Pandora:
Facilitating IP Development for Hardware Specialization

Michael K. Papamichael

CMU-CS-15-121
August 2015

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
James C. Hoe, Chair
Ken Mai
Todd Mowry
Onur Mutlu
Mark Horowitz, Stanford University

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

This research was sponsored by equipment and tool donations from Xilinx and Bluespec, an Intel PhD Fellowship, and the National Science Foundation under grants CCF-0811702 and CCF-1012851.

Keywords: Computer Architecture, Network-on-Chip, NoC, FPGA, ASIC, Reconfigurable Hardware, Hardware Specialization, Hardware Acceleration, IP Block, IP Development, IP Optimization, Design Space Exploration, Hardware Instrumentation, Runtime Monitoring

to Azra

Abstract

In an effort to continue increasing performance in the power-constrained setting of the post-Dennard era, there is growing interest in hardware specialization. However, a major obstacle to the more widespread use of hardware acceleration is the level of difficulty in hardware design today. Despite the increased availability of rich IP libraries and even IP generators that span a wide range of application domains, developing hardware today is limited to experts, takes more time and is more expensive than ever.

This thesis presents the Pandora IP development paradigm that facilitates hardware development and specialization by extending the concept of generator-based IPs. Pandora encapsulates the IP author's expertise and domain knowledge to offer supporting infrastructure and assist the users' interactions with the IP. In contrast to existing IPs and IP generators that only capture the structural and microarchitectural view of a design, Pandora argues for augmenting IPs with: (1) detailed IP design space characterization to help the user understand the effects of parameter choices with respect to hardware implementation and IP-specific metrics, (2) application-level goal-oriented parameterization that is meaningful to the IP user and automatically sets low-level structural parameters to achieve the desired design optimizations, and (3) purpose-built domain-aware simulation-time and run-time monitoring mechanisms to assist functional and performance debugging.

To highlight the benefits of hardware specialization and demonstrate the key principles of the Pandora IP development paradigm, this thesis presents our research efforts on: (1) CONNECT, a flexible Network-on-Chip (NoC) IP generator that embodies the Pandora principles and is actively used by hundreds of researchers around the

world, (2) DELPHI, a framework for fast easy IP characterization that facilitates mapping the design space of arbitrary RTL-based IPs, (3) Nautilus, an IP optimization engine that demonstrates how incorporating IP author knowledge in genetic algorithms can enable very fast—orders of magnitude faster than conventional methods—high-level goal-oriented IP optimization, and (4) IRIS, an instrumentation and introspection framework that combines hardware monitors with software-based post-processing and visualization engines to accelerate debugging of complex IPs and enable higher system-level visibility.

Acknowledgments

First and foremost, I would like to thank my advisor James Hoe for supporting, guiding, and encouraging me throughout my PhD, always being available to help and offer advice, giving me freedom to explore and grow as a researcher, and maintaining a happy, friendly, comfortable environment. James taught me how to think, ask the right questions, identify truly interesting research problems, and perhaps more importantly, how to become a better, happier, more complete human being. Thank you.

I thank my committee members, Professors Ken Mai, Onur Mutlu, and Todd Mowry for their support, feedback, and for always being available to meet with me and discuss about research. I thank my external committee member, Professor Mark Horowitz for his advice and feedback that helped shape and improve this thesis. I owe special thanks to Professor Babak Falsafi for offering his support and advice throughout the years even all the way from Switzerland. I thank Professor Derek Chiou and my mentors at Intel Labs, Hong Wang and Graham Schelle, for their support. I thank Professor Manolis Katevenis, who introduced me to computer architecture and encouraged me to pursue a PhD. I also thank the Computer Science Department staff, especially Deb Cavlovich, for working tirelessly to ensure student happiness.

I thank all of my friends and collaborators at Carnegie Mellon. I am grateful to my friend, office-mate and collaborator Eric Chung for mentoring me during my first years as a graduate student and for the lengthy research discussions (and ping-pong sessions) that often ran into the early morning. I thank my CMU collaborators Yoongu Kim, Peter Milder, Gabriel Weisz, Eriko Nurvitadhi, Cagla Cakir, Professor Mor Harchol-Balter, as well as my external collaborators, Chen Sun, Chia-Hsin Owen Chen, and Professors Li-Shiuan Peh and Vladimir Stojanovic. I thank my

friends and fellow CMU graduate students Nikos Hardavellas, Evangelos Vlachos, Kiki Levanti, Theodoros Strigkos, Ippokratis Pandis, Brian Gold, Mike Ferdman, Jared Smolens, Peter Klemperer, Berkin Akin, Yu Wang, Marie Nguyen, Malcolm Taylor, Felix Hutchinson, Jangwoo Kim, Chris Fallin, Chris Craik, Vivek Sheshadri, Lavanya Subramanian, Rachata Ausavarungnirun, Gennady Pekhimenko, Justin Meza, Samira Khan, Stephen Somogyi, Michelle Goodstein, Burak Erbagci, Kaushik Vaidyanathan, Mark McCartney for their support, advice, and for making grad school fun. Finally, I want to thank my friends in Greece, as well as all the new friends I made in Pittsburgh for all the fun times and for keeping me company during work and the occasional all-nighter.

I thank my family, my father Konstantinos, my mother Irene, and my wife Marina for their unconditional love, support and constant encouragement. I love you very much.

This work was supported in part by hardware and tool donations from Xilinx and Bluespec, NSF grants CCF-0811702 and CCF-1012851, and an Intel PhD Fellowship.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

We are currently witnessing fundamental and disruptive changes in the semiconductor industry. On the one hand, rapidly growing transistor counts driven by Moore's Law [74], coupled with recent technology advances and trends, such as die stacking and the increased presence of on-die reconfigurable logic, are enabling the development of massive, diverse Systems-on-Chip (SoCs) comprising of tens or even hundreds of interacting Intellectual Property (IP) blocks with unique and demanding communication requirements. On the other hand, our inability to further scale supply voltage is leading to the breakdown of classical CMOS scaling as described by Dennard [36]. As a result, after decades of continuous growth, transistor efficiency, which has been the primary driving force behind performance improvements in general-purpose computing, has now reached a stalemate [41, 106].

In an effort to continue increasing performance in the power-constrained setting of the post-Dennard era, there is a growing interest in hardware specialization [89]. While hardware specialization is key to power-efficient computing, hardware development today is notoriously hard. As a result, even though there is great demand for mapping applications to specialized hardware to

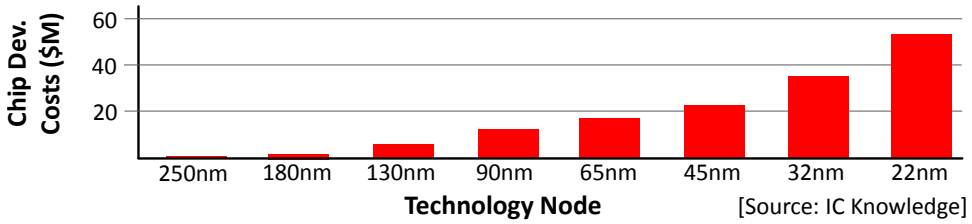


Figure 1.1: Chip Development Cost by Process Node.

achieve higher performance and power efficiency, and despite the availability of an ever-growing number of rich Intellectual Property (IP) block catalogs, hardware development today is not only limited to experts, but takes more time, requires larger design teams, and is more expensive than ever (Figure 1.1).

1.1 The Rise of Hardware Complexity

An alarming trend of increasing complexity in hardware design was first recognized about fifteen years ago and was labeled the "design productivity gap" (Figure 1.2), which pertains to the difference between the transistors available on a single die and the number of transistors a designer is able to effectively design for. Efforts to bridge this gap sparked research efforts in multiple aspects of hardware design, including high-level synthesis techniques [44, 65], validation tools, more powerful hardware description languages [20, 102] and frameworks [95] that enable the development of flexible IP generators and new hardware design methodologies, such as platform-based design [53].

Of particular interest to this work is the increased use of IP blocks (which refer to pre-made, pre-validated, reusable packaged units of hardware design). With IP reuse, instead of designing every component in a chip from scratch, designers build entire chips or portions thereof by leveraging existing IP blocks, often developed by third parties. This practice greatly reduces the development

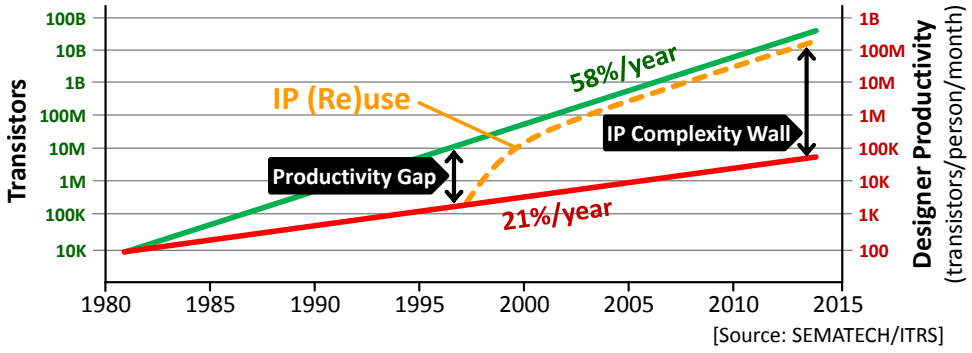


Figure 1.2: The Design Productivity Gap and IP Complexity Wall.

time and cost of individual submodules within a larger chip. Sure enough, IP reuse is ubiquitous today and has become an indispensable part of modern chip design. Over the years, IP blocks, which started as simple design instances, have now evolved to sophisticated, highly-parameterized and complex domain-specific IP generators responsible for multi-million-transistor blocks in a chip.

The IP Complexity Wall. Despite an enormous rise in scale, complexity, and specialization, the way IP blocks are developed and used has not fundamentally changed since the introduction of modern Hardware Description Languages (HDLs) and the proliferation of ASIC-based design flows more than two decades ago. The result is a "complexity explosion" as designers build chips as "fragile" collections of many complex IP blocks, each with its own set of cryptic (for the non-domain-expert) low-level knobs, which can often be traced back to a crude hardware schematic or specification document. Today, we are facing a new encounter with the "design productivity gap" at a different scale—not at the level of the transistor, but instead at the level of the IP block. Current hardware design methodologies are struggling to keep up with the complexity involved in configuring, tuning, integrating, and validating the multiple interacting IP blocks within a modern chip. Consequently, the complexity and the associated development time, cost, and manpower required to build a chip today continue to increase prohibitively.

1.2 Pandora: Facilitating IP Development for Hardware Specialization

The focus of this thesis is on a novel knowledge-encapsulating IP development paradigm, called Pandora[1], that is aligned with existing efforts to tackle design complexity and aims at retaining the benefits of highly parameterized IP design and generation to facilitate hardware specialization, while at the same time addressing the associated complexity explosion. The Pandora paradigm is motivated by the highly specialized nature of modern IP blocks and argues for tailoring them with supporting functionalities according to their specific application domain. Pandora's ultimate goal is to lower the barrier-to-entry for building specialized hardware accelerators and allowing application-experts to more easily and efficiently realize their ideas in hardware.

In Pandora, IP blocks not only capture the microarchitectural or low-level structural view of a design, but also encapsulate domain-specific infrastructure and additional dimensions of knowledge that the IP author has to offer. The Pandora paradigm marks a departure from the current status quo in hardware design by combining a set of key ideas and principles that empower IP authors and enhance how users interact with hardware IPs. The three defining aspects of the Pandora IP paradigm can be summarized as:

- Facilitate fast detailed IP design space characterization to help the user understand the effects of parameter choices and allow for obtaining quick estimates with respect to hardware implementation and IP-specific metrics.

- Provide application-level goal-oriented domain-specific optimization interfaces that are mean-

[1]The name Pandora is inspired from Greek mythology and has a dual meaning. The first meaning relates to how Pandora was created through unique gifts from each god, which resembles how the proposed design paradigm encapsulates rich domain-expert ("gods") knowledge to support complexity-reducing interfaces, mechanisms, and tools ("gifts"). The second meaning pertains to Pandora's box, which kept sealed all of the evils of the world, similar to how the proposed hardware design paradigm tries to hide or restrain complexity within the IP and avoid exposing the user to the "evils" or complexities of hardware design.

ingful to the IP user and automatically set low-level structural parameters to achieve the desired design optimizations.

- Support purpose-built domain-aware simulation-time and run-time instrumentation and introspection mechanisms that gather, present, and analyze the gathered data to identify or even diagnose higher-order correctness and performance issues.

In addition to keeping complexity under control and boosting productivity, Pandora also dramatically reduces the combined total effort because work that would otherwise be repeated by each IP user, is now only performed once by the authors of the IP.

As part of my work on Pandora, I have developed CONNECT, DELPHI, Nautilus, and IRIS. CONNECT [68, 81, 82] is a flexible Network-on-Chip (NoC) IP generator I developed and publicly released, that is actively used by hundreds of researchers around the world, and has both served as an inspiration as well as demonstration vehicle for Pandora. DELPHI [84] is a framework for performing fast and efficient IP characterization (power, area, frequency) across multiple technology nodes. Nautilus [83] is a high-level goal-oriented IP optimization engine that uses modified genetic algorithms that incorporate IP author knowledge to perform fast guided design space search. IRIS is an IP instrumentation framework that facilitates system-level debugging, monitoring, and analysis. In the context of this thesis, the CONNECT NoC IP generator highlights the benefits of specialization and serves as a driving example for Pandora, while DELPHI, IRIS, and Nautilus serve as demonstration vehicles for Pandora's key principles.

1.3 Thesis Contributions

This thesis makes the following contributions:

- Proposes the Pandora IP development paradigm and demonstrate many of its key principles

through the development of CONNECT, DELPHI, Nautilus, and IRIS.

- Presents the CONNECT Network-on-Chip IP Generator, which is actively used by hundreds of researchers around the world.
- Demonstrates the benefits of hardware specialization through an investigation of interconnect specialization and tuning in the context of FPGAs and CoRAM applications.
- Presents DELPHI, a framework for fast, easy, efficient RTL-based characterization of IP designs.
- Presents Nautilus, which demonstrates how IP author knowledge can be used to vastly accelerate hardware IP optimization and design space search using guided genetic algorithms.
- Proposes and develops IRIS, a flexible systematic instrumentation framework that allows incorporating IP author knowledge for efficient hardware debugging and system-level monitoring.
- Demonstrates how Pandora can drastically enhance how IP users interact with IP generation frameworks by developing and showcasing a proof-of-concept Pandora-powered version of the CONNECT Network-on-Chip IP generator that incorporates the key ideas and research artifacts presented in this thesis.

1.4 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 describes the CONNECT Network-on-Chip IP generator that was developed and extended in support of this thesis, and demonstrates how CONNECT allows specializing the generated NoC IP instances for the underlying hardware, as well as the traffic characteristics of a given application. Chapter 3 presents the Pandora IP development paradigm and elaborates on the Pandora principles, which are demonstrated

through the frameworks described in the next three chapters. In particular, Chapter 4 presents the DELPHI framework for fast IP characterization. Chapter 5 presents the Nautilus IP optimization framework and Chapter 6 presents the IRIS instrumentation and introspection framework. Chapter 7 describes how the CONNECT Network-on-Chip IP generator was extended to incorporate and demonstrate many of the Pandora key ideas and research artifacts presented in this thesis. Finally, Chapter 8 concludes and discusses potential future directions.

Chapter 2

The CONNECT Network-on-Chip IP Generator

2.1 Introduction

Today's integrated circuits contain billions of transistors organized as tens to hundreds of interacting modules as a System-on-Chip (SoC). As the scale and complexity of modern SoCs grow, building the on-chip interconnect to enable this vast number of modules to communicate is an increasingly important and challenging task. As a result, the interconnect has become a central element in modern chip designs. Previously, the communication needs of smaller, simpler chip designs with just a handful of major modules could be met by ad-hoc point-to-point wires or a shared bus. Such approaches do not scale to handle the more complex and demanding communication requirements of the interacting modules in today's SoCs. This realization led to a push towards more sophisticated and scalable systematic interconnect schemes, like Networks-on-Chip (NoCs), which, as the name implies, implement a dedicated network of links and routers to act as

the communication substrate of a chip [33].

The proliferation of NoCs has been followed by a surge in NoC-related research that spans all the way from low-level hardware implementation issues, such as efficient allocator designs, to higher level issues that affect the application, such as providing traffic isolation or Quality-of-Service guarantees. As a result, NoC designs today form a broad and very diverse landscape that mirrors the equally diverse communication needs and requirements of the various applications running on modern SoCs. Despite this vast design space and the fact that there is no "one-size-fits-all" solution to the interconnect problem, deployable NoC IPs currently available to the research community are limited to fixed NoC instances or target only localized portions of the NoC design space.

CONNECT. In an effort to support my work on Pandora and significantly expand the NoC options available to academic and other research communities, we developed and released CONNECT, a flexible NoC IP generation engine that produces high quality synthesizable[1] Verilog RTL NoC implementations. CONNECT, evolved from a tool originally created to support our own FPGA-specialized NoC design exploration research [27, 29, 81, 82] and was publicly released in 2012 in the form of a web-based NoC generation service (http://www.ece.cmu.edu/calcm/connect). CONNECT allows quickly navigating the NoC design space and generating fast lightweight NoC IPs. To satisfy the diverse communication needs and design constraints of different applications, CONNECT offers a high degree of parameterization to support a very wide range of NoC design variants. By adhering to a set of common interfaces, CONNECT allows for quickly switching between different NoC alternatives, thus enabling rapid experimentation and prototyping.

[1]Synthesizable designs are those described at a sufficient level of detail for Electronic Design Automation (EDA) tools to implement them or "synthesize" them in hardware, e.g., using FPGA or ASIC design flows.

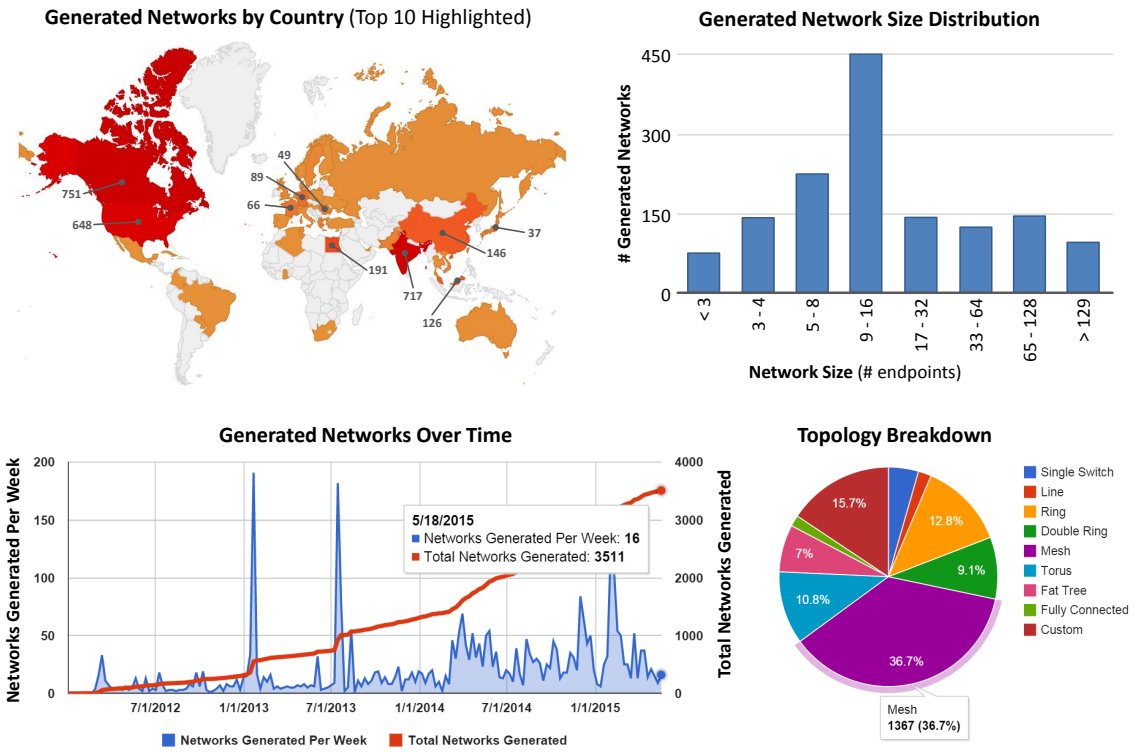


Figure 2.1: CONNECT Network Generation Statistics.

Since its release in 2012, CONNECT has been used by a growing number of users around the world. As of this writing, the CONNECT website has seen more than 15,000 hits and 10,000 unique visitors. The CONNECT service has generated more than 3,500 networks for more than 800 users in more than 50 countries. There have been multiple third-party research papers that use CONNECT to generate NoCs for design space research or for direct use as production IP in design projects. Figure 2.1 shows collected network generation statistics, such as topology breakdown and network size, to give a sense of the types and scale of networks that CONNECT users have requested. More detailed statistics and usage information are available on the CONNECT website [68].

2.2 Background: NoC Terminology

This section offers a brief review of key NoC terminology and concepts relevant to this chapter. For a more comprehensive introduction please see [31]. Readers already familiar with NoCs may continue directly to Section 2.3.

Topology. The topology of a network specifies how routers and endpoints are arranged and connected.

Packets. A packet is the basic logical unit of transfer within the network and can consist of multiple flits (see below).

Flits. When traversing a network, packets are often broken into flits (flow control digits), which are the basic unit of resource allocation and flow control within the network. Some NoCs require special additional "header" or "tail" flits to carry control information and to mark the beginning and end of a packet.

Head-of-line Blocking. Head-of-line blocking [63] refers to a situation where the first packet in a queue is blocking all the remaining packets waiting in the queue, when they could otherwise

be making progress (if not for the first "stuck" packet). Head-of-line blocking can severely limit network performance.

Virtual Channels. A channel corresponds to a path between two points in a network. NoCs often employ a technique called virtual channels (VCs) to provide the abstraction of multiple logical channels over a physical underlying channel. Routers implement VCs by having non-interfering flit buffers for different VCs and time-multiplexed sharing of the switches and links. Thus, the number of implemented VCs has a large impact on the buffer requirements of an NoC. Employing VCs can help in the implementation of protocols that require traffic isolation between different message classes (e.g., to prevent deadlock [32]), but can also increase network performance by reducing the effects of head-of-line blocking.

Deadlock. Deadlock refers to a situation where there is a cyclic dependency involving multiple network resources (e.g., buffer space), which is preventing the network from making any progress.

Quality-of-Service (QoS). QoS refers to a network's capability of providing guarantees (e.g., with respect to latency or bandwidth) or prioritizing specific types of traffic over others.

Flow Control. In lossless networks a router can only send a flit to a downstream receiving router if it is known that the downstream router's buffer has space to receive the flit. "Flow control" refers to the protocol for managing and negotiating the available buffer space between routers. Due to physical separation and the speed of router operation, it is not always possible for the sending router to have immediate, up-to-date knowledge of the buffer status at the receiving router. In credit-based flow-control, the sending router tracks credits from its downstream receiving routers. At any moment, the number of accumulated credits indicates the guaranteed available buffer space (equal to or less than what is actually available due to delay in receiving credits) at the downstream router's buffer. Flow control is typically performed on a per-VC basis.

Input-Output Allocation. Allocation refers to the process or algorithm of matching a router's

input requests with the available router outputs. Different allocators offer different trade-offs in terms of hardware cost, speed and matching efficiency. Separable allocators [31] form a class of allocators that are commonly used in NoCs. They perform matching in two independent steps, which sacrifices matching efficiency for speed and low hardware cost.

NoC Performance Characterization. The most common way of characterizing an NoC is through load-delay curves, which are obtained by measuring packet delay under varying degrees of load for a set of traffic patterns. A common metric for load is the average number of injected flits per cycle per network input port. Packet delay represents the elapsed time from the cycle the first flit of a packet is injected into the network until the cycle its last flit is delivered. For a given clock frequency, load and delay are often reported in absolute terms, e.g., Gbits/s and ns.

2.3 NoC Design Parameterization

Besides CONNECT, there are several other freely available synthesizable NoC IPs (e.g., [12, 43, 92, 98]; additional examples and discussion in Section 2.7). These IPs typically come in the form of structural RTL design modules. As such, their parameterization is limited by the expressiveness of current hardware description languages, such as Verilog or VHDL. The static nature of these design modules limits their configurability (e.g., usually restricted to a single or a limited set of topology configurations). IP users trying to use such NoC IPs also have to deal with low-level details, such as editing RTL design files to set parameters or configure individual routers and writing additional RTL code to arrange routers in a desired topology and populate routing tables.

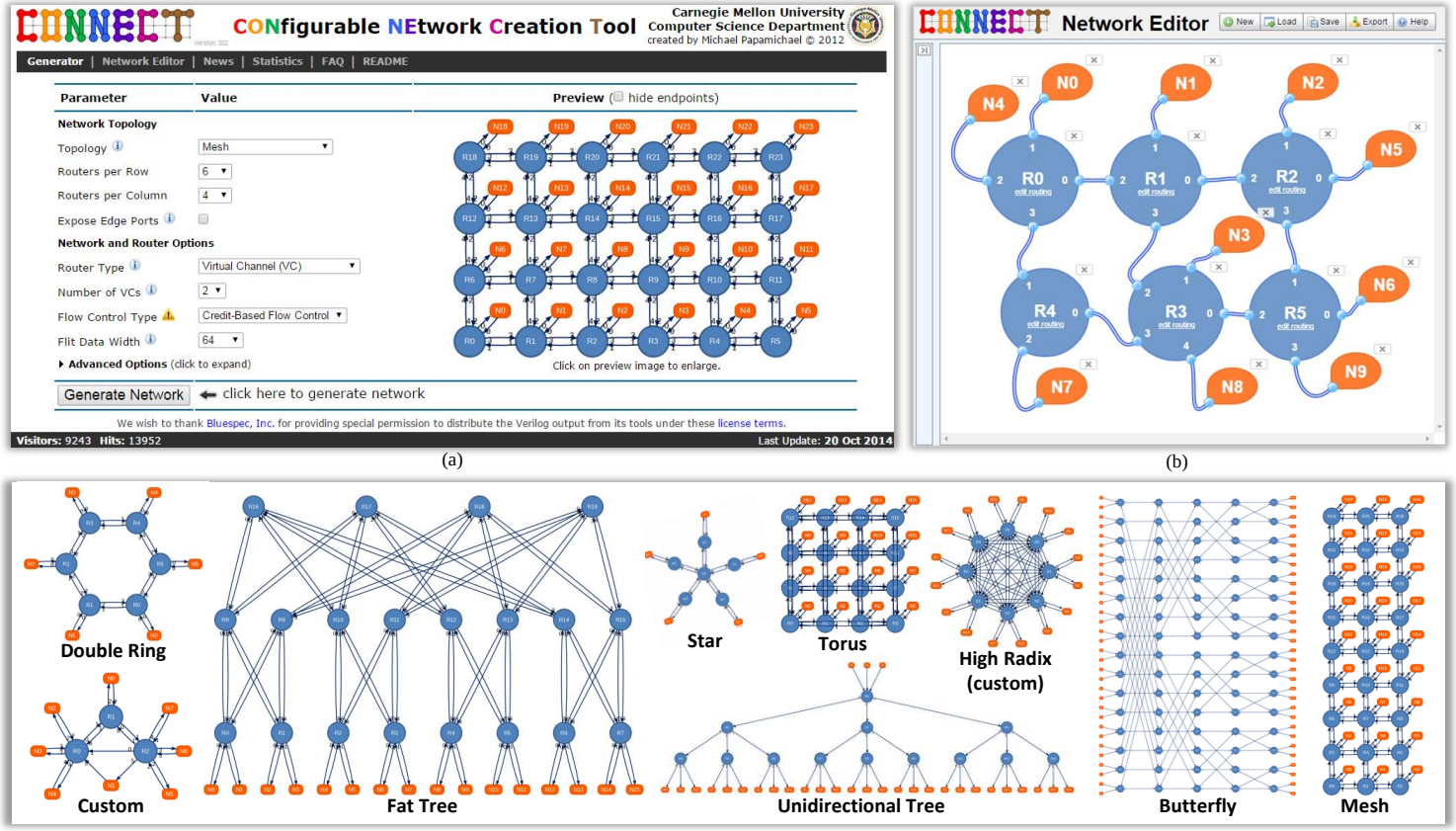


Figure 2.2: The Web-Based CONNECT NoC Generator (a) and Network Editor (b) along with Samples of Pre-Selected and Custom Topologies (c).

CONNECT embodies many of the Pandora key principles discussed in Chapter 3. One of CONNECT's main goals is to drastically reduce the complexity involved in configuring and generating working NoC designs. To this end, CONNECT offers a "push-button" solution for generating a very wide range of NoC configurations. To support a high degree of parameterization, which spans multiple key NoC design choices (e.g., topology, router architecture, flow control, allocation algorithms, pipelining options, buffer sizing, etc.), the CONNECT NoC generation engine dynamically generates the requested NoC designs on-demand. The rich design space of CONNECT NoC IPs is presented to the user through a web-based front-end interface, which consists of multiple high-level user-friendly configuration interfaces that are dynamically updated to guide users while they interact with the generator. Figure 2.2 shows screenshots of CONNECT's web interface along with sample network topologies. CONNECT's main interface offers support for a wide range of common network topologies and displays a dynamically-generated visual preview of the router and endpoint arrangement for each candidate network. Below we highlight CONNECT's most prominent configuration options with emphasis on network topology and router design options.

2.3.1 Network Topologies

CONNECT offers direct support for a wide range of pre-selected common unidirectional and bidirectional topologies (single switch, ring, double ring, star, mesh, torus, fat tree, fully connected, butterfly, distribution/aggregation tree). Each supported topology family includes its own extensive set of scaling and configuration parameters. CONNECT also populates the routing tables in each network using a default routing scheme according to the selected topology variant (which the user is free to override). This wide range of topology and configuration options allows a meaningful degree of customization to satisfy the connectivity needs of many common applications with a very low barrier-to-entry and this is already an important capability over other available NoC IP alternatives,

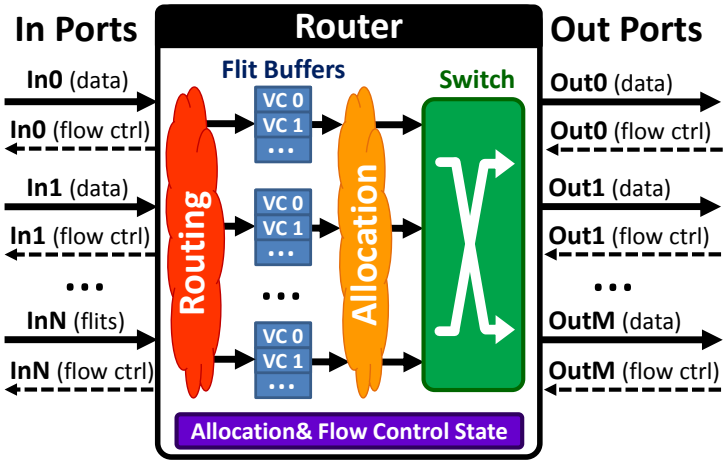


Figure 2.3: High-Level Architectural Diagram of a CONNECT Router.

which typically only target a single fixed topology and often require significant knowledge and manual effort to configure.

In addition to the pre-selected set of topologies described above, the CONNECT NoC generator also supports the creation of custom arbitrary-topology NoCs through the use of a visual network editor (Figure 2.2 (b)) or by using a custom network specification language. These advanced interfaces allow for instantiating routers of different radix, mixing unidirectional and bidirectional links, and attaching multiple (or no) endpoints to each router in the generated network. This high degree of customization allows expert users to build networks that precisely match the connectivity and communication characteristics of their application. Figure 2.2 (c) shows samples of some of CONNECT's pre-selected topologies, as well as some instances of custom topologies.

2.3.2 Router Architectures

Figure 2.3 shows an abstracted block diagram of the basic structure of a CONNECT router. Communication with other routers happens through input and output port interfaces, which can vary in number depending on the router configuration. Each input (or output) port interface consists

of two channels; one channel for receiving (or sending) data and one side channel running in the opposite direction for flow control. Input and output interfaces are either connected to network endpoints or are used to form links with other routers in the network. Optional auxiliary signaling is available to query each router's unique ID and to dynamically update routing information.

To tailor to a wide variety of diverse application communication requirements and to meet different design objectives, CONNECT routers are available in three major variants: "Virtual Channel", "Virtual Output Queued", and "Input Queued".

- **Virtual Channel (VC):** The VC router supports a variable number of VCs and organizes incoming traffic at each input into separate buffers based on VC information carried by packets. Employing VCs can help in the implementation of protocols that require enforcing Quality-of-Service (QoS) guarantees, such as traffic isolation and prioritization between different message classes (e.g., prioritize responses over requests to prevent deadlock [32]), but can also be used to increase network performance by mitigating the effects of head-of-line blocking [63].
- **Virtual Output Queued (VOQ):** The VOQ router can offer the highest performance out of the three supported architectures. For each input it steers incoming traffic into per-output dedicated buffers, eliminating the effects of head-of-line blocking and offering very high levels of performance that can approach that of an ideal (but impractical) output-queued router architecture [31]. VOQ routers are well suited for demanding applications with heavy communication requirements and less structured traffic patterns that would still suffer from head-of-line blocking using a conventional VC-based router.
- **Simple Input Queued (IQ):** The IQ router employs a single buffer per router input and is often well suited as a baseline design. It is useful for building simple bare-bones NoCs for applications that require basic connectivity at low hardware cost. NoCs built around IQ

routers are a good match for applications with non-critical or simple communication needs that do not require the isolation, prioritization, or higher performance offered by the VC and VOQ router architectures.

For each of the variants above, CONNECT allows configuring an extensive set of low-level details, including different pipelining options, multiple allocator alternatives, and flexible user-editable routing, to enable implementation-level trade-offs with respect to performance, area, frequency, as well as to meet specific traffic prioritization and fairness goals. CONNECT also offers support for two interfacing options that implement different flow control protocols ("credit-based" and "peek" [81]) to better match the communication assumptions and requirements of a given application.

2.4 Generated NoCs

NoCs are typically used or studied as part of larger designs with multiple interacting modules that exhibit diverse communication characteristics and often impose stringent hardware resource constraints. Thus, in addition to providing a wide range of design options to meet the communication demands of a given design, it is also important for CONNECT to produce high-quality NoC implementations that map well to the available hardware resources. To this end, all NoC IPs generated by CONNECT, including any debug and instrumentation structures, comprise of fully synthesizable Verilog descriptions that map efficiently to both Field Programmable Gate Arrays (FPGAs) and Application Specific Integrated Circuits (ASICs). Moreover, to ensure that CONNECT NoCs can coexist in harmony and share resources with other hardware-resident components in FPGA environments with tight resource constraints, CONNECT is capable of generating very lightweight NoCs by taking special consideration of unique FPGA implementation characteristics.

	Network Details				FPGA Implementation (LX760T)			ASIC Implementation (32nm)		
Network (# indicates endpoints)	**Link Width**	**Ports / Router**	**#Routers (Arch.)**	**#Virtual Channels**	**Area (% LUTs)**	**Max Freq. (MHz)**	**Peak Bisection BW (in Gbps)**	**Area (um^2)**	**Max Freq. (GHz)**	**Peak Bisection BW (in Gbps)**
Ring_128	128	2	128 (IQ)	N/A	9.0	312	80	566406	5.0	1278
Ring_64	64	2	64 (VC)	2	3.3	250	32	294455	4.2	538
DoubleRing_16	48	3	16 (VC)	4	1.8	160	31	102095	1.8	342
DoubleRing_32	64	3	32 (VC)	2	2.8	177	45	144086	4.0	1014
FatTree_16	32	4	20 (VOQ)	N/A	2.0	138	71	22208	4.0	2067
Mesh_16 (4x4)	32	4	16 (VC)	4	2.9	116	30	78458	3.5	907
Mesh_48 (6x8)	24	4	48 (VC)	2	3.0	124	24	49862	4.3	821
Torus_20 (4x5)	64	5	16 (VOQ)	N/A	5.7	128	131	49230	4.7	4768
FullyConnected_8	32	8	8 (VC)	2	3.2	85	22	19817	4.2	1079
HighRadixCustom_16	48	9	8 (IQ)	N/A	4.3	78	30	31259	5.1	1960

Table 2.1: Sample CONNECT NoC Implementation Results for a 32nm ASIC Commercial Standard Cell Library and a Xilinx LX760T FPGA.

To provide a broad indication of CONNECT NoC quality on FPGAs and ASICs, Table 2.1 shows implementation results for select realistic CONNECT NoC configurations. The presented network configurations capture a representative sample of CONNECT networks that vary in the number of endpoints, link width, number and architecture of routers, and number of virtual channels (when using VC routers). Results include maximum frequency, area, and peak bisection bandwidth based on synthesis estimates for FPGA and ASIC implementations. FPGA logic area is reported as a percentage of total FPGA LUT capacity (LX760T); it is worth noting that all of these sample networks fit well within 10% of a moderately-sized Xilinx LX760T FPGA.

2.4.1 Specializaing for the FPGA Substrate

What is not apparent in Table 2.1 is that the detailed parameter settings leading to optimal results on FPGAs vs. ASICs can be drastically different because the two implementation environments are very different with respect to the relative speed and cost of logic, wires, and memory primitives. CONNECT takes into account the unique mapping and operating characteristics of FPGAs, such as their dense configurable routing substrate, on-chip storage peculiarities, and frequency limitations, to produce specialized NoCs that make very efficient use of FPGA resources.

We focus on these FPGA characteristics that influence fundamental CONNECT NOC design decisions: (1) the relative abundance of wires compared to logic and memory; (2) the scarcity of on-die storage resources in the form of a large number of modest-sized buffers; (3) the rapidly diminishing return on performance from deep pipelining; and (4) the field reconfigurability that allows for an extreme degree of application-specific fine-tuning.

Abundance of Wires. As previously also noted by other work [60], FPGAs are provisioned, even over-provisioned, with a highly and densely connected wiring substrate. As a result, wires are plentiful, or even “free”, especially relative to the availability of other resources like configurable

logic blocks and on-chip storage. In CONNECT we make the datapaths and channels between routers as wide as possible to consume the largest possible fraction of the available (otherwise unused) wires. In addition, we also adapt the format of network packets; information that would otherwise be carried in a separate header flit is carried through additional dedicated control wires that run along the data wires. Finally, we also adapt flow control mechanisms to occupy fewer storage resources by using wider interfaces [81].

Storage Shortage. Modern FPGAs provide storage in two forms: (1) Block RAMs with tens of kilo-bits of capacity, and (2) small tens-of-bits Distributed RAMs built using logic Look-Up Tables (LUTs). Both of these monolithic memory macros can not be subdivided, which can lead to inefficiencies. This sets up a situation where NoCs on FPGAs pay a disproportionately high premium for storage because NoCs typically require a large number of buffers whose capacities are each much bigger than Distributed RAMs but much smaller than Block RAMs. To make the most efficient use of storage resources, CONNECT only uses Distributed RAM and implements multiple logical flit buffers in each physically allocated buffer on the FPGA. CONNECT does not use any Block RAMs, which are typically in high demand from the rest of the FPGA-resident user logic.

Low Clock Frequency. FPGA designs tend to operate at significantly lower clock frequencies compared to ASIC designs, which was one of the gaps studied in [58]. This frequency gap can be attributed to the use of LUTs and long interconnect wires and results in rapidly diminishing returns when attempting to deeply pipeline a FPGA design to improve its frequency. To minimize FPGA resource usage and network latency, CONNECT routers are based on a shallow single-cycle pipeline architecture. As we will see later, the single-stage router used in CONNECT reaches lower but still comparable frequency as an ASIC-tuned 3-stage-pipelined router. The FPGA's performance penalty from running at a lower frequency can be much more efficiently made up by

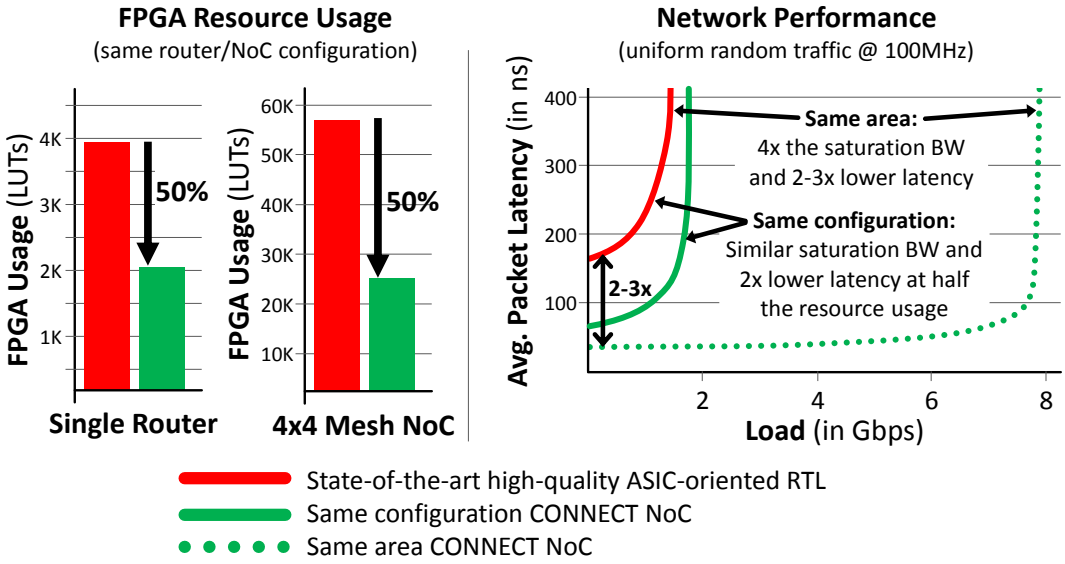
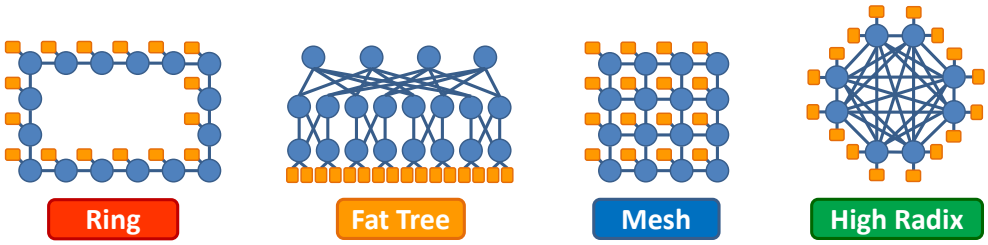


Figure 2.4: FPGA Efficiency and Network Performance Comparison of CONNECT-Generated RTL Against High-Quality ASIC-Oriented RTL [98] for a 4x4 Mesh Network.

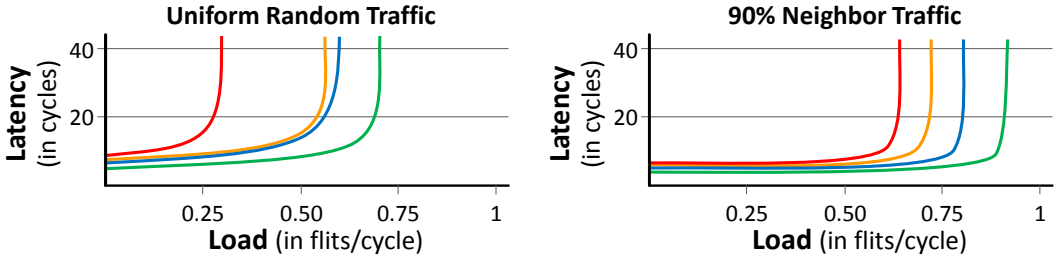
increasing the width of the datapath and links or even switching to an entirely different topology.

Reconfigurabilitiy. Given the flexibility of FPGAs stemming from their reconfigurable nature, an effective NoC design is likely to be called to match up against a diverse range of applications. To cover the needs of such a diverse and rapidly changing set of applications, the CONNECT NoC generator is fully parameterized and more importantly topology-agnostic, which means that individual routers can be composed to form arbitrary custom network topologies. Moreover, to minimize changes in the user logic, all CONNECT networks adhere to the same simple standard common interface. From the user's perspective the NoC appears to be a plug-and-play black box device that receives and delivers packets. Rapid prototyping and design space exploration become effortless as any CONNECT network can be seamlessly swapped for another CONNECT network that has the same number of endpoints.

Figure 2.4 showcases an example of the efficiency benefits of CONNECT-generated NoCs



(a) Four sample CONNECT networks, all interchangeable from user perspective.



(a) Performance of four networks under "uniform random" and "90% neighbor" traffic.

Figure 2.5: Performance of Four Sample CONNECT Networks Under "Uniform Random" and "Unbalanced" Traffic.

specially tuned for FPGAs [81]. When compared against a high-quality publicly available ASIC-oriented 4x4 mesh NoC [98], equivalent CONNECT-generated NoC instances can offer comparable network performance at one-half the FPGA resource cost; or alternatively, three to four times higher network performance at approximately the same FPGA resource cost.

2.4.2 Specializing for the Application

In addition to specializing an IP's implementation for the hardware substrate, it is also important to specialize an IP for the specific usage scenario or application. As an example in the context of NoCs, consider the four NoCs shown in Figure 2.5 (a). All of these networks support 16 endpoints, and as such would be interchangeable from an application perspective. Figure 2.5 (b) shows the load-delay results for these four networks under two different traffic patterns, "uniform random", where the destination for each packet is randomly selected, and "unbalanced", where

90% of the generated packets are local and are sent to neighboring nodes. These two example traffic patterns can be thought of as corresponding to two different classes of applications, each with different degrees of local communication. The fact that the relative performance of each network drastically changes depending on the specific traffic pattern points to the importance of application-specific specialization of the NoC.

Shrinkwrap. In the Shrinkwrap work [27] we experimented with compiler-guided development of application-specific Networks-on-Chip within the CoRAM FPGA memory abstraction [28]. For this work, we extended CONNECT to support a class of tree-based topologies that were a good fit for the traffic patterns exercised by various CoRAM applications. Compared to using a baseline generic NoC, across a number of CoRAM application instances, the customized NoCs generated through CONNECT reduced FPGA resource usage (for the interconnect) by almost an order of magnitude, while retaining the same application performance levels as a baseline generic NoC [29]. As an example, the overall efficiency (throughput/area) gains for two FPGA-based applications (Dense Matrix Multiply and Black Scholes) from switching from a baseline generic mesh interconnect to a custom application-tuned tree-based interconnect ranged from 37% to 48%.

2.5 CONNECT Principles

Our motivation for developing and releasing the CONNECT NoC generation framework was to create a powerful and user-friendly research tool that would be useful to the broader research community. To achieve this goal we adhered to a set of design principles that span all aspects of CONNECT from front-end user interface to back-end hardware generation engine. Below we highlight some key design elements that we conscientiously engineered into CONNECT to enhance its usability as a NoC research tool.

Matching Interfaces to User Expertise Level. CONNECT offers a variety of user interfaces for configuring and generating NoCs. These interfaces can take different forms and are tailored to the expertise of the specific IP user. They include a basic web-based front-end that offers a set of preconfigured common topologies and settings, a more advanced visual custom topology network editor coupled with a custom specification language that eliminates the need for low-level bug-prone RTL coding. To further aid IP users, CONNECT's interfaces also guard against erroneous configurations—which are a common problem when dealing with complex highly-parameterized IP blocks—by, for example, dynamically updating the available options as the user is making selections and automatically populating routing tables. In addition, the interface guides the user through visual cues (e.g., preview of topology and endpoint arrangement) and feedback (e.g., tips on how different options affect hardware implementation). Separately, for expert users who wish to drive CONNECT through the command-line or by automated scripts, we also provide a non-GUI command-line front-end (summarized in Appendix A) that generates NoC instances by remotely connecting to the CONNECT framework.

Rapid Prototyping and Exploration. The CONNECT NoC architecture is designed around a simple set of link-level interfaces that are common among all CONNECT-generated NoCs as to allow easy integration within a design project. From a user perspective, the NoC appears to be a plug-and-play black box module that receives and delivers packets. This simplifies rapid prototyping and design space exploration as all CONNECT NoCs with the same number of endpoints are interface-compatible at the network boundary. Moreover, CONNECT supports dynamic runtime updating of routing tables, which not only opens up interesting research directions, such as experimenting with adaptive routing techniques, but can also drastically reduce experiment turn-around time. Regarding the latter, a user can build a system around a dense highly-connected topology and then modify the routing scheme on the fly to emulate other topologies (e.g., overlay a ring or mesh

on top of a torus), without having to repeat the time-consuming synthesis process, which can take many hours.

Easy Integration. In addition to the plug-and-play nature of the generated networks, CONNECT also offers features specifically targeted at easing endpoint implementation. For example, "peek" flow control allows endpoints to directly observe buffer occupancy of routers (in lieu of credit-based protocols), and "virtual links" guarantee contiguous transmission and delivery of multi-flit packets eliminating the need for reassembly logic and buffering at the receiving endpoints. Features like these push complexity, which would otherwise be handled by the network endpoints, back into the network. Last but not the least, each generated NoC is accompanied by a host of supporting material, such as documentation, testbenches, scripts, as well as user-editable routing and topology files, all custom-generated to match the specific NoC configuration.

CONNECT as a Service. Behind the scenes, CONNECT is a sophisticated tool comprising of many components developed using several different tools. Releasing CONNECT to users as a self-maintained package would be quite challenging both in terms of initial installation (e.g., installing tools and libraries, setting up a proper environment, acquiring licenses, etc.) and continuing upkeep. To this end, we decided to release CONNECT in the form of a web-based IP generation service. This approach to IP dissemination reduces complexity for the user and greatly lowers the barrier to entry; the only requirement for generating CONNECT NoC designs is an internet connection. As an added benefit, having a single point of distribution allows quick and transparent delivery of bug fixes or improvements.

2.6 Pandora Motivation

Our experience with building and releasing CONNECT has provided us with invaluable insight into multiple aspects of IP development, dissemination, and usage. For NoC design experts,

CONNECT is a very powerful tool in saving them the time and effort to code—not to mention debug—the Verilog design of the NoC; they only have to dial-in exactly the configuration they are looking for. However, user feedback has revealed that a large portion of CONNECT users are not NoC experts. This class of users do not know what configuration to ask for and many times do not understand all of the low-level NoC parameters that CONNECT offers.

We see in our generation statistics that most CONNECT users only configure very few high-level parameters (and often suboptimally), such as topology or datapath width, and typically leave most other options, such as router architecture or allocator type, untouched, despite their significant impact on cost, performance, and correctness. This problem continues after non-NoC-expert users integrate a CONNECT NoC into their design, as they are likely to also be unable to properly diagnose performance and correctness issues (e.g., degraded performance or deadlock due to sub-optimal router architecture or allocator choice). Despite the ease of use promised by IP generators like CONNECT, there can still exist a wide "knowledge gap" between "domain-experts" who develop the IP and "non-domain-experts" who use the IP. This gap is becoming evident in general as IPs encapsulate greater complexity and support higher degrees of detailed parameterizations. This observation was the original inspiration to start work on the Pandora IP development paradigm to help close this knowledge gap.

2.7 Related Work

The rapid growth in both research interest and commercial applications of Networks-on-Chip has led to the development of several NoC-related tools and frameworks. In particular, academic and research-oriented efforts have yielded a variety of NoC-related public releases over the past few years. The Stanford Open-Source Router RTL [98] provides a flexible state-of-the-art Virtual-Channel router implementation in synthesizable Verilog. Netmaker [92] consists of a library of

various synthesizable NoC components, along with supporting material and scripts to run simulations under different traffic patterns. Atlas [88] is a mesh and torus NoC generation and evaluation framework. Bluetiles and Bluetree, both part of Blueshell [91], are mesh and tree NoC implementations in Bluespec System Verilog for connecting processor cores to each other and with memory. NoCBench [37] provides a set of hardware and software models and tools to help evaluate NoC designs. Finally, the FPGA NoC Designer tool [73] offers implementation estimates for hard and soft NoCs targeting FPGA devices.

In the commercial space, several companies offer interconnect solutions, which however are typically not publicly available for academic or research use. This includes SoC-oriented solutions, such as the Spidergon STNoC, Arteris' FlexNoC, ARM's AMBA, Sonics' NoCs products, as well as interconnect architectures that are commonly also used in FPGA environments, such as ARM's AXI, found in modern Xilinx FPGAs, or Altera's Qsys. Academic and research solutions, such as the ones mentioned above, or CONNECT, which is presented in this article, can often synergistically coexist with commercial interconnects to cover the diverse communication needs of FPGA-based research and emerging SoCs.

Chapter 3

The Pandora IP Development Paradigm

This chapter describes the key ideas and principles of the Pandora IP development paradigm that aims at reining in the growing complexity of modern highly-parameterized IP generators. To achieve this goal, in Pandora, IP blocks not only capture the microarchitectural and structural view of a design but also encapsulate additional dimensions of knowledge that the IP author has to offer, which can come in the form of: (1) detailed IP design space characterization to help the user understand the effects of parameter choices with respect to hardware implementation and IP-specific metrics, (2) application-level goal-oriented parameterization that is meaningful to the IP user and automatically sets low-level structural parameters to achieve the desired design optimizations, and (3) purpose-built domain-aware simulation-time and run-time monitoring mechanisms to assist functional and performance debugging. In addition to reducing complexity and boosting productivity, the Pandora approach also dramatically reduces the combined total effort, because work that would potentially otherwise be repeated by each IP user, is now only performed once and can be leveraged by others.

Pandora marks a departure from the current status quo in hardware design by combining a set

of key ideas and principles that aim at empowering both IP developers and users. In addition to encompassing the low-level crude hardware description of a design, the IP is enriched with domain-expert knowledge and includes supporting mechanisms, tools, and a diverse set of interface layers to match the expertise of the IP user. Combined, these features give the IP a sense of "smartness" or "self awareness" that enhance how the user interacts with it and can simplify and accelerate the integration, tuning, and validation phases of the design cycle.

The rest of this chapter is organized as follows. Section 3.1 provides background on the rise of hardware design complexity and motivates the need for Pandora. Section 3.2 describes the salient ideas and principles underlying Pandora. Section 3.3 touches on existing efforts that, like Pandora, also try to tackle various aspects of the hardware design complexity problem. Section 3.4 introduces and gives an overview of projects described later in this thesis that demonstrate the Pandora principles. While this chapter presents Pandora at the conceptual level and focuses on the broader guidelines underlying Pandora, Chapters 4, 5, and 6 present more tangible research artifacts that embody the principles described in this chapter and demonstrate key aspects of the Pandora IP development paradigm.

3.1 Background: The Rise of Design Complexity

Over the last few decades technology scaling has closely tracked Moore's Law [74], which refers to the empirical observation that the number of transistors in a chip doubles approximately every 18 months. This exponential growth in transistor counts, which has surprisingly persisted until today, along with Dennard scaling [36], which predicted that power density remains constant as transistors get smaller, have served as major drivers in the semiconductor industry for multiple decades. As a result, by the 1990s integrated circuits already contained tens of millions of transistors while power consumption still remained a second-order concern.

The Productivity Gap. As chip density continued its exponential growth, hardware designers struggled to keep up. This discrepancy between the number of transistors available on a single chip and the ability of designers to efficiently use these transistors was identified about fifteen years ago and was labeled the "design productivity gap", which is illustrated in Figure 1.2 of Chapter 1. This worrying trend sparked research in multiple aspects of hardware design, including high-level synthesis techniques [44, 65], validation tools, more powerful hardware description languages [20, 102] and frameworks [95] that enable the development of flexible IP generators and new hardware design methodologies, such as platform-based design [53].

In particular, the increased (re)use of Intellectual Property (IP) blocks, which refer to pre-made, pre-validated, reusable packaged units of hardware, has been recognized as a very promising approach to alleviate the productivity gap. Instead of designing every component in a chip from scratch, designers can build entire chips or portions thereof by leveraging third-party prepackaged IP blocks, which can greatly reduce the development time and cost of individual submodules within a larger chip. Compared to other approaches trying to tackle the growing productivity concerns, IP reuse faired as a simpler, more tangible and immediate solution, that can be quickly adopted by the semiconductor industry, because it does not require significant changes to the design process. Sure enough, the proliferation of an ever-growing number of rich IP catalogs came as a much-needed productivity boost that would bridge the productivity gap to some extent through modular design and heavy IP reuse.

The Power-Constrained Era. Since this design productivity gap was identified in the 1990s, transistor counts have continued to rapidly increase driven both by Moore's Law, as well as recent technological advances in Integrated Circuit (IC) fabrication, such as the use of silicon interposers or other forms of 3D stacking technologies. However, the inability to further scale supply voltage (due to leakage concerns) has led to the breakdown of classical CMOS scaling as described

by Dennard [36]. This, in turn, has cast power dissipation as a first-order concern in hardware design affecting all facets of computing, from embedded systems and smart phones to datacenter servers or even high-performance computing. In an effort to continue increasing performance in the power-constrained setting of the post-Dennard era, designers are turning to increased use of application-specific special-purpose hardware. Hardware specialization is a promising path towards more energy-efficient computing because it can lower the energy required to perform a task.

The confluence of these trends—namely the ever-growing availability of transistors, which are now in the billions, and the increased need for power-efficient special-purpose hardware within a chip—is leading to the development of massive chips, that include tens or hundreds of interacting modules, organized in intricate multi-level hierarchies. Yet, despite the clear power efficiency benefits of special-purpose hardware and our ability to fabricate denser chips with billions of transistors, current design methodologies have not evolved at the same pace to handle the complexity associated with such massive, diverse designs. As a result, designing a chip today requires large skilled hardware design teams and costs more than ever, even without considering manufacturing costs.

The IP Complexity Wall. Despite an enormous rise in scale, complexity, and specialization, the way IP blocks are developed and used has not fundamentally changed since the introduction of modern Hardware Description Languages (HDLs) and the proliferation of ASIC-based design flows more than two decades ago. The result is a "complexity explosion" as designers build chips as "fragile" collections of complex IP blocks, each with its own set of cryptic (for the non-domain-expert) low-level knobs, which can often be traced back to a crude hardware schematic or specification document. Today, we are facing a new encounter with the "design productivity gap" at a different scale—not at the level of the transistor, but instead at the level of the IP block. Current hardware design methodologies are struggling to keep up with the complexity involved in config-

uring, tuning, integrating, and validating the multiple interacting IP blocks within a modern chip. Consequently, the complexity and the associated development time, cost, and manpower required to build a chip today continue to increase prohibitively.

3.2 Pandora Principles

The overarching goal of this work is to tackle the problem of increasing complexity in hardware design and make "reusable" IPs more "usable" through the Pandora IP development paradigm. To achieve this goal, Pandora raises the level of abstraction to allow non-domain-experts to easily and efficiently navigate and identify sweet spots within the design space, and use and debug highly parameterized IPs, without having to deal with the low-level and often cryptic details of modern IP design. The rest of this section delves into the salient principles of Pandora, which all share the common goal of overcoming the "Complexity Wall" in hardware design.

3.2.1 Detailed IP Characterization

In addition to capturing the microarchitectural and structural view of a design, Pandora IPs also carry qualitative and quantitative meta-data. This information captures how the various knobs and parameter settings affect the IP design space with respect to hardware implementation and higher-level domain-specific properties and performance characteristics. This embedded knowledge provides the foundations for a Pandora IP to provide an elevated design abstraction and supporting functionalities for the IP users.

Hardware Implementation Characterization. This refers to capturing how the IP parameters affect implementation characteristics of the design, such as area, critical path, and power dissipation. In its simplest form this can be a database or characterization library that is attached to and maintained along with the IP. Alternatively, this can also be complemented or take the form of

predictive tools or analytical formulas that approximate implementation trends, such as DELPHI, presented in Chapter 4. Implementation information can be maintained at different degrees of fidelity, ranging from classes of hardware devices (e.g., FPGAs or ASICs), specific device families (e.g., Altera Stratix V), or instances (e.g., Virtex-6 XC6VLX760), or technology libraries (e.g., TSMC 32nm). Moreover, additional information can be maintained depending on the specific IP and implementation target. For instance, in an FPGA environment, area can be broken down into LookUp Tables (LUTs), BlockRAMs, DSPs, etc., or similarly in a design with multiple clock domains, critical path information can be kept on a per-clock basis.

Domain-Specific Metrics and Properties. Besides hardware implementation details, which are typically common across all different types of IP, Pandora also argues for capturing how the various IP parameters affect higher-level metrics that are specific to the domain at hand. These are also typically the metrics that the IP end-users are interested in and try to adjust to meet application-specific goals. For instance, in the case of a processor core, such a metric could be IPC (Instructions Per Cycle), or in the case of a NoC IP, such metrics could be the saturation bandwidth and idle latency of the network. This characterization can also include high-level properties, e.g., in the case of a NoC IP, capture how IP parameters affect packet delivery and ordering guarantees, traffic isolation or Quality-of-Service properties.

These characterization libraries are meant to aid the design process and do not necessarily need to be exhaustive or perfectly accurate. As we show in Chapter 5 with Nautilus, exploring even a small subset of the design space is often sufficient to vastly accelerate the IP design space search and optimization process. Depending on the degree of parameterization and complexity of the IP, the characterization of the IP design space can be done in many ways and varying degrees of detail: e.g., (1) collectively for the entire IP (i.e., sweeping over externally-exposed parameters), (2) per submodule (i.e., considering the internal parameters of each submodule in the hierarchy),

(3) through selective sampling of the design space (driven by domain-expert knowledge), or (4) through the derivation of analytical formulas that predict implementation and performance results based on expert knowledge or experience.

Even in its raw form, whether it is detailed characterization libraries based on experimental data (e.g., synthesis runs or simulations) or coarser grain predictive data based on analytical formulas and designer experience, this extra knowledge that is coupled with the IP is already very useful to the IP user. Not only does it facilitate faster and more informed navigation of the design space, but can also help drastically prune the design space by identifying parameter combinations that do not constitute interesting or feasible design points. To demonstrate this aspect of Pandora, the CONNECT Network-on-Chip generator has been extended as part of this thesis to provide FPGA and ASIC estimates through the DELPHI framework, which is described in Chapter 4.

3.2.2 Automated IP Optimization

It is typical for modern parameterized IP blocks to expose a large set of low-level raw "structural" parameters, which are directly tied to and affect low-level implementation details of the resulting hardware, e.g., datapath widths, buffer depths, memory dimensions, arithmetic operation precision, etc. For example, the top-level router module of the Stanford Open Source Network-on-Chip Router project [98] exposes 42 parameters with multiple additional parameters per each submodule, leading to billions of valid design variants. Access to such low-level parameters gives fine-grain control to expert hardware designers familiar with the domain pertaining to the IP. However, as hardware designs scale in size and comprise of a growing number of IP blocks spanning different expertise domains and are often developed and maintained by different engineers or even third-party IP vendors, properly setting and tuning the myriad of low-level parameters associated with each submodule becomes unmanageable for IP users who are looking to be shielded from the

internal complexity of the IP in the first place.

To alleviate this problem, the authors of Pandora IPs should raise the level of abstraction at which the user interacts with the IP by exposing high-level configuration and tuning interfaces that are meaningful at the application level and intuitive to the end-user of the IP. In Pandora, IP authors enrich their IP blocks with high-level information about their functionality, parameter settings, tuning, capabilities, and also capture how the various knob settings relate to hardware implementation details, such as area, power, or clock frequency, instead of raw structural choices. This abstraction layer empowers non-domain-experts to easily and effectively navigate the design space to meet application-level design goals. This high-level parameterization allows users to achieve the desired customization objectives but nevertheless very effectively shields the users from the inner workings or low-level details of the IP.

Depending on the domain, user objectives and level of expertise, the provided interfaces can take different forms, span varying levels of abstraction, and be tailored to match different classes of users. For example, in a NoC setting these interfaces can range from something very simple, such as providing the user with a set of known-to-be good specialized configurations or "personalities" to choose from (e.g., pre-selected NoC configurations tailored for stencil computations, one-to-many data distribution, or point-to-point bulk DMA transfers) to a sophisticated objective and constraint-driven query system (e.g., "find minimal area NoC that runs at 800MHz and offers 50Gbps of bisection bandwidth with 2 virtual channels for traffic isolation and prioritization") or even a specialized interface that is only meaningful within the specific domain (e.g., analyze a weighted graph or set of traces that capture the traffic patterns of an application to find best NoC configuration).

To support this thesis, our work on CONNECT and Nautilus demonstrates some of the Pandora ideas described above. CONNECT, presented in Chapter 2, offers a multitude of configuration

interfaces that are tailored to the expertise of the specific IP user and guard against configuration errors. These interfaces include (1) a web-based front-end that offers a set of preconfigured common topologies for the beginner user, (2) a graphical user interface [67] that allows designing semi-custom topologies for the more advanced user, and (3) a detailed command-line interface, as well as a specification language for describing fully custom networks that offers full control to expert users, but still eliminates the need for low-level bug-prone RTL coding. Our work on Nautilus, presented in Chapter 5, provides an even more general solution to the IP configuration and optimization problem by demonstrating how IP author knowledge can be incorporated in guided genetic algorithms to vastly accelerate and automate hardware IP design space search and optimization across multiple types of IPs. Chapter 7 shows how we augmented CONNECT with additional high-level configuration interfaces powered by Nautilus.

3.2.3 Sophisticated IP Instrumentation

To reduce the complexity of using an IP and accelerate the hardware development cycle once an IP instance has been chosen and integrated within a larger hardware design, Pandora argues for including instrumentation and introspection mechanisms that constantly monitor, collect, analyze, and visualize detailed information about the IP's operation. These mechanisms, which can either be both part of the generated hardware (whether simulation only or synthesizable) and also consist of supporting tools performing analysis in a post-processing step, enable accelerated validation as well as more effective performance and cost optimization by guiding the designer through meaningful feedback.

To be amenable to the non-domain-expert and simplify how users interact with the IP, instead of flooding the user with low-level raw data, Pandora IPs, imbued with the IP author's domain-expert knowledge, come with the necessary tools to analyze and interpret the raw data to (1) capture high-

level effects (e.g., in the case of a NoC detect deadlocks, capture congestion effects, or identify bottlenecks); and (2) trace back and inform the user about the root cause of any correctness and performance issues. As a result, the IP user receives meaningful feedback that relates to and captures application-level behaviors, which, in turn, accelerates the verification process and enables effective performance and cost optimization.

Instrumentation. Instrumentation refers to tapping into a design for monitoring, validation, or statistics collection purposes, and is commonly done in an ad-hoc fashion by IP users while working with an IP instance (e.g., to identify performance issues or find a bug). Pandora argues for:

- Introducing instrumentation support during the IP authoring process and bundling it as part of the IP generator. An immediate obvious benefit of this approach is reduced effort, as this process is done once during the IP authoring stage and does not have to be repeated by each IP user. More importantly, when the instrumentation is staged by the IP developer, who is naturally more familiar with the domain and has a much better grasp of the inner workings of the IP, it will be of higher quality and more effective at collecting the proper set of low-level data needed to draw conclusions about potential IP correctness and performance issues.
- Keeping all or the majority of the instrumentation synthesizable to allow the IP user to maintain a unified and consistent view of the IP's operation, whether the IP is used within a simulation environment or running on actual hardware.

Benefits of Synthesizable Instrumentation. Synthesizable instrumentation is especially useful in a reconfigurable FPGA setting for a number of reasons. Firstly, even considering the possible area and timing penalty of turning on instrumentation, running in hardware is still multiple orders of magnitude faster than RTL simulations. This not only shortens the development cycle, but also improves coverage from a verification perspective, as the design can now quickly reach states that

would otherwise never be exercised in a simulation environment. Secondly, the flexible nature of FPGAs allows for quickly turning instrumentation on and off, which can be, for example, particularly useful when trying to diagnose performance bottlenecks. Instead of having to recreate the same scenario in a simulation, the designer can quickly switch to an instance of the design with instrumentation and start collecting data. Thirdly, synthesizable instrumentation allows designers to capture hardware artifacts and behaviors that would otherwise be very hard or impossible to reproduce in a simulation environment (e.g., DRAM controller refresh). Finally, since all measurements are taken by directly probing the actual running hardware, they are bound to be more accurate than those obtained within a simulation environment.

Instrumentation can take many different forms, depending on its purpose and the nature of the IP; it can range from a simple set of passive counters that monitor interesting events to more sophisticated stateful pieces of logic that can keep track of sequences of events and even interact with the IP. For instance, in the case of a processor IP, a simple example of instrumentation could monitor cache misses or keep track of the types of events that cause stalls. Similarly, in a NoC IP, instrumentation can take many different forms and span different levels of the internal IP hierarchy. At a basic level it can be used to monitor link utilization, network load, packet latency, buffer occupancy, average number of hops, or lower level data such as allocator unit matching quality, or how many times a higher priority traffic class blocked a lower level traffic class. A more advanced instrumentation example could pertain to a sophisticated monitoring and diagnostic block that implements the network flow control protocol and is attached to a network port to monitor or stress-test the NoC. Chapter 6, which describes the IRIS instrumentation framework, showcases several such instrumentation examples in the context of CONNECT-generated Networks-on-Chip.

Introspection. Introspection in Pandora refers to the supporting mechanisms and logic that are bundled with the IP and can analyze and visualize the data collected during instrumentation.

This can happen dynamically while the IP is actively being used (or simulated) or in a separate post-processing step that analyzes logs of collected data.

At a basic level, Pandora's introspection mechanisms facilitate better visibiliy into a design and can also help accelerate the verification process by quickly identifying design issues. These can range from static configuration mistakes (e.g., invalid routing table that prevents packets from reaching their destination) to improper integration or use of the IP (e.g., network endpoints do not properly implement flow control). At a more advanced level, Pandora leverages the embedded domain-expert knowledge to capture higher-level domain-specific dynamic effects and behaviors to offer feedback that is more natural, meaningful, and intuitive to the end-user of the IP. For instance, in the case of an NoC IP, such feedback could range from detecting deadlocks to capturing congestion effects or identifying bottlenecks. Chapter 7 describes how we used IRIS to incorporate such introspection features in CONNECT-generated NoC instances.

3.2.4 The IP "Uncore"[1]

The Pandora principles presented up to this point were tied in one way or another to the core functionality of the IP and described Pandora's approach to characterizing, tuning and debugging an IP. These final principles of Pandora span a variety of related topics and mechanisms that affect hardware design and pertain to IP supporting material and toolsets, as well as release and packaging strategies. Although not crucial to the functionality of the IP, this final set of Pandora principles greatly enhance the IP user experience and contribute towards reducing the complexity involved in developing and using an IP block.

Supporting Infrastructure. Contrary to software projects, hardware designs are often released in a very crude form, which can be attributed, at least in part, to the limited expressiveness of

[1]The uncore is a term used by Intel to describe the functions of a microprocessor that are not in the core, but which are essential for core operation and performance.

conventional HDLs. Pandora argues for augmenting the IP with supporting infrastructure that boosts productivity and enhances how the user interacts with the IP. In addition to elementary supporting material, such as documentation and testbenches, this includes supporting toolsets, such as scripts and interfaces for configuring the IP or processing output logs, as well as more advanced supporting infrastructure, such as sophisticated optimization frameworks.

This supporting infrastructure is often domain-specific and as such, needs to be tailored by domain-experts that have a better grasp of the type of supporting material that might be useful to the end user. When building these auxiliary mechanisms, Pandora can leverage the characterization and domain-expert knowledge described previously. For example, in an NoC setting, a configuration tool could enhance design space navigation by tapping into the characterization database to provide instant feedback on hardware implementation characteristics (e.g., frequency, area), predict network performance (e.g., bisection bandwidth, latency), and show previews of the generated network topology or even give hints as to what types of applications would be a good match for the selected configuration. A more advanced example could be a sophisticated feedback-driven optimization framework that processes instrumentation data and iteratively tunes network parameters to reach a design sweet spot.

Releasing the IP as a Service. Given the inability of conventional HDLs to express and capture the high degrees of parameterization required to develop a flexible IP block, developers typically have to either package their IP with ad-hoc auxiliary tools, such as Java applets, that generate instances of the IP (e.g., Xilinx's CORE Generator System [1]) or develop their IP within special languages and frameworks that natively support hardware generation, such as Bluespec [20] or Genesis2 [45]. Regardless of the specific packaging approach, this typically entails additional effort and increases the complexity for the IP user, who typically has to perform multiple steps on his end to start using the IP, such as setting up a proper environment, acquire licenses, install a

series of tools, take care of any library dependencies, etc.

In an effort to reduce complexity and lower the barrier to entry for the IP user, Pandora argues for packaging and releasing IP generators as a service, that isolates the IP user/client from the IP developer/provider. This can be done in the form of a portal that combines the various high-level configuration and tuning interfaces along with interactive feedback mechanisms and other supporting material, such as documentation. The primary benefit of such an approach is that the user can focus on using the IP, without having to worry about equipment, environment and tools setup, or development and configuration logistics.

This service-oriented distribution of the IP also has multiple advantages from an IP developer standpoint, such as facilitating prompt and transparent IP updates. This becomes especially interesting in the presence of the high-level interfaces described earlier, e.g., in an NoC setting, the user can receive an internally improved NoC instance that still meets the same high-level criteria. Providing IP generation as a service through a single point of distribution also allows the IP provider to gather usage statistics, which can be used to improve the IP and guide development or even potentially facilitate crowd-sourced characterization of the IP.

The publicly available CONNECT Network-on-Chip IP Generator, presented in Chapter 2, originally served as the inspiration and eventually as the demonstration vehicle for many of the ideas described above.

3.3 Existing Efforts to Tackle Design Complexity

Other researchers have also recognized complexity as a major obstacle in hardware design, which has led to a number of proposals that attempt to mitigate various aspects of this problem. These efforts vary in scale and range from the development of new hardware description languages, tools, and algorithms that can enhance existing chip development flows, to novel design method-

ologies that fundamentally rethink the way we design hardware. Despite the wide variety, most proposed approaches share many similar underlying themes, such as design reuse, modularity, abstraction, hierarchical design, and orthogonalization of concerns. The remainder of this section highlights some of these efforts that are relevant to or aligned with the work in this thesis.

Design Reuse and Patterns. The reuse of existing design efforts has been a common recurrent theme across many efforts to tackle design complexity. Design reuse can take various forms, from module or IP replication to extensive parameterization to reusing concepts and techniques, and can span multiple levels, from smaller hardware elements, such as an adder circuit, to larger chip modules or components, such as a complex processor block or memory controller. Recent research has also studied the use of design patterns [34], which refer to a more systematic and organized way of classifying and cataloging existing hardware designs in an effort to more effectively leverage design reuse.

Alternative Hardware Description Languages. The majority of hardware design today is carried out using the Verilog and VHDL Hardware Description Languages (HDLs), which were introduced in the 1980s. Even though both of these languages have been updated over the years and do include support for modular design and some primitive forms of parameterization, they still require that designs are described at a low structural level. This makes hardware design a very tedious and bug-prone process and has a negative impact on designer productivity.

Over the last years there have been a number of proposals to replace the aging Verilog and VHDL HDLs to allow designers to shift their focus on the behavior, functionality, and algorithmic view of their hardware design instead of its structural composition and implementation details. Bluespec [20] and Chisel [102] are two examples of recently proposed hardware description languages that allow designers to describe hardware at a higher level, borrowing software constructs, such as loops, conditionals and recursion. Such languages enable high degrees of parameterization

and modularity by allowing for well-structured typed interfaces and support for polymorphism. In the case of Bluespec, designers also benefit from compiler-enabled type checking and scheduling.

High-Level Synthesis (HLS). A more aggressive approach towards the same goal of raising the level of abstraction and increasing designer productivity is synthesis of hardware from high level languages. The main idea behind this approach is to allow designers to express their algorithms or desired behaviors using high-level software-like languages. HLS has gained significant traction in recent years and current research and commercial solutions allow designers to write code using existing software languages (e.g., C/C++), which is eventually converted into hardware. Examples of high-level synthesis tools include Vivado HLS [113], LegUp [25], ROCCC [5], Catapult [66], and Impulse [2].

Despite having a lot of potential, HLS is still a long way from replacing traditional HDL-based hardware design. For arbitrary hardware designs, current high-level synthesis solutions typically produce lower quality results compared to conventional hardware design flows and are also limited in terms of their expressiveness as they can only handle subsets of existing software languages. Still, recent research has shown promising results when using HLS tools within specific problem domains, such as signal processing [77] or nested loop transformations [75, 110].

Communication Architectures. As the number of interacting modules on a chip continued to rapidly grow, communication was quickly recognized as a critical and time-consuming part of the design process. To keep design time and cost under control, both academia and industry have studied tools and frameworks [15, 19, 49, 87, 93] that automate and facilitate the process of designing and implementing communication mechanisms. As the number of interacting modules kept increasing, more recent efforts have shifted their focus from traditional bus-based and ad-hoc interconnect solutions to the more scalable approach of using Networks-on-Chip [33, 46]. This shift also came with a push for interface standardization to promote modularity and design reuse.

New Design Approaches and Methodologies. In an effort to overcome the increasing challenges in chip design, researchers have also explored higher-level unified approaches and methodologies to tackle design complexity, which often combine or leverage some of the techniques and approaches already described above, such as design reuse or interface standardization. In platform-based design [53], chips are built as platform instances, which are compositions of library elements that are represented by models of varying fidelity and adhere to a common set of rules and interconnection standard interfaces defined by the "platform". The "platform" serves as an abstraction layer or API that allows for quick design space exploration and shields designers from low-level details. This approach is particularly useful in a System-on-a-Chip (SoC) setting, where designs are implemented as collections of pre-made IP blocks.

Complementary approaches to platform-based design that also aim towards the same goal of raising the level of abstraction have looked at new ways to describe hardware, how to embed designer knowledge in IP blocks and how to create templates or generators that can produce multiple variants of a hardware design from a common description. These efforts have led to the development of higher level software-like languages, such as SystemC [8], as well as highly specialized hardware description languages, often specifically tuned to a particular domain, such as digital signal processing [69, 77]. To enable embedding of designer knowledge and template-based design, the Genesis2 project [45] offers a powerful framework that builds on top of SystemVerilog and allows designers to build chip generators [95], such as a multiprocessor generator [97] or a floating-point unit generator [96].

3.4 Demonstrating Pandora

In support of this thesis and to demonstrate various aspects of Pandora, we have: (1) developed and extended the CONNECT Network-on-Chip generator to demonstrate many of the

Pandora design principles, such as application-specific goal-oriented NoC optimization [83], and NoC-specialized system-level instrumentation and introspection mechanisms, (2) developed DELPHI [84] for performing fast and efficient IP characterization (power, area, frequency) across multiple technology nodes, (3) developed and evaluated Nautilus [83], an IP optimization engine, which uses modified genetic algorithms that incorporate IP author knowledge to perform automated guided design space search, and (4) developed a set of flexible reusable simulation-time and run-time instrumentation components to aid in hardware functional and performance debugging.

DELPHI, presented in Chapter 4, pertains to a flexible open framework that leverages the DSENT modeling engine [99] for fast, easy, and efficient characterization of RTL hardware designs. In the context of Pandora, DELPHI can be used to accelerate the IP characterization process and rapidly map an IP's design space with respect to implementation characteristics or other IP-specific metrics of interest. The DELPHI flow processes Verilog or VHDL RTL hardware descriptions to generate a technology-independent DSENT design model, which can then be used to perform very fast—one to two orders of magnitude faster than full RTL synthesis—estimation of hardware performance characteristics, such as frequency, area, and power.

As a next step towards realizing the Pandora vision, we developed Nautilus [11], presented in Chapter 5, which builds on top of the DELPHI characterization engine to help IP users perform parameter optimization and navigate an IP's design space in a fast and automated manner. At the core of Nautilus is a modified genetic algorithm, that allows embedding of IP-author knowledge pertaining to the IP design space. This knowledge, coming in the form of "hints", captures the IP author's intuition about how IP parameters relate to the various metrics of interest; the goal is to help steer the optimization search process toward profitable regions or directions in the design space. Our evaluations across multiple IPs show that author-guided instances of Nautilus can achieve the same quality of results up to an order of magnitude faster than a baseline genetic algorithm.

To demonstrate the instrumentation and introspection ideas of Pandora, we developed IRIS, which is presented in Chapter 6 and pertains to a flexible systematic instrumentation framework that (1) allows for fast and efficient hardware debugging and monitoring, and (2) enables system-level visibility and analysis. IRIS includes a library of parameterized simulation and/or runtime instrumentation modules, as well as a post-processing and visualization engine that can help capture and identify higher level system behaviors and design issues, which not only help the designer during the development cycle but also enable the end-customer of a system gain high-level insights about the system's operation.

Finally, parts of all of these efforts are combined in a proof-of-concept Pandora-powered version of CONNECT, which is showcased in Chapter 7.

Chapter 4

DELPHI - Fast IP Characterization

This chapter presents DELPHI, a flexible, open IP characterization framework that facilitates the first key principle of Pandora, namely detailed IP design space characterization. DELPHI leverages the DSENT [99] modeling engine for faster, easier, and more efficient characterization of RTL-based hardware IPs. DELPHI first synthesizes a Verilog or VHDL RTL design (either using the industry-standard Synopsys Design Compiler tool or a combination of open-source tools) to an intermediate structural netlist. It then processes the resulting synthesized netlist to generate a technology-independent DSENT design model. This model can then be used within a modified version of the DSENT flow to perform very fast—one to two orders of magnitude faster than full RTL synthesis—estimation of hardware performance characteristics, such as frequency, area, and power across a variety of DSENT technology models (e.g., 65nm Bulk, 32nm SOI, 11nm Tri-Gate, etc.). In our evaluation using 26 RTL design examples, DELPHI and DSENT were consistently able to closely track and capture design trends of conventional RTL synthesis results without the associated delay and complexity.

The rest of this chapter is organized as follows. Section 4.1 motivates the need for RTL-based

design evaluation. Section 4.2.1 provides background on RTL synthesis and characterization flows and known issues with relying on RTL-synthesis results. Section 4.2.2 offers additional background on the DSENT tool. In Section 4.3 we present the DELPHI flow and discuss its strengths and limitations. Section 4.4 reports our evaluation methodology and experimental results. Finally, we discuss related work in Section 4.5.

4.1 The Need for Fast Accurate Architecture Design Evaluation

Computer architects have predominantly relied on software-based simulation to evaluate the performance and other qualities of design proposals. Unfortunately, simulators—even high fidelity cycle-accurate ones—typically do not capture low-level hardware implementation artifacts, such as area overhead, increase in power consumption, or timing-related side-effects of micro/architectural design choices. To overcome these limitations of simulation and gain more precise insight and deeper understanding of a proposed idea, computer micro/architectural studies have increasingly incorporated register-transfer level (RTL) design investigation to complement simulation studies.

Evaluating a design or sub-component of a system as RTL models typically entails designing and developing a low-level structural hardware implementation using a Hardware Description Language (HDL), such as Verilog or VHDL, and then taking this HDL through an Electronic Design Automation (EDA) flow. Even though EDA flows consist of multiple steps and tools, architects typically only focus on the first step, *synthesis*, which takes the HDL source and implements it using a set of primitive logic standard cells. While still far from the final chip, synthesis can provide a rough—Section 4.2.1 elaborates on this—estimate on the quality of a design (power, area, timing).

As is true for most steps of an EDA flow, synthesis can be a tedious, time-consuming, slow, and error-prone process that requires expert knowledge and access to costly and hard-to-get tools, proprietary process design kits, and standard cell libraries. Moreover, synthesis—which can take

from a few minutes to several hours depending on design size, complexity, and implementation goals/constraints—has to be repeated for each implementation variant targeting different technology nodes and standard cell libraries. Finally, because synthesis tools are designed to be chained and used in conjunction with other commercial closed-source specialized EDA tools, they maintain and operate on a low-level internal representation of a design, which hinders integration with traditional software-based simulation frameworks. This makes it very challenging to co-simulate or combine hardware RTL modules with software-based components.

This chapter presents DELPHI, a flexible open framework that leverages DSENT electrical modeling for timing, area, and power [99] for faster, easier, and more efficient characterization of RTL hardware designs. Previously, DSENT could only be applied to hand-created DSENT design models in a DSENT-specific format. DELPHI enables DSENT modeling to be applied to generic RTL designs by translating a synthesized RTL-based design (in the form of a structural netlist) into a software-based technology-independent DSENT design model. This model can then be used within a modified version of the DSENT flow to perform very fast—one to two orders of magnitude faster than synthesis—estimation of hardware performance characteristics, such as frequency, area, and power.

Compared to traditional EDA flows, DELPHI is a simpler and faster alternative that captures design trends and is consistent with actual synthesis results. By utilizing DSENT design models, DELPHI allows researchers to use existing or new custom-built DSENT technology models to quickly sweep over a large number of target technology nodes and standard cell variants without having to repeat the lengthy and complex synthesis process. Conversely, researchers are also able to quickly explore how low-level technology parameter changes impact high-level characteristics of designs. The generated models seamlessly interface with the DSENT framework and facilitate integration within larger software simulation frameworks.

4.2 Background

4.2.1 RTL Synthesis

Detailed evaluation of a hardware design is a multi-step process that involves taking a design through a series of Electronic Design Automation (EDA) tools. This process starts with a description of a design, typically using a Hardware Description Language (HDL), such as Verilog or VHDL, and ends with a hardware implementation that corresponds to the detailed layout of a chip. This description has to be at a sufficiently low level, also known as register-transfer level (RTL), in order for the tools to be able to map it or "synthesize" it to hardware. As a design progresses through an EDA flow and gets closer to the final hardware implementation, more details are incorporated, and thus the tools can do a better job of accurately estimating implementation characteristics, such as how much area it occupies, how much power it consumes, and how fast it can run.

Logic synthesis is the first step in an EDA flow and is responsible for turning an RTL description of a design into a set of fundamental logic building blocks (logic gates and registers), which, for an ASIC technology, are then mapped to a collection of standard cells. The resulting standard cell instantiations and their interconnections become what is known as a (gate-level) netlist. This synthesized netlist can vary significantly based on a number of factors, including tool quality and user-specified design goals and constraints. While this post-synthesis netlist corresponds to a functionally correct implementation of the original RTL description, it does not capture significant hardware implementation details pertaining to the physical layout of a circuit, which are determined in later steps of an EDA flow, such as "Place and Route (P&R)"[1]. As such, hardware design char-

[1]P&R refers to the EDA process of physically laying out a chip that includes finding a valid placement of its standard cells, creating a network of wires to connect them together, taking care of power distribution, and creating a clock network.

acterization based on synthesis output can often deviate significantly from the final implementation produced at the end of the EDA flow.

Confidence in synthesis results. Hardware designers and the EDA community are well aware of the "noisy" and speculative nature of synthesis results, as well as of the gap between post-synthesis and post-P&R results [21, 56, 90]. However, in the architecture community, it is quite common for researchers to place a lot of trust in synthesis results and treat them as the "ground truth". Moreover, given the complex nature of modern synthesis tools and EDA flows, it is not uncommon for non-experts to misconfigure or misuse synthesis tools, which only exacerbates the observed variance and unrepresentative nature of synthesis results.

Below we list the most important factors that can cause variance or introduce "noise" in synthesis results, along with some illustrative examples.

- **Lack of Physical Layout Information.** Synthesis tools treat a design at an abstract level as a collection of interconnected standard cells. Layout and other implementation details are determined at later stages of the EDA flow. As such, synthesis tools either completely ignore layout artifacts (e.g., ignore wire length and assume an ideal clock network) or use simplified models to estimate layout effects. In practice, synthesis results usually tend to be "optimistic" and it is not uncommon to see them deviate by 30%-40% compared to post-layout results. As an example, in a recent ASIC design effort within our group, our finalized design could only achieve a clock frequency that was 38% lower than the synthesis-based estimates.

- **Optimization Goals, Constraints, and Settings.** Synthesis is an iterative process that is guided by user-specified optimization goals (e.g., optimize for area), constraints (e.g., run at 2GHz), and many other settings (e.g., effort level, driving/load assumptions for circuit input-

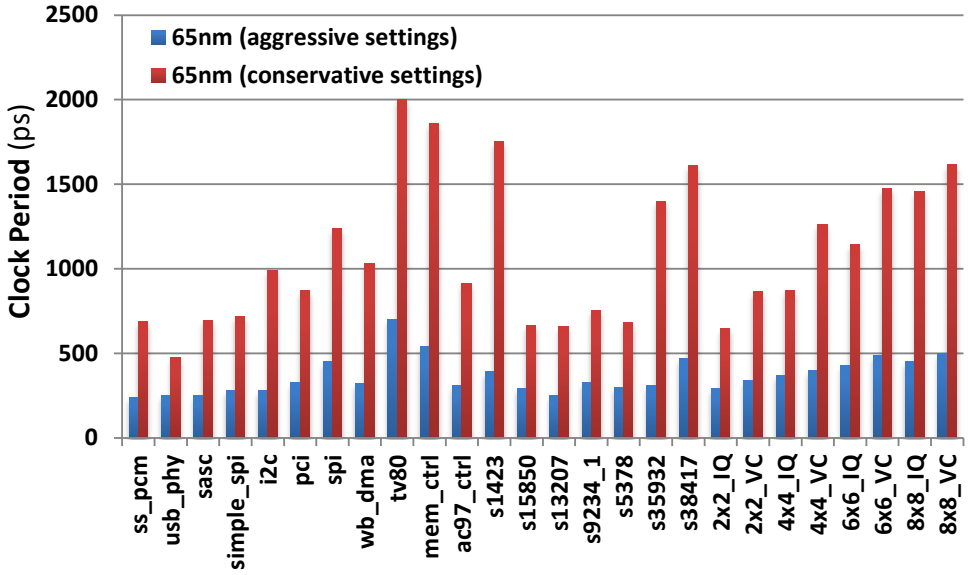


Figure 4.1: Clock Period with Aggressive vs. Conservative Synthesis Settings.

s/outputs, etc.), which can all significantly affect the characteristics and performance of the resulting netlist (e.g., different implementations of design subcomponents, standard cell sizing, buffer insertion, register retiming, etc.). To illustrate this point, Figure 4.1 compares the range of reported minimum clock period for a variety of RTL designs, which are synthesized targeting the same commercial 65nm standard cell library; for each design, the differences between the red and blue bars arise purely from using different synthesis optimization/constraint settings.

- **Synthesis Tool Features and Quality.** Synthesis is a complex, highly configurable process. Depending on the selected options, different tools might use different algorithms and employ models of varying fidelity (e.g., different wire and clock tree models) during design optimization or performance estimation. Moreover, commercial tools, such as Synopsys Design Compiler, offer different variants of their tools (e.g., DC Explorer or DC Ultra) and extensions (e.g., use of Designware components, topographical technology, etc.), which can

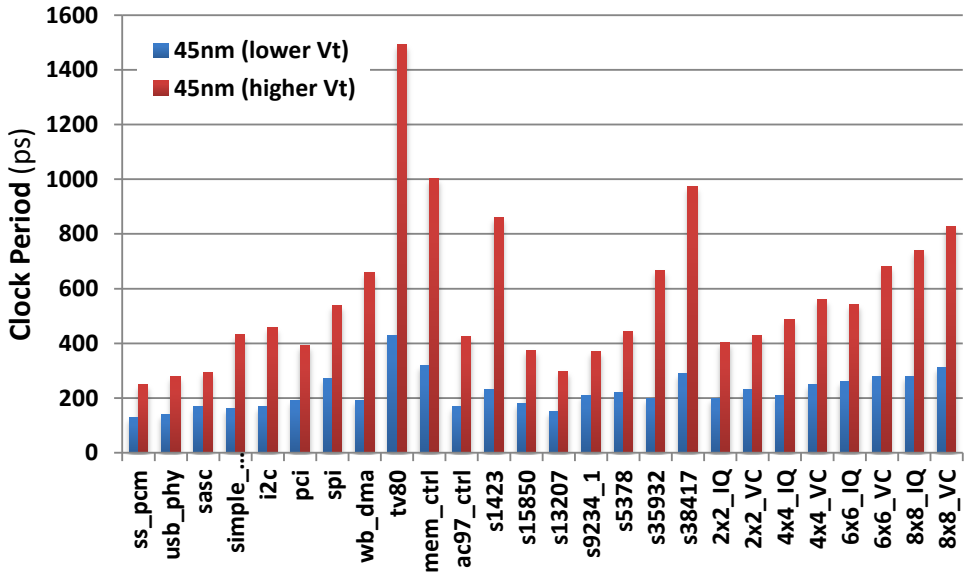


Figure 4.2: Clock Period with Lower vs. Higher Vt Cells.

significantly affect both the generated netlist and the characterization accuracy.

- **Variance Across Standard Cell Libraries.** For a particular process technology, there are usually multiple standard cell libraries from multiple sources. The foundry will often have two distinct sets of libraries, one for internal use and one for external customers. Further, there are third-party standard cell vendors who produce libraries for a variety of process technologies, and within each of these sets of libraries there will be multiple versions (e.g., low power, high performance, compact area). As a result, even within the realm of a single process technology, synthesis results can vary dramatically. To illustrate this point, Figure 4.2 shows the minimum clock period achieved by a variety of designs, all targeting the same 45nm process, but using cell variants with a different threshold voltage (Vt).

In summary, when studying an RTL design at the synthesis level, it is important to keep in mind the various factors that can cause inaccuracies and variance. One must be knowledgeable and

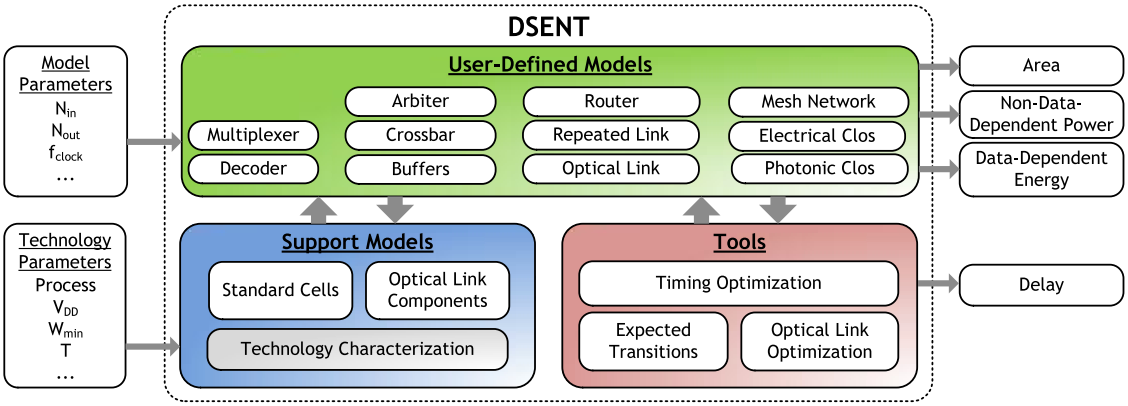


Figure 4.3: DSENT Internal Hierarchy (with authors' permission).

diligent about properly setting the many configuration options pertaining to synthesis. It should be noted that while synthesis results are not to be taken at face value, they are still very useful for performing first-order characterizations of designs and help guide the RTL development and optimization process.

4.2.2 The DSENT Tool

DSENT (Design Space Exploration for Network Tool) is an open-source tool developed for rapid design space exploration of photonic and conventional electrical Networks-on-Chip (NoCs), which was released in 2012 [72]. DSENT can be either used standalone as a dedicated NoC evaluation tool or it may be integrated within an architectural simulator for interconnect modeling [59]. In DELPHI, we take advantage of DSENT's design characterization technology and generalize it to support arbitrary RTL design inputs.

DSENT is written in C++ and is internally organized as three distinct parts shown in Figure 4.3 (from [99] with authors' permission) and described below:

- **User-defined design models** serve as the "front-end" of DSENT that most users interact

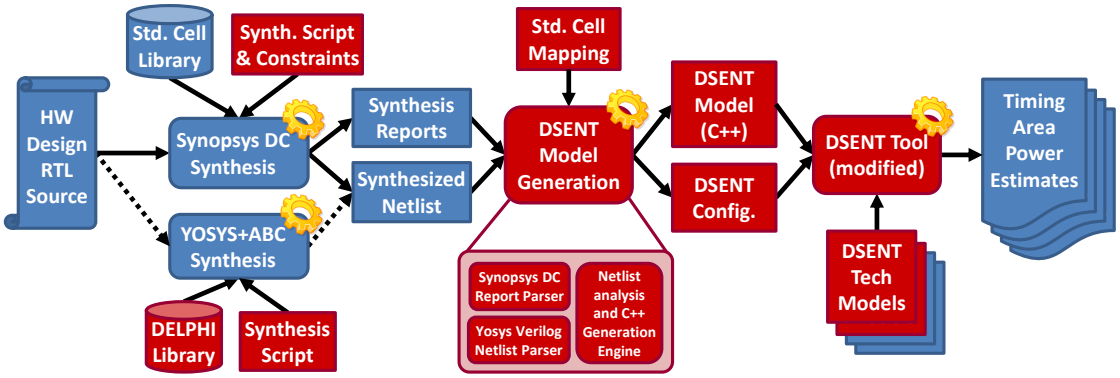


Figure 4.4: The DELPHI Flow (parts shown in red were developed or modified in support of the DELPHI flow, or produced by the DELPHI flow).

with. Within the context of NoC studies, this "front-end" contains a hierarchically organized set of parameterized building blocks that can be easily combined to assemble and experiment with a wide range of on-chip networks. In general, users that wish to extend DSENT's design library with their own custom-defined models are free to develop their own models in C++ or modify DSENT's pre-existing design models.

- **Support technology models** rely on a set of technology parameters to provide the fundamental building blocks that are used to implement (either directly or indirectly) all user-defined models. These building blocks come in the form of standard cells and optical components, whose characteristics are shaped based on a supplied technology model. The technology models used by DSENT aim at capturing the major characteristics of deep sub-100nm technologies based on a minimal set of technology parameters. While DSENT already includes technology models for 45nm, 32nm, 22nm, and 11nm technology nodes, the simple nature of the models allows users to define and calibrate their own technology models based on ITRS [50] data, SPICE models, or actual process design kits.

- **Tools** provided by DSENT include a timing analysis and optimization tool, as well as infras-

tructure for capturing and propagating circuit switching activity information that is used to obtain accurate power estimates. Tools and support technology models form the "back-end" of DSENT, which is responsible for estimating power and timing of a design.

Expanding DSENT's Front-End. DSENT's "back-end" circuit modeling engine is (1) fast, capable of characterizing a circuit in a matter of seconds; (2) high fidelity, as it performs modeling at the standard cell level; and (3) flexible, allowing targeting different technology nodes through the use of simple technology models. However, this powerful back-end is limited by the library of available NoC-specific user-defined models. Users that want to characterize their own (non-NoC) arbitrary hardware designs have to write C++ from the ground up to build new design models. This can be a tedious process and essentially resembles performing "synthesis by hand". To bridge this gap, the subject of this chapter, DELPHI, provides an automated flow that can take RTL descriptions of arbitrary hardware designs and generate a DSENT-compatible design model.

4.3 DELPHI

DELPHI consists of a set of tools aimed at simplifying and accelerating hardware design characterization (determining area, clock frequency, and power). Like conventional RTL synthesis, DELPHI starts from an RTL description of a design. This RTL description is then processed through a conventional synthesis tool (e.g., Synopsys Design Compiler) to produce a netlist targeting a particular standard cell library. This netlist along with other design information is then processed in an automated manner to produce a technology-independent representation of the original design in the form of a DSENT design model. A lightly modified version of DSENT can then use this design model to perform very fast power, area, and frequency estimations across multiple technology models.

As is the case with traditional synthesis tools, DELPHI is useful for performing coarse characterizations of RTL designs and obtaining first-order power, area, and timing results. Despite their approximate nature, DELPHI, as well as DSENT, retain and analyze a design at a low structural level. This offers higher fidelity compared to other commonly-used architectural modeling tools (e.g., [62, 103]) and allows for capturing more subtle design trends as well as identifying more specific potential areas for improvement (e.g., identify critical path in a design).

Using DELPHI. DELPHI includes all of the necessary scripts and tools required to take a design through all of the steps of the flow summarized above, including synthesizing RTL using commercial or open-source tools, generating DSENT design models, and running DSENT. From a user-perspective, DELPHI can be used in the form of a command-line utility. The user only needs to specify a minimal set of details about the RTL module to be processed (such as the name of the top-level module and the name of clock and reset ports) and, if not using the open-source flow, provide details about the particular commercial standard cell library in use.

4.3.1 The DELPHI Flow

Internally, the DELPHI flow consists of a series of steps, shown in Figure 4.4 and outlined below. Parts of the flow that we developed (or extended) in support of the DELPHI flow are colored in red.

Synthesis. As a starting step towards importing a hardware design, DELPHI takes the RTL design through a customized synthesis flow that ensures the resulting netlist is compatible with later steps. DELPHI supports the commercial industry-standard Synopsys Design Compiler (DC) tool, as well as a combination of open-source tools. When working with Synopsys DC, the user can synthesize a design either targeting a commercial standard cell library or using one of the freely available libraries, such as FreePDK [78] or the cell libraries provided by SPORT Lab [105].

DELPHI generates custom scripts that instruct Synopsys DC to only use the subset of standard cells available in DSENT and to generate a set of design reports that capture essential information about the design used by DELPHI.

Once synthesis is completed, DELPHI parses the synthesis output, including the generated netlist, as well as synthesis reports that include port and clock-tree information pertaining to a design, and recreates an intermediate representation of the netlist. At this point, this intermediate representation still corresponds to the standard cells belonging to the original standard cell library used during synthesis. In order for DELPHI to generate a technology-independent DSENT design model, it needs to map the vendor-specific standard cells to generic DSENT standard cells. DELPHI captures this mapping in a custom-defined specification file that assigns the various standard cell variants to their equivalent DSENT counterparts. This file also includes information on standard cell driving strength and pin mapping.

For users that do not have access to commercial synthesis tools or commercial standard cell libraries, DELPHI provides an alternative flow based on open-source tools. In particular, the tool YOSYS [111] is used to parse the RTL of a design and then the ABC synthesis tool [18] is used with a custom-created DELPHI library and script to directly target DSENT's native standard cells[2]. Since the output of these tools is different from Synopsys DC, DELPHI includes a very basic Verilog parser that supports the subset of Verilog used in the netlists produced by these tools.

DSENT Model Generation. DELPHI analyzes the netlist and other information produced during synthesis to automatically generate the C++ code required to implement a DSENT design model. The creation of this model starts with the definition of input and output ports that form the model's interface. DELPHI then instantiates all of the design's standard cells and creates nets, which are used to connect all of the standard cell pins with each other and with the input and output ports

[2]This flow currently supports Verilog, but not VHDL.

of the design. If desired, DELPHI can size the instantiated cells to match the driving strengths suggested by synthesis.

At this point, the DSENT design model is structurally equivalent to the synthesized netlist. DELPHI now generates auxiliary C++ code that takes care of defining model parameters, creating a clock tree, initializing transition info for the input and output ports, as well as the clock and reset signals, and defining the events that should be monitored, gathered, and reported by DSENT. The remaining code that is generated pertains to properly propagating transition probability information in the correct order across all of the instantiated standard cells, which is necessary for performing static power analysis.

Proper Propagation of Switching Activity Information. Dynamic power is dissipated when internal nets are charged and discharged with signal transitions. Thus, to estimate power accurately, DSENT relies on annotating every input/output port and internal net of a design with signal transition probability information, which captures the likelihood of each possible signal transition ($0\rightarrow0$, $0\rightarrow1$, $1\rightarrow1$, $1\rightarrow0$). Based on the type of standard cell and its input transition probability information, DSENT can calculate the transition probabilities of its outputs, which then have to be propagated to all downstream input ports of other standard cells. This propagation needs to happen in the correct order, to ensure all parts of the circuit are annotated with the correct transition probability information, which is eventually converted to a switching activity for a given frequency and used for power estimation.

To this end, a challenge in the development of DELPHI was to deduce the ordering constraints among all nodes in a netlist and then use this information to generate DSENT C++ code that will trigger the propagation of transition probability information in the correct order. DELPHI also takes special care to detect and handle sequential logic (e.g., finite state machines), which can form feedback loops that create circular dependencies. The examples that follow demonstrate the

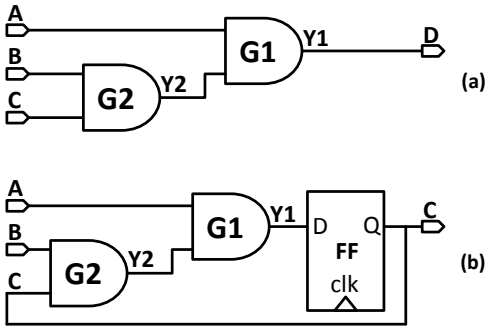


Figure 4.5: Probability Propagation Example Circuits.

importance of updating transition probability information in the correct order. To keep matters simple in these examples, instead of looking at transition probabilities, we only focus on the state probability that a logic signal is high, which still exposes the problem at hand.

Consider the very simple circuit shown in Figure 4.5(a) that consists of the two AND gates G1 and G2. For this example, let's assume that all inputs A, B and C have been annotated with a 50% chance of being high, i.e., P(A)=P(B)=P(C)=0.5, and internal probabilities are also initially set to 0.5, i.e., P(Y1)=P(Y2)=0.5. Our task is to find an order in which we need to process the standard cells in this circuit to correctly deduce the internal probabilities, P(Y1) and P(Y2). Since G1 depends on the output of G2, we first need to process G2 and then G1, which will set P(Y2)=0.25 and then process G1, which will set P(Y1)=0.125. Note that if we processed G1 before G2, we would end up with the wrong probability on the output of G1, i.e., P(Y1)=0.25, as we would be missing the updated probability info for P(Y2).

Now consider a slightly modified version of the previous circuit, shown in Figure 4.5(b), which now also includes a register FF (D Flip-Flop) that forms a feedback loop. As before, assume that all input probabilities are set to 50%, i.e., P(A)=P(B)=0.5. In this case, since G1 depends on G2 and G2 depends on the output of FF, which in turn depends on G1, it is not clear anymore in what order to propagate the state probabilities. To handle such cases, DELPHI takes a simple approach where

it detects such loops and then exercises them in the correct order until the cells participating in the loop have been processed up to a number of times, determined by a user-configurable threshold[3].

To see this in action, assume that we set this threshold to five, which means that DELPHI will update the state probability of the cells that are part of this loop up to five times, respecting their ordering dependencies. Table 4.1 shows how the state probabilities of the various internal nodes evolve over each iteration, assuming we process the nodes in the order G2, G1 and FF. After five iterations, all signals, Y2, Y1, and C, have already been annotated with very low probabilities ($<$1/1000). Intuitively, this makes sense because in this simple circuit it is clear that if either A or B become low, it is then impossible for any of the internal nodes to ever become high again. Note that if we did not iterate over this part of the circuit, we would have ended up with much higher, clearly non-representative state probabilities, which would in turn distort static power analysis and estimation.

Iteration#	**P(Y2)**	**P(Y1)**	**P(C)**
1	0.25	0.125	0.125
2	0.0625	0.03125	0.03125
3	0.015625	0.0078125	0.0078125
4	0.00390625	0.001953125	0.001953125
5	0.0009765625	0.00048828125	0.00048828125

Table 4.1: State Probabilities Over Successive Iterations.

Running DSENT. The C++ code generated in the previous step is now ready to be compiled along with the rest of the DSENT components to get power, area, and timing estimates. As the size of the generated code can reach several megabytes, DELPHI takes special care to optimize the generated

[3]Static switching activity propagation and estimation is a critical part of power estimation and has been studied extensively (e.g., [64]). While DELPHI's approach is simple and does not guarantee probability convergence, it is sufficient for getting first-order power estimates.

C++ for fast compilation, which typically finishes in a matter of seconds even for large designs[4]. DELPHI also offers a special "debug" mode that produces C++ code that is slower to compile, but much easier to read and work with for users that wish to study or modify the resulting DSENT design model.

Appendix B provides an overview of the DELPHI flow steps, including cell mapping specification format, and tool invocations examples.

4.3.2 Strengths of the DELPHI Approach

Leverages DSENT's Speed, Flexibility, and Accuracy. After the initial synthesis phase, which only needs to be performed once, circuit characterization using DSENT technology models only takes seconds and can be repeated targeting different technology models without requiring additional time-consuming synthesis or recompilation of DSENT design models, as DSENT constructs its library at run-time. Moreover, new DSENT technology models are easy to define as they only require specifying a minimal number of parameters. Finally, when properly calibrated, DSENT's integrated timing, area, and power models have been shown to be accurate within 20% against SPICE simulations [99].

Harnesses the Power of Software. Creating a software-based representation of an RTL design opens the doors to many opportunities. Firstly, since the generated models appear as standard "user-defined" DSENT design models, they can be integrated and combined with all of DSENT's existing library of design models. Moreover, these models are portable, meaning that DSENT users can exchange and instantiate them along or inside their own DSENT models. Secondly, the entire DSENT framework can be modified to be integrated with software-based architectural simulation frameworks, as was done in [11, 59, 71].

[4]This was not a trivial task, as our initial implementations generated C++ code that could take up large amounts of memory and tens of minutes to compile and would even occasionally crash g++ for very large designs.

Enables Fast, Automatic Model Creation. When manually building a DSENT model from scratch, users have to either assemble their designs out of library components or directly build them out of standard cells, which from a hardware development perspective is analogous to writing assembly code in software. This can be a tedious and time-consuming process that also implicitly places limits on the size and complexity of user-created models. By automating this process, DELPHI greatly accelerates building DSENT design models, as a single synthesis run is much faster than coding the necessary C++, and also eliminates the inevitable introduction of bugs. Moreover, given the abundance of freely available RTL hardware designs (e.g., from OpenCores [79]), it tremendously expands the potential of DSENT.

Preserves Synthesis Optimizations. Traditional hand-written DSENT models have to explicitly specify and fix all of the implementation details of a given hardware design. Since DELPHI relies on synthesis to build DSENT design models, it can preserve any low-level optimizations performed during synthesis. For instance, given the same RTL design, synthesis might implement subcomponents differently based on optimization goals, (e.g., switch between different adder implementations depending on power/area/timing trade-offs). By using DELPHI, optimizations performed during synthesis are preserved and can be carried onto and modeled within DSENT.

4.3.3 Limitations of the DELPHI Approach

Only as Accurate as Synthesis. The fidelity at which a design is modeled through DELPHI and DSENT is comparable to synthesis tools. This means that design characterization in DELPHI, like conventional RTL synthesis, ignores or makes simplifying assumptions about physical layout artifacts, such as long wire delays, congestion, and clock distribution.

Garbage in, Garbage out. DELPHI and DSENT are only as good as the design model and technology model that they are used with. A low-quality poorly optimized RTL design or a wrong

(or badly calibrated) DSENT technology model will obviously produce wrong or severely skewed results and could even hide or distort trends.

Minimal Set of Standard Cells. The minimal set of cells used in DSENT's technology-independent standard cell library is not as rich as commercial standard cell libraries, which can lead to suboptimal synthesis results, especially for designs that make heavy use of cells that are not supported by DSENT. However, as we show in Section 4.4, this constraint does not affect designs by much. Moreover, new standard cells can be easily added to DSENT and supported by the DELPHI flow to overcome this limitation.

Multiple Clocks. DELPHI currently only supports designs with a single clock domain. To work around this limitation, designs with multiple clocks can be broken down across clock boundaries and modeled piecewise.

Large Memories. By default, synthesis tools build memories in an RTL design as collections of latch or flip-flop standard cells. When dealing with RTL designs that contain large memories, designers will sometimes use proprietary vendor-provided tools, called "memory compilers", to generate optimized memory layouts that are typically more efficient than large memories built using conventional standard cells. DELPHI only supports "synthesized" memories, i.e., built out of standard cells, as custom memories generated using memory compilers are treated as "black boxes" during synthesis and cannot be processed by DELPHI. As future work, to improve large memory modeling, we are considering extending the DELPHI flow to incorporate the use of CACTI [103] memory models.

Limited to Static Analysis. DELPHI and DSENT support static analysis for power estimation, which can provide satisfactory estimates based on probabilistic models of signal transitions. However, more accurate power modeling requires performing simulations that stimulate the circuit using

representative inputs to capture actual switching activity information, which is then fed to commercial power analysis tools, such as PrimePower.

4.4 Evaluation

This section presents our evaluations, based on actual synthesis results targeting commercial standard cell libraries, as well as results obtained through DELPHI and DSENT. In the presentation below, we first validate the assumptions underlying the DELPHI approach before directly evaluating the accuracy of DELPHI estimates.

4.4.1 Methodology

All synthesis results were obtained using Synopsys Design Compiler (Version I-2013.12-SP3) targeting 32nm, 45nm, 65nm, or 130nm commercial standard cell libraries that come from three different vendors. When reporting timing results we synthesize with aggressive constraints to force synthesis to reach the highest possible operating frequency for each target design. For all other results, we synthesize and obtain power results targeting the same fixed frequency of 500MHz[5]. All power estimation results assume a switching activity factor of 0.5 across all design inputs.

For DELPHI results, we use a lightly modified version of DSENT (0.91) targeting the default 45nm, 32nm, 22nm, and 11nm technology models that come bundled with the publicly released version of DSENT, which, in our results, we denote as "DSENT_45", "DSENT_32", "DSENT_22", and "DSENT_11". Our evaluations span 26 hardware designs: 18 benchmarks from the IWLS2005 benchmark suite [26] (including 11 designs from OpenCores and 7 from ISCAS) and 8 on-chip router designs of varying size and architecture obtained through the CONNECT NoC generator [68,

[5]We pick this frequency as it was the highest frequency that could be met for all designs and for all technology nodes used in our evaluation.

81]. Within each group, benchmarks are sorted according to their size.

Finally, it is important to point out that the DELPHI results shown in this section were obtained using the publicly available DSENT "generic" technology models, which do not correspond to any of the commercial standard cell libraries used in this evaluation (and which are protected under strict NDA agreements). As such, when comparing DELPHI results with actual synthesis results, emphasis is placed on capturing the same trends, instead of matching the same absolute numbers. While we focus our evaluation on the 32nm results, the most modern technology node for which we have standard cells, we observe similar results across all technology nodes we have access to.

4.4.2 Results

Constraining Synthesis to DSENT Standard Cell Subset. An artifact of using DELPHI is that synthesis must be constrained to target the DSENT-supported set of standard cells, typically a subset of commercial libraries. The first set of results show the effects of constraining synthesis to only using the subset of standard cells that are available in DSENT. Figures 4.6, 4.7, and 4.8 compare power, area, and timing results from baseline ("unconstrained") and "constrained" synthesis runs targeting a commercial 32nm process.

Overall, "constrained" synthesis can experience approximately 5%-20% quality loss in the design metrics. This is expected as, e.g., a design that could make use of a 3-input AND gate will now have to switch to chaining two AND gates to implement the same logic, which in turn can increase critical path, area, and power. As will be shown later, these minor—in light of the speculative nature of synthesis—distortions do not prevent DELPHI from capturing design trends.

Independence from Standard Cell Library Choice for DSENT Model Creation. DELPHI and the DSENT back-end provide technology-independent flows and modeling that are tolerant to using different RTL-synthesis standard cell library targets when originally generating DSENT

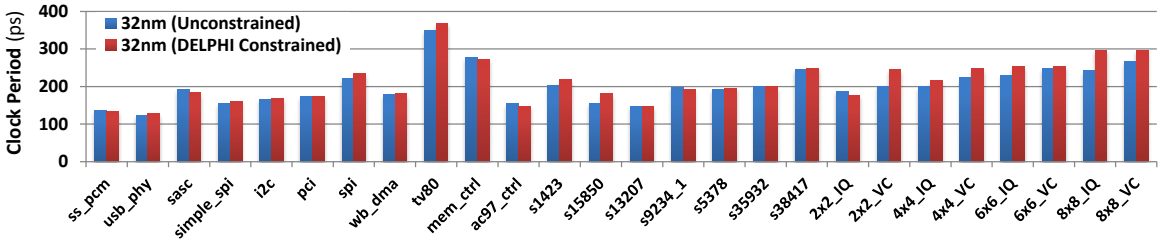


Figure 4.6: Timing Estimates of Regular vs. DELPHI-Constrained Synthesis.

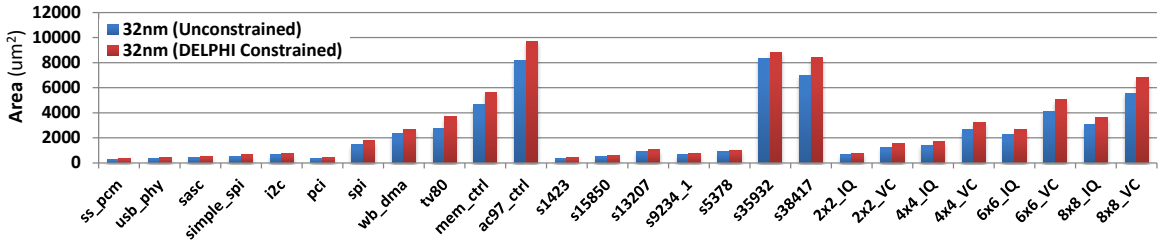


Figure 4.7: Area Estimates of Regular vs. DELPHI-Constrained Synthesis.

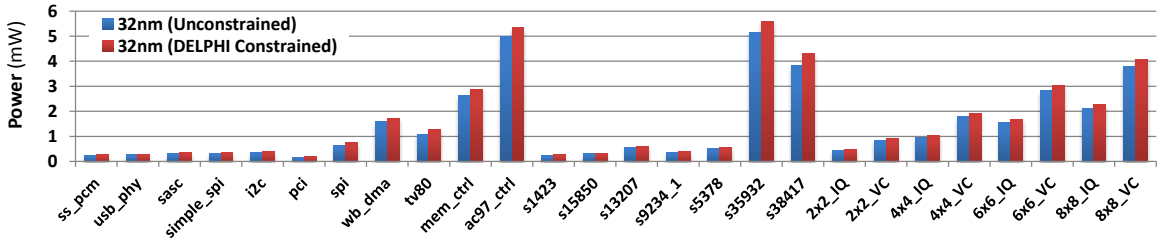


Figure 4.8: Power Estimates of Regular vs. DELPHI-Constrained Synthesis.

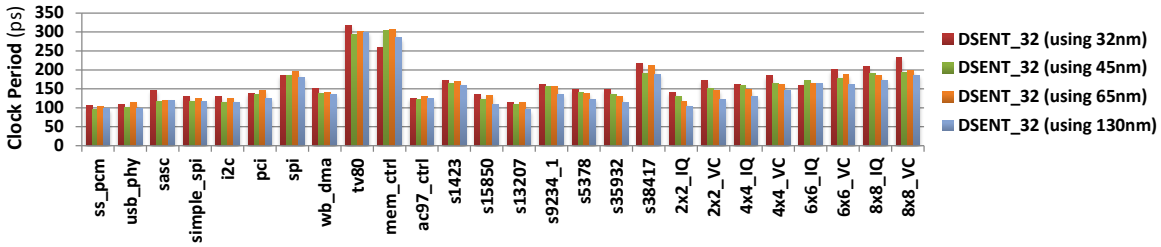


Figure 4.9: Timing Estimates of Four DSENT Design Models All Targeting DSENT_32, But Generated Using Different Intermediate Synthesis Results.

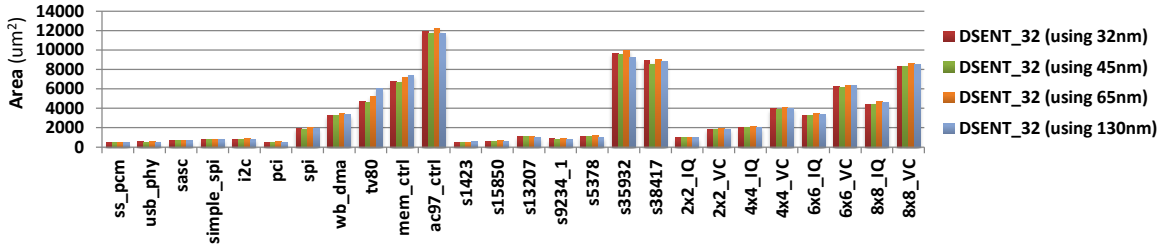


Figure 4.10: Area Estimates of Four DSENT Design Models All Targeting DSENT_32, But Generated Using Different Intermediate Synthesis Results.

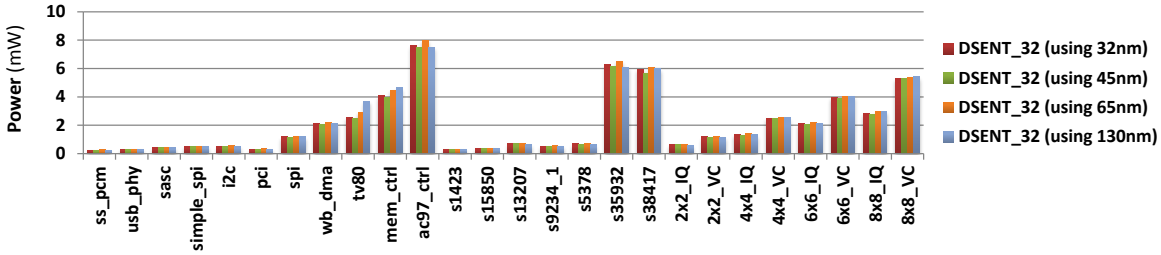


Figure 4.11: Power Estimates of Four DSENT Design Models All Targeting DSENT_32, But Generated Using Different Intermediate Synthesis Results.

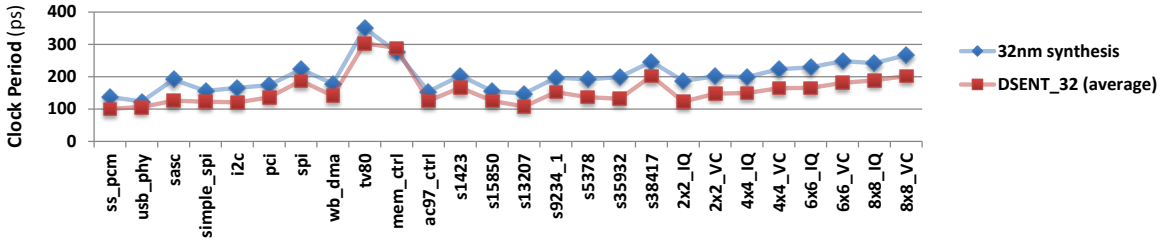


Figure 4.12: Comparing Trends Between the Average Timing Estimates of Four DSENT Models Targeting DSENT_32 vs. 32nm Synthesis Results.

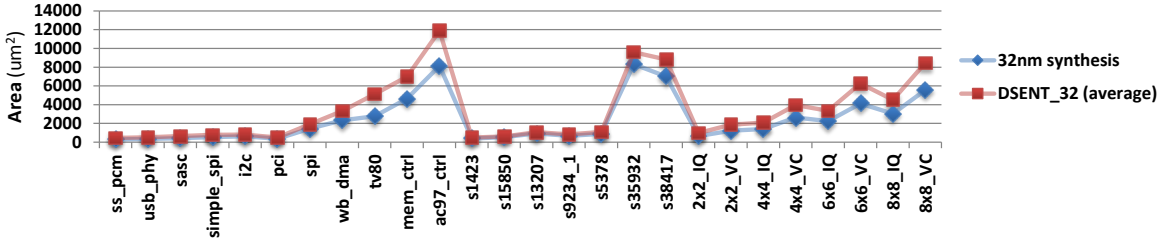


Figure 4.13: Comparing Trends Between the Average Area Estimates of Four DSENT Models Targeting DSENT_32 vs. 32nm Synthesis Results.

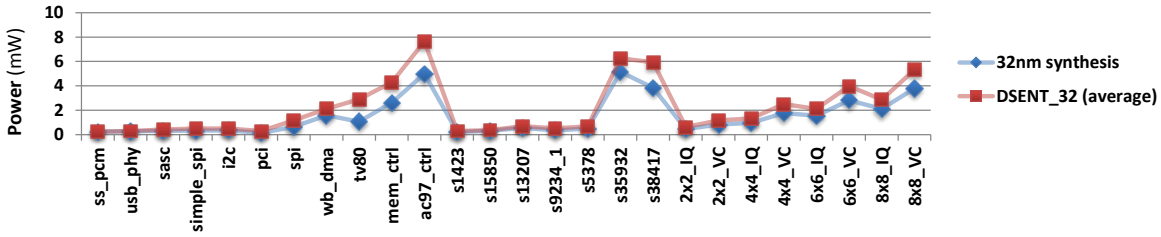


Figure 4.14: Comparing Trends Between the Average Power Estimates of Four DSENT Models Targeting DSENT_32 vs. 32nm Synthesis Results.

design models from RTL synthesis. For this next set of results we assess this premise by synthesizing each design targeting all four commercial standard cell libraries (32nm, 45nm, 65nm, and 130nm) and then processing the synthesized netlists using the DELPHI flow to generate four separate DSENT design models. We then use these four models to target DSENT's 32nm technology model (DSENT_32) to obtain timing, area, and power estimates, shown in Figures 4.9, 4.10, and 4.11.

The fact that these results show low variance and behave consistently across all benchmarks and estimated metrics (timing, area, and power), demonstrates that the DELPHI flow is robust to using different RTL-synthesis standard cell libraries as a proxy for generating a DSENT design model. The extent of "technology independence" becomes more apparent if one considers that these standard cells not only span a wide range of technology nodes, but also come from three different vendors, which vary significantly in their standard cell offerings.

Capturing Power, Area, Timing Trends. After having established the validity of the DELPHI/DSENT approach in the above studies, this last triplet of Figures, 4.12, 4.13, and 4.14, show how well DELPHI captures timing, area, and power trends by juxtaposing results from baseline ("unconstrained") full RTL synthesis targeting a commercial 32nm standard cell library against the average of the DSENT_32 estimates presented in the previous set of results. Note that the DSENT estimates behave consistently and exhibit similar trends with the actual synthesis results. As was

mentioned earlier, it is important to note that the absolute values of the presented data are not meant to match, as the DSENT_32 model does not correspond and was not calibrated to match the commercial 32nm cell library used for synthesis.

DELPHI Usage Example. As an example usage case for DELPHI, consider a hypothetical scenario where a computer architecture researcher is interested in obtaining coarse power trends for two different Network-on-Chip (NoC) router RTL designs as technology nodes scale, including future technology nodes or nodes for which she or he does not have access to. The first router is based on a simple minimal low-performance Input-Queued (IQ) router architecture and the second router is based on a high-performance Virtual-Channel (VC) router architecture.

Assuming DELPHI is used, as a first step, these designs would have to be synthesized targeting some available standard cell library, a process that would take in the order of tens of minutes and would have to be repeated separately for each of the two router variants. The two synthesized netlists are then taken through the DELPHI flow to generate DSENT design models and compile DSENT, which typically takes less than two minutes for both models. These DSENT design models are then used to target five DSENT technology models (65nm, 45nm, 32nm, 22nm, and 11nm) and obtain power estimation results, which are shown in Table 4.2. From these results, it is clear that the high-performance VC router becomes increasingly more attractive (from a power perspective) in future technology nodes, especially at 11nm, where the power difference compared to the IC router has decreased by an order of magnitude and has become negligible.

If the same characterization was to be performed through traditional full synthesis, it would have taken many orders of magnitude longer to perform an equivalent characterization, which would require running 10 synthesis jobs (not to mention the overhead of properly configuring the synthesis environment for five different standard cell libraries). In fact, this technology forecast study may not be possible at all using traditional synthesis, since it would require a prohibitive

investment to create standard cell libraries for future non-existent technology nodes.

	Power Estimates (mW)	
Technology Model	**8x8 IQ Router**	**8x8 VC Router**
DSENT_65	11.88	21.92
DSENT_45	8.65	15.94
DSENT_32	5.49	10.10
DSENT_22	3.34	6.16
DSENT_11	1.03	1.88

Table 4.2: Sample NoC Power Study Using the DELPHI Flow.

Summary. Overall, in our experiments DELPHI exhibits consistent and robust estimates across a variety of benchmarks, standard cell libraries, and technology nodes. It is important to reiterate, that DELPHI, like post-synthesis evaluation, is meant only to expose trends and perform first-order characterization of a design. Our hope is that the results presented in this section will aid in calibrating expectations with regards to DELPHI's capabilities and the extent to which it is able to perform hardware design characterization and capture trends.

4.5 Related Work

To overcome complexity and speed limitations, the architecture community has a history of building and relying on models of varying fidelity to gauge the performance and characteristics of hardware components, such as processors, memories, adders, Networks-on-Chip, etc. Examples of such models include McPat [62], an integrated power, area, and timing modeling framework specific to multicore processor architectures, and CACTI [103], a tool for modeling power, area, and timing characteristics of cache memories. Other tools place particular focus on specific metrics and types of modeling, such as Wattch [22], which focuses on power estimation for microprocessors,

or Orion 2.0 [52], which focuses on power and area for interconnection networks.

Compared to such models, DELPHI combined with DSENT can be thought of as a "meta-model" in the sense that it can be used to generate fast power, area, and timing estimation models based on any existing RTL-based hardware design. While previous models are either high-level, such as McPat, which introduces abstraction errors, or limited to very specific hardware subcomponents, such as CACTI, DELPHI sidesteps these issues and offers both high-fidelity and generality by operating directly at the register-transfer level.

Chapter 5

Nautilus - Guided IP Optimization

This chapter presents Nautilus, which demonstrates the second key principle of Pandora, namely how IP author knowledge can be used to facilitate fast automated design space navigation and application-level goal-oriented IP optimization that is meaningful to the IP users. Nautilus is based on a modified genetic algorithm we developed that allows embedding of IP author knowledge pertaining to the IP design space. This knowledge, coming in the form of "hints", captures the IP author's intuition about how IP parameters relate to the various metrics of interest; the goal is to help steer the optimization search process more quickly toward profitable regions of or directions in the design space. Our evaluations across two IPs show that author-guided instances of Nautilus can achieve the same quality of results up to an order of magnitude faster than a baseline genetic algorithm.

The rest of this chapter is organized as follows. Section 5.1 motivates the need for fast automated IP design space searh and introduces Nautilus. Section 5.2 provides background on genetic algorithms (GAs) and describes how a baseline GA can be used to perform IP optimization. Section 5.3 introduces Nautilus and describes our extensions to GA that incorporates author hints.

Section 5.4 reports our evaluation methodology and experimental results. Finally, we discuss related work in Section 5.5.

5.1 Introduction

The use of IPs has become an indispensable part of modern hardware design flows. Instead of designing every component in a chip from scratch, designers can build entire chips or portions thereof by leveraging existing IP blocks, often developed by third parties. This practice greatly reduces the development time and cost of individual submodules within a larger chip. Over the years, IP blocks, which started as basic primitives (adders and multipliers), have now grown to complex IP blocks and even sophisticated on-demand design generators (e.g., [70, 81, 97]). A single IP block could be responsible for multiple millions of transistors in a chip.

To maintain performance and energy efficiency, today's IP blocks have to be highly parameterized to allow tailoring an instantiation to match specific application and user requirements. The flexibility of user customization however leads to the formation of a vast complex design space that has to be navigated by the IP user. The sheer number and details of the parameters are burdens to handle and a source for error. Moreover, many low-level module-specific parameters are cryptic and incomprehensible to an average IP user who is not already deeply familiar with the specific domain pertaining to the IP (e.g., signal processing, arithmetic units, on-chip interconnects). All of the above result in suboptimal outcomes from an otherwise more-than-capable IP generator.

The Scale of the Problem. Consider the top-level router module of the Stanford Open Source Network-on-Chip Router project [17], a highly-parameterized state-of-the-art router IP block, which exposes 42 parameters (not including any additional sub-module parameters). The design space of a single router already spans multiple billions of possible design points; this does not even consider the countless ways these routers can be arranged and connected to form a network. To give a sense

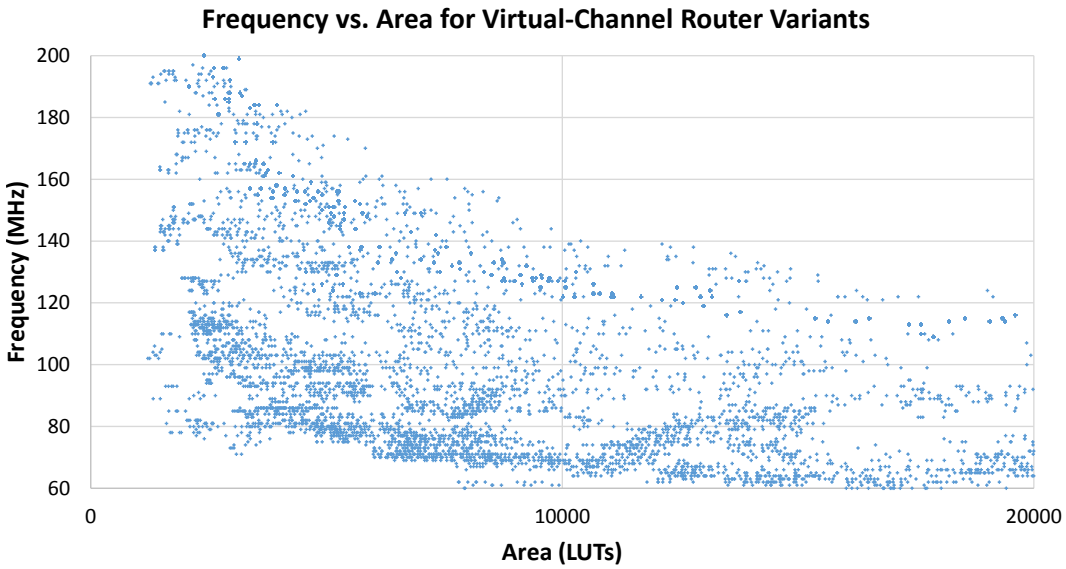


Figure 5.1: LUT Usage and Maximum Frequency for Approximately 30,000 Router Design Points Based on FPGA Synthesis Results.

of what this design space can look like, Figure 5.1 plots FPGA LUT usage and maximum frequency across approximately 30,000 design points—all interchangeable at a functional-level from an IP user's perspective—that belong to a subset of the full design space formed by only 12 out of the 48 parameters.

As another example, we used the publicly available CONNECT NoC IP generator [68] to generate a large collection of different network configurations (router design + network topology) targeting a commercial 65nm technology. Figure 5.2 plots how power and area relate to peak network bisection bandwidth (a network performance metric) across a variety of different 64-endpoint NoC configurations (different colors represent different topology families). Note that, once again, all of these design instances are only a small subset out of the myriad of potential NoC configurations, which are all interchangeable from an IP application perspective; an IP user could pick any of these to satisfy the functional-level connectivity requirements of his or her application. The fact

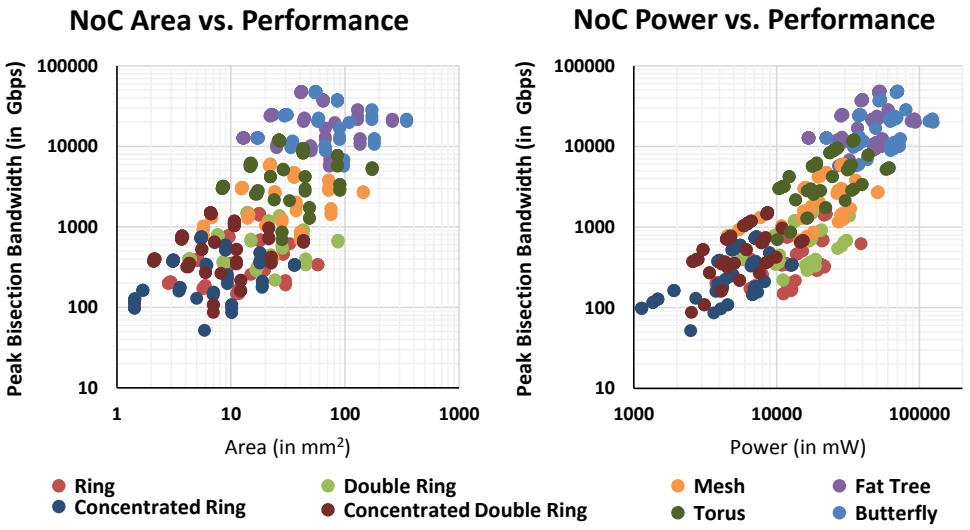


Figure 5.2: Area, Power, and Performance for Various 64-Endpoint CONNECT NoCs Targeting a Commercial 65nm ASIC Technology Node.

that these design points exhibit two to three orders of magnitude of variation across all presented metrics (power, area, performance) highlights the need to be able to efficiently and quickly navigate the design space and the criticality of picking a design point that makes the right trade off and fits the constraints of the project.

Our Solution: Nautilus. The sheer scale and vastness of design spaces of parameterized IPs, such as the ones presented above, and the fact that evaluating each design point can be very costly (requiring long runs of CAD and/or simulation tools, each of which can take several minutes to hours) makes exhaustive search prohibitive and motivates the need for automated, fast, and efficient navigation of the design space. To this end, we present Nautilus, an IP author guided design space exploration engine. This automatic design tuning approach is especially fitting in the context of parameterized IP generators which are already software-driven active objects.

At the core of Nautilus is a modified genetic algorithm (GA) that allows embedding of IP author knowledge pertaining to the IP design space. This knowledge, coming in the form of "hints,"

captures the IP author's intuition about how IP parameters relate to the various metrics of interest; the goal is to help steer the optimization search process more quickly toward profitable directions. In this work, we offer a taxonomy of key classes of IP author knowledge and discuss how to incorporate them into GAs. A particularly important issue in embedding IP author guidance into a GA is to balance the strength of the author's guidance (which will be imperfect) and the stochastic nature of the underlying GA, which is critical for overcoming local optima and for handling design regions that may defy the author's intuition.

We present an evaluation of the Nautilus approach in the context of two hardware IP generators, targeting Networks-on-Chip and fast Fourier transforms (FFTs). The results show that GAs are effective in the automatic tuning of IP parameters when optimizing for a number of key design metrics (e.g., resource usage, efficiency), reducing the number of design points that have to be evaluated by orders of magnitude compared to a naive brute force approach. The results further show that the Nautilus guided GA, with the help of IP author knowledge, can further accelerate the GA search process, reaching the same quality of results as a baseline GA with up to an order of magnitude fewer design points evaluated. This is significant in light of the fact that each evaluation typically corresponds to minutes to hours of EDA execution time depending on the complexity of the IP.

5.2 Background: Genetic Algorithms

Genetic algorithms (GAs) are a class of stochastic adaptive optimization algorithms based on the ideas of evolution, natural selection, and genetics [16]. An initial "population," which consists of randomly selected points in the solution space, is allowed to evolve over a number of generations. During each generation, samples of the population can mutate or combine with each other (crossover) and at the end of each generation, the samples are evaluated and assigned a fitness

score. The most "fit" samples resulting from this process form the next generation.

Applying GA to solve a new optimization problem consists of three main steps. The first step is defining the genetic representation, i.e., expressing each possible solution in the solution space as series of genes, called a genome. Once this mapping has been established, the second step is to define and implement the genetic operators, such as mutation and crossover, that can be applied to the population. Mutation operations modify genes of individual members of the population, while crossover operations combine subsets of the genes of two existing members of the population to produce a new member (i.e., "breeding"). Finally, the third step is defining a fitness (a.k.a. objective) function that assigns a fitness score to each sample in the population. This fitness score is used during the ranking and selection process of the GA that determines which members of the population will survive to form the next generation, where they will have a chance to further mutate and breed.

The quality of results and runtime of a GA algorithm depends on several factors, including:

- **Population Size.** A large initial population increases the solution space coverage, but also increases the amount of work that the algorithm performs during each generation. Since successive generations depend on each other, the population size effectively caps the available parallelism during the evaluation phase of the algorithm that calculates the fitness scores.
- **Mutation Rate.** The mutation rate controls the probability of mutations. A low mutation rate restricts the algorithm to localized search around existing solutions (exploitation), while a high mutation rate allows the algorithm to make larger leaps and potentially overcome local optima to reach unexplored portions of the design space (exploration). As is also the case with other stochastic optimization algorithms, striking a good balance between exploration and exploitation is an important aspect of tuning a GA to a particular problem.
- **Fitness Function.** The fitness function is used to assign a score to each sample in the pop-

ulation which is used at the end of each generation for ranking the available samples and selecting the ones that will make it on to the next generation. The fitness function is used to guide the evolution process and is one of the most central elements of a GA. Not only is it used to pick different optimization goals, but it can also be adapted to constrain the algorithm to only explore specific portions of the solution space (e.g., by assigning very low scores to solutions lying in regions of the design space that are not of interest or should be avoided).

GAs for IP Optimization. In this work we use genetic algorithms to automatically tune IP parameters for a given optimization goal. In the context of IP optimization, the initial population consists of potential design instances with different low-level parameter configurations, each corresponding to a distinct point in the design space. Mutations correspond to changing a parameter value, and crossovers combine parameter settings of different samples in the design space. Depending on the type of IP and the metric being optimized, "fitness" can take many different forms and even incorporate multiple metrics, which allows for great flexibility. For instance, in the case of a Network-on-Chip router, fitness can correspond to FPGA resource usage, throughput, energy efficiency, or even a custom-defined composite function that can combine these metrics in arbitrary ways.

Evaluating GAs. We are mainly interested in two metrics when evaluating a GA: (1) runtime, i.e., how long it takes for the GA algorithm to run, and (2) quality of results, i.e., how good of a solution the GA finds. In the context of IP parameter optimization, runtime is directly tied to the number of fitness function evaluations, since each evaluation requires running computationally expensive CAD tools (e.g., FPGA/ASIC synthesis or place-and-route tools) and/or simulations. The quality of results can be either expressed as the fitness score of the best solution in the population or as a percentage with respect to the best scoring sample for the specific optimization (if that is known). It is important to understand that the goal of Nautilus is not to find the absolute best design

point. In real usage scenarios, we want to help an average IP user find a good design point that is within some threshold of what the IP generator can offer. The goal is to do much better than what an average IP user could do by trial-and-error or, worse yet, by taking the default. Thus, for evaluation, we examine how many distinct design points have to be evaluated (at the cost of up to hours each) in order to reach a desired quality of results.

5.3 Incorporating Author Knowledge

Overview. Traditional GAs, such as the baseline GA described above, explore the design space in a random fashion, assuming no knowledge of how individual genes or parameters relate to optimization goals or affect the fitness of the samples in the population; each point in the design space is equally likely to be visited. This oblivious nature of the baseline GA makes it ideal for exploring unknown or highly unpredictable non-convex solution spaces. Compared to naive brute force design space exploration approaches, such as exhaustive search or random sampling, using a GA already marks a significant improvement in terms of the number of design points that have to be evaluated until a desirable one is found.

The baseline GA might be a good approach when an IP user is dealing with an IP he or she is unfamiliar with. However, it is wasteful to forgo the wealth of knowledge the *author* of the IP possesses about the design space. In Nautilus, we want the IP authors to embed knowledge about the design space as an integral act of creating an IP. This IP author knowledge can drastically improve the GA search and optimization process, even if this knowledge is limited and comes in the form of partial or approximate hints.

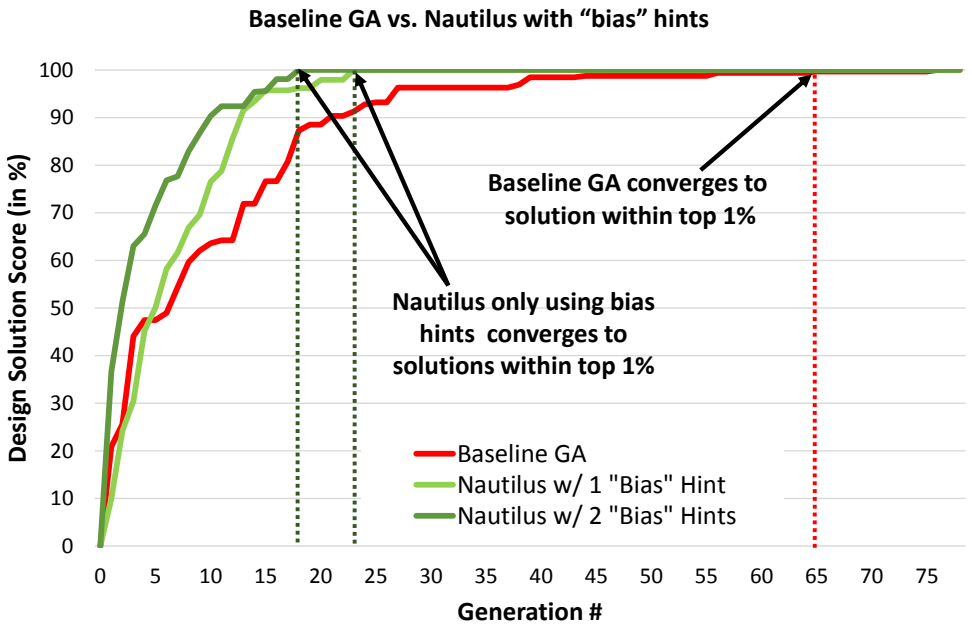


Figure 5.3: Baseline GA vs. Nautilus Only Using 1 or 2 “Bias” Hints.

5.3.1 Nautilus Hints

To implement Nautilus we modified an existing open-source GA implementation, called PyEvolve [3], and extended it to support various forms of IP author or domain-expert hints pertaining to all or a subset of IP parameters. The idea behind these hints is to skew the search process towards specific regions of the design space.[1]

To illustrate the effect of hints, Figure 5.3 compares how the “fitness” (average of 20 runs) of a set of FFT hardware designs evolves over time using a baseline GA and Nautilus using a varying number of hints. In this example the baseline GA takes 56 generations to find a solution within the top 1%, while Nautilus can reach the same quality of results within 15 to 23 generations, depending on how many hints are provided.

Supported Classes of Hints. Below we describe some of the supported classes of Nautilus

[1]Note that these hints are incorporated in a probabilistic manner, maintaining the stochastic nature of GA, which is still free to explore the full design space and overcome local optima.

hints and explain their effect with respect to the baseline GA behavior and how they are used to guide the search process. It is important to keep in mind that the goal of Nautilus is to provide infrastructural support for different classes of hints; the exact instances are specific to the given IP generator and metric. The IP author's specialized knowledge allows him/her to provide hints as part of creating an IP. The IP author is free to supply as many or few hints as desired; if it lacks sufficient hint information, Nautilus will fall back to using default values or employ the baseline GA. Unless specified otherwise, each hint needs to be supplied per metric of interest and per IP parameter, i.e., the IP author's knowledge about how IP parameters relate to high-level metrics is captured by a vector of hints, such as the ones described here.

- **Importance:** The importance hint assigns values from 1 to 100 to each parameter that captures how drastically the parameter is expected to affect the metric being optimized. In Nautilus, the parameters' relative importance skews which genes are more likely to be picked for mutation during a genetic operation. This accelerates the algorithm, because it can directly focus on the parameters most likely to matter, wasting less time experimenting with others.
- **Importance Decay:** The "importance decay" hint takes a value from 0 to 1 and allows Nautilus to gradually adjust the relative importance differences of parameters. The idea is to allow for the importance of some parameters to "decay" over time (at the rate defined by the "importance decay" hint) as the algorithm progresses through generations.

  This hint can be used to introduce a *temporal aspect* to Nautilus, allowing it to initially focus on parameters believed to be "important" to coarsely navigate towards promising regions of the design space and then gradually shift focus to experimenting with less "important" parameters to perform more localized fine-tuning within those promising regions.
- **Bias and Target:** While the two previous hints affect which genes get selected to mutate, the

bias and target hints affect the values that these genes will be assigned during each genetic operation. Each parameter can either be assigned a bias hint or a target hint (but not both). Bias takes values from -1 to 1 for each parameter and captures the correlation between the parameter and the metric being optimized. A positive (negative) bias means that increasing the parameter will increase (decrease) the metric being optimized. The target is a more direct hint that allows the IP author to specify that good (or balanced) solutions are known to cluster around a particular value. Bias and target can be used to guide Nautilus in a coarse manner, e.g., towards a specific direction, or in a more fine grained manner, i.e., towards a specific region in the design space. The bias and target hints cause Nautilus to behave in a much more "directed" fashion compared to a baseline GA; when correctly set by an expert who understands the design space, they can allow the algorithm to find efficient regions of the design space quickly.

- **Confidence:** The confidence hint can be viewed as a high-level knob that controls how much trust Nautilus should place in the author hints. In other words, this determines how "guided" the algorithm will be. Setting low confidence values will make the algorithm behave more similarly to the baseline GA, while setting high confidence values along with strongly-guided hints (e.g., setting target values of high bias values) will cause the algorithm to perform very directed optimization that starts to resemble convex optimization methods such as gradient descent. Confidence allows Nautilus to more easily incorporate heuristics or even low-confidence experimental hints that might be purely based on "gut feeling" without breaking the search or optimization process.

Finally, in addition to the hints described above, Nautilus includes some additional low-level auxiliary settings that, e.g., determine the "stepping" of the algorithm or define ordering relationships among values that a specific parameter can take (e.g., order different allocator options with

respect to clock frequency or area). These settings control subtle aspects of the algorithm or are used to ensure smooth operation of the algorithm under non-trivial design spaces (e.g., sparsely populated design spaces that included infeasible points or regions).

In our targeted usage scenario, these hints are calibrated by the IP author during the IP development phase and are packaged and provided along with Nautilus as part of the IP (preferably in the form of an IP generator). However, in the absence of an IP "expert" these hints can also be set directly by a knowledgeable IP user. Additionally, an IP user could try sweeping each IP parameter independently and then observe how the various metrics of interest respond to estimate approximate hint values.

5.4 Evaluation

We evaluate Nautilus with varying degrees of guidance against a baseline GA using a publicly available [17] highly-parameterized state-of-the-art Virtual-Channel Network-on-Chip (NoC) router IP, as well as the Spiral FFT IP design generator; for the remainder of this section we will refer to these IP as "NoC" and "FFT".

5.4.1 Methodology

As a preparatory step, we map a large portion of each IP's design space consisting of comparable—from an IP user perspective—design instances. The resulting datasets consist of approximately 12,000 design instances for the FFT IP (varying 6 parameters) and 30,000 design instances for the router IP (varying 9 parameters). We run FPGA synthesis and/or simulations for each design instance to characterize it with respect to hardware implementation metrics (e.g., area, frequency), metrics specific to the IP domain (e.g., SNR values for the FFT IP), and composite metrics (e.g., throughput-per-LUT). FPGA synthesis results are obtained using Xilinx XST 14.7 targeting a Xil-

inx Virtex-6 LX760T FPGA (part xc6vlx760). This characterization step was done "offline" (using a dedicated cluster with 200+ cores running non-stop for about 2 weeks) to produce the datasets that we used to evaluate Nautilus.

Both Nautilus and our baseline GA implementation are based on modified versions of the PyEvolve genetic algorithm framework [3]. For each IP we define queries (e.g., optimize for throughput/area) and then compare the baseline GA with Nautilus in terms of the computational cost to run the query and quality of results (with respect to the given query). Unless otherwise noted, for both the baseline GA and for Nautilus, we use an initial population of 10 samples, a mutation rate of 0.1 (this means that each gene that belongs to a sample has a 10% chance of mutating during each generation), and run for 80 generations. Results are averaged over 40 runs for each experiment to compensate for the "noisy" nature of the stochastic process.

In the case of FFT, the Nautilus engine is expert-guided as the hints are provided from a member of the Spiral development team. For the NoC IP we estimated hints by synthesizing 80 designs (less than 0.3% of the design space) and observing trends; this is equivalent to an IP user (or some other non-expert) supplying the hints using limited empirical knowledge or gut intuition about the IP.

5.4.2 Results

NoC. We first look at the NoC results, where Nautilus is guided by non-expert hints. We compare two Nautilus variants ("strongly guided" and "weakly guided"[2]) against the baseline GA. Figure 5.4 shows the result of a query aimed at finding designs in the NoC design space that can achieve the highest frequency. The y-axis shows the maximum frequency for the best sample in each algorithm's population and essentially captures how the quality of results changes as the algorithms progress. The x-axis shows the cumulative number of synthesis jobs needed for each

[2]The strongly and weakly guided lines differ only in the "confidence" hint.

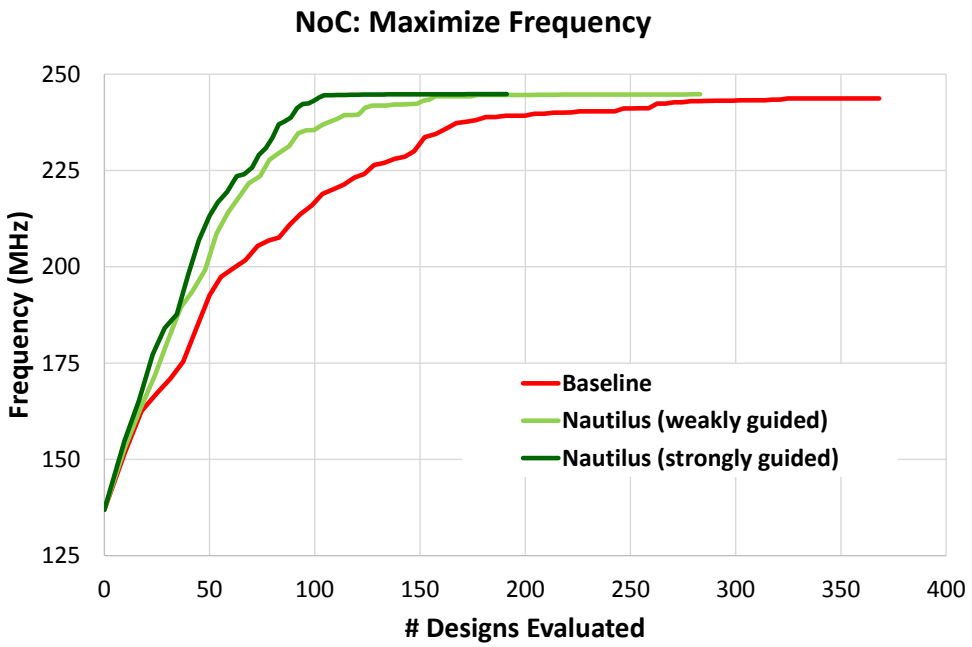


Figure 5.4: Maximizing Frequency in the NoC Design Space.

of the three techniques, reflecting the computational cost of the query. All three techniques are run for 80 generations of the GA. Note however that the Nautilus lines require fewer designs to be synthesized (because the GA revisits previously-synthesized results as it converges), allowing their lines to stop after fewer designs. Both the strongly-guided and weakly-guided configurations of Nautilus approach good solutions much faster than the baseline GA. The baseline GA requires about 2.8x and 1.8x the number of synthesis jobs to converge to a solution within 1% of the best solution.

Figure 5.5 shows the results for our second NoC query, which aims at minimizing the area-delay product of a design. Here, results are shown only for the first 20 generations, because both techniques converged to the optimal solution within this time. While our previous query only used hints related to frequency, this query also incorporates hints related to the importance and bias of IP parameters that affect area, such as virtual-channel buffer depth. In this case, Nautilus achieves

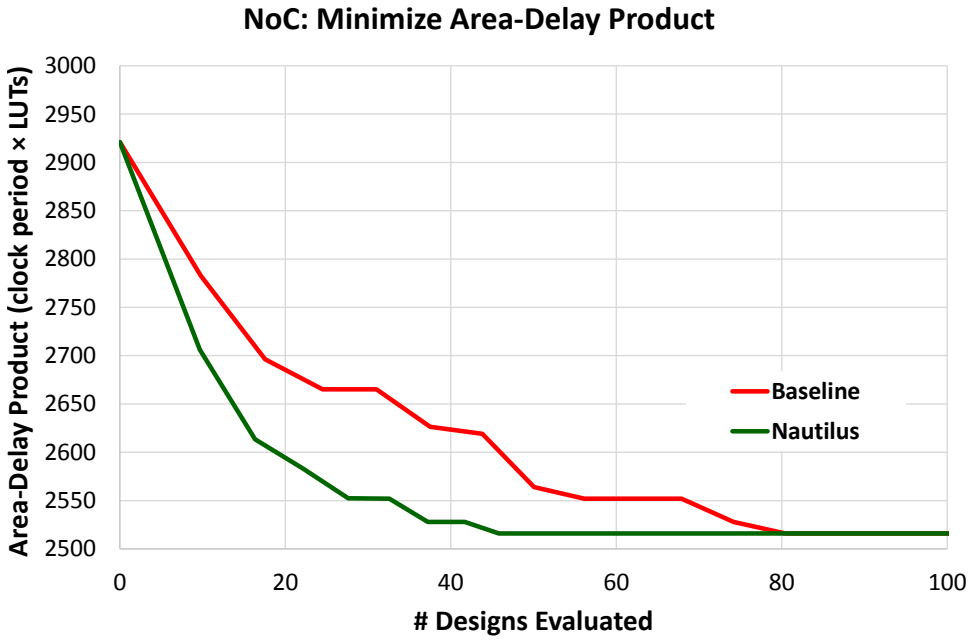


Figure 5.5: Minimizing the Area-Delay Product in the NoC Design Space.

similar quality of results with about half the number of synthesis runs required by the baseline. It is interesting to note that even the non-expert guided Nautilus performs significantly better than the baseline, offering much better quality of results for the same computational cost or the same quality of results after significantly fewer synthesis jobs.

FFT. We next turn to the FFT results. As mentioned earlier, in this case Nautilus is "expert-guided", i.e., a developer of the FFT IP generator sets the hints. Figure 5.6 shows the result of a query aimed at minimizing the number of LUTs used by an implementation from the FFT dataset.

Here we see that all three methods eventually converge on a same result (about 540 LUTs), but the Nautilus designs converge much more quickly and require far fewer designs to be synthesized: the strongly guided Nautilus strategy converges on the optimal design using an average of 101 synthesis runs, while the baseline GA requires 463 designs to be synthesized (on average) to reach an equivalent result. If we relax the goal to 1,071 LUTs (twice the minimum), we see that the

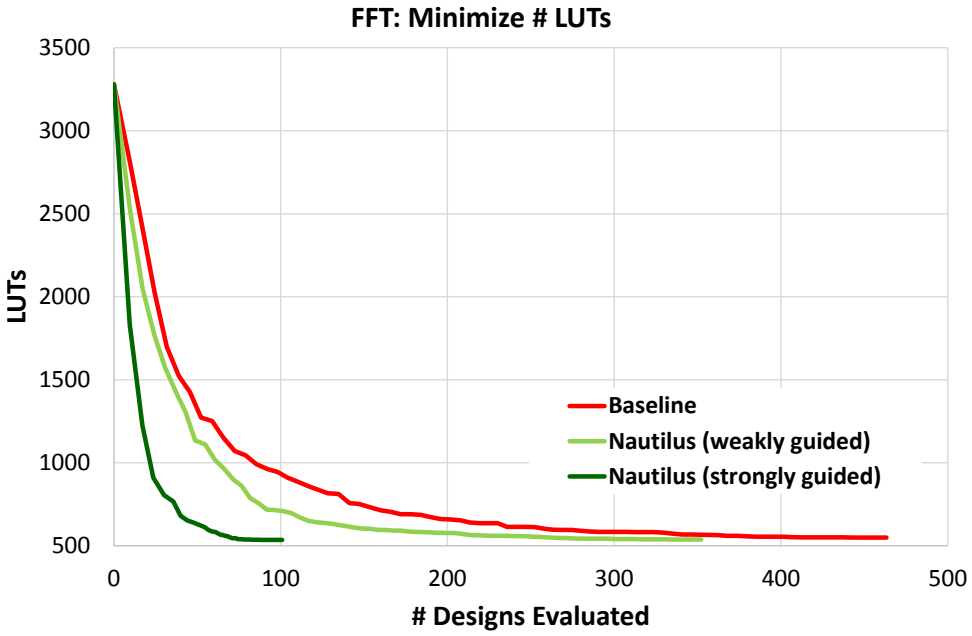


Figure 5.6: Minimizing the Number of LUTs in the FFT Design Space.

strongly guided Nautilus technique is able to meet the goal synthesizing 23.6 designs (on average), while the baseline GA requires synthesizing an average of 78.9 designs to reach the same quality result.[3]

Figure 5.7 aims to search the FFT design space for a design that uses a composite metric to maximize the ratio of throughput to logic area consumed: throughput in million samples per second (MSPS) divided by the number of LUTs. In this case, once again the strongly and weakly guided Nautilus variants find significantly better solutions in less time; for example, the strongly guided Nautilus strategy is able to reach 1.45 MSPS per LUT using 61.6 synthesis runs (on average), while the baseline GA requires more than 8x synthesis runs (501.4 on average) to reach the same value. Moreover, Nautilus is able to reach high-quality solutions exhibiting more than 1.5 MSPS per LUT, which the baseline is never able to approach even after having explored a much larger

[3]For comparison, if random sampling was used, it would take on average 11,921 synthesis runs to find a design meeting this goal.

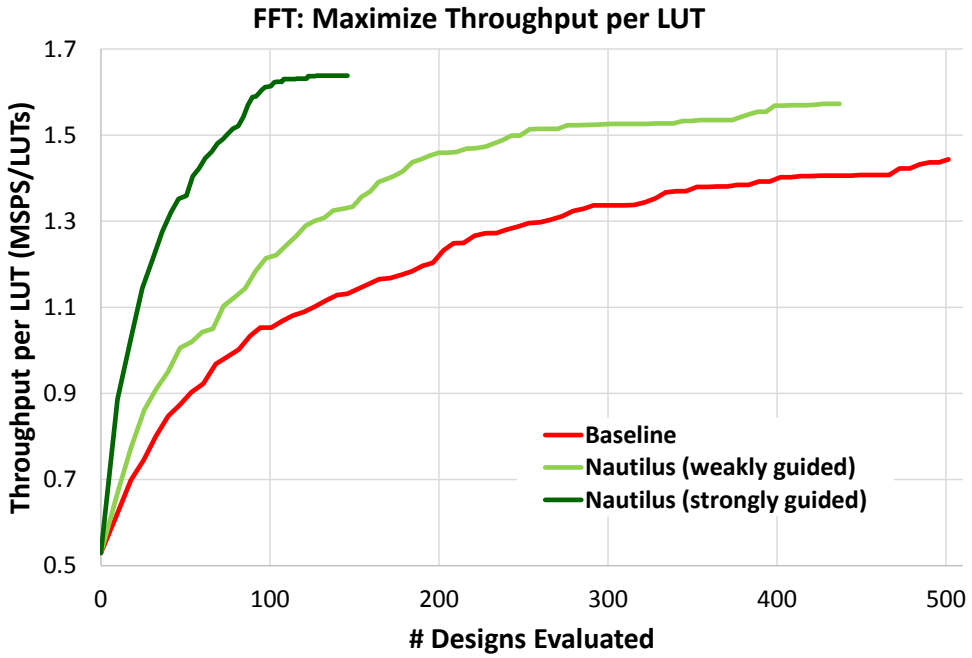


Figure 5.7: Maximizing Throughput per LUT in the FFT Design Space.

portion (>5x) of the design space.

Overall, Nautilus outperforms the baseline GA, both when guided by a non-expert (NoC), and especially when given IP-expert guidance (FFT). Nautilus consistently produces higher quality of results at much lower computational cost across multiple optimization queries in two separate IP domain spaces.

5.5 Related Work

Stochastic Optimization in EDA. Stochastic methods such as genetic algorithms have been used for a variety of aspects of hardware exploration, from the circuit level (e.g., low-level VLSI layout optimization [40] and yield-aware circuit sizing [104]), to the system level (such as [94] and [38], in which genetic algorithms are used to optimize aspects of hardware-software co-design). [80] uses GAs to perform exploration across a space of closely-related processor systems,

which can be evaluated in only a few seconds per design. Other approaches focus on high-level synthesis, such as [23, 57], which use genetic algorithms and Monte Carlo methods (respectively) for HLS optimization. Lastly, simulated annealing has long been used in physical design automation problems (e.g., [13]).

Active Learning. Another important class of related work is research on *active learning*, a family of learning techniques that iteratively evaluate potentially useful points within a set. Several recent works [24, 54, 115, 116] use active learning techniques to model the entire Pareto-optimal set of design points across a multi-objective space; specifically, [115, 116] consider costs and performance of generated IPs. These approaches differ from our work in that they aim to understand all design points that give a Pareto-optimal trade-off among any of the design characteristics. As we are considering design spaces with tens of thousands of possible designs and on the order of ten cost and performance metrics, this generalized approach would become extremely difficult; instead we aim to provide a system that can answer a given query about this complicated design space using as few synthesis steps as possible.

Chapter 6

IRIS - IP Instrumentation & Introspection

This chapter presents IRIS, a flexible systematic instrumentation and introspection framework that demonstrates the third key principle of Pandora, sophisticated IP instrumentation. IRIS allows for fast and efficient hardware debugging and monitoring, and enables system-level visibility and analysis. It consists of (1) a library of parameterized "logic probe" modules that can be introduced by the designer into a register-transfer-level design to count, sample, and record traces of events, as well as capture distribution of scalars (e.g., buffer occupancy, transaction latency), (2) a software post-processing engine that analyzes and summarizes the data collected by the instrumentation components, and (3) a visualization engine that presents the analyzed instrumentation data through an easy-to-navigate graphical user interface that highlights key metrics of interest and includes dynamically generated interactive charts. The IRIS post-processing and visualization engines can help capture and identify higher level system behaviors and design issues, which not only help the designer during the development cycle but also enable the end-customer of a system gain high-

level insights about the system's operation (e.g., identify a bottleneck or hotspot). Because the instrumentation components are implemented as synthesizable modules, they can be retained in simulation, FPGA prototyping, and even in the final product. We demonstrate IRIS through case studies in the context of the CONNECT Network-on-Chip generator. Our evaluation shows that IRIS only has a small impact on simulation speed even for moderately instrumented designs. When implemented in hardware, IRIS instrumentation components make efficient use of resources and allow for fine-grain tuning to adjust the area and timing overhead.

The rest of this chapter is organized as follows. Section 6.1 introduces IRIS. Section 6.2 describes classes of IRIS instrumentation components, and the IRIS post-processing and visualization engines. Section 6.3 evaluates IRIS with respect to hardware implementation cost and simulation speed overhead, and also demonstrates the use of IRIS instrumentation in the context of CONNECT NoCs. Finally, Section 6.4 covers related work in the areas of hardware instrumentation, design-for-test (DFT), and design-for-debug (DFD).

6.1 Introduction

Traditional approaches to hardware monitoring and debugging can take many forms depending on the development phase and environment (e.g., RTL simulation vs. FPGA deployment). Common approaches used by IP users include adding assertions and display statements to the source HDL, studying waveforms for a hand-picked subset of the design signals, or even instrumenting a simulated design or actual hardware using logic analyzer tools (e.g., Xilinx Chipscope [112], Altera Signaltap [14]) to directly probe signals and monitor their state at runtime.

While such approaches and tools can be powerful in the hands of expert designers looking to observe or debug isolated small portions of a design in detail, they also have several drawbacks, especially when dealing with large complex systems: (1) they are tedious, low-level, and have to

be performed from scratch in an ad-hoc manner, (2) they have to be repeated and readjusted for each submodule and as a design progresses through different phases of the development cycle, and (3) they do not scale and are not able to capture higher level behaviors and identify system-level issues as designs become larger and contain multiple complex large interacting IPs.

In this chapter, we present IRIS, a flexible systematic instrumentation framework that is aimed at accelerating hardware debugging and monitoring, and enabling system-level visibility and analysis. IRIS combines a library of parameterized hardware instrumentation components that monitor and collect information about a hardware design with software tools that post-process and visualize the collected data. To maximize flexibility and minimize repeated effort, hardware and software in IRIS work synergistically and allow a designer to instrument a design once and then dynamically select which portions and to what extent will be retained and implemented in hardware, and which portions of the instrumentation will be handled in software.

6.2 The IRIS Framework

The IRIS framework consists of three parts:

1. The **IRIS Instrumentation Library (IIL)**, which is a collection of highly parameterized synthesizable instrumentation components that provide different implementation options (e.g., hardware-resident, simulation-only) and can be spatially and temporally extended.
2. The **IRIS Introspection Engine (IIE)**, which is a software-based tool designed to process, analyze, and summarize event logs generated by the IRIS instrumentation components.
3. The **IRIS Visualization Engine (IVE)**, which is responsible for visualizing and presenting the post-processed summarized collection of instrumentation data through an interactive web-based graphical user interface.

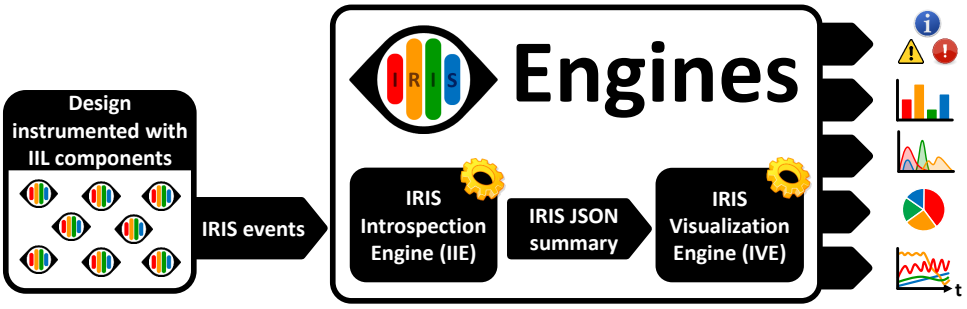


Figure 6.1: IRIS Usage Flow.

Using IRIS. Figure 6.1 shows a high level diagram of what the IRIS flow looks like. To use IRIS, an IP has to first be instrumented using IRIS components from the IIL (or, as we explain later, other components that adhere to the same spec). Then, as the hardware design is active, the instantiated IRIS components generate events, which are processed by the IIE. The IIE fully replicates the state of the IIL components present in hardware, and allows the user to define additional components that build on top of the ones defined in the hardware. The IIE produces a JSON-formatted summary of the information it gathered and processed from the IRIS components, which is passed to the IVE. The IVE handles the visualization and generates a web-based interface that enables viewing statistics, charts, and events pertaining to the instrumented system.

When designing the IRIS framework, we conscientiously chose to decouple and standardize the interfaces between the IIL, IIE, and IVE to allow for greater flexibility, and reusability. In particular, the IRIS event format, as well as the IRIS JSON format are both open and allow for using each part of IRIS independently. For example, a user is free to use his own instrumentation components, as long as they generate IRIS-compatible events. Similarly, a user is free to just use the IRIS visualization engine standalone, by producing a JSON summary that conforms to the one defined by IRIS.

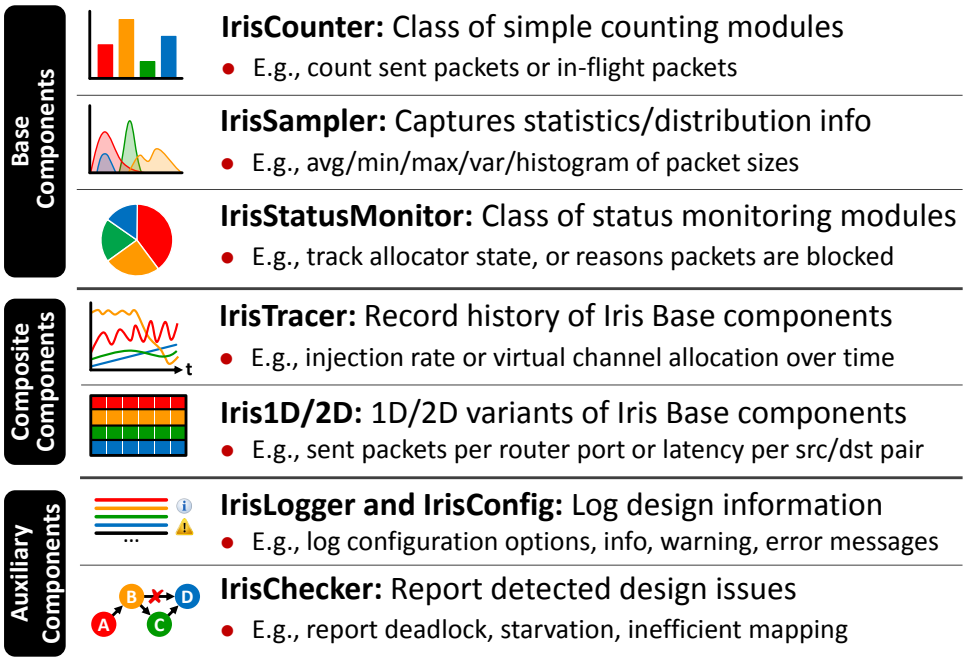


Figure 6.2: Overview of IRIS Components.

The rest of this section describes the IRIS Instrumentation Library (IIL), the IRIS Introspection Engine (IIE), and the IRIS Visualization Engine (IVE).

6.2.1 IRIS Instrumentation Library

The IRIS Instrumentation Library (IIL) defines a set of instrumentation components logically organized depending on their function, intended use, and visualization outcome. For each one of these components, the IIL provides highly parameterized RTL implementations in both Bluespec and Verilog. Each type of IIL component supports a set of methods and generates a set of events. IIL components are organized in 3 groups, **base**, **composite**, and **auxiliary** components. Figure 6.2 provides an overview of the three IIL component groups.

The base IIL components are the IrisCounter, used for counting events in a system, the IrisSampler, used for capturing distribution characteristics of scalar collections, and the IrisStatusMonitor,

used to keep track of how pieces of hardware spend their time. The Composite IIL components spatially and temporally extend the base IIL components to keep relevant information grouped during the post-processing and visualization steps, or to observe hardware behaviors over time. They include the Iris1D and Iris2D composite components, which allow creating 1D and 2D collections of IIL base components (e.g., record number of packets per source-destination pair), as well as the IrisTracer composite components, which allow tracing the state of IIL base component over time (e.g., monitoring average network load per router, or variance in packet latency over time). Finally, the auxiliary components include the IrisLogger, IrisConfig, and IrisChecker, which are primarily used for communicating information about the instantiated instrumentation components, system configuration, and operation of a design to the IRIS introspection and visualization engine.

Configuration and Parameterization. All IRIS components support a common set of configuration options that control their behavior in simulation, hardware implementation, logical grouping, and visualization. In addition, each type of IRIS base component also has its own individual configuration options that are specific to that type of component. Finally, all IIL components allow for structural parameterization (e.g., width of IrisCounter, number of histogram bins of IrisSampler, dimensions of Iris1D/Iris2D collections, history length of IrisTracers, etc.) that affects the hardware implementation cost and allows tailoring the component to a given environment and application requirements. A summary of the configuration and parameter options that are common across all IRIS base components are shown in Table 6.1.

IrisCounter. The IrisCounter is the simplest IIL component and is intended to be used for counting events or keeping track of a value in a system, e.g., number of sent packets, pending memory responses, in-flight instructions, pipeline stalls, etc. The full set of configuration parameters, features, and methods supported by the IrisCounter are shown in Table 6.2. Depending on how an IrisCounter is configured it can support a subset of these methods. In its simplest config-

IRIS Parameter Name	Description	Default Value
showWarnings	Enables displaying of warnings (only affects simulation)	TRUE
stopOnWarnings	Controls is system will halt on warnings	FALSE
visualize	Controls if component should be included in visualization	TRUE
dumpEvents	Controls if component should produce events for the IIE	TRUE
buildHardware	Controls if component is implemented in actual hardware	TRUE
buildAccessInterface	Controls if HW interface for reading/writing component state is implemented	TRUE
periodicallySyncState	Controls if component state should be periodically synced with IIE	FALSE
syncAtCycles	Controls how frequently component should be synced with IIE	0
numDimensions	Set to the number of dimensions if component is part of 1D or 2D collections	0
sizeX	Indicates size of 1D/2D collection	0
sizeY	Indicates size of 1D/2D collection	0
i	Indicates position of component instance within 1D/2D collection	0
j	Indicates position of component instance within 1D/2D collection	0
debugLevel	Controls verbosity of component (0:None, 1:Low, 2:Medium, 3:High)	0

Table 6.1: Common IRIS Component Parameters.

uration the IrisCounter only supports the "incByOne" method and acts as a simple accumulator. Additional optional methods include "incByN", "decByOne", and "decByN", as well as a "clear", "write", and "readValue" methods to directly clear, overwrite, or read the counter value. The set of enabled methods affects the hardware implementation cost, which is reported in Section 6.3.

IrisSampler. The IrisSampler IIL component samples scalar values and can keep track of statistical distribution information, such as average, minimum, and maximum value, as well as variance, and even record a histogram that captures a more detailed representation of the actual distribution. It can either sample an existing instance of an IrisCounter or directly sample values in a system, such as the occupancy of a buffer or the latency of packets betwen two endpoints in the system. Table 6.3 shows the configuration parameters, features, and methods supported by the IrisSampler IIL component. In addition to the basic "addSample" method, the IrisSampler can also support optional methods that allow accessing and manipulating its state. As was also the case with the IrisCounter, hardware implementation cost is directly tied to the configuration and subset of enabled methods.

IrisStateMonitor. The IrisStateMonitor (a.k.a. IrisEventTracker) IIL component is initialized

IrisCounter Parameters	Description	Default Value
bitWidth	Bit width of counter	32
IrisCounter Features		**Default Value**
allowIncByN	Allow incrementing counter by arbitrary integers	FALSE
allowDecByOne	Allow decrementing counter by one	FALSE
AllowDecByN	Allow decrementing counter by arbitrary integers	FALSE
isSigned	Treat the stored value as signed	FALSE
allowClear	Allow clearing the counter	FALSE
allowRead	Build read interface	FALSE
allowWrite	Build write interface	FALSE
IrisCounter Methods	**Description**	
incByOne	Increments counter by one	
decByOne	Decrements counter by one	
incByN(inc_by_n)	Increments counter by inc_by_n	
decByN(dec_by_n)	Decrements counter by dec_by_n	
clear	Clears counter	
write(stat_t value)	Overwrites counter value	
stat_t readValue	Returns counter value	

Table 6.2: IrisCounter Parameters, Features, and Methods.

with a set of distinct states (or events) and then keeps track of the state in which a part of a system is spending its time in. For example, in a Network-on-Chip setting, this could be useful for monitoring the state of a router (e.g., idle, sending, blocked) or composing a detailed breakdown of the reasons that a packet might not be making progress (e.g., blocked behind higher priority packet, not picked by allocator, blocked due to flow control, etc.). The IrisStateMonitor states are assumed to be mutually exclusive by design to allow for an efficient hardware implementation. The main method is "setState" (or "addEvent"), and as with the other two base IIL components, the IrisStateMonitor offers some configuration parameters and optional features that adjust its hardware implementation cost. Table 6.4 shows the configuration parameters, features, and methods supported by the IrisStateMonitor IIL component.

Iris1D and Iris2D. The Iris1D and Iris2D are composite IIL components that allow logically organizing instances of the Iris base ILL components described above in 1D and 2D groups. This grouping is convenient when monitoring collections of things (e.g., a set of processors) and working

IrisSampler Parameters	Description	Default Value
bitWidth	Bit width of sampled data type	32
maxSamples	Specifies maximum number of sampled data points	1024
numBins	Specifies number of histogram bins	8
IrisSampler Features	**Description**	**Default Value**
keepMinMax	Maintains minimum and maximum values of sampled data	TRUE
keepSqTotal	Maintains sum of squared samples to calculate variance	TRUE
isSigned	Treat the samples as signed	FALSE
allowClear	Allow clearing the counter	FALSE
hasBins	Specifies if component maintains histogram bins	FALSE
IrisSampler Methods	**Description**	
addSample(stat_t sample)	Add sample to the collection of samples	
clear	Clears all IrisSampler state	
readTotal	Returns sum of all samples	
readNumSamples	Returns counter of all samples	
readSqTotal	Returns sum of squares of all samples	
readMin	Returns minimum sampled value	
readMax	Returns maximum sampled value	
getMaxBinId	Returns highest available bin id	
readBin(bin_id)	Returns value of bin bin_id	

Table 6.3: IrisSampler Parameters, Features, and Methods.

with pairs or combinations (e.g., keeping track latency per source-destination pair, or counting number of packets per router and port). The grouping information is also used later on by the IVE to pick appropriate visualizations for 1D and 2D collections of IIL components. The supported methods are similar to those used for the base IIL component, but also contain a single or a pair of indices that indicate an element's position within the 1D or 2D array. Hardware cost is directly proportional to the size across each dimension.

IrisTracer. While the Iris1D and Iris2D IIL components allow for spatial grouping of IIL base components, the IrisTracer allows for temporally extending the base IIL components by coupling them with a trace buffer of configurable granularity. This is useful for keeping track of a metric as it changes over user-defined quantums of time, e.g., injection rate or average packet latency. The hardware implementation cost of the IrisTracer components is directly tied to the history length of the trace buffer.

IrisStateMonitor Parameters	Description	Default Value
numStates	Specifies the number of distinct available states	8
maxEventCount	Specifies maximum value for each state count	4294967296
IrisSampler Features	**Description**	**Default Value**
specifyStates	Allows specifying custom state names	FALSE
stateNames	Used to ”name” the different states to be monitored	””
allowClear	Allow clearing all state in the component	FALSE
IrisStateMonitor Methods	**Description**	
addState(state_id)	Indicate current monitored state is state_id	
clear	Clears all IrisStateMonitor state	
readStateCount(state_id)	Reads counter associated with state_id	

Table 6.4: IrisStateMonitor Parameters, Features, and Methods.

IrisLogger and IrisConfig. The IrisLogger and the IrisConfig are auxiliary IIL components. An IrisLogger component allows a system to generate a stream of messages, which can belong to one of three categories, “Info”, “Warning”, or “Error”. The IrisConfig component captures configuration options of a system in a structured manner, represented as collections of key-value pairs. Both the IrisLogger and IrisConfig are purely simulation-based components and generate log information that is later used by the IIE and IVE engines.

IrisChecker. The IrisChecker is an auxiliary IIL component that acts as a shell for logic that implements monitors that checks for and reports suboptimal or improper operation of the system (e.g., detect deadlock, report starvation, etc.). The hardware cost of an IrisChecker depends purely on the complexity of the logic that performs the system monitoring and checking, and which is up to the designer to implement.

Appendix C provides an overview of the IRIS event specification, as well as an example of what the output of an IRIS-instrumented CONNECT-generated NoC design looks like.

6.2.2 IRIS Introspection Engine

The IRIS Introspection Engine (IIE) processes events generated by the IIL components described above. It is written in Python and contains fully equivalent software implementations of

each IIL component, which are used to mirror the state of the instrumentation components as an instrumented hardware design is running. The user is free to instantiate additional software-only IIL components that can build on top of the hardware-resident IIL components. The user is also free to execute arbitrary code in callback functions for each type of event to build even more sophisticated software-based instrumentation.

When simulating a hardware design, the IIE runs in tandem with the RTL simulator and processes events on the fly, as they are produced by the various IRIS instrumentation components. In a multi-core system, when the IIE and RTL simulator run as separate threads on different cores, impact on simulation speed is minimal. Once the event stream is over or if instructed by the user, the IIE generates a JSON-based summary of the state of all instrumentation components, including the ones that are part of the hardware, as well as the ones that are only defined in software. This JSON-based summary is then used by the IVE to generate the visualizations.

6.2.3 IRIS Visualization Engine

The IRIS Visualization Engine is a separate Python-based component that parses the JSON-based summary produced by the IIE to generate a web-based interface that presents the gathered statistics. Each IIL component gets visualized depending on its type. IrisSamplers produce a table with statistical information, as well as a dynamic histogram that allows juxtaposing different IrisSampler instances. IrisStatusMonitors generate pie charts indicating the breakdown for each state. Iris1D and Iris2D composite components generate collections of bar graphs that can be dynamically organized by dimension. Finally, IrisTracers generate a timeline chart that allows zooming in and out, smoothing over a configurable window of time, and enabling/disabling individual datasets. Examples of the visualization output are shown in Section 6.3.

6.3 Evaluation

This section evaluates the IRIS IIL components with respect to hardware implementation and simulation speed overhead, and also demonstrates the use of IRIS instrumentation in the context of CONNECT-generated Networks-on-Chip.

6.3.1 Hardware Implementation Characteristics

To give a sense of the hardware implementation characteristics of the various IRIS components we synthesize a wide range of variants of each base IIL component targeting a moderately-sized Xilinx FPGA and present resource usage and maximum clock frequency estimates. For each component we vary the subset of feature and structural configuration parameters that affect hardware implementation characteristics. Synthesis results were obtained using Xilinx XST 14.7 targeting a Xilinx Virtex-6 LX760T FPGA (part xc6vlx760).

Table 6.5 shows FPGA synthesis results for variants of the IrisCounter base IIL component. As expected, FPGA resource usage (LUTs and FFs) increases as the width and enabled features of an IrisCounter are increased. In particular, the number of used FFs always matches the width of the IrisCounter, while enabling extra features can increase LUT usage by up to 4x-5x. Maximum frequency is most affected by the "bitWidth" parameter, and in all cases the maximum frequency is at least 375MHz, which is well above typical target operating frequencies for FPGA designs.

Table 6.6 reports synthesis results for variants of the IrisSampler base IIL component. Increasing the values of the structural parameters of the IrisSampler ("bitWidth", "numBins", and "maxSamples") increases FPGA resource usage (LUTs and FFs). The "bitWidth" and "numBins" parameters have the strongest impact on the number of Flip-Flops (FFs) and LUTs respectively. Enabling the various "features" of the IrisSampler also increases FPGA resource usage, and in particular enabling the "keepSqTotal" feature also makes use of DSP hard blocks to implement the

	Structural Parameters	Feature Parameters					Synthesis Estimates		
IIL Component	**bitWidth**	**allowIncByN**	**allowDecByOne**	**allowDecByN**	**allowClear**	**allowWrite**	**LUTs**	**FFs**	**Max Freq.**
Varying structural parameters for IrisCounter with all features enabled									
mkIrisCounter	8	TRUE	TRUE	TRUE	TRUE	TRUE	37	8	583
mkIrisCounter	16	TRUE	TRUE	TRUE	TRUE	TRUE	73	16	541
mkIrisCounter	32	TRUE	TRUE	TRUE	TRUE	TRUE	145	32	471
mkIrisCounter	48	TRUE	TRUE	TRUE	TRUE	TRUE	203	48	417
mkIrisCounter	64	TRUE	TRUE	TRUE	TRUE	TRUE	259	64	375
Varying feature parameters for a 32-bit IrisCounter									
mkIrisCounter	32	FALSE	FALSE	FALSE	FALSE	FALSE	32	32	589
mkIrisCounter	32	FALSE	FALSE	FALSE	FALSE	TRUE	35	32	589
mkIrisCounter	32	FALSE	FALSE	FALSE	TRUE	FALSE	33	32	589
mkIrisCounter	32	FALSE	FALSE	FALSE	TRUE	TRUE	65	32	472
mkIrisCounter	32	FALSE	FALSE	TRUE	FALSE	FALSE	34	32	589
mkIrisCounter	32	FALSE	FALSE	TRUE	FALSE	TRUE	97	32	471
mkIrisCounter	32	FALSE	FALSE	TRUE	TRUE	FALSE	97	32	471
mkIrisCounter	32	FALSE	FALSE	TRUE	TRUE	TRUE	97	32	471
mkIrisCounter	32	FALSE	TRUE	FALSE	FALSE	FALSE	34	32	589
mkIrisCounter	32	FALSE	TRUE	FALSE	FALSE	TRUE	97	32	471
mkIrisCounter	32	FALSE	TRUE	FALSE	TRUE	FALSE	97	32	471
mkIrisCounter	32	FALSE	TRUE	FALSE	TRUE	TRUE	97	32	471
mkIrisCounter	32	FALSE	TRUE	TRUE	FALSE	FALSE	66	32	587
mkIrisCounter	32	FALSE	TRUE	TRUE	FALSE	TRUE	98	32	471
mkIrisCounter	32	FALSE	TRUE	TRUE	TRUE	FALSE	98	32	471
mkIrisCounter	32	FALSE	TRUE	TRUE	TRUE	TRUE	98	32	471
mkIrisCounter	32	TRUE	FALSE	FALSE	FALSE	FALSE	34	32	589
mkIrisCounter	32	TRUE	FALSE	FALSE	FALSE	TRUE	97	32	471
mkIrisCounter	32	TRUE	FALSE	FALSE	TRUE	FALSE	97	32	471
mkIrisCounter	32	TRUE	FALSE	FALSE	TRUE	TRUE	97	32	471
mkIrisCounter	32	TRUE	FALSE	TRUE	FALSE	FALSE	65	32	587
mkIrisCounter	32	TRUE	FALSE	TRUE	FALSE	TRUE	97	32	471
mkIrisCounter	32	TRUE	FALSE	TRUE	TRUE	FALSE	97	32	471
mkIrisCounter	32	TRUE	FALSE	TRUE	TRUE	TRUE	97	32	471
mkIrisCounter	32	TRUE	TRUE	FALSE	FALSE	FALSE	66	32	587
mkIrisCounter	32	TRUE	TRUE	FALSE	FALSE	TRUE	98	32	471
mkIrisCounter	32	TRUE	TRUE	FALSE	TRUE	FALSE	98	32	471
mkIrisCounter	32	TRUE	TRUE	FALSE	TRUE	TRUE	98	32	471
mkIrisCounter	32	TRUE	TRUE	TRUE	FALSE	FALSE	113	32	587
mkIrisCounter	32	TRUE	TRUE	TRUE	FALSE	TRUE	145	32	471
mkIrisCounter	32	TRUE	TRUE	TRUE	TRUE	FALSE	145	32	471
mkIrisCounter	32	TRUE	TRUE	TRUE	TRUE	TRUE	145	32	471

Table 6.5: FPGA Synthesis Estimates for Various IrisCounter Configurations.

required multiplier units. Frequency estimates for all IrisSampler variants range from 259MHz to 535MHz.

Table 6.7 shows FPGA synthesis results for several variants of the IrisStateMonitor base IIL component. To make efficient use of FPGA resources, in its default configuration, the IrisStateMonitor makes use of LUTRAM to store the state history. Thus, even at its largest configuration (32 states, 64 bits), which requires 2 Kbits of storage, the IrisStateMonitor only occupies 116 LUTs, 48 of which are used as LUTRAM. When the "allowClear" feature is enabled, the IrisSampler switches to a less efficient implementation and uses FFs, since LUTRAM cannot be cleared in a single clock cycle. Clock frequency estimates for the IrisStateMonitor range from 264MHz to 420MHz.

	Structural Parameters			Feature Parameters				Synthesis Estimates			
IIL Component	**bitWidth**	**numBins**	**maxSamples**	**keepMinMax**	**keepSqTotal**	**allowClear**	**hasBins**	**LUTs**	**FFs**	**DSPs**	**Max Freq.**
Varying structural parameters of mkIrisSampler with all features enabled											
mkIrisSampler	8	4	256	TRUE	TRUE	TRUE	TRUE	100	97	1	348
mkIrisSampler	8	4	1024	TRUE	TRUE	TRUE	TRUE	112	111	1	348
mkIrisSampler	8	4	4096	TRUE	TRUE	TRUE	TRUE	122	125	1	348
mkIrisSampler	8	8	256	TRUE	TRUE	TRUE	TRUE	121	129	1	348
mkIrisSampler	8	8	1024	TRUE	TRUE	TRUE	TRUE	136	151	1	348
mkIrisSampler	8	8	4096	TRUE	TRUE	TRUE	TRUE	161	173	1	348
mkIrisSampler	8	16	256	TRUE	TRUE	TRUE	TRUE	160	193	1	348
mkIrisSampler	8	16	1024	TRUE	TRUE	TRUE	TRUE	184	231	1	348
mkIrisSampler	8	16	4096	TRUE	TRUE	TRUE	TRUE	217	269	1	348
mkIrisSampler	8	32	256	TRUE	TRUE	TRUE	TRUE	276	321	1	335
mkIrisSampler	8	32	1024	TRUE	TRUE	TRUE	TRUE	324	391	1	334
mkIrisSampler	8	32	4096	TRUE	TRUE	TRUE	TRUE	380	461	1	309
mkIrisSampler	8	64	256	TRUE	TRUE	TRUE	TRUE	484	577	1	294
mkIrisSampler	8	64	1024	TRUE	TRUE	TRUE	TRUE	576	711	1	293
mkIrisSampler	8	64	4096	TRUE	TRUE	TRUE	TRUE	665	845	1	308
mkIrisSampler	8	128	256	TRUE	TRUE	TRUE	TRUE	888	1089	1	277
mkIrisSampler	8	128	1024	TRUE	TRUE	TRUE	TRUE	1065	1351	1	276
mkIrisSampler	8	128	4096	TRUE	TRUE	TRUE	TRUE	1248	1613	1	259
mkIrisSampler	16	4	256	TRUE	TRUE	TRUE	TRUE	172	137	1	427
mkIrisSampler	16	4	1024	TRUE	TRUE	TRUE	TRUE	184	151	1	427
mkIrisSampler	16	4	4096	TRUE	TRUE	TRUE	TRUE	194	165	1	427
mkIrisSampler	16	8	256	TRUE	TRUE	TRUE	TRUE	193	169	1	427
mkIrisSampler	16	8	1024	TRUE	TRUE	TRUE	TRUE	208	191	1	423
mkIrisSampler	16	8	4096	TRUE	TRUE	TRUE	TRUE	233	213	1	384
mkIrisSampler	16	16	256	TRUE	TRUE	TRUE	TRUE	232	233	1	400
mkIrisSampler	16	16	1024	TRUE	TRUE	TRUE	TRUE	256	271	1	398
mkIrisSampler	16	16	4096	TRUE	TRUE	TRUE	TRUE	289	309	1	363
mkIrisSampler	16	32	256	TRUE	TRUE	TRUE	TRUE	348	361	1	335
mkIrisSampler	16	32	1024	TRUE	TRUE	TRUE	TRUE	396	431	1	334
mkIrisSampler	16	32	4096	TRUE	TRUE	TRUE	TRUE	453	501	1	309
mkIrisSampler	16	64	256	TRUE	TRUE	TRUE	TRUE	556	617	1	294
mkIrisSampler	16	64	1024	TRUE	TRUE	TRUE	TRUE	648	751	1	293
mkIrisSampler	16	64	4096	TRUE	TRUE	TRUE	TRUE	738	885	1	308
mkIrisSampler	16	128	256	TRUE	TRUE	TRUE	TRUE	960	1129	1	277
mkIrisSampler	16	128	1024	TRUE	TRUE	TRUE	TRUE	1136	1391	1	276
mkIrisSampler	16	128	4096	TRUE	TRUE	TRUE	TRUE	1307	1653	1	259
mkIrisSampler	32	4	256	TRUE	TRUE	TRUE	TRUE	326	217	4	357
mkIrisSampler	32	4	1024	TRUE	TRUE	TRUE	TRUE	340	231	4	353
mkIrisSampler	32	4	4096	TRUE	TRUE	TRUE	TRUE	352	245	4	349
mkIrisSampler	32	8	256	TRUE	TRUE	TRUE	TRUE	347	249	4	357
mkIrisSampler	32	8	1024	TRUE	TRUE	TRUE	TRUE	367	271	4	353
mkIrisSampler	32	8	4096	TRUE	TRUE	TRUE	TRUE	391	293	4	349
mkIrisSampler	32	16	256	TRUE	TRUE	TRUE	TRUE	387	313	4	357
mkIrisSampler	32	16	1024	TRUE	TRUE	TRUE	TRUE	415	351	4	353
mkIrisSampler	32	16	4096	TRUE	TRUE	TRUE	TRUE	447	389	4	349
mkIrisSampler	32	32	256	TRUE	TRUE	TRUE	TRUE	502	441	4	335
mkIrisSampler	32	32	1024	TRUE	TRUE	TRUE	TRUE	552	511	4	334
mkIrisSampler	32	32	4096	TRUE	TRUE	TRUE	TRUE	611	581	4	309
mkIrisSampler	32	64	256	TRUE	TRUE	TRUE	TRUE	710	697	4	294
mkIrisSampler	32	64	1024	TRUE	TRUE	TRUE	TRUE	804	831	4	293
mkIrisSampler	32	64	4096	TRUE	TRUE	TRUE	TRUE	896	965	4	308
mkIrisSampler	32	128	256	TRUE	TRUE	TRUE	TRUE	1114	1209	4	277
mkIrisSampler	32	128	1024	TRUE	TRUE	TRUE	TRUE	1292	1471	4	276
mkIrisSampler	32	128	4096	TRUE	TRUE	TRUE	TRUE	1479	1733	4	259
Varying feature parameters for a 32-bit mkIrisSampler with 8 bins and 1024 max samples											
mkIrisSampler	32	8	1024	FALSE	FALSE	FALSE	FALSE	51	52	0	535
mkIrisSampler	32	8	1024	FALSE	FALSE	FALSE	TRUE	110	132	0	423
mkIrisSampler	32	8	1024	FALSE	FALSE	TRUE	FALSE	53	52	0	535
mkIrisSampler	32	8	1024	FALSE	FALSE	TRUE	TRUE	113	132	0	423
mkIrisSampler	32	8	1024	FALSE	TRUE	FALSE	FALSE	125	126	4	414
mkIrisSampler	32	8	1024	FALSE	TRUE	FALSE	TRUE	184	206	4	414
mkIrisSampler	32	8	1024	FALSE	TRUE	TRUE	FALSE	201	126	4	353
mkIrisSampler	32	8	1024	FALSE	TRUE	TRUE	TRUE	261	206	4	353
mkIrisSampler	32	8	1024	TRUE	FALSE	FALSE	FALSE	180	117	0	458
mkIrisSampler	32	8	1024	TRUE	FALSE	FALSE	TRUE	239	197	0	423
mkIrisSampler	32	8	1024	TRUE	FALSE	TRUE	FALSE	182	117	0	458
mkIrisSampler	32	8	1024	TRUE	FALSE	TRUE	TRUE	242	197	0	423
mkIrisSampler	32	8	1024	TRUE	TRUE	FALSE	FALSE	254	191	4	414
mkIrisSampler	32	8	1024	TRUE	TRUE	FALSE	TRUE	313	271	4	414
mkIrisSampler	32	8	1024	TRUE	TRUE	TRUE	FALSE	300	191	4	353
mkIrisSampler	32	8	1024	TRUE	TRUE	TRUE	TRUE	363	271	4	353

Table 6.6: FPGA Synthesis Estimates for Various IrisSampler Configurations.

	Structural Parameters		Feature Parameters	Synthesis Estimates			
IIL Component	**numStates**	**maxEventCount**	**allowClear**	**LUTs**	**LUTRAM**	**FFS**	**Max Freq.**
Varying structural parameters of IrisStateMonitor with all features enabled							
mkIrisStateMonitor	4	2^8	FALSE	20	8	0	420
mkIrisStateMonitor	4	2^16	FALSE	33	16	0	337
mkIrisStateMonitor	4	2^32	FALSE	57	24	0	305
mkIrisStateMonitor	4	2^48	FALSE	81	32	0	279
mkIrisStateMonitor	4	2^64	FALSE	113	48	0	264
mkIrisStateMonitor	8	2^8	FALSE	21	8	0	420
mkIrisStateMonitor	8	2^16	FALSE	34	16	0	337
mkIrisStateMonitor	8	2^32	FALSE	58	24	0	305
mkIrisStateMonitor	8	2^48	FALSE	82	32	0	279
mkIrisStateMonitor	8	2^64	FALSE	114	48	0	264
mkIrisStateMonitor	16	2^8	FALSE	22	8	0	420
mkIrisStateMonitor	16	2^16	FALSE	35	16	0	337
mkIrisStateMonitor	16	2^32	FALSE	59	24	0	305
mkIrisStateMonitor	16	2^48	FALSE	83	32	0	279
mkIrisStateMonitor	16	2^64	FALSE	115	48	0	264
mkIrisStateMonitor	32	2^8	FALSE	23	8	0	420
mkIrisStateMonitor	32	2^16	FALSE	36	16	0	337
mkIrisStateMonitor	32	2^32	FALSE	60	24	0	305
mkIrisStateMonitor	32	2^48	FALSE	84	32	0	279
mkIrisStateMonitor	32	2^64	FALSE	116	48	0	264
Varying feature parameters for a 32-bit IrisStateMonitor with 8 states							
mkIrisStateMonitor	8	2^32	FALSE	58	24	0	305
mkIrisStateMonitor	8	2^32	TRUE	200	0	256	339

Table 6.7: FPGA Synthesis Estimates for Various IrisStateMonitor Configurations.

The FPGA resource usage of IRIS composite components, such as Iris1D/Iris2D and IrisTracer, scale according to the size of the collection and the size of the trace buffer respectively. The clock frequency of IRIS composite components tracks the clock frequency of the underlying base components that they are using. The hardware implementation characteristics of an IrisChecker component can vary and depend on the complexity and the associate storage and logic requirements of the specific type of checker that the IP author is implementing. Finally, the IrisLogger and IrisConfig components are simulation-only components.

6.3.2 IRIS Instrumentation in CONNECT

In this section we demonstrate how IRIS instrumentation is used in the context of the CONNECT Network-on-Chip IP generator for accumulating statistics, investigating performance issues, as well as debugging functional correctness problems. We first describe the types of instrumentation components employed, then discuss the impact of instrumentation on simulation speed, and finally show some visualization results from sample CONNECT networks driven by actual traffic.

To investigate the use of IRIS as part of an actual IP generator, we modified the CONNECT IP generator to produce NoCs that include a wide range of representative sample IRIS instrumentation components, including:

- IrisConfig components that capture configuration information about the network being tested.
- IrisCounter components, including Iris1D and Iris2D variants, for counting packets and flits, including total counts, counts per source, per destination, as well as per source-destination pair.
- IrisSampler components, including Iris1D and IrisTracer variants, for measuring average packet and flit latency, across all traffic, as well as per source and per destination.
- IrisStateMonitor components to track the state of multiple routers in the network.
- An IrisChecker component to detect and report when a deadlock has occurred.

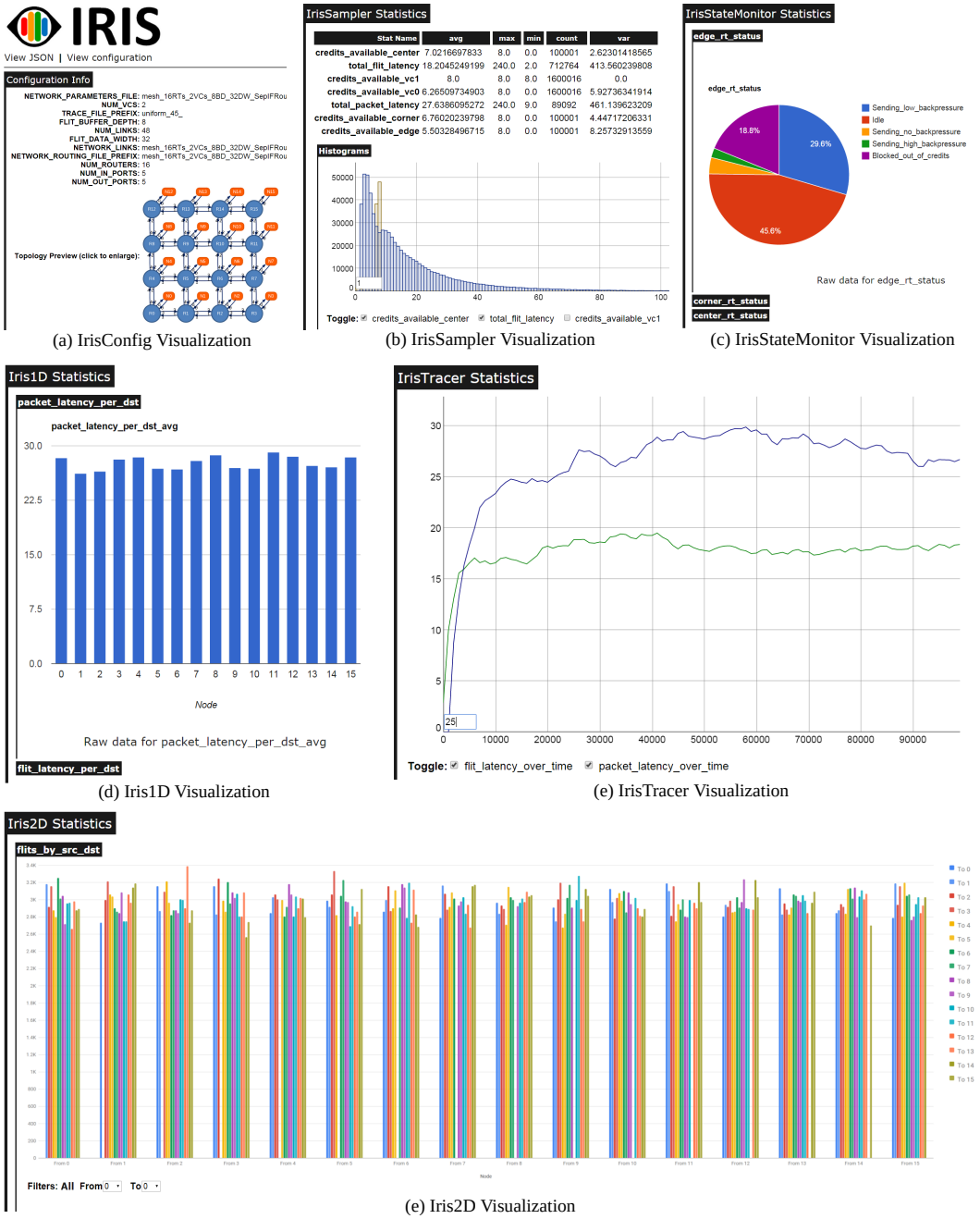
Impact on Simulation Speed. As mentioned earlier, IRIS has been designed to have minimal impact on simulation speed, especially on modern multi-core systems. To give a sense of the actual simulation speed overhead of IRIS in highly instrumented designs, we report simulation runtime overhead for a CONNECT NoC design with IRIS instrumentation. In particular, we generate a heavily instrumented sample 4x4 mesh NoC design (using Input-Queued routers, 32-bit wide links, 8-entry flit buffers) and drive it using a uniform random traffic pattern (with a load of 0.45 flits/cycle and 8-flit packets) for 250,000 cycles. For each simulation we report the average runtime of five simulation runs. Results are reported using the Synopsys VCS RTL simulator [7] (version I-2014.03) running on a Core i5-2500 @ 3.2GHz with 8GB of RAM. The Python-based IIE and IVE componets are run using the PyPy Python interpreter [4] (version 2.5.1).

When instrumentation is disabled, the average simulation runtime is 105s. Enabling IRIS instrumentation increases average runtime to 154.6s. Note that this does not include any IRIS-specific

processing and is the pure overhead due to adding the instrumentation logic and probes, which would be incurred when instrumenting a design, regardless if using IRIS or not. Next we enable the IRIS introspection engine (IIE). Enabling the IIE increases simulation time by 21.2s, which corresponds to an overhead of 13.7%. Enabling the IRIS visualization engine (IVE) has negligible overhead and only increases the average runtime by 0.14s. Overall in our experiments, the IIE has been able to process approximately 380,000 IRIS events per second. Obviously, the overhead of IRIS heavily depends on the exact hardware design, the type and configuration of instrumentation components, as well as the RTL simulator; these results are meant to give a rough estimate of the impact of IRIS on simulation speed and calibrate expectations.

Visualization. Here we showcase sample visualizations generated by the IRIS Visualization Engine (IVE). The IVE engine consumes the JSON-formatted summary produced by the IIE and generates a dynamic graphical user interface using HTML and Javascript. This interface presents all of the collected instrumentation data organized by type and allows interacting with different components and even juxtaposing the data collected by different instrumentation components.

Figure 6.3 (a) shows the output of IrisConfig components, which capture the configuration parameters of the specific simulation runs, including a graphical representation of the network topology. Figure 6.3 (b) shows the output of IrisSampler components, which include statistical information, such as the average, min, max, and variance, as well as a histogram that allows juxtaposing different IrisSamplers. Figure 6.3 (c) shows a sample pie chart used to visualize an IrisStateMonitor that breaks down how individual routers in the design spend their time. Figure 6.3 (d) shows a collection of bar graphs that corresponds to an Iris1D collection of IrisCounters, and Figure 6.3 (e) shows an Iris2D collection of IrisCounters that show the packets sent between each source-destination pair. Finally, Figure 6.3 (f) shows the output of an IrisTracer component that keeps track of two IrisSamplers that capture the average packet and flit latency over time.



(a) IrisConfig Visualization

(b) IrisSampler Visualization

(c) IrisStateMonitor Visualization

(d) Iris1D Visualization

(e) IrisTracer Visualization

(e) Iris2D Visualization

Figure 6.3: Examples of How Different IRIS Components are Presented by the IRIS Visualization Engine (IVE).

6.4 Related Work

There is a large body of existing work that has looked at various aspects of hardware instrumentation, improving visibility, and collecting, extracting, or syncing state from a hardware platform, as well as methodologies and techniques, such as Design-for-Test (DFT) and Design-for-Debug (DFD), for faster, more efficient debugging and validation. Compared to such existing work, the main contributions of IRIS are focused on providing a reusable parameterizable library of basic instrumentation components along with a post-processing and visualization engine that help IP authors distill and present the information that would be truly useful to a Pandora IP user.

Previous work has looked at facilitating the development of hardware-based run-time monitors in processor designs. In particular, [51] presents a processor architecture framework that allows designers to specify monitors in higher-level languages or even high-level synthesis (HLS) tools. FlexCore [35] presents a hybrid processor architecture that allows dynamic fine-grain control of the monitoring and bookkeeping functions of hardware monitors using an on-chip reconfigurable fabric. MAMon [39] shares ideas with IRIS and proposes a monitoring system that collects low-level events in hardware, which can be post-processed in software to study system-level behaviors in a System-on-a-Chip environment. [76] looks at how monitoring and introspection mechanisms can be built on separate active layers in a future 3D IC technology.

In the context of Network-on-Chip (NoC) and on-chip monitoring, existing work [9, 10, 85] has looked at how to more efficiently detect and debug functional and performance issues in a NoC. In addition, there is also a lot of work [30, 42, 100, 101, 107, 108, 114] on using the NoC, by examining the communication and sequence of transactions in a system, to offer higher-level visibility and accelerate debugging in embedded systems and Systems-on-a-Chip.

An overview of on-chip monitoring and debug strategies for SoCs is provided in [48], while [61] summarizes and characterizes existing efforts of debug methods for hardware and software com-

ponents of embedded systems. Finally, [109] offers an introduction and overview of debugging strategies and methods for multi-core Systems-on-Chip, while [55] offers a very thorough and extensive survey and taxonomy of on-chip monitoring of multicore Systems-on-Chip.

Chapter 7

Putting It All Together

This chapter describes how we extended the CONNECT Network-on-Chip (NoC) IP generator, first presented in Chapter 2, to incorporate and demonstrate many of the Pandora key ideas and research artifacts presented in Chapters 3, 4, 5, and 6. In particular, this proof-of-concept Pandora-powered version of CONNECT (1) leverages DELPHI to provide fast hardware implementation and performance estimates, (2) uses the Nautilus optimization engine to build high-level configuration and tuning interfaces, and (3) incorporates IRIS instrumentation and introspection capabilities into each generated network to facilitate efficient runtime debugging and monitoring. The remainder of this chapter briefly describes and showcases each one of these CONNECT extensions.

7.1 Fast NoC Design Cost and Performance Estimation

This section describes how we leverage DELPHI along with multivariate interpolation to extend CONNECT to provide real-time—in a matter of seconds—hardware implementation and performance estimates for each candidate network configuration. These estimates are updated in real-

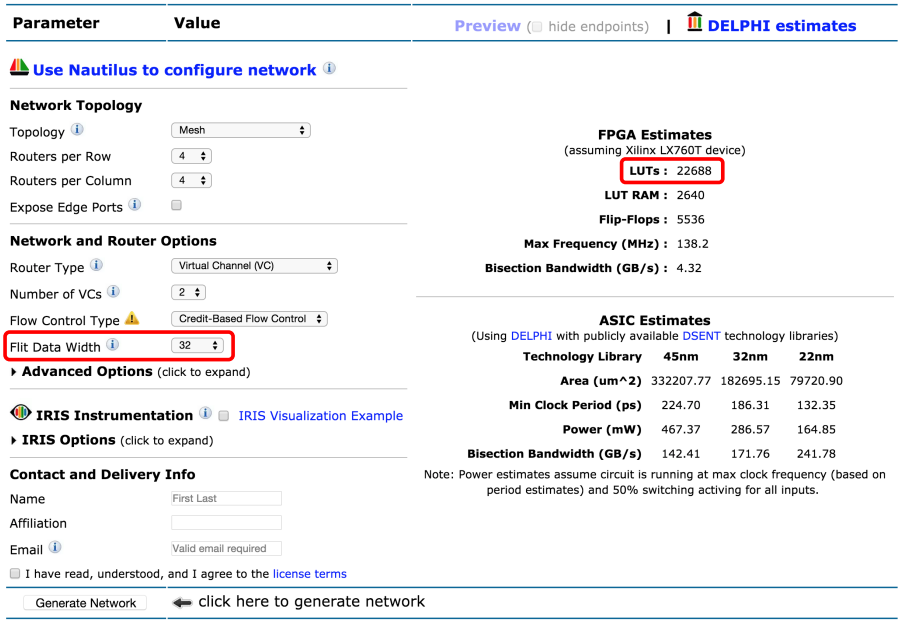
time as the user modifies the various configuration parameters, which allows for fast informed navigation of the design space.

Figure 7.1 shows two screenshots of the updated web-based interface of CONNECT, which includes a new preview tab that provides instant on-demand feedback for each network configuration, including implementation and performance estimates for FPGAs and multiple ASIC technology nodes. In this particular example varying the flit data width from 32 in Figure 7.1 (a) to 8 in Figure 7.1 (b) can yield a significant reduction in implementation area, e.g., from 22688 to 16272 LUTs when targeting an FPGA. This type of instant feedback can dramatically reduce the lengthy trial-and-error process that IP users typically have to endure in order to find an IP configuration that meets application goals and design constraints.

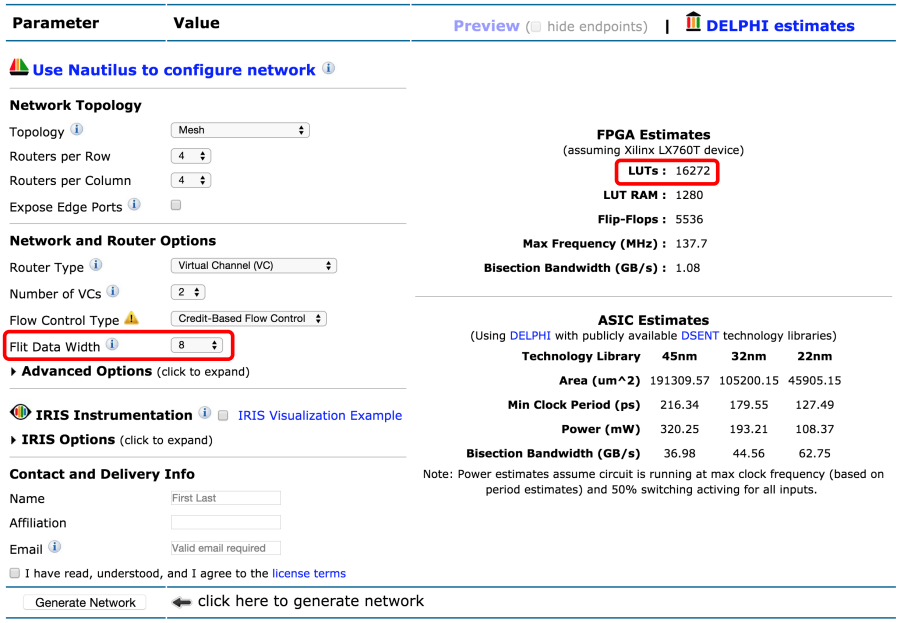
This CONNECT feature relies on DELPHI for quickly characterizing several thousands of CONNECT NoC configurations, which are then used as the anchor grid points in the multivariate interpolation. In our proof-of-concept implementation, we used 10370 DELPHI-based estimates to populate the interpolation anchor point database and then leveraged the SciPy Interpolation library [6] to characterize the full design space.

7.2 High-level NoC Configuration and Optimization

This section presents sample high-level configuration interfaces for CONNECT that were built utilizing the Nautilus IP optimization engine. These proof-of-concept interfaces include (1) a simple "fuzzy" trade-off slider that allows novice users to set the desired balance between implementation cost and network performance, (2) a constraint-driven single-metric optimization interface that supports maximizing/minimizing various high-level design metrics and arbitrary constraint expressions, as well as (3) a custom query interface that allows expert users to directly define the fitness function used by the guided genetic algorithm during the Nautilus optimization process.



(a)



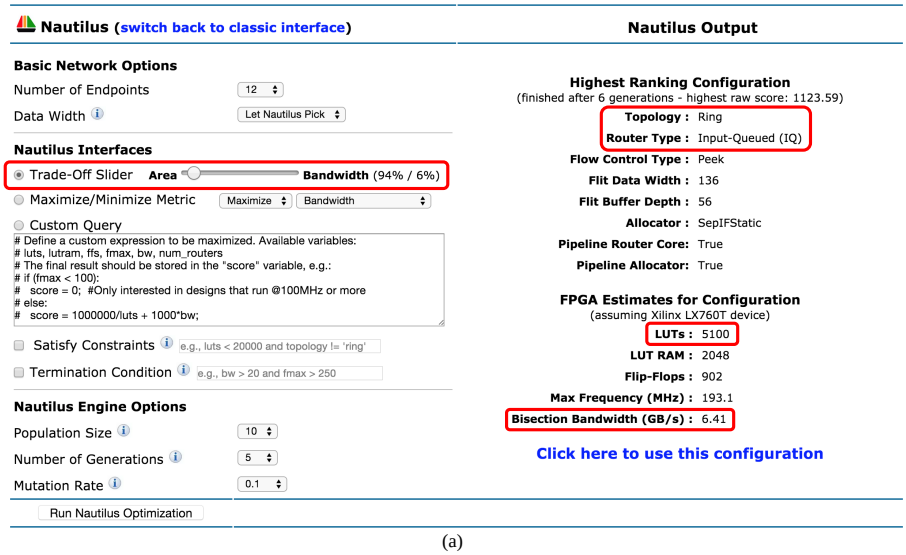
(b)

Figure 7.1: Example Snapshots of the DELPHI-Powered Interface, which Provides Real-Time Hardware Implementation and Performance Estimates for CONNECT Network Configurations.

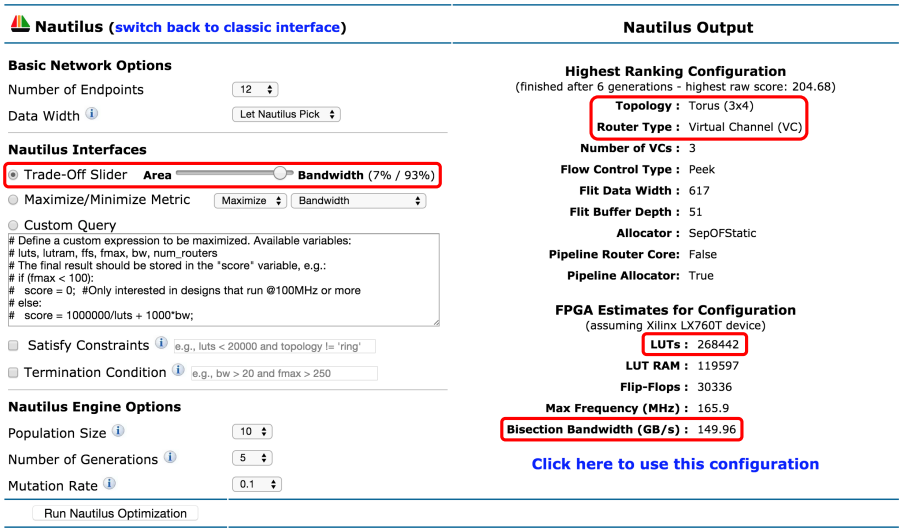
Behind the scenes, all of these interfaces are built on top of the Nautilus IP optimization engine presented in Chapter 5. Compared to the "traditional" version of CONNECT, where it was left to the IP user to explicitly set all of the low-level IP parameters, these new interface extensions allow for high-level goal-oriented optimization. The IP user now only needs to specify the number of network endpoints; all low-level parameters are automatically populated by Nautilus.

The snapshots in Figure 7.2 show examples of using these new Nautilus-based configuration interfaces. Figures 7.2 (a) and (b) show the resulting network configurations when using the trade-off slider to pick a low-cost vs. high-performance network respectively. In this example, when opting for a low-cost network, Nautilus suggests a lightweight network that consists of simple Input-Queued routers arranged in a Ring topology, which only occupies 5100 LUTs and can offer up to 6.41 GB/s of bisection bandwidth. When opting for a high-performance network, Nautilus suggests a much denser 3x4 Torus topology built out of more sophisticated Virtual-Channel routers, which occupies 268,442 LUTs and can offer up to 149.96 GB/s.

Figures 7.3 (a) and (b) show examples of using the constraint-driven single-metric optimization interface. Figure 7.3 (a) shows the suggested network configuration for a simple query that tries to maximize the operating frequency of the resulting design. In this case Nautilus suggests a Line topology network that uses low-radix (2x2) fast Input-Queued routers, which have the shortest critical path, leading to a max estimated operating frequency of 333.5 MHz. Figure 7.3 (b) shows how queries can be further fine-tuned and tailored by adding constraints. In this case the constraint dictates that the topology is not a "Line" and that the routers are not "Input-Queued", causing Nautilus to now suggest a Ring topology built out of Virtual-Output-Queued routers, which are the next simplest and fastest topology and router combination.



(a)

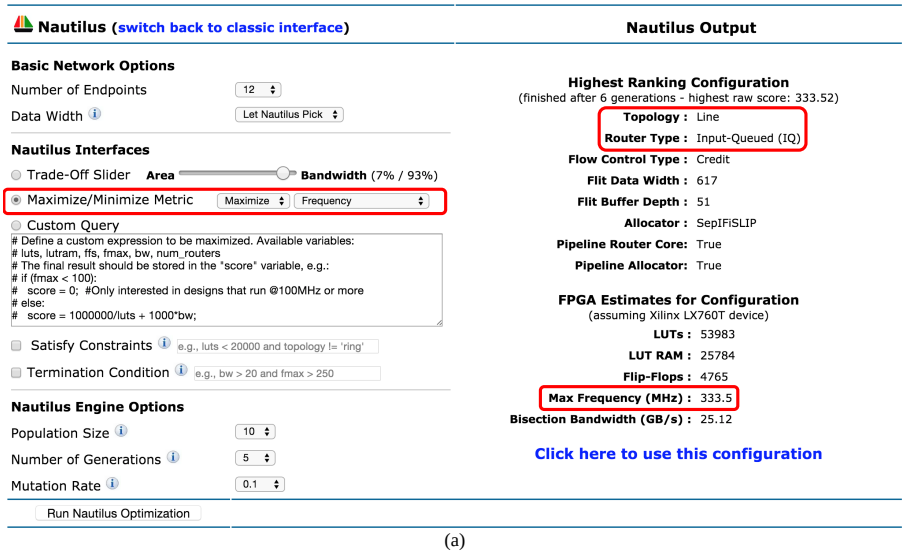


(b)

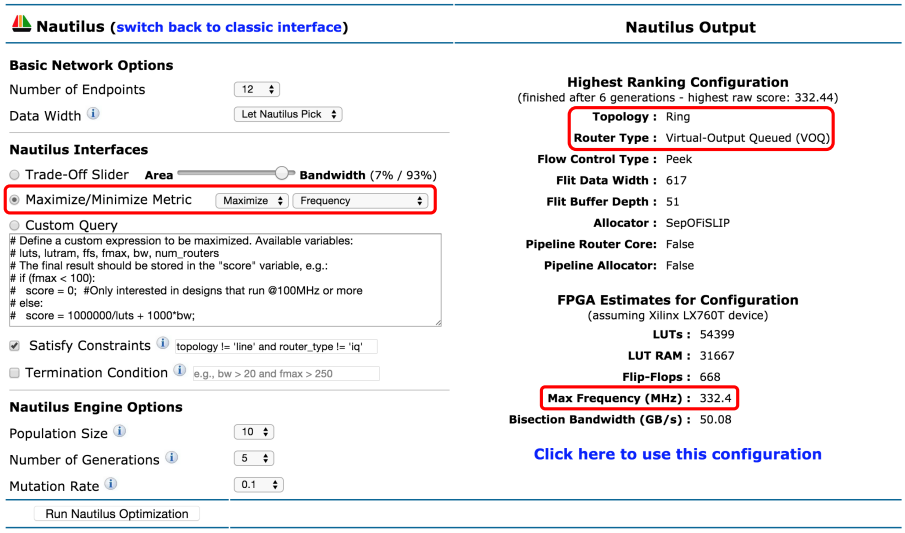
Figure 7.2: Example Snapshots of the Nautilus-Powered CONNECT Configuration Interface Highlighting the Area vs. Bandwidth Trade-Off Slider Subinterface.

7.3 Effortless Monitoring and Debugging

This section highlights how we extended the CONNECT NoC generation engine to incorporate many of the IRIS instrumentation components presented in Chapter 6. Enabling IRIS instrumenta-



(a)



(b)

Figure 7.3: Example Snapshots of the Nautilus-Powered CONNECT Configuration Interface Highlighting the Constraint-Driven Single-Metric Optimization Subinterface.

tion allows the generated NoC designs to leverage the IRIS Introspection and Visualization Engines for efficient high-level monitoring and debugging.

Figure 7.4 shows how the CONNECT web-based generation interface was augmented to sup-

port IRIS instrumentation. Checkbox elements allow IP users to fine-tune the types and granularity of instrumentation components to match their application requirements and debugging needs. The majority of the supported instrumentation options can be temporally or spatially extended to track behaviors over time or at a finer granularity, e.g., per network router (1D) or per network source-destination pair (2D). Figure 6.3 shows visualization samples of IRIS-instrumented CONNECT networks.

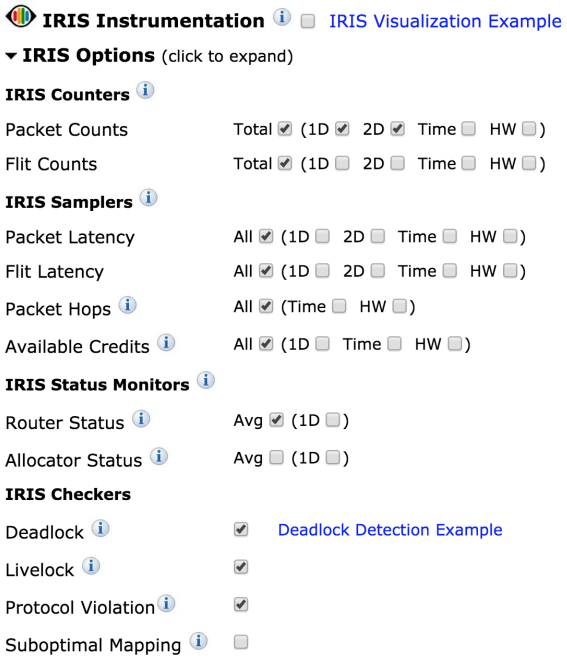


Figure 7.4: Snapshot Showing the Various IRIS-Powered CONNECT Instrumentation Options.

Even though this Pandora-powered version of CONNECT was built as a proof of concept, it already demonstrates how the key Pandora ideas and research artifacts presented throughout this thesis can dramatically enhance how IP users interact with IP generation frameworks and increase productivity when working with the resulting IPs.

Chapter 8

Conclusions

Hardware specialization is proving to be a promising answer [89] to our quest to continue scaling system performance in the power-constrained post-Dennard era. However, a major obstacle to the more widespread use of hardware acceleration is the level of difficulty in hardware design today. Despite the increased availability of rich IP libraries and even IP generators that span a wide range of application domains, developing hardware today not only takes more time and is more expensive than ever, but, more importantly, is limited to experts. As the complexity of IPs has grown to the level of complete processors or Networks-on-Chip, today's IPs require elaborate parameterization and even often take the form of IP design generators to support the high degree of parameterization. This unprecedented rise in IP complexity has led to a new design challenge where just understanding and properly setting the myriads of domain-specific parameters of an IP to produce a working and well-tuned design is becoming unmanageable to the average IP user.

This thesis presents the Pandora IP development paradigm that facilitates hardware specialization by extending the concept of generator-based IPs. Pandora encapsulates the IP author's expertise and knowledge to offer supporting infrastructure that enhances how IP users interact with

the IP. In contrast to existing IPs and IP generators that only capture the structural and microarchitectural view of a design, Pandora argues for augmenting IPs with: (1) detailed IP design space characterization to help the user understand the effects of parameter choices with respect to hardware implementation and IP-specific metrics, (2) application-level goal-oriented parameterization that is meaningful to the IP user and automatically sets low-level structural parameters to achieve the desired design optimizations, and (3) purpose-built domain-aware simulation-time and run-time monitoring mechanisms to assist functional and performance debugging.

To demonstrate the effectiveness of the Pandora IP development paradigm, this thesis presents our work on: (1) CONNECT, a highly parameterized Network-on-Chip IP generator that embodies and demonstrates many of the Pandora principles, (2) DELPHI, a framework for fast easy IP characterization that facilitates mapping the design space of arbitrary RTL-based IPs, (3) Nautilus, an IP optimization engine that demonstrates how incorporating IP author knowledge in genetic algorithms can enable very fast—orders of magnitude faster than conventional methods—high-level goal-oriented IP optimization, and (4) IRIS, an instrumentation and introspection framework that combines hardware monitors with software-based post-processing and visualization engines to accelerate debugging of complex IPs and enable higher system-level visibility.

The Pandora IP development paradigm and our efforts on CONNECT, DELPHI, Nautilus, and IRIS are important steps towards realizing our overarching goal of tackling the increasing complexity in hardware design and enabling the more widespread adoption of hardware acceleration for sustained performance and power efficiency. Today, a computer vision or signal processing expert can relatively easily and quickly experiment in software (e.g., using Matlab), but doing so in hardware is out of the question. Our hope with Pandora is to help bridge this gap, so that in the near future application-experts are able to experiment with their ideas in hardware the same way that they are able to do so in software today.

8.1 Future Directions

While CONNECT, DELPHI, Nautilus, and IRIS concretely demonstrate multiple aspects of the Pandora IP development paradigm, the scope and key principles of Pandora were conscientiously defined to transcend the individual tools, techniques, and methods developed in these projects. As such, the broad nature of Pandora paves the way for many interesting extensions and future research directions in the quest to reign in hardware design complexity.

Automated Design Space Sampling and Mapping. While DELPHI can dramatically reduce the time required to map the design space of an IP and CONNECT includes hand-crafted and heuristic-based predictive models, it would be interesting to investigate more general and "smarter" approaches to mapping the design space of an IP that can work across multiple domains. IP author hints, such as the ones used in Nautilus, could be used to define interesting regions of the design space to steer the sampling and mapping process. To even further automate and accelerate the IP design space mapping, it would be interesting to experiment with and potentially adapt some of the ideas used in PinPoints [86] to identify representative regions of software benchmarks.

Multiple Pandora IPs. While this thesis has focused on demonstrating Pandora in the context of a single IP, typical Systems-on-Chip are composed of multiple such IP blocks that span multiple domains. As such, an interesting future direction is to investigate how multiple Pandora IPs can be composed, and in particular, focus on (1) how to abstract away the interfacing details between IPs, which are a common source of errors, and (2) how to implement a common cross-IP optimization framework that can tune the potentially domain-specific "knobs" of each IP in a meaningful and coordinated fashion to meet higher system-level goals and requirements.

Machine Learning and Automated IP Optimization. In Nautilus, we demonstrated how IP author knowledge can be incorporated in a guided genetic algorithm to vastly accelerate hardware IP optimization. While using a stochastic optimization process, such as the genetic algorithms in

Nautilus, yielded promising results, it would be interesting to investigate if training machine learning models, such as artificial neural networks [47], can lead to an even faster and more effective approach to IP optimization that performs well even in the absence of IP author hints. Along similar lines, it would be interesting to explore how the output of instrumentation can be analyzed and post-processed to directly feed the optimization process, thus even further automating and tightening the IP optimization loop.

Appendix A

CONNECT Command-Line Interface Options

```
Usage
=====

SYNOPSIS
    gen_network.py [-h,--help] [-v,--verbose] [--version]

DESCRIPTION
    This script generates configuration files that are used by the network rtl and represent
different topologies and network configurations.  In particular these files are generated:
      - network_parameters.bsv : specifies network and router parameters.
      - network_links.bsv      : specifies how routers are connected (i.e. topology)
      - network_routing_X.hex  : specifies contents of router tables for router X (i.e. routing)

EXAMPLES
    gen_network.py -t ring -n 4 -v 2 -d 8 -w 256

EXIT STATUS
```

Exit codes

AUTHOR

Michael K. Papamichael <papamix@cs.cmu.edu>

LICENSE

Please contact the author for license information.

VERSION

0.6

Options

=======

--version show program's version number and exit

--help, -h show this help message and exit

--verbose verbose output

--topology=TOPOLOGY, -t TOPOLOGY

specifies topology (can take values "single_switch", "line", "ring", "double_ring", "star", "mesh", "torus", "fat_tree", "butterfly", "fully_connected", "uni_single_switch", "uni_tree", uni_tree_up", "uni_tree_down", "custom", "ideal", "xbar")

--sendmail=SENDMAIL Send email notification when done

--num_routers=NUM_ROUTERS, -n NUM_ROUTERS

Specifies number of endpoint routers

--send_endpoints=SEND_ENDPOINTS

Specifies number of send endpoints for uni-directional topologies

--recv_endpoints=RECV_ENDPOINTS

Specifies number of receive endpoints for uni-directional topologies

--override_num_dests=OVERRIDE_NUM_DESTS

Allows overriding the number of total network destinations (e.g., to build larger networks out of

```
                        smaller subnetworks). This leads to larger routing
                        tables and a wider destination field in flits. (if not
                        set network destinations will equal network receive
                        ports of current topology)
--routers_per_row=ROUTERS_PER_ROW, -r ROUTERS_PER_ROW
                        specifies number of routers in each row (only used for
                        mesh and torus)
--routers_per_column=ROUTERS_PER_COLUMN, -c ROUTERS_PER_COLUMN
                        specifies number of routers in each column (only used
                        for mesh and torus)
--num_vcs=NUM_VCS, -v NUM_VCS
                        specifies number of virtual channels
--alloc_type=ALLOC_TYPE, -a ALLOC_TYPE
                        specifies type of allocator (can take values
                        "SepIFRoundRobin", "SepOFRoundRobin", "SepIFiSLIP",
                        "SepOFiSLIP", "SepIFStatic", "SepOFStatic",
                        "Memocode")
--use_virtual_links     Enables locking of virtual links (VC+OutPort) in the
                        presence of multi-flit packets.
--flit_buffer_depth=FLIT_BUFFER_DEPTH, -d FLIT_BUFFER_DEPTH
                        specifies depth of flit buffers
--sink_buffer_depth=SINK_BUFFER_DEPTH, -s SINK_BUFFER_DEPTH
                        specifies depth of buffers at receiving endpoints. If
                        not specified flit_buffer_depth is assumed.
--flit_data_width=FLIT_DATA_WIDTH, -w FLIT_DATA_WIDTH
                        specifies flit data width
--cut=CUT, -i CUT       specifies the cut in an ideal or xbar network
--file_prefix=FILE_PREFIX, -p FILE_PREFIX
                        override default file prefix
--xbar_lanes=XBAR_LANES, -l XBAR_LANES
                        specifies number of lanes in Xbar network
--output_dir=OUTPUT_DIR, -o OUTPUT_DIR
                        specifies output directory (default is ./)
--gen_rtl, -g           invokes bsc compiler to generate rtl
```

```
--run_xst, -x           generates rtl and invokes xst for synthesis
--run_dc                generates rtl and invokes Synopsys DC for synthesis
--expose_unused_ports   Exposes unused user ports for Mesh topology.
--flow_control_type=FLOW_CONTROL_TYPE
                        specifies flow control type, Credit-based or Peek (can
                        take values "credit", "peek")
--peek_flow_control     Uses simpler peek flow control interface instead of
                        credit-based interface.
--router_type=ROUTER_TYPE
                        specifies router type, Virtual-Channel-based, Virtual-
                        Output-Queued or Input-Queued (can take values "vc",
                        "voq", "iq")
--voq_routers           Use Virtual-Output-Queued (VOQ) routers instead of
                        Virtual-Channel (VC) routers.
--uni_tree_inputs=UNI_TREE_INPUTS
                        Number of tree input ports.
--uni_tree_outputs=UNI_TREE_OUTPUTS
                        Number of tree input ports.
--uni_tree_fanout=UNI_TREE_FANOUT
                        Fan-out of each tree router (will be calculated
                        automatically if set to 0).
--uni_tree_distribute_leaves
                        Distributes the leaf nodes to the available routers,
                        when the tree does not perfectly fit the available
                        leaf nodes.
--pipeline_core         Pipelines router core.
--pipeline_alloc        Pipelines router allocator.
--pipeline_links        Pipelines flit and credit links.
--concentration_factor=CONCENTRATION_FACTOR
                        specifies number of user ports per endpoint router
                        (not implemented yet)
--custom_topology=CUSTOM_TOPOLOGY
                        specifies custom topology file.
--custom_routing=CUSTOM_ROUTING
```

```
                        specifies custom routing file.
--dump_topology_file    dumps the topology spec file for the generated
                        network.
--dump_routing_file     dumps the routing spec file for the generated network.
--dbg                   Enables debug messages in generated rtl.
--dbg_detail            Enables more detailed debug messages in generated rtl.
--gen_graph             Visualizes network graph using graphviz.
--graph_nodes           Also includes endpoint nodes in generated graph.
--graph_format=GRAPH_FORMAT
                        Specifies output format for graphviz (e.g., png, jpg
                        or svg)
--graph_layout=GRAPH_LAYOUT
                        Specifies graphviz layout engine (e.g., dot, neato,
                        circle)
```

Appendix B

DELPHI Flow Overview

```
#==========================================================================
#
# DELPHI - RTL-Based Architecture Design Evaluation Using DSENT Models
# Copyright (c) 2014 by Michael K. Papamichael, Carnegie Mellon University
#
#==========================================================================

This document describes the steps involved in taking a design through the
DELPHI flow.

##########################################################################
#### 1. Synopsys DC Synthesis
##########################################################################

DELPHI requires that synthesis in Synopsys DC is constrained to only use cells
that are supported in DSENT. The cells currently supported in DSENT are (the
user is free to further modify DSENT and add models for other standard cells):

AND2    - 2-input AND gate with input pins A, B and output pin Y
OR2     - 2-input OR gate with input pins A, B and output pin Y
```

```
BUF    – Buffer with input pin A and output pin Y
INV    – Inverter with input pin A and output pin Y
NAND2  – 2–input NAND gate with input pins A, B and output pin Y
NOR2   – 2–input NOR  gate with input pins A, B and output pin Y
MUX2   – 2–input multiplexer with input pins A, B, S0 and output pin Y
XOR2   – 2–input XOR  gate with input pins A, B and output pin Y
ADDF   – Full adder with input pins A, B, CI and output pins S and CO
DFFQ   – D Flip–Flop with input pin D and output pin Q, clocked by pin CKN
DFFQ   – Latch with input pins D, G and output pin Q

The default driving strengths supported for these cells are (the user can
modify the available driving strengths by modifying the DSENT technology
models):
1.0, 1.4, 2.0, 3.0, 4.0, 6.0, 8.0, 10.0, 12.0, 16.0


##############################################################################
#### 1.1 Creating a DC–DSENT cell mapping file

The DELPHI flow requires that the standard cells used during synthesis are
mapped to the above DSENT–compatible cells. This mapping is captured through a
user–defined file that uses a custom specification language to map cells and
pins of the standard cells used in DC synthesis to the cells and pins available
in DSENT. Each line of the mapping file can contain one of the following
commands (lines starting with # are comments):

MAP_CELL
  Info:    Maps DC cell to DSENT cell
  Syntax:  MAP_CELL          <DC_CELL_NAME> <DSENT_CELL_NAME> <DRIVING_STRENGTH>
  Example: MAP_CELL            NAND2X4            NAND2                 4

MAP_CELL_PIN
  Info:    Maps pin of DC cell to pin of DSENT cell
  Syntax:  MAP_CELL_PIN     <DC_CELL_NAME> <DC_PIN_NAME> <DSENT_pin_name>
```

```
  Example: MAP_CELL_PIN          NAND2X4          Z                Y

IGNORE_CELL_PIN
  Info:    Instructs DELPHI to ignore a specific pin of a DC shell
  Syntax:  IGNORE_CELL_PIN <DC_CELL_NAME> <DC_PIN_NAME>
  Example: IGNORE_CELL_PIN      DFFSRX1          QN

ADD_CELL_IN_PIN  (only used for YOSYS+ABC flow)
  Info:    Instructs DELPHI to treat a pin on a cell as an input
  Syntax:  ADD_CELL_IN_PIN <ABC_CELL_NAME>    <ABC_PIN_NAME>
  Example: ADD_CELL_IN_PIN     NAND2X2              A

ADD_CELL_OUT_PIN  (only used for YOSYS+ABC flow)
  Info:    Instructs DELPHI to treat a pin on a cell as an output
  Syntax:  ADD_CELL_OUT_PIN <ABC_CELL_NAME>    <ABC_PIN_NAME>
  Example: ADD_CELL_OUT_PIN     NAND2X2              Y


##############################################################################
#### 1.2 Constraining synthesis

Constraining synthesis within Synopsys DC to only DSENT-compatible cells the
requires properly setting the "dont_use" attribute for all non-supported cells
in the target standard cell library. This needs to be done before running
synthesis through the dc_shell using the "set_dont_use" and "remove_attribute"
DC commands, e.g.:

# Setting the dont_use attribute for cells that are not compatible with DSENT
# Note: To find the name of your target library run the list_libs command
  set_dont_use <your_lib_name>/OR3X1
  set_dont_use <your_lib_name>/OR3X2
  set_dont_use <your_lib_name>/OR3X4

# Alternatively you can first set the dont_use attribute for all cells
```

```
  set_dont_use <your_lib_name>/*
# And then selectively remove the dont_use attribute for the subset of cells
# that have a mapping to DSENT-compatible cells
  remove_attribute <your_lib_name>/NAND2X1       dont_use
  remove_attribute <your_lib_name>/NAND2X2       dont_use
  remove_attribute <your_lib_name>/NAND2X4       dont_use
```

If the user has already prepared the DC-DSENT mapping file described above, the DELPHI tool can automatically generate a tcl script that contains the commands needed to constrain synthesis by invoking it with these command-line options:

```
  ./DELPHI --gen_dc_constrain_script --dc_lib_name <your_lib_name> \
--dc_dsent_map_file <your_dc_dsent_map_file>
```

The resulting tcl script <your_lib_name>_constrain.tcl needs to be sourced before running synthesis, e.g.:

```
  ...
  source ./<your_lib_name>_constrain.tcl
  compile_ultra
  ...
```

Note: To verify that the dont_use attributes have been properly set you can check the output of the "report_lib <your_lib_name>" command. Cells that have a "u" next to them have the "dont_use" attribute set.

```
##############################################################################
#### 1.3 Generating the necessary synthesis reports
```

DELPHI relies on various synthesis reports that are parsed and analyzed to build a DSENT model. Once constrained synthesis in DC is finished, you can generate these reports by running the following commands at a dc_shell:

```
  ungroup -all -flatten
  report_cell -nosplit -connections > dc_cell_report
```

```
report_port -nosplit > dc_port_report
report_clock_tree -nosplit > dc_clock_report


################################################################################
#### 2. Generating a DSENT model and Running DSENT
################################################################################

################################################################################
#### 2.1 Processing synthesis results to generate a DSENT model

The DC reports generated during the last step of DC synthesis are now ready to
be fed into the DELPHI tool to generate a DSENT model that corresponds to the
synthesized netlist. In addition to the generated reports the user needs to
specify the names of any clock or reset signals in the design using the
"--clock_net_name" and "--reset_net_name" command-line options.  Below is an
example of an invocation of the DELPHI tool:

  ./DELPHI --dc_dsent_map_file <your_dc_dsent_map_file> --dc_cells_file \
<dc_cell_report> --dc_ports_file <dc_port_report> --dc_clock_tree_file \
<dc_clock_report> --clock_net_name <clk_net_name> --reset_net_name <reset_net_name>

Note: If the DC synthesis target library used a ns timescale (instead of ps),
make sure to also add the flag "--ns_timescale". To check the time units of your
DC library run the command "report_units" at the dc_shell.

For a full list of DELPHI command-line options along with their descriptions,
run:
  ./DELPHI --help

Once the synthesis reports are processed, the DELPHI tool generates the
following files:

 - C++ code that corresponds to a DSENT netlist for the original netlist:
```

DELPHI_module.*

- Sample configuration files targeting different technology models to be used when running DSENT:

  DELPHI_module*.cfg

##

2.2 Compiling and running the new DSENT model

Extract the delphi_dsent0.91.tar.gz tarball (tar -zxvf delphi_dsent0.91.tar.gz) and then copy the generated C++ files (DELPHI_module.cc and DELPHI_module.h) to the delphi_dsent0.91/model/delphi/ directory and the configuration files (DELPHI_module*.cfg) to the delphi_dsent0.91/configs directory.

To compile the generated DSENT models navigate to the delphi_dsent0.91 directory and run:

```
make
```

Once compilation is succesfully finished you are now ready to run DSENT using one of the generated (or a custom) configuration file, e.g.:

```
./dsent -cfg configs/DELPHI_module_32.cfg
```

For more information on running DSENT and the various options in its configuration files, please see the README file under delphi_dsent0.91.

Appendix C

IRIS Event Specification and Output Examples

C.1 IRIS Event Specification

```
# IRIS EVENT SPEC - Includes most basic IRIS events
# Lines that sart with # are comments
# Default prefix is "__IRIS__"
# Any parameters in [] are optional

# General events - not needed for SW
IRIS_RESET_ALL
IRIS_FINISH

# IrisConfig - Print system/simulation configuration information
IRIS_PARAM_SETTING [@ cycle] <param> <setting>

# IrisLogger - Logging and printing events
IRIS_INFO [@ cycle] <message>
IRIS_LOG [@ cycle] <log_message>
```

```
IRIS_WARN [@ cycle] <warning_message>
IRIS_ERROR [@ cycle] <error_message>

# IRIS_COUNTER events
NEW_IRIS_COUNTER <iris_counter_name> [@ cycle]
IC_INC_BY_ONE <iris_counter_name> [@ cycle]
IC_INC_BY_N <iris_counter_name> [@ cycle] <n>
IC_DEC_BY_ONE <iris_counter_name> [@ cycle]
IC_DEC_BY_N <iris_counter_name> [@ cycle] <n>
IC_RESET <iris_counter_name> [@ cycle]

# IRIS_COUNTER_1D
NEW_IRIS_COUNTER_1D <iris_counter_vector_name> <size>
IC_1D_INC_BY_ONE <iris_counter_name> [@ cycle] <id>
IC_1D_INC_BY_N <iris_counter_name> [@ cycle] <id> <n>
IC_1D_DEC_BY_ONE <iris_counter_name> [@ cycle] <id>
IC_1D_DEC_BY_N <iris_counter_name> [@ cycle] <id> <n>
IC_1D_RESET <iris_counter_name> [@ cycle] <id>

# IRIS_COUNTER_2D
NEW_IRIS_COUNTER_2D <iris_counter_vector_name> <size_x> <size_y>
IC_2D_INC_BY_ONE <iris_counter_name> [@ cycle] <i> <j>
IC_2D_INC_BY_N <iris_counter_name> [@ cycle] <i> <j> <n>
IC_2D_DEC_BY_ONE <iris_counter_name> [@ cycle] <i> <j>
IC_2D_DEC_BY_N <iris_counter_name> [@ cycle] <i> <j> <n>
IC_2D_RESET <iris_counter_name> [@ cycle] <i> <j>

# IRIS_SAMPLER events
# <num_bins> defines bins used for histogram
# <bin_min> <bin_max> defines range of values used for histogram and size of bins
NEW_IRIS_SAMPLER <iris_sampler_name> <num_bins> <bin_min> <bin_max>
IS_ADD_SAMPLE <iris_sampler_name> [@ cycle] <sample>
IS_CLEAR <iris_sampler_name> [@ cycle]
```

```
# IRIS_SAMPLER_1D
NEW_IRIS_SAMPLER_1D <iris_sampler_name> <size> <num_bins> <bin_min> <bin_max>
IS_1D_ADD_SAMPLE <iris_sampler_name> [@ cycle] <id> <sample>
IS_1D_CLEAR <iris_sampler_name> [@ cycle] <id>

# IRIS_SAMPLER_2D
NEW_IRIS_SAMPLER_2D <iris_sampler_name> <size_x> <size_y> <num_bins> <bin_min> <bin_max>
IS_2D_ADD_SAMPLE <iris_sampler_name> [@ cycle] <i> <j> <sample>
IS_2D_CLEAR <iris_sampler_name> [@ cycle] <i> <j>

# IRIS_TRACE_SAMPLER events
# num_quantums and quantum_length not needed in SW - use any values
NEW_IRIS_TRACE_SAMPLER <iris_trace_sampler_name> <num_quantums> <quantum_length>
ITS_ADD_SAMPLE <iris_trace_sampler_name> [@ cycle] <sample>
# Starts new quantum - call periodically in SW
ITS_NEW_QUANTUM <iris_trace_sampler_name> [@ cycle]
ITS_CLEAR <iris_trace_sampler_name> [@ cycle]

# IRIS_EVENT_TRACKER
NEW_IRIS_EVENT_TRACKER <iris_event_tracker_name> [@ cycle] <num_events>
IET_EVENT_NAME <iris_event_tracker_name> [@ cycle] <event_id> <event_name>
IET_ADD_EVENT <iris_event_tracker_name> [@ cycle] <event_id>

# IRIS_EVENT_TRACKER_1D
NEW_IRIS_EVENT_TRACKER_1D <iris_event_tracker_name> [@ cycle] <size> <num_events>
IET_1D_EVENT_NAME <iris_event_tracker_name> [@ cycle] <id> <event_id> <event_name>
IET_1D_ADD_EVENT <iris_event_tracker_name> [@ cycle] <id> <event_id>

# IRIS_EVENT_TRACKER_2D
NEW_IRIS_EVENT_TRACKER_2D <iris_event_tracker_name> [@ cycle] <size_x> <size_y> <num_events>
IET_2D_EVENT_NAME <iris_event_tracker_name> [@ cycle] <i> <j> <event_id> <event_name>
IET_2D_ADD_EVENT <iris_event_tracker_name> [@ cycle] <i> <j> <event_id>
```

C.2 IRIS Event Output Example

```
__IRIS__ IRIS_PARAM_SETTING NETWORK_PARAMETERS_FILE
mesh_16RTs_2VCs_8BD_32DW_SepIFRoundRobinAlloc_4RTsPerRow_4RTsPerCol_parameters.bsv
__IRIS__ IRIS_PARAM_SETTING NETWORK_LINKS
mesh_16RTs_2VCs_8BD_32DW_SepIFRoundRobinAlloc_4RTsPerRow_4RTsPerCol_links.bsv
__IRIS__ IRIS_PARAM_SETTING NETWORK_ROUTING_FILE_PREFIX
mesh_16RTs_2VCs_8BD_32DW_SepIFRoundRobinAlloc_4RTsPerRow_4RTsPerCol_routing_
__IRIS__ IRIS_PARAM_SETTING NUM_ROUTERS 16
__IRIS__ IRIS_PARAM_SETTING NUM_IN_PORTS 5
__IRIS__ IRIS_PARAM_SETTING NUM_OUT_PORTS 5
__IRIS__ IRIS_PARAM_SETTING NUM_VCS 2
__IRIS__ IRIS_PARAM_SETTING FLIT_BUFFER_DEPTH 8
__IRIS__ IRIS_PARAM_SETTING FLIT_DATA_WIDTH 32
__IRIS__ IRIS_PARAM_SETTING NUM_LINKS 48
__IRIS__ IRIS_PARAM_SETTING TRACE_FILE_PREFIX uniform_45_
__IRIS__ NEW_IRIS_COUNTER_1D packets_per_src 16
__IRIS__ NEW_IRIS_COUNTER_1D flits_per_src 16
__IRIS__ NEW_IRIS_COUNTER_1D packets_per_dst 16
__IRIS__ NEW_IRIS_COUNTER_1D flits_per_dst 16
__IRIS__ NEW_IRIS_COUNTER_2D packets_by_src_dst 16 16
__IRIS__ NEW_IRIS_COUNTER_2D flits_by_src_dst 16 16
__IRIS__ NEW_IRIS_SAMPLER_1D flit_latency_per_dst 16 256 0 255
__IRIS__ NEW_IRIS_SAMPLER_1D packet_latency_per_dst 16 256 0 255
__IRIS__ NEW_IRIS_COUNTER @ 0 total_packets_sent
__IRIS__ NEW_IRIS_COUNTER @ 0 total_flits_sent
__IRIS__ NEW_IRIS_COUNTER @ 0 total_packets_received
__IRIS__ NEW_IRIS_COUNTER @ 0 total_flits_received
__IRIS__ NEW_IRIS_SAMPLER @ 0 total_packet_latency 256 0 255
__IRIS__ NEW_IRIS_SAMPLER @ 0 total_flit_latency 256 0 255
__IRIS__ NEW_IRIS_SAMPLER @ 0 credits_available_vc0 256 0 255
__IRIS__ NEW_IRIS_SAMPLER @ 0 credits_available_vc1 256 0 255
__IRIS__ NEW_IRIS_SAMPLER @ 0 credits_available_corner 256 0 255
__IRIS__ NEW_IRIS_SAMPLER @ 0 credits_available_edge 256 0 255
```

```
__IRIS__ NEW_IRIS_SAMPLER @ 0 credits_available_center 256 0 255
__IRIS__ NEW_IRIS_TRACE_SAMPLER @ 0 packet_latency_over_time 100 500
__IRIS__ NEW_IRIS_TRACE_SAMPLER @ 0 flit_latency_over_time 100 1000
__IRIS__ NEW_IRIS_EVENT_TRACKER @ 0 corner_rt_status 5
__IRIS__ IET_SET_EVENT_NAME @ 0 corner_rt_status 0 Idle
__IRIS__ IET_SET_EVENT_NAME @ 0 corner_rt_status 1 Sending_no_backpressure
__IRIS__ IET_SET_EVENT_NAME @ 0 corner_rt_status 2 Sending_low_backpressure
__IRIS__ IET_SET_EVENT_NAME @ 0 corner_rt_status 3 Sending_high_backpressure
__IRIS__ IET_SET_EVENT_NAME @ 0 corner_rt_status 4 Blocked_out_of_credits
__IRIS__ NEW_IRIS_COUNTER @ 0 Flits_Sent_By_Router
...
__IRIS__ IC_INC_BY_ONE @ 9157 Flits_Sent_By_Router
__IRIS__ IC_1D_INC_BY_ONE @ 9158 flits_per_dst 1
__IRIS__ IS_ADD_SAMPLE @ 9158 total_flit_latency 13
__IRIS__ ITS_ADD_SAMPLE @ 9158 flit_latency_over_time 13
__IRIS__ IS_1D_ADD_SAMPLE @ 9158 flit_latency_per_dst 1 13
__IRIS__ IC_1D_INC_BY_ONE @ 9158 flits_per_dst 3
__IRIS__ IS_ADD_SAMPLE @ 9158 total_flit_latency 20
__IRIS__ ITS_ADD_SAMPLE @ 9158 flit_latency_over_time 20
__IRIS__ IS_1D_ADD_SAMPLE @ 9158 flit_latency_per_dst 3 20
__IRIS__ IC_1D_INC_BY_ONE @ 9158 flits_per_dst 5
__IRIS__ IS_ADD_SAMPLE @ 9158 total_flit_latency 3
__IRIS__ ITS_ADD_SAMPLE @ 9158 flit_latency_over_time 3
__IRIS__ IS_1D_ADD_SAMPLE @ 9158 flit_latency_per_dst 5 3
__IRIS__ IC_1D_INC_BY_ONE @ 9158 flits_per_dst 7
__IRIS__ IS_ADD_SAMPLE @ 9158 total_flit_latency 11
__IRIS__ ITS_ADD_SAMPLE @ 9158 flit_latency_over_time 11
__IRIS__ IS_1D_ADD_SAMPLE @ 9158 flit_latency_per_dst 7 11
__IRIS__ IC_1D_INC_BY_ONE @ 9158 flits_per_dst 9
__IRIS__ IS_ADD_SAMPLE @ 9158 total_flit_latency 2
__IRIS__ ITS_ADD_SAMPLE @ 9158 flit_latency_over_time 2
__IRIS__ IS_1D_ADD_SAMPLE @ 9158 flit_latency_per_dst 9 2
__IRIS__ IC_1D_INC_BY_ONE @ 9158 flits_per_dst 11
__IRIS__ IS_ADD_SAMPLE @ 9158 total_flit_latency 19
```

```
__IRIS__ ITS_ADD_SAMPLE @ 9158 flit_latency_over_time 19
__IRIS__ IS_1D_ADD_SAMPLE @ 9158 flit_latency_per_dst 11 19
__IRIS__ IC_1D_INC_BY_ONE @ 9158 flits_per_dst 14
__IRIS__ IS_ADD_SAMPLE @ 9158 total_flit_latency 6
__IRIS__ ITS_ADD_SAMPLE @ 9158 flit_latency_over_time 6
__IRIS__ IS_1D_ADD_SAMPLE @ 9158 flit_latency_per_dst 14 6
__IRIS__ IC_INC_BY_N @ 9158 total_packets_received 0
__IRIS__ IC_INC_BY_N @ 9158 total_flits_received 7
__IRIS__ IET_ADD_EVENT @ 9158 corner_rt_status 2
__IRIS__ IC_1D_INC_BY_ONE @ 9158 flits_per_src 0
__IRIS__ IC_2D_INC_BY_ONE @ 9158 flits_by_src_dst 0 5
__IRIS__ IET_ADD_EVENT @ 9158 edge_rt_status 0
__IRIS__ IC_1D_INC_BY_ONE @ 9158 flits_per_src 2
__IRIS__ IC_2D_INC_BY_ONE @ 9158 flits_by_src_dst 2 3
__IRIS__ IC_1D_INC_BY_ONE @ 9158 flits_per_src 4
__IRIS__ IC_2D_INC_BY_ONE @ 9158 flits_by_src_dst 4 7
__IRIS__ IC_1D_INC_BY_ONE @ 9158 flits_per_src 6
__IRIS__ IC_2D_INC_BY_ONE @ 9158 flits_by_src_dst 6 8
__IRIS__ IC_1D_INC_BY_ONE @ 9158 flits_per_src 7
__IRIS__ IC_2D_INC_BY_ONE @ 9158 flits_by_src_dst 7 8
__IRIS__ IET_ADD_EVENT @ 9158 center_rt_status 0
__IRIS__ IC_1D_INC_BY_ONE @ 9158 flits_per_src 10
__IRIS__ IC_2D_INC_BY_ONE @ 9158 flits_by_src_dst 10 9
__IRIS__ IC_1D_INC_BY_ONE @ 9158 flits_per_src 12
__IRIS__ IC_2D_INC_BY_ONE @ 9158 flits_by_src_dst 12 7
__IRIS__ IC_1D_INC_BY_ONE @ 9158 packets_per_src 12
__IRIS__ IC_2D_INC_BY_ONE @ 9158 packets_by_src_dst 12 7
__IRIS__ IC_INC_BY_N @ 9158 total_packets_sent 1
__IRIS__ IC_INC_BY_N @ 9158 total_flits_sent 7
__IRIS__ IS_ADD_SAMPLE credits_available_vc0 7
__IRIS__ IS_ADD_SAMPLE credits_available_corner 7
__IRIS__ IS_ADD_SAMPLE credits_available_vc1 8
__IRIS__ IRIS_ERROR @ 9273 DeadlockChecker - deadlock detected!
```

References

[1] Xilinx CORE Generator System. http://www.xilinx.com/tools/coregen.htm. 3.2.4

[2] Impulse CoDeveloper. http://www.impulseaccelerated.com. 3.3

[3] Pyevolve. http://pyevolve.sourceforge.net. 5.3.1, 5.4.1

[4] The PyPy Python Interpreter and Just-In-Time Compiler. http://www.pypy.org. 6.3.2

[5] Riverside Optimizing Compiler for Configurable Computing. http://roccc.cs.ucr.edu/. 3.3

[6] SciPy Interpolation. http://docs.scipy.org/doc/scipy/reference/interpolate.html. 7.1

[7] Synopsys VCS. http://www.synopsys.com/Tools/Verification/FunctionalVerification/Pages/VCS.aspx. 6.3.2

[8] SystemC: The Open SystemC Initiative. http://www.systemc.org. 3.3

[9] R. Abdel-Khalek and V. Bertacco. Functional Post-Silicon Diagnosis and Debug for Networks-on-Chip. In *Computer-Aided Design (ICCAD), 2012 IEEE/ACM International Conference on*, pages 557–563, Nov 2012. 6.4

[10] R. Abdel-Khalek and V. Bertacco. DiAMOND: Distributed Alteration of Messages for On-Chip Network Debug. In *Networks-on-Chip (NoCS), 2014 Eighth IEEE/ACM International Symposium on*, pages 127–134, Sept 2014. 6.4

[11] N. Agarwal, T. Krishna, Li-Shiuan Peh, and N.K. Jha. GARNET: A Detailed On-Chip Network Model Inside a Full-System Simulator. In *ISPASS*, April 2009. 4.3.2

[12] Alireza Monemi. Open Source Low Latency Network-on-Chip (NoC) Router RTL in Verilog for FPGA. http://nocrouter.codeplex.com/. 2.3

[13] Charles J Alpert, Dinesh P Mehta, and Sachin S Sapatnekar. *Handbook of Algorithms for Physical Design Automation*. CRC Press, 2008. 5.5

[14] Altera. Design Debugging Using the SignalTap II Logic Analyzer. `https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/qts/qts_qii53009.pdf`. 6.1

[15] Altera. Qsys System Integration Tool. `http://www.altera.com/products/software/quartus-ii/subscription-edition/qsys/qts-qsys.html`. 3.3

[16] Thomas Back. An Overview of Evolutionary Algorithms for Parameter Optimization. *Evolutionary Computation*, 1(1):1–23, 1993. 5.2

[17] Daniel U. Becker. *Efficient Microarchitecture for Network-on-Chip Routers*. PhD thesis, 2012. 5.1, 5.4

[18] Berkeley Logic Synthesis and Verification Group. ABC: A System for Sequential Synthesis and Verification. `http://www.eecs.berkeley.edu/~alanmi/abc/`. 4.3.1

[19] D. Bertozzi, A. Jalabert, Srinivasan Murali, R. Tamhankar, S. Stergiou, L. Benini, and G. De Micheli. NoC Synthesis Flow for Customized Domain Specific Multiprocessor Systems-on-Chip. *Parallel and Distributed Systems, IEEE Transactions on*, 2005. 3.3

[20] Bluespec, Inc. Bluespec SystemVerilog. `http://www.bluespec.com/high-level-synthesis-tools.html`. 1.1, 3.1, 3.2.4, 3.3

[21] D. Brand and C. Visweswariah. Inaccuracies in Power Estimation During Logic Synthesis. In *ICCAD*, November 1996. 4.2.1

[22] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *ISCA*, June 2000. 4.5

[23] D. Bruni, A. Bogliolo, and L. Benini. Statistical Design Space Exploration for Application-Specific Unit Synthesis. In *Design Automation Conference (DAC)*, pages 641–646, 2001. 5.5

[24] Paolo Campigotto, Andrea Passerini, and Roberto Battiti. Active Learning of Pareto Fronts. *IEEE Transactions on Neural Networks and Learning Systems*, 25(3):506–519, March 2014. 5.5

[25] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. LegUp: High-level Synthesis for

FPGA-based Processor/Accelerator Systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, New York, NY, USA, 2011. ACM. 3.3

[26] Christoph Albrecht, Cadence Research Laboratories at Berkeley. IWLS 2005 Benchmarks. `http://iwls.org/iwls2005/benchmarks.html`. 4.4.1

[27] Eric S. Chung and Michael K. Papamichael. ShrinkWrap: Compiler-Enabled Optimization and Customization of Soft Memory Interconnects. In *FCCM*, 2013. 2.1, 2.4.2

[28] Eric S. Chung, James C. Hoe, and Ken Mai. CoRAM: An In-Fabric Memory Abstraction for FPGA-based Computing. In *FPGA*, 2011. 2.4.2

[29] Eric S. Chung, Michael K. Papamichael, Gabriel Weisz, James C. Hoe, and Ken Mai. Prototype and Evaluation of the CoRAM Memory Architecture for FPGA-Based Computing. In *FPGA*, 2012. 2.1, 2.4.2

[30] Calin Ciordas, Twan Basten, Andrei Rădulescu, Kees Goossens, and Jef Van Meerbergen. An Event-based Monitoring Service for Networks on Chip. *ACM Trans. Des. Autom. Electron. Syst.*, 10(4):702–723, October 2005. 6.4

[31] William Dally and Brian Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003. ISBN 0122007514. 2.2, 2.3.2

[32] W.J. Dally and C.L. Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Transactions on Computers*, 1987. 2.2, 2.3.2

[33] W.J. Dally and B. Towles. Route Packets, Not Wires: On-chip Interconnection Networks. In *Design Automation Conference, 2001. Proceedings*, 2001. 2.1, 3.3

[34] A. DeHon, J. Adams, M. deLorimier, N. Kapre, Y. Matsuda, H. Naeimi, M. Vanier, and M. Wrighton. Design Patterns for Reconfigurable Computing. In *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*, pages 13–23, 2004. 3.3

[35] Daniel Y. Deng, Daniel Lo, Greg Malysa, Skyler Schneider, and G. Edward Suh. Flexible and Efficient Instruction-Grained Run-Time Monitoring Using On-Chip Reconfigurable Fabric. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 137–148, Washington, DC, USA, 2010. IEEE Computer Society. 6.4

[36] R.H. Dennard, F.H. Gaensslen, V.L. Rideout, E. Bassous, and A.R. LeBlanc. Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions. *Solid-State Circuits, IEEE Journal of*, October 1974. 1, 3.1

[37] Department of Computer Systems, Tampere University of Technology, Finland . NoCBench. `http://www.tkt.cs.tut.fi/research/nocbench/`. 2.7

[38] R.P. Dick and N.K. Jha. MOGAC: A Multiobjective Genetic Algorithm for Hardware-Software Cosynthesis of Distributed Embedded Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(10), 1998. 5.5

[39] M. El Shobaki and L. Lindh. A Hardware and Software Monitor for High-Level System-on-Chip Verification. In *Quality Electronic Design, 2001 International Symposium on*, pages 56–61, 2001. 6.4

[40] H. Esbensen and E.S. Kuh. Design Space Exploration Using the Genetic Algorithm. In *IEEE International Symposium on Circuits and Systems*, volume 4, pages 500–503, 1996. 5.5

[41] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon and the End of Multicore Scaling. ISCA '11, New York, NY, USA, 2011. 1

[42] L. Fiorin, G. Palermo, and C. Silvano. A Configurable Monitoring Infrastructure for NoC-Based Architectures. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 22(11):2436–2440, Nov 2014. 6.4

[43] G. Schelle and D. Grunwald. Exploring FPGA Network on Chip Implementations Across Various Application and Network Loads. In *FPL*, 2008. 2.3

[44] Daniel D. Gajski, Nikil D. Dutt, Allen C.-H. Wu, and Steve Y.-L. Lin. High Level Synthesis: Introduction to Chip and System Design. 1992. 1.1, 3.1

[45] Genesis 2. Creating Chip Generators. `http://genesis2.stanford.edu`. 3.2.4, 3.3

[46] P. Guerrier and A. Greiner. A Generic Architecture for On-Chip Packet-Switched Interconnections. In *Design, Automation and Test in Europe Conference and Exhibition 2000. Proceedings*, 2000. 3.3

[47] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 1998. ISBN 0132733501. 8.1

[48] A.B.T. Hopkins and K.D. McDonald-Maier. Debug Support for Complex Systems-on-Chip:

A Review. *Computers and Digital Techniques, IEEE Proceedings on*, 153(4):197–207, July 2006. 6.4

[49] IBM. The Coreconnect Bus Architecture. `https://www-01.ibm.com/chips/techlib/techlib.nsf/products/CoreConnect_Bus_Architecture`, 1999. 3.3

[50] International Technology Roadmap for Semiconductors. `http://www.itrs.net`. 4.2.2

[51] M. Ismail and G.E. Suh. Fast Development of Hardware-Based Run-Time Monitors Through Architecture Framework and High-Level Synthesis. In *Computer Design (ICCD), 2012 IEEE 30th International Conference on*, pages 393–400, Sept 2012. 6.4

[52] Andrew B. Kahng, Bin Li, Li-Shiuan Peh, and Kambiz Samadi. ORION 2.0: A Fast and Accurate NoC Power and Area Model for Early-Stage Design Space Exploration. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE'09, pages 423–428, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association. 4.5

[53] K. Keutzer, A. R. Newton, J. M. Rabaey, and A. Sangiovanni-Vincentelli. System-Level Design: Orthogonalization of Concerns and Platform-based Design. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 19(12), November 2006. 1.1, 3.1, 3.3

[54] J. Knowles. ParEGO: A Hybrid Algorithm with On-Line Landscape Approximation for Expensive Multiobjective Optimization Problems. *IEEE Transactions on Evolutionary Computation*, 10(1):50–66, February 2006. 5.5

[55] Georgios Kornaros and Dionisios Pnevmatikatos. A Survey and Taxonomy of On-chip Monitoring of Multicore Systems-on-chip. *ACM Trans. Des. Autom. Electron. Syst.*, 18(2):17:1–17:38, April 2013. 6.4

[56] Evriklis Kounalakis. The Mythical IP Block: An Investigation of Contemporary IP Characteristics. Technical Report ICS-TR366, Institute of Computer Science, Foundation for Research and Technology - Hellas (FORTH), October 2005. 4.2.1

[57] Vyas Krishnan and Srinivas Katkoori. A Genetic Algorithm for the Design Space Exploration of Datapaths During High-Level Synthesis. *IEEE Transactions on Evolutionary Computation*, 10(3):213–229, 2006. 5.5

[58] Ian Kuon and Jonathan Rose. Measuring the Gap Between FPGAs and ASICs. In *FPGA'06: Proceedings of the 2006 ACM/SIGDA 14th International Symposium on Field*

Programmable Gate Arrays, pages 21–30, New York, NY, USA, 2006. ACM. 2.4.1

[59] G. Kurian, S.M. Neuman, G. Bezerra, A. Giovinazzo, S. Devadas, and J.E. Miller. Power Modeling and Other New Features in the Graphite Simulator. In *ISPASS*, March 2014. 4.2.2, 4.3.2

[60] J. Lee and L. Shannon. The Effect of Node Size, Heterogeneity, and Network Size on FPGA based NoCs. In *FPT*, 2009. 2.4.1

[61] Jong Chul Lee, F. Kouteib, and R. Lysecky. Event-Driven Framework for Configurable Runtime System Observability for SoC Designs. In *Test Conference (ITC), 2012 IEEE International*, pages 1–10, Nov 2012. 6.4

[62] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *MICRO'09: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 469–480, New York, NY, USA, 2009. ACM. 4.3, 4.5

[63] M. Karo; M. Hluchyj; S. Morgan. Input Versus Output Queuing on a Space-Division Packet Switch. In *IEEE Transactions on Communications*, 1987. 2.2, 2.3.2

[64] R. Marculescu, D. Marculescu, and M. Pedram. Probabilistic Modeling of Dependencies During Switching Activity Analysis. *IEEE Transactions on CAD*, 17:73–83, Feb 1998. 3

[65] G. Martin and G. Smith. High-Level Synthesis: Past, Present, and Future. *Design Test of Computers, IEEE*, 26(4):18–25, July 2009. 1.1, 3.1

[66] Mentor Graphics. Catapult C. `http://www.mentor.com/esl`, 2009. 3.3

[67] Michael K. Papamichael. CONNECT Network Editor. `http://users.ece.cmu.edu/~mpapamic/connect/network_editor/`, . 3.2.2

[68] Michael K. Papamichael. CONNECT NoC Generation Framework. `http://users.ece.cmu.edu/~mpapamic/connect/`, . 1.2, 2.1, 4.4.1, 5.1

[69] P. A. Milder, F. Franchetti, J. C. Hoe, and M. Püschel. Formal Datapath Representation and Manipulation for Implementing DSP Transforms. In *Proc. Design Automation Conference*, pages 385–390, 2008. 3.3

[70] Peter Milder, Franz Franchetti, James C Hoe, and Markus Püschel. Computer Generation of Hardware for Linear Digital Signal Processing Transforms. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 17(2):15, 2012. 5.1

[71] J.E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A Agarwal. Graphite: A Distributed Parallel Simulator for Multicores. In *HPCA*, Jan 2010. 4.3.2

[72] MIT. DSENT (Design Space Exploration for Network Tool). `https://sites.google.com/site/mitdsent/`. 4.2.2

[73] Mohamed S. Abdelfattah and Vaughn Betz. FPGA NoC Designer. `http://www.eecg.utoronto.ca/~mohamed/noc_designer.html`. 2.7

[74] Gordon E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38 (8), April 1965. 1, 3.1

[75] A. Morvan, S. Derrien, and P. Quinton. Efficient Nested Loop Pipelining in High Level Synthesis Using Polyhedral Bubble Insertion. In *Field-Programmable Technology (FPT), 2011 International Conference on*, 2011. 3.3

[76] Shashidhar Mysore, Banit Agrawal, Navin Srivastava, Sheng-Chih Lin, Kaustav Banerjee, and Timothy Sherwood. 3D Integration for Introspection. *IEEE Micro*, 27(1):77–83, January 2007. 6.4

[77] Grace Nordin, Peter A. Milder, James C. Hoe, and Markus Püschel. Automatic Generation of Customized Discrete Fourier Transform IPs. In *Design Automation Conference (DAC)*, pages 471–474, 2005. 3.3

[78] North Carolina State University. NCSU FreePDK. `http://www.eda.ncsu.edu/wiki/FreePDK`. 4.3.1

[79] OpenCores. OpenCores: Open Source Hardware IP-Cores. `http://opencores.org/`. 4.3.2

[80] M. Palesi and T. Givargis. Multi-Objective Design Space Exploration Using Genetic Algorithms. In *CODES*, 2002. 5.5

[81] Michael K. Papamichael and James C. Hoe. CONNECT: Re-Examining Conventional Wisdom for Designing NoCs in the Context of FPGAs. In *FPGA*, 2012. 1.2, 2.1, 2.3.2, 2.4.1, 2.4.1, 4.4.1, 5.1

[82] Michael K. Papamichael and James C. Hoe. The CONNECT Network-on-Chip IP Generator. *IEEE Computer*, To Appear. 1.2, 2.1

[83] Michael K. Papamichael, Peter Milder, and James C. Hoe. Nautilus: Fast Automated IP Design Space Search Using Guided Genetic Algorithms. In *DAC'15: Proceedings of the*

52nd Annual Design Automation Conference, 2015. 1.2, 3.4

[84] M.K. Papamichael, C. Cakir, Chen Suny Chia-Hsin, O. Cheny, J.C. Hoe, K. Mai, Li-Shiuan Pehy, and V. Stojanovic. DELPHI: A Framework for RTL-Based Architecture Design Evaluation Using DSENT Models. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, March 2015. 1.2, 3.4

[85] Ritesh Parikh and Valeria Bertacco. Formally Enhanced Runtime Verification to Ensure NoC Functional Correctness. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 410–419, New York, NY, USA, 2011. ACM. 6.4

[86] Harish Patil, Robert Cohn, Mark Charney, Rajiv Kapoor, Andrew Sun, and Anand Karunanidhi. Pinpointing Representative Portions of Large Intel Itanium Programs with Dynamic Instrumentation. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pages 81–92, Washington, DC, USA, 2004. IEEE Computer Society. 8.1

[87] A. Pinto, L.P. Carloni, and A. Sangiovanni-Vincentelli. COSI: A Framework for the Design of Interconnection Networks. *Design Test of Computers, IEEE*, 2008. 3.3

[88] Pontificia Universidade Catolica do Rio Grande do Sul - PUCRS. ATLAS. `https://corfu.pucrs.br/redmine/projects/atlas`. 2.7

[89] A. Putnam, A.M. Caulfield, E.S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G.P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P.Y. Xiao, and D. Burger. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 13–24, June 2014. 1, 8

[90] Ivan Ratkovic, Oscar Palomar, Milan Stanic, Osman Unsal, Adrian Cristal, and Mateo Valero. Physical vs. Physically-Aware Estimation Flow: Case Study of Design Space Exploration of Adders. In *VLSI (ISVLSI)*, July 2014. 4.2.1

[91] Real-Time Systems Research Group, University of York. Blueshell. `https://rtslab.wikispaces.com/Blueshell`. 2.7

[92] Robert Mullins. Netmaker. `http://www-dyn.cl.cam.ac.uk/~rdm34/wiki`. 2.3, 2.7

[93] Kyeong Keol Ryu and V.J. Mooney. Automated Bus Generation for Multiprocessor SoC Design. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 23(11), 2004. 3.3

[94] D Saha, R S Mitra, and A Basu. Hardware Software Partitioning using Genetic Algorithm. In *International Conference on VLSI Design*, pages 155–160, 1997. 5.5

[95] O. Shacham, O. Azizi, M. Wachs, W. Qadeer, Z. Asgar, K. Kelley, J.P. Stevenson, S. Richardson, M. Horowitz, B. Lee, A. Solomatnikov, and A. Firoozshahian. Rethinking Digital Design: Why Design Must Change. *Micro, IEEE*, 30(6), 2010. 1.1, 3.1, 3.3

[96] O. Shacham, S. Galal, S. Sankaranarayanan, M. Wachs, J. Brunhaver, A. Vassiliev, M. Horowitz, A. Danowitz, W. Qadeer, and S. Richardson. Avoiding Game Over: Bringing Design to the Next Level. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, 2012. 3.3

[97] Ofer Shacham. *Chip Multiprocessor Generator: Automatic Generation of Custom and Heterogeneous Compute Platforms*. PhD thesis, 2011. 3.3, 5.1

[98] Stanford Concurrent VLSI Architecture Group. Open Source Network-on-Chip Router RTL. `https://nocs.stanford.edu/cgi-bin/trac.cgi/wiki/Resources/Router`. (document), 2.3, 2.4, 2.4.1, 2.7, 3.2.2

[99] Chen Sun, C.-H.O. Chen, G. Kurian, Lan Wei, J. Miller, A Agarwal, Li-Shiuan Peh, and V. Stojanovic. DSENT - A Tool Connecting Emerging Photonics with Electronics for Opto-Electronic Networks-on-Chip Modeling. In *NOCS*, 2012. 3.4, 4, 4.1, 4.2.2, 4.3.2

[100] Shan Tang and Qiang Xu. A Multi-Core Debug Platform for NoC-Based Systems. In *Design, Automation Test in Europe Conference Exhibition, 2007. DATE '07*, pages 1–6, April 2007. 6.4

[101] Shan Tang and Qiang Xu. A Debug Probe for Concurrently Debugging Multiple Embedded Cores and Inter-Core Transactions in NoC-Based Systems. In *Design Automation Conference, 2008. ASPDAC 2008. Asia and South Pacific*, pages 416–421, March 2008. 6.4

[102] The Chisel Hardware Construction Language. `http://chisel.eecs.berkeley.edu`. 1.1, 3.1, 3.3

[103] David Tarjan Shyamkumar Thoziyoor, David Tarjan, and Shyamkumar Thoziyoor. Cacti 4.0. Technical Report HPL-2006-86, HP Labs, 2006. 4.3, 4.3.3, 4.5

[104] Saurabh K Tiwary, Pragati K Tiwary, and Rob A Rutenbar. Generation of Yield-Aware

Pareto Surfaces for Hierarchical. In *Design Automation Conference (DAC)*, pages 31–36, 2006. 5.5

[105] University of Southern California. System Power Optimization and Regulation Technology (SPORT) Lab. `http://sportlab.usc.edu/`. 4.3.1

[106] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation Cores: Reducing the Energy of Mature Computations. In *ASPLOS'10: Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 205–218, New York, NY, USA, 2010. ACM. 1

[107] B. Vermeulen and K. Goossens. A Network-on-Chip Monitoring Infrastructure for Communication-Centric Debug of Embedded Multi-Processor SoCs. In *VLSI Design, Automation and Test, 2009. VLSI-DAT '09. International Symposium on*, pages 183–186, April 2009. 6.4

[108] B. Vermeulen, K. Goossens, and S. Umrani. Debugging Distributed-Shared-Memory Communication at Multiple Granularities in Networks on Chip. In *Networks-on-Chip, 2008. NoCS 2008. Second ACM/IEEE International Symposium on*, pages 3–12, April 2008. 6.4

[109] Bart Vermeulen and Kees Goossens. *Debugging Multi-Core Systems-on-Chip*. CRC Press, Taylor & Francis Group, 2010. 6.4

[110] G. Weisz and J. C. Hoe. C-To-CoRAM: Compiling Perfect Loop Nests to the Portable CoRAM Abstraction. In *FPGA*, 2013. 3.3

[111] Clifford Wolf. Yosys Open SYnthesis Suite. `http://www.clifford.at/yosys/`. 4.3.1

[112] Xilinx. Chipscope Pro and the Serial I/O Toolkit. `http://www.xilinx.com/tools/cspro.htm`, . 6.1

[113] Inc. Xilinx. Vivado High-Level Synthesis. `http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html`, . 3.3

[114] Hyunbean Yi, Sungju Park, and S. Kundu. A Design-for-Debug (DfD) for NoC-Based SoC Debugging via NoC. In *Asian Test Symposium, 2008. ATS '08. 17th*, pages 289–294, Nov 2008. 6.4

[115] Marcela Zuluaga, Andreas Krause, Peter Milder, and Markus Püschel. "Smart" Design Space Sampling to Predict Pareto-optimal Solutions. In *Proceedings of International Con-*

ference on Languages, Compilers, Tools and Theory for Embedded Systems, 2012. 5.5

[116] Marcela Zuluaga, Andreas Krause, Guillaume Sergent, and Markus Püschel. Active Learning for Multi-Objective Optimization. In *International Conference on Machine Learning*, volume 28, 2013. 5.5