University of California
Santa Barbara

Full-System Collaboration in Heterogeneous SoCs with a Hardware Network Stack

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

Master of Science
in
Electrical and Computer Engineering

by

Brian Li

Committee in charge:

Professor Jonathan Balkind, Chair
Professor Timothy Sherwood
Professor Rich Wolski

September 2024

The Thesis of Brian Li is approved.

Professor Timothy Sherwood

Professor Rich Wolski

Professor Jonathan Balkind, Committee Chair

September 2024

Full-System Collaboration in Heterogeneous SoCs with a Hardware Network Stack

Brian Li

Acknowledgements

I would first like to acknowledge Professor Jonathan Balkind for accepting me into his lab as a naive sophomore almost four years ago. Even though I lacked any prior experience, he provided me a space to explore the world of computer architecture, mentoring me both personally and academically being someone I can always trust. I would also like to deeply thank my fellow researchers Katie, Naz, Guillem, Guy, Joan, and the rest of the ArchLab who have all helped me countless times throughout the years with our collective research efforts, from pulling all-nighters together in the lab to helping me better understand their own research. Additionally, I want to express my gratitude for professors Rich Wolski and Timothy Sherwood for taking their time to serve on my thesis committee. Last but not least, I want to sincerely thank my family and friends that supported me and helped me grow as a person throughout my five years at UCSB. This thesis would not have been possible without all of you.

Abstract

Full-System Collaboration in Heterogeneous SoCs with a Hardware Network Stack

by

Brian Li

As we approach some of the physical limits of transistor scale and performance, modern system-on-chips (SoCs) have become increasingly heterogeneous, incorporating more hardware accelerators in their designs. The general design philosophy has been to treat these accelerators simply as an off-loading co-processor, each with their own custom software drivers. This wastes system performance and kernel developer time and inherently prohibits the amount of collaboration between all of the SoC components. Our project, Pengwing, seeks to alter the current standard surrounding accelerator usage to provide powerful collaboration between hardware and software system services. We implement a novel blended operating system design that follows the idea of software-oriented acceleration. In this paradigm, robust software-hardware interactions are achieved by providing a system substrate that enables communication with accelerators using existing software abstractions like shared-memory queues. My contributions to this project revolve around incorporating Beehive, a hardware network stack, into our SoC design. By intentionally decoupling Beehive from the cores during integration, software and hardware alike can utilize various network functions provided through Beehive using our standard software queue API. This thesis will detail the versatility present in our blended OS implementation and showcase robust software-hardware interaction through various setups that leverage Beehive utilizing the same underlying hardware SoC design.

Table of Contents

List of Figures

Chapter 1

Introduction

The deceleration of Dennard scaling and Moore's law prompted the prominence of accelerators in system-on-chip designs, with their ability to achieve efficient specialized computations with regards to power and performance. These SoCs utilize a heterogeneous architecture incorporating various accelerators as off-load based compute resources to minimize the effects of dark silicon [1, 2]. This design philosophy is particularly useful in mobile embedded systems owing to its delicate balance of space and efficiency and is permeating into desktop [3] and server [4, 5] devices as well. Consequently, research into more effective ways of utilizing accelerator-rich platforms will only continue to increase.

These increasingly heterogeneous SoC designs leverage various hardware accelerators ranging from application-class modules like audio/video encoding [6], neural processing engines [7], and cryptographic engines [8], to, more recently, system service components such as memory allocation [9], garbage collection [10], and networking [11, 12, 13, 14]. The improvements in computational efficiency brought about by these hardware accelerators can not be understated, but the current usage of accelerators can be improved upon in two ways: more uniform interfaces and more hardware-software collaboration. Current accelerator designs focus on maximum computational optimization, leaving software to burden any mismatch in hardware-software communication. The software abstractions used to coalesce the differing interfaces leak hardware details, forcing changes to data

format and memory management. This harms generality and programming efficiency, as developers must acutely understand each accelerator's microarchitecture in order to integrate and program them into their designs. Additionally, this often requires the use of custom complex device drivers that handle the semantics for interaction and memory management, which will invariably differ between devices. Ultimately, this limits collaboration across the whole SoC as individual hardware accelerators are tied to the calling software application due to these highly specialized driver stacks and runtimes, making composition of different services impossible.

Pengwing seeks to fundamentally alter this relationship between the hardware accelerators and software running on general-purpose processor cores. We aim to increase collaboration between hardware and software as well as between accelerators under a single OS. The Pengwing project explores the idea that either hardware or software can perform system service functionality interchangeably with the ability to freely invoke each other. In order to do this, Pengwing realizes a common system substrate leveraging shared memory queues that provide a standard interface to both software and hardware [15]. These queues are an implementation of an idea called software-oriented acceleration (SOA) proposed in Cohort [15] wherein programmers can use existing software abstractions to communicate with accelerators, replacing the custom device drivers with more straightforward interfaces. Hardware accelerators can also leverage these queues to communicate with other hardware accelerators, which permits a greater level of collaboration across these accelerator-rich SoCs by chaining various services together, bypassing traditional OS intervention. We refer to this concept design as a blended OS that reshapes the system stack to provide peer-to-peer communication between hardware components and their software counterparts. This design "blends" the conventional responsibilities held by software and hardware in a typical top-down application stack, enabling full-system collaboration in a user-definable manner previously unavailable in traditional operating

system designs through the SOA approach.

One important system service that we wanted to incorporate into Pengwing was networking. In accelerator-rich environments with many distributed systems like servers and databases, the ability to quickly communicate with other devices is essential. While general-purpose cores are able to service networking functionality for these hardware accelerators, bypassing the software implementations by using a hardware network stack can help reduce end-to-end latency and increase throughput. Beehive [16] is a flexible FPGA network stack initially designed for direct-attached accelerators, but I modified the design to seamlessly incorporate its core functionality into our design, enabling hardware-based networking for the SoC as a whole.

My contributions to Pengwing encompass this integration and testing of Beehive into our system. The network stack is introduced as another accelerator in the design, and because of the changes enabled by Pengwing's paradigm, it is able to service invocations from both software and other accelerators. We currently support UDP packet transmission and reception, and Beehive can be configured to support the processing of raw data streams or packet descriptors.

- Chapter 2 explores the background research that enables Pengwing as a project, from the manycore research processor that we leverage to Beehive's network stack design.

- Chapter 3 expands on my process of adapting Beehive from its original usage for direct-attached accelerators to a version that connects to our system substrate and can process network functionality for both software and hardware.

- Chapter 4 showcases four distinct experimental setups with Beehive and other accelerators that highlight the robust hardware-software collaboration enabled by Pengwing, each evaluated on simulated throughput metrics.

- Chapter 5 considers related works pertaining to OS designs and hardware-software co-designs that also address the question of increasing collaboration across heterogeneous SoCs. Additionally, we look into other system services and network stack implementations to determine potential future additions to the Pengwing system.

- Finally, Chapter 6 concludes the thesis and examines potential future improvements for the current Beehive setup and Pengwing as a whole that could aid with deeper analysis of the benefits of a blended OS.

Chapter 2

Background Research

As a multi-faceted project, Pengwing [17] would not be possible without some critical pre-existing research. From the underlying SoC design framework to the individual accelerators, the various components that go into Pengwing enable us to construct the system substrate and blended OS that we envision to facilitate greater hardware-software collaboration.

The project begins with our open-source manycore processor framework, OpenPiton [18], whose robust interconnect structure and cache coherence protocol facilitates incorporating heterogeneous tiles and cores into the design. In order to add accelerators into the design, we leverage the Cohort engine which provides a standardized interface to shared-memory queues for both software and hardware. I then modify Beehive, the hardware network stack, to attach to a Cohort engine-containing OpenPiton tile, enabling the rest of the SoC to interface with the network by treating Beehive as a decoupled accelerator.

2.1 OpenPiton

OpenPiton [18] is a multi-threaded manycore processor with a robust simulation and synthesis framework that enables quick and scalable research. The general architecture

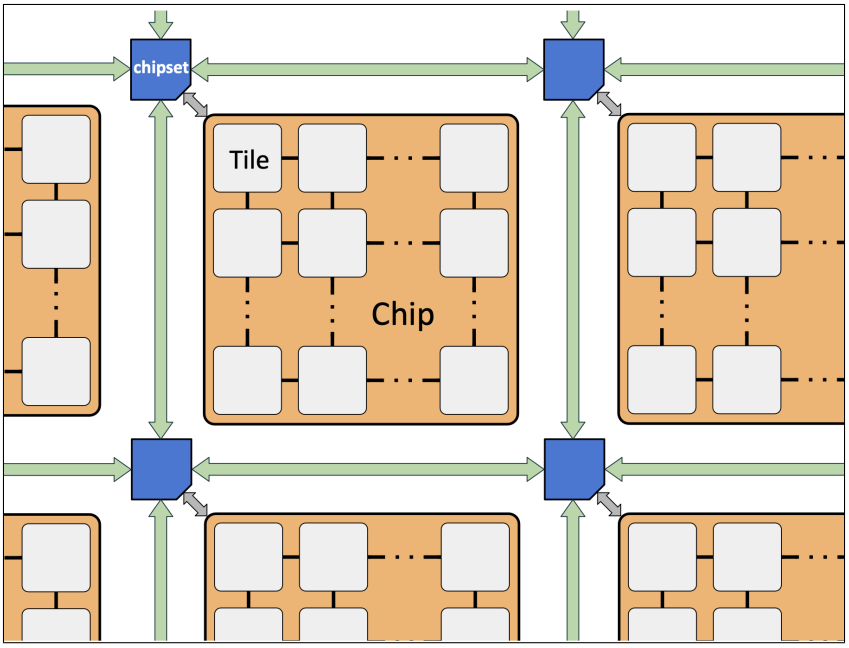

Figure 2.1: OpenPiton Architecture. Each manycore chip consists of a 2D mesh of tiles, connected together via chipset logic and networks (reproduced [18]).

consists of individual manycore chips connected together via chipset logic and networks, visualized in Figure 2.1. At the inter-chip level, various topologies like crossbar, 2D mesh, and 3D mesh have been implemented and can be configured through the chipset network routers. Each manycore chip consists of a configurable 2D mesh of tiles that use 3 sets of network-on-chips (NoCs) to facilitate communication, also shown in Figure 2.1. For our research purposes, we configure our design to use an architecture with one chipset connected to one chip that houses a variable-sized mesh of tiles, focusing on the inter-tile activity rather than the inter-chip behavior.

2.1.1 Tile

The tile is the basic building block of any OpenPiton design. In our configuration shown in Figure 2.2, each general purpose tile consists of a private L1.5 cache, a subset of the shared and distributed L2 cache, three NoC routers, and a modified Ariane (CVA6)

RISC-V core [19, 20, 21].

The L1.5 data cache is a write-back cache that serves to reduce the bandwidth needed by Ariane's L1 write-through cache and to interface with OpenPiton's P-Mesh cache coherence system. Each tile contains a portion of the shared L2 cache that accepts memory requests from the L1.5 through the P-Mesh NoC routers and sends memory requests to off-chip DRAM and cache fill responses back to the L1.5.

The P-Mesh NoC routers facilitate the inter-tile communication required by OpenPiton's two cache levels to maintain cache coherence within the chip. OpenPiton chips use three 64-bit physical NoCs (no virtual channels) designed to ensure a deadlock-free network. Each NoC enables bi-directional communication through two uni-directional links that each operate under credit-based flow control. To prevent deadlocks, the routers leverage dimension-ordered wormhole routing and the NoCs are assigned different priorities, with NoC3 having the highest priority and NoC1 having the lowest priority. This guarantees no cycles in the resource dependency graph, logically ensuring the absence of deadlocks.

2.1.2 Chipset

The chipset houses all the off-chip logic, such as the I/O and DRAM controllers and inter-chip network routers. The chipset pulls in traffic from the chip through a chip bridge connected to the top left tile in the tile array, and directs requests to the correct controller using a chipset crossbar or to other chipsets using the network routers. Because Pengwing uses a one chip+chipset design, the chipset is primarily used for the DRAM controller since we do not have the need for inter-chip routing.

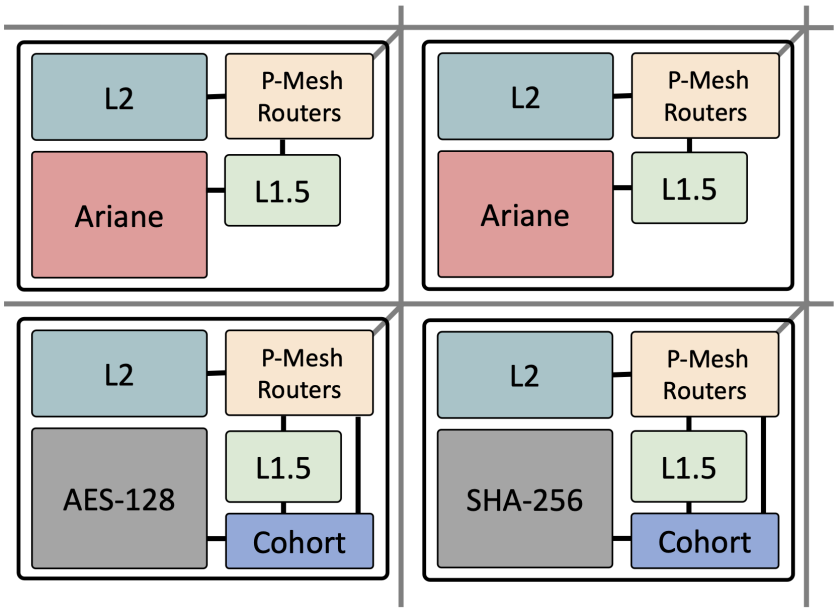

Figure 2.2: Example Cohort Layout. The design consists of two OpenPiton+Ariane [18, 21] tiles and two Cohort tiles integrated with different accelerators (reproduced [15]).

2.1.3 Core

The original OpenPiton architecture uses an OpenSPARC T1 core [22] but further research work [21, 20] provided support for cores with different instruction set architectures (ISAs) such as Ariane [19], an open source RISC-V core now known as CVA6. Ariane is a 64-bit, 6-stage, single-issue, in-order processor implementing the RV64GC instruction set and extensions as well as the M (machine), S (supervisor), and U (user) privilege levels required for booting Linux. Ariane's support for these the privilege modes enables us to quickly test Pengwing functionalities in bare metal or Linux as well as develop our full blended OS prototype. In order to integrate Ariane into OpenPiton, the P-Mesh was adapted to support RISC-V atomic operations and Ariane's cache system was modified to connect to the P-Mesh NoCs, but otherwise the two systems' core designs were left unaltered. With OpenPiton+Ariane, we have the foundational hardware on which we can iterate upon to prototype our blended OS design.

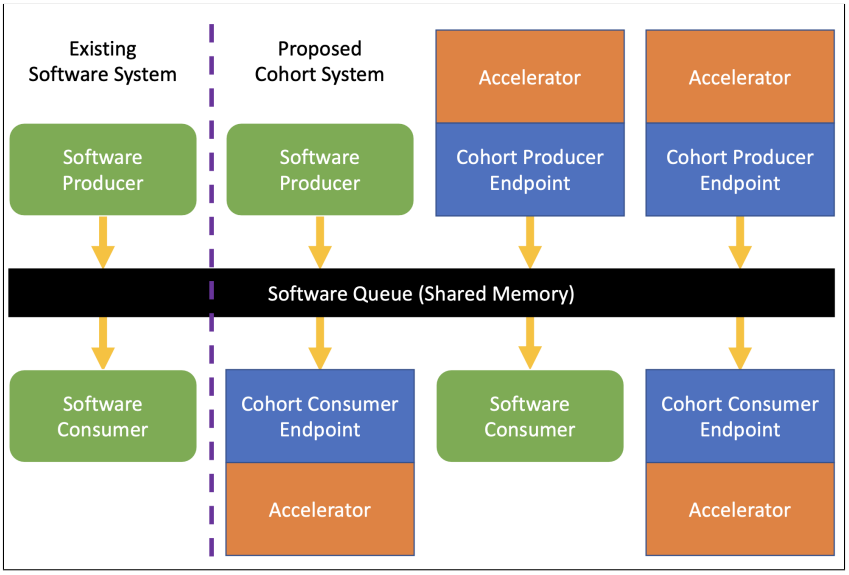

Figure 2.3: Proposed Cohort system design enabling hardware-software communication through shared memory software queues (reproduced [15]).

2.2 Cohort

While OpenPiton+Ariane provides the manycore framework, it does not offer explicit support for accelerators in the design. Cohort [15] serves to solve this issue by standardizing a queue-based interface accessible by both hardware and software. This work introduces the Software-Oriented Acceleration (SOA) approach that emphasizes the use of existing software paradigms to facilitate accelerator communication, implemented into an OpenPiton+Ariane setup. Cohort supports single-producer single-consumer (SPSC) queues as they are widely used in many existing high performance libraries. The end goal is to provide one system for hardware-to-software, software-to-hardware, and hardware-to-hardware communication through an abstraction such that different hardware accelerators can be more efficiently composed into one system as shown in Figure 2.3.

On the hardware interface side, Cohort implements an engine that attaches to the accelerator, the L1.5, and the NoC routers, as shown in the bottom two tiles of Figure 2.2. The Cohort engine hides the cache coherence system of OpenPiton from the accelerator

replacing the Ariane core to serve as the compute unit of the tile. The engine provides a pair of latency-insensitive valid-ready interfaces as the only connections between the accelerator and the rest of the system; there is the consumer endpoint used as the source of inputs into the accelerator, and the producer endpoint is used as a sink for the accelerator's outputs. Once these two endpoints are registered with individual SPSC queues, they can perform the necessary coherence operations to track the read and write pointers to properly push to and pop from the queues described in the next section.

In terms of software, Cohort provides a user-mode API to register and interface with the SPSC queues. Cohort queues are implemented as a ring buffer with a configurable descriptor struct that supports read and write pointers, base address, element size, and length. Programmers can initialize queues using their existing queue libraries with these configurable descriptors, register the queues using the Cohort API to a specific Cohort engine using a tile ID, and push to or pull from the queue based on if it was registered to a consumer endpoint or producer endpoint as described previously. The queues are not limited to purely software-hardware interaction either, as programmers can also register the queue endpoints between two Cohort engines, chaining two or more hardware accelerators together as shown in Figure 2.3. Because the queues are initialized and configured at runtime, they can be dynamically re-allocated to different accelerator tiles on the fly, a very powerful feature useful for adaptive workloads.

2.3 Beehive

Networking functions are one of the system services we were most interested in when developing Pengwing. Reducing network overhead on the general-purpose cores will improve overall system efficiency, and with a flexible hardware network stack we can both adapt the stack for different networking situations as well as enable other components of

the SoC to directly interface with the network using Cohort.

Beehive [16] was originally designed as a modular NoC-based FPGA network stack for direct-attached accelerators, enabling application accelerators to interface with a network independent of CPU. The design consisted of an array of different hardware tiles that each perform a specific networking function such as UDP TX/RX connected by a NoC. Figure 2.4 highlights the full ingress and egress path of an Ethernet frame through the design. The annotations in the figure indicate the full ingress path along with the intermediate data structures, starting with step 1 where the frame enters Beehive through the ETH RX tile. This Ethernet frame gets processed into header, metadata, and data flits and routed to the IP RX tile through the NoC in step 2. Label 3 shows the en route processed flits, specifically how the data flit lacks an Ethernet header due to the ETH RX tile consuming this information to determine the routing to the IP RX tile. Step 4 denotes the IP RX tile routing the information in label 5 to the UDP RX tile. The final step in the ingress path involves routing the flits in label 7 with the final de-encapsulated UDP data payload to the application tile for processing. The design shown houses a UDP echo application, where it re-encapsulates the incoming data payload into a UDP packet and directs it through the egress path where each of the TX tiles will encapsulate the data with the appropriate headers until the final Ethernet frame is transmitted onto the network.

While our Pengwing design does not directly use the sample UDP echo application shown in Figure 4, the modularity provided by Beehive by separating different functions into discrete engines is immensely useful. Not only does this increase performance because the receive and transmit paths are decoupled, it provides a clear framework for a new modular design to be integrated into Pengwing, rivaling the usual complexities when working with software network driver stacks and enabling network access for previously isolated accelerators in the SoC.

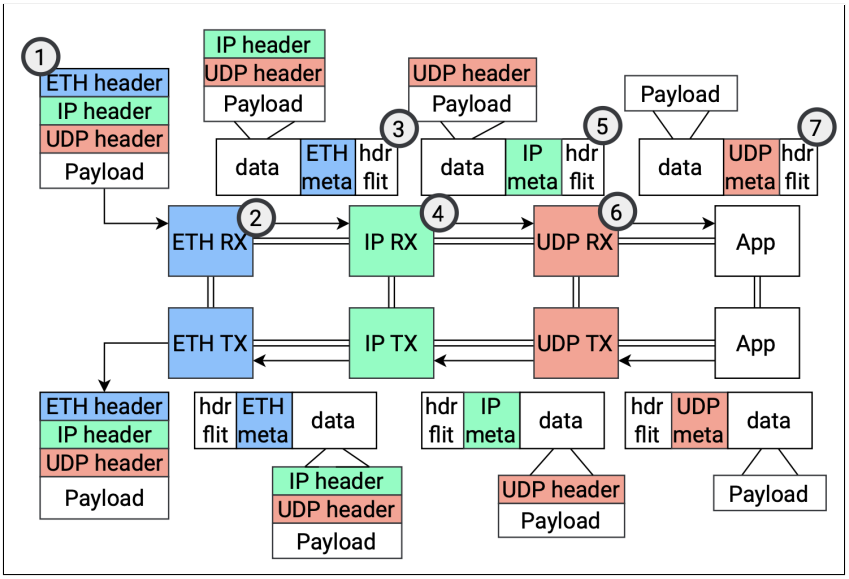

Figure 2.4: Sample Beehive UDP Echo Design. This diagram outlines a UDP datagram's path through the Beehive topology with a demo application implemented to echo the payload back out as another UDP datagram (reproduced [16]).

2.4 Pengwing

In order to test the blended OS aspect of Pengwing, we must also construct an appropriate accelerator-rich hardware platform. While Cohort provides the initial support to attach accelerators decoupled from the processor, Pengwing develops on this idea by designing a powerful system substrate capable of supporting more features than base Cohort. This substrate retains the shared memory SPSC queue semantics and memory management from Cohort and adds support for multiple-producer multiple-consumer (MPMC) queues, a load store unit (LSU) with MAPLE [23], a key-value store (KVS) unit with METAL [24], a hardware memory allocator with Falafel [25], and an in-house hardware garbage collector, with the flexibility to integrate other Cohort-enabled accelerators as well. Certain functionalities like the MAPLE LSU and MPMC queue support are integrated directly into the Cohort engine found in each tile, whereas some components like the KVS unit and the memory allocator are introduced to the system as decoupled accel-

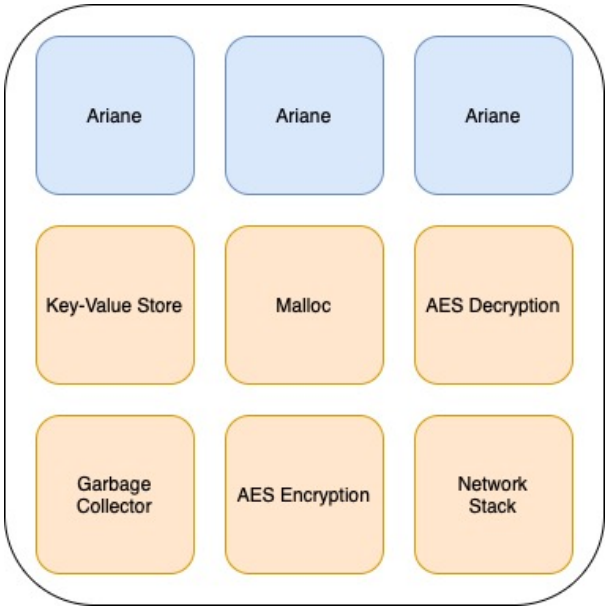

Figure 2.5: Combined Pengwing SoC showing the tile layout with all the accelerators in the design.

erators. As part of my research, I designed and implemented our final 3x3 SoC design for Pengwing consisting of three Ariane tiles and six Cohort tiles with various accelerators attached as shown in Figure 2.5.

2.4.1 MAPLE

A key component of the substrate that Beehive interacts with is the Memory Access Parallel-Load Engine (MAPLE) [23] that reduces latency of irregular memory accesses with prefetching. MAPLE does not require any modifications to the ISA, core, or memory hierarchy of the SoC and provides a queue interface for memory requests. While MAPLE supports various operations, we are only concerned with its store capability (to store data to a pointer address) and its load capability (to load data from a pointer address) for Beehive's integration into the SoC. Effectively, MAPLE serves as a robust LSU for Beehive to access memory that the standard Cohort engine would not normally be able to access.

2.4.2 Falafel

As shown in Figure 2.5, our SoC design also contains a memory allocation accelerator, Falafel [25]. There are many high-performance allocators [26, 27] written in software, and research efforts into hardware implementations [9, 28, 29, 30] show good promise with respect to performance. However, these allocators typically use a software-hardware hybrid design [29, 28], attach into systems using a shared bus [30], or are tightly integrated into the core [9], all of which clash with Pengwing's NoC-based decoupling philosophy for system services. In an effort to bridge this gap, a researcher from our lab designed Falafel [25], an in-house hardware memory allocator accessible by both software and hardware through Cohort to facilitate the memory demands of various components in the processor. Falafel implements the first-fit allocation algorithm based on a singly linked list. By being developed with Pengwing in mind, this allocator uses separate queue interfaces for allocating and freeing memory, which aligns well with the Cohort queues utilized heavily in our system. This enables either software or hardware to request and free memory independently of each other; specifically, Beehive can directly request memory for inbound payloads and user applications can free that memory within software whenever necessary as described later in subsection 3.3.2. This example highlights the essence of Pengwing, that software and hardware can co-exist within a system as peers rather than under a strict privilege hierarchy.

Chapter 3

Integrating Beehive into Pengwing

Beehive's modular design with separate tiles for each network layer presents a flexible framework for adoption into Pengwing, even though its intended usage is for direct-attached accelerators and not full SoCs. As part of my research, I extended Beehive's functionality to properly interact with Cohort queue semantics, inherently enabling any hardware accelerator or software service in Pengwing to use the hardware network stack. I built initial prototypes for integrating Beehive into the SoC to enable receiving and transmitting UDP packets and added configurable support for handling either raw data streams or packet descriptors allocated through Falafel as enabled by Pengwing.

3.1 Initial Exploration

To begin my research efforts, I first had to understand Beehive's framework from design to testing. Beehive leverages cocotb [31], a Python-based testbench environment, FuseSoC [32], a package manager and build system for hardware description languages, and Siemen's QuestaSim for its simulation framework. Using its standard tools, I was able to replicate Beehive's sample User Datagram Protocol (UDP) echo application described in Section 2.3 within simulation. We chose UDP for our first integration efforts because of its connectionless nature, but this work is the basis on which we can enable

other Beehive-supported connection-based protocols like Transmission Control Protocol (TCP) within Pengwing. It is important to note that my research was fully conducted in RTL behavioral simulation due to current limitations which will be discussed later in the chapter.

3.1.1 Initial Hiccups

As I progressed in my understanding of Beehive's design with regards to general architecture and codebase structure, I inevitably ran into two major issues: simulation framework differences and module name aliasing. Beehive dynamically generates certain files using FuseSoC at build time, which is not easily integrated into OpenPiton's own simulation scripts. My workaround was to iterate on the generated files and then add them into OpenPiton's filelists statically to maintain a consistent source file structure, bypassing Beehive's entire simulation structure to completely use OpenPiton's framework. This is highly beneficial as it means future contributors to OpenPiton+Beehive will only need to rely on one set of software dependencies.

The module name aliasing was a trickier issue in terms of reconciling two different frameworks. Beehive's NoC routers are modernized and specialized versions of OpenPiton's routers, sharing similar module and macro names. This resulted in odd simulation behavior during initial integration attempts as certain modules and macros would be instantiated and defined incorrectly across the two codebases, which was hard to debug due to the lack of errors in the compilation logs. The simulator would instantiate the general OpenPiton version of certain modules with the wrong NoC sizes within Beehive, preventing data from being routed and processed properly within the Beehive topology. After careful examination of the waveforms to diagnose this issue, I patched this bug so that there were no overlaps in any module or macro naming between the two different systems and was able to debug the hardware I developed.

3.1.2 Data Structures within Beehive

To understand how to convert Cohort queue semantics into UDP packets, I first had to understand the internal representation of data within the hardware network stack. Beehive splits UDP packets into three types of flow control units (flits) to be transmitted on the NoC between the UDP TX/RX tiles and the UDP echo application tile.

The header flit contains information regarding the routing of the full packet within Beehive's NoC. Fields within this flit include source and destination tiles in Beehive's topology, the message type which is either UDP_{TX/RX}_SEGMENT, and the message length in terms of number of non-header flits.

There are two different types of metadata flits, one for receiving and one for transmitting, but they both contain the same fields. This flit includes information regarding source and destination IP addresses and ports, as well as the data length (not to be confused with the header's message length) in terms of number of bytes in the following data flit(s). This value is crucial for masking the correct subset of the data flit in the case that the data size is not perfectly aligned to Beehive's NoC width.

The data flit simply contains the data payload the caller of the network stack wishes to put in the UDP datagram. If the amount of data is greater than the Beehive NoC width, the data should be split into consecutive flits that align with the width.

3.1.3 File Structure of Beehive

In order to understand the process of integrating Beehive into OpenPiton using Cohort, I believe it is important to explain the module hierarchy as shown in Figure 3.1. There are 7 modules of particular note to my research.

- **udp_echo_top**: This module is the top level wrapper for the whole Beehive system, instantiating the grid layout of Beehive and exposing media access control (MAC)

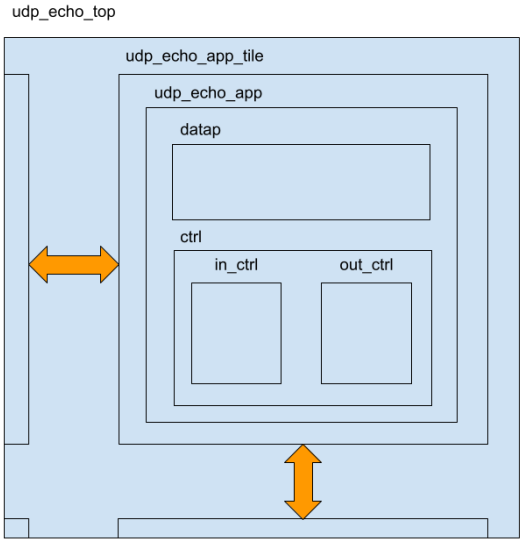

Figure 3.1: Original UDP Echo Module Hierarchy.

level ports that can tie into third-party network interface cards (NICs). All the tiles of the different network layers (Ethernet, IP, UDP) are not shown in the diagram but are instantiated at this level.

- **udp_echo_app_tile**: This is the tile within **udp_echo_top** that we are most interested in. It houses all the necessary hardware to perform NoC routing and UDP packet echoing.

- **udp_echo_app**: This module contains the logic for the actual UDP echo application, split into two internal modules.

  - **datap**: Within **udp_echo_app**, this module houses the datapath logic, from holding the various inbound flits in internal buffers to generating the parameters in the outbound flits.

  - **ctrl**: This is the control module that processes when to accept inbound flits from the UDP RX tile and when to transmit flits to the UDP TX tile. This

logic is actually contained in two different internal modules, one for inbound data (**in_ctrl**) and one for outbound data (**out_ctrl**).

3.2 Adapting UDP Echo App

As mentioned before, Beehive provides a sample UDP echo test with a design layout shown in Figure 2.4 in Section 2.3. If we examine the datapath, we can see that the UDP packet flows from Ethernet Media Access Control (MAC) → ETH RX → IP RX → UDP RX → udp_echo_app_tile → UDP TX → IP TX → ETH TX → Ethernet MAC, with the frame being de-encapsulated along the RX path and re-encapsulated along the TX path. In this arrangement, Beehive's design only has one path for data to flow. Because I am trying to link this hardware network stack to other system services in the SoC, it must be able to handle separate ingress/inbound and egress/outbound data flows independently.

To decouple these two paths for usability in Pengwing, I designed a new module within udp_echo_app called cohort_to_beehive that could interact with the valid-ready Cohort endpoints (see Figure 3.2). This also required modifications to the existing control and datapath architecture, as outbound packets now needed to be driven by the incoming Cohort queue data and inbound packets needed to be fed into the outgoing Cohort queue. My new module contains two finite state machines (FSMs), one for outbound data and one for inbound data.

The outbound data FSM did three steps. It was hardcoded to accept 4 Cohort elements at a time from the consumer endpoint and store them in an internal 32 byte buffer to match the size of Beehive's NoC width. It would then raise the correct signals to notify the control module to generate hard-coded header and metadata flits through the datapath module. Finally, it would send the 32 bytes within its internal buffer to the datapath in order to form the data flit that would be passed on to the UDP TX tile.

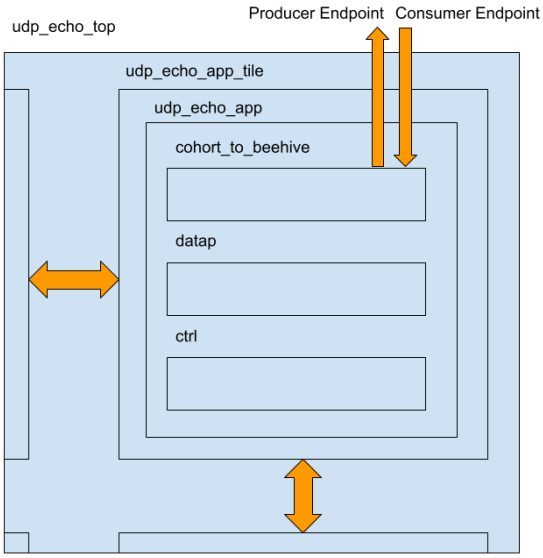

Figure 3.2: Modified UDP Echo Module Hierarchy.

The inbound data FSM essentially reversed this process. The control module would flag this FSM when it received the header and metadata flits from the UDP RX tile through Beehive's NoC. When the data flit arrived, it would be routed into the FSM and pulled into its internal buffer. Then, it would increment through the buffer 8 bytes at a time (the size of a Cohort queue element) and push that segment into the producer endpoint until it sent out all of the data.

This iteration of cohort_to_beehive was very naive and served only as a prototype design as certain parameters like message length were hardcoded. However, it was a proof-of-concept that Beehive could be used within Pengwing. I successfully simulated a test application on a 2x2 OpenPiton+Cohort design with same layout as seen in Figure 2.2 with this modified Beehive integrated into one of the Cohort tiles. In this test, a pair of Cohort queues were registered between software and the Cohort producer and consumer endpoints attached to Beehive, and for simulation purposes, Beehive's outward facing MAC layer is directly looped back to itself instead of tapping into an actual network.

This way, Beehive accepts data through the consumer endpoint, sends out a UDP packet through the egress path which gets looped back into its ingress path, and then Beehive pushes the data payload into the producer endpoint to be consumed by software.

3.3 Improvements to the Design

After verifying the usability of the hardware network stack within the SoC with the test case just described, I iterated on the design to support other features. The main issue with the prototype solution was that message lengths were hardcoded to be 32 bytes, so I expanded the functionality to support variable-sized inputs. Additionally, I modified cohort_to_beehive to be more generic in order to enable the processing of both raw data streams and packet descriptors within Beehive in a configurable manner.

3.3.1 Variable Width Inputs

My solution to processing variable sized inputs for the outbound path was to consider two situations: sizes up to 32 bytes and sizes greater than 32 bytes. The first modification was to add a size value into the agreed protocol between the user and the hardware network stack itself. For example, if a software service wanted to set the payload size of UDP datagram at 24 bytes, it would first push the byte size value (24) into the queue registered to Beehive's consumer endpoint followed by the actual data stream separated into 8 byte chunks to match the Cohort queue element size. If the size value is not aligned to 8 bytes, the final queue element should be zero padded to 8 bytes. The FSM handling outbound data then sets the correct values in the header and metadata flits to reflect the correct size of the UDP packet and sends the data flit out.

To account for sizes greater than 32 bytes, I further modify this FSM to keep track of the number of 32 byte data flits needed to contain the requested payload. The FSM still

Figure 3.3: A sample outbound UDP datagram request showing how software sends the requested payload size followed by the data stream into a Cohort FIFO queue.

pulls in 8 bytes at a time from the consumer endpoint, but it also tracks which quartile of the current data flit this piece of data corresponds to. Once the FSM has pulled in 32 bytes into its internal buffer, it will then send that data flit out onto Beehive's NoC and proceed to pulling in data for the next data flit. Once the FSM gets to the last flit, it functions exactly as described before for sizes up to 32 bytes. Figure 3.3 shows how software would send data to Beehive using a Cohort queue and highlights how internally Beehive would discern between the multiple data flits.

As for the inbound path, I made similar modifications to the prototype design. The FSM first reads the size from the metadata flit and sets the correct values for the number of data flits to process. It then pulls in a 32 byte data flit from the Beehive NoC and iterates through 8 bytes at a time sending each chunk out to the producer endpoint. Once the full flit has been sent out through Cohort, the FSM repeats this cycle by pulling in the next data flit for processing, continuing until all the data from the UDP payload has been processed.

3.3.2 Packet Descriptors

The second improvement I implemented was enabling users of the network stack to send and receive packet descriptors rather than raw data streams. This is useful for reducing the amount of data transmitted through Cohort queues in the case of much larger data frames to prevent the queues from filling up too quickly.

For the outbound path, I designed a new FSM that performs the same initialization

of header and metadata flits using a size value passed through Cohort but it only accepts a packet descriptor as the next Cohort queue item rather than raw data. Currently, the protocol treats the packet descriptor as a pointer to an array of `uint64_t` data values, but support for more robust descriptors can eventually be added. The outbound FSM then requests 8 bytes at a time from the memory location provided by the descriptor using MAPLE, the LSU of Pengwing's system substrate. Once it receives the memory response from MAPLE, the FSM performs the same operations as before to batch the data then send out the full data flit to the UDP TX tile through Beehive's NoC. Essentially, the core logic of the FSM remains the same with certain modifications to generate the data flits using MAPLE rather than Cohort queues.

Modifying the inbound path to support packet descriptors required more consideration due to the need for memory allocation. The new design grabs the data size value from the metadata flit normally. Before anything else occurs, though, it requests memory of that size from Falafel, our hardware memory allocator, using another set of Cohort queues initialized between Beehive and the allocator. To be clear, the versatility of Pengwing enables us to call software `malloc()` through the same queue-based API if so desired. Rather than servicing the allocation through a software library though, this chaining of accelerators enabled by Cohort showcases how Pengwing enables system services to interact independently of the processor cores. Once the inbound FSM accepts the pointer to the memory allocated by Falafel, it then processes the data flit from the NoC. The module then uses MAPLE requests again to store 8 bytes of the data flit at a time, cycling through as before until all data is stored. The size and pointer are then pushed into the producer endpoint queue for use by the caller of the network stack.

3.4 Improved UDP Design

The central theme of Pengwing is to enable collaboration across different levels, which is highlighted by the hardware-to-hardware accelerator chaining of Beehive and Falafel as well as the varying levels of hardware-software interactions by providing the ability to choose how much data processing is performed on the hardware network stack. Designers are able to define the interactions across the SoC rather than being limited to predetermined conventions. With the additional support for variable sized packets and descriptors, I refined the integration of Beehive into Pengwing to be more adaptable between different scenarios. By defining new SystemVerilog parameters and using generate blocks within cohort_to_beehive, the user is able to configure between processing data streams or pointers for each of the inbound and outbound paths, enabling other collaborators to more easily design and integrate modules for new processing methods as well. The available modules to be generated are explained below.

3.4.1 in_data

This module performs the necessary logic to process inbound UDP datagrams and push the data to a Cohort queue interface as a data stream.

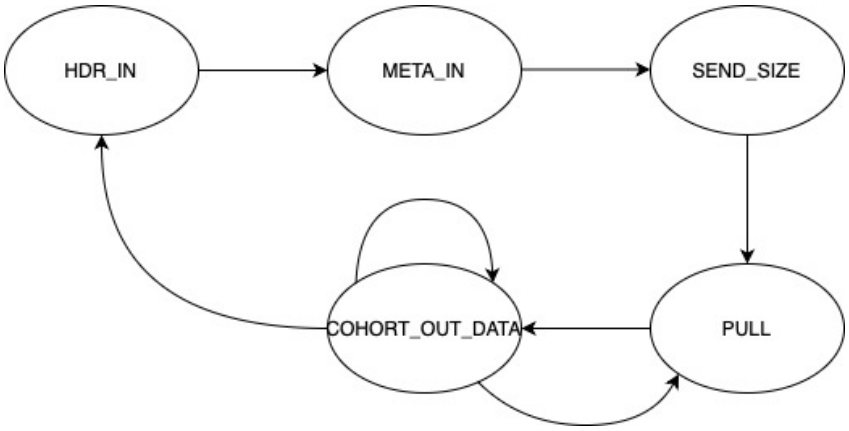

Figure 3.4: in_data State Machine.

Figure 3.4 shows the states and transitions of this FSM with a brief description of each state below.

- HDR_IN: waits for the control architecture to check if a header flit came through the NoC.
- META_IN: waits for the control architecture to check for the metadata flit.
- SEND_SIZE: sets the necessary internal counters to process the payload based on the size from the meta flit and sends the size value to the producer endpoint.
- PULL: pulls in a single data flit from the NoC into its internal buffer.
- COHORT_OUT_DATA: increments through the buffer 8 bytes at a time, sending it to the producer endpoint. If there is still data in the buffer to process, it increments its internal counters and stays in the same state. If the buffer is fully processed but there are still data flits left in the payload, it transitions to PULL to grab the next data flit. If the payload has been fully processed, it will transition back to HDR_IN to wait for the next UDP datagram.

3.4.2 in_pointer

This module performs the necessary logic to process inbound UDP datagrams, allocate and store the data in memory, and push the packet descriptor to a Cohort queue interface.

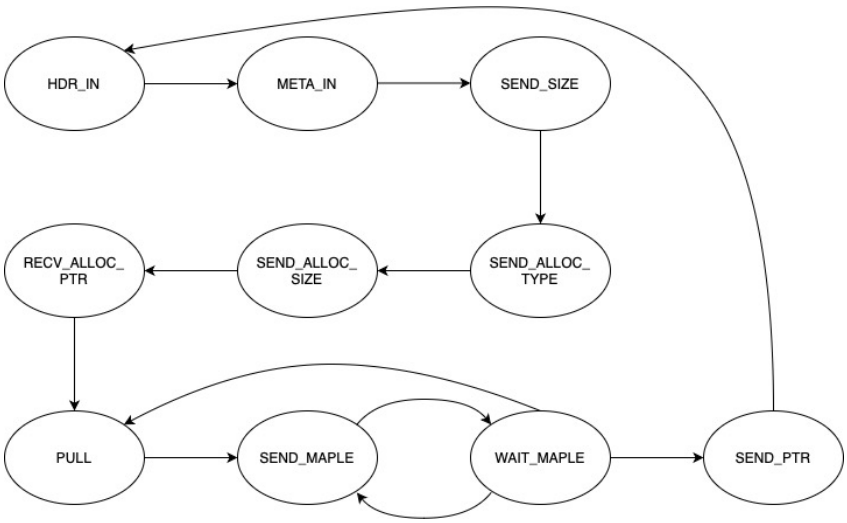

Figure 3.5: in_pointer State Machine.

Figure 3.5 shows the states and transitions of this FSM with a brief description of each state below.

- HDR_IN: waits for the control architecture to check if a header flit came through the NoC.
- META_IN: waits for the control architecture to check for the metadata flit.
- SEND_SIZE: sets the necessary internal counters to process the payload based on the size from the meta flit and sends the size value to the producer endpoint.
- SEND_ALLOC_TYPE: sends the request type to Falafel using a second producer endpoint.
- SEND_ALLOC_SIZE: sends the request size (of the UDP payload) to Falafel.

- RECV_ALLOC_PTR: receives the requested pointer from Falafel through a dedicated consumer endpoint.
- PULL: pulls in a single data flit from the NoC into its internal buffer.
- SEND_MAPLE: sends the memory location and 8 byte data chunk we want to store to MAPLE through its dedicated queue interface.
- WAIT_MAPLE: increments the internal counters to move to the correct location within the internal buffer. If there is still data left to process in the buffer, it will transition to SEND_MAPLE to send the next data chunk to MAPLE. If the buffer is fully processed but there are still data flits left in the payload, it will transition to PULL to pull in the next data flit. If the payload is fully processed, it will transition to SEND_PTR.
- SEND_PTR: sends the packet descriptor (pointer) out to the main producer endpoint.

3.4.3 out_data

This module performs the necessary logic to process data from a Cohort queue interface as a data stream and generate the correct Beehive flits to send to the UDP TX tile.

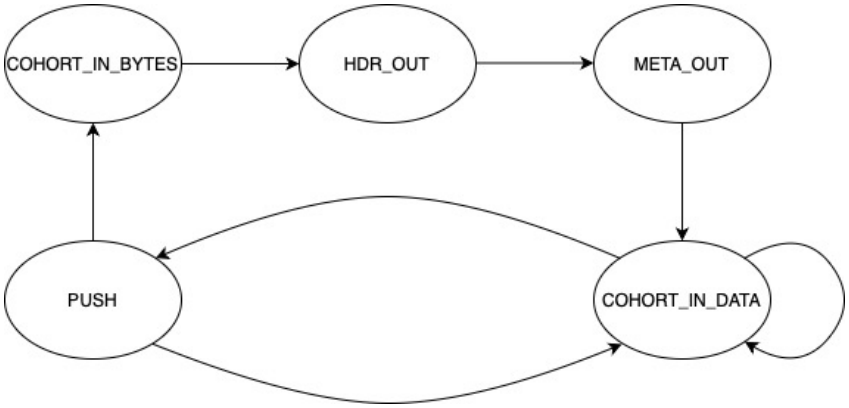

Figure 3.6: out_data State Machine.

Figure 3.6 shows the states and transitions of this FSM with a brief description of each state below.

- COHORT_IN_BYTES: pulls in the requested payload size from the consumer endpoint.
- HDR_OUT: signals the control and datapath architecture to generate the header flit to send out on the NoC.
- META_OUT: signals the control and datapath architecture to generate the metadata flit containing the payload size information to send out on the NoC.
- COHORT_IN_DATA: pulls in 8 bytes of data from the consumer endpoint into its internal buffer. If the buffer is not full, it will increment its interal counters and stay in this state. If the buffer is full, it will transition to PUSH.
- PUSH: pushes the full buffer onto the NoC as a full data flit. If the full payload has been sent out, it will transition to COHORT_IN_BYTES. Otherwise, it will transition back to COHORT_IN_DATA.

3.4.4 out_pointer

This module performs the necessary logic to grab a packet descriptor from a Cohort queue interface, load the data from memory, and generate the correct Beehive flits to send to the UDP TX tile.

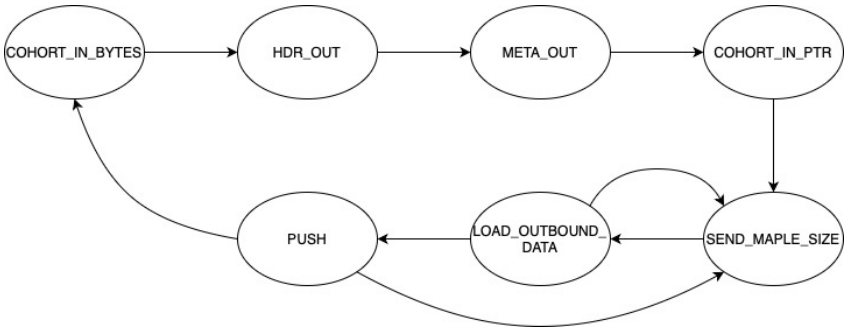

Figure 3.7: out_pointer State Machine.

Figure 3.7 shows the states and transitions of this FSM with a brief description of each state below.

- COHORT_IN_BYTES: pulls in the requested payload size from the consumer endpoint.
- HDR_OUT: signals the control and datapath architecture to generate the header flit to send out on the NoC.
- META_OUT: signals the control and datapath architecture to generate the metadata flit containing the payload size information to send out on the NoC.
- COHORT_IN_PTR: pulls in the packet descriptor from the consumer endpoint.
- SEND_MAPLE_SIZE: requests 8 bytes of data at a time from the memory location given by the packet descriptor using MAPLE's dedicated queue interface.

- LOAD_OUTBOUND_DATA: receives MAPLE's response and loads the data into its internal buffer. If the buffer is not full, it will transition back to SEND_MAPLE_SIZE after incrementing the pointer address. If the buffer is full, it will transition to PUSH.

- PUSH: pushes the full buffer onto the NoC as a single data flit. If the full payload has been sent out, it will transition to COHORT_IN_BYTES. Otherwise, it will transition back to SEND_MAPLE_SIZE to generate the next data flit.

3.5 Full Pengwing SoC

After I was able to test the functionality of OpenPiton+Beehive with the added features, I also designed the full SoC on which we would evaluate Pengwing as a whole. The layout for Cohort-enabled OpenPiton processors outlined in the Cohort paper followed a 2x2 grid design, with two Ariane tiles and two Cohort tiles containing various accelerators. We use this layout to test individual accelerators like our key-value store unit or our memory allocator. However, this layout is not big enough to contain everything in the Pengwing substrate.

To incorporate everything, I used a 3x3 topology for our manycore SoC, with 3 Ariane tiles and 6 Cohort tiles as shown in Figure 2.5 back in Section 2.4. While OpenPiton and Cohort are both rather configurable, this process still involved some hardware design adjustments as well as software library modifications as certain parameters relied on false assumptions about the layout. The philosophy of the ordering of accelerators within this design was to minimize NoC traffic for the evaluation setups to be described in the next chapter. Our hardware network stack was also intentionally placed at an outer edge to account for layout as both designs use a 2D mesh topology, similar to how in OpenPiton the top left Ariane tile connects to the chipset. This combined SoC will be the

default evaluation platform for Pengwing, enabling our collaborators to simulate various applications involving any subset of accelerators on a single hardware design.

3.6 Future Improvements

While I was able to improve the utility of Beehive within the platform from the initial prototype, there are still many advancements that can be made on this design. In terms of certain configurations, we can increase the NoC widths for both OpenPiton and Beehive. For OpenPiton, this can minimize the amount of NoC messages required for Cohort queue operations; for Beehive, this will reduce the number of flits needed to be transmitted as well. Another major upgrade would be to expand the size of Cohort queue elements to require less pushes to Beehive for the raw data stream. In theory, this should be easily done due to the configurable parameters within the platform, but in practice, there is much left to debug in order for the expanded sizes to work. Also, Beehive only requests 8 bytes at a time from MAPLE, but with more tinkering batching these requests to 64 bytes should be possible, which will increase memory throughput and reduce end-to-end latency.

Another architectural change would be to expand the packet descriptor to incorporate more information. The current design only contains the data pointer in the descriptor, but ideally it should contain all the other metadata information like IP address and ports that are still hardwired. I would also enable Beehive to use multiple-producer-multiple-consumer (MPMC) queues, allowing multiple threads or hardware services to call on the network stack through one interface.

Within Beehive, I would explore more into the software configurability between processing data streams and packet descriptors. Currently, I can select which module to generate for inbound and outbound data, but this parameter can only be set at build time. I

believe that changing this to be configurable by software would be much more flexible, as programmers can choose which processing method to use dynamically based on system needs. However, this would require a significant rework to the design of cohort_to_beehive by implementing an LSU within the module to handle storing this configuration within a register file and routing the data to the correct processing module. A potential drawback to this method is that both versions of the inbound and outbound modules would be instantiated resulting in half of the logic being left dormant. A workaround would be to design just one FSM for each path with more intricate logic to differentiate the processing method based on the software configured registers.

Additionally, Beehive contains the necessary hardware tiles to fully support other protocols like IP, TCP, and remote procedure call (RPC), so integrating support for them in the Pengwing version of the hardware network stack should also be a priority. While connectionless protocols like UDP and IP are fairly straightfoward to handle, connection-based protocols like TCP are more difficult to integrate into Pengwing due to the need for additional logic to handle establishing, maintaining, and releasing the connection properly.

Lastly, we are currently debugging issues with an AXI bridge in OpenPiton to be able to test the system on FPGA rather than just in simulation. This would enable us to further verify our full Pengwing design by being able to boot Linux as well as fully test the network hardware stack by leveraging the MAC layer of an FPGA-based network interface card (NIC) like Corundum [33]. With an FPGA implementation, we could transmit the UDP packets over a physical network to another machine to validate the integration of Beehive into Pengwing.

Chapter 4

Experimental Setups and Analysis

After designing the full Pengwing SoC, I evaluated the setup across different use cases that would utilize a hardware network stack. It is important to highlight that all of these experimental setups leverage the same hardware design just with differing software applications, showcasing Pengwing's flexibility with regards to dynamically blending different system service components together.

4.1 Original Cohort+Beehive

Testing the original prototype integration of Beehive using Cohort served as a baseline for the more complex Pengwing setups. Although the hardware design slightly differs with regards to tile layout in OpenPiton (refer to Figure 2.2 for the original tile layout compared to Figure 2.5 for the Pengwing tile layout), the general structure of the test application remains the same.

Each application begins with initializing a pair of FIFO queues using existing software libraries and then registering them using the Cohort API, which requires designating which queue is acting as an input to the accelerator and binding to its Cohort engine's consumer endpoint and which queue is acting as an output for the accelerator and binding to the producer endpoint. The original intent of the Cohort API is to be accelerator-

oriented, focusing on registering one pair of queues to a single accelerator's Cohort engine and designating software as the other endpoint for the queues; however, with clever manipulation of the API, it is possible to chain the queues between two accelerators. Ideally though, we would have native support for chaining between accelerators in Pengwing by defining a graph of the SoC components and its communication layout rather than this temporary solution. It is also important to note that the Cohort engine can be configured to open multiple pairs of consumer/producer endpoints, enabling multiple sets of queues to be registered to a single Cohort engine.

Once the Cohort queues are properly configured and enabled, programmers can then push to or pull from these queues. In the standard Cohort+Beehive setup, the test application generates 4 data elements, each 8 bytes in size, and push them sequentially into the queue connected to the consumer endpoint of the Cohort tile containing Beehive. After a push sync which updates the write pointer of the queue, the application then pulls 4 elements from the queue attached to the producer endpoint of Beehive, confirming that the data matches what was pushed into the network stack. Finally, the software performs a pull sync to update the read pointer of the queue. This is the general framework for applications that utilize Cohort-connected accelerators, and the following sections will detail more involved setups.

4.2 Pengwing Setups

With the robust system substrate, I developed experimental setups that showcase various capabilities enabled by Pengwing. The following setups include hardware collaboration with our AES encryption unit, our key-value store unit, and our memory allocator unit. I also evaluated the difference between the two Beehive processing options of raw data streams or packet descriptors.

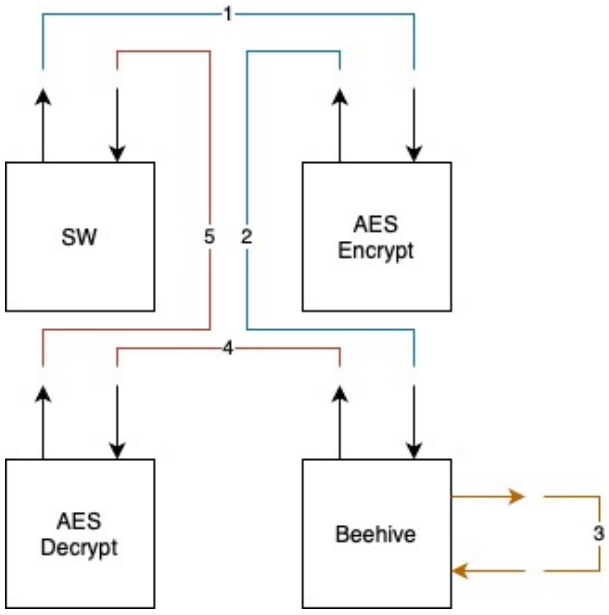

Figure 4.1: Beehive+AES setup showing the logical flow of data.

4.2.1 Beehive+AES

The Beehive+AES setup simulates a situation where a user wants to automatically encrypt data to be sent out on UDP packets as well as decrypt encrypted data coming in from the network. The logical flow is shown in Figure 4.1.

The diagram does not represent the actual layout within Pengwing, but rather the logical flow of data throughout the system. The inbound arrows represent the consumer endpoint of that tile, and the outbound arrows represent the producer endpoint of that tile. Queues are represented by the lines connecting two endpoints, and queues with the same color are part of the same queue pair explained in Section 4.1. As evident in this diagram, Pengwing setups do not necessarily need to adhere to traditional vertical software stacks that tightly bind the accelerator to the core. Pengwing enables dynamic configuration of the system to facilitate services in different ways based on the needs of the programmer.

In the Beehive+AES setup, the application generates and pushes data into the first

queue configured between software and the first AES accelerator. Once the unit properly encrypts the data, it is then piped into the second queue between its producer endpoint and Beehive's consumer endpoint. In this experiment, Beehive is configured to process raw data streams for both ingress and egress packets. Beehive's second pair of endpoints is not for Cohort, however; it represents the interfaces that can eventually be wired into a NIC but are currently wired together in a loopback. This way, Beehive essentially generates its own UDP packets to process, allowing us to test both data paths within the network stack within the same application. Once Beehive processes the UDP datagram consisting of the encrypted value from the first AES unit, it pushes the data into the queue connected to the second AES engine, which is used to decrypt the encrypted value to be sent back into software through the last queue.

4.2.2 Beehive+METAL

The Beehive+METAL setup combines the hardware network stack with our key-value store unit implementing a subset of METAL's capabilities to emulate a scenario where another device on the network requests a value from METAL and our system is able to fulfill these requests without software interaction. The setup is shown in Figure 4.2.

The data flow in this setup is relatively straightforward. Like before, software uses Beehive to generate UDP datagrams containing METAL requests that are then routed back into its own inbound path. In an actual use case, this process would not be needed as the requests would come directly from another module on the network. Once the network stack de-encapsulates the UDP datagram, the request is then directed into the queue connected to METAL. After fetching the value associated with the queried key, it is then sent back to software. Once again, the fetched value would normally be routed back into the network stack to send back to the device issuing the request, but in simulation this is how we validate proper functionality throughout the whole system.

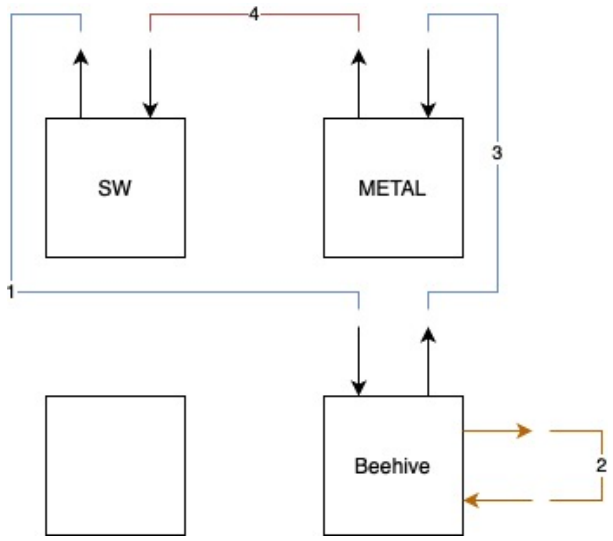

Figure 4.2: Beehive+METAL setup showing the logical flow of data.

4.2.3 Beehive+Falafel

The Beehive+Falafel setup utilizes the hardware network stack, memory allocator, and memory access unit to showcase a fully descriptor-based Beehive configuration, dynamically allocating packet buffers as enabled by Pengwing. The logical flow is shown in Figure 4.3.

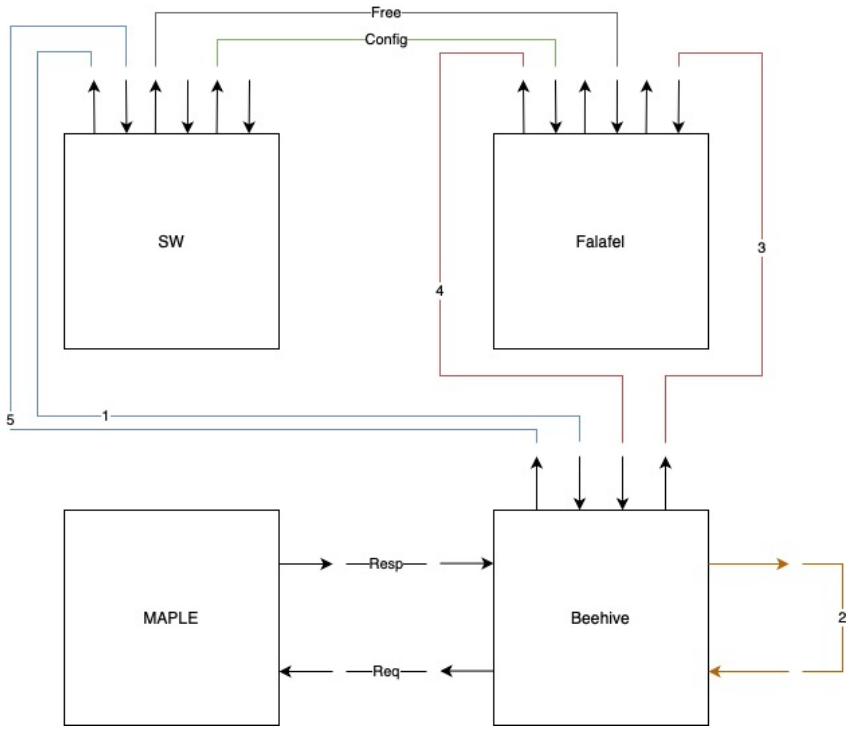

Figure 4.3: Beehive+Falafel setup showing the logical flow of data.

This experimental setup highlights a more complex configuration where each accelerator tile must interact with multiple sets of queues accordingly. The application begins by configuring Falafel using a single queue attached to a special configuration interface within the allocator. After allocating and writing to a buffer in memory, it then passes the size and pointer to Beehive through another Cohort queue. The Beehive tile then requests the data using the MAPLE interface and generates the UDP packet for loop back. While processing this echoed datagram, the Beehive tile then sends a memory allocation request to Falafel, which returns a pointer through a queue attached to the initial configuration endpoints. Beehive then sends requests to MAPLE once again to store the inbound data at this memory location, returning the size and new pointer back to software.

4.2.4 Data Stream vs Descriptors

Lastly, I also evaluated the aforementioned Beehive+Falafel setup against the baseline Beehive setup that processes data streams with regards to simulated throughput. Because we are not testing on FPGA yet, latency numbers would not be an accurate comparison between the two setups. In this benchmark, I also compared the performance between smaller and larger packet sizes to observe if there would be a difference in throughput based on the width of UDP datagrams.

4.3 Results and Analysis

All of the Pengwing setups were evaluated in RTL simulation using Synopsys VCS version Q-2020.03 running on Ubuntu 20.04.6 LTS with a simulated frequency of 2.5GHz and a Cohort queue size of 1024 elements. I recorded cycle and instruction numbers using Ariane hardware counters accessed through a CSR read call in software. With these performance counters, we measure the counts between the first Cohort queue push and the last pull from a Cohort queue, determining the full path cycle count for each test application. Combined with the simulated frequency, this data provides the necessary information to evaluate the throughput performance of each experimental setup.

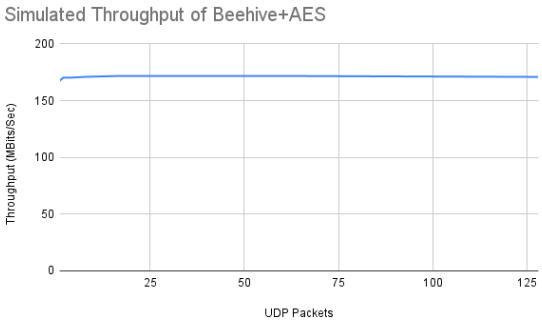

Figure 4.4: Beehive+AES Throughput.

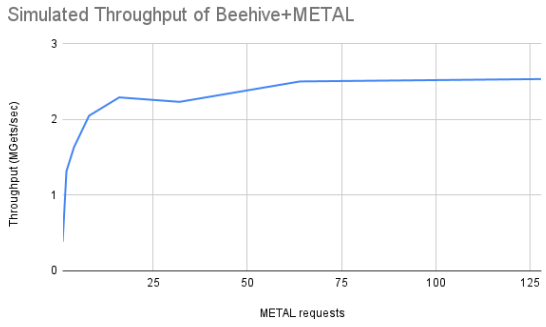

Figure 4.5: Beehive+METAL Throughput.

For the Beehive+AES setup, I set the UDP payload size at 32 bytes, each containing a single output from the AES encryption engine. As shown in Figure 4.4, the simulated throughput of the system hovers around 170 MBits/sec, which corresponds to around 660,000 packets per second.

Figure 4.5 shows the throughput metrics for the Beehive+METAL setup, where each METAL request (get) corresponds to one Beehive UDP datagram. Measured in MGets per second, the performance of the setup starts at 0.3 MGets/sec with just 1 METAL request and scales up to 2.5 MGets/sec with 128 requests. The progressive increase and eventual stabilization of the throughput can be explained by the caching performed by METAL as it traverses its internal data structure to access stored values. At the start, METAL's internal cache is empty, forcing full traversal of its data structure, increasing the number of cycles required to fulfill a request. However, after many get operations, the cache fills up, permitting quicker response times from METAL as well as higher throughput.

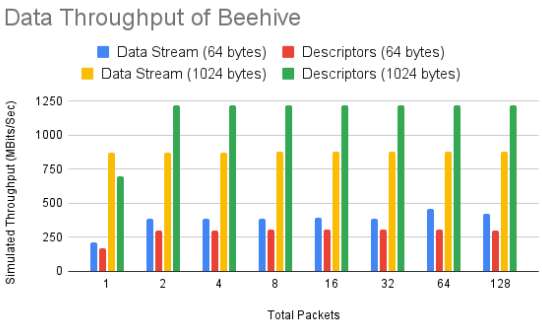

Figure 4.6: Data Throughput.

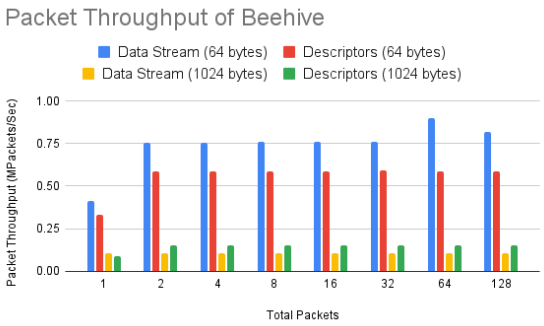

Figure 4.7: Packet Throughput.

Lastly, we look into the results of the Beehive setup with regards to both processing methods and packet sizes. As shown in Figure 4.6, we observe that batching data into larger packet sizes of 1024 bytes yielded higher data throughput compared to the smaller packet sizes of 64 bytes. For the raw data streams, we were able to push roughly 2.2x

more data through the system per simulated second with the larger packets compared to the smaller datagrams, whereas this ratio was approximately 4x when using packet descriptors. In fact, we are driving over a gigabit per second of throughput with the setup with full descriptor processing and larger packet sizes from a simple 6-stage in-order core. When we look at just the number of packets processed per second in Figure 4.7 though, we observe that using smaller packets actually enabled roughly 7x as many packets handled using data streams and around 4x as many for descriptors. This makes sense because while the smaller packets each contain one-eighth of the data of a larger packet, the overhead introduced by the extra header and metadata flits needed to be processed results in the 1024 byte packets being more efficient for throughput. Now, if we compare between the processing methods, data streams perform slightly better for smaller packets and descriptors process faster for larger packets. This is due to the overhead introduced by Cohort queues and the MAPLE subsystem, where queue accesses might be more efficient in small bursts but MAPLE can handle the larger number of memory requests quicker.

Chapter 5

Related Works

There have been related works researching similar aspects of increased collaboration across the whole system design. The solutions range from specific OS designs to other hardware-software co-design approaches. There are also other hardware system service implementations that we could potentially integrate into the Pengwing substrate in the future. Additionally, there are parallel projects into hardware networking solutions worth exploring.

5.1 OS Designs

OS designs can adopt a spectrum of data structure sharing models, from the shared-nothing model of Barrelfish [34] and fos [35] to the shared-most model of K2 [36] to the shared-everything model of Linux. Pengwing is unique in that system designers are not bound to a specific model but rather have the freedom to implement system services under any of these models, but it is still important to understand the utility behind each model. Barrelfish [34] presents a multikernel implementation where the OS is restructured into separate hardware-neutral functional units communicating through message passing. This improves multi-core scalability as new cores can be introduced to the system more freely as each node can function independently and only replicate state using

asynchronous messages when necessary. fos [35], which stands for factored operating system, utilizes message passing as well by factoring system services into smaller process servers that communicate to facilitate the full service, and it separates kernel services and applications into different cores to prevent resource contention. OSes like K2, on the other hand, utilize the shared-most model, where certain resources can be explicitly architected to be shared by different parts of the system. K2 [36] specifically deals with different cache coherency domains in mobile heterogeneous SoCs, where only some OS services like device drivers can extend state across domains while certain services like page allocation and interrupt management are run independently on each coherence domain with no state sharing between them. These seminal works in operating systems design provided the foundation for understanding how to optimize scalability and performance across the OS sharedness spectrum.

5.2 HW+SW Co-Design

Other research into better leveraging heterogeneous systems utilize a hardware-software co-design solution, similar to Pengwing. Works like M^3 [37], M^{3}x [38], and M^{3}v [39] introduce the idea of treating accelerators not as devices but at the same level as general-purpose cores through techniques like NoC-level isolation using data transfer units and a microkernel-based system. M^{3}x [38] improves on this by trading some isolation to allow multiplexing between different applications within a general-purpose or accelerator tile using a single centralized OS tile, enabling fast-path communication and context switching. M^{3}v [39] further extends the fast path by providing the framework for general-purpose cores to multiplex between applications without involvement from the OS tile. Other research studies like AuRORA [40] propose accelerator interfaces that preserve existing software semantics similar to Pengwing. However, the difference is AuRORA

promotes a tight coupling of accelerators and CPUs through their interconnect, whereas Pengwing encourages collaboration across the SoC using decoupled interfaces that enable more robust communication.

5.3 Other System Services

While this project focuses on a specific hardware network stack implementation, it is important to consider other system services that would be suitable in our design. I/O-facing accelerators for TCP offloading [41, 13, 14] and other network stacks like Limago [42], Tonic [43], and Microsoft's Catapult [12] are fairly well-researched, but only few examples of internal system services are available. Outside of our own Falafel allocator, there is Mallacc [9] which presents an in-core solution for acceleration of size class computation, free memory block retrieval, and memory usage sampling to make software `malloc()` calls faster. Another work titled NextGen-Malloc [44] restructures metadata for memory management, decoupling this information from the rest of application data in order to off-load the allocation and management functions; however, it expects a separate device/core to be available rather than implementing the off-loading hardware themselves. Apart from memory allocation, there is an accelerator for garbage collection [10] in a stop-the-world setting using separate hardware traversal and reclamation units, evaluated on a modified JikesRVM. Additionally, there is a hardware-based demand paging implementation [45] designed so that the CPU will try to locate missing pages using the accelerator while stalling its pipelines rather than raising an exception to the OS.

5.4 Hardware Networking Services

Focusing on works in the space of hardware network stacks, other researchers have their own approach to packet processing compared to Beehive. PANIC [46] is a programmable

NIC designed to offload computation from various applications running on different endpoints including accelerators and general-purpose cores. However, PANIC utilizes a central scheduler that drops packets when buffer space fills up to deal with deadlocks, which is problematic for connection-based protocols. Conversely, Beehive does not encounter this issue due to its robust NoC interface that preempts deadlocks, enabling support for protocols like TCP. ClickNP [47] is another hardware-based packet processing framework with support for integrating arbitrary networking functions. It is used to accelerate software network functions such as packet generation and load balancing, which means there is no support for higher-level protocols like in Beehive. Additionally, ClickNP necessitates a PCIE connection to a host CPU, which contrasts with the Pengwing's principle of configurability and collaboration.

Chapter 6

Conclusion

As processor designs begin to rely on more hardware acceleration, building a new OS concept that enables richer collaboration across all system components inherently becomes a hardware/software co-design problem, where we need to employ the right software methodology to leverage the accelerator-rich platform that we develop. Pengwing seeks to reshape the hardware/software stack, shifting the current usage model of accelerators to enable them to be used in more varied ways throughout modern OSes other than conventional user applications.

In this thesis, I focus on the networking aspect of our endeavor. We decided the best approach would be to modify Beehive, a modular NoC-based hardware network stack originally designed for direct-attached accelerators, to attach to Pengwing's system substrate. I began with an initial prototype integration of Beehive into our OpenPiton design, eventually implementing further features like configurable support for data streams or packet descriptors. After designing the full Pengwing SoC, I demonstrated the capabilities of flexible collaboration possible in our blended OS through a variety of setups that incorporate various system services available on our current SoC design like memory allocation and data encryption with the network stack.

There are still improvements to be made regarding Beehive's integration into Pengwing. Currently, only UDP is supported in the setup but more involved protocols like

TCP and RPC can potentially be integrated alongside UDP owing to Beehive's modular design. The current setup will also need to be slightly altered to process critical metadata information regarding each packet like IP address and ports for full functionality once we are able to test the system on FPGA. Section 3.6 expands on the other improvements we can make to the current design.

The evaluation setups that we do currently have, though, highlight just a small subset of the possibilities of efficient full system collaboration enabled by Pengwing. The integration of a hardware network stack into our processor design represents a crucial system service that can be utilized by other system services regardless of if they are implemented in software or hardware, serving as a basis for future works to incorporate more utility throughout our Pengwing hardware/software co-design.

References

[1] D. Giri, P. Mantovani, and L. P. Carloni, *Accelerators and Coherence: An SoC Perspective*, *IEEE Micro* **38** (2018), no. 6 36–45.

[2] C. Maertin, *Post-Dennard Scaling and the final Years of Moore's Law*, tech. rep., Hochschule Ausburg, University of Applied Sciences, Sept., 2014.

[3] J. Clover, "Apple Silicon: The Complete Guide." `https://www.macrumors.com/guide/apple-silicon/`. [Accessed 01-09-2024].

[4] Synopsys, "Cloud Servers." `https://www.synopsys.com/designware-ip/ip-market-segments/cloud-computing/servers.html`. [Accessed 01-09-2024].

[5] S. Steers, "SoCs are making an appearance in the data centre industry." `https://datacentremagazine.com/data-centres/socs-are-making-an-appearance-in-the-data-centre-industry`. [Accessed 01-09-2024].

[6] M. Braly, *A configurable H.265-compatible motion estimation accelerator architecture suitable for realtime 4K video encoding*, 2016.

[7] N. Jouppi, G. Kurian, S. Li, P. Ma, R. Nagarajan, L. Nai, N. Patil, S. Subramanian, A. Swing, B. Towles, C. Young, X. Zhou, Z. Zhou, and D. A. Patterson, *TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings*, in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, (New York, NY, USA), Association for Computing Machinery, 2023.

[8] A. Salah, "AES-128 Pipeline Encryption." `https://github.com/freecores/aes-128_pipelined_encryption/`. [Accessed 01-09-2024].

[9] S. Kanev, S. L. Xi, G.-Y. Wei, and D. Brooks, *Mallacc: Accelerating Memory Allocation*, *Operating systems review* **51** (2017), no. 2 33–45.

[10] M. Maas, K. Asanovic, and J. Kubiatowicz, *A Hardware Accelerator for Tracing Garbage Collection*, *IEEE MICRO* **39** (2019), no. 3 38–46.

[11] M. T. Arashloo, A. Lavrov, M. Ghobadi, J. Rexford, D. Walker, and D. Wentzlaff, *Enabling programmable transport protocols in high-speed NICs*, in *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation*, NSDI'20, (USA), p. 93–110, USENIX Association, 2020.

[12] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, *A cloud-scale acceleration architecture*, in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–13, IEEE, 2016.

[13] easics, "TCP Offload Engine." `https://www.easics.com/tcp-offload-engine/`. [Accessed 01-09-2024].

[14] missing link electronics, "TCP/UDP/IP Network Protocol Accelerator Platform (NPAP)." `https://www.missinglinkelectronics.com/ip-cores/npap-tcp-udp-ip-stack/`. [Accessed 01-09-2024].

[15] T. Wei, N. Turtayeva, M. Orenes-Vera, O. Lonkar, and J. Balkind, *Cohort: Software-Oriented Acceleration for Heterogeneous SoCs*, in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, (New York, NY, USA), pp. 105–117, ACM, 2023.

[16] K. Lim, M. Giordano, T. Stavrinos, I. Zhang, J. Nelson, B. Kasikci, and T. Anderson, *Beehive: A Flexible Network Stack for Direct-Attached Accelerators*, in *2024 57th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2024.

[17] N. Turtayeva, G. López-Paradís, K. Lim, B. Li, J. Farres Garcia, Z. Ma, M. Bui, B. Kong, T. Wei, T. Schlunk, and J. Balkind, "Pengwing: A Blended OS to Reshape the HW-SW System Stack." 2024.

[18] J. Balkind, X. Liang, M. Matl, D. Wentzlaff, M. McKeown, Y. Fu, T. Nguyen, Y. Zhou, A. Lavrov, M. Shahrad, A. Fuchs, and S. Payne, *OpenPiton: An Open Source Manycore Research Framework*, *Operating systems review* **50** (2016), no. 2 217–232.

[19] OpenHardwareGroup, "CVA6 RISC-V CPU." `https://github.com/openhwgroup/cva6`. [Accessed 01-09-2024].

[20] J. Balkind, K. Lim, M. Schaffner, F. Gao, G. Chirkov, A. Li, A. Lavrov, T. M. Nguyen, Y. Fu, F. Zaruba, K. Gulati, L. Benini, and D. Wentzlaff, *BYOC: A "Bring Your Own Core" Framework for Heterogeneous-ISA Research*, in *Proceedings of the Twenty-Fifth International Conference on Architectural Support*

for Programming Languages and Operating Systems, (New York, NY, USA), pp. 699–714, ACM, 2020.

[21] J. Balkind, K. Lim, F. Gao, J. Tu, D. Wentzlaff, M. Schaffner, F. Zaruba, and L. Benini, "OpenPiton+Ariane: The First Open-Source, SMP Linux-booting RISC-V System Scaling From One to Many Cores." `https://carrv.github.io/2019/papers/carrv2019_paper_12.pdf`, 2019. Accessed: 2024-09-01.

[22] Oracle, "OpenSPARC T1." `http://www.oracle.com/technetwork/systems/opensparc/opensparc-t1-page-1444609.html`. [Accessed 01-09-2024].

[23] M. Orenes-Vera, A. Manocha, J. Balkind, F. Gao, J. L. Aragón, D. Wentzlaff, and M. Martonosi, *Tiny but mighty: designing and realizing scalable latency tolerance for manycore SoCs*, in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, (New York, NY, USA), pp. 817–830, ACM, 2022.

[24] A. M. A. Kumar, A. Prasanna, J. Balkind, and A. Shriraman, *METAL: Caching Multi-level Indexes in Domain-Specific Architectures*, in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, (New York, NY, USA), p. 715–729, Association for Computing Machinery, 2024.

[25] J. F. Garcia, *Exploring hardware memory allocation in heterogeneous systems*, Master's thesis, Universitat Politechnica de Catalunya, May, 2024.

[26] M. D. Moffitt, *MiniMalloc: A Lightweight Memory Allocator for Hardware-Accelerated Machine Learning*, in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, (New York, NY, USA), pp. 238–252, ACM, 2023.

[27] Microsoft, "snmalloc." `https://github.com/microsoft/snmalloc`. [Accessed 01-09-2024].

[28] M. Kim, B. S. Kim, E. Lee, and S. Lee, *A Case Study of a DRAM-NVM Hybrid Memory Allocator for Key-Value Stores*, *IEEE computer architecture letters* **21** (2022), no. 2 81–84.

[29] W. Li, S. Mohanty, and K. Kavi, *A Page-based Hybrid (Software-Hardware) Dynamic Memory Allocator*, *IEEE computer architecture letters* **5** (2006), no. 2 13–13.

[30] Z. Xue and D. B. Thomas, *SysAlloc: A hardware manager for dynamic memory allocation in heterogeneous systems*, in *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–7, 2015.

[31] T. F. Foundation, "cocotb." `https://github.com/cocotb/cocotb`. [Accessed 01-09-2024].

[32] O. Kindgren, "FuseSoC: Package manager and build abstraction tool for FPGA/ASIC development." `https://github.com/olofk/fusesoc`. [Accessed 01-09-2024].

[33] A. Forencich, A. C. Snoeren, G. Porter, and G. Papen, *Corundum: An Open-Source 100-Gbps NIC*, in *28th IEEE International Symposium on Field-Programmable Custom Computing Machines*, 2020.

[34] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, *The multikernel: a new OS architecture for scalable multicore systems*, in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, (New York, NY, USA), pp. 29–44, ACM, 2009.

[35] D. Wentzlaff and A. Agarwal, *Factored operating systems (fos): the case for a scalable operating system for multicores*, *SIGOPS Oper. Syst. Rev.* **43** (apr, 2009) 76–85.

[36] F. X. Lin, Z. Wang, and L. Zhong, *K2: a mobile operating system for heterogeneous coherence domains*, *SIGPLAN Not.* **49** (feb, 2014) 285–300.

[37] N. Asmussen, M. Völp, B. Nöthen, H. Härtig, and G. Fettweis, *M3: A Hardware/Operating-System Co-Design to Tame Heterogeneous Manycores*, *SIGARCH Comput. Archit. News* **44** (mar, 2016) 189–203.

[38] N. Asmussen, M. Roitzsch, and H. Härtig, *M³x: Autonomous Accelerators via Context-Enabled Fast-Path Communication*, in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, (Renton, WA), pp. 617–632, USENIX Association, July, 2019.

[39] N. Asmussen, S. Haas, C. Weinhold, T. Miemietz, and M. Roitzsch, *Efficient and scalable core multiplexing with M³v*, in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, (New York, NY, USA), p. 452–466, Association for Computing Machinery, 2022.

[40] S. Kim, J. Zhao, K. Asanovic, B. Nikolic, and Y. S. Shao, *AuRORA: A Full-Stack Solution for Scalable and Virtualized Accelerator Integration*, *IEEE MICRO* **44** (2024), no. 4 1–9.

[41] Chelsio, "Terminator 6 asic." `https://www.chelsio.com/terminator-6-asic`. [Accessed 01-09-2024].

[42] M. Ruiz, D. Sidler, G. Sutter, G. Alonso, and S. Lopez-Buedo, *Limago: An FPGA-Based Open-Source 100 GbE TCP/IP Stack*, in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 286–292, IEEE, 2019.

[43] M. T. Arashloo, A. Lavrov, M. Ghobadi, J. Rexford, D. Walker, and D. Wentzlaff, *Enabling programmable transport protocols in high-speed NICs*, in *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation*, NSDI'20, (USA), p. 93–110, USENIX Association, 2020.

[44] R. Li, Q. Wu, K. Kavi, G. Mehta, N. J. Yadwadkar, and L. K. John, *NextGen-Malloc: Giving Memory Allocator Its Own Room in the House*, in *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, HOTOS '23, (New York, NY, USA), p. 135–142, Association for Computing Machinery, 2023.

[45] G. Lee, W. Jin, W. Song, J. Gong, J. Bae, T. J. Ham, J. W. Lee, and J. Jeong, *A Case for Hardware-Based Demand Paging*, in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 1103–1116, 2020.

[46] J. Lin, K. Patel, B. E. Stephens, A. Sivaraman, and A. Akella, *PANIC: A High-Performance programmable NIC for Multi-tenant Networks*, in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 243–259, USENIX Association, Nov., 2020.

[47] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, P. Cheng, and E. Chen, *ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware*, in *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, (New York, NY, USA), p. 1–14, Association for Computing Machinery, 2016.