This chapter presents the results of implementing the monitoring system in three example designs. The information captured from the low-level monitoring tool and the system-level monitoring system are compared. The resource requirements of both tools are also contrasted.

6.1 Introduction

In order to demonstrate the class of errors detected by the system-level transaction monitoring system, three case studies will be presented. These examples also provide an insight into the resource requirements of the monitoring system in a typical application and demonstrate how a design could be debugged at the system-level. The first case study will discuss a web server implemented on a soft processor. The second case study will discuss an advanced web server, which executes software on a hardened embedded processor. This system also implements some traditional software functions, such as checksum calculation, as hardware components. The third case study will examine a firewall imple-

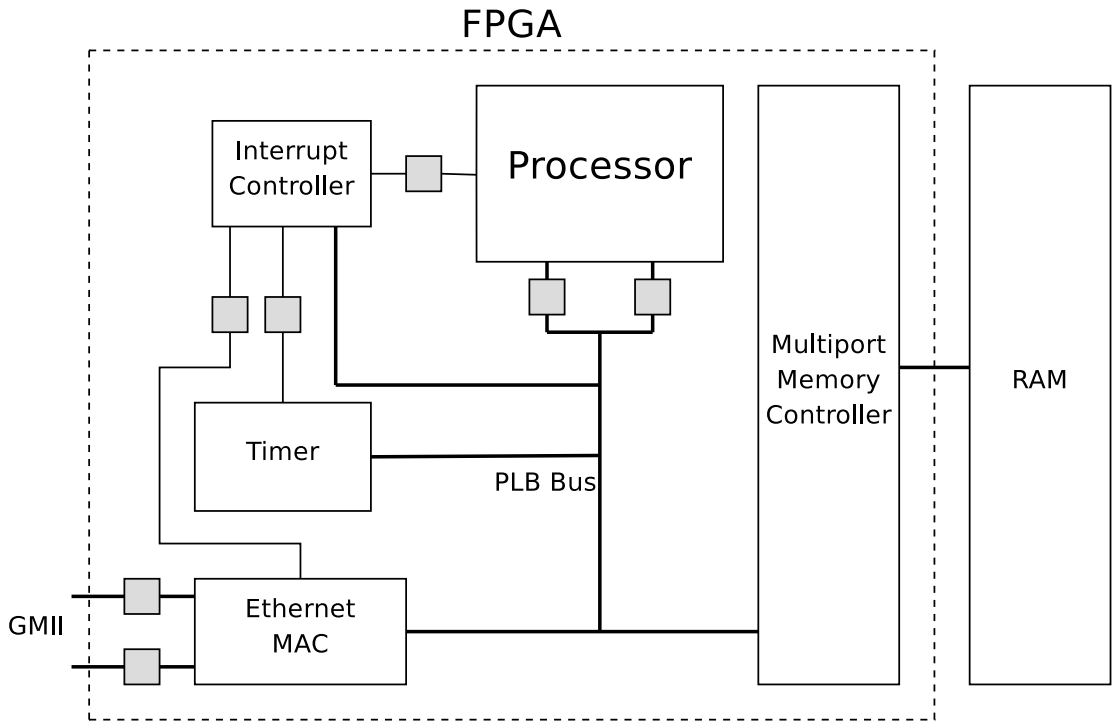

Figure 6.1: System architecture of simple web server

mented entirely as hardware logic in the FPGA fabric. An algorithm for automatically instrumenting designs with this monitoring system will be presented in Chapter 7.

6.2 Simple Web Server

The simple web server is designed to demonstrate combined software and hardware monitoring. The system is comprised of a Microblaze soft processor, soft Ethernet MAC, system timer, multi-port memory controller and interrupt controller, as shown in Figure 6.1. There are two interrupts in the system, which are the timer interrupt and the Ethernet MAC interrupt. The timer interrupt occurs at regular 10 ms intervals and is used by the kernel to manage software timers, thread scheduling and other activities. The Ethernet MAC interrupt is raised each time a packet is sent or received on the MII interface. The interrupt handler is responsible for scheduling the transfer of the packet from the internal buffer of the Ethernet MAC to the system memory.

As shown in Figure 6.1, the Ethernet MAC is connected to an external PHY, which is

connected to an Ethernet network. Internally, the Ethernet MAC is connected to the interrupt controller and the PLB bus. The Ethernet MAC has an internal frame buffer, which is accessible from the PLB bus. The Microblaze processor is also connected to the PLB bus, which allows it to communicate with all of the peripherals. The processor has two master interfaces to the bus, which provide data and memory operations. The bus is used for configuring the peripherals and for transferring data between IP blocks.

The system uses a simple kernel, called Xilkernel, for thread scheduling, interrupt handling and IO operations. This is supplemented by the lwIP stack, which provides facilities for processing IP packets. A simple web server application was written for the kernel, which is responsible for initialising the peripheral IP cores and configuring the lwIP stack during system boot. The web server also binds itself to port 80 and listens for connections on that port.

While the system is executing the following responses are produced when a packet is received on the Ethernet MAC MII interface. First, a packet is received on the MII interface and stored in the Ethernet MAC's internal buffer. The Ethernet MAC then raises an interrupt request, which stops the current execution of the processor and causes it to service the interrupt. The interrupt service routine transfers the packet from the Ethernet MAC into the system memory using the processor to coordinate transfer as no DMA controller is available in the system.

Once the packet has been copied to the system memory, the lwIP stack is scheduled for execution by the kernel. The lwIP stack validates the frame by checking the addresses and calculating the appropriate checksums. Upon receiving a connection, the web server application spawns a separate thread to respond to the request. The response returns a standard web page, which is then processed by the requesting entity. The response is converted into a series of packets by the lwIP stack and copied from system memory to the Ethernet MAC buffer, which transmits the packets to the requesting party.

6.2.1 Profiling

The simple web server has been instrumented with both a profiling collector and an event collector. For the purpose of this example the profiling collector has been used and PLB

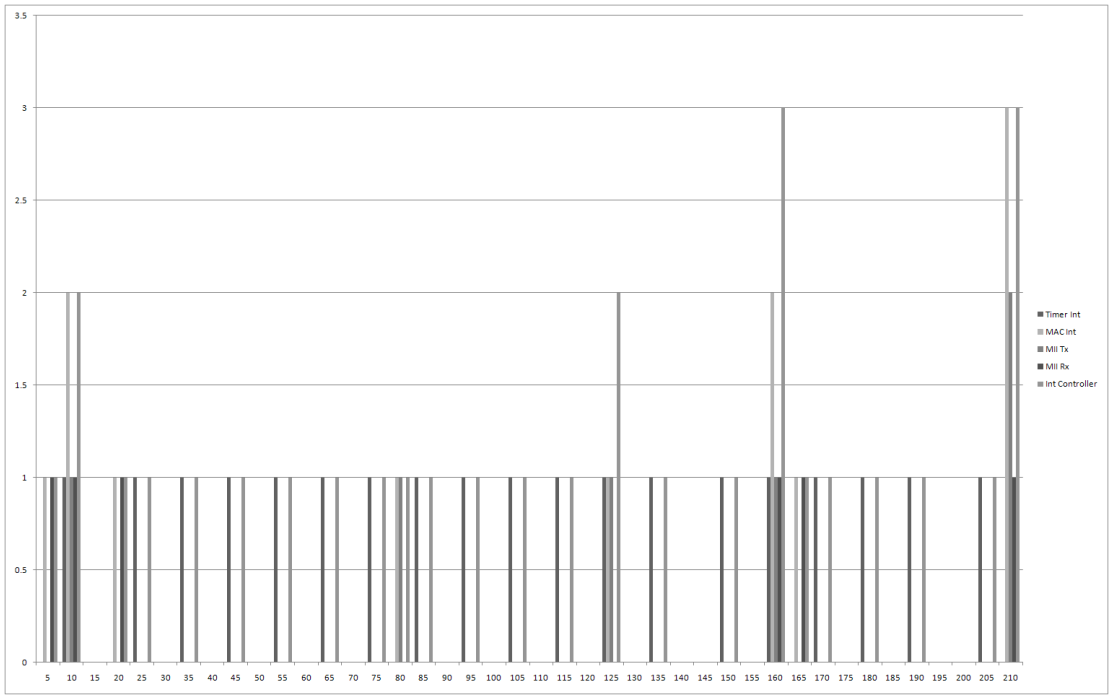

Figure 6.2: Simple web server response to HTTP GET request

bus has not been instrumented. In this example, the web server has been instrumented with transition probes to monitor interrupts between the timer, the Ethernet MAC and the processor. Finally, the MII interface has been instrumented with two generic probes for detecting packet transactions. Filtering is only performed on the MII interface as the other signals are system interrupts. The software perspective will be examined with the event collector and the advanced web server.

Figure 6.2 shows the response of the system when subjected to an HTTP GET request. The regular interval of the system timer is shown and is clearly visible with the sampling period of 5ms. The subsequent interrupt generated by the interrupt controller is also visible. Figure 6.2 also shows that for every packet sent or received on the MII interface, there is an associated interrupt. While Figure 6.2 shows the response for a single HTTP GET request, the response for multiple HTTP GET requests can be easily extrapolated. In this instance, there will be multiple MII transactions, MII interrupt requests and subsequent processor interrupts. The total number of processor interrupts should equal the number of MII transactions and timer interrupts. However, there may be a temporal delay before the totals equalise due to the quantisation effect of the sampling period and

Table 6.1: Resource utilisation of the simple web server with profiling collector

	Uninstrumented Design	Profiling Collector	ChipScope	Device Resources
Slices	3728	4231	4779	4656
LUTs	5481	6281	6901	9312
Registers	4347	4908	5323	9312
BRAMs	13	13	19	20

the delay of the system.

The delay of the system means that events may not occur until later samples, which is an effect of the design's architecture and not a result of the monitoring system. However, the quantisation effect is due to the monitoring system. Quantisation means that an event which starts in one time sample may not be recorded until the following time sample. This may cause two events which are closely related temporally to be placed into separate time samples. Thus, a quantisation error can be calculated for the samples in a time interval.

The resource requirements of the profiling collector are comparable to those of Xilinx's ChipScope tool, as shown in Table 6.1. Although the system without instrumentation requires a significant amount of resources, both monitoring systems can be applied in this instance. Unlike ChipScope, the profiling collector provides observation of event ordering and does not use any BRAM resources. Furthermore, the temporal information obtained can also replace that of network monitors. Like network monitors, the time between packet injection and response reception can be observed but more information on the internal system events can also be obtained.

6.2.2 Event Capture

The simple web server has also been instrumented with an event collector and has the same architecture as shown in Figure 6.1. In this example, the web server has been instrumented with transition probes to monitor interrupts between the timer, the Ethernet MAC and the processor. The PLB bus has also been instrumented with address monitors to observe transactions on the bus. Finally, the MII interface has been instrumented with two generic

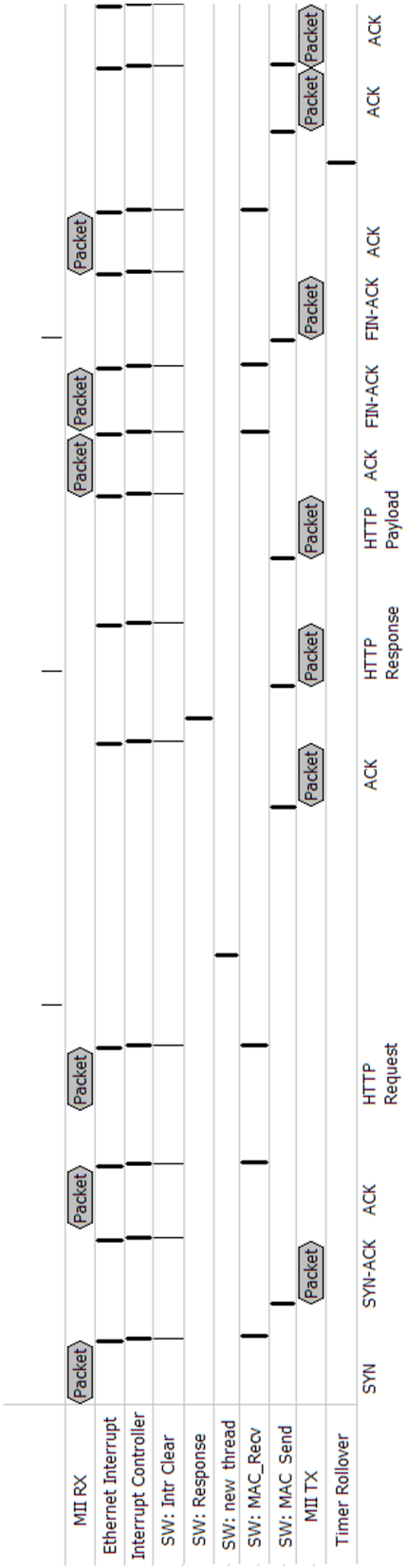

Figure 6.3: Web server response for HTTP web page request

Table 6.2: Resource utilisation of the simple web server with event collector

	Uninstrumented Design	Event Collector	ChipScope	Device Resources
Slices	3800	4013	4026	4656
LUTs	4628	5162	5731	9312
Registers	3878	4296	4825	9312
BRAM	13	14	19	20
BUFG	7	7	8	24

probes for detecting packet transactions. Figure 6.3 illustrates the response obtained when the web server is stimulated with an HTTP GET request. The figure also shows the path of events from frame reception on the MII interface through to the interrupt clearance by the interrupt handler. The execution of software functions designed for receiving and sending Ethernet frames and generating the HTTP response are also shown. Finally, Figure 6.3 shows the flow of packets used to create a TCP connection, initiate an HTTP request and terminate the connection. The observed flow of packets has been validated by comparing the results to those captured using Wireshark [128].

Table 6.2 shows the resource requirements of the web server and the available resources of the Spartan 3E device. The system without monitoring circuitry uses 82% of the available slices, which can make instrumentation difficult. The instrumented design has the ability to match several addresses on the PLB bus and allows software function tracing while only using 86% of the available slices. As the web server does not use a cache, all processor instructions are transferred over the PLB bus, which allows the addresses of software functions to be matched as they are fetched. The ChipScope implementation uses approximately 86% of the available slices but also requires an additional 6 Block RAMs. The ChipScope instrumentation can support software function monitoring but is unable to relate the timing of software instructions relative to other events in the system. The low-level monitoring mechanism used by ChipScope allows processor instructions to be captured and related to the assembly source code.

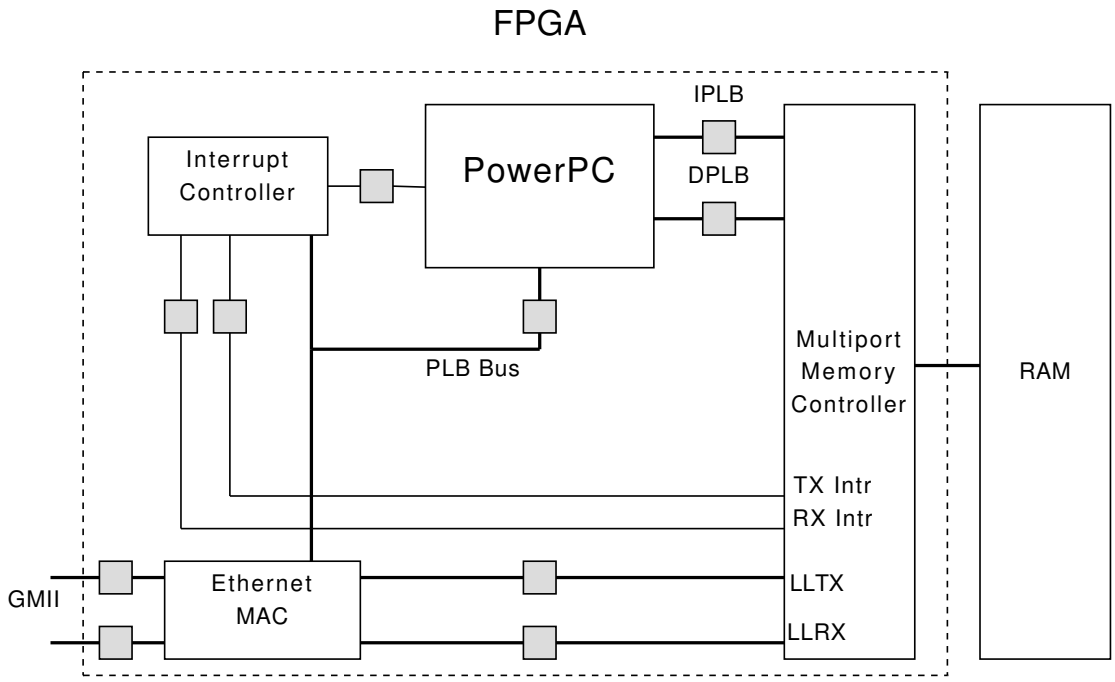

Figure 6.4: System architecture of Virtex web server

6.3 Advanced Web Server

The advanced web server consists of a PowerPC, Multiport Memory Controller (MPMC), hardened Tri-mode Ethernet MAC (TEMAC), interrupt controller and debug infrastructure, as shown in Figure 6.4. The PowerPC has an integrated timer, which is not visible on the FPGA fabric. The TEMAC provides the ability to perform TCP and UDP checksum offloading, which calculates packet checksums using the FPGA fabric. The TEMAC is directly connected to the MPMC which provides direct access to memory. Upon receiving a packet the MPMC raises an interrupt, which is received by the interrupt controller. Following this, the interrupt is passed onto the processor, which halts its current execution. The interrupt handler then obtains a pointer for the received packet and the software continues processing. The software compares the calculated checksum with that present in the packet and performs other checks as appropriate for TCP packets and UDP datagrams.

Conversely, when sending a packet, the processor creates a packet descriptor in memory and passes a pointer to the MPMC. The MPMC then uses the descriptor to transfer the packet to the TEMAC, which in turn transmits the packet over the GMII interface. The

Table 6.3: Resource utilisation of the advanced web server with profiling collector

	Basic Design	Probes	ChipScope	Device Resources
Slices	6195	6667	x	8544
Registers	6569	7175	x	17088
LUTs	7404	8165	x	17088
BRAMs	35	35	x	68
BUFGs	10	11	x	32

TEMAC is also responsible for calculating the appropriate checksums.

The system has three PLB buses. Two are used for instruction and data accesses via the MPMC and the third is used for communicating with on-chip peripherals such as the TEMAC and MPMC registers. The processor executes Xilkernel for scheduling threads for processing, which is supplemented by the lwIP stack. The lwIP stack provides packet processing facilities and functions for handling IP packets. Using the features of both libraries, a relatively advanced web server was created. The application has also been extended to support a simple time server as well.

The web server application is responsible for initialising the peripheral IP cores and configuring the Ethernet interface to use the lwIP stack. The web server is bound to port 80 and listens for TCP connections. Upon receiving a connection a separate thread is spawned and the response is sent. The response is a standard web page. The time server also spawns a separate thread for each request, which is inefficient in comparison to modern software engineering techniques but serves well to demonstrate the operation of the system.

The web server was stimulated by sending an HTTP GET request from a standard web browser and the response on the GMII interface is shown in Figure 6.5. The figure shows the time taken for creating a TCP connection, the time lapsed for an acknowledgement to be transmitted, the delay before the payload is sent and the time taken for tearing down the connection. The results shown in Figure 6.5 are comparable to the results obtained through WireShark using the same stimulus.

Although the interface to the system can be monitored, it was also insightful to observe

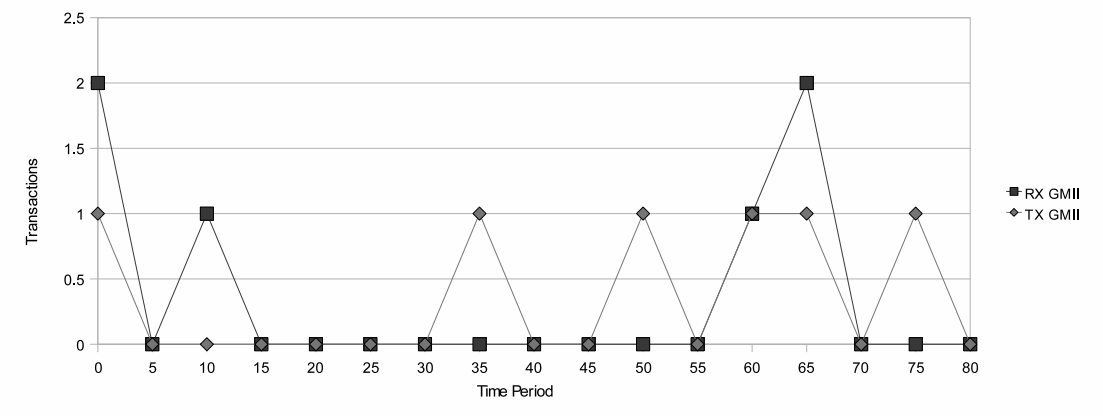

Figure 6.5: Web server system response to HTTP GET request

Table 6.4: Advanced web server response to received packets

Time (ms)	RX GMII	RX LL	RX Int	MAC Int	PPC Int	PLB Writes	PLB Reads
0	0	0	0	0	0	0	0
5	1	1	1	1	1	6	7
10	0	0	0	0	0	0	0

Table 6.5: Advanced web server response to transmitted packets

Time (ms)	TX GMII	TX LL	TX Int	MAC Int	PPC Int	PLB Writes	PLB Reads
0	0	0	0	0	0	0	0
5	1	1	1	1	1	4	5
10	0	0	0	0	0	0	0

the internal operation of the system. Table 6.4 shows the system response for receiving a packet while Table 6.5 shows the system response for transmitting a packet. These tables show that the reception and transmission of a packet on the GMII interface cause other events in the system on the LocalLink interface, interrupt controller and PLB bus. However, due to the large sampling interval of the profiling collector, it is not possible to infer the ordering of individual events. A faster upload mechanism would allow smaller sampling intervals to be used and event ordering to be distinguished. Using the current sampling period of 5ms, the temporal ordering of events cannot be observed.

The instruction and data PLB buses were also monitored for the same HTTP request. As shown in Figure 6.6, utilisation rates of the buses are different. The instruction PLB bus operates at the highest rate as instructions are required to perform processing. Data reads and writes depend on the processor instructions so the data PLB bus operates at a lower rate. The overall data bus utilisation is the sum of reads and writes within a time period.

The system was also tested with a trivial time server. The time server was stimulated with UDP time requests at regular intervals. As the time interval between packets decreased, the system ignored more requests. As shown in Figure 6.7 the system receives packets at regular intervals but the responses are sporadic. Viewing the PLB buses is also interesting as the system fails with the sustained packet reception. As shown in Figure 6.8, the system maintains a steady transaction rate on the PLB buses until the system crashes and the buses enter an oscillation state. The system fails due to the sustained rate of packets being faster than the system can respond. The system spawns a thread for every packet and the system also has a maximum number of threads that can be spawned at any given time. However, it is unclear why the buses begin to oscillate.

While instrumenting the advanced web server, there was a problem with the number of interrupts seen at the processor. It was observed that for every packet received a single interrupt was created at the interface to the interrupt controller. However, two interrupts were seen at the interface to the processor. Initially it was thought that the probes were incorrectly implemented. However, it soon became apparent that the cause for this problem was more complex. The interrupt controller expected a level high interrupt request whereas the processor expected a rising edge interrupt request. Initially, it was

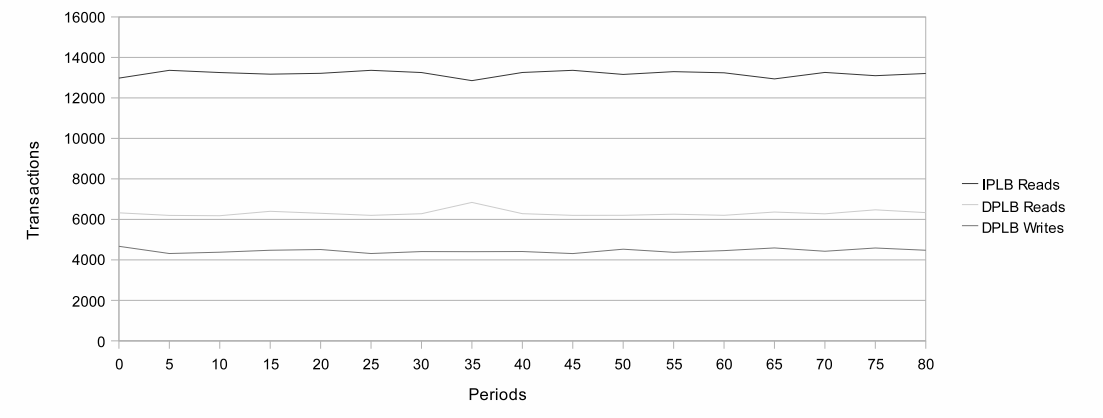

Figure 6.6: PLB bus utilisation

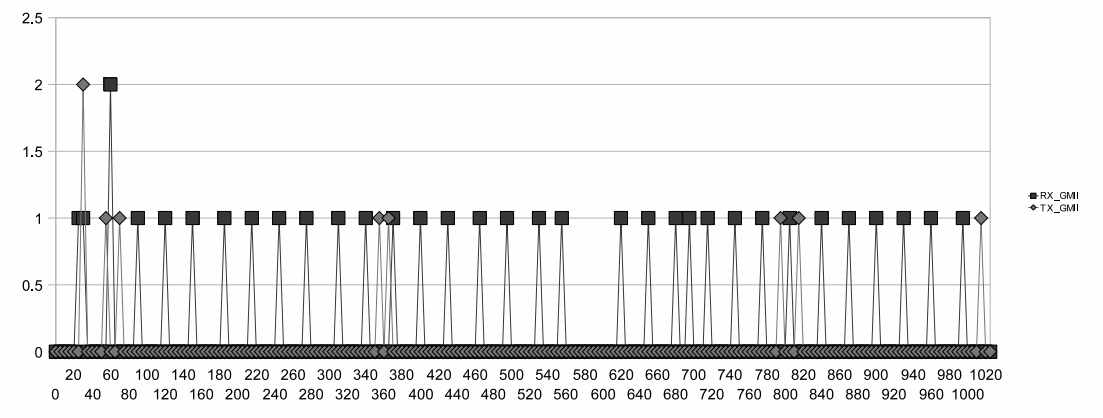

Figure 6.7: Time server response on overloaded conditions

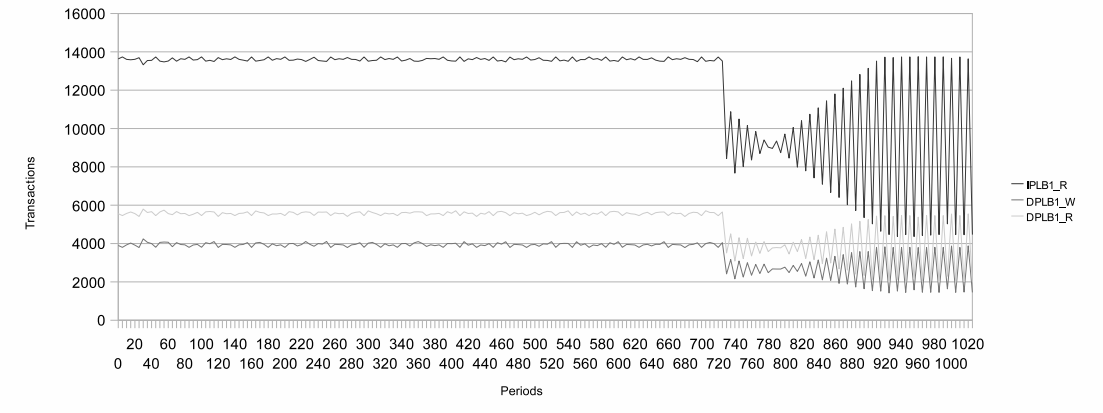

Figure 6.8: PLB bus utilisation on overloaded time server

thought that both were rising edge. This mismatch highlighted the difficulty in monitoring transactions as they can be dependant on other parts of the system. For example, in order to monitor the MPMC interrupts accurately, the probe would need to monitor the MPMC interrupt status register to determine when the interrupt was cleared. If the status register was cleared and the interrupt line remained level high then an additional interrupt could be inferred.

The observation of two interrupts was caused by the order in which the interrupt handler cleared the interrupt. First, the handler would clear the interrupt controller, which would dutifully release the interrupt request. However, the MPMC interrupt has not been cleared so the MPMC requests another interrupt as the signal is level high. Since the interrupt was masked, it waited until the current handler had completed, which also included clearing the MPMC interrupt. Following the completion of the interrupt handler, the interrupt controller raises an interrupt. As the MPMC interrupt has now been cleared the handler has no work to do, which terminates the handler and allows the processor to continue its previous execution. In order to correct this problem, the order for clearing the interrupts was altered so that the MPMC interrupt was cleared before the interrupt controller.

6.4 Hardware Firewall

The last case study is a prototype firewall, which parses, classifies and forwards packets entirely in the FPGA fabric. It has been implemented on a ML405 board and has been instrumented with each type of payload filter. As shown in Figure 6.9 the firewall uses a Gigabit Ethernet MAC to send and receive Ethernet frames, which contain IP packets. The Ethernet MAC is connected by LocalLink to a bus width converter, which changes the width of the datapath from 8 bits to 32 bits. This also allows the clock frequency to be reduced. Following width conversion, the packet is parsed to determine whether it is a configuration packet and to extract the 5-tuple. The rulebase is a content addressable memory, which contains the rules defining packet flows that are permitted for forwarding. Once a packet has been parsed, the rulebase is searched to determine whether it meets the criteria for forwarding. In the event that a configuration packet is received, the rulebase

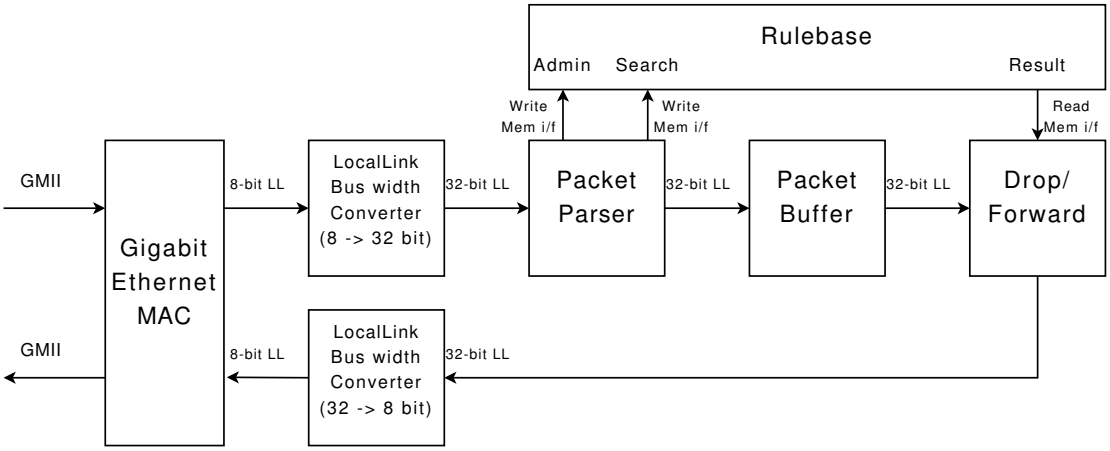

Figure 6.9: Firewall architecture.

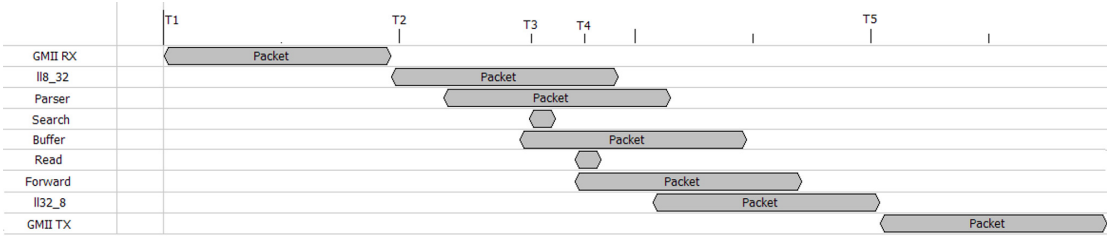

Figure 6.10: Transactions over firewall

is updated and the packet is dropped. After parsing, non-configuration packets are sent to a buffer that delays processing to allow the rulebase to search its rule table. Following the delay, the packet is either dropped or forwarded depending on the outcome of the rulebase search. Assuming that the packet is to be forwarded, it is then transferred to another bus width converter, which changes the width of the datapath from 32-bits to 8-bits. After which, the packet is then transmitted through the Ethernet MAC.

The IP blocks in this system allow monitoring in 11 locations, these are on the RX and TX GMII interfaces of the Ethernet MAC, the 6 LocalLink interfaces comprising the pipeline and the 3 memory interfaces. Figure 6.10 is an illustration of the information obtained from a working system, which clearly shows the packet flow through the firewall. Time T1 shows a packet entering the system. T2 shows the completion of packet reception on the MII interface and the start of packet transmission to the parser. T3 and T4 show the rulebase search and result transactions respectively. Finally, T5 shows the packet exiting the system. Figure 6.9 also demonstrates the pipeline effect, which can only be observed

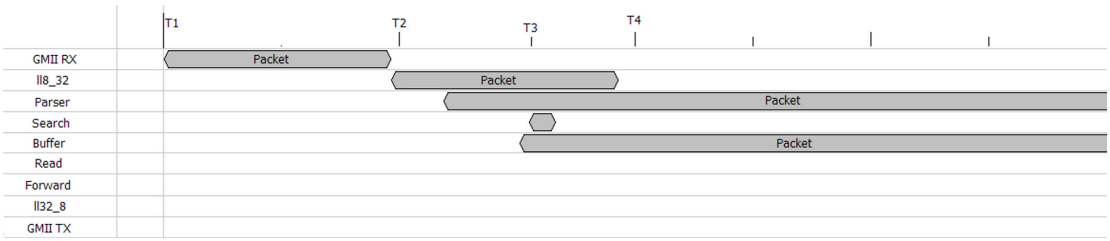

Figure 6.11: Resultant transactions over firewall when packet buffer has overflowed.

Table 6.6: The resource utilisation of the hardware firewall instrumented with different probe architectures

Firewall	Slices	LUTs	FFs
Uninstrumented	4291	3890	5216
Specialised Parallel	8133	10066	6503
Specialised Sequential	5254	5425	5802
Generic	6406	7235	5735
Generic Offsets	7050	8312	5813
Scalable (1 Match)	6166	6908	5779
Scalable (16 Matches)	8075	10556	5839

by recording the start and end times of transactions.

If the packet buffer was to overflow, as illustrated in Figure 6.11, there would be no transactions recorded further down the pipeline. At stages before the packet buffer, executing transactions would not terminate due to back pressure signalled by the overflowed packet buffer, which is propagated to all previous stages in the pipeline. This would clearly identify the buffer as the source of the problem and as such make the location of an error obvious. This system-level perspective allows the designer to easily infer and isolate erroneous components. Alternatively, if the search was not started then no read would be recorded and the packet transactions would stop at the forwarding IP block. This information, which is not shown by other tools, makes the location of an error more obvious. It can also allow the designer to infer and isolate erroneous components.

Table 6.6 shows the resource implications of different architectures. The parallel specialised probe requires the most resources, which is due to the spatial layout of the comparators. As interfaces in this system are at most 32-bits, this is an inefficient use of

Table 6.7: Resource utilisation of hardware firewall when instrumented with various monitoring tools

	Uninstrumented Design	Transaction Capture	ChipScope	Device Resources
Slices	4300	6587	7906	8544
Registers	5220	6503	9343	17088
LUTs	3908	7414	7070	17088
BRAM	9	10	31	68
BUFG	9	9	10	32

resources. The sequential version of the specialised probe is the most efficient. It takes advantage of the 8-bit LocalLink interfaces and GMII interfaces to produce smaller monitors. The memory interfaces retain the parallel matching and the 32-bit LocalLink interfaces require larger comparators but this method is suited to this design. The sequential probe provides the same facilities as the parallel version but requires fewer resources.

The generic probe requires more resources than the sequential specialised probe. The increase is greater than suggested by the individual probe counts, which may be due to the side-effect of placing probes on an interface. However, the increase in resource utilisation is significantly smaller than the parallel specialised probe. In any case, the generic probe can provide greater flexibility to allow the designer to observe other IP packets passing through the firewall such as ICMP.

The generic probe with offsets has higher resource requirements again. The resource requirements are close to those of the parallel specialised probe. Although the generic probe with offsets provides deeper insight into the packet, this may be unnecessary for this system as the firewall only operates on the 5-tuple.

The scalable probe is interesting as the resource requirements can be customised to the need of the designer at synthesis time. The ability to scale resources makes this an attractive option for this design. The combinator may also provide the necessary flexibility to debug the firewall rules accurately.

Table 6.7 shows the resource requirements of the firewall compared to ChipScope and

the available resources of the device. The firewall uses approximately 50% of the available slices. The instrumented design uses about 77% of the available slices to permit transaction level monitoring. The monitoring system uses a fixed IP packet parser with parallel matching, which is a resource intensive probe. The monitoring system uses a single Block RAM as a FIFO for uploading the results. The same design instrumented using ChipScope requires approximately 92% of the available slices and an additional 21 Block RAMs. As each probe has its own Block RAM for data capture it is difficult to place the monitoring circuitry close to the locations of interest. This requires additional circuitry and resources to route signals from the location of interest to the BRAMs, which suggests that a central collection mechanism may be better suited for monitoring system component interactions.

6.5 Summary

Three complex example systems have been instrumented to validate the approach to on-chip monitoring. These examples show that the system-level transaction system is significantly smaller than conventional tools. They further illustrate that transaction-level observations are useful as they provide a system-level understanding of the design by abstracting away the complexity of low-level signalling. In this way, they expose a different class of errors compared to traditional tools. Furthermore, the amount of data transmitted off-chip is significantly reduced.

Chapter 7

Automated System Instrumentation

Although many tools exist to detect errors before synthesis, errors still occur in the final implementation. Following implementation, errors can only be observed by on-chip monitoring, which requires the design to be instrumented. Typically, instrumentation circuitry is manually inserted, which is error prone and time consuming. This chapter presents an algorithm that automatically instruments systems created in high-level design environments. Automated instrumentation exploits the type system of high-level design environments to select and configure the probes for system-level monitoring. The alterations to the implementation flow are described and the interaction with monitoring software is presented.

7.1 Introduction

The FPGA implementation flow supports a variety of tools that validate and verify designs. For example, functional verification validates designs by exercising a set of executions, while static timing analysis formally verifies that designs meet timing constraints. An overview of validation and verification techniques frequently used in the FPGA design flow was presented in Chapter 3. A type system was also described in Chapter 4, which

formally verifies the interconnection of IP blocks. However, even with the plethora of tools available, errors may still occur in the final implementation.

In order to detect and correct implementation errors, the system needs to be observed on-chip. Several tools exist for monitoring FPGA-based systems but these tools typically require manual instantiation within a design. Furthermore, the majority of on-chip monitoring tools observe systems at the RTL level, which makes design instrumentation tedious, error prone and time consuming as signals need to be connected individually. To correctly insert a probe the designer might need to connect many signals, which compounds the problem. The architecture of a system-level transaction monitoring tool, as presented in Chapter 5, relates hardware interactions to the abstractions used by high-level design environments. However, instrumentation using the system-level monitoring tool is a manual process.

To address the limitation of manual instrumentation, the structural and transactional information in high-level design environments can be exploited to automatically instrument designs with the system-level transaction monitoring tool presented in Chapter 5. Low-level tools do not contain sufficient information to extend the implementation flow with an instrumentation algorithm for system-level monitoring. System Stitcher, a prototype high-level design environment for packet processing applications, has been extended to implement the instrumentation algorithm during its elaboration phase and to interact with the external host software of the system-level transaction monitoring tool. Consequently, the external host software can then configure and control the monitoring tool as part of the design environment.

The algorithm has been validated by application to the hardware firewall case study. System Stitcher has been used to automatically instrument the design from a Click description and invoke the external host software.

The use of automatic instrumentation creates an integrated development environment akin to those available for software development. Although the techniques for hardware development are different to software, an integrated development environment should improve designer productivity. Unfortunately, there are few integrated environments that combine hardware design, simulation and on-chip monitoring. The integration of multi-

Table 7.1: Simplified Click syntax of System Stitcher

A,B::=		Types
	Packet	Packet type
	Memory	Memory access type
	Interrupt	Interrupt type
	Input	Input type
	Output	Output type
	$A \times B$	Interface type
	$\{l_i : A_i^{i \in 1..n}\}$	Component type
D::=		Declarations
	$I :: A_1 \times B_1, \ldots, A_n \times B_n$	Interface declaration
C::=		Commands
	$I_1 \rightarrow I_2$	Interface connection
	$C_1;C_2$	Subsequent commands
	I[l]	Interface projection
I		Identifier

ple validation and verification techniques makes the contribution of the instrumentation algorithm significant to hardware design.

7.2 System Stitcher

System Stitcher is a prototype high-level design environment for creating packet processing applications, which has been extended to automatically instrument designs for on-chip run-time monitoring. It supports IP blocks described in a functional packet processing language called G, which describes the format of packets and the operations performed on those packets. System Stitcher is responsible for connecting IP blocks described in G and other languages to form a system. Click is used as the system description language and a simplified grammar is shown in Table 7.1.

System Stitcher defines a set of types, which can be either abstract or structural. An abstract type represents the physical and semantic properties of an interface without

Table 7.2: Type rules for Click syntax

$$\frac{}{\Gamma \vdash \diamond}\ (\textsc{Env}\ \phi) \qquad \frac{\Gamma \vdash A \qquad I \notin dom(\Gamma)}{\Gamma, I : A \vdash \diamond}\ (\textsc{Env I})$$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash Packet}\ (\textsc{Type Packet}) \qquad \frac{\Gamma \vdash \diamond}{\Gamma \vdash Memory}\ (\textsc{Type Memory}) \qquad \frac{\Gamma \vdash \diamond}{\Gamma \vdash Interrupt}\ (\textsc{Type Interrupt}) \qquad \frac{\Gamma \vdash \diamond}{\Gamma \vdash Input}\ (\textsc{Type Input})$$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash Output}\ (\textsc{Type Output}) \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \times B}\ (\textsc{Type Interface}) \qquad \frac{\Gamma \vdash A_1 \times B_1 \qquad \ldots \qquad \Gamma \vdash A_n \times B_n}{\Gamma \vdash \{l_i : A_i \times B_i^{i \in 1..n}\}}\ (\textsc{Type Component})$$

$$\frac{\Gamma \vdash \{l_i : A_i \times B_i^{i \in 1..n}\} \qquad A \in \{Packet, Memory, Interrupt\} \qquad B \in \{Input, Output\}}{\Gamma \vdash (I :: \{A_1 \times B_i, \ldots, A_n \times B_n\}) : \{l_i : A_i \times B_i^{i \in 1..n}\}}\ (\textsc{Decl Component})$$

$$\frac{\Gamma \vdash I_1 : A \times Input \qquad \Gamma \vdash I_2 : A \times Output}{\Gamma \vdash I_1 \rightarrow I_2}\ (\textsc{Comm Input Connect})$$

$$\frac{\Gamma \vdash I_1 : A \times Output \qquad \Gamma \vdash I_2 : A \times Input}{\Gamma \vdash I_1 \rightarrow I_2}\ (\textsc{Comm Output Connect}) \qquad \frac{\Gamma \vdash C_1 \qquad \Gamma \vdash C_2}{\Gamma \vdash C_1; C_2}\ (\textsc{Comm Subsequent})$$

$$\frac{\Gamma \vdash I_1 : \{l_i : A_i \times B_i^{i \in 1..n}\} \qquad j \in 1..n}{\Gamma \vdash I_1[l_j] : A_j \times B_j}\ (\textsc{Proj Interface})$$

specifying the exact composition. Although it might not be immediately obvious, the use of abstract types is useful for the semantic analysis of designs. Type checking can be performed on abstract types to detect errors in the interconnection of abstract IP blocks before the structural composition is known. The abstract types also negate the need for the designer to manually specify the structural composition of an interface as the high-level design environment can determine the structural composition during compilation. Structural types are created from abstract types before code generation as the composition of the interfaces need to be specified at this stage. However, the structural representation loses the semantic meaning of the signals.

The abstract types supported in System Stitcher are *Packet*, *Memory* and *Interrupt*, as shown in Table 7.2. The *Packet* type represents an interface that transfers packets over the Xilinx LocalLink [125] standard. The *Memory* type represents a wide range of interfaces including the common register, FIFO and BRAM. An *Interrupt* type is a single wire that is connected to components and indicates the occurrence of an event. Each type is also associated with a direction, which can be either *Input* or *Output*. The rules for connecting abstract interfaces are shown in Table 7.2, which state that *Packet*, *Memory* and *Interrupt* interfaces must be connected to an interface of the same type. They also state that an *Output* interface must be connected to an *Input* interface.

System Stitcher supports the specification of structural types to allow low-level descriptions to be used in the system. However, System Stitcher cannot infer any high-level information from structural interfaces and they are connected to other components in the system immediately before code generation.

System Stitcher follows the design flow shown in Figure 7.1, which loosely follows the architecture of a compiler. The initial operations are lexical analysis and parsing, which convert the Click description into an abstract system model. The abstract system model uses abstract types to represent interfaces where possible. Following creation, the abstract system model is subjected to semantic analysis, which includes type checking. Semantic analysis detects high-level errors in the system design such as connecting a packet interface to a memory access interface. It also detects other errors such as connecting multiple inputs to an interface without appropriate multiplexing circuitry.

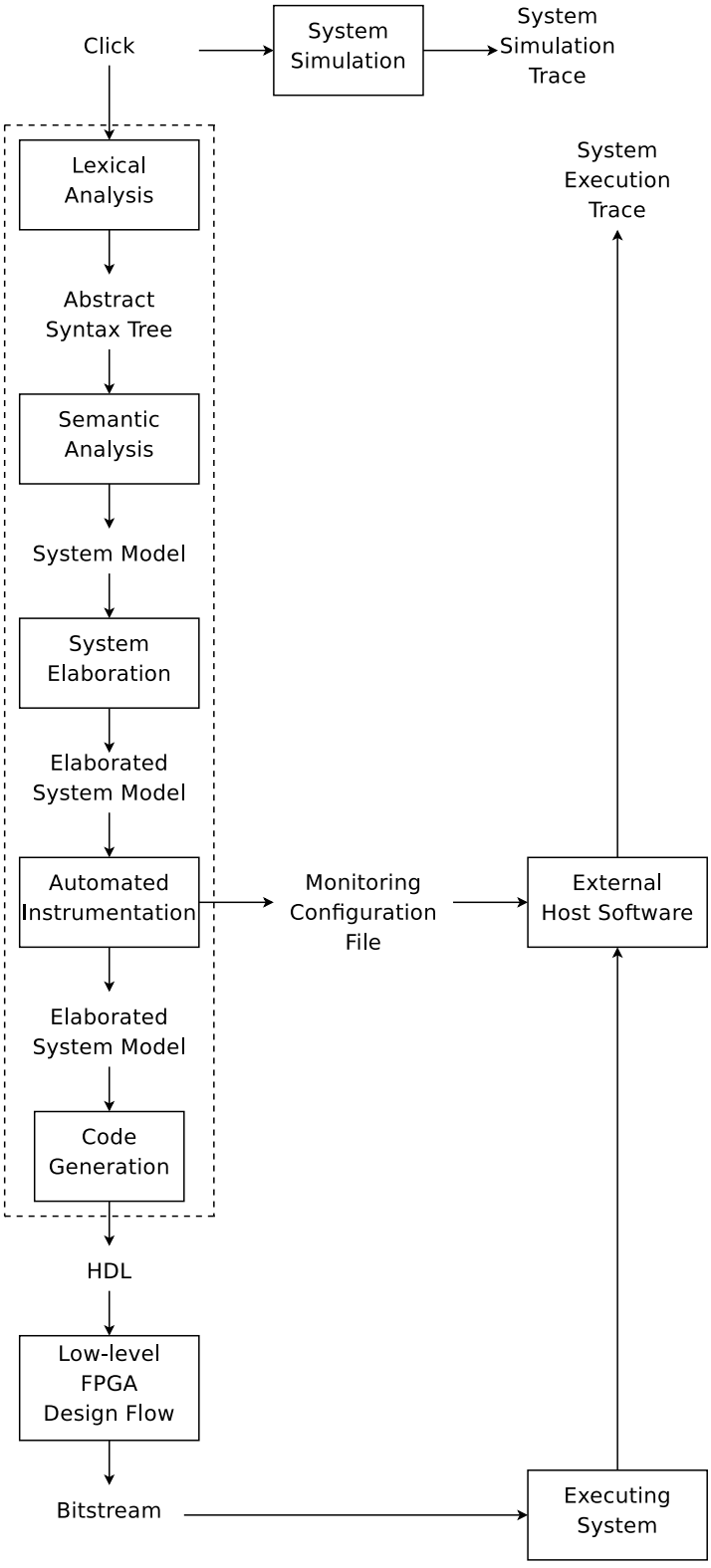

Figure 7.1: An overview of the design flow of System Stitcher including validation and verification points.

The system model is represented as a graph, which is comprised of a set of component instances and a set of component interconnections. The component instances are each given a type, where the type defines the set of interfaces present on the component. The interfaces also have a type as already described. Component interconnections are represented by a tuple comprising the source and destination of a connection, which only allows direct connections between components. The source and destination refer to specific interfaces on component instances.

Once the system model has passed semantic checking, it is then elaborated, which converts abstract interfaces to structural representations. Elaboration results in a structural system model that is used during code generation to produce a RTL representation of the system connections. The RTL description is written in a hardware description language such as VHDL.

Following code generation, the resultant RTL code is integrated with the netlists and RTL code of the IP blocks instantiated in the system. This set of files is then passed through the traditional FPGA design flow, which consists of synthesis, mapping, placement and routing. Once a placed and routed netlist is obtained, a bitstream is generated and used to program the target device.

7.3 On-chip Monitoring

The architecture of a system-level monitoring tool and the results of implementation have been presented in Chapter 5. The monitoring system is comprised of a set of probes connected directly to an event collector and software executing on a host computer. The set of probes monitor different interface types, where each probe is designed to monitor and interpret the signal transitions of a specific interface type. The probes are capable of filtering any payload and the functionality can be tailored to the needs of the system being monitored. The probes are designed to act as pass-through components so that they can be easily inserted between interfaces.

The collector is responsible for configuring the probes at run-time and communicating the results from the FPGA to an external host system. The collector provides a spatial

representation of events and records temporal information dependent on the processing provided by the collector. The information transmitted by the collector and class of errors detected were described in Chapter 6.

The external host system executes test software, which configures the monitoring system. It is also responsible for receiving, interpreting and presenting the results to the designer.

7.4 Automated Instrumentation

The implementation of the instrumentation algorithm is dependent on the data structures being manipulated and the point of execution in the design flow. The algorithm is executed as part of the elaboration phase of the high-level system compiler, as shown in Figure 7.1, for a number of reasons. At this stage, semantic checking will have been performed ensuring that the connections are valid. Performing automated instrumentation during elaboration also allows the algorithm to operate on the abstract model. The elaboration stage computes the low-level data for each interface, which is necessary to configure the monitors to match the interfaces being monitored. Instrumentation becomes infeasible later in the design flow as the transaction-level semantic information of the abstract system model is discarded following elaboration.

The instrumentation algorithm consists of two main phases. The first phase inserts the probes into the system model. The second phase inserts the collector and connects the outputs of the probes to the collector. As described in Figure 7.2, the algorithm iterates over the set of components in the system and their interfaces. Each interface is only allowed a single connection so only the output interfaces are instrumented, which avoids connecting multiple probes to a single connection. The type of each interface is determined from the abstract system model and the interfaces are evaluated to determine the structural properties. Following this, the abstract connection is elaborated as two structural connections. The output interface is connected to the input of the probe and the output of the probe is connected to the input interface of the original connection, which has the effect of inserting the monitor into the connection.

Once every connection has been instrumented, the algorithm must then insert a collector

```
1: for each component in system do
2:   for each output interface on component do
3:     if interface is connected then
4:       if interface is packet type then
5:         insert packet probe
6:       else if interface is memory type then
7:         insert memory probe
8:       end if
9:     end if
10:  end for
11: end for
12: insert collector
13: for each component in system do
14:   if component is probe then
15:     connect monitor to collector
16:     record monitor identifier
17:   end if
18: end for
```

Figure 7.2: Algorithm to instrument system with monitors.

into the system. Following insertion, the algorithm then iterates over the system model again and identifies the probes that have been inserted. During this iteration, each probe is connected to the collector, which completes the connections required for the monitoring system to operate. As the algorithm connects the monitors to the collectors, it can also compile a configuration file that relates the probe identifiers to the probe locations. Following creation, the configuration file can be passed to the actual test software that configures, monitors and interprets the device under test. The instrumentation algorithm can be generalised to any high-level environment and could be extended to support other monitoring tools.

To validate this approach, the proposed algorithm has been used to instrument a hardware firewall, which is shown in Figure 7.3. The firewall uses both packet and memory interface types as specified in the high-level environment. The first phase of the instrumentation

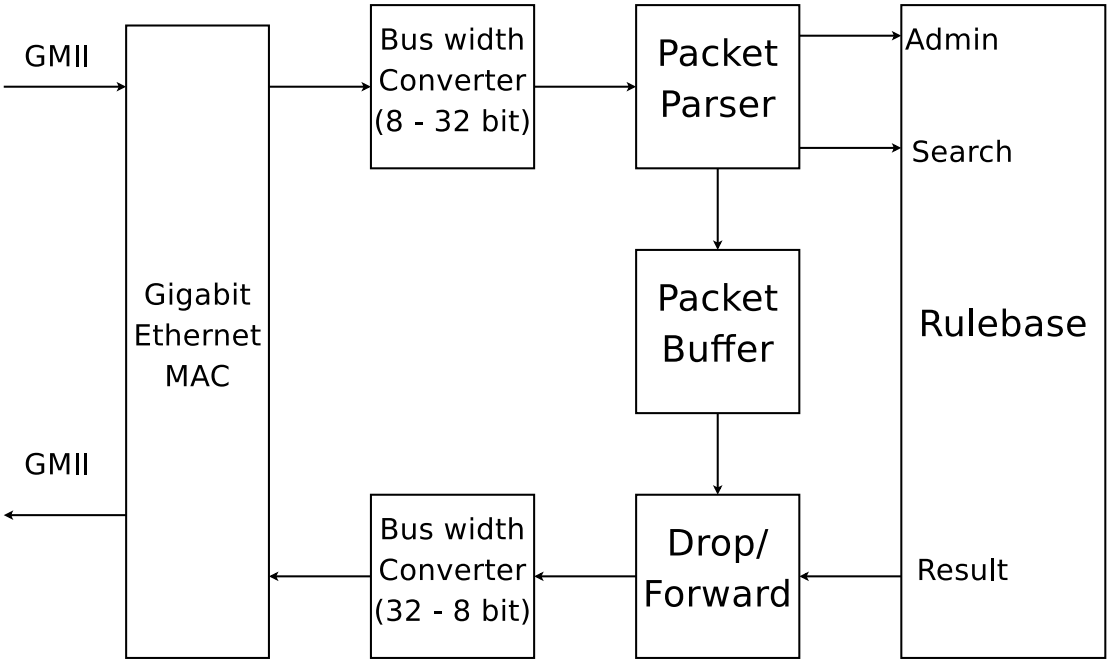

Figure 7.3: The architecture of the hardware firewall.

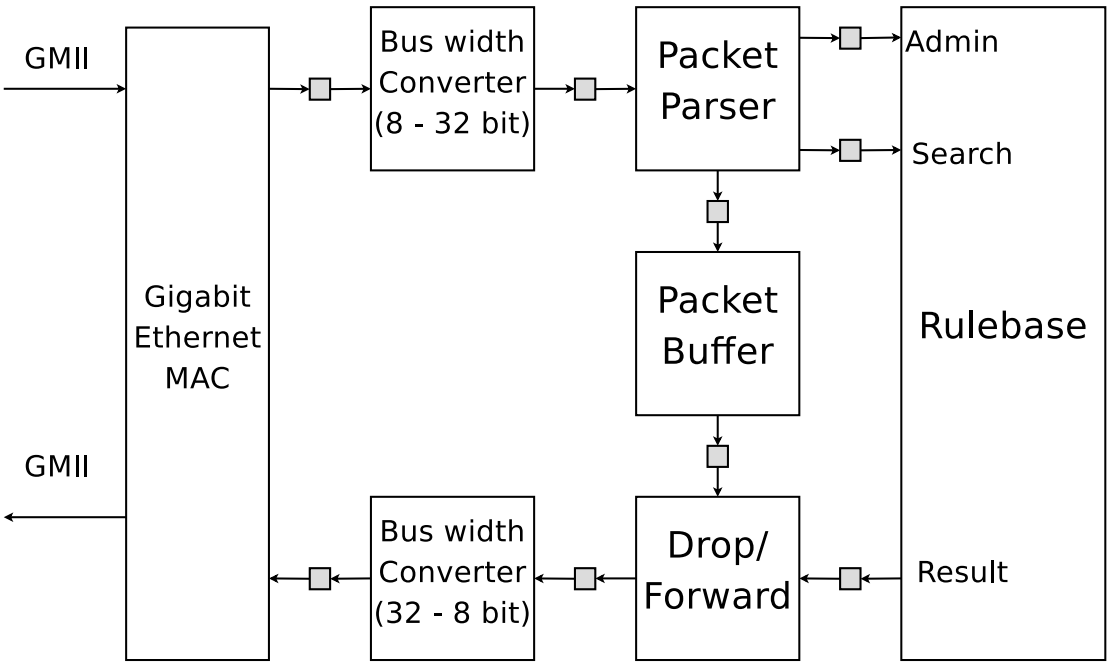

Figure 7.4: The architecture of the hardware firewall instrumented with transaction monitoring probes.

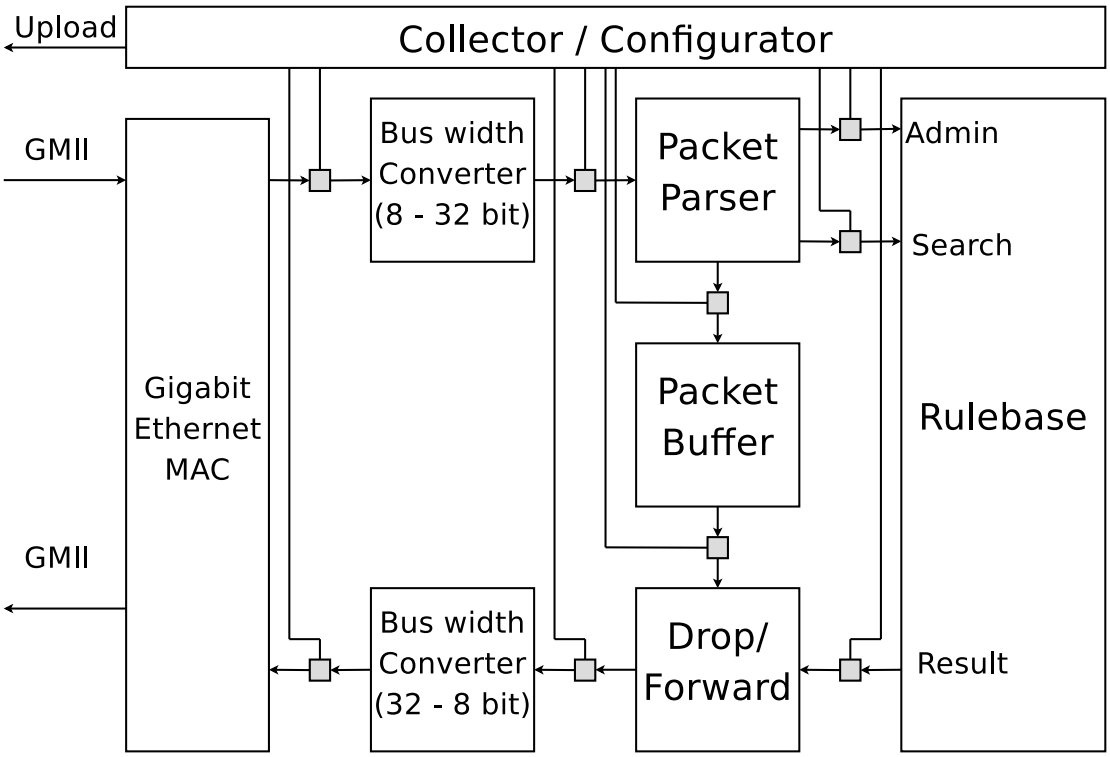

Figure 7.5: The architecture of the hardware firewall instrumented with system-level transaction monitoring circuitry.

Table 7.3: Resource utilisation of hardware firewall when instrumented using the automated instrumentation algorithm and generic probes

	Slices	Registers	LUTs	BRAM	Period (ns)
Uninstrumented	4300	5220	3908	9	19.440
Instrumented	6406	7235	5735	10	19.995

algorithm inserts probes into each connection in the system, while matching the probe with the correct interface type as shown in Figure 7.4. Following insertion of probes, the collector is inserted into the system and each probe is connected to the collector, as demonstrated in Figure 7.5. This then completes the instrumentation of the hardware firewall and at this stage, the monitoring mechanism can be used to obtain similar results to those presented in Chapter 6. The resource requirements of the hardware firewall are illustrated in Table 7.3, which presents the resource requirements of the hardware firewall without any instrumentation and the resource requirements when instrumented with generic probes.

Compared to a traditional tool flow using ChipScope, this algorithm saves time and is less error prone. The instrumentation algorithm negates the need for the designer to manually instrument the design as the configuration of the probes and test software is performed automatically. However, the system-level transaction circuitry might not be able to identify every error present in the system and a low-level monitoring tool might be required to examine the internal signals of IP blocks.

7.5 Summary

To improve the productivity of designers, high-level design environments and system-level transaction monitoring tools have been created. This chapter has demonstrated that the transaction-level semantics of interfaces specified in high-level design environments can be exploited to permit automatic instrumentation of designs. Automated instrumentation reduces the burden of probe insertion and configuration. Additionally, it reduces the time required to instrument designs compared to traditional design flows and it is less error prone.

Chapter 8

Future Work

This thesis has presented three techniques for validating and verifying the interconnection of IP blocks comprising packet processing systems. These techniques are methods of static verification, dynamic validation and error elimination through design automation. However, they do not prevent every error that can occur in a system. This chapter discusses potential avenues for future research and divides the discussion into three sections related to the techniques presented in this thesis. The list is not exhaustive but it does demonstrate the range of potential research avenues inspired by the work presented.

8.1 Static Verification

Presented in this thesis is a type system that statically verifies the connection of components within a system. The type system catches more errors and catches errors earlier compared to the traditional design flow. Consequently the designer is more productive as fewer synthesis executions are required and time is saved. As static verification improves designer productivity, the pursuit of further research into static verification techniques is likely to further reduce the number of errors that occur in the final implementation.

Further research into static verification of packet processing systems could explore improvements to the type system, connection dependencies, interface relationships and model checking. The type system could be extended to enforce more logical rules and catch

more errors. The dependencies between interconnected components could be examined to eliminate configuration and parametrisation errors. Information regarding component dependencies could potentially be used to automatically configure IP blocks. The relationship between interfaces in a single component could be explored to highlight data and control paths in a system. Finally, high-level functional descriptions of IP blocks could be used to create a model of the system that can be statically verified. Model checking can eliminate deadlock in a system and highlight unreachable states.

8.1.1 Extended Type System

As discussed in Chapter 4, the type system catches many errors that are currently not detected by the existing design flow. Although more errors are caught, the type system can be extended to increase the range of errors that are detected. One technique that can be applied is dependent typing [129]. Within software engineering, dependent types are used to statically check array bounds, which can remove the overhead of run-time dynamic checking.

Within Click, the application of dependent types could be used to enforce limits on the number of connections to an interface. With care, they could be used to limit the number of drivers in a connection before resolution to nets. For example, the type system currently verifies connections without regard for other statements in the Click description. The type system is only applied to the connection function, which connects components irrespective of the other connections in the system. As dependent typing can verify accesses to the boundaries of an array, the same principle can be applied to limit the number of connections made to a single interface. Some component interfaces restrict the number of connections that can be made to them. By way of example, direct media comprised of initiators and targets can only have one connection on each interface. This implies that both interfaces in a direct connection have only one shared connection. Assuming that a bus component has separate interfaces for individual masters and slaves, dependent typing can be used to enforce the connection of only one component to each interface. Alternatively, if a shared interface is used for each type of component then dependent typing can still be used to enforce the connection limits. Finally, fan-out of individual signals can be restricted by dependent typing which can improve system timing.

8.1.2 Interdependent Component Connections

IP blocks tend to be parameterisable so that they can be used in a variety of applications. For example, the width of a data bus can usually be specified as a parameter and the number of interfaces on a component might also be configurable. The multi-port memory controller (MPMC) is an example of a component with configurable interfaces [130]. In addition to specifying the width of data buses and the number of interfaces, the MPMC also allows the type of interfaces to be parameterised.

The type system is able to detect errors related to the connection of two interfaces. It is able to catch errors related to mismatched data bus widths, the number of interfaces and incompatible interface types. However, it is not able to detect system-level errors where properties of two separate connections are interdependent. The mesh fabric reference design [130] is an example of a complex system with interdependent component interfaces. This system creates multiple ingress interfaces to receive packets before scheduling and prioritising them for transmission to the packet switch. As the packets need to be multiplexed, limited buffering is available to temporarily store them. The dependency in the system relates to the number of ingress interfaces and the data bus width of another component. The number of ingress interfaces is defined by a parameter and the width of the data bus is defined by a calculation based on the number of ingress interfaces. The type system cannot catch errors related to this property as the interfaces between components are always correct.

The relationship between the number of ingress interfaces and the width of the data bus is a system-level property. The Click syntax will require extensions to capture such properties between connections. The properties could be specified either directly or indirectly. A direct specification will configure dependent components using a calculation based on a root parameter. These properties would be correct by construction during the elaboration of the system description. An indirect specification would assert the validity of a property by determining whether it exceeded a specified set of ranges. In this case the semantic analysis phase could be extended to verify these properties.

8.1.3 Interface Relationships

Both Brace and System Stitcher treat the IP blocks comprising the system as black boxes. No information about the internal structure or operation of the component is provided and only the interface types are available for verification. In order to improve static verification of the system, simple descriptions of the internal interface relationships could be provided with the IP blocks. For example, two packet interfaces could be related as the packets received over one interface are transmitted over the other after processing within the IP block.

The internal relationship between interfaces could be described using the Click language. The syntax would need to be extended to allow relationships between different interface types to be described. These relationships could highlight the datapaths in the system when the IP blocks are interconnected. Highlighting the datapaths could permit incomplete paths to be identified, which could be the result of an error.

8.1.4 Model Checking

Although System Stitcher treats the IP blocks comprising the system as black boxes, the descriptions of the IP blocks are available in a high-level design language. System Stitcher could be extended to perform static verification using those descriptions and the Click description of the system. Model checking would assume that IP blocks are correct and could identify potential deadlock, livelock and starvation conditions in the system as a result of the interconnection of those IP blocks. Model checking could also identify unreachable states and prove specific properties of the design.

8.2 Dynamic Validation

This thesis has presented a system-level transaction monitoring system that dynamically validates the connections between components in a design. The system-level transaction monitoring system abstracts the low-level details typically captured by traditional monitoring tools and provides greater visibility into the operation of the system. This

monitoring tool provides a foundation which can be built upon to further improve the monitoring of FPGA-based packet processing systems.

The system-level transaction monitoring tool could be extended to support co-simulation with a network simulator, system state inference, targeted data capture and automatic generation of the probe state machines. Co-simulation with a network simulator would allow the implemented system to be tested in conjunction with a simulation environment. System state inference uses the sequence of transactions to infer the state of the system at a given instant. Targeted data capture stores specific fields in packets to provide deeper insight into the system operation, while maintaining a low resource penalty. Finally, automatic generation of the probe state machines would permit rapid description of probes to monitor any interface type.

8.2.1 Co-simulation with a Network Simulator

Network simulators are frequently used to test protocols and their performance within a variety of network topologies. However, the abstractions used in these simulators can return results that differ from real world implementations. To address this limitation some network simulators permit co-simulation, which supports simulation using actual implementation of network stacks and the transmission of packets over real networks. Co-simulation alleviates the burden of modelling complex nodes and end points within a network, while returning more accurate results.

Brace and System Stitcher could be extended to exploit the functionality of co-simulation to provide testbeds for packet processing systems written in Click. The network simulator can be used to generate and record traffic that occurs in a variety of conditions and network topologies. The response of the Click packet processing system can also be observed using the system-level transaction monitoring tool. Using hardware-in-the-loop would provide an insight into the operation of the packet processing system within the network and present the response of the components comprising the system. Co-simulation may provide a faster and more accurate testing mechanism compared to the simulation techniques currently employed.

8.2.2 State Inference from Event Sequences

The system-level transaction monitoring tool only records transactions that occur on a connection and it does not record the state of any of the system components. The monitoring tool has already demonstrated that it can isolate and locate errors quickly. It can also detect errors that are unobserved by other low-level monitoring mechanisms.

Within complex systems, there may be many events which make it difficult for designers to manually determine whether the sequence of transactions is correct. The system-level monitoring tool could be extended to infer system state. For example, a TCP connection consists of several phases including setup, request, response and tear down. Inferring the state of the system from the sequence of transactions could assist the designer in understanding the design. This technique is likely to require high-level descriptions relating the sequences of events that occur between different interfaces. These descriptions would form an extension to the interface relationships described earlier in this chapter. The descriptions would describe the relationship between interfaces on an IP block and the events that result from an event on another interface. State inference is likely to be done off-chip to maintain the low resource utilisation of the transaction monitoring mechanisms and to reduce the bandwidth required to communicate events off-chip.

8.2.3 Probe State Machine Generation

The system-level transaction monitoring mechanism has a small resource requirement due to its architecture and the design of the probes and collectors. The probes consume a small amount of resources as their functionality is fixed during synthesis and they have little run-time configurability beyond their filtering capabilities. Unfortunately, this means that probes need to be created for each interface type that needs to be monitored in a system. While the architecture of the probes has been standardised to facilitate rapid development, the specification of the state machine is described at a low-level.

The specification of the interface operation can be described using a high-level language, which might also include annotations for describing packet formats. These descriptions would describe the signal transitions that indicate a transaction and the conditions under which a transfer is stalled. The descriptions could be similar to *modports* used in

SystemVerilog to describe buses and their transactions. Ideally, the interface descriptions would be included with an IP block and not require the system integrator to specify the supported transactions.

8.2.4 Data Capture from Probes

The system-level transaction monitoring tool does not capture any data from the packets that it observes. Instead, it filters the packets to determine which events to record. As the filtering mechanism supports masked matches, the designer frequently desires to know which value matched the filter. For example, if the data bus was configured with the wrong endian then the designer would be unable to tell from the information obtained by the filtering mechanism. In order to determine the exact value matched by the filter, data capture needs to be employed.

The key feature of the probes used in the system-level monitoring mechanism is the low resource utilisation. Any implementation of data capture needs to maintain the low resource utilisation, which can be achieved by capturing only the fields of the packet that are matched for filtering. The monitoring mechanism does not need to capture every bit within a field. It only needs to capture the bits that are not matched as these are the only values that are unknown. The mechanism for transferring captured data to the host computer is also vital in maintaining a low resource utilisation for the probes. One technique for transferring data from the probes is to employ a scan chain, which would keep the utilisation of the routing resources to a minimum. However, the data capture mechanism might not be able to record the fields of every packet that is observed depending on the clock and data rates. Alternatively, the captured data could be transferred to the collector with every recorded transaction but this would increase the complexity of the collector and its resource requirements.

8.3 Error Elimination Through Automation

This thesis has presented an algorithm for the automatic instrumentation of designs specified in an IP interconnection language. This technique uses the high-level information

present in the design environment to determine the types of interfaces being monitored. The current approach instruments every interface in the system to provide a system-level perspective. While monitoring every interface is desirable in most situations, it is not suitable for all designs.

Due to the expense of FPGA devices, designers will endeavour to use the smallest and cheapest device possible. Although the system-level transaction monitoring system has a low resource utilisation there are still instances where the monitoring mechanism might not be able to monitor every interface. This might occur as a result of exceeding the device's resources, high routing congestion and the inability to place the components to meet timing constraints.

To address these issues the instrumentation algorithm can be extended to provide the designer with greater control over the instrumentation process, while maintaining the benefits of automated instrumentation. First, the algorithm could be extended to determine the minimal set of monitoring points to observe the entire system. Combined with models of the components comprising the system, the instrumentation algorithm can insert probes in key locations and use simulation techniques to infer the transactions in locations that are not monitored. Second, the algorithm could be extended to support descriptions supplied by the designer specifying the locations of interest in the system. Such descriptions would only direct the instrumentation algorithm to the areas of interest and would not restrict the placement of probes to those locations.

8.3.1 Minimal Monitoring Points

On-chip monitoring systems must not exceed the available resources on a device and must minimise the impact on system timing and component displacement. As device utilisation tends to be high, the automated instrumentation algorithm could be extended to tailor the monitoring system to the needs of the target device and application. By combining on-chip monitoring with high-level simulation of IP blocks, it might be possible to reduce the number of probes required to accurately observe and test a system.

System Stitcher co-ordinates the synthesis of high-level descriptions to low-level implementations and it is capable of simulating the IP blocks comprising the system in a high-level

simulator. As the system-level transaction monitoring mechanism relates events to the operations described by System Stitcher, it may be possible to infer the events between components using high-level simulation. By monitoring events in key locations on-chip and simulating other connections, greater visibility into the system could be obtained and the resource requirements of the monitoring mechanism might be further reduced.

This technique would require automated instrumentation and would be a suitable extension for the algorithm described in Chapter 7. This extension would determine the minimal set of points that need to be monitored in order to observe the system accurately through simulation. The advantages of automated instrumentation would be maintained and would improve the resource utilisation of the monitoring system.

8.3.2 Debug Criteria

Designers frequently wish to change the location of interest when monitoring and debugging a system. For example, a designer may wish to observe the entire design to detect errors in the operation of the system. If an error is detected then the designer would desire to monitor the connections preceding the location of the error and potentially to monitor the connections at a lower level. The designer would like to determine the cause of the error so that it can be corrected. If the designer's hypothesis was incorrect then they would like to change the location of interest with ease.

Currently, monitoring tools do not provide a method of describing locations of interest. The designer must manually insert probes and determine the locations most suitable for debugging their application. The automated instrumentation algorithm could be extended to permit the use of monitoring criteria to express the interest of the designer. For example, the designer might be interested in the path of a specific packet type through the system or the control signals associated with interrupt processing. In the case of following a specific packet type through the system, the instrumentation algorithm might place probes on the connections where that packet type is expected to flow. For interrupt processing, the algorithm could instrument the component raising the interrupt, any interrupt controllers and the processor receiving the interrupt. Again, this method would need a description describing the relationship between the components.

8.4 Summary

This chapter has presented several avenues for future research based on the work presented in this thesis. Potential avenues for further work are centred around the three main themes of the thesis, which are static verification, dynamic validation and error elimination through automation. Within the domain of static verification, it is possible to extend the type system to catch more errors before synthesis. The dependencies between component connections could be modelled to catch system-level errors and the relationship between interfaces on a single component could be described to analyse data paths and other properties. By exploiting the high-level descriptions already available with System Stitcher, it is possible to perform model checking on systems to statically isolate unreachable states and deadlock conditions.

Potential directions for extending the work on dynamic validation include co-simulation, system state inference, automated probe generation and data capture. By co-simulating the implemented packet processing system with a network simulator, it is possible to monitor the system accurately and observe the impact on other components in the network. The IP blocks used in the system can also be supplemented with annotations that can describe the causality of events permitting the system state to be inferred and to automatically generate the state machine for probes to monitor an interface. The main limitation of the system-level transaction monitoring mechanism can be addressed by recording specific data from the probes. Data capture allows specific fields of packets to be observed and could be implemented in a manner that does not increase resource utilisation significantly.

Finally, error elimination through automation can be improved by extending the automated instrumentation algorithm to determine the minimal set of monitoring points required to observe a system and permitting a set of debug criteria to be described. By simulating specific components within the system it might be possible to observe a subset of the system connections and infer the events in the simulated locations. Simulation of components will further reduce the resource utilisation of the monitoring mechanism and improve visibility into the system under observation. Defining debug criteria allows the system to be instrumented in a manner that reflects the area of interest. The designer will

be interested in different parts of the system at different times, which can be supported efficiently by automatically observing the location and connections of interest.

Chapter 9

Conclusion

In this chapter, a summary of the work presented in this thesis is given and its contributions are highlighted. The outcomes of the work are also discussed in relation to the motivations and objectives of the project.

This thesis has examined the validation and verification of packet processing systems implemented in FPGA devices. In particular, it has explored the validation and verification of systems created in high-level design environments, which aim to reduce the designer productivity gap through the use of abstraction.

Traditional validation and verification techniques have focused on eliminating errors in the low-level design flow as systems were typically created using low-level descriptions. However, the insatiable demand for more computation in networks and the increasing density of modern devices has driven the need for high-level design environments. These high-level design environments improve productivity and reduce costs but the validation and verification capabilities are not as mature as the low-level implementation flow.

This thesis has presented three techniques for validating and verifying systems created in two high-level design environments called Brace and System Stitcher. The three techniques presented are:

1. Static verification by type checking connections specified in Click descriptions.

2. Dynamic validation by monitoring Click systems using transaction observations of component communications.

3. Automatic instrumentation of Click systems for observation using the system-level transaction monitoring tool.

9.1 Thesis Contributions

With regards to Brace and System Stitcher, this thesis has made several contributions to these research tools. It has also made several contributions to the validation and verification of packet processing systems in general. The contributions will be discussed as they relate to the three main themes of this thesis. First, the thesis contributed to the static verification of Click systems in Brace in the following ways:

- Definition of type system for packet processing systems on a FPGA.
- Implementation of the type system as part of the semantic analysis phase in Brace.
- Evaluation of the type system using two reference designs.

Second, this thesis has made several contributions to the dynamic validation of packet processing systems on a FPGA. The contributions to dynamic validation might also be applicable to other domains beyond packet processing. However, the following specific contributions have been made here:

- Specification and design of the architecture of an on-chip monitoring tool.
- Exploration of lightweight probe architectures with varying transaction interpreters and filter designs.
- Exploration of collector architectures which define two distinct collection mechanisms for profiling and event capture.
- Implementation of the various monitoring probes and collectors, which allow the resource requirements to be compared and contrasted.

- Implementation of host software for receiving results from the monitoring mechanism.
- Implementation of three example systems to validate the implementation of the monitoring system and demonstrate the class of errors that can be detected.

Finally, the work on automated instrumentation has made two contributions to the validation and verification of FPGA-based packet processing systems. Automated instrumentation is not widely supported by many monitoring tools but is an important addition to the thesis, which combines the static verification of the type system and the dynamic validation of the system-level transaction monitoring tool. This thesis has made the following contributions:

- Specification of an automated instrumentation algorithm.
- Implementation of the instrumentation algorithm in System Stitcher.

9.2 Thesis Summary

As the designer productivity gap widens, systems will be created at higher levels of abstraction. The use of abstraction is supported by high-level design environments, which can improve designer productivity. The main conclusion of this thesis is that validation and verification techniques, applied appropriately to high-level designs, can further improve the productivity of designers. This conclusion is supported by the results presented in this thesis and is supported by the summaries of each theme.

The type system is capable of proving the absence of connection errors in designs and catches more errors than the low-level FPGA implementation flow. The additional information embodied in the high-level design environment rigorously verifies properties that were previously understood by the designer. It has been demonstrated that more errors are caught by the type system but the class of errors is restricted to those within individual connections. The type system is not capable of verifying properties that are dependent on multiple connections. Finally, type checking catches errors before compilation and synthesis, which saves time and reduces synthesis iterations.

The system-level transaction monitoring mechanism has a small resource requirement compared to traditional low-level monitoring tools. It has been demonstrated that the monitoring tool can be used in a wider set of applications and that a different class of errors can be detected. The errors observed by the system-level transaction monitoring tool are also undetected by traditional low-level monitoring tools. Furthermore, the system-level transaction monitoring tool abstracts low-level signal transitions to events, which reduces the amount of data that needs to be transmitted off-chip. The data rates are also reduced by filtering payloads, which also provides information as to which packets have been transmitted over an interface. Finally, the restriction of monitoring communication between components improves the localisation of errors when combined with the filtering information.

The algorithm for performing automated instrumentation eliminates errors in the design flow, which result from incorrect manual insertion or configuration. While not demonstrated directly, this technique reduces the number of errors injected into the design and saves time.

In conclusion, this thesis has made several significant contributions to the validation and verification of FPGA-based packet processing systems. Methods of static verification, dynamic monitoring and error elimination through automation have improved the error detection of high-level design environments. These techniques can be applied in other settings beyond the research tools described in this thesis. Finally, the techniques have been shown to reduce the time required to complete a design and ultimately improve designer productivity.

Appendix A

List of Publications

The following articles have been published as a result of this research.

Poster. Paul E. McKechnie, Nathan A. Lindop and Wim A. Vanderbauwhede, "A type system for static typing of a domain-specific language", pp. 258, FPGA '08: Proceedings of 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays, Monterey, California, February, 2008

Paper. Wim Vanderbauwhede, Paul Mckechnie, Chidambaram Thirunavukkarasu, "The Gannet Service Manager: A Distributed Dataflow Controller for Heterogeneous Multicore SoCs", pp.301-308, 2008 NASA/ESA Conference on Adaptive Hardware and Systems, Noordwijk, The Netherlands, June, 2008

Paper. Paul E. McKechnie, Michaela Blott and Wim A. Vanderbauwhede, "Architectural Comparison of Instruments for Transaction Level Monitoring of FPGA-based Packet Processing Systems", pp. 175-182, in Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), Napa, California, April 2009

Paper. Paul E. McKechnie, Michaela Blott and Wim A. Vanderbauwhede, "Debugging FPGA-based Packet Processing Systems Through Transaction-level Communication-centric Monitoring", pp. 129-136, LCTES '09: Proceedings of the 2009 ACM SIG-

PLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES), Dublin, Ireland, June, 2009

Paper. Paul E. McKechnie, Michaela Blott and Wim A. Vanderbauwhede, "Automated Instrumentation of FPGA-based Systems for System-level Transaction Monitoring", pp. 168-171, in Proceedings of IEEE 2009 International Symposium on System-on-Chip (SoC), Tampere, Finland, October 2009

Patent Application. Paul E. McKechnie and Nathan A. Lindop, "Lightweight Probe and Data Collection within a Programmable Integrated Circuit"

Bibliography

[1] S. Brown and J. Rose, "Architecture of FPGAs and CPLDs: A tutorial," *IEEE Design & Test of Computers*, vol. 12, pp. 42–57, 1996.

[2] W. Arden, P. Cogez, M. Graef, H. Ishiuchi, T. Osada, J. Moon, J. Roh, M.-S. Liang, C. Diaz, P. Apte, B. Doering and P. Gargini, "International Technology Roadmap for Semiconductors: 2000 Update," International Technology Roadmap for Semiconductors, Tech. Rep., 2000. [Online]. Available: http://www.itrs.net

[3] M. Keating and P. Bricaud, *Reuse Methodology Manual for System-on-a-Chip Designs*. Massachusetts, USA: Kluwer Academic Publishers, 2002.

[4] *Platform Specification Format Reference Manual*, Xilinx, 2006. [Online]. Available: http://www.xilinx.com

[5] *System Generator for DSP User Guide*, Xilinx, February 2009. [Online]. Available: http://www.xilinx.com/

[6] A. Varga, "The OMNeT++ discrete event simulation system," in *Proceedings of the European Simulation Multiconference (ESM)*, June 2001, pp. 319–324.

[7] L. Breslau, D. Estrin, K. Fall, S. Floyd, H. Yu, J. Heidemann, A. Helmy, P. Huang, S. McCanne, K. Varadhan and Y. Xu, "Advances in network simulation," *Computer*, vol. 33, no. 5, pp. 59–67, 2000.

[8] N. Drago, F. Fummi and M. Poncino, "Modeling network embedded systems with NS-2 and SystemC," *Proceedings. ICCSC '02. 1st IEEE International Conference on Circuits and Systems for Communications.*, pp. 240–245, 2002.

[9] F. Fummi, S. Martini, G. Perbellini, M. Poncino, F. Ricciato and M. Turolla, "Heterogeneous co-simulation of networked embedded systems," *Proceedings of Design, Automation and Test in Europe Conference and Exhibition.*, vol. 3, p. 30168, 2004.

[10] S. Jansen and A. McGregor, "Simulation with real world network stacks," *Proceedings of the Winter Simulation Conference (WSC)*, pp. 2454–2463, 2005.

[11] M. Neufeld, A. Jain and D. Grunwald, "Nsclick: bridging network simulation and deployment," in *Proceedings of the 5th ACM international workshop on Modeling analysis and simulation of wireless and mobile systems*, 2002, pp. 74–81.

[12] R. Neogi, K. Lee, K. Panesar and J. Zhou, "Design and performance of a network-processor-based intelligent DSLAM," *IEEE Network*, pp. 56–62, July/August 2003.

[13] H. Yu-jie and L. Tao, "Building a robust packet control unit with network processors," in *Second International Conference on Embedded Software and Systems*, Dec. 2005, p. 229.

[14] J. DeHart, W. Richard, E. Spitznagel and D. Taylor, "The smart port card: An embedded unix processor architecture for network management and active networking," Washington University, Department of Computer Science, Technical Memorandum WUCS-TM-01-15, 2001. [Online]. Available: http://www.arl.wustl.edu/~det3/papers/spc_tech_report.pdf

[15] T. Chaney, J. A. Fingerhut, M. Flucke and J. S. Turner, "Design of a gigabit ATM switch," in *Proceedings of IEEE INFOCOM 97*, vol. 1, April 1997, pp. 2–11.

[16] W. Vanderbauwhede, P. McKechnie and C. Thirunavukkarasu, "The Gannet service manager: a distributed dataflow controller for heterogeneous multi-core SoCs," in *Proceedings of 2008 NASA/ESA Conference on Adaptive Hardware and Systems*, 2008, pp. 301–308.

[17] J. Luo, J. Pettit, M. Casado, N. McKeown and J. Lockwood, "Prototyping fast, simple, secure switches for Ethane," in *IEEE Symposium on High Performance Interconnects*, Stanford, California, USA, August 2007, pp. 73–82.

[18] J. Naous, D. Erickson, A. Covington, G. Appenzeller and N. McKeown, "Implementing an OpenFlow switch on the NetFPGA platform," in *Proceedings of the*

4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS'08), San Jose, California, USA, November 2008, pp. 1–9.

[19] J. W. Lockwood, N. Naufel, J. S. Turner and D. E. Taylor, "Reprogrammable network packet processing on the Field programmable Port Extender (FPX)," in *ACM International Symposium on Field Programmable Gate Arrays (FPGA)*, Monterey, CA, USA, Feb. 2001, pp. 87–93.

[20] J. W. Lockwood, C. Neely, C. Zuver, J. Moscola, S. Dharmapurikar and D. Lim, "An extensible, system-on-programmable-chip, content-aware Internet firewall," in *Field Programmable Logic and Applications (FPL)*, Lisbon, Portugal, Sep. 2003, pp. 859–868.

[21] J. W. Lockwood, J. Moscola, D. Reddick, M. Kulig and T. Brooks, "Application of hardware accelerated extensible network nodes for internet worm and virus protection," in *International Working Conference on Active Networks (IWAN)*, Kyoto, Japan, December 2003, pp. 44–57.

[22] J. W. Lockwood, J. Moscola, M. Kulig, D. Reddick and T. Brooks, "Internet worm and virus protection in dynamically reconfigurable hardware," in *Military and Aerospace Programmable Logic Device (MAPLD)*, Washington DC, Sep. 2003, p. E10.

[23] J. W. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman and J. Luo, "NetFPGA - an open platform for gigabit-rate network switching and routing," in *IEEE International Conference on Microelectronic Systems Education (MSE)*, San Diego, California, June 2007, pp. 160–161.

[24] P. Yiannacouras, J. Rose and J. G. Steffan, "The microarchitecture of FPGA-based soft processors," in *Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, 2005, p. 212.

[25] P. Buciak and J. Botwicz, "Lightweight multi-threaded network processor core in FPGA," in *IEEE Design and Diagnostics of Electronic Circuits and Systems*, 2007, pp. 1–5.

[26] D. Munteanu and C. Williamson, "An FPGA-based network processor for IP packet compression," *Proceedings of SCS SPECTS 2005*, pp. 599–608, July 2005.

[27] G. A. Covington, G. Gibb, J. Lockwood and N. McKeown, "A packet generator on the NetFPGA platform," in *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2009, pp. 235–238.

[28] D. E. Taylor, "Survey and taxonomy of packet classification techniques," *ACM Comput. Surv.*, vol. 37, no. 3, pp. 238–275, 2005.

[29] P. Gupta and N. McKeown, "Algorithms for packet classification," *IEEE Network*, pp. 24–32, March/April 2001.

[30] A. Nikitakis and I. Papaefstathiou, "A multi gigabit FPGA-based 5-tuple classification system," *Proceedings IEEE International Conference on Communications 2008*, pp. 2081–2085, May 2008.

[31] H. Song and J. W. Lockwood, "Efficient packet classification for network intrusion detection using FPGA," in *Proceedings of the 2005 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'05)*, Monterey, CA, Feb. 2005, pp. 238–245.

[32] D. E. Taylor, J. S. Turner, J. W. Lockwood, T. S. Sproull and D. B. Parlour, "Scalable IP lookup for Internet routers," *IEEE Journal on Selected Areas in Communications (JSAC)*, vol. 21, no. 4, pp. 522–534, May 2003.

[33] H. Le, W. Jiang and V. K. Prasanna, "A SRAM-based architecture for trie-based IP lookup using FPGA," in *FCCM '08: Proceedings of the 2008 16th International Symposium on Field-Programmable Custom Computing Machines*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 33–42.

[34] P.-C. Lin, Y.-D. Lin, T.-H. Lee and Y.-C. Lai, "Using string matching for deep packet inspection," *IEEE Computer*, vol. 41, no. 4, pp. 23–28, April 2008.

[35] J. Moscola, J. Lockwood, R. P. Loui and M. Pachos, "Implementation of a content-scanning module for an Internet firewall," in *Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa, California, April 2003, pp. 31–38.

[36] J. Moscola, M. Pachos, J. Lockwood and R. P. Loui, "FPsed: A streaming content search-and-replace module for an Internet firewall," in *Proceedings of 11th Symposium on High Performance Interconnects*, 2003, p. 122.

[37] D. V. Schuehler, J. Moscola and J. W. Lockwood, "Architecture for a hardware based, TCP/IP content scanning system," in *Hot Interconnects*, Stanford, CA, Aug. 2003, pp. 89–94.

[38] K. Mackenzie and A. Johnson, "Rapid synthesis of pattern classification circuits," *Proceedings of the 9th annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 285–286, 2001.

[39] A. Johnson and K. Mackenzie, "Pattern matching in reconfigurable logic for packet classification," in *Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*. ACM New York, NY, USA, November 2001, pp. 126–130.

[40] M. Attig and J. W. Lockwood, "SIFT: Snort Intrusion Filter for TCP," in *IEEE Symposium on High Performance Interconnects (HotI-13)*, Stanford, CA, August 2005, pp. 121–127.

[41] M. E. Attig and J. W. Lockwood, "A framework for rule processing in reconfigurable network systems," in *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Napa, CA, April 2005, pp. 225–234.

[42] M. Attig, S. Dharmapurikar and J. Lockwood, "Implementation results of Bloom filters for string matching," in *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE Computer Society Washington, DC, USA, 2004, pp. 322–323.

[43] C. Hylands, E. Lee, J. Liu, X. Liu, S. Neuendorffer, Y. Xiong, Y. Zhao and H. Zheng, "Overview of the Ptolemy project," University of California, Berkeley, Technical Memorandum, 2003. [Online]. Available: http://ptolemy.eecs.berkeley.edu/

[44] K. Hammond, "An abstract machine for resource-bounded computations in Hume," *Draft Proceedings of Implementation of Functional Languages*, 2003.

[45] K. Hammond, "The dynamic properties of Hume: a functionally-based concurrent language with bounded time and space behaviour," in *Implementation of Functional Languages : 12th International Workshop, IFL 2000 Aachen, Germany, September 4-7, 2000, Selected Papers*, ser. Lecture Notes in Computer Science, vol. 2011. Springer, 2001, p. 122.

[46] G. Michaelson and K. Hammond, "The Hume language definition and report, version 0.1," *Heriot-Watt University and University of St. Andrews*, July 2000.

[47] G. Michaelson, K. Hammond and J. Serot, "FSM-Hume: programming resource-limited systems using bounded automata," *Applied computing*, pp. 1455–1461, 2004.

[48] Y. Gottlieb and L. Peterson, "A comparative study of extensible routers," in *Open Architectures and Network Programming Proceedings.* IEEE, June 2002, pp. 51 – 62.

[49] D. Mosberger and L. L. Peterson, "Making paths explicit in the scout operating system," in *Proceedings of the Second USENIX Symposium on Operating System Design and Implementation (OSDI)*, Seattle, Washington, USA, October 1996, pp. 153–167.

[50] D. Decasper, Z. Dittia, G. Parulkar and B. Plattner, "Router plugins: a software architecture for next-generation routers," *IEEE/ACM Transactions on Networking*, vol. 8, pp. 2–15, 2000.

[51] E. Kohler, "The Click modular router," Ph.D. dissertation, MIT, November 2000.

[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, 2000.

[53] G. Brebner, C. Kulkarni and G. Schelle, "Mapping a domain specific language to a platform FPGA," in *Proc. 41st Design Automation Conference*, San Diego, California, USA, June 2004, pp. 924–927.

[54] L. Cardelli, *The Computer Science and Engineering Handbook.* CRC Press, 2004, ch. 97.

[55] B. C. Pierce, *Types and Programming Languages.* Massachusetts, USA: The MIT Press, 2002.

[56] *IEEE Standard VHDL Language Reference Manual*, IEEE Std 1076-1993 ed., The Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, NY 10017-2394, USA, 1993.

[57] *IEEE Standard Verilog Hardware Description Language*, IEEE Std 1364-2001 ed., The Institute of Electrical and Electronics Engineers, Inc., 3 Park Avenue, New York, NY 10016-5997, USA, 2001.

[58] *IEEE Standard SystemC Language Reference Manual*, IEEE Std 1666-2005 ed., The Institute of Electrical and Electronics Engineers, Inc., 3 Park Avenue, New York, NY 10016-5997, USA, 2005.

[59] *IEEE Standard for SystemVerilog - Unified Hardware Design, Specification, and Verification Language*, IEEE Std 1800-2005 ed., The Institute of Electrical and Electronics Engineers, Inc., 3 Park Avenue, New York, NY 10016-5997, USA, 2005.

[60] P. Bjesse, K. Claessen, M. Sheeran and S. Singh, "Lava: Hardware design in Haskell," *Proc. Int. Conf. Lisp Functional Programming*, pp. 174–184, 1998.

[61] J. T. O'Donnell, "Overview of Hydra: A concurrent language for synchronous digital circuit design," *International Parallel and Distributed Processing Symposium: IPDPS 2002 Workshops*, p. 249, April 2002.

[62] S. Singh, "System level specification in Lava," *Design, Automation and Test in Europe Conference and Exhibition*, pp. 370–375, 2003.

[63] R. A. Bergamaschi, S. Bhattacharya, R. Wagner, C. Fellenz, M. Muhlada, W. R. Lee, F. White and J.-M. Daveau, "Automating the design of SOCs using cores," *IEEE Design & Test Magazine*, pp. 32–45, 2001.

[64] V. Berman, "Standards: The P1685 IP-XACT IP metadata standard," *IEEE Design & Test*, vol. 23, no. 4, pp. 316–317, 2006.

[65] R. Ludwig, T. Hollstein, F. Schutz and M. Glasner, "Rapid prototyping of an integrated testing and debugging unit," in *Proceedings of 15th IEEE International Workshop on Rapid System Prototyping*, 2004, pp. 187–192.

[66] M.-C. Hsieh and C.-T. Huang, "An embedded infrastructure of debug and trace interface for the DSP platform," in *Proceedings of the 45th annual Design Automation Conference*, 2008, pp. 866–871.

[67] C.-F. Kao, I.-J. Huang and C.-H. Lin, "An embedded multi-resolution AMBA trace analyzer for microprocessor-based SoC integration," in *Proceedings of the 44th annual Design Automation Conference*, 2007, pp. 477–482.

[68] C.-F. Kao, C.-H. Lin and I.-J. Huang, "Configurable AMBA on-chip real-time signal tracer," in *Proceedings of the 2007 Asia and South Pacific Design Automation Conference*, 2007, pp. 114–115.

[69] B. R. Quinton and S. J. E. Wilton, "Post-silicon debug using programmable logic cores," *Proceedings 2005 IEEE International Conference on Field-Programmable Technology (FPT)*, pp. 241–247, 2005.

[70] E. de la Torre, M. Garcia, T. Riesgo, Y. Torroja and J. Uceda, "Non-intrusive debugging using the JTAG interface of FPGA-based prototypes," in *Proceedings of the 2002 IEEE International Symposium on Industrial Electronics*, vol. 2, 2002, pp. 666–671.

[71] *ChipScope Pro 10.1 Software and Cores User Guide*, Xilinx, March 2008. [Online]. Available: http://www.xilinx.com/ise/optional_prod/cspro.htm

[72] K. Arshak, E. Jafer and C. Ibala, "Testing FPGA based digital system using Xilinx ChipScope logic analyzer," in *29th International Spring Seminar on Electronics Technology*, May 2006, pp. 355–360.

[73] O. Oltu, P. L. Milea and A. Simion, "Testing of digital circuitry using Xilinx chipscope logic analyzer," in *Proc. International Semiconductor Conference (CAS)*, Sinaia, Romania, October 2005, pp. 471–474.

[74] A. Penttinen, R. Jastrzebski, R. Pollanen and O. Pyrhonen, "Run-time debugging and monitoring of FPGA circuits using embedded microprocessor," in *Proceedings 2006 IEEE Design and Diagnostics of Electronic Circuits and Systems*, 2006, pp. 147–148.

[75] P. S. Graham, "Logical hardware debuggers for FPGA-based systems," Ph.D. dissertation, Brigham Young University, 2001.

[76] B. Hutchings and B. Nelson, "Developing and debugging FPGA applications in hardware with JHDL," in *Conference record of the Asilomar Conference on Signals, Systems & Computers*, vol. 1, 1999, pp. 554–558.

[77] P. Graham, B. Hutchings and B. Nelson, "Improving the FPGA design process through determining and applying logical-to-physical mappings," in *Proceedings IEEE Symposium on Field-Programmable Custom Computing Machines*, 2000, p. 305.

[78] P. Graham, B. Nelson and B. Hutchings, "Instrumenting bitstreams for debugging FPGA circuits," in *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Rohnert Park, California, USA, April 2001, pp. 41–50.

[79] P. Graham, T. Wheeler, B. Nelson and B. Hutchings, "Using design-level scan to improve design," *Lecture Notes in Computer Science*, pp. 483–492, 2001.

[80] A. Donlin, P. Lysaght, B. Blodget and G. Troeger, "A virtual file system for dynamically reconfigurable FPGAs," in *Field Programmable Logic and Application*, 2004, pp. 1127–1129.

[81] D. Levi and S. A. Guccione, "Boardscope: A debug tool for reconfigurable systems," in *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, 1998, pp. 239–246.

[82] N. Bartzoudis and K. McDonald-Maier, "Online monitoring of FPGA-based coprocessing engines embedded in dependable workstations," in *13th IEEE International On-Line Testing Symposium*, 2007, pp. 79–84.

[83] A. M. Gharehbaghi, M. Babagoli and S. Hessabi, "Assertion-based debug infrastructure for SoC designs," in *Proceedings of the IEEE International Conference on Microelectronics (ICM)*, 2007, pp. 137–140.

[84] M. Straka, J. Tobola and Z. Kotasek, "Checker design for on-line testing of Xilinx FPGA communication protocols," in *Proceedings of the 22nd IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems*, 2007, pp. 152–160.

[85] K. Goossens, C. Ciordaş, T. Basten, A. Rădulescu and A. Boon, "Transaction monitoring in networks on chip: The on-chip run-time perspective," in *Proc. IEEE Symposium on Industrial Embedded Systems (SIES)*. Antibes Juan-Les-Pins, France: IEEE, October 2006, pp. 1–10.

[86] C. Ciordas, T. Basten, A. Radulescu, K. Goossens and J. V. Meerbergen, "An event-based monitoring service for networks on chip," in *ACM Transactions on Design Automation of Electronic Systems*, vol. 10, no. 4, October 2005, pp. 702–723.

[87] C. Ciordas, T. Basten, A. Radulescu and K. Goossens, "An event-based network-on-chip monitoring service," in *Proceedings of the 9th IEEE International High-level Design Validation and Test Workshop*, 2004, pp. 149–154.

[88] C. Ciordas, A. Hansson, K. Goossens and T. Basten, "A monitoring-aware network-on-chip design flow," *9th EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools (DSD)*, vol. 54, pp. 397–410, 2006.

[89] K. Goossens, B. Vermeulen, R. van Steeden and M. Bennebroek, "Transaction-based communication-centric debug," in *Proc. IEEE First International Symposium on Networks-on-Chip*, Princeton, NJ, USA, May 2007, pp. 95–106.

[90] B. Vermeulen, K. Goossens, R. van Steeden and M. Bennebroek, "Communication-centric SoC debug using transactions," in *Proceedings of the European Test Symposium (ETS)*, 2007, pp. 69–76.

[91] K. Goossens, P. Res and N. Eindhoven, "Formal methods for networks on chips," *Fifth International Conference on Application of Concurrency to System Design, ACSD.*, pp. 188–189, 2005.

[92] S. Tang and Q. Xu, "In-band cross-trigger event transmission for transaction-based debug," in *Proceedings of ACM conference on Design, Automation and Test in Europe*, 2008, pp. 414–419.

[93] S. Tang and Q. Xu, "A debug probe for concurrently debugging multiple embedded cores and inter-core transactions in NoC-based systems," in *Proceedings of the 2008 Conference on Asia and South Pacific Design Automation*, 2008, pp. 416–421.

[94] E. Roesler and B. Nelson, "Debug methods for hybrid CPU/FPGA systems," in *Proceedings 2002 IEEE International Conference on Field-Programmable Technology, (FPT).*, 2002, pp. 243–250.

[95] B. Yu and X. Zou, "The software/hardware co-debug environment with emulator," in *Proceedings of the 9th International Database Engineering & Application Symposium*, 2005, pp. 34–38.

[96] K. Camera, H. K.-H. So and R. W. Brodersen, "An integrated debugging environment for reprogrammble hardware systems," in *Proc. International Symposium on Automated Analysis-driven debugging (AADEBUG)*, Monterey, California, USA, September 2005, pp. 111–116.

[97] K. Camera and R. W. Brodersen, "An integrated debugging environment for FPGA computing platforms," in *International Conference on Field Programmable Logic and Applications (FPL)*, Heidelberg, Germany, September 2008, pp. 311–316.

[98] C. Hochberger and A. Weiss, "A new methodology for debugging and validation of soft cores," in *International Conference on Field Programmable Logic and Applications (FPL)*, 2008, pp. 551–554.

[99] J. G. Tong and M. A. S. Khalid, "Profiling CAD tools: A proposed classification," in *Proceedings of the 19th International Conference on Microelectronics*, 2007, pp. 253–256.

[100] S. L. Graham, P. B. Kessler and M. K. Mckusick, "Gprof: A call graph execution profiler," in *Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, vol. 17, no. 6. New York, NY, USA: ACM Press, 1982, pp. 120–126.

[101] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI)*, San Diego, California, USA, June 2007.

[102] *IA-32 Intel Architecture Software Developers Manual*, Intel Corporation, September 2009. [Online]. Available: http://www.intel.com/products/processor/manuals/

[103] L. Shannon and P. Chow, "Using reconfigurability to achieve real-time profiling for hardware/software codesign," in *Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays (FPGA)*, 2004, pp. 190–199.

[104] J. G. Tong and M. A. S. Khalid, "Profiling tools for FPGA-based embedded systems: Survey and quantitative comparison," in *Journal of Computers*, vol. 3, no. 6, 2008, pp. 1–14.

[105] R. Hough, P. Jones, S. Friedman, R. Chamberlain, J. Fritts, J. Lockwood and R. Cytron, "Cycle-accurate microarchitecture performance evaluation," in *Proceedings of Workshop on Introspective Architecture*, 2006.

[106] R. Hough, P. Krishnamurthy, R. D. Chamberlain, R. K. Cytron, J. Lockwood and J. Fritts, "Empirical performance assessment using soft-core processors on reconfigurable hardware," in *Proceedings of the 2007 workshop on Experimental Computer Science*, June 2007.

[107] S. Padmanabhan, P. Jones and D. V. Schuehler, "Extracting and improving microarchitecture performance on reconfigurable architectures," *Workshop on Compilers and Tools for Constrained Embedded Systems*, pp. 115–136, 2004.

[108] D. Nunes, M. Saldana and P. Chow, "A profiler for a heterogeneous multi-core multi-FPGA system," in *Proc. IEEE International Conference on Field-Programmable Technology (ICFPT)*, Taipei, Taiwan, December 2008, pp. 113–120.

[109] L. Shannon, "Simplifying system-on-chip design through architecture and system CAD tools," Ph.D. dissertation, University of Toronto, 2006.

[110] L. Shannon and P. Chow, "Leveraging reconfigurability in the hardware/software codesign process," in *Proceedings IEEE International Symposium on Field Programmable Logic and Applications (FPL)*, 2005, pp. 731–732.

[111] L. Shannon and P. Chow, "SIMPPL: An adaptable SoC framework using a programmable controller IP interface to facilitate design reuse," in *IEEE Transactions on Very Large Scale Integration (VLSI) systems*, vol. 15, no. 4, April 2007, pp. 377–390.

[112] L. Shannon and P. Chow, "Maximizing system performance: Using reconfigurability to monitor system communications," in *Proc. IEEE International Conference on Field-Programmable Technology (ICFPT)*, Brisbane, Australia, December 2004, pp. 231–238.

[113] W. Chen and L. Shannon, "An on-chip testbed that emulates runtime traffic and reduces design verification time for FPGA designs," in *Proc. IEEE International Conference on Field-Programmable Technology (ICFPT)*, Taipei, Taiwan, December 2008, pp. 361–364.

[114] *Platform Express Manual*, Mentor Graphics, 2006. [Online]. Available: http://www.mentor.com

[115] *Identify RTL Debugger*, Synopsys, Inc., August 2008. [Online]. Available: http://www.synplicity.com/products/identify/

[116] K. Araki, Z. Furukawa and J. Cheng, "A general framework for debugging," *IEEE Software*, vol. 8, no. 3, pp. 14–20, May 1991.

[117] D. Josephson, "The good, the bad, and the ugly of silicon debug," in *Proceedings of the 43rd annual Design Automation Conference*, 2006, pp. 3–6.

[118] *Introduction to the Quartus II Software*, Altera Corporation, 101 Innovation Drive, San Jose, California, USA, 2009. [Online]. Available: http://www.altera.com/literature/manual/intro_to_quartus2.pdf

[119] *ISE In-Depth Tutorial*, Xilinx Inc., 2100 Logic Drive, San Jose, California, USA, June 2009. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/ise11tut.pdf

[120] *Libero IDE v9.0 Users Guide*, Actel Corporation, Mountain View, California, USA, February 2010. [Online]. Available: http://www.actel.com/documents/libero_ug.pdf

[121] *Advanced Verification Methodology*, Mentor Graphics. [Online]. Available: http://www.mentor.com

[122] *Open Verification Methodology*, Cadence. [Online]. Available: http://www.cadence.com

[123] *SystemC Modeling, Synthesis, and Verification in Catapult C*, Mentor Graphics Corporation, 8005 SW Boeckman Road, Wilsonville, Oregon, USA. [Online]. Available: http://www.mentor.com/products/esl/techpubs/

[124] D. Pursley and B. Cline, "A practical approach to hardware and software soc tradeoffs using high-level synthesis for architectural exploration," in *Proceedings of the GSPx Conference*. Forte Design Systems, April 2003. [Online]. Available: http://www.forteds.com/products/paper_0207_silverbullet.pdf

[125] *LocalLink Interface Specification*, Xilinx, 2005.

[126] Xilinx, "Ethernet cores hardware demonstration platform," *Xilinx Application Note 443*, 2005. [Online]. Available: http://www.xilinx.com/support/documentation/application_notes/xapp443.pdf

[127] J. Lo, "An Ethernet-to-MFRD traffic groomer," *Xilinx Application Note*, no. 541, 2006. [Online]. Available: http://www.xilinx.com/support/documentation/application_notes/xapp541.pdf

[128] U. Lamping, R. Sharpe and E. Warnicke, *Wireshark User's Guide*, Wireshark Foundation, 2008. [Online]. Available: http://www.wireshark.org/

[129] B. C. Pierce, *Advanced Topics in Types and Programming Languages*. Massachusetts, USA: The MIT Press, 2005.

[130] Xilinx, "Multi-port memory controller," *XAPP*, June 2007. [Online]. Available: http://www.xilinx.com/esp/wired/optical/xlnx_net/mpmc.htm