UC Berkeley
UC Berkeley Electronic Theses and Dissertations

Title

A Modular Design Flow for NoC-embedded FPGAs

Permalink

https://escholarship.org/uc/item/6zg505rh

Author

Nguyen, Tan Quoc Duy

Publication Date

2023

Peer reviewed|Thesis/dissertation

A Modular Design Flow for NoC-embedded FPGAs

By

Tan Quoc Duy Nguyen

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Engineering – Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor John Wawrzynek, Chair
Professor James Demmel
Professor Sophia Shao
Dr. Stephen Neuendorffer

Fall 2023

A Modular Design Flow for NoC-embedded FPGAs

Abstract

A Modular Design Flow for NoC-embedded FPGAs

by

Tan Quoc Duy Nguyen

Doctor of Philosophy in Engineering – Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor John Wawrzynek, Chair

With the increasing growth of complexity and heterogeneity of modern FPGA fabrics, the conventional digital design flow relying on the standard vendor tools, from synthesis to implementation, has become more arduous than ever. This leads to an inordinate turn-around time which severely impacts the productivity of application developers in their quest to explore the design space. We propose an open-source tool flow built upon three principles: modularity, composability, and reusability to address the FPGA tooling productivity issue. Our tool flow features the design and implementation of Spatially Distributed Ensemble of Sockets (SPADES). SPADES modularizes an application into a parallel system of socket engines interconnected by a Network-on-chip. Each socket contains a fixed and reusable component for control and communication in addition to a custom partition tailored to specific application computation and memory demands. SPADES also presents a customized back-end tool that allows flexible and rapid composition of the socket physical implementations to generate a complete design. The results demonstrate that SPADES is 7.1x faster on average by reducing hours of compile time to minutes, while achieving comparable system-level performance when compared to a standard vendor tool for a set of benchmarks targeting a state-of-the-art FPGA architecture.

Contents

List of Figures

List of Tables

Listings

Acknowledgments

Graduate school is an arduous journey. Now that there is light at the end of the tunnel, thinking back, I would not have made it to this point without the support of many people.

First, I would like to thank my former colleagues and supervisors from the hardware group at Illinois Research Center in Singapore (ADSC): Kyle Rupnow, Swathi Gurumani, Liwei Yang, Yao Chen, and Professor Deming Chen (UIUC). I gained valuable research experience and skills thanks to the time that I worked with them. The experience that I had earned at ADSC continued to help me throughout my graduate study.

I would also like to thank my advisor, John Wawrzynek, for bringing me to Berkeley to join his research group, as well as his guidance and patience over the years. I was frustrated with myself in the first few years of my study for not meeting my own expectations on the progress of my research. Yet, John was very encouraging and tolerating. John gave me ample freedom to explore the research topics that I found most interesting. At the same time, he also put effort into making himself available whenever I would like to reach out for his advice. He took great care in ensuring that we were feeling comfortable and inclusive in our group. There are two things that I always try to learn from John. First is the ability to ask follow-up questions to guide the discussions and conversations. Second is managing busy schedules, such as teaching large classes while maintaining an optimistic outlook and enjoying life. Thank you, John.

The other person who had the most significant impact on my graduate study is Stephen Neuendorffer. Steve was my manager during my internships at Xilinx Research Labs. Steve was generous with his time by agreeing to be on my Qualifying and Dissertation committee member and advised my dissertation topic. Steve gave me plenty of advice — be it research, career, or life. When I felt unsure if I should pick a particular research problem; "If you do not do anything, you are not going to make any progress"; what he had said then still strikes me today. In retrospect, that was one of the most helpful advice I have gotten during my graduate school. Thank you, Steve.

I would like to thank Professor James Demmel and Professor Sophia Shao for serving on my Qualifying and Dissertation committee. Thank you for offering your advice on my research at the Qual, proofreading my thesis, and giving constructive feedback.

I would like to thank my industrial collaborators and researchers, in particular, Zachary Blair and Dr. Chris Lavin from AMD/Xilinx. Zac initiated the socket idea when I worked with him as an intern and continued to give me helpful advice related to the Vivado tool throughout this project. Chris was always prompt and active in giving technical assistance of the RapidWright software, or when I needed some feature requests. This work would not have been possible without the RapidWright developers and contributors. I also thank Dr. Jonathan Greene, Dr. Alan Mischenko, and Dr. Eddie Hung for giving feedback on our research as well as sharing their projects during our group's FPGA seminars.

Chapter 1

Introduction

With the end of Dennard Scaling [23] and the slowing down of Moore's law [53], we are entering a new era of domain-specific accelerators to unlock higher performance and better energy efficiency than the traditional multi-processor systems. Among many different accelerator types, Field Programmable Gate Arrays (FPGAs) offer a compelling solution thanks to their reconfigurability and spatial parallelism nature. FPGAs have been shown to outperform general-purpose processors and Graphic Processing Units (GPGPUs) in certain applications. Thus, they have been increasingly gaining their footing in many applications ranging from data centers in the cloud to embedded devices on the edge. For instance, Microsoft uses FPGAs to accelerate their web search engine [12], and deep learning workloads [24]. They are also being used for signal processing at wireless base stations [51], and computer vision pipelines in the automotive industry [7]. Besides application acceleration, FPGAs are well-suited for Application-Specific Integrated Circuits (ASICs) prototyping and emulation before chip tapeouts [10].

To cope with the growing computation and communication demands, FPGA vendors have been augmenting their devices with millions of logic elements in addition to a multitude of heterogeneous hardened blocks [82, 35]. This further exacerbates the longstanding, pathological FPGA tooling issues which severely affects their usability and inhibits their accessibility and applicability outside the niche community of hardware design experts.

To begin with, programming an FPGA is not straightforward. The programming model is Register Transfer Level (RTL) which captures bits and cycles. The act of programming an FPGA entails describing a hardware circuit (composed of logic gates and memory blocks) and its behavioral cycle-by-cycle in a Hardware Description Language (HDL) such as Verilog/VHDL. It is a tedious and error-prone process. High-level Synthesis [19] aims to bridge the gap between software programmers and low-level RTL designs by introducing a parallel software model that embraces procedural languages such as C/C++, thus improving programmers' productivity [43].

The next hurdle is vendor tools may consume hours or days to generate a bitstream of a hardware implementation, especially for some large and complex designs. If the implementation failed to satisfy user-defined constraints, such as desirable operating frequency, area,

or functionality, the whole development cycle would repeat, further hurting the productivity. Therefore, we think that the FPGA compilation time is among, if not the most, important problem to be addressed as we deal with the continuing trend of future FPGA device and design scaling.

The primary challenge is how we could improve the compile time while minimizing the impact on Quality-of-Result (QoR). The QoR is primarily concerned with two metrics: area utilization and maximum achievable frequency. The former measures how much on-chip resource consumed by an implementation. The latter informs the maximum clock speed that the implementation can operate. To mitigate the compile time issue, one could virtualize an FPGA with overlays. An overlay is typically a pre-defined architectural RTL design that gets implemented on an FPGA. The implementation usually does not change throughout running one or several applications. The overlay serves as a virtualization layer so that users do not need to interact with the FPGA directly. The overlay technique raises the abstraction level by allowing users to program an FPGA using a higher-level programming language than HDLs in tandem with a software-like compilation experience, thereby improving overall productivity. In addition, the overlay approach eschews the tooling problem entirely, since one usually does not need to modify the underlying overlay implementation. However, the key limitation of the overlay approach is that it yields sub-optimal QoR in comparison to custom hardware design. Therefore, overlays are often designed for domain-specific classes of applications [17, 78, 62] to achieve high efficiency. General-purpose overlays, such as softcore CPUs often leads to poor performance due to low QoR. They are mainly used for housekeeping tasks such as control and monitoring that do not require significant processing power [79]. In some cases, they provide a starting point for newly adopters to leverage to reduce the barrier to FPGA usability, and then a more custom solution could be built over time if needed.

In many applications, it is possible to modularize their design structures into smaller partitions. Some partitions, either within the same or from different applications, can share similar implementations. Unfortunately, the conventional vendor CAD flows fail to recognize such properties. A design is usually implemented in a top-down ("flat") manner, thus failing to take advantage of the design modularity to shorten the compile time either by parallel compilations of separate partitions or reusing existing compiled results of certain partitions. In some cases, a top-down implementation might not even yield the best QoR.

1.1 Contribution

Drawing from the aforementioned observations, we think that modularity is the first-order important factor to improve the compile time. We start by defining a modular execution model, namely SPADES, that well-suited for different types of parallel applications, including data- and task-parallel. We then build our tool flow on top of this model. SPADES is an ensemble of sockets. The socket physical implementations could be dropped in and out of a design without incurring any recompilation, hence facilitating design composability. The

sockets could also be reusable within one or different applications. All of these requirements could not have been accomplished without the support of a customized backend tooling and the architectural features of a target device.

In summary, our contributions are as follows. [1] [2]

1. We design and implement a novel tool flow for mapping parallel applications onto contemporary FPGAs. Our tool achieves fast compilation by promoting three primary ideas: modularity, composability, and reusability.

2. We discuss various design and implementation tradeoffs that influence compile time and QoR/performance. We present the principles of mapping an application to our tool flow in order to achieve high performance.

3. We evaluate the effectiveness of our tool flow in a multitude of aspects, such as compile time, QoR, and system-level performance. The experimental platform is the AMD Versal VCK5000 data center card from a state-of-the-art commercial FPGA architecture.

To the best of our knowledge, this is the first work in studying the compilation issue on the Versal architecture which features a novel hardened Network-on-chip. While some prior works are proposing and evaluating parallel or separate compilations to accelerate the compile time, our work is unique in adopting a holistic approach in co-developing the frontend and the backend flow following a unified model. We also exploit design reuse at multiple levels, as well as hardware/software partitioning techniques to further improve the compile time and QoR.

The rest of the dissertation is organized as follows.

- Chapter 2: We will review some key concepts that are relevant to this dissertation. We also discuss related work to this research.

- Chapter 3: We will provide the details of the SPADES execution model, and the micro-architectural design of a socket — the basic building block of our model.

- Chapter 4: We will show how to go about implementing a socket on the target device. We discuss the tradeoffs of various implementation schemes that we have considered.

- Chapter 4: We will present the backend flow which compiles the sockets to the FPGA, as well as techniques for improving QoR and compile time.

- Chapter 5: We will give some insights on how an application is mapped in our flow.

- Chapter 6: We will cover the experimental results and discussion.

---

[1]Our work has been published at [57].
[2]Our work is open-source at github.com/nqdtan/spades

- Chapter 7: We will conclude the research with closing thoughts, lessons learned, and future work.

Throughout the remaining sections, we use the terms *design* and *implementation* to describe our tool flow at various stages. When referring to *Design optimizations*, they include microarchitectural-level optimizations. This occurs at the block level (e.g., in RTL form). This also concerns scheduling optimizations related to data movements and compute executions. On the other hand, *Implementation optimizations* cover optimizations on the physical netlists. Such optimizations involve floorplanning, placement, and routing that improve QoR.

Chapter 2

Background and Related work

2.1 FPGA Architecture

Figure 2.1 demonstrates an example of what an FPGA looks like. An FPGA contains a multitude of logic elements and storage elements. One example of a logic element is a lookup table (LUT). There are typically two types of LUTs: logical LUT and memory LUT. The former implements a K-input Boolean function with a K-input LUT, while the latter serves as a small memory block, often known as distributed RAM. The memory LUT is asynchronous-read, synchronous-write; a read operation occurs immediately, whereas a write operation incurs one clock cycle. One other FPGA logic element is carry chain slices for implementing arithmetic circuits using fast carry lookahead. To hold the states of a circuit, there are some primitive storage elements called flip-flops (FF), or registers. The number of logic elements ranges from a few hundred in embedded FPGAs with small form factors to millions in data center-grade FPGAs. The logic elements and FFs are organized into columns of clusters, or Configurable Logic Block (CLB). Within a CLB, there can be two logic slices (or sites). A slice may contain only memory LUTs or logical LUTs. There is a fixed number of LUTs and FFs as well as carry logic per slice. The numbers of LUTs and FFs within a logic block vary from one FPGA architecture generation to another [77, 80, 81], as well as from different FPGA vendors. We refer to the logic and storage primitives of a slice as Basic Element Logic (BEL). Within a slice, some routing wires and multiplexers enable the connections between the BELs; those are the intra-site routing. The CLBs are inter-site routed by the Interconnect tiles (INT tile). There are a vast number of Programmable Interconnect Points (PIPs) in an INT tile for routing signals coming from the South, East, North, and West sides.

An FPGA is reconfigurable; this essentially means there is an underlying configuration circuitry. The circuitry configures LUTs, FFs, intra-routing within a site, and inter-routing among the sites based on the content of an input bitstream generated by an FPGA tool. One can reprogram an FPGA to perform different functionalities over time. Therefore, an FPGA can often be called a programmable fabric, or programmable logic (PL). The
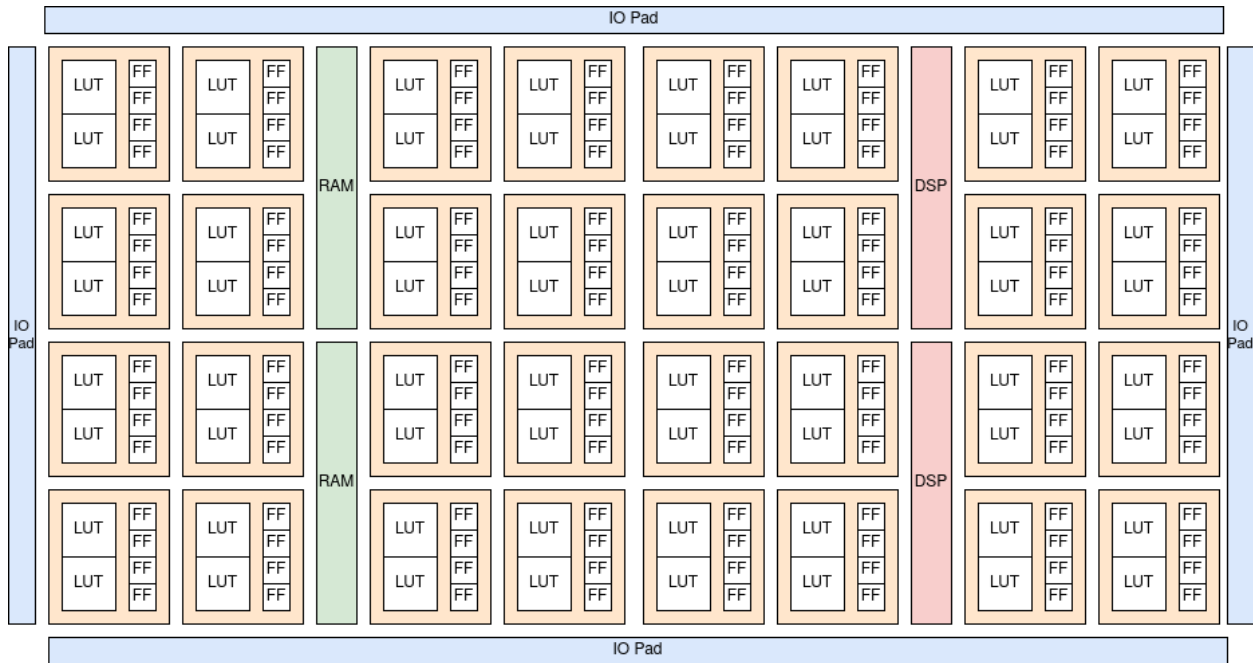
Figure 2.1: A hypothetical tile-based, columnar FPGA architecture of 4x8 Configurable Logic Blocks (CLB). Each CLB consists of 2 LUTs and 4 FFs. There is a column of on-chip RAM blocks and another DSP slice mixing between. At the rear are the IO pads for interfacing with IO devices. Not shown in this figure is the intra-site routing resource within a tile, the inter-site routing among the tiles, as well as the clocking network.

flexibility of FPGAs is the primary advantage over ASICs. For some applications whose requirements are rapidly evolving, an ASIC solution may quickly become obsolete, leading to higher engineering costs.

To improve the performance, FPGA vendors have been augmenting their fabrics with more heterogeneous hard blocks, such as Block RAM, Ultra RAM for dense on-chip storage, and DSP slices for supporting efficient multipliers and other math operations: those that would have cost numerous logic elements if they were implemented by LUTs and FFs. Some FPGA vendors offer application-specific hard blocks on the programmable fabric, such as Tensor blocks targeting Deep learning applications.

Another important resource of an FPGA device is I/O banks which are usually located at the edges of the device. Typical examples of I/O resources are PCIe interface, HDMI interface, Ethernet interface, external DRAM interface, etc. The IO banks offer interfaces to a wide variety of connected peripherals in a system. Therefore, this brings about a tremendous advantage of using FPGAs for IO computing at the edge rather than other programmable substrates such as CPUs, since FPGAs can directly process the data coming in and out of the IO banks.

One of the notable features of an FPGA is partial reconfiguration [71]. Certain regions of the programmable fabric can be reconfigured at *runtime* while the rest remains intact. Therefore, the overall compile time could be significantly reduced if we know in advance which pieces of the implementation that are fixed, and which could be changed at runtime. Typically, an FPGA design consists of two parts. One is the shell design that offers system functionality, such as clock management and data transfers via PCIe IO. The other is a custom design that implements user-specific functionality for a target application. The shell design is usually invariant, hence it is reasonable to keep its implementation unchanged, and the FPGA could be partially reconfigured for the user logic only. Previous work demonstrated how to exploit this feature to time share an FPGA with a computer vision pipeline that could not be mapped entirely to the FPGA [56].

Recent strides in FPGA architecture introduce a hardened Network-on-chip (NoC) embedded directly on the programmable fabric as well as other connected subsystems, thus forming a complex System-on-chip platform(SoC). The NoC offers many advantages to FPGA designs and implementations, chief among those are mitigating long routing wires between distant modules, or between a module and an IO interface such as a DRAM Memory Controller.

2.2 FPGA Tooling

2.2.1 Synthesis and Implementation

An FPGA backend flow typically consists of several phases as demonstrated in Figure 2.2. Firstly, a logic synthesis engine performs logic optimizations on an input HDL design. It then translates the optimized design to a representation that uses FPGA primitives (e.g., LUTs, FFs, BRAMs, etc.), a phase known as technology mapping. Next, the backend tool performs placement on the technology-mapped design to place the cells on the target FPGAs. A router then routes the placed design using the routing muxes and wires. The placer and router will try to optimize the design so that it can meet user-specified constraints on achievable maximum frequency, or area utilization. Therefore, the tool may carry out additional optimization steps before the placement and/or routing. Those usually involve physical synthesis optimizations that operate directly on the physical FPGA netlists. The last step is generating a bitstream from the final placed-and-routed implementation to configure the target FPGA.

Among the phases of the backend flow, placement, and routing usually dominate the overall compile time. Here, we refer to compile time as the time taken from an HDL input to a final implementation. If the user constraint is extremely tight, or if the input design is complex (in terms of the gate count), the tool will take longer to produce a solution. In particular, the tool may try replacing or rerouting the implementation several times to improve the QoR.

It is also worth noting that certain vendor tool may emit an output checkpoint at every
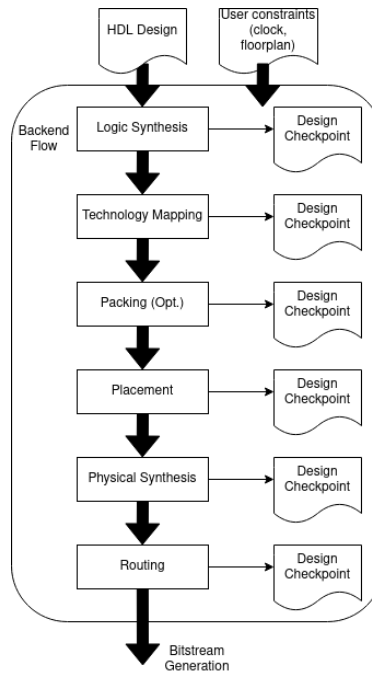
Figure 2.2: A Standard backend flow. It takes an HDL design and user constraints as input and produces a fully placed-and-routed implementation ready for bitstream generation. At each step, a design checkpoint could be generated to save the state of the step.

step of the compilation process. These design checkpoints save the states of the implementation at the corresponding steps. The checkpoints could be imported back into the design flow to continue with the subsequent steps. This is useful in cases where we would like to reuse the results of the previous steps for different explorations of the current step. Another scenario is that one could modify a checkpoint from a certain step, then import it to the design tool and invoke the next step. RapidWright [44] is a custom backend tool providing such capability. It can read a design checkpoint generated by the standard vendor tool, perform modifications on the netlist captured in the checkpoint, and write a new checkpoint. This flexibility allows users to add their own custom modifications, such as custom placer or router [47], and bypass certain steps in the design flow to reduce the compile time.

Finally, we note that there are several open-source FPGA tools besides the commercial vendor products such as AMD Vitis [5] or Intel Quartus [34]. Yosys & nextpnr [61] presents a fully open-source FPGA toolchain from synthesis to bitstream generation. The tool can be extended to support as many FPGA devices as possible by providing an FPGA architectural description file as input. Similarly, VTR [55] is an established open-source CAD framework for FPGA CAD-related research, including improving placement and routing algorithms, modeling new architecture, and so on. Previous work also studied the QoR gap between circuits generated by open-source tools versus vendors, and showed that the open-source

tools are catching up [31, 9] thanks to the novel optimizations of the open-source logic synthesis tool ABC [1].

2.2.2 High-level Synthesis

An input to a Logic synthesis tool is normally in the form of a Hardware Description Language (HDL), such as Verilog/ VHDL. HDL adopts a Register Transfer Level (RTL) model in which there is a notion of time (clock cycle); all assignments happen at every clock cycle. In addition, HDL constructs are declarative in ways that are used to compose certain graphical structures such as logical circuits. They do not share the same control and data flow semantics with procedural languages. Therefore, programming HDL requires a different thinking. It is often a tedious and error-prone process, especially for software programmers who are not familiar with hardware design.

High-level Synthesis (HLS) aims to bridge the gap introduced by the RTL model [11, 70, 33]. It allows users to write a program in procedural code, such as C/C++, and compiles the program to RTL, thus improving FPGA accessibility to software programmers. In essence, an HLS compiler transforms an untimed representation to a timed representation that eventually produces a valid, functionally corrected circuit description. That means the generated circuit must give output values that match its software counterpart. There are typically three phases in an HLS compiler: resource allocation, scheduling, and resource binding [20]. The scheduler computes the places to insert pipelined registers to break the control-data flow graph of an input program into multiple states controlled by one or several Finite State Machines (FSMs). In addition, stateful variables are mapped to registers or memory blocks in RTL. The arithmetic and memory operations of the input program are also transformed into the corresponding operations in the RTL model. Expensive arithmetic operations such as large-bitwidth multipliers could get mapped to hard blocks such as DSP slices for resource-wise efficient implementation. An important factor to consider is the QoR of the generated circuit. Ideally, we would like the generated circuit, when being implemented on an FPGA, to meet a target clock frequency. Therefore, an HLS compiler also takes as input a user clock constraint. The clock constraint impacts the HLS scheduler in how it schedules the operations, and inserts pipelined registers. It builds an estimated timing model of various operations to ensure reasonable delay estimations when performing scheduling. For example, if there is a chain of operations whose total estimated delay exceeds the target timing, the compiler will place pipelined registers on the chain to break it into smaller paths. Different target input clock constraints lead to varying generated circuits by an HLS compiler.

Besides, for an HLS compiler to produce a highly-optimal circuit, an input program needs to follow several best HLS practices or HLS model. This is due to the use of procedural languages whose model is inherently sequential execution semantics. When coding an HLS-based program, a software designer should reason how the code gets compiled to some computational structures or datapaths. Thus, the designer is still expected to possess some familiarity with hardware design concepts. Nevertheless, the advantage of relying on an HLS model is that the designer does not need to be concerned with the *time* notion, or clock

cycle-based behaviors as in the RTL model. In other words, the exact timing, or cycle of any operation is not a concern to a designer and is abstracted away in the HLS model, leading to a productivity boost.

The HLS approach attempts to address the programmability issue, which in turn ameliorates productivity. Yet the detachment of an HLS tool from the backend tool may impact the overall compile time and QoR. For example, the hierarchical information of the input program may be lost during the lowering to RTL form by HLS. In addition, HLS typically generates less optimal circuits in comparison to hand-designed RTL; it may incur additional resource usage or longer critical path delays. This is primarily because it relies on some delay estimations of certain operations that may not be accurate. Nonetheless, some studies demonstrated that HLS is catching up with hand-designed RTL [60], or enabling design space exploration for performance optimizations that would have been difficult to achieve with the RTL approach [87, 84]. HLS was also employed in designing complex applications such as video decoders or deep learning accelerators [49, 13, 83]. Some studies proposed alternative HLS input languages than C/C++ in terms of explicit parallel programming models [58], or productivity and popularity with software programmers [30]. [14] took a different approach by proposing high-level synthesis on an application binary program that aims to bypass the user programming step completely. In most cases, however, the prior works usually rely on existing compiler frameworks such as LLVM [65], to perform scalar and loop optimizations on an input program (the frontend phase) before invoking HLS-specific transformations.

Traditional HLS tools that employ static schedulers suffer from generating sub-optimal circuits for applications that exhibit runtime dependencies. Since they cannot resolve the dependencies at compile time, they resort to conservative scheduling solutions. Dynamically-scheduled HLS aims to address such inefficiencies by introducing several latency-insensitive, dataflow constructs for resolving the dependencies at runtime with extra area overhead [39].

2.3 Embedding NoC on FPGAs

2.3.1 Academic Perspectives

Traditional routing network on a programmable fabric contains wires and switchboxes. Inside a switchbox, there are PIPs for configuring the wire connections to form longer wires to route from a source to one or multiple sinks. The routing is determined by the tool (router), and implemented at compile time (loading a bitstream) using a circuit-switch routing style. A routing path is fixed and non-shareable. Circuit-switch routing is efficient in that the routed paths are static, thus the behavior of the circuit netlist is deterministic. However, the drawback is that it imposes a large routing footprint, thus leading to inefficient implementation due to long routing wires between cells that are far apart if the tool could not find any shorter connections. Another issue is routing congestion, in that there are limited amount of routing resources for a particular fabric region that are densely packed with logic cells that have high connectivity demands.

To mitigate these issues, there are existing works that build a soft NoC on top of the programmable fabric [42]. A soft NoC is constructed by primitive elements such as LUTs, FFs, and occasionally dense RAM blocks such as BRAMs. A NoC helps decentralize the routing of PL modules, thereby alleviating issues such as long connected wires and congested regions. For instance, assume a design has several modules that want to access an external DRAM controller. A typical implementation would lead to a situation in which the region near the DRAM controller (usually at the edge of a device) is highly packed and congested. Instead, if a NoC were virtualized on the fabric, the placement of the modules would spread out since each module connects to a NoC router. Furthermore, the routing from modules to the DRAM is contained within the NoC links. The apparent disadvantage of a soft NoC is due to its cost, and poor performance since it is made of programmable fabric resources. The NoC incurs extra latency on the communication paths due to the extra pipelined registers or buffers. Several works propose cost-efficient soft NoC architectures [40, 41] by reducing the amount of buffering in a NoC router. Additionally, some forms of NoC routing, such as packet-switching mode [21], do not maintain the deterministic of a circuit netlist, i.e., the communication latency could vary. Since packet-switch routing allows sharing connections, a NoC router introduces an arbitration mechanism to select among concurrent flows of signal. This typically should not be a problem if the PL modules employ some latency-insensitive communication protocol, such as handshakes. If, however, the determinism constraint is imposed on the communication between PL modules, some NoC designs allow users to configure the Quality-of-Service (QoS) to produce a NoC routing that provides deterministic latency.

Some researchers [2] propose embedding a hard NoC on an FPGA to bypass the issues of soft NoC. Similar to other hard block resources, a PL module could directly access a hard NoC resource on the programmable fabric. A hard NoC is generally cheaper and faster than its counterpart implemented by programmable logic resources. [63] discusses NoC modeling on an open-source CAD tool flow and placement optimizations to achieve better QoR in terms of congestion, critical path delay, and aggregated NoC bandwidth.

2.3.2 Commercial Architectures

Versal is the latest AMD FPGA architecture featuring a novel hardened NoC [25, 64]. The NoC presents a unified communication backbone with high-bandwidth data transfer between the Programmable Logic (PL), the Processing Systems, the DDR, and other platform subsystems. According to [69], on the PL, the NoC endpoints, namely NoC Master Unit (NMU) and NoC Slave Unit (NSU) are laid out in columns. A NoC Packet Switches (NPS) performs transport and packet switching on a path between an NMU and NSU. An NMU issues requests from the PL to the network, whereas an NSU services requests from the network to the PL. The NoC supports both AXI Memory-mapped and AXI Stream interfaces. Regarding our target FPGA device, the hard NoC runs at 1GHz on a 128-bit datapath. Hence, the maximum bandwidth of a NoC link is 16GB/s. The NoC is statically routed by the vendor NoC compiler during the implementation. Users characterize the NoC con-

figuration at the design phase, such as the connectivities between which NMUs and NSUs, the estimated bandwidth and latency requirements for each connection. The NoC compiler takes into consideration these constraints to provide a solution that satisfies them.

Another commercial FPGA architecture, Intel AgileX, also employs a hardened NoC interconnect called FlexNoC [32]. FlexNoC adopts a packet-switch routing style and 128-bit datawidth. Achronix Speedster7t builds a 2D NoC with rows and columns running through the programmable fabric whereby each row or column features two 256-bit AXI channels [4]. It also supports packet-based transactions.

2.4   Overlays

Overlay is a well-known technique to address productivity issues, especially those that are concerned with programmability and compile time. An overlay is a virtualization of a logical compute organization on a physical Programmable substrate. The flexibility of FPGAs lends itself adaptable to a variety of compute organizations. For example, one could implement a general-purpose processor on an FPGA. The processor could have several vector units, or support out-of-order execution. It could also incorporate multi-level caches to improve data locality. The more generality an overlay architecture possesses, the more applicability it becomes. This means it can support a wide range of applications. Therefore, it removes the need for recompilation of the FPGA design. General-purpose processors also bring the benefit of allowing programmers to write programs in high-level languages. Thus, the programmability issue could be solved as well, if the overlay implements an architecture whose Instruction Set Architecture (ISA) is supported by some existing software compiler toolchains.

One of the main applications of FPGAs is hardware emulation and validation. Before taping out, an ASIC design is synthesized into one or several FPGAs to confirm its synthesizability and functionality. In the FPGA computing domain, this emulation, or overlay, approach is utilized to provide a software-like experience to users. Nonetheless, there are a few important considerations to keep in mind when synthesizing an overlay to the programmable fabric. The first issue is the QoR of the overlay implementation. Some constructs, intended for ASIC implementation, do not map efficiently to FPGA resources. One example is a large multi-ported Register File (RF) in a CPU. In ASIC technology, an RF could be mapped to SRAMs or latches compactly. However, an RF causes a huge register pressure when mapping to FPGAs. Therefore, an overlay architecture often needs some redesigns to ensure efficient implementation on FPGAs, otherwise, it could yield a circuit with poor QoR (e.g., low achievable maximum frequency). The second point is the tradeoff between generality and performance. A more generalized overlay may entail a more expensive circuit, leading to longer critical paths. A less generic one will require recompilation more often when targeting different applications requiring distinct functionalities. The third aspect is the high-level mapping toolchain. We need to provide a tool flow that compiles an input program to the overlay architecture. This is typically not an issue if the overlay is based on

some machine organization that already has supported compilers. If that is not the case, the designers must take great care in devising an effective compilation flow to reach reasonable performance.

[37, 36] exploited the DSP blocks to build efficient FPGA overlay with high throughput, and used OpenCL as an input programming language. [45] built an architectural template that allows users to vary the hardware design for design exploration, while also employing OpenCL for programming. In a similar vein, [15] translated an input program to a multi-stage dataflow pipeline architectural template for a more effective overlapping of memory latency and computation. [46] presented a framework for a rapid loop accelerator by building a soft coarse-grained reconfigurable array (CRGA). [66] built a CRGA with keen attention to the communication network and data movement that uses HLS for code generation. [48] is an automated framework for generating efficient domain-specific overlays by employing design space exploration techniques and is also compatible with HLS. [47] is a custom router overlay that exploits the fabric regularity to achieve fast compilation and better achievable frequency. There are overlays built towards specific domains, such as sparse matrix computation [62, 38], graph processing [29], stencil [16], or neural network [3, 88, 86, 85, 50, 8, 67, 28]. These are typically accelerator-based designs that offer programmable capability at runtime to a certain degree. On the other hand, most general-purpose overlays utilize common parallel languages/models such as OpenCL or HLS for input designs. We also note that besides computational or accelerator based overlays, there are others that serve as communication infrastructure, such as on-chip network routers, or virtualized memory architecture as shown in CoRAM [18]. Our work is similar to CoRAM in several aspects. We also develop a distributed compute-memory architecture whereby each processing element (or socket) defines its own compute and memory space, and the control of data movement between off-chip and on-chip memory is carried out by a software mechanism. Yet, the details of the control mechanism and data movement are different between both works. For example, we use DMA-based data transfers instead of stream-based communication in CoRAM. Our controller also schedules computation in addition to data movement tasks. More importantly, CoRAM does not address the compile time issue as in our work. Still, the idea of employing a control thread for explicit data movement through a high-level software code helps simplify application development, and subsequently improves the productivity.

2.5   Separate Compilation and Task Partitioning

Separate compilation refers to techniques that divide up an application to smaller modules for parallel compilation. Standard vendor tools make use of this approach during the synthesis phase in which there are different RTL blocks of a design that could be synthesized or implemented separately at the same time [72]. [76] used partial reconfiguration and built a packet-switched overlay network to support incremental and parallel compilation of separate modules to reduce the compile time. [75] built an array of soft processor cores and introduced a compilation flow that is akin to the traditional software compiler toolchains.

[73] proposed dedicated interconnect wires between processing elements to remove the need for a soft NoC overlay for communication. HIPR [74] took advantage of an HLS tool for application mapping instead of manual RTL designs. An application is split into smaller modules for fast incremental compilation with partial reconfiguration support. [59] implemented a hierarchical partial reconfiguration flow whereby smaller partitions (or pages) can be combined into larger pages if needed to increase the flexibility of application mapping. Autobridge [26] optimized the HLS designs for multi-die FPGAs by providing the floorplanning information earlier in the HLS phase, thus facilitating the insertion of pipelined registers to optimize timing for die-crossing nets. RapidStream [27] split an HLS dataflow design into separate tasks for parallel compilation, and then assemble the physical implementations.

We adopt a similar approach in separate, parallel compilations of smaller tasks in our backend flow, and we also use HLS to provide a design entry to our flow. Nevertheless, we would like to highlight a few key differences. The uniqueness of our approach is that we tailor an input application to follow our parallel execution model, hence facilitating better mapping in terms of compile time and performance. In addition, we exploit design reuse opportunities within each separate task or processing engine. We conduct our studies on a state-of-the-art FPGA device that features a hardened NoC. While we do not attempt to implement task-level partial reconfiguration in this work due to time constraints, we think that we could benefit from previous works by extending their ideas to work with our tool flow.

Chapter 3

From SPADES to Socket

3.1 SPADES execution model

SPADES presents a parallel system of sockets connected by a NoC. As illustrated in Figure 3.1, a socket is a processing element containing a control unit, memory block, and a compute block. The control unit is the brain of a socket. It performs the scheduling of several operations such as data movements or computations that involve the memory and compute block. The control unit of one socket can also synchronize with other sockets by means of exchanging messages over the network.

The memory block is a software-managed scratchpad rather than a cache. Therefore, a socket must initiate data transfer to/from the memory block. The compute unit is not exposed to the network; it accesses the memory unit for data. Different sockets may have unique implementations of the compute units or memory units or a group of sockets may share similar implementations. Note that the socket's sizes are not necessarily identical; some sockets may require larger areas for compute or memory demand.

The NoC is statically routed; the routing is done at compile time. In addition, we adopt a packet switch routing for the NoC in this model. This allows different communication paths to share common routing channels. A routing channel is a connection path between two adjacent routers. In this model, each router is associated with one socket. There is a maximum bandwidth for each routing channel.

Here we only define the high-level organization of computation and communication, hence an execution model. The internal details of a compute, memory, or control unit remain abstract. The advantage of this model is its flexibility in adapting to different parallelism models, such as Task-level parallelism (each socket implements different tasks), or Data-level parallelism (all sockets share the same implementation of compute and memory, but operates on different workloads.) as shown in Figure 3.2. The drawback of this model is that it is not efficient for applications consist of simple dataflow pipelines. In particulars, if each dataflow task in the pipeline is mapped to different sockets, this will incur some overhead of scheduling and coordinating the data movement between the sockets; it involves getting
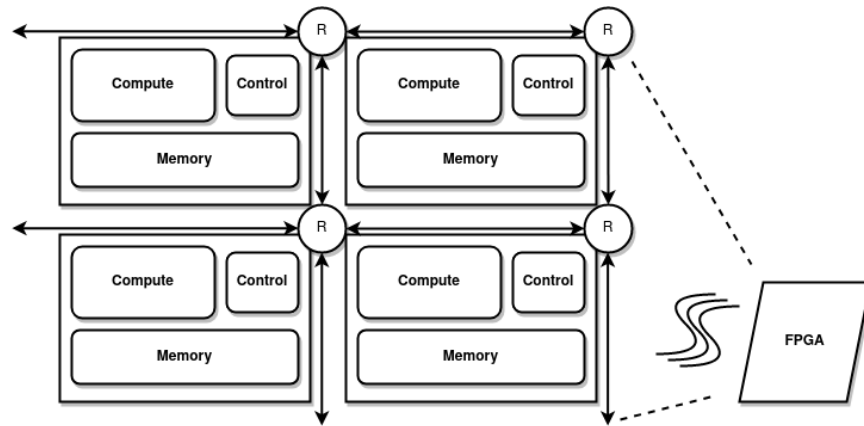
Figure 3.1: SPADES featuring a parallel, distributed system of socket engines.
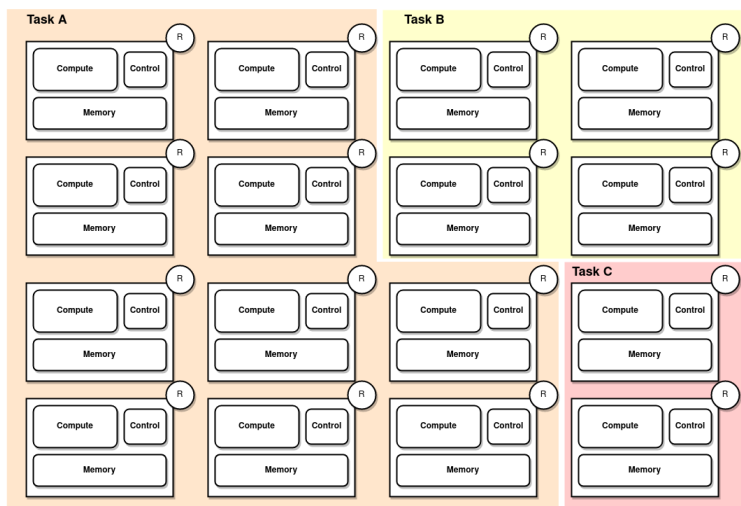
Figure 3.2: Each task of an application could be assigned to different groups of SPADES' sockets.

correct data from a memory unit of one socket, setting up a data stream, sending the stream to another socket over the network, and writing it to the destination socket's memory unit. It would have been more efficient with a stream-based communication (e.g., FIFO). In that regard, one could consider merging all or part of the dataflow pipeline to a compute unit of a socket to minimize the communication overhead.

3.2 SPADES Realization on Programmable Logic

The previous section discusses SPADES as an abstract, high-level execution model. Since the target device of this work is FPGA, this brings a question of whether this model could be tailored to a programmable fabric. We could virtualize an FPGA with this model as an overlay. However, if all the sockets employ fixed implementation, it will lead to a loss of efficiency for some applications that do not mesh well with the socket's compute or memory characteristics. A better approach is allowing a socket to be reconfigurable: one could generate a specialized datapath for a socket according to a specific implementation. But this may impact the compile time if we have several sockets of distinct implementations.

As seen from previous work, this problem could be fixed by doing separate, or parallel compilation to compile different socket's implementations in parallel. To ensure better composability of stitching the sockets into a complete system, the compilation must support location-agnostic implementations. In other words, a socket's implementation could be placed in similar groups of physical slots across the device. We could also improve the compile time within a socket. If there is a socket component that is invariant to the socket's performance in different applications, or similar in different sockets, we could produce a fixed implementation for it. For instance, a control unit could be fixed if the sockets employ the same scalar processor as their controller.

Since different applications likely require different communication patterns, if the NoC is mapped to the FPGA routing fabric, this would require additional compile time to reroute the NoC. However, if the NoC is hardened, their routing could be reconfigured faster. Additionally, the NoC should use a standard and unified interface with the sockets to enhance the composability.

In general, there are several considerations if one would like to achieve better compile time and performance when mapping the SPADES model to PL:

- Having smaller, modular sockets of different functionalities is better than a monolithic socket.

- Compatible physical placement locations on the target device enable location-agnostic socket implementations.

- Hard system interconnect with a standardized interface simplifies the routing between the sockets.

These considerations imply that ultimately, a co-design between hardware (target device, physical implementations) and software (execution model, logical designs) is essential in reaching our goals.
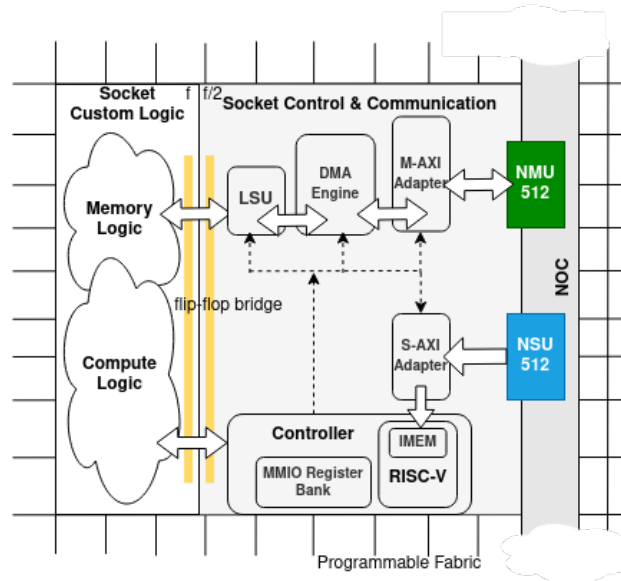
Figure 3.3: Socket Microarchitecture. The thick arrows denote the direction of data flows (e.g., read/write, request/response). The thin black arrows denote control relationships.

3.3 Socket Microarchitecture Design

Figure 3.3 shows the socket microarchitecture that consists of two components: Socket Control & Communication (CC), and Socket Custom Logic (CL). Socket CC is a fixed overlay logic, while Socket CL is customized to application-specific compute and memory demands.

3.3.1 Socket Control & Communication

3.3.1.1 AXI Logic

A socket utilizes the NoC endpoints to communicate with other modules through the NoC. As the NoC endpoints use AXI interface, we implement an AXI adapter logic for each endpoint present in a socket. AXI is a standard bus protocol introduced by ARM [6]. AXI features a scalable, high-throughput on-chip interface a master (requestor) and a slave (client). The master sends AXI requests, while the slave serves the requests. Multiple masters and slaves could participate in the communication as long as they were assigned unique IDs. There are separate channels for read and write operations.

Depending on a socket configuration, it could have several NoC endpoints enabling independent streams of requests/data. Our design supports both AXI Memory-mapped and AXI Stream interfaces. A master AXI interface attaches to an NMU, whereas a slave AXI interface connects to a NSU. Each socket's NSU (or slave interface) is assigned a unique

address in the systems. There are many different scenarios involving AXI transactions. A socket may communicate with the external DRAM for data by either issuing write requests or read requests. This is achieved with a NoC routing from the socket's NMU to one of the NSUs of a Memory controller of the DRAM. In our target device, there are 4 DRAM Memory controllers (DDRMC) whereby each has 4 NSUs. Note, however, that an NMU can only connect to one of the four NSUs of a Memory controller. In addition, this communication requires the AXI Memory-mapped protocol. Another scenario is inter-PL-module communication achieved by routing a source's NMU to a destination's NSU. The NMU and NSU are on the PL fabric. An NMU can connect to multiple NSUs due to each NSUs being assigned a unique address. We can implement this communication using either AXI Memory-mapped or AXI Stream interface.

The AXI datawidth of an NMU and NSU on Versal PL is 512-bit. To reach the full NoC bandwidth of 16GB/s, the PL AXI-interface logic must either achieve 500MHz at 256-bit, or 250MHz at 512-bit. We adopt the latter in our AXI adapter design since this timing target is more feasible. The AXI adapter is a complex logic consisting of separate read and write datapaths. Each functionality requires some buffering to achieve high throughput data transfer. In particular, the requests can be issued successively in a pipeline fashion without waiting for prior requests to be completed. This would allow multiple in-flight requests to overlap effectively to attain high throughput data transfer. The AXI adapter does not reorder the AXI transactions; the requests are issued in order using FIFOs to the NoC. Therefore, we only need to handle the responses in the order they are received from the network. Besides the buffering logic, the AXI adapter also needs to handle AXI transactions that cross the 4KB boundary as stipulated in the AXI specification. Any transaction whose length is longer than 4KB will be chopped into multiple smaller transactions that fit to 4KB limit. The AXI adapter connects to the socket's DMA engine using multiple FIFO interface channels (latency-insensitive) that encompass read request, read data, write request, and write data. It is the responsibility of the DMA engine to direct the data stream to/from the on-chip memory resource of the socket. Figure 3.4 and 3.5 show the block diagrams of our AXI Read and Write logic designs, respectively.

There are two kinds of AXI adapters in our socket design. The first one is a master AXI which is controlled by a DMA engine. The other one is a slave AXI. The slave AXI logic can also access the memory resource of the socket. Figure 3.6 shows the slave AXI block design. An incoming AXI transaction received by the slave AXI will either access the memory unit for a data bulk or the set of memory-mapped control registers. The functionalities of these registers include reset, synchronization, status, and control (CSR), or writing to the Instructional memory of the socket softcore.

All of the data channels of the master and slave AXI adapters use a datawidth of 512 bits. This allows us to pack as many bytes of data in a single data beat, thus leading to better utilization of memory bandwidth. However, this could be a waste of bandwidth if we only send a handful of control signals between the sockets. But such events should be rare, and only necessary when one socket needs to control or synchronize with another. Vivado NoC compiler allows users to specify the required bandwidth per NoC connection. We could
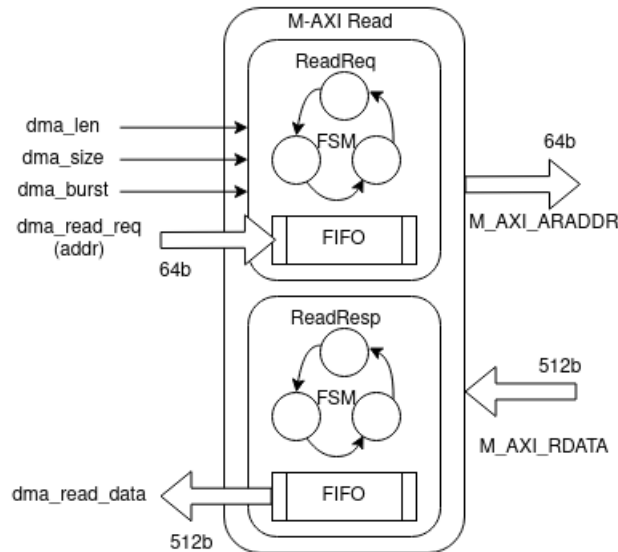
Figure 3.4: Master AXI MM Read design. Separate FSMs control the request and response logic. FIFOs are used to buffer incoming requests or responses.

set the bandwidth to minimal for these control paths so that the NoC compiler can utilize the routing resource for more demanding communication paths.

Besides the master AXI adapters for moving bulks of data, the socket uses a simple AXI module for sending transactions intended for control purposes, namely Control MAXI. The control transaction does not use burst mode; it only contains a single data beat. The purpose of this module is to exchange short messages between different NoC clients, such as checking one's status (Control MAXI Read) or notifying another it has completed some operation or reached a checkpoint (Control MAXI Write). The Control MAXI's transactions are low-bandwidth and should not occur frequently. The AXI Control shares an NMU with the AXI master adapter as shown in Figure 3.7.

3.3.1.2 DMA Logic

The DMA engine configures the master AXI adapter to issue the AXI requests to the NoC and handle the responses received from the NoC. Figure 3.8 shows the design of a DMA engine. The DMA engine implements a simplified version of the AXI logic. For instance, the transaction length could be set arbitrarily in a single request. The AXI adapter handles chopping off the transactions according to the maximum AXI burst length parameter. This reduces the complexity of the DMA design. The DMA provides additional buffering for each AXI channel and ensures that the responses are processed in the order that their request counterparts have been sent.
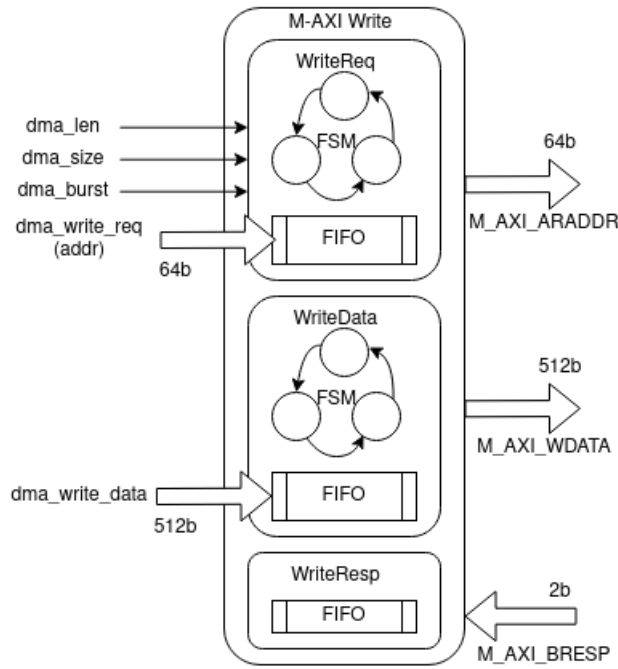
Figure 3.5: Master AXI MM Write design. Separate FSMs control the request and response logic. FIFOs are used to buffer incoming requests or responses. There is additional logic to handle the write response.
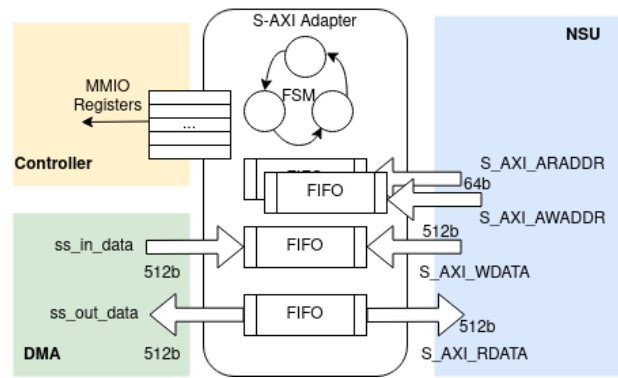
Figure 3.6: Slave AXI MM block design. There is a single FSM that governs both read and write operations. FIFOs are used to buffer requests/responses via the NSU and data streams via the DMA. The Slave AXI adapter uses a set of MMIO registers accessed by external masters via the NSU to interact with the controller.
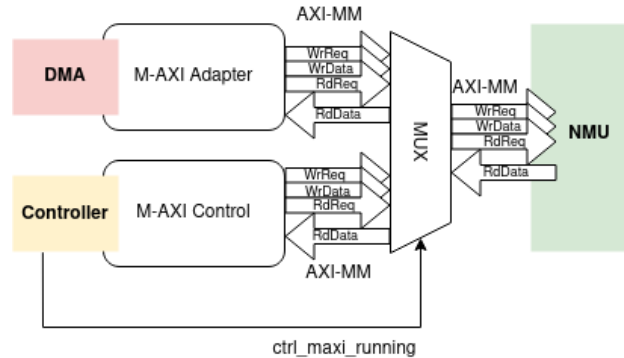
Figure 3.7: MUXing Master AXI-MM between AXI Adapter and AXI Control. Only one could interface with the NMU at a time. The selection is statically governed by the controller.
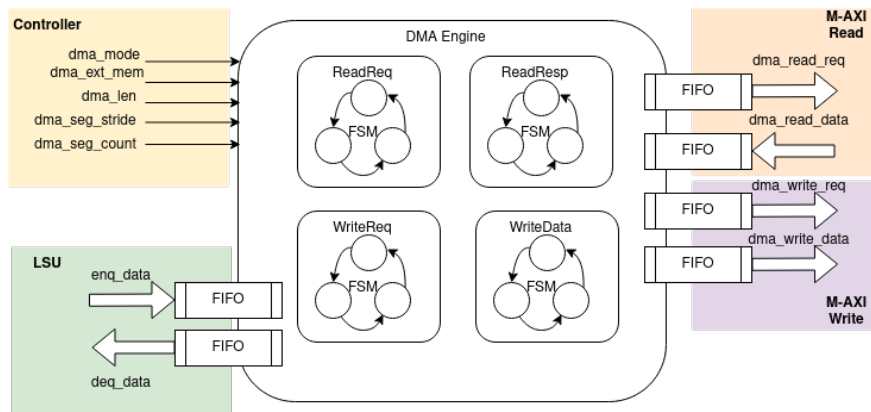
Figure 3.8: DMA Engine Block diagram. There are 4 separate FSMs for each read/write request and response logic for an M-AXI Read and Write module. The DMA sends or receives a data stream from an LSU. It receives the configuration from the controller.

The DMA engine employs a set of registers to control its functionality. Table 3.1 shows the list of control registers and their descriptions Broadly speaking, the registers specify a layout of the external memory to fetch to the socket (or write from the socket). The register values are unchanged until the DMA operation completes (textitdma_done is asserted). With these configurations, one can perform a DMA transfer of memory tile with a variety of rectangular shapes as shown in Figure 3.9.

One might be familiar with the term "block descriptor" as seen from other DMA designs that essentially achieve similar outcomes. The internal control logic of the DMA engine will transform a block descriptor into corresponding AXI requests. Typically, multiple AXI requests are needed to accomplish one DMA transfer. The DMA registers are managed by a

Table 3.1: DMA Registers

Register	Functional description
dma_start, dma_done	Provides Control and Status checking of the DMA
dma_mode	Provides the mode of the DMA (Read/Write)
dma_ext_addr	Provides the offset of the external memory for access
dma_len	Provides the segment length of a memory transfer
dma_seg_stride	Provides the stride between each external memory segment.
dma_seg_count	Provides the number of segments per DMA operation
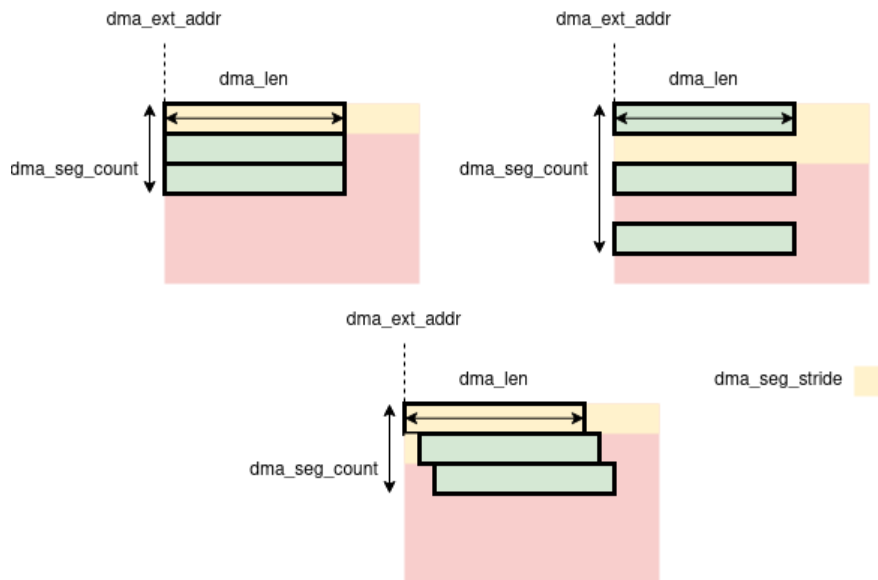
Figure 3.9: Several examples of external memory layouts specified by different DMA configurations

socket controller. Details of the controller will be provided in the subsequent section. Note that these registers are run-time programmable. Their values are modifiable at runtime instead of re-implementing the design, which helps to improve productivity. The mechanism for setting up the DMA registers will be discussed when we cover the controller functionality.

3.3.1.3 Load Store Unit

While the DMA engine handles communicating with the AXI adapter for external memory, the Load/Store Unit (LSU) works with the on-chip memory resource of the socket or Memory Unit. We will cover the memory unit in greater detail later; one important thing to keep in mind for now is that the memory unit comprises of multiple memory blocks organized

Table 3.2: LSU Registers

Register	Functional description
lsu_start, lsu_done	Provides Control and Status checking of the LSU
lsu_mode	Provides the mode of the LSU (Read/Write)
ram_start_idx	Provides the starting RAM block for access
ram_block_factor	Provides the block length
ram_cyclic_factor	Provides the number of participating RAM blocks
ram_stride	Provides the stride between consecutive accesses of a RAM block
ram_seg_stride	Provides the segment stride between consecutive *len* of elements
ram_offset	Provides the initial offset of the access
len	Provides the DMA len (similar to *dma_len*)
seg_count	Provides the number of segments (similar to *dma_seg_count*)

in groups of four. The LSU addresses the question: when we receive a data stream from the NoC, how would we store it in the on-chip memory blocks? Conversely, how do we construct a data stream to the NoC from the on-chip memory?

Generally, the LSU supports two modes of access patterns: block and cyclic. A block access pattern means the data stream writes to a single memory block sequentially with a fixed stride. On the other hand, a cyclic pattern distributes the data stream to multiple memory blocks. These modes, either standalone or combined, lead to different configurations of handling the external data stream (hereafter we only discuss the write direction; the read direction works similarly):

- *block* only: this pattern enables a data stream to be fully written to a single memory block with a specified length.

- *cyclic* only: this pattern allows a data stream to be written to a specified number of memory blocks in a cyclical fashion with a specified length.

- *block* and *cyclic*: this pattern allows a data stream to be written to a single memory block for a specified length. When that length of data elements is reached, the next memory block will be written with the same amount of elements.

Figure 3.11 illustrates the LSU block design, and Table 3.2 shows the list of control registers and their descriptions.

Note that if multiple RAM blocks are being used (when cyclic mode is enabled), we only specify the index of the starting RAM block through *ram_start_idx*. The LSU increments the RAM block index by one when servicing the next block. This simplifies the LSU design.

Since the total amount of data elements exchanged between the on-chip RAMs and the NoC are the same, the DMA's *len* and *seg_count* values are passed to the LSU as well.
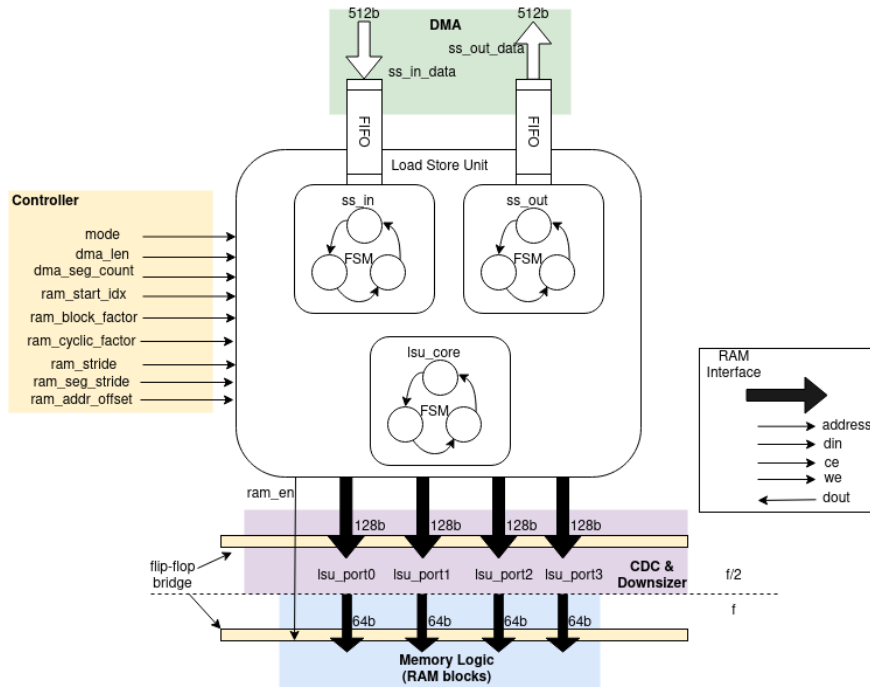
Figure 3.10: LSU Block Diagram
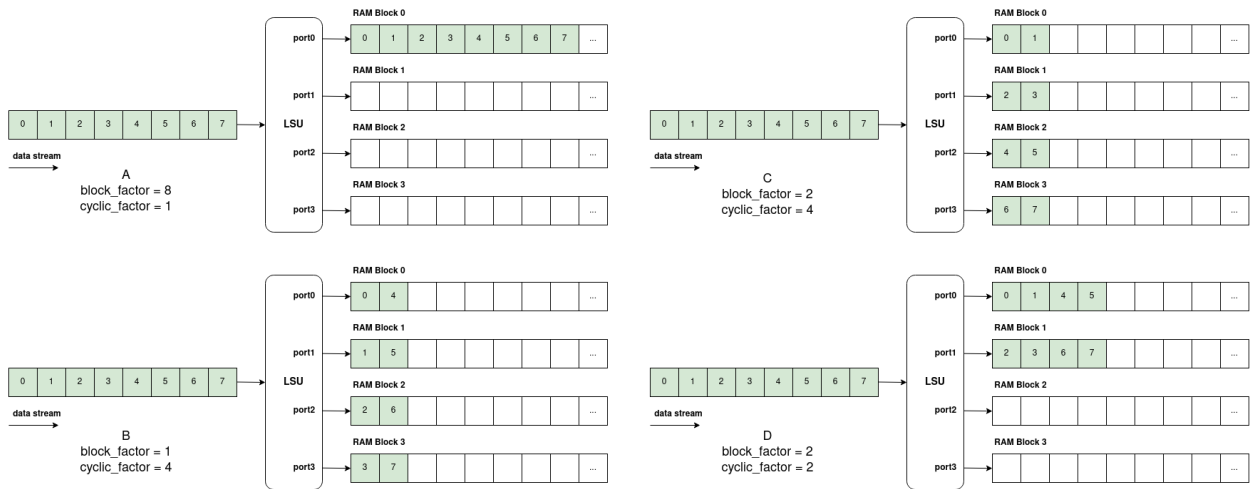
Figure 3.11: Several examples of different (block, cyclic) settings

Similar to the DMA registers, the LSU registers shape the memory layout of the internal memory space for the transfer.

In addition, the three access patterns (block, cyclic, block + cyclic), as described earlier, are implicitly derived from the values of ram_block_factor, ram_cyclic_factor, len, and seg_count.

- If $ram\_block\_factor > 1$ and $ram\_cyclic\_factor = 1$, the LSU operates in block mode. ram_block_factor should match $len \times seg\_count$.

- If $ram\_block\_factor = 1$ and $ram\_cyclic\_factor > 1$, the LSU operates in cyclic mode until the end of the data stream (i.e., $len \times seg\_count$).

- If $ram\_block\_factor > 1$ and $ram\_cyclic\_factor > 1$, the LSU operates in hybrid mode. The data stream runs for ram_block_factor on one RAM block before moving to the next RAM block. This continues until the end of the data stream (i.e., $len \times seg\_count$).

Note that $ram\_block\_factor \times ram\_cyclic\_factor$ does not necessarily equals to $len \times seg\_count$, since cyclic mode, if enabled, implies wrapping the RAM index around.

To improve the data stream access latency, the LSU employs four RAM interfaces. An LSU's RAM interface may connect to one or many RAM blocks. The connectivity specification between an LSU and groups of RAM blocks is supplied in a user configuration. If cyclic mode is activated, multiple RAM interfaces operate concurrently to read/write the RAM blocks. For example, if the cyclic factor is 2, two LSU's RAM interfaces will access the connected RAM blocks in the same cycle. Another performance optimization is the datawidth of each LSU's RAM interface doubles the width of a RAM block (128-bit vs. 64-bit), and as we shall see in the later section, the RAM blocks run twice as fast as the LSU. Therefore, a LSU's RAM port could read/write two 64-bit data elements from/to a RAM block in one cycle. To understand the effectiveness of these optimizations, imagine we would like to write one data beat (512-bit) of a data stream from the NoC to 4 RAM blocks (64-bit). If the block factor is 2, and the cyclic factor is 4, the LSU would essentially complete the writing in one cycle. If cyclic mode is not enabled, this would under-utilize the LSU bandwidth, thus leading to poorer memory access performance, although the double-word optimization could still be used. Figure 3.11 demonstrates some examples of an LSU processing an incoming data stream under different block and cyclic factors.

The LSU also supports dual-ported mode. This mode is only possible for BRAM-type RAM block, since BRAMs support true dual-ported access. Additionally, it requires a RAM block to be accessed by two LSU's RAM ports. It is useful in some cases in which we want to speed up the memory access when the cyclic factor is lesser than the 4 (i.e., not fully utilizing all available LSU's RAM ports). As an example, if the block factor is 1, and the cyclic factor is 2, we could write 512-bit data beats in 2 cycles (again, assuming the RAM block's datatype is 64-bit) instead of 4 if the LSU enables the dual-ported mode. In one cycle, each LSU's RAM port updates two RAM blocks, thus four 64-bit words could be written per cycle.

Another useful mode is broadcast whereby each word of a data stream will be broadcast, or copied to multiple RAM blocks. This is useful in some scenarios where an application requires access to identical data; the broadcast mode eliminates redundant data transfer.

Finally, we note that the incoming data stream to the LSU could originate from different sources (similarly to the outgoing data stream). It could be either from a DMA read response via an NMU. The other case is a write request by an NSU through the Slave AXI adapter. For instance, one socket could write data to another socket's on-chip memory space.

3.3.1.4 Controller

To program the socket, we employ a tiny softcore controller. The soft core is a 32-bit integer in-order RISC-V core (supporting RV32I instruction set) cost around 1000 LUTs. It is the brain of a socket. The softcore allows users to write single-threaded C code to facilitate the operations of the socket. It provides more productivity than designing a hardwired finite state machine for control logic. Besides, users can modify the code quickly to attempt different control schemes without re-implementing the design. This is owing to the C compiler toolchain (RISC-V GCC) that we rely on to generate binary code from the C programs. We refer to the C programs for controlling the socket as controller software. A controller software typically encompasses the following tasks:

- Setting up the DMA and LSU control registers.

- Scheduling data transfers (between the NoC and the DMA engines).

- Setting up the custom logic control and scalar registers.

- Scheduling custom logic executions.

- Synchronizing with other sockets.

- Notifying the socket manager on completion.

The components of the socket are registered in a memory-mapped IO (MMIO) register bank. These MMIO registers are accessible to the softcore's MMIO interface, and they are reserved an MMIO memory space (besides the Instructional and Data memory space). Therefore, a controller software could read or write to these registers via their addresses. The MMIO registers should not be confused with the memory-mapped registers from the Slave AXI adapter. The former is managed by the socket's softcore, while the latter is driven by an external client (e.g., another socket). The MMIO registers cover several functional components, such as Control MAXI, DMA engines, and LSUs as we have discussed earlier. There are several registers used for synchronization with other sockets. These registers link with the memory-mapped registers of the Slave AXI adapter. For instance, the controller software keeps polling on a synchronizing register expecting a specific value until the socket
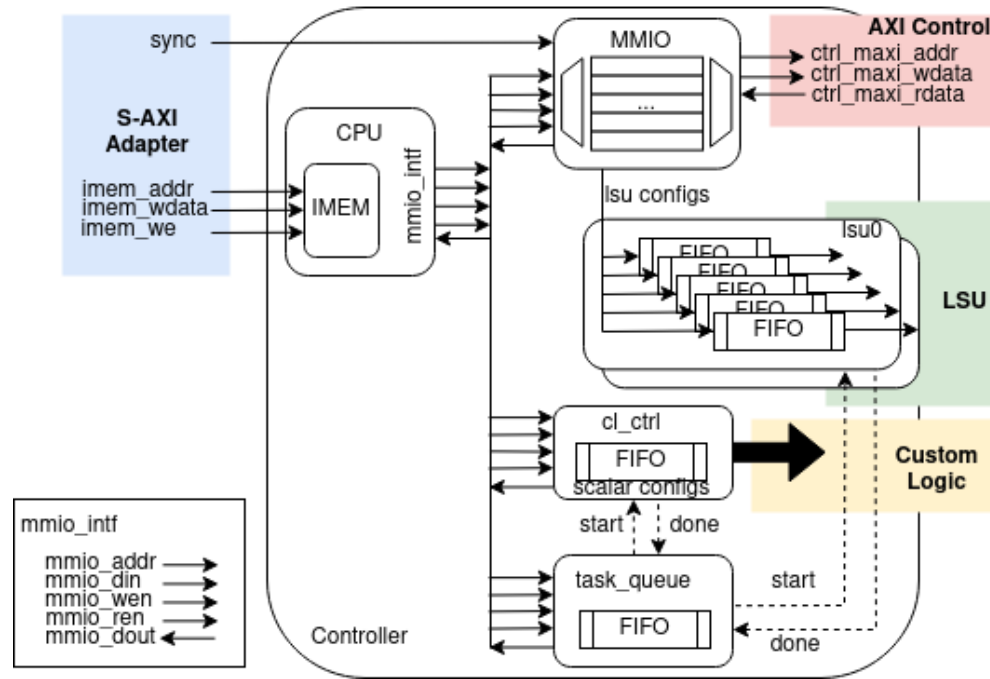
Figure 3.12: Controller Block diagram

that it wants to sync with sends an AXI transaction to write the specified value to the address of that register. Figure 3.12 shows the block diagram of the controller.

The scheduling of data transfers and custom logic executions is primarily concerned with the control flow of the mapping application (e.g., looping constructs). Therefore, to tailor an application to our socket model, one needs to identify and extract the memory transfer parts and the compute parts (hardware) from the control part (software). The hardware/software co-design approach is vital in our model. In theory, since the control part of an application is delegated to the softcore's controller software, we could explore different scheduling options in the softcore to tune the performance. This would have been less productive if the controller was made "hard" (designed and implemented on a case-by-case basis).

This raises the question of whether it is efficient to use a softcore for control. Certainly, there is a large latency overhead of using a softcore in general instead of custom control and datapath. The overhead comes from the many steps required to execute a CPU instruction: instruction fetch, decode, execute, and write back. Therefore, we only use the softcore for housekeeping tasks that do not invoke frequent executions (e.g., hot loops), such as setting up MMIO registers or checking their values. Still, that does not eliminate most overhead. Assume we want to schedule the custom logic to run after a data transfer. The controller software would do the following tasks in order:

1. Setting up DMA and LSU registers

2. Starting the DMA engine to begin data transfer

3. Waiting for the DMA engine to complete

4. Setting up the custom logic scalar registers

5. Starting the custom logic execution

6. Waiting for the custom logic to complete

It takes a few cycles to read an MMIO register in the controller software. Therefore, Step (3) and (6) are particularly costly. It is even more expensive in the case of checking custom logic status since we have to retrieve the CSR (control and status register) value from the Socket CL region to the Socket CC region, as this path involves many pipelined registers. To mitigate the control overhead, we employ a task queue (TQ). It employs a FIFO that enqueues task descriptions (start/done) as they are generated by the controller software. The TQ's control logic will be responsible for the actual scheduling of the DMA engines and the custom logic, not the softcore. Thus, when a DMA completes its run, the TQ dequeues the subsequent task to execute, i.e., starting the custom logic. One example of a sequence of executions orchestrated by TQ is illustrated in Figure 3.13. The controller software does not need to be aware of when a task begins or finishes; its responsibility is adding tasks to TQ. When the TQ is full, it will assert the stall signal to halt the softcore until the current task is completed and dequeues the TQ.

In a sense, the softcore runs asynchronously with the other components of the socket and that would help to overlap the control overhead with the useful work by the DMA engines or the custom logic. The state of the controller is typically ahead of the actual progress of the DMAs or custom logic, so one should not assume a cycle-to-cycle correspondence between the controller software and other components. We also need to provide some buffering to store the registers' values to ensure that by the time a task is scheduled to run, it will get the correct register configuration as indicated in the controller software. If the configuration buffers are full, the softcore will also stall its run in order not to lose any data.

3.3.2 Socket Custom Logic

Socket CL implements the computation and memory requirements of an application. Broadly, it consists of two components: compute logic (or custom logic) and memory logic. The memory logic comprises multiple memory blocks implemented by the on-chip RAM resources on the PL, such as BRAMs, URAMs, LUTRAMs. We refer them as RAM blocks. The datawidth of a RAM block is 64-bit, and its depth depends on the RAM type as follows.

- For BRAM-type RAM block, the memory depth is 1024.

- For URAM-type RAM block, the memory depth is 4096.
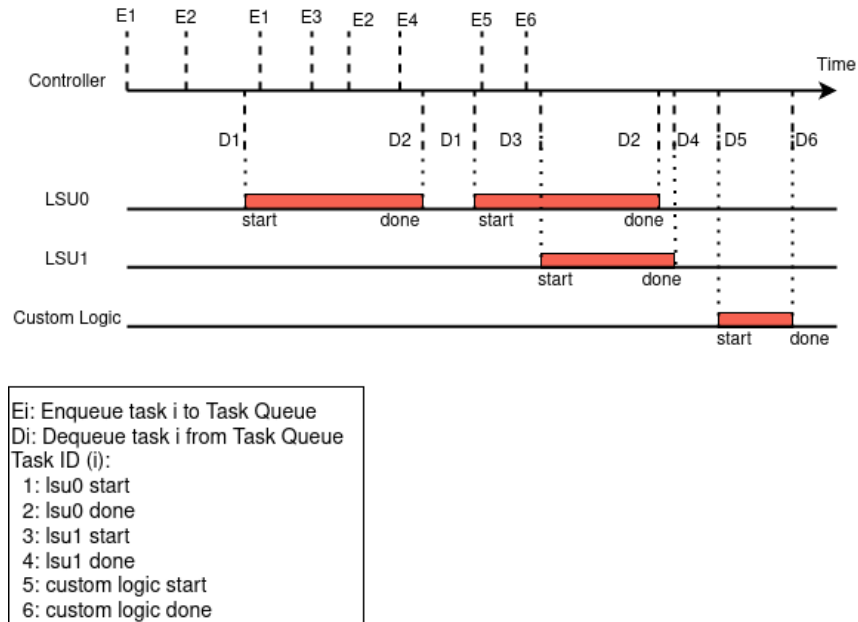
Figure 3.13: Task Queue timing diagram

- For LUTRAM-type RAM block, the memory depth is configurable.

A BRAM capacity is 36Kb. Hence, one BRAM-type RAM block of 1024 x 64-bit requires 2 BRAMs. On the other hand, a URAM is 288Kb, so one URAM-type RAM block of 4096 x 64-bit consumes 1 URAM. We attempt to configure the memory logic as close to the on-chip RAM granularity. A different approach would be allocating a dedicated number of on-chip RAMs as one RAM block of the memory logic. However, it would lead to poorer utilization of on-chip RAM resources. Typically, for most applications to achieve high performance, they require multiple memory banks for parallel accesses. In most cases, the number of available memory ports to access in a cycle is a limiting factor to performance, not the number of DSP slices or multipliers. Therefore, the storage requirement per bank could be quite shallow. Since the total memory capacity of the memory logic is constant, a large RAM block would reduce the number of memory banks available for the compute logic. Therefore, we opt for the on-chip RAM granularity as the size of a RAM block to increase the number of RAM blocks available for the compute logic. At the same time, this design decision would lead to a more complex interconnect demand between the LSUs and the memory logic (remember that the LSUs need access to the memory logic to read/write data to the RAM blocks). To address this issue, we divide the RAM blocks into groups of four RAM blocks. We refer to a group as *RAM group*. The number of RAM blocks per group is a design decision. We think four will suffice in most cases. This also matches the number of RAM ports in an LSU. When we specify that an LSU connects to RAM group rg_2, it implies that port i will connect to
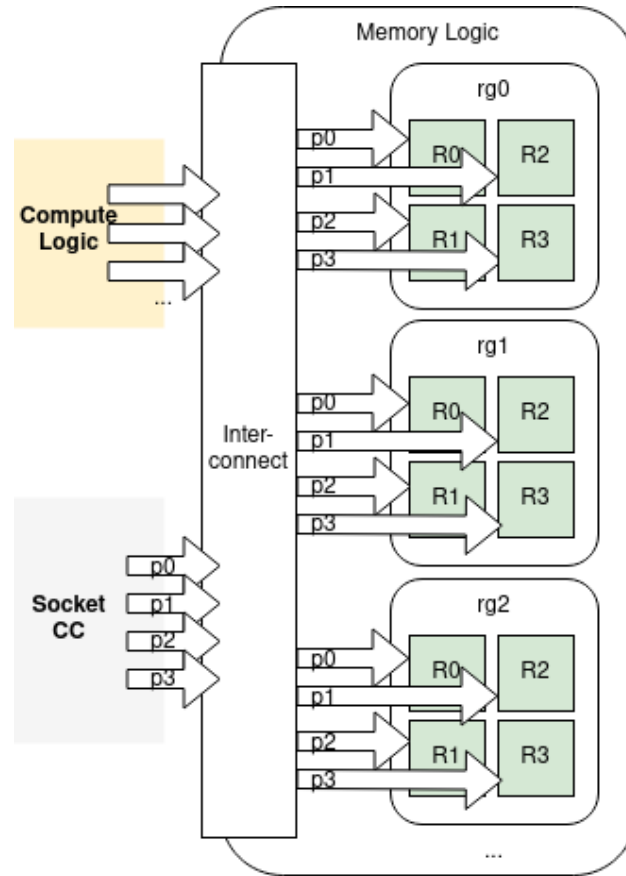
Figure 3.14: Memory Logic organization

RAM block i of rg_2 ($i = 0, 1, 2, 3$), unless the dual-ported mode is enabled in which port i and $(i + 2)\%2$ will both connect to RAM block i as well as RAM block $(i + 2)\%2$. The two-level hierarchy of the RAM blocks alleviates the interconnect complexity between the LSUs and the memory logic, albeit with lesser flexibility. Figure 3.14 shows the organization of the memory logic as well as its role related to the compute logic and Socket CC.

To achieve high performance, the memory logic should be co-designed with the compute logic. Chiefly, the memory logic needs to provide sufficient bandwidth (number of parallel ports) to the compute logic to achieve high throughput and low latency execution.

One could rely on a High-level Synthesis (HLS) flow to design the compute and memory logic altogether from a single high-level specification. Doing so leads to several difficulties. First, the on-chip buffers do not get exposed to external components, such as the LSUs. Second, it is less flexible in how we would want to select the places to insert pipelined registers for timing optimization. Thus, we choose to design them separately. We generate the memory logic via a template-based generator in which we configure the number of

RAM groups, their connectivity to the compute logic and the LSUs, and their RAM types (BRAM/URAM/LUTRAM).

With the memory logic offloaded to a different module, the compute logic consists of largely of computational datapaths. The read/write latency between the compute logic and the RAM blocks of the memory logic is 3 cycles. This is because we add pipelined registers to both the input pins (address, din, we, ce) and the output pin (dout) of each RAM block, and coupled with the fact that a RAM block is synchronous read - synchronous write, their access latency is 3. The latency is made known to the compute logic to schedule loads and stores to these RAM blocks correctly. For now, we do not use any latency-insensitive interfaces between the compute logic and memory logic. In some cases, the compute logic may instantiate some internal buffers whose data do not require communication with the external components such as LSUs. For data need to be fetch or write externally, they should be kept in the memory logic's RAM blocks outside the compute logic.

Besides interfacing to the RAM blocks of the memory logic, the compute unit needs to communicate with the controller as well, so that the controller can control the compute unit (starting the execution, checking the status), and set up its scalar parameters as runtime. Since scalar parameters vary from one application to another, we would have to change the interface between the compute logic and the controller had we allocated separate ports per parameter. This implies re-implementing the controller of Socket CC, which further increases compile time and reduces productivity. Instead, we opt to devise a fixed interface between the controller and the compute logic. We utilize a RAM interface whereby the scalar parameters and control registers of the compute logic are set via the controller software. We allocate kernel address space for these parameters. Therefore, if we want to change the compute logic's parameter list, we only need to update the kernel address space in the software. Overall, the communication latency between the components of Socket CC and Socket CL are statically known by design.

3.3.3 Flip-flop Bridge

As stated in the previous section, we use the RAM interfaces for the communication between Socket CC and Socket CL. The access latency between their components is statically known by design.

The RAM interfaces between Socket CC and CL are pipelined to improve timing by a set of registers referred as *Flip-flop bridge*. The registers form pairs of Flip-flops: one from Socket CC's region and the other from Socket CL's region. We refer to Figure 3.15 for the design of FF bridge.

The bridge, along with Socket CC, are pre-implemented and reused in different applications to reduce the compile time. The bridge essentially provides an anchor to the implementation of Socket CL: we only need to route the interface of Socket CL to its side of the bridge. Therefore, Socket CC partition remains invariant to Socket CL, and could be reusable. On the other hand, the bridge adds an extra 2 cycles to the latency between Socket CC and Socket CL. When mapping an application to the socket model, one should
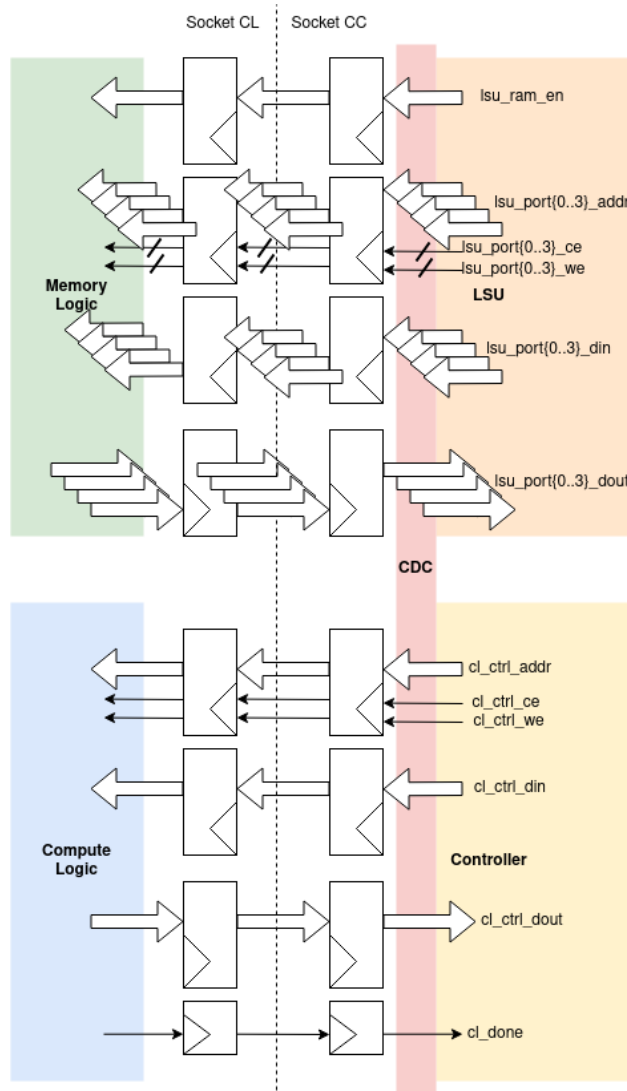
Figure 3.15: Flip-flop bridge connecting the interfaces of Socket CC and Socket CL. CDC stands for "Clock-domain crossing". Socket CC operates at half the speed of Socket CL.

avoid incurring many short yet expensive roundtrips between the partitions, such as frequent status checking. To amortize the cost, the read/write operations should be pipelined. Since the interfaces are designed statically, there is no backpressure between the two partitions. This eliminates the need for checking valid or ready data when designing the compute logic or the controller software. Backpressure would add even more complexity to the already significant delay between Socket CC and Socket CL owing to the FF bridge.

3.3.4   Multi-clock domains

We leverage the multi-clock domain technique for our socket design. In particular, the entire Socket CC region is clocked at half the speed of Socket CL (e.g., 250MHz vs. 500MHz). One of the rationales behind this design decision is that the AXI logic could achieve the full bandwidth of the NoC with 512-bit datawidth at 250MHz, and the wider width AXI bus is capable of packing more data and filling (or reading) multiple on-chip RAMs in parallel. Furthermore, as we have mentioned, it is challenging to meet higher timing target for the communication logic (AXI adapters, DMA engines, LSUs) due to its complexity (in terms of buffering, address calculations, 4KB-limit handling, etc.) The same reasoning applies to the control part, especially the softcore (due to the decoding logic, data forwarding, branch prediction, and arithmetic logic).

In addition, for many data-parallel applications, the control operations (e.g., task sequencing, general address/scalar parameter computations) are typically not on the critical path of the performance (as time spent mostly on data copy or computation). Therefore, it might be possible to slow down the non-critical portion of the design without severely affecting the overall performance. At the same time, the tool could expend more effort on optimizing the critical regions.

Primarily, we have employed two approaches to compensate for the loss of performance due to the clock rate mismatch. First is doubling the datawidth of the communication logic. Secondly, we use a task queue (FIFOs) for overlapping the executions of the DMAs, the custom logic, and the controller. The latter also helps mitigate the control flow overhead induced by the soft CPU core.

Chapter 4

Socket Physical Implementation

In this chapter, we discuss how to implement a socket design to a programmable fabric. Our target PL architecture is the state-of-the-art AMD Versal FPGA. We start with devices that have a single Super Logic Region (SLR), or single-die FPGAs. Our approach would also apply to multi-die FPGAs since the dies' fabrics are usually identical in most devices.

4.1   Socket Floorplanning

Our target device is a single-SLR (Super Logic Region, or single-die) architecture. We floorplan an SLR to sub-fabric regions that have repeatable patterns to implement a socket. The guiding principle is relocatability: a socket implemented on a particular region should be able to be relocated to a different, compatible region without re-implementation. As shown in Figure 4.1, the FPGA floorplan is divided into several clock regions $XiYj$. In both horizontal and vertical directions, we can find individual clock regions, or some combination of them, that are compatible: that is, they have the same fabric structures. In other words, they have similar column patterns. Examples of such regions are $X5Y2$ with $X5Y3$ (the red boxes), and $\{X3Y1, X4Y1\}$ with $\{X5Y1, X6Y1\}$ (the pink boxes). Nonetheless, we do not necessarily constrain our floorplan scheme to the clock regions as long as we can discover some combination of fabric column resources that form compatible regions.

Relocatable netlists bring a paramount benefit in terms of reusing existing implemented sockets, especially at the netlist level because that means we do not have to expend additional compile time. To reduce design effort, vendors usually offer a library of HDL or HLS IPs. But those IPs would need to be synthesized and implemented every time a user would like to use them. In some cases, the IPs could be archived to design checkpoints to preserve the synthesis or implementation details for reuse. Nonetheless, they are still rigid and inflexible, especially if one wants to perform IP integration in which there are some existing implemented netlists in the design that overlaps with the IP's implementation.

A relocatable socket would increase the overall composability of the design thanks to its flexibility. Consequentially, it will improve the overall productivity. However, there are
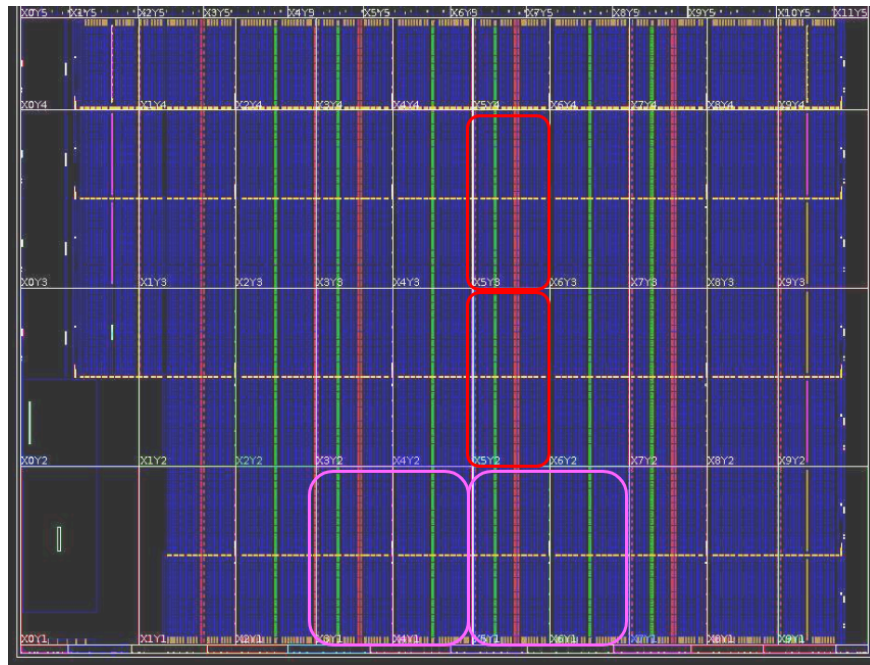
Figure 4.1: The target FPGA device contains multiple clock regions. Some clock regions have similar fabric structures.

several challenges associated with the floorplanning approach.

4.1.1 Routing conflicts

Firstly, there might be some routing conflicts between adjacent sockets. This could happen when there exists some nets from different sockets that share the same routing switches (or Programmable Interconnect Points — PIPs). This could be addressed with the CONTAIN_ROUTING constraint during Vivado implementation. The constraint instructs the router to route all the nets of the socket netlist within the specified floorplan. Therefore, the final implementation result will have both the physical cells and nets be contained within the floorplan.

Nonetheless, we found that, despite the CONTAIN_ROUTING constraint enabled for the socket floorplan, if the column at the edge of the floorplan is a CLB-tile column, Vivado router still expands the routing footprint to one interconnect tile (INT) column next to the CLB column outside the floorplan. This will likely lead to a routing conflict issue if there is an adjacent socket which also utilizes the routing resource of that INT column. Figure 4.2 illustrates one such scenario. We can observe that the routing footprint expands to one INT-tile column to the right side of the floorplan (white rectangular box).

Hence, we work around this problem by leaving a gap of one CLB-tile column between
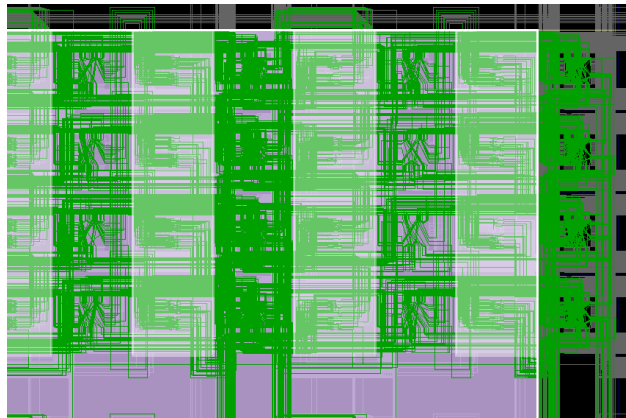
Figure 4.2: Routing expansion issue. The routed nets (in green color) bleed over to the adjacent INT column outside the floorplan (white box).

the socket floorplans. A socket might expand the routing footprint to the extra INT column outside of the floorplan. But now it will not lead to a routing conflict since the adjacent socket does not use the routing resource of that column. The immediate advantage is that we do not have to resolve route conflicts between sockets. The drawback is that we reduce the logic as well as the routing capacity within a socket floorplan. However, in most cases, a socket may not need to utilize 100% logic or routing resources, since doing so would lead to a significant routing congestion issue, and might result in frequency degradation and worsening compile time. [22] reasoned why we do not need to fully utilize the resources available on an FPGA. Therefore, leaving a few unused logic and routing resources is unlikely to negatively impact the design routability or implementability.

4.1.2   Fabric Irregularity

The second challenge is fabric irregularity. Even though we aim to find as many identical regions for socket relocatability, there are incompatible regions on the device that inhibit fabric placement and routing. Real devices often consist of many engineering constraints that do not perfectly mesh well with our socket model and floorplanning strategy. Since we strive to build a working tool flow, these must be fully taken into account regardless. Particularly, the bottom four rows do not contain any logic resource as shown in Figure 4.3.

Therefore, if we want to relocate a socket to the bottom regions, we must make sure that the socket's existing implementation does not contain any cells or nets in the bottom four rows. In other words, we constrain the socket floorplan to omit these rows. This, again, reduces the logic and routing resource of a socket floorplan to compensate for the benefit of increasing the success of relocatability. The tradeoff is worthwhile if we make use of the remaining fabric region effectively. Furthermore, these fabric gaps between socket floorplans
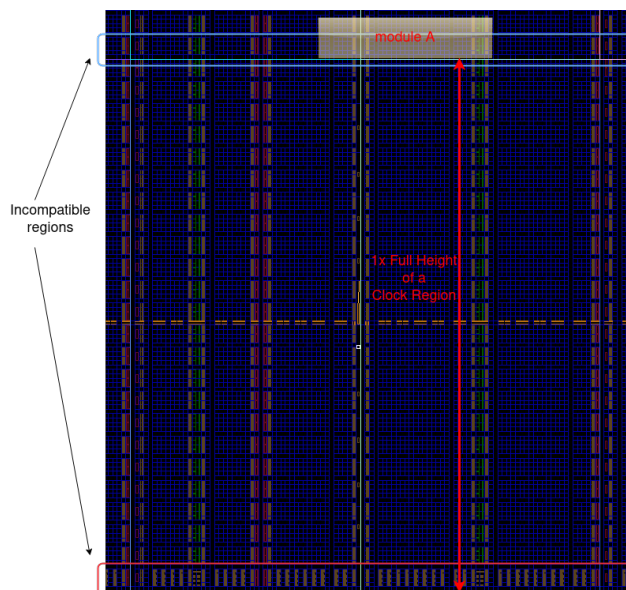
Figure 4.3: Fabric irregularity. The bottom four rows (highlighted by the red oval box) do not have any logic elements. It is incompatible with the region marked by the blue oval box, since the latter does contain logic slices (such as LUTs and FFs). Therefore, *module A* whose implementation contains resources in that region could not be relocated vertically to the clock region below (by a distance of one clock region's height) due to incompatible fabrics.

could be useful in several cases. Since we know that there are no socket nets crossing these gaps, we could utilize them for some "glue" logic or routing. As we will see in the next chapter, we rely on these gaps to perform fast routing of global signals, such as reset, from the shell to every socket.

4.1.3 Floorplan size

The next problem is to determine the size of a socket, and the maximum number of sockets our flow can accommodate. A tiny socket might not have enough resources to implement a custom logic optimally. On the other hand, a bulky socket likely leads to significantly longer compile time. In general, a socket requires at least one NMU for data, and one NSU for control (and optionally data). We refer to such sockets as the base socket (1 NMU + 1 NSU). Given that there are 28 NMUs and NSUs on the target device, our flow could theoretically support up to 28 base sockets of size as small as a half Fabric Sub Region (FSR). Note that not all FSRs are compatible. For example, in a Clock Region (CR), the upper FSR has a different fabric structure from the lower FSR in terms of clocking resource. In practice, a socket should be sized such that the precompiled overlay logic (Socket CC) could fit while

leaving enough area for the custom logic (Socket CL). Another consideration is the number of logic blocks, especially BRAMs/URAMs and DSP slices, available in the socket region. They are usually the deciding factors for achieving high performance. As the NMUs and NSUs are organized in columns, a base socket either has to stretch over two adjacent NoC columns (NMU and NSU on different columns), or run through a single column (NMU and NSU on the same column). In the former case, we could align the base socket to a CR. The advantage of mapping a socket to one CR is that many CRs are compatible, therefore relocatability is guaranteed. However, because a NoC endpoint (NMU/NSU) lies at the edge of a CR, this approach will likely result in routing conflicts because the router will expand the routing footprint to the adjacent CR. In this case, we cannot leave any fabric columns between the CRs as they are NoC columns! Moreover, this scheme would result in the Socket CC's netlist spanning from one side of the CR to another which subsequently makes the integration with Socket CL later more challenging. Therefore, we adopt the option of having a single NoC column in the base socket implementation to alleviate the route conflict issue.

Barring one NoC column utilized by the Shell, we could instantiate as many as 9 sockets of size x2 the base socket (socket_m) with this scheme, plus additional 3 base sockets (socket_s) shown in Figure 4.4. We leverage these as 1x1 building blocks to construct a variety of different socket floorplan shapes and sizes as needed.

4.2   Socket Implementation

As discussed in the previous chapter, Socket CC's responsibilities are communication (getting/sending data over the NoC) and control (orchestrating the configurations and executions of socket components). Its functionality remains similar across different applications. Therefore, we aim to pre-compile Socket CC to a fixed and reusable implementation. For a more effective integration with Socket CL, we divide the socket floorplan into two non-overlapping partitions, one for Socket CC and the other Socket CL. Since we use an NMU and NSU on the same NoC column, this would enable us to implement Socket CC compactly to one side of the socket floorplan and leave the other side to the custom logic as demonstrated in Figure 4.5. Next, we are concerned with how much fabric resources to offer Socket CC's partition. Chiefly, we aim to avoid allocating dense on-chip RAM blocks and DSP slices in Socket CC's floorplan, because those resources would be better utilized by the compute and memory logic in Socket CL. At the same time, we circumvent the usage of large memory buffers and multipliers in the design of Socket CC to ensure that the Synthesis tool does not infer any dense RAM blocks or DSPs. Therefore, we select the floorplan column range such that it does not contain any BRAM/URAM/DSP columns.

Next is the Socket CC-CL interface: the Flip-flop bridge consisting of pairs of FFs from Socket CC and Socket CL regions. We also produce a fixed implementation of the FF bridge. The bridge plays an essential role in stitching Socket CC and Socket CL. It provides a shim to dock the interfaces of Socket CL. During the implementation phase of Socket CL, the only
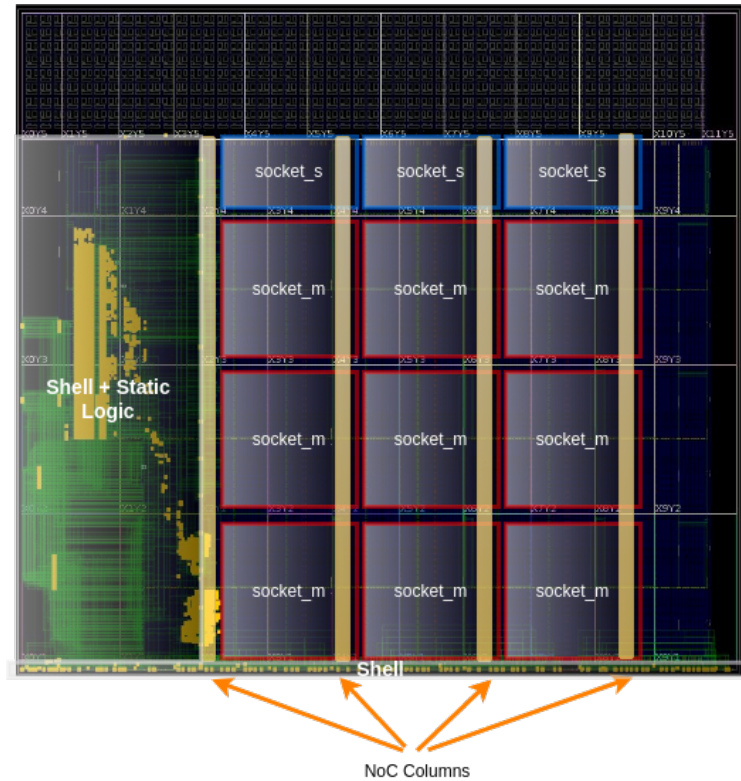
Figure 4.4: SPADES floorplaning scheme. *socket_m* denotes medium-sized socket, whereas *socket_s* denotes small-sized socket. The leftmost NoC column is already used by the shell and static logic. *socket_s* is roughly half the size of *socket_m*.

information of Socket CC available to Socket CC is the bridge; the internal detail of Socket CC is not of interest to Socket CL. There are several ways to build the bridge. In practice, we want the paired FFs to be close to reduce the net delay, so we place them near the joint boundary of the two partitions. The placements of these FF cells influence the QoRs of Socket CC and Socket CL as well as their compile times. For example, if the cells are packed too tightly, the router might run out of routing resources to find paths to/from them, thus leading to a congestion problem and worsening the compile time. In contrast, if we spread the cells too far apart, the QoR could be degraded due to long routing nets. The same thing could apply to cells too close to the joint boundary, as the signals required to route to them could originate from some very distant cells (e.g., AXI signals from the NoC column). We experiment with different strategies for placing the FF bridge and evaluate them in terms of routability (directly impacting the compile time) and QoRs of both Socket CC and Socket CL. Certainly, different Socket CLs will influence the results in different ways based on their routing demand, and there is no one-size-fits-all approach. Nevertheless, we find
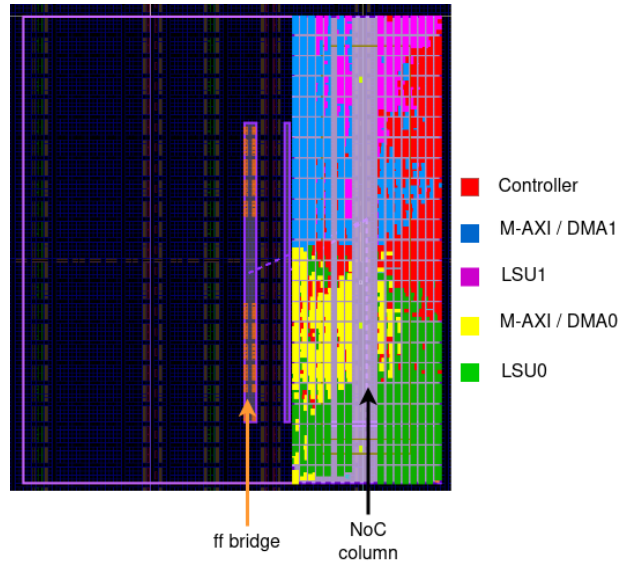
Figure 4.5: Socket CC pre-implementation. There are two NMUs and one NSU in this implementation. The remaining logic of the PBlock is reserved for Socket CL netlist.

the following placement approach works in most cases (taking into consideration different socket sizes).

- On Socket CL's side, the FFs align to a single column. The relevant FFs are placed near each other (for example, the set of FFs connected to an LSU interface, or the set of FFs connected to the controller interface) (1)

- On Socket CC's side, the FFs' placement is unconstrained.

The first technique increases the likelihood of reducing net delays if the FFs are close to the implemented logic. In addition, the FFs are organized neatly in a column so that they do not consume substantial routing resource reserved for the custom logic. Regarding the second technique, since the area in the Socket CC region is limited, we do not constrain the placement of the FFs, as that would increase the routing congestion, especially in the case of socket_s. Another reason is that because Socket CC and the FF bridge are implemented in the same flow, there is a better chance to meet target timing for Socket CC if the bridge FFs (on its side) are placed by the tool alongside Socket CC's cells. We will discuss the Socket CC implementation flow in more detail in the next chapter.

Table 4.1 shows the area consumption of Socket CC. The first column lists the resource types. The second column refers to the numbers of each resource type available within *socket_m* floorplan. In this implementation, Socket CC uses two NMUs and one NSU. There is no BRAM/DSP/DSP consumed by Socket CC. From Figure 4.5, we can observe that

Table 4.1: Socket Area

Resource	socket_m	Socket CC util.
LUT	51,520	26.1%
FF	103,040	16.6%
BRAM+URAM	69+23	0
DSP	184	0
NMU	2	100%
NSU	2	50%

Socket CC almost fully utilizes the resource in its partition which implies that the floorplan we allocate for Socket CC is adequate to meet its implementation requirement. Furthermore, the implementation can meet the target timing at 250MHz. Therefore, the critical paths of a socket implementation should be in Socket CL whose target timing is 500MHz.

We note that Socket CL is compiled separately, and later combined with Socket CC. The details of this procedure will be provided when we discuss the backend flow. Since Socket CC is pre-compiled and reused, their compile time will not be taken into account when we report the socket compile time.

4.3 Static Logic Region

On modern FPGAs that support partial reconfiguration, there is typically two parts of a design: the base logic and the user logic. The base logic is a fixed, pre-implemented partition, whereas the user logic is reconfigurable. The base logic is usually referred to as the shell of the design. The shell, interacting with the IO resources of a device, offers the system utilities enabling the interaction between the host and the device through the runtime drivers (XRT). Several such utilities include frequency scaling, i.e., updating the operating clock frequency of the PL kernels, copying data between the host and the device, or controlling PL modules. Normally, the shell should be implemented compactly in one or a few regions of the fabric to leave sufficient area for the user logic. The shell-based design method greatly improves user productivity. First, users do not need to implement IO-interfacing logic, such as DRAM controller, or PCIe controller to manage data coming in and out of the device. Users also do not need to be concerned with managing clock generators and buffers. All these functionalities are provided in the shell design. This helps reduce the complexity of user designs. Second, the shell is pre-implemented, so it also shortens the compile time and minimizes the memory footprint of the overall implementation. Finally, it brings down the reconfiguration time, since the shell's bitstream is loaded at device startup. Thus, users only need to program the device with user logic's bitstream; the configured shell partition remains intact.

In this work, we utilize the partial reconfiguration approach. We utilize the shell provided

by the vendor (AMD VCK5000 shell version 2022.2). However, the original shell's implementation is incompatible to our flow, since there are routed nets that cross over the fabric region where we would like to implement our socket logic (referred to as Socket Region). We modified the shell implementation by rerouting those nets to avoid conflict with the Socket Region.

In addition, we designed a small central controller for the sockets in the shell region, namely *socket_manager*. The socket manager appears as a PL module to the host via the runtime drivers and can be controlled in a host program. The socket manager is in charge of dispatching the socket instructional memory generated by a host program to all sockets present in the system at application startup over the NoC, as well as kicking off their executions. The socket manager can also reset the state of the sockets, and reset the clock buffer division cells (for providing clock signals in the socket region). The sockets notify the socket manager upon their completion. Note that there is no fabric routing between the socket manager and the socket engines. The communications are carried out using AXI Memory-mapped transactions via the NoC. The socket manager needs one NMU to send requests to the sockets, and one NSU to receive requests from them.

Chapter 5

The Backend Flow

In this section, we will discuss the backend flow. Starting from a Socket CL design input from the frontend stage, we generate a bitstream of a complete design for configuring the reconfigurable partition (or dynamic region) of the target device. Figure 5.1 characterizes all the steps in the flow. We will go through each step in detail.

5.1 Static Logic Compilation

As shown in the previous section, the static logic comprises of the Shell and the socket manager. The socket manager is in charge of dispatching the socket program binaries (generated by a host program) to all sockets present in the system at application startup over the NoC, as well as controlling and monitoring their status. The functionality of the static logic mainly concerns system management. Therefore, it is a one-time compilation and is reusable in different applications. We use AMD Vivado to implement the static logic and generate a design checkpoint (DCP) for subsequent steps.

We floorplan the design such that no logic cells or nets are crossing the socket region. For our target device, the socket region ranges from tile INT_X29Y4 (lower left) to INT_X112Y331 (upper right). This ensures that a socket adjacent to the static region does not have any placement or routing conflicts. The entire static region owns the leftmost NoC column. Therefore, the NoC column is not accessible to any sockets. The socket region could not expand to this NoC column and further to the left because of the routing footprint between the cells of the static logic and the NMUs/NSUs of the NoC column. Among those, the socket manager uses one NMU and one NSU to communicate with the sockets.

Regarding clock signals, the shell provides two kernel clock signals for user partition logic. We use one of the clocks to drive the socket manager. Note that the shell, or static region, is a multi-clock design. There are other components of the shell design utilizing different clock generators. Our static logic implementation meets 500MHz regarding the kernel clock driving socket manager and its relevant logic.

Since a socket has fast and slow partitions (Socket CL and Socket CC, respectively),
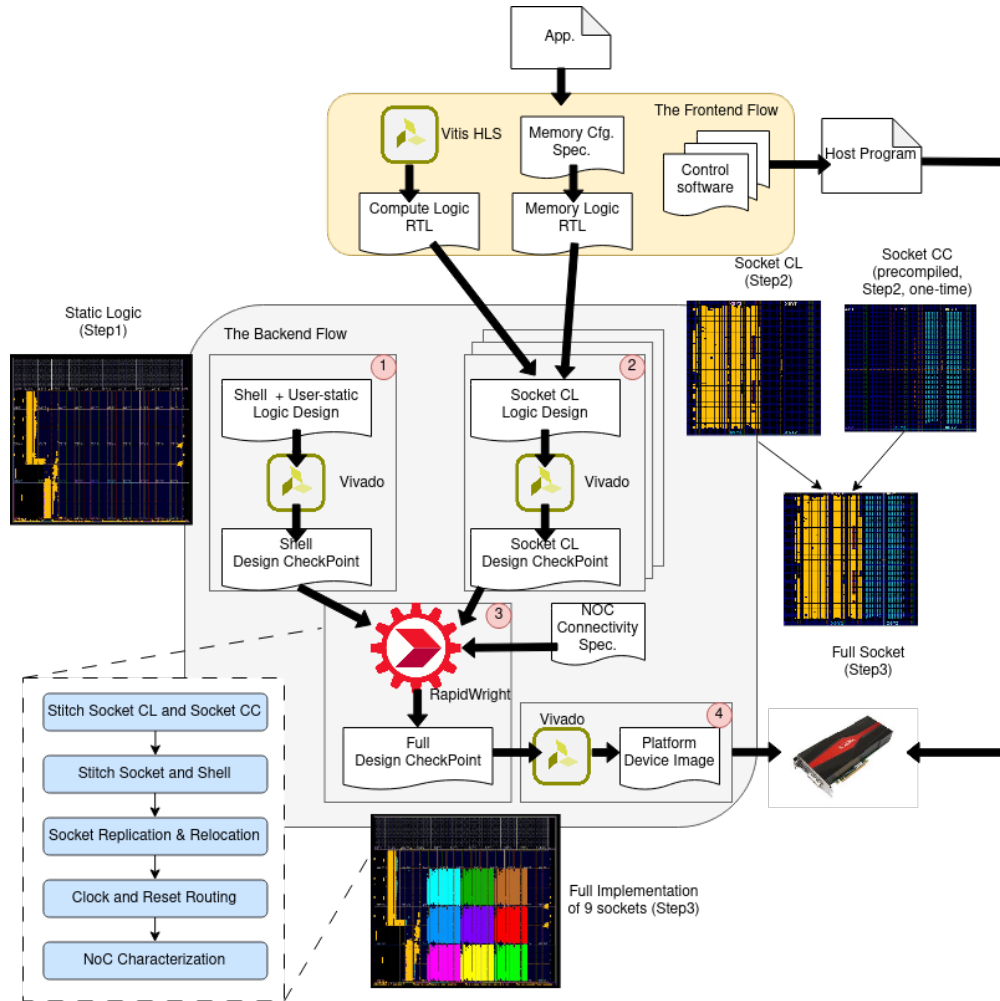
Figure 5.1: The SPADES flow

the static region needs to provide the two clock signals to the corresponding partitions as well. Since there are two user clocks, we could utilize those for the sockets. However, since the signals are coming from different clock generators, this may lead to a substantial clock skewing issue. In addition, we need to handle clock-domain crossing between the two partitions. All in all, this creates a challenging situation to meet target timing. Instead, we opt to use one clock signal to drive the socket region. To generate different frequencies from a single clock, we use the multi-clock buffer (MBUFGCE) cell present in Versal. An MBUFGCE cell takes a single clock signal, f, and generates up to four clocks with different power-of-two division factors: f, f/2, f/4, f/8 for pin O1, O2, O3, and O4, respectively (depending on which configuration is applied to the cell). We use the output pins O1 and O2 to drive the sockets. Note that MBUFGCE is a logical cell; the multiple output pins are

representations that appear at the design level. In actuality, there is only a single output pin from the physical buffer cell corresponding to MBUFGCE. The clock division takes place in a user logic's clock region. In particular, the horizontal clock spine in the clock region contains the clock divide leaf cells, namely BUFDIV_LEAF, since they are the closest to the leaf synchronous cells of the implementation. The input to these cells is the clock signal generated by MBUFGCE. The output is the divided clock driving the FFs of the user logic (or socket). As an example, for cells that require half the clock frequency, the BUFDIV_LEAF driving them will be configured with DIVIDE_FACTOR of 2. Because the clock signal is divided near the leaf cells, clock skew is minimized. In addition, thanks to the fact that the fast and slow clocks for the socket logic originate from the same source, we can reduce the number of FFs needed to handle the clock-domain crossing issue between Socket CC and Socket CL.

The BUFDIV_LEAF cells need to be reset initially so that their DIVIDE_FACTOR can take effect. There is an input reset pin to the MBUFGCE cell. Again, since it is a logical cell, the reality is that the reset signal will need to be routed to all BUFDIV_LEAF cells in the user logic's clock region, not the physical clock buffer cell of MBUFGCE. Thanks to the abstraction introduced by MBUFGCEs, the design specification becomes simpler. Vivado inspects the design specification that makes use of MBUFGCEs and performs the routing of the reset signal accordingly.

We would like to be able to enable and reset these clock buffers at runtime. Therefore, we incorporate a small logic in the socket manager to control the clock enable (CE) and reset pins of the MBUFGCE cells. The control signals are registered in the slave address space of the socket manager.

In addition, we employ many MBUFGCE cells in the design, one for each socket. The reason for that shall be explained when we discuss the Design Assembly step. One important point to keep in mind is that each clock buffer cell owns a distinct clock track (CT). There are 24 CTs in Versal clocking architecture. Some are already owned by existing clock buffers utilized by the shell. Therefore, we add constraints to force the tool to omit those clock buffers from being selected for the socket clocks.

Furthermore, because the clocking resource is located at the bottom of the device, the routing paths between the socket manager to the CE pins of the MBUFGCE cells might be long. We specify those as false paths in our constraints since the timing characteristics of the paths do not concern the functionality of the entire design; they are only used to enable the clock buffers at startup. Therefore, the tool can ignore those paths when performing timing analysis.

The reset signals for the BUFDIV_LEAF cells are not routed when we implement the static logic. This is because we compile the static logic separately, and there is no integration with the socket netlists at this step yet. The actual routing of these signals is carried out in the Design Assembly step.

We use Vivado Partial Reconfiguration flow (DFx) to compile the static logic. The flow partitions the design into two portions: base logic partition (BLP) and user logic partition (ULP). The BLP is provided by the vendor, normally referred as the shell. It instantiates

the required logic to interface with the IOs of the systems. The ULP appears as a black box module of the BLP. The ULP design must conform with the interface signals to the BLP. It is marked as a Reconfigurable module and floorplanned in the dynamic region. This two-stage approach allows users to reconfigure the ULP for different applications without reprogramming the whole device. In this work, the ULP contains the socket manager and one or several socket engines. At this step, we only incorporate the socket manager in the ULP. The socket engines are to be done at later steps. The tool links the ULP design with the BLP design during the *link_design* step and then proceeds to finish placement and routing the combined static logic design. We obtain the DCP of the static logic implementation for full design integration in the later step.

5.2   Socket Logic Compilation

The socket logic contains two components: Socket CC and Socket CL. We will discuss their compilation flows in this section. As they are implemented separately from the static logic, no pre-compiled shell is pre-loaded ahead of the compilation, thereby reducing IO overhead (disk, memory).

5.2.1   Socket CC Compilation

As mentioned before, Socket CC's design is invariant to target applications. Thus, only the fixed portion of the design (i.e., template) is compiled in this flow. It essentially includes the master and slave AXI adapters, one or several DMA engines, one or several LSUs, and the controller. We also incorporate the FF bridge during the compilation. The FFs on Socket CL are fixed, while the FFs on Socket CC are placed and routed by the tool in this step. This helps improve the feasibility of the implementation, especially for socket_s. We instantiate an MBUFGCE cell to drive the clock signal for Socket CC thanks to the multi-clock division functionality. The placement of the clock cell is predetermined: we select one of the clock buffer cells located in a clock buffer tile in the clocking resource at the bottom of the device. Since Socket CC operates at half of the input clock, all the BUFDIV_LEAF cells utilized in its partition have a DIVIDE_FACTOR of 2.

We also need to connect a reset signal to the MBUFGCE cell. We use a dummy LUT with a constant output of 1 to drive the clock reset signal (note that the reset pin is ACTIVE LOW). The LUT is pre-placed by a LOC constraint (location) to the bottom right corner of Socket CC's floorplan. In essence, the LUT is just a stub to guide the tool to perform routing to the BUFDIV_LEAF cells. The LUT itself will be removed from Socket CC in the later step. But the routes are still preserved, and the origin of the nets to BUFDIV_LEAF cells will be replaced by the reset signals coming from the static region. This brings the benefit of not having to do additional fabric routing in the later step. One might wonder what about the routing of the reset signal to the BUFDIV_LEAF cells in Socket CL's region since we only work on Socket CC's floorplan in this step. Could we use the same LUT output

to drive the BUFDIV_LEAF cells in Socket CL's region as well? It turns out that Socket CL is not required to reset these cells at all, since all of its synchronous logic is clocked at the original frequency. Therefore, no clock division is needed, and the BUFDIV_LEAF cells are configured in the pass-through mode which does not need to be reset.

Similarly, the MBUFGCE cell used in this flow will also be replaced by a clock buffer cell from the static region implementation. Thanks to the stub cells, we can reuse almost all the routing result done in this step. This is important to reduce the overall compile time.

One problem that arises with this approach is that the clock signal that feeds to the MBUFGCE cell does not have the exact timing characteristics as the real clock from the static logic, as we compile this design in an Out-of-context mode (IO pins are not connected to any sources or sinks). The real clock is generated by some clock generator (MMCM/PLL). We model the delay of the input clock as similar to that of the real clock from the static logic. We apply latency constraints to mimic the delay from a source clock generator to the input pin of the MBUFGCE cell. This will help Vivado perform a more accurate timing analysis for Socket CC. Ideally, the slack of a particular path in Socket CC should stay similar to the slack as if Socket CC were to be implemented within the context of the static logic.

We then discuss the NoC connectivity of the design. Because Socket CC interfaces with the NoC via one or many NMUs and NSUs, we need to characterize the connectivity of each of these NoC cells with others in the system. But how do we achieve that without knowing prior how many sockets present in the system, and their logical connections, since at this step, the only information available to the flow is Socket CC and its NoC utilization? We rely on the custom backend tool, RapidWright, to generate a NoC specification once we have a complete detail of the entire design. Therefore, in this step, the NoC specification is just a stub and their information (address apertures/bandwidth/latency) is irrelevant to the entire design. We will discard it at the design assembly step. What is important is the placement and routing of the NoC cells (NMUs/NSUs) with the rest of Socket CC's logic.

The next problem we are concerned with is the routing footprint of Socket CC. Except for signals that must originate from outside such as the clock, we would like all nets to be contained in Socket CC's floorplan. However, as we have discussed in the previous chapter regarding the routing expansion issue, the tool will expand the routing footprint of Socket CC to one extra INT column on both left and right boundaries. This could conflict with Socket CL implementation, unless we also leave out one CLB column at the joint boundary with Socket CC. To avoid resolving route conflicts, we adopt this approach. This certainly limits the routing capacity of both Socket CC and Socket CL. In future work, we will explore possible alternatives. During the compilation of Socket CC, we read in the design checkpoint of the FF bridge. Only the FFs on the Socket CL's side are present in the design checkpoint, as they are fixed. The Socket CC's nets to/from these FFs are routed in this step. We carefully craft the constraints such that no nets cross Socket CL's floorplan other than the shared region between FF bridge and Socket CC so that we can preserve most of the routing resources of Socket CL. Particularly, we create an additional floorplan of the shared region and set the CONTAIN_ROUTING constraint on that region. This forces the tool to contain all nets in this region within the specified floorplan. Figure 5.2 shows the floorplan and the
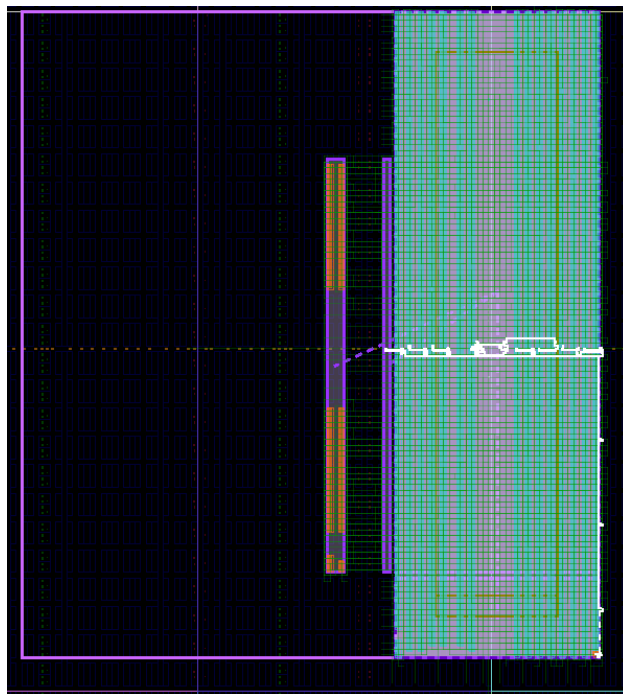
Figure 5.2: Socket CC floorplan. FF Bridge cells are highlighted in orange.

routing result of Socket CC. Note how the routes to and from FF Bridge (highlighted in orange) are contained to the length of FF Bridge. The selected net in white originates from the bottom right corner is for resetting the BUFDIV_LEAF cells located at the horizontal clock distribution of the socket region. The net is driven by a stub LUT cell which will be replaced by a real source later in the flow.

Socket CC meets the timing target of 250MHz. The final output of this step is a design checkpoint of the implementation of Socket CC.

5.2.2   Socket CL Compilation

Socket CL comprises a compute logic and a memory logic. Therefore, only those two get compiled in this step. They are obtained from the frontend flow, and their designs depend on a target application. In addition, different Socket CL designs could be implemented in parallel to reduce the compile time if need be. This applies to Socket CLs whose custom logic is unique, or identical Socket CL designs with different floorplans.

Before compiling Socket CL, we need to obtain the latest FF bridge implementation result from Socket CC's checkpoint. We extract the FF cells of the bridge both from Socket CL and Socket CC, and their routed nets to a checkpoint file, and use that during the compilation of Socket CL. The reason is that the routing footprint of the FF bridge region

overlaps with Socket CL's. Hence, incorporating the latest FF bridge implementation forces the tool to avoid using the existing PIPs of this region. We develop a RapidWright program, namely *FFBridgeExtract*, to perform the extraction. The input to the program is Socket CC's design checkpoint generated in the Socket CC compilation flow, and the output is the design checkpoint containing the cells and nets of the FF bridges.

We apply a similar strategy regarding adding clock delay constraints for Socket CL implementation. This ensures the timing characteristics of the routed nets are as close as if they were implemented with the static logic. The difference is that we do not make use of a MBUFGCE cell in this flow; there is no need for an extra divided clock signal due to Socket CL operating at the original frequency.

The FF bridge's checkpoint is consumed by the tool at the beginning of the *opt_design* step. The tool will honor the existing placement and routing result of the checkpoint during the implementation of Socket CL. Furthermore, we also specify the routing containment constraint to ensure Socket CL's nets do not cross over Socket CC's region.

We set the target timing of Socket CL to 2ns (500 MHz). In practice, whether Socket CL can meet this timing depends largely on the complexity of Socket CL's design. The same goes for the compile time of Socket CL. The higher the resource utilization of Socket CL, the harder it is to achieve routability (implying compile time) and timing closure.

The final output of this step is a design checkpoint of the implementation of Socket CL, or multiple design checkpoints in case of parallel compilation of several different Socket CL designs or implementations. Similar to Socket CC's implementation, the checkpoint must be fully placed and routed as shown in Figure 5.3. The routed nets that cross over to Socket CC's region are those that connect the FF bridge cells together (from one side to the other). These nets were routed during Socket CC compilation and preserved in the extracted FF bridge implementation that we use in this step. Thus, they do not cause routing violations when we stitch Socket CC and Socket CL later.

As mentioned before, one important property of a socket is relocatability. We would like a socket to be location-agnostic for better composability. Therefore, although we set the floorplans of Socket CC and Socket CL to a specific region of the device, we note that this is not the ultimate placement of the socket, but rather a "virtual placement". The socket is relocatable and could be placed in a different compatible socket region. We refer to this as coarse-grained placement since it happens at a greater scale than individual cells. The next step, Design Assembly, will carry out such placement.

5.3   Design Assembly

This step combines the design checkpoints of the static logic and the socket logic to produce a full system implementation. We leverage the open-source backend tool, RapidWright [44] for netlist customization. RapidWright reads a design checkpoint to an internal representation of its classes and objects. The representation captures both the logical design (logical cells and nets) and the physical implementation (physical cells and nets). Users take

ff_bridge
extracted from
Socket CC
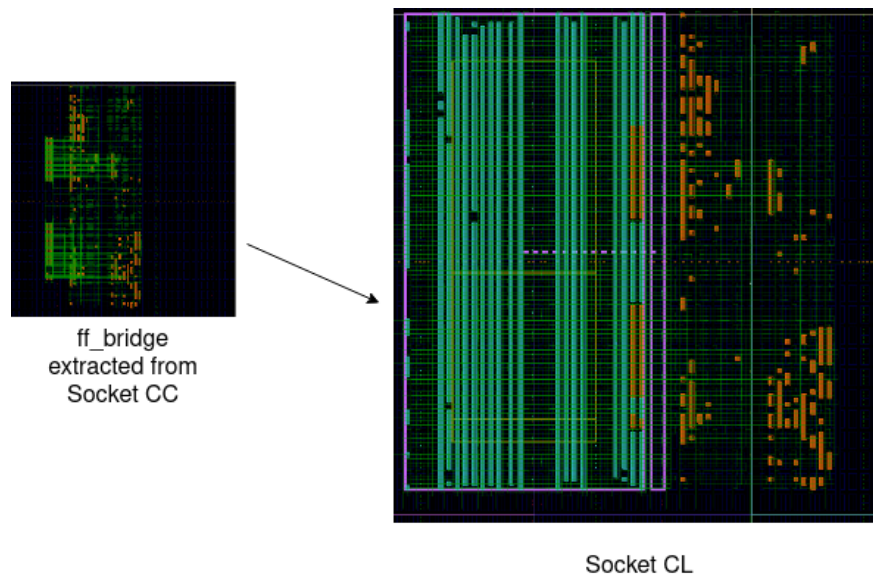
Socket CL

Figure 5.3: Socket CL floorplan. FF bridge cells are highlighted in orange. The cells outside of the floorplan are from Socket CC.

advantage of RapidWright API to manipulate the design in many ways, such as adding new cells, modifying existing P&R of some cells or nets, etc. The important takeaway is that those are done in the context of procedural programming. In other words, we can write a RapidWright program (Java-based) to modify an existing design. This leads to higher productivity than modifying the design in Vivado via TCL commands with equivalent functionality. In some cases, there are a few things that RapidWright can achieve while it is difficult or even impossible to do in Vivado. Most importantly, RapidWright can construct a new design based on one or several existing designs as inputs. To demonstrate this point, let us discuss the Socket Stitching phase of this step in the following subsection.

5.3.1   Socket Stitching

This step combines Socket CC and Socket CL checkpoints into a single socket implementation. It is as if the socket were implemented as a whole. We write a RapidWright program, namely *SocketStitch*, to merge the two checkpoints into one. There are some overlaps between the two partitions, such as the FF bridge's cells and nets, since they are incorporated into both the compile flows of Socket CC and Socket CL. Initially, to handle the overlapping issue, we removed the FFBridge's source and sink cells from Socket CC of Socket CL's checkpoint. Those cells are available in Socket CC's checkpoint. Below are the general steps (not comprehensive) in our RapidWright program in order to guide the combination of the two designs.

1. Read in the design checkpoints of Socket CC and Socket CL (logical netlists in addition to physical implementations).

2. Create a new design. Construct its netlist by copying the logical cells and nets from Socket CC and Socket CL.

3. Create additional ports and wires to connect the two partitions. In particular, we create the LSU logical ports, since they are the interfaces of Socket CC and Socket CL.

4. Unify the clock nets of the two partitions. The clock signal should come from the identical clock buffer location in both partitions.

5. Copy the physical cell placements of Socket CC and Socket CL to the new design. Note that because of the overlap of FF bridge cells, we only copy the cells from each partition.

6. Copy the physical net routings of Socket CC and Socket CL to the new design. In particular, we enumerate all the nets of the original designs and copy their PIPs' usage to the equivalent nets of the new design. Care must be taken when dealing with the overlapping nets from FF bridge between the two designs. We avoid duplicating the copies of the PIPs for those nets.

7. Merge the BEL Attribute databases of Socket CC and Socket CL to the new design. In Versal architecture, there are several primitive physical cells whose configurations cannot be modified by users, but only by the tool. Although they do not appear in the P&R result, they help improve a design's QoR, for example, by adding delays to clock signals to mitigate clock skewing issues. Those are referred as BEL Attributes, and their configurations are packaged in the design checkpoint of an implemented design. Since we would like to preserve the QoRs of Socket CC and Socket CL after the combination, we also need to combine their BEL Attributes.

In addition, the two designs, Socket CC and Socket CL, make use of the same clock buffer. Since we do not need to carry this information, the clock buffer does not get copied to the new design. We also un-route the path from the clock buffer to the clock routing node immediately outside the socket floorplan using algorithm 1. In this algorithm, *OriginClkPIP* in L2 refers to the original clock PIP of the net. It is located at the output pin of the clock buffer. The algorithm starts with this PIP, and then walks forward to the subsequent PIP. When examining a PIP, *isTileValid()* checks whether the PIP's tile lies within the socket floorplan. If yes, it moves forward with the next clock PIP. The algorithm stops when we find a PIP within the socket floorplan. L19-21 removes the outside PIPs from the clock net and renders it detached from the source clock buffer.

Algorithm 1 Clock detachment algorithm

1: $toRemovePIPs \leftarrow \{\}$
2: $clkPIP \leftarrow OriginClkPIP$
3: **while** $isTileValid(clkPIP.getTile())$ is False **do**
4:     $toRemovePIPs.add(clkPIP)$
5:     $enode \leftarrow clkPIP.getEndNode()$
6:     **if** $enode$ is Bidirectional **then**
7:         $enode \leftarrow clkPIP.getStartNode()$
8:     **end if**
9:     **for** $pip : enode.getAllDownhillPIPs()$ **do**
10:         **if** $toRemovePIPs$ contains pip **then**
11:             **continue**
12:         **end if**
13:         **if** $clkNet.getPIPs()$ contains pip **then**
14:             $clkPIP \leftarrow pip$;
15:             **break**
16:         **end if**
17:     **end for**
18: **end while**
19: **for** $pip : toRemovePIPs$ **do**
20:     $clkNet.removePIP(pip)$;
21: **end for**

5.3.2 Socket Assembly

One or several socket checkpoints generated from the previous phase are combined in this step to produce a complete design using our RapidWright program, *SpadesFlow*. The static logic checkpoint is also loaded at this phase. The sockets are assembled following the full design description provided as an input to *SpadesFlow*. The design description, or SPADES configuration, characterizes the details of the sockets present in the full SPADES design. Generally, it includes the socket types, the number of sockets per type, and their NoC connectivity. A socket type is a distinct socket checkpoint. We associate a socket type with an identifier called *designId*. Per socket type, we assign a *socketId* to distinguish between different socket instances of the same type. We specify the placement location for every socket. The placement location is a pair of integers that represent the X and Y offsets, respectively of a socket relative to its original placement guided by the socket floorplan discussed in previous steps. The design specification must ensure that there is no overlap between the sockets. This means socket placements are done manually via the input design specification. Another responsibility of the design specification is the communication between the sockets. It provides the logical connections between one's NMU to another's NSU. *SPADESFlow* ingests all this information to build a complete design.

Concretely, below are the general steps done by this phase.

1. Read input designs.

First, we read the static logic checkpoint and create a new design based on the static logic. We then read and parse the design specification. The flow will retrieve all the required socket checkpoints as instructed by the specs. To avoid excessive IO overhead, we only read a checkpoint once, even if it is to be duplicated (multiple *socketId*s with identical *designId*).

2. Copy the logical netlists of all socket designs to the new design.

The socket cells should appear in the netlist as if they were implemented altogether with the static logic. They should be children cells to *ulp* from static logic. Additionally, we connect the clock and reset signals from the static logic to the sockets by creating logical nets. Each pair of clock and reset comes from an MBUFGCE cell instantiated in the static logic and each socket uses a pair from a unique MBUFGCE cell.

3. Copy the physical cell placements and net routings from the reference socket designs to the new design.

This step also performs netlist relocation. The flow enumerates all socket instances. A socket instance has a unique *socketId*. The placement offsets per socket instance are known to the flow through the design specification. We use RapidWright's relocation tool to perform copying the implementation simultaneously with updating its location to a new region according to the offsets. The algorithm works by first identifying the anchor site of the implementation. The P&R result of the implementation is made relative to this anchor. The new anchor site is determined based on the offsets. The algorithm then updates the remaining netlist accordingly. For the algorithm to execute successfully, all sites of the socket implementation must be movable, and there must be a compatible site for each site of the implementation in the new region. This is another reason why we need to detach a socket checkpoint obtained from the previous step from its clock buffer. One caveat of using RapidWright relocation tool is that the static nets (VCC and GND) are unrouted after relocation. We fix this by reusing the routing results from the reference designs to route these static nets.

4. Characterize NoC traffic solution.

RapidWright provides a NoC API for this purpose. We parse the NoC connectivity information from the design specification to inform the flow the connections between each pair of NMU-NSU. Typically, the NMU and NSU are from different socket instances. The NSU could also be the DDR Memory controller (DDRMC). We build upon the existing NoC solution of the shell present in the static logic. We utilize RapidWright NoC API to create an NMU or NSU instance for each NoC cells were found in the combined design. Next, we create a NoC Connection stipulating a path between an NMU-NSU pair. The path also needs other information such as estimated required bandwidth, latency, communication type, and datawidth. The datawidth is always set to 512-bit since this applies to our socket's AXI bus design. The communication type could either be AXI memory-mapped or AXI stream. The required bandwidth is the bandwidth demanded by this path. For high-volume data traffic (e.g., long bursts), it likely needs maximum bandwidth. For control traffic, the bandwidth

demand is low, since it does not occur very often and the burst length is typically short (e.g., one).

To determine the amount of bandwidth to allocate for each path, we examine the type of the path. If the path is for control communication, we set the required bandwidth low (e.g., 5 MB/s). If the path is for data communication as in the case of socket to/from DDR, we divide the total bandwidth of a DDRMC evenly to the number of connections whose NSU is the DDRMC. For instance, if the DDRMC bandwidth is 25 GB/s, and there are 4 sockets reading data from it. Then the required bandwidth of a NoC path from each socket's NMU to the DDRMC is 6.25 GB/s. Of course, this is with the assumption that the traffic of all NoC paths is active at the same time, so this is the worst-case bandwidth demand. In reality, some traffic paths may happen earlier or later depending on different applications. Note that these are just estimations; the actual bandwidth of a NoC path at runtime might be different. The required NoC bandwidths specified at this step form a set of NoC constraints fed to the Vivado NoC compiler in a later step to conduct the actual NoC routing. The more accurate the NoC constraints depict the traffic, the better the NoC routing solution becomes. As an example, a low-traffic path, such as control, signifies to the NoC compiler that this path could afford a longer route (more switches), and the compiler can prioritize other paths that demand higher bandwidth.

Besides socket-socket and socket-DDRMC communications, we also describe the connections between the socket manager and individual socket instances. As stated in the previous chapter, the socket manager commands the sockets. Hence, their NoC paths are mainly for control. In summary, this step brings about the logical NoC connections between various pairs of NMU-NSU, including socket - socket, socket manager - socket, and socket - DDRMC. We preserve the original NoC solution of the shell. The complete NoC routing of all paths is accomplished by the Vivado NoC compiler in the final step.

5. Route global signals such as clocks and resets.

This step is necessary in order to create a legal implementation result. Technically, this could be done in Vivado, since at this point we already have a valid logical design. We could invoke the Vivado router to assist with routing the clock and reset signals. However, doing so incurs additional overhead to the compile time. Before routing, the Vivado router needs to construct a Route database based on the existing routing result of the design checkpoint. The routing database considers *all* nets in the design. For some large designs, this construction even takes longer to complete than the actual clock and reset routing. Since we have already materialized all routing information in our *SPADESFlow*, and we are only interested in routing specific signals, we could perform the routing of these signals in the RapidWright program instead to avoid the tooling overhead of doing so in Vivado.

As mentioned earlier, we employ multiple clock buffer MBUFGCE cells, each for one socket. If one clock buffer were used for all socket instances, it would lead to a situation in which there existed multiple clock roots, one in each socket region. When routing a clock signal, the tool first determines the clock root; it is essentially the central point of all leaf nodes (synchronous elements driven by the clock). Its role is to balance the delays between itself and the leaf nodes. The clock signal gets routed to the clock root initially and then
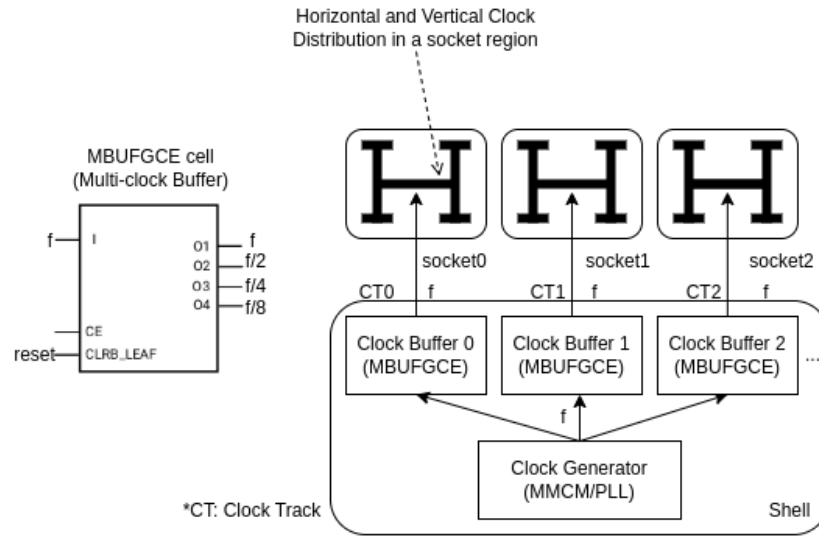
Figure 5.4: Each socket owns a unique Clock track driven by a separate MBUFGCE clock buffer. Note that, although a socket uses two clock domains (f and $f/2$), there requires only a single physical clock net routed from a clock buffer. The frequency division is achieved by the buffer division cells (BUFDIV_LEAF) located in the horizontal clock distribution inside every socket region.

branches out to the leaves. The location of the clock root is important in mitigating the clock skewing issue, since it is affected by the difference between the delays of the clock root to the source node and the sink node of a synchronous path. In order to ensure an accurate timing analysis, the tool mandates that there should be a single clock root per clock signal. As the socket instances are implemented separately and relocated to different regions, their clock roots differ. If we used a single clock buffer, or in other words, a single clock net to drive all socket instances, we need to reroute the clock for the entire design, since their clock roots need to be unified. Therefore, this not only led to removing existing clock routing in every socket and rerouting them which lengthens the compile time, but also changes the timing characteristics of the sockets since their clock trees were not preserved. These issues motivate us to adopt multiple clock buffers which one is owned by a distinct socket instance. This would allow a socket to keep its clock root as well as the clock tree structure obtained from the previous compilation steps. Hence, the critical paths of the socket implementation would stay the same after design assembly. Each clock buffer is situated on a different Clock track (CT). There are 24 CTs in the target device. Excluding those used by the shell clocks, there remain 15 CTs available to at most 15 socket instances in our flow.
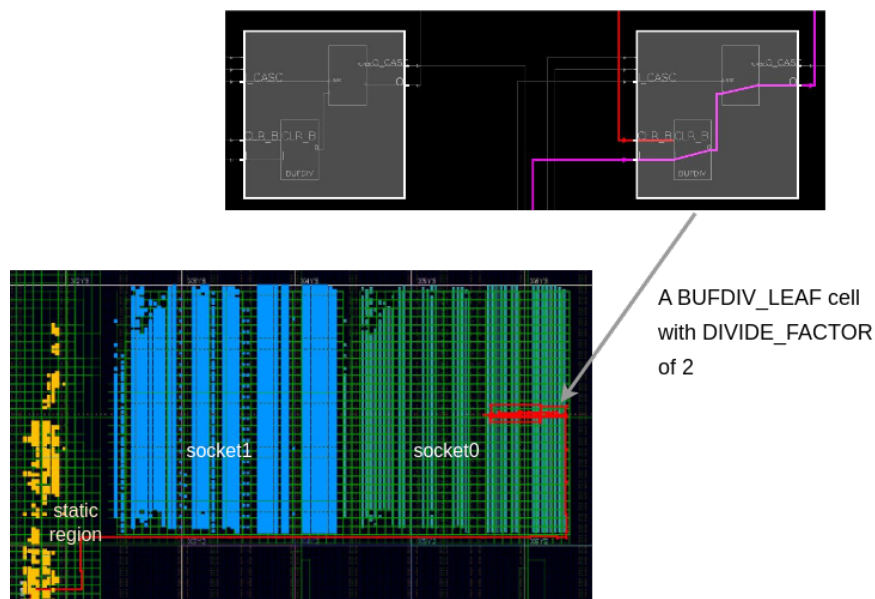
Figure 5.5: Routing of the reset signal (highlighted in red) originated from the static logic region to the reset pins of the BUFDIV_LEAF cells in socket0 region. We exploit the gap beneath the sockets (unused fabric regions) for fast routing.

Algorithm 2 Socket Clock Routing algorithm

1: $workPath \leftarrow \{\}$
2: $workList \leftarrow \{\{sourcePIP\}\}$
3: $curMinLen \leftarrow 0$
4: **while** $workList.isNotEmpty()$ **do**
5:      $currentWork \leftarrow workList.getLast()$
6:      **if** $currentWork.isEmpty()$ **then**
7:          **if** $workList.size() \equiv 1$ **then**
8:              **break**
9:          **end if**
10:         $workList.removeLastList()$
11:         $workPath.removeLastPIP()$
12:         **continue**
13:     **end if**
14:     $pip \leftarrow currentWork.getAndRemoveFirst()$
15:     $workPath.add(pip)$
16:     $enode \leftarrow pip.getEndNode()$
17:     **if** $enode$ is Bidirectional **then**
18:         $enode \leftarrow clkPIP.getStartNode()$

```
19:     end if
20:     if enode is null then
21:         workPath.removeLast()
22:         continue
23:     end if
24:     if enode ≡ sinkNode then
25:         if routePath ≥ workPath.size() then
26:             routePath ← workPath
27:             curMinLen ← workPath.size()
28:         end if
29:         workPath.removeLast()
30:         continue
31:     end if
32:     if workPath.size() ≥ curMinLen then
33:         workPath.removeLast()
34:         continue
35:     end if
36:     if workList.size() ≡ maxThreshold then
37:         workPath.removeLast()
38:         continue
39:     end if
40:     curDist ← getManhattanDistance(enode, sinkNode)
41:     nextPIPs ← {}
42:     for npip : enode.getAllDownhillPIPs() do
43:         if routedWires contains pip.getStartWire() or pip.getEndWire() then
44:             continue
45:         end if
46:         if npip ≡ pip then
47:             continue
48:         end if
49:         nextDist ← getManhattanDistance(npip.getEndNode(), sinkNode)
50:         if nextDist ≥ curDist then
51:             continue
52:         end if
53:         nextPIPs.add(npip)
54:     end for
55:     if nextPIPs.isNotEmpty() then
56:         workList.add(nextPIPs)
57:     else
58:         workPath.removeLast()
59:     end if
60: end while
```

61: **for** $pip : routedPath$ **do**
62:     $clkNet.addPIP(pip)$;
63: **end for**

Algorithm 2 demonstrates the routing of the clock signal per socket instance. We repeat this procedure until all socket instances' clocks are fully routed.

Next, we route the reset signals to the socket instances. The origins of the reset signals are from the socket manager's cells within the static logic implementation. Each reset origin corresponds to one socket instance. Note that the reset operations are intended to reset the BUFDIV_LEAF cells within the socket regions so that their divide factor configuration can take effect, thus ensuring that the implemented design working properly on board. The reset operations are accomplished via a software mechanism; we run a host program to assert the reset signals controlled by the socket manager thanks to the runtime driver.

We route a controlled reset cell of the socket manager to the BUFDIV_LEAF cells within the corresponding socket region. As a reminder, the routes to those BUFDIV_LEAF cells were taken care of by the stub LUT cell in the Socket CC partition of a socket instance. Therefore, this step entails removing that stub cell while keeping its routed nets. By using the stub cell to anchor the reset routing initially, we only need to perform routing from the reset origin of the socket manager to the end node of the PIP connected to the LUT output pin. Furthermore, we exploit the unused fabric region ("gaps") beneath the socket to build the routing path, since we know for certain that there are no other nets utilizing the fabric routing resource of those gaps thanks to the floorplan strategy and compilations from previous steps. This significantly simplifies the routing problem, and we could build a fast router to accomplish the task. Similar to the clock routing algorithm, we use depth-first search to route a reset signal from its origin to the relevant socket instance. We enforce extra constraints on the X and Y tile coordinates of every routing node to ensure that it navigates the path through the gaps, and does not violate any socket regions. We use the Manhattan distance as the cost function for the router. We repeat the algorithm to cover all the reset routing. One may ask whether these long routing paths could affect the QoR of the entire design since we do not make use of any FFs to pipeline these reset paths. It appears that these are false paths. Their role is resetting the BUFDIV_LEAF cells in the socket regions, and they do not impact the performance of the design in any way. Therefore, it is not important to enact timing optimizations on these paths.

6. Update the design's BEL Attribute database.

We incorporate the BEL Attributes from all reference socket designs with the static logic's BEL Attributes to the new design. This ensures that all timing optimizations of the reference designs are carried over to the final design.

5.4   Bitstream Generation

The full design checkpoint is imported to Vivado for NoC routing. The Vivado NoC compiler assesses the NoC constraints specified in the previous step to generate a NoC

solution that meets the connectivity and bandwidth demands. We also generate the timing and area reports in this step to evaluate the QoR of the implementation. The final step is generating a platform device image (PDI) containing the implementation of the design. The PDI shall be packaged with necessary metadata files, such as operating clock frequency, to produce a bitstream ready for programming the target device.

Chapter 6

The Frontend Flow

6.1   Design Principles

In this work, we target C/C++-based HLS applications. The applications are designed based on the HLS model. The model pertains to the following properties:

- An application consists of one or several modules. The top-level module employs a bus interface (such as AXI) to communicate with external entities (such as other modules, or the memory subsystems). Bus read and write operations typically consume tens of cycles depending on the external clients.

- Local, internal arrays are mapped to on-chip RAMs for fast access. The access latency is statically defined and may vary for different arrays. The arrays may be partitioned into smaller arrays to increase the number of memory ports available in a cycle.

- Code that moves data coming in and out of the top-level bus interface to the local arrays.

- Code that performs computation directly on the local arrays which usually contain one or several loop nests. To improve the throughput of execution, the loops can be unrolled or pipelined.

To achieve high performance, there are a few important considerations. First, one should avoid redundant external read/write operations, since they incur substantially high latency. Therefore, one should aim to maximize the data reuse opportunity by copying data to local arrays, doing work, and writing the result back to the external memory. Since the model does not adopt caches, it is the responsibility of the designers to manage data communication on and off-chip. Second, one should avoid using a monolithic local array. Instead, partitioning the array into smaller buffers will increase the number of memory ports available for parallel accesses, thus leading to higher performance. However, array partitioning may also cause a QoR degradation if the partitioning result causes the use of huge multiplexer trees, especially

if the partitioning value is big and/or the partitioning mode does not fit well with the access pattern. Another key factor in the performance is the overlapping of data movements and computation. This often could be accomplished by the double-buffering technique in which the compute logic works on one buffer, while the data mover accesses another buffer concurrently. Then, on the next iteration, they swap the buffers' accesses.

From these practices, we can observe that there are three primary "actors" with distinct roles that are influential to the performance of an application: the on-chip memory structure, the compute datapath, and the scheduler (or control). The scheduler is in charge of moving data between external memory space and the on-chip memory, as well as coordinating such movements with the computation tasks. Generally, an HLS design threads these actors together in a single source. We argue that doing so would lead to sub-optimal circuits in some cases. For instance, the data copy logic may be duplicated in several places, hence a waste of resources. Complex control sequences in an HLS design may generate huge state machine transitions with high fanout signals (such as clock enables). Often the critical paths of an HLS design are not in the compute datapaths, but rather the control FSMs, or data movement logic. By isolating the control and memory structure, we let the HLS tool concentrate solely on optimizing the compute logic to meet the target clock constraint. Another benefit of having a separate controller is that we can explore different design parameters at *runtime*, i.e., design space exploration, by just changing the control code instead of redesigning the control FSM circuit as in the case of full HLS design. This could be useful in cases we want to experiment with different block sizes for loop tiling at runtime when there is a certain number of active sockets competing for the external memory bandwidth.

In summary, to ensure compatible mapping to our SPADES mode, we need to extract the control, memory, and compute logic out of an input HLS design. For now, let us focus on the mapping issue to a single socket in the SPADES model. Recall that a socket consists of two parts: Socket CC and Socket CL. Socket CC's design and implementation is fixed; the only variance is the software running on the controller of Socket CC. On the other hand, Socket CL must be tailored to the application. Firstly, we identify and extract the data movement code from the input design, since it will run on the control unit. Next, we examine the on-chip memory structure of the design to determine how many RAM groups we would need for the memory unit. Recall that one RAM group holds four RAM blocks. The hierarchical memory design helps ease the interconnect complexity. It is also important to pay attention to the available memory ports that the memory unit can supply to the compute unit. For example, if the compute unit requires 8 parallel memory accesses per cycle, this means we need to allocate at least 8 RAM blocks (or two RAM groups). Additionally, if the compute unit desires for double-buffering, there should be at least 16 RAM blocks among which 8 are accessed per iteration. Therefore, the memory logic design also constitutes the design of compute logic's RAM interface. The next step concerns the connectivity of the RAM groups with the control unit, in particular the LSUs. There may be one or several LSUs in the design depending on the number of concurrent AXI buses in the input design. Note that a socket DMA engine can only manage one AXI bus interface at a time, and there is only one LSU attached to it. If the design consists of two concurrent AXI bus interfaces, two DMA

engines must be employed. Therefore, we specify the sets of RAM groups for each LSU that has access to them. Once a RAM group is mapped to an LSU, the LSU can read/write to all four RAM blocks within the group. Therefore, the LSU specification occurs at the group level, whereas the compute specification happens at individual blocks. The latter increases the design flexibility, and the former reduces the wire utilization between Socket CC and Socket CL. Once the memory connectivity topology is settled, we compute the DMA and LSU configurations for each data copy task. This is based on the data copy logic that we had extracted earlier. Chiefly, the configurations should entail external memory offsets, transfer length and stride, and LSU block/cyclic mode. These will govern where to obtain the data stream and how to store the data stream in the RAM blocks inside the memory logic (for the read case). There are likely several data copy tasks that require different DMA/LSU configurations at different points of program execution. They are reflected in the software code (we often refer to them as the schedules of data movement). Since our controller software adopts a single-threaded execution model, it should be clear from the software when a task occurs.

Finally, what is left of the design is the compute logic. In a sense, the compute logic is neither data copy logic nor memory blocks; it is not as straightforwardly recognizable as the other two. The term "computational datapath" is rather ambiguous; we sometimes refer to it as custom logic. The compute, or custom logic may have its own control logic or memory buffers. For instance, it may keep some temporary buffers for data locality that do not require writing back to the external memory. It may also keep different states depending on input parameters. Therefore, our approach is to loosely define the compute/custom logic. Ultimately, we aim to offload most complex and redundant logic from the design through Socket CC and the RAM group structure of the memory unit, so that the HLS tool could better optimize the remaining custom logic. Finally, we incorporate the schedule of the compute logic in the software code. The outcome of the process is the custom logic module (input to HLS tool), the memory logic module (containing RAM groups and their connectivity topology to Socket CC and the custom logic), and the single-threaded software C code. The former two are combined to produce Socket CL design, whereas the latter is compiled by a C compiler to generate a control binary program for Socket CC's controller.

For performance reasons, we usually would like to run multiple sockets concurrently. Each socket could operate on different design kernels in an application as in the task-level parallelism model. This may require each socket to have a distinct design and controller software. For the data-level parallelism model, the sockets execute similar kernel on different slices of data. In this case, the socket design is reusable; we only need to take care of how to distribute the workload among them in the controller software code of each socket. We could perform a static distribution. Concretely, each socket is assigned a unique identifier. Workload distribution is a function of the socket identifier and the total number of sockets in the full design. Dynamic distribution is slightly more complex; it usually requires some coordination between the sockets. There should be a socket acting as a master that updates a job queue and other sockets retrieve work items from the queue.

Currently, the high-level mapping procedure entails manual effort from users. Automat-

ing this process is left to future work. We shall see the aforementioned procedure in a better light with a concrete design served as a running example in the following section.

6.2   Design Example: Matrix Multiply

Matrix Multiply is an important kernel for numerous applications. We make an assumption that our applications require a large dataset that could not fit entirely into the on-chip memory resource of the target device. Therefore, we focus on block-based algorithms (tiling) when designing applications for the front-end flow. In particular, a socket works on a smaller problem at a time, then writes back the result, and continues with the next batch. Tiling also improves data reuse, thus reducing the number of external memory accesses, leading to better performance.

6.2.1   General socket design

Listing 6.1 demonstrates a typical tiling-based design of matrix multiply. In this example, A, B, and C denote pointers to external memory space. On the other hand, *local_a*, *local_b*, and *local_c* represent on-chip memory buffers. $B \times B$ is the block size of each memory buffer, and $N \times N$ is the matrix size of A, B, and C. For the sake of simplicity, we examine square matrices for now.

```
1   for (i = 0; i < N / B; i++) {
2     for (j = 0; j < N / B; j++) {
3
4       for (ii = 0; ii < SIZE; ii++) {
5         for (jj = 0; jj < SIZE; jj++) {
6           local_c(ii, jj) = C(i + ii, j + jj);
7         }
8       }
9
10      for (k = 0; k < N / B; k++) {
11        for (ii = 0; ii < SIZE; ii++) {
12          for (kk = 0; kk < SIZE; kk++) {
13            local_a(ii, kk) = A(i + ii, k + kk);
14          }
15        }
16
17        for (kk = 0; kk < SIZE; kk++) {
18          for (jj = 0; jj < SIZE; jj++) {
19            local_b(kk, jj) = B(k + kk, j + jj);
20          }
21        }
22
23        for (ii = 0; ii < SIZE; ii++) {
24          for (jj = 0; jj < SIZE; jj++) {
25            for (kk = 0; kk < SIZE; kk++) {
26              local_c(ii, jj) += local_a(ii, kk) * local_b(kk, jj);
27            }
28          }
29        }
30      }
31
32      for (ii = 0; ii < SIZE; ii++) {
33        for (jj = 0; jj < SIZE; jj++) {
34          C(i + ii, j + jj) = local_c(ii, jj);
35        }
36      }
37    }
38  }
```

Listing 6.1: Tiled Matrix Multiply

Since our model adopts software-managed memory logic, data copy is explicit. L4–8 fetches a block of C data to $local\_c$. The computation is being performed on this block, and once it is done, the result is written back to C as shown in L32-36. A tile of A and B are read in L11–15 and L17–21, respectively. These tiles are reused throughout the computation of $local\_c$ in L23–L29.

From the outline of the algorithm, we can identify the structure and control flow of the design that fits our model. In particular, we recognize the memory buffers, the data movement tasks, the compute task, and the control sequence tasks. Listing 6.2 rewrites the

algorithm to abstract away most of the details to leave just the important tasks that govern the functionality of the design as well as the control flow among them.

```
1  local_a[SIZE * SIZE];
2  local_b[SIZE * SIZE];
3  local_c[SIZE * SIZE];
4
5  for (i = 0; i < N / B; i++) {
6    for (j = 0; j < N / B; j++) {
7      dma_copy(read, local_c, C, i, j);
8
9      for (k = 0; k < N / B; k++) {
10       dma_copy(read, local_a, A, i, k);
11       dma_copy(read, local_b, B, k, j);
12       compute(local_a, local_b, local_c);
13     }
14
15     dma_copy(write, local_c, C, i, j);
16   }
17 }
```

Listing 6.2: Task-based Matrix Multiply

This gives us a sketch of the controller software. The details of the DMA copy operations depend on the memory logic configuration, which is our next concern. We could use a single RAM block for each local buffer. However, this will unlikely give an optimal performance, as this will present a limited number of memory ports for the compute logic. Refer to the compute task (L23-L29) from Listing 6.1 again, assume the operation in L26 could finish in one clock cycle, the compute task would consume $SIZE \times SIZE \times SIZE$ cycles. If we fully unroll the innermost loop in L25, and if the loop body could finish in one cycle, the speedup would be $SIZE$ times. That is only possible if the compute logic can read from $SIZE$ different $local\_A$ and $local\_B$ memory ports in a cycle. Therefore, we could partition the local arrays $local\_A$ and $local\_B$ by a factor of $SIZE$. $local\_A$ should be partitioned on the second dimension (cyclic mode), and $local\_B$ partitioned on the first dimension (block mode) that corresponds to their kk subscript. (Here we use the convention that the array dimensions are counted from the highest to the lowest.) Thus, $SIZE$ determines the number of RAM blocks for $local\_a$ and $local\_b$. Assuming $SIZE$ is 16, we will need 16 RAM blocks, or 4 RAM groups to represent $local\_a$ and $local\_b$ each, for a total of 32 RAM blocks. We can use one RAM block for $local\_c$ to store the accumulation result. This leads to 33 RAM blocks for the local arrays. It also shapes the interface for the compute logic. The design of the compute logic could be accomplished with a 16-wide dot product engine that contains 16 multipliers and 1 adder (for accumulation). Figure 6.1 demonstrate one example of CL design.

Once the memory structure is settled, we need to devise the DMA/LSU configurations for the memory copy tasks. We can use different AXI buses for the data copy tasks of A and B in L10-11 from Listing 6.2, respectively so that their data transfers could happen concurrently. This requires two DMA engines (and two LSUs) in the socket design. We
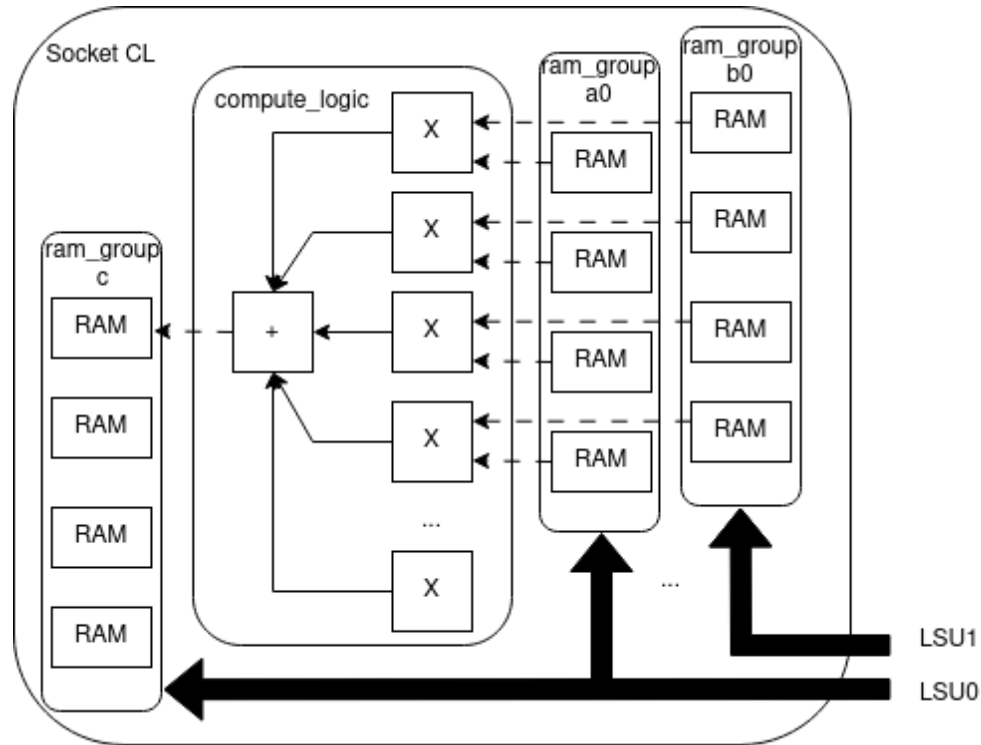
Figure 6.1: Custom Logic design showing the dot product engine and connected RAM blocks. The dash lines denote RAM read/write operations. The thin dash black lines denote the datapath within the compute logic. The thick black lines denote the LSU interfaces with the RAM groups. An LSU interface consists of 4 sets of RAM interfaces, one for each RAM block within a group.

could map the RAM blocks corresponding to *local_a* and *local_c* to DMA engine 0, and *local_b* to DMA engine 1.

Listing 6.3 shows the DMA/LSU configuration for the data copy task in L7 of Listing 6.2. We provide DMA0 and LSU0 with the configuration for retrieving a data stream from external memory (C array) to the local on-chip buffer. The local buffer is mapped to the RAM block at index 33. Since there is no partitioning involved, we set the block factor to the total amount of data to copy (which is $SIZE \times SIZE$), and the cyclic factor to 1. L6-9 denotes the tile layout that we would like to extract from C. This will invoke the DMA to issue $SIZE$ memory read requests of $SIZE$ length for each row of C with the specified starting offset. L10 sets the mode of the DMA (1 for Read, 0 for Write). L11-12 enqueue the DMA tasks to the task queue (TQ). The tasks only run when TQ removes them at runtime. The start task will commence the LSU and DMA operation, whereas the done task will block TQ until it receives the done signal from the LSU and the DMA. This prevents TQ from firing a new DMA operation while the current one is still in flight.

```
1
2  // DMA/LSU configurations for reading C to local_c
3  LSU0_RAM_START_IDX = 33;
4  LSU0_RAM_BLOCK_FACTOR = SIZE * SIZE;
5  LSU0_RAM_CYCLIC_FACTOR = 1;
6  LSU0_EXT_MEM_OFFSET = &C(i + 0, j + 0);
7  LSU0_SEG_STRIDE = N;
8  LSU0_SEG_COUNT = SIZE;
9  LSU0_LEN = SIZE;
10 LSU0_MODE = 1; // read
11 TQ_LSU0_START();
12 TQ_LSU0_DONE();
```

Listing 6.3: DMA/LSU configurations for reading C

Listing 6.4 shows the DMA/LSU configurations for the tasks in L10-11 from Listing 6.2. We specify the starting RAM block index as 0 for *local_a*, and 16 for *local_b*. The configuration code does not specify which other RAM blocks correspond to those arrays. By design, the LSU logic only needs the starting RAM block index to start an operation. It will access the RAM block and the consecutive blocks from that index. The number of blocks involved in an operation is deduced from other configurations such as the block and cyclic factor, the transfer length, and the number of transfer segments. In the *local_a* case, since the block factor is 1 and the cyclic factor is $SIZE$ (16), LSU0 will perform its operation in cyclic mode on RAM block 0-15.

On the other hand, for *local_b*, since the block factor is $SIZE$ and the cyclic factor is 1, LSU1 makes data copy in block mode. For every $SIZE$ data element copied, it increments the index counter by 1 to move to the consecutive RAM block. As the configuration specifies $SIZE$ transfer segments (L18), it requires access to the RAM block 16-31.

The external memory tile layout specifications of A and B are similar to the case of reading from C. Since LSU0 and LSU1 utilize separate AXI buses, they can operate concurrently. Therefore, we can enqueue their start operations one after another as demonstrated in L22-23. To make sure that one does not block the operation of the other, the done operations are pushed to TQ after all the start operations (L24-25). This effectively overlaps the operations of the two LSUs.

```
1
2  // DMA/LSU configurations for reading A to local_a
3  LSU0_RAM_START_IDX = 0;
4  LSU0_RAM_BLOCK_FACTOR = 1;
5  LSU0_RAM_CYCLIC_FACTOR = SIZE;
6  LSU0_EXT_MEM_OFFSET = &A(i + 0, k + 0);
7  LSU0_SEG_STRIDE = N;
8  LSU0_SEG_COUNT = SIZE;
9  LSU0_LEN = SIZE;
10 LSU0_MODE = 1; // read
11
12 // DMA/LSU configurations for reading B to local_b
13 LSU1_RAM_START_IDX = 16;
```

```
14 LSU1_RAM_BLOCK_FACTOR = SIZE;
15 LSU1_RAM_CYCLIC_FACTOR = 1;
16 LSU1_EXT_MEM_OFFSET = &B(k + 0, j + 0);
17 LSU1_SEG_STRIDE = N;
18 LSU1_SEG_COUNT = SIZE;
19 LSU1_LEN = SIZE;
20 LSU1_MODE = 1; // read
21
22 TQ_LSU0_START();
23 TQ_LSU1_START();
24 TQ_LSU0_DONE();
25 TQ_LSU1_DONE();
```

Listing 6.4: DMA/LSU configurations for reading A and B

Note that the order of the done operations does not matter in this case. Once an LSU asserts its done signal, a corresponding sticky bit in the controller will be asserted; it only resets when TQ dequeues the relevant done operation of that LSU. Hence, even if LSU1 finishes earlier than LSU0, the LSU1 sticky done bit remains HIGH until LSU0 is done and TQ removes its done operation. Then it removes LSU1's done operation from the queue. However, there might case where we want to kick off another run of LSU1 immediately after it finishes. In this case, it would have to wait until LSU0 is done. This bottleneck is caused by the single, centralized task queue that is shared by every task. This issue could be mitigated by using more than one queue, yet it would require more complex control logic to handle the dependency among the tasks. Using a centralized queue simplifies the dependency handling, and presents a truly single-threaded execution model from the software viewpoint, hence making it more apparent to reason about the performance or debugging.

In some cases, one could estimate the task latency at compile time. That would assist in figuring out the optimal ordering of multiple start and done operations of different DMAs/LSUs to ensure the most overlapping of their executions. For example, if one LSU0 operation is as long as two LSU1 operations, we could apply the following configuration.

```
1 // LSU0 configuration
2 ...
3 TQ_LSU0_START();
4
5 // LSU1 configuration (first run)
6 ...
7 TQ_LSU1_START();
8
9 // LSU1 configuration (second run)
10 ...
11 TQ_LSU1_DONE();
12 TQ_LSU1_START();
13 TQ_LSU0_DONE();
```

Listing 6.5: Example of overlapping multiple DMA operations

Note that the configuration setup (setting various DMA/LSU parameters) also consumes cycles. Each parameter update invokes a CPU store operation to the MMIO address of the register holding that parameter which takes a cycle. Therefore, a start operation should be enqueued as soon as we are done setting up the configuration for the DMA/LSU. We maintain separate queues for buffering the configurations, namely configuration queues. Per each start operation (once $TQ\_LSUX\_START()$ completes), the configuration queues will dequeue a configuration to send it to the corresponding LSU, otherwise they will keep enqueuing new configurations as the software code steps through the parameter updates. If the queues are full, the controller will generate a stall signal to halt the CPU operation to ensure no information is lost. The queues become available again once an LSU operation is completed, and the next configuration are popped from the queues for the next LSU run.

Finally, Listing 6.6 shows the DMA/LSU write configuration for C. It is similar to 6.3 by and large, except for setting the LSU mode to write instead of read in L10.

```
1
2 // DMA/LSU configurations for writing C from local_c
3 LSU0_RAM_START_IDX = 33;
4 LSU0_RAM_BLOCK_FACTOR = SIZE * SIZE;
5 LSU0_RAM_CYCLIC_FACTOR = 1;
6 LSU0_EXT_MEM_OFFSET = &C(i + 0, j + 0);
7 LSU0_SEG_STRIDE = N;
8 LSU0_SEG_COUNT = SIZE;
9 LSU0_LEN = SIZE;
10 LSU0_MODE = 2; // write
11 TQ_LSU0_START();
12 TQ_LSU0_DONE();
```

Listing 6.6: DMA/LSU configurations for writing C

These encapsulate the configurations for the data copy tasks. Next, we cover the configuration of the compute logic (CL). Listing 6.7 presents the CL configuration for the compute task. The task uses one scalar parameter, KRN_LEN which denotes the dot-product length. The scalar parameters are CL-specific. Their MMIO registers are not defined in Socket CC's MMIO space. The controller uses a separate RAM interface to inform CL of the parameter value set via the software. The scalar parameters are also buffered to a queue reserved for CL scalars in Socket CC. Since the number of CL scalars depends on specific CL design and is unknown to Socket CC, L2 serves as a delimiter between different batches of parameter configurations (one batch corresponds to one CL run). L3-4 enqueue the CL start and done operations to TQ. Concretely, when TQ dequeues a CL start operation, the controller will dequeue the CL configuration queue until it reaches the delimiter. Each retrieved parameter will be sent by the controller's RAM interface to Socket CL. The last parameter will assert the start signal of CL. Note that the access latency between Socket CC and Socket CL is long due to extra pipelined registers (on FF bridge). Therefore, one should avoid the roundtrip communication between the two entities, such as read operations (e.g., checking CL done signal, or certain parameter values). Consequently, there is a separate *cl_done* signal from Socket CL to Socket CC to inform the latter once its CL operation completes. The signal

will assert the CL sticky bit in the controller, and then TQ could dequeue the CL done operation.

```
1 KRN_LEN = SIZE;
2 KRN_START = 1;
3 TQ_CL_START();
4 TQ_CL_DONE;
```

Listing 6.7: CL configuration

All the complexity regarding latency between different socket partitions, as well as task queue and various configuration buffers' size are handled by the controller, and hidden from the software perspective. Certainly, different controller designs would vary the performance. For example, if the task or buffers' sizes are too small, this will lead to more frequent stalls in the software execution, thus increasing the control overhead. If they are too big, the controller design will become very complex, hence yielding sub-optimal QoR. In practice, stalls are tolerable as long as the control overhead could be largely hidden by the DMA/LSU and/or CL operations. The control overhead is manifested in the parameter setup and the control flow of the software program.

Listing 6.8 shows the final piece of the program. L1 will keep polling the status of TQ. It waits until TQ is fully drained, thus becoming empy. This implies all the enqueued tasks are complete, and the socket can wrap up its execution. L3-5 sets the AXI control module to inform socket manager that this socket (at $CORE\_ID$) has finished the execution. The communication entails a single non-burst AXI transaction from the socket's NMU to the socket manager's NSU over the NoC. Finally, L7 asserts the socket done signal.

```
1 while (TQ_EMPTY_N == 1);
2 CTRL_MAXI_SOCKET_OFFSET = SOCKET_MANAGER_NOC_ADDR + STATUS_COMPLETE_OFFSET
    (CORE_ID);
3 CTRL_MAXI_WRITE = 1;
4 while (CTRL_MAXI_WRITE_DONE == 0);
5 CPU_STATUS = 1;
```

Listing 6.8: Closing code

6.2.2   Optimizations

There are several approaches to further improve the performance of the general design in the previous section. First, we do not need to use a square-based tile. Similarly, the tile dimension is not necessarily a power of two. Square tiles limit the design space, while rectangular tiles allow us to choose a tile size that best fits the available RAM blocks of our socket model. Hence, we denote BHA, BWA (height and width, respectively) as the tile dimensions for matrix A, and BHB, BWB as the tile dimensions for B. This implies that BWA and BHB are identical, and BHA and BWB are the tile dimensions of C. This will give us more design choices to explore. Notably, BWA plays the key role in deciding the number of RAM blocks for $local\_a$ and $local\_b$, in addition to the width of the dot-product engine in CL.

Second, in the general design, we only use one RAM block for *local_c*. Although this is not a bottleneck in CL's performance (the dot-product engine can only generate one result in a cycle), it will severely impact the data copy of *local_c*. Since the cyclic factor is 1, only one LSU port is used. Assume our AXI data width is 512-bit and the LSU data width is 64-bit. Per cycle, the LSU receives a data stream of 8×64-bit items. Since only one LSU port is active, it requires 8 cycles to copy one data stream! During this interval, the subsequent data streams need to be buffered, or back-pressured to the source, reducing memory bandwidth utilization. By employing 4 RAM blocks instead, it will speed up the data transfer by 4 times. Therefore, we could set the cyclic factor to 4 to enable cyclic mode. This needs the accommodation of some extra logic in CL design to handle the updates to 4 *local_c* RAM blocks cyclically.

The third optimization is that we can overlap the executions of the data copy tasks of A and B with the compute task by employing double buffering techniques. This gives rise to doubling the number of RAM blocks for *local_a* and *local_b*. Concretely, if BWA is 16, we need 64 RAM blocks in total ($4 \times BWA$) of which 32 are for *local_a* and *local_b* each. Each group of 32 RAM blocks is further divided into two sets of 16 RAM blocks ("ping" and "pong" buffers). While the data copy tasks operate on the ping RAM blocks, CL performs the computation on the pong ones, and vice versa. CL needs an additional parameter to control the buffer switching in software.

The double buffering optimization unfortunately leads to the explosion of RAM block usage. The design may not be feasible to fit in a socket area for a large BWA value. How do we manage to reduce the total RAM block counts? Recall that the RAM blocks are built upon BRAM, URAM, or LUTRAM. BRAM-type block allows dual-ported read accesses in a cycle. We could make use of such property to reduce the RAM count. In particular, two adjacent elements of matrix A could be stored in one RAM block. Similarly, we could hold two consecutive lines of matrix B in another RAM block. Issuing 2 read operations per RAM block in a cycle will increase the amount of data available to CL twofold. Therefore, we can reduce the total number of RAM blocks by 2, while keeping the width of the CL's dot-product engine intact.

Notice that, so far we only discuss the design choices of a single socket. We could invoke multiple sockets to do matrix multiply in parallel. In this algorithm we can observe that a tile of C only depends on the current (i, j) iteration; there is no data dependence among the iterations of the outermost loops. Therefore, each tile of C could be computed independently by one socket. We modify the outermost loops (i, j) to distribute the workload (C tiles in this case) among the known number of sockets in the system. The distribution could be performed cyclically as follows.

```
for (int w = CORE_ID; w < total_work; w += NUM_CORES
```

Here, $CORE\_ID$ and NUM_CORES are constant parameters in each socket software. The latter refers to the number of sockets, while the former is the socket identifier, which is a unique value from 0 to $NUM\_CORES - 1$. Some sockets could also share tiles of A and B to minimize the data movement to and from the DRAM. We will leave such optimization

(sharing data among the sockets) to future work.

Putting them all together, Listing A.1 demonstrates the complete optimized controller software program for this benchmark.

Chapter 7

Evaluation

7.1   Experimental Setup and Benchmarks

We use AMD Xilinx Vitis and Vivado 2022.2 for our experiments. The EDA software flows run on a single host machine (AMD Ryzen Threadripper 2990WX 32-Core @1.71GHz, 32GB RAM). We installed an AMD Versal VCK5000 card to one PCIe-x16 slot of the machine. We use XRT Runtime API to interact with the card from host programs. We select 5 benchmarks from different application domains, such as linear algebra (*matmul*, *cholesky*), stencil (*jacobi_2d*), sparse computation (*spmv*), deep learning (*linear*, or *matrix-vector multiply*). Some benchmarks operate on 64-bit integers, while others use 32-bit floating point. The benchmarks contain several loop nests. The outermost loop of a benchmark is parallelized with multiple socket instances. Each socket instance works on different slices of data. Within a socket, we perform inner-loop optimizations (tiling, unrolling, pipelining, double buffering) to maximize the performance of a socket under the resource constraint of the socket floorplan. In this experiment, we set the socket floorplan to *socket_m*. Therefore, we can instantiate 9 sockets as the maximum. Benchmarks such as *cholesky* and *jacobi-2d* require synchronization between sockets at some loop level during the execution to ensure no data race.

We use Vitis HLS to generate RTL for the custom logic, and port it to our backend compile flow. All 9 sockets share the same implementation per benchmark. The baseline for comparison is the full top-down Vitis flow in which we also implement a 9-core design without any floorplanning. We design each benchmark core using Vitis HLS. To ensure a fair assessment, each core has similar optimizations to a socket, so their cycle counts are equivalent. Note that the Vitis HLS-designed core does not employ our Socket CC; the compute, memory, scheduler, and AXI bus logic are designed altogether in the Vitis HLS core. The Vitis HLS core employs a 256-bit AXI interface, and we do not apply multi-clock domain optimization on it. The target frequency for both SPADES and Vitis flow is 500MHz. We set the Vivado *maxThreads* parameter to 32 to utilize all CPU cores of our test machine per Vitis/Vivado run.

Table 7.1: Benchmarks

Benchmark	Domain	Data type	Problem size	Socket Cycle count	Vcore Cycle count
matmul	Linear algebra	64-bit integer	128x128	120113	111994
cholesky	Linear algebra	64-bit integer [a]	128x128	99301	91883
jacobi-2d	Stencil	64-bit integer	256x256	136481	117894
spmv	Sparse computation	64-bit integer	256, 256x256	22917	21775
linear	Deep Learning	32-bit float	512, 512x512	131166	129106
conv3d	Deep Learning	32-bit float	32x13x13, 32x32x3x3	47381	46919

[a]Since the original benchmark contains square root and division operations, their results are rounded to the nearest integers.

Table 7.1 gives a summary of the benchmarks employed in our studies and their respective single instance's cycle counts obtained by RTL simulations for the given problem size is indicated in the fourth column of the table. **Socket Cycle count** refers to the number of cycles taken by our socket design to execute the given problem size. **Vcore Cycle count** is the single Vitis core performance of similar problem size. We can observe that their cycle counts are comparable across all benchmarks. The Vcore is slightly better in all cases. This is within our expectation due to the overhead induced by the fixed Socket CC overlay of our socket design. We shall see that the cycle count is not the absolute metric in deciding the overall performance.

7.2 Compile Time of the Full Design

We compare the compile time of the full design produced by our flow against Vitis. The full design refers to a design containing 9 SPADES sockets (or Vitis cores). Regarding the SPADES flow, the compile time includes the time of compiling Socket CL, and design assembly. On the other hand, Vitis compile time encompasses the time taken by the Synthesis and Implementation engines. In particulars, we take into consideration the runtime of the following steps: *synth_design*, *opt_design*, *place_design*, *phys_opt_design*, *route_design*. The runtimes are extracted from the Vitis/Vivado log files. There are some steps that we do not factor in the compile time. First is the *link_design* time that occurs immediately after

Compile time (Synthesis + Implementation) of 9-socket Design
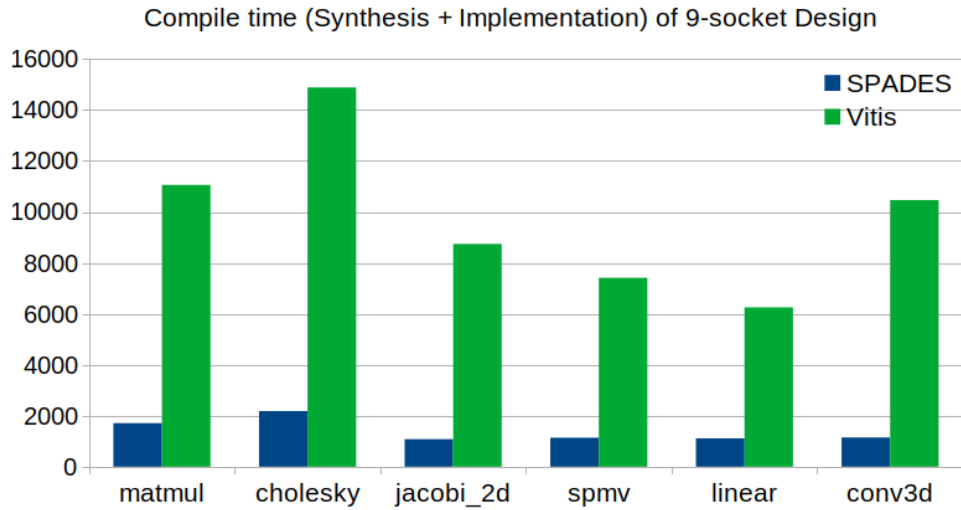
Figure 7.1: Total Compile Time of a 9-socket design: SPADES vs. Vitis (lower is better)

the synthesis step. This step combines the shell checkpoint with the synthesized checkpoint of the design to produce a complete design before handing it off to the next step in the implementation phase. It incurs heavy IO usage as the shell checkpoint is usually large. Additionally, we do not count this step in the SPADES compile time (*link_design* is also needed in our flow, but the difference is that it occurs at the final stage once we have a full P&R socket design). Therefore, to ensure a fair comparison, we omit the runtime of this step from both flows.

We also do not consider the bitstream generation step when evaluating the compile time. For complex and large designs, this step may take 30-45 minutes. Because the bitstream generation engine is identical in both flows, omitting the runtime of this process still ensures fair assessment between SPADES and Vitis. Optimizing the runtime of bitstream generation is not within the scope of this work. Finally, we exclude the front-end design time, e.g., HLS runtime, since it is generally negligible in comparison to the later steps. Therefore, the compile time in this context essentially means the time taken to synthesize and implement an RTL design to a P&R netlist. Note that, although a Vitis design uses 9 identical accelerator cores, we only factor the synthesis time of one core in the total compile time, as the tool can reuse the synthesized result of the core for other instances.

Figure 7.1 shows the comparison of SPADES and Vitis in compile time. Our flow is **6.6x** faster than Vitis on average and able to finish within 20-30 minutes. In contrast, Vitis requires several hours to finish the compilation. Particularly, benchmark *matmul* compiled in 3 hours. Chiefly, the *place_design* and *route_design* steps are the most time-consuming tasks in Vitis flow. The primary contribution to the substantial speedup is that our flow is able to reuse the P&R netlist of a single socket. Particularly, we perform P&R at a smaller

[SPADES] Compile time Breakdown

conv3d	130	334	344	193	
linear	163	307	340	176	
spmv	147	266	351	172	
jacobi_2d	110	286	339	175	
cholesky	163	483	381	210	
matmul	143	491	608	260	

Legend: synth, link+opt, place, phys_opt, route, socket_stitch, assembly, noc_route
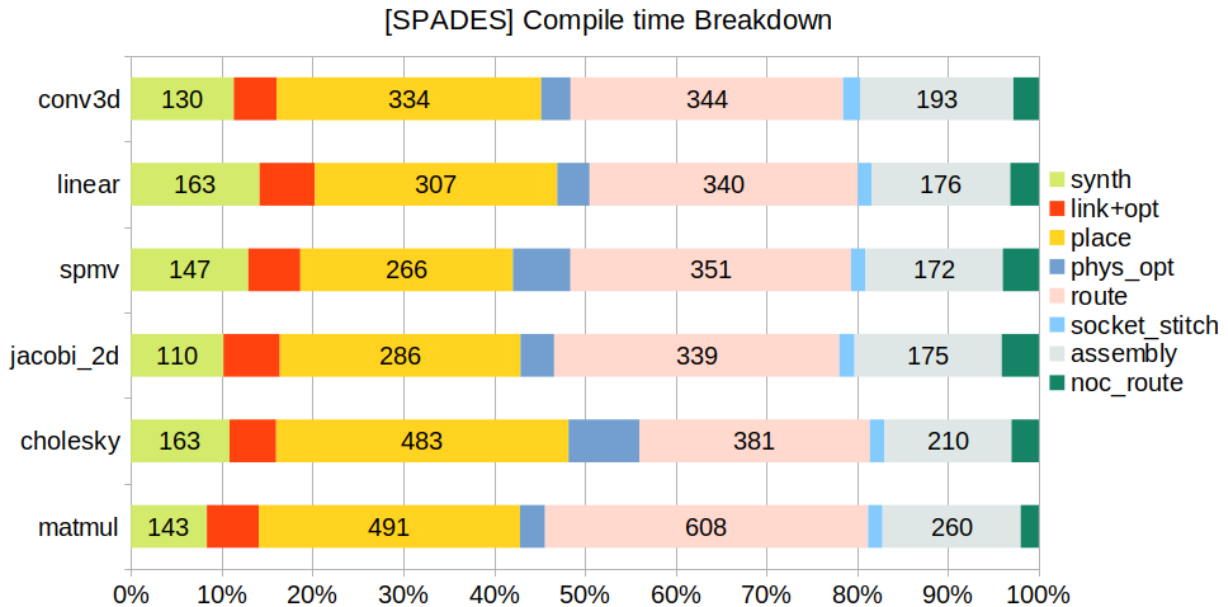
Figure 7.2: A breakdown showing the task runtime percentages of the overall SPADES compile time. Time-consuming tasks are annotated with their runtimes in seconds.

scale and carry out integration to compose the full design. On the other hand, although Vitis is capable of reusing synthesized results, it still implements the design in a top-down manner.

7.3 Analysis of Socket Compile time

We now analyze the compile time of a single socket to understand the bottleneck and how it could be improved. Figure 7.2 gives a breakdown of the time taken by each task in SPADES. The custom logic compilation (Step 2), especially placement and routing, dominates most of the time, whereas Step 3 (Socket Stitching and Design Assembly) is relatively fast (3-5 min.) for our 9-socket design. Therefore, to further reduce the compile time of a socket, it is important to optimize the runtime of P&R processes. There are several choices that one could make to improve their runtime. First, we found that user timing constraint significantly impacts the P&R runtime. If the constraint is tight, the tool will try to perform various optimizations during the placement and routing steps to meet such timing. Those often include several attempts of re-placing several cells or re-routing some nets to improve the critical paths. Thus, it might help to relax the timing constraint if one could afford to do so in compensate for an improvement in compile time.

Second, routing congestion negatively influences both the compile time and achievable

frequency. Congestion usually occurs in designs in which there are nets with huge fanouts, high usage of FFs and/or LUTRAMs. One approach to mitigate this issue is avoiding excessive LUTRAM utilization by redesigning Socket CL to use dense RAM blocks instead. Congestion is also a sign of inadequate routing resources, so one might try resizing the socket floorplan. For instance, one could make use of multiple *socket_m* floorplans.

Lastly, if one has a precompiled Socket CL netlist, it essentially leads to a zero compile time for Step 2. In that case, the overall socket compile time only accounts for Socket Stitching and Design Assembly from Step 3, in addition to the NoC routing time of Step 4. This could happen in situations where Socket CL's functionality gets reused in many applications, e.g., common kernel operations or utilities such as matrix multiplication. We could distribute a library of pre-compiled, highly optimized Socket CL netlists for some particular domains, an approach akin to popular software libraries such as BLAS, OpenCV, etc. That would not only significantly improve the compile time within a socket design, but also the overall design productivity.

7.3.1 Does pre-compiling Socket CC help the compile time?

As a reminder, our approach entails partitioning the socket floorplan into two disjoint partitions: Socket CC and Socket CL. We pre-compile Socket CC and reuse it to improve compile time. In this section, we analyze how much improvement is gained from this approach versus compiling Socket CC and Socket CL as a whole. We refer to the latter as the Standalone approach. Firstly, we examine the Vivado runtime for the steps of the Pre-compiled Socket CL flow (shortly as Pre-compiled flow) with their counterpart of the Standalone flow as shown in Figure 7.3. In this graph, each column represents a difference in the measured runtimes of Pre-compiled flow from Standalone flow for one particular step:

$$\frac{Standalone\ flow_{stepX} - Precompiled\ flow_{stepX}}{Standalone\ flow_{stepX}}$$

A column above 0 means Standalone flow took longer to finish than Pre-compiled flow, and vice versa. We observe that the Pre-compiled flow is generally faster in most cases. The synthesis runtime is consistently better by around 50% on average in the Pre-compiled flow, since it only needs to synthesize Socket CL logic. Placement runtime also improves over the Standalone flow, as the Pre-compiled flow works on a smaller floorplan and number of cells. On the other hand, routing runtime hardly gets any speedup. For some benchmarks such as *matmul* and *spmv*, the Pre-compiled flow performs worse than the Standalone flow. The constraint on Socket CL's floorplan reduces the available routing resources, hence leading to routing congestion for these benchmarks.

Table 7.2 compares the total compile time between the pre-compiled flow and the standalone flow when we add up the runtimes of all the steps in each flow. Numbers in parentheses represent socket CC-CL stitching time that is only applicable to the Precompiled flow, and as seen from the table, they are negligible in comparison to the overall compile time. In general, the Pre-compiled flow is 57% faster on average than the Standalone flow. The

Socket Compile time Comparison between Precompiled flow and Standalone low
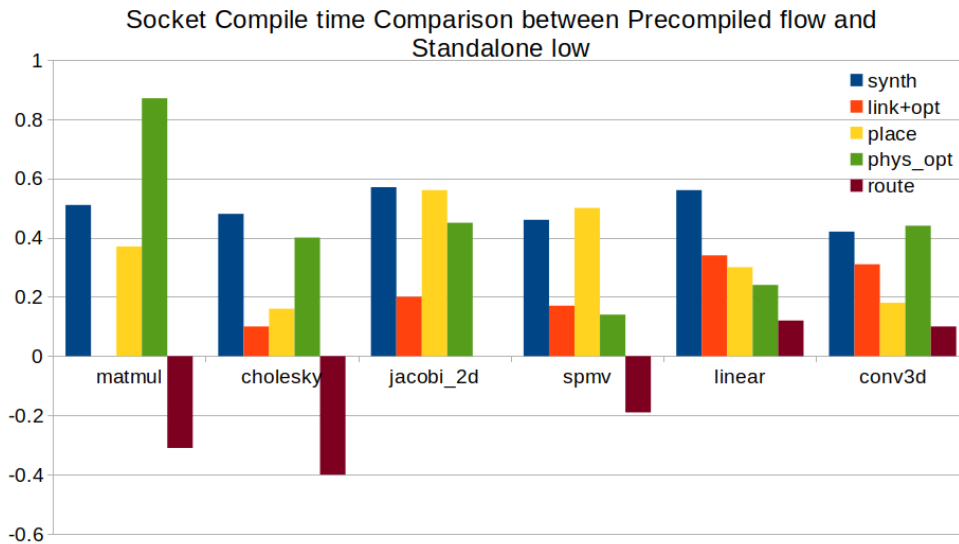
Figure 7.3: Runtime improvement of steps from Pre-compiled flow over Standalone flow for a socket

Table 7.2: Socket Compile time of Pre-compiled flow vs. Standalone flow per socket

Benchmark	Pre-compiled flow (s)	Standalone flow (s)	Improvement
matmul	1414 (27)	1990	40.7%
cholesky	1834 (28)	2016	10%
jacobi-2d	860 (18)	1404	63.3%
spmv	919 (18)	1259	37%
linear	906 (17)	1281	41.3%
conv3d	921 (22)	1152	25.1%

significant contributors to the speedup are the synthesis and placement runtime reductions thanks to reusing Socket CC's logic and implementation. Nonetheless, there is some room for enhancement, especially concerning the routing step.

7.3.2   Does pre-compiling Socket CC help the QoR?

From the previous section, we have seen that pre-compiling a portion of socket design does help the compile time. Here, we further investigate its impact on the QoR of a socket. Table 7.3 compares the QoR between the Pre-compiled flow and the Standalone flow. The numbers of FFs and LUTs of the Pre-compiled flow are the total resource utilization of Socket CC and Socket CL implementations per each resource type. Only LUT and FF are

Table 7.3: Socket QoR Comparison (Pre-compiled flow / Standalone flow)

Benchmark	FF	LUT	FMax
matmul	36679 / 33786	34848 / 31017	496 / 500
cholesky	49943 / 46433	37355 / 35813	422 / 438
jacobi-2d	30068 / 26192	28963 / 27137	500 / 500
spmv	30133 / 27846	30890 / 28840	444 / 432
linear	30171 / 27147	22854 / 21492	472 / 492
conv3d	35903 / 31349	28864 / 28129	426 / 426

considered in the comparison since the BRAM/URAM and DSP utilization are similar in both flows. Note that Socket CC's implementation does not employ BRAM/URAM or DSP slices. The FMax numbers are calculated based on the slack values (Worst-case Negative Slack) from the timing reports. If the slack is positive, we report 500MHz as achievable FMax (target timing was met). We observe that the Pre-compiled flow consumes more FFs and LUTs than the Standalone flow, although they both start from the same RTL block designs. This is because the tool may introduce additional resources during Synthesis and Implementation to optimize the design for timing and/or area. Standalone flow tends to produce a cheaper implementation since it jointly optimize both Socket CC and Socket CL. Thus, there are more opportunities for certain optimizations such as LUT packing which reduces LUT consumption. Other optimizations such as retiming could further vary the differences of FFs between the results of both flows. Next, the Standalone flow also produces a better QoR implementation than the Precompiled flow overall. Since the Standalone flow does not have any constraints regarding the FF bridge and the floorplan partitioning as in the Precompiled flow, it can explore a larger P&R space for timing optimization. Nonetheless, we would like to emphasize that the Precompiled flow's achievable FMax does not degrade as much compared to the Standalone's results, and there are cases in which it delivers better FMax or meets the target timing.

From these results, we conclude that the Precompiled flow improves the compile time of a socket while degrading its QoR a bit. The tradeoff is worthwhile since the speedup of compile time does not severely impact the socket performance. An area for future enhancement would be devising a more effective partitioning between Socket CC and Socket CL without impeding their routing capacity. In this work, we have to spare one CLB tile column between the two partitions to circumvent route conflict issues. We also make use of FF bridge for stitching the two partitions. While it is a simple and fast approach (leading to trivial stitching time), it also becomes an obstruction to the implementation of Socket CC and Socket CL. A more effective stitching strategy is also vital in refining the QoR of the Pre-compiled flow.
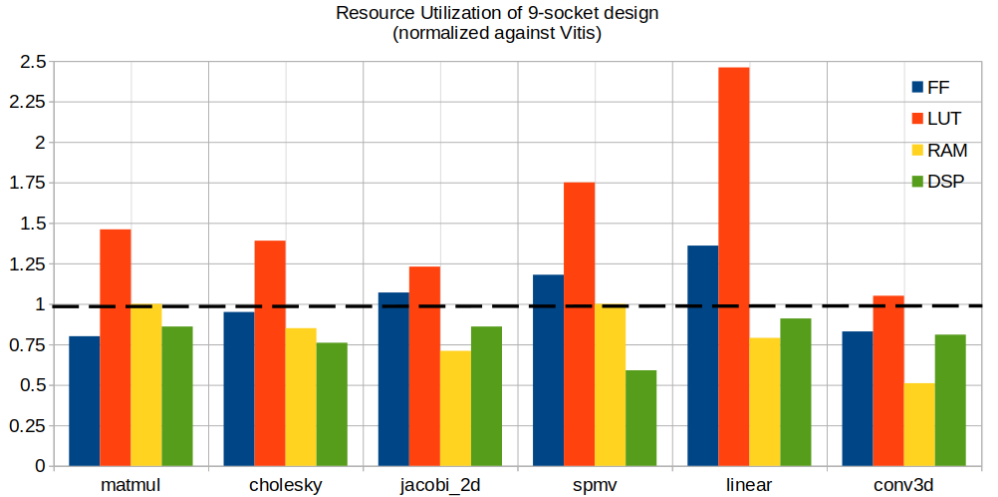
Figure 7.4: Resource utilization of a 9-socket design (SPADES results normalized against Vitis). Lower is better.

7.4 QoR of the Full Design

The previous section analyzes the compile time and QoR of a socket. This section evaluates the QoR of the full design of 9 sockets against the similar design implemented by Vitis.

Figure 7.4 compares the resource utilization of the full design. Each column represents one resource category (FF, LUT, BRAM/URAM or RAM, DSP). We normalize SPADES results over Vitis results per resource category. A column above 1 (or the horizontal dash line) means SPADES consumes more resources for that category, thus the lower the better.

SPADES designs generally consume more LUTs than Vitis designs. This is because a socket in SPADES is more sophisticated than a Vitis core. Particularly, the socket employs many fixed functional blocks such as softcore, DMA engines, LSUs, and AXI adapters for communication and control, whereas the Vitis core build custom control and datapath for communication and control based on the HLS input description. Here, SPADES pays the price of generality in terms of LUT usage.

SPADES consumes more FFs than Vitis for some benchmarks (about 20% at most), while less for others. Since Vitis implements the designs in top-down mode, all cores could be jointly optimized and there might be additional FFs introduced thanks to the tool's optimizations. On the contrary, the total FFs in a SPADES implementation is more or less nine times of the number of FFs utilized by a socket. Another factor is that SPADES employ multi-clock domains. Therefore, Socket CC particularly does not need to consume a substantial number of FFs to meet its lower target timing. On the other hand, the communication logic of a Vitis core needs to meet higher timing. Therefore, the HLS tool might insert more

Table 7.4: FMax of the Full design

Benchmark	SPADES	Vitis	Improvement
matmul	497	322	54%
cholesky	442	310	43%
jacobi-2d	497	374	33%
spmv	442	362	22%
linear	472	361	37%
conv3d	426	294	45%

pipelined registers to the communication logic, leading to higher FF consumption in the case of Vitis designs.

Our on-chip RAM utilization is lower than Vitis because we use LUTRAMs for buffering AXI requests and responses instead of BRAMs as in Vitis HLS design. We also use less DSP resources since we circumvent the multiplier usage when designing the address generation logic for the DMA engines and LSUs. Thus, our BRAM/URAM and DSP utilization are fully dedicated to Socket CL. Another point is that the Vitis core needs to be designed with scalar parameters regarding application parameters (e.g., problem size), core identifier, and number of cores. Therefore, scalar computation logic such as index calculations might involve some multiply operations, leading to additional DSP usage. In our socket design, we offload the majority of index calculation tasks to the software controller, and we can afford to change the values of the parameters in the software. Therefore, the software compiler can compute the results of these calculations statically at compile time. This approach relieves the burden of the custom logic in Socket CL on scalar logic, and typically results in lesser DSP utilization.

Table 7.4 shows the maximum achievable frequency of both flows. Our flow is 38% better than Vitis on average, and we nearly meet the timing target of 500MHz for some benchmarks. To gain a better understanding of the results, we examine the FMax variation (scaling) as we add more SPADES sockets (Vitis core) to the design as illustrated in Figure 7.5. We observe that a SPADES socket achieves better FMax than a Vitis core in all benchmarks thanks to our design and implementation optimizations. In particular, the offloading of control, scalar computation, and communication logic to a separate partition significantly simplifies the custom logic part of the socket. In addition, the control and communication part is clocked at a lower timing target, thus we do not have to aggressively optimize timing for this partition. This leaves the tool ample resources to optimize the custom logic partition. These results in better achievable FMax for SPADES socket. We find that the critical paths of Vitis cores often involve the FSM control or AXI bus logic. Now, we inspect the impact on FMax when adding a socket (core) instances to the design. Our exploit of multiple clock buffers allows us to add more sockets to a design with little or no frequency degradation, while Vitis struggles with keeping up the frequency when the design becomes increasingly
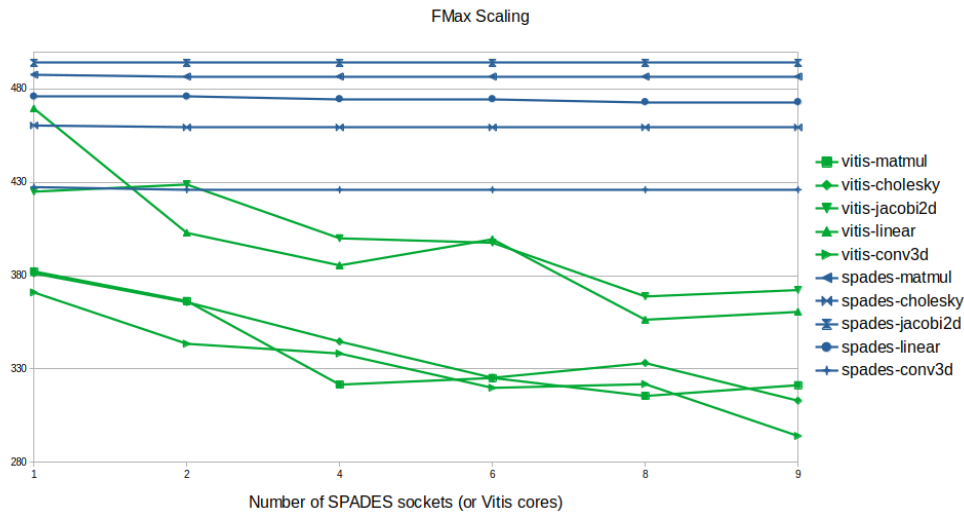
Figure 7.5: Maximum Achievable Frequency scaling concerning the number of sockets (cores) in the design

complex with more cores.

7.4.1   FMax Evaluation between Socket CL and Full design

This section sheds more light on the SPADES FMax results that we have seen previously. As a reminder, we compile Socket CL out-of-context in Step 2 without the static logic. Therefore, we have to model the clock timing characteristics (such as clock insertion delay, and clock uncertainty) as if it were implemented in the context of a real clock from the shell. We are interested in examining how much the Socket CL's timing result obtained in this step deviates from the final full design's timing. Figure 7.6 reports the Worst-case Negative slack (WNS) results of Socket CL and the full design. WNS directly correlates with FMax of our timing evaluation in this work.

$$F_{Max} = \frac{1000}{2 - WNS} \ MHz$$

We can observe that the WNS values are identical between Socket CL and the full design in most cases. This implies that our clock modeling in Step 2 is fairly accurate, and in most cases, the critical paths found in Socket CL are also the same as in the final full designs. This also means that our design assembly phase in Step 3 is effective in preserving all the timing optimizations of Socket CL to the final implementation. Thus, our flow ensures a predictable outcome in timing characteristics of the final design by simply looking at the timing results of individual sockets. Several factors could influence the WNS difference between Socket CL and full design. First, the socket netlist may get relocated to a different region. Therefore,
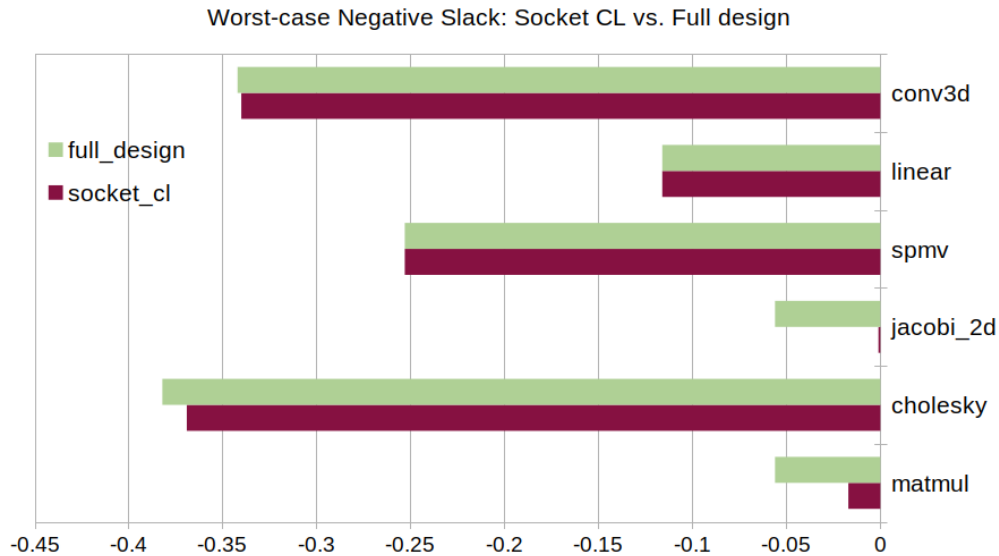
Figure 7.6: Worst-case negative slack: Socket CL vs. Full design

the clock delay to its new clock root could change. Second, each socket instance employs different clock tracks, and we need to perform track switching for sockets that use a track that is different from the one in its Socket CL implementation. The different timing delays between the clock tracks may contribute slightly to the variation in WNS. Nevertheless, the key takeaway here is that it is important to preserve the clock tree of the Socket CL implementation, otherwise the timing results could change drastically.

7.5 The impact of Clock and Reset routing on FMax and Compile time

In this section, we evaluate the effectiveness of our custom clock and reset router in terms of FMax and compile time. Note that "reset" in this context refers to routing a reset signal to the reset pins of the BUFDIV_LEAF cells in each socket region.

Table 7.5 evaluates three different techniques of clock and reset routing. The first technique (1) is employed in our flow. That is, we route the clock and reset signals during the design assembly phase in Step 3. The second technique (2) invokes Vivado to route these signals in Step 4. The last technique (3) also relies on Vivado router, but we use one clock buffer for the entire design, i.e., all sockets share a similar clock buffer (and clock track).

The first three columns of the table show the WNS results of three techniques. We observe that *WNS1* and *WNS2* are identical. This means our custom clock router results in a similar QoR to the standard tool's router. When comparing against single-clock buffer

Table 7.5: WNS results and Routing runtimes for clock and reset signals in different approaches

Benchmark	WNS1 (ns)	WNS2 (ns)	WNS3 (ns)	rt1 (s)	rt2 (s)	rt3 (s)
matmul	-0.056	-0.056	-0.407	16.457	1211	1226
cholesky	-0.382	-0.382	-0.769	17.337	1191	1214
jacobi-2d	-0.056	-0.056	-0.392	8.85	1002	1152
spmv	-0.253	-0.253	-0.634	8.724	985	1077
linear	-0.116	-0.116	-0.329	8.795	896	990
conv3d	-0.342	-0.342	-0.724	9.761	933	971

usage, *WNS1* is significantly better than *WNS3*. This emphasizes the influence of multi-clock buffers in FMax optimization. Had we only utilized a single clock buffer, the clock would have been rerouted for the entire design, since the tool only permits a single clock root per clock track, which subsequently led to an unpredictable WNS result. We suspect that the WNS degradation of (3) is due to the worsening clock skews for sockets that were placed far apart from the clock's central point (root).

The last three columns of the table report the routing runtimes in each case. Our custom router is significantly faster than the standard tool's router. However, this is because Vivado router needs to kick off an initial phase of building a routing graph of the entire design before any actual routing jobs could execute. This phase consumes the majority of the routing runtime in (2) and (3). In contrast, our custom router is relatively simple in that it is aware of what routing resource to build the routes upon: for the clock signals, it only needs to examine the clock interconnect tiles, whereas in the case of reset signals, it knows the fabric regions that are free of routed nets (i.e., "gaps" beneath and beside adjacent socket regions). More importantly, the heavy-lifting task has already been done in the former step, that is to route the clocks and resets within individual socket regions.

7.6 System-level Performance

This section evaluates the system performance of the benchmarks on VCK5000. We program the accelerator card with the bitstreams generated from SPADES and Vitis flow. For each benchmark, we set a host program to initialize and copy data from the x86 host to the FPGA DRAM, invoke the PL execution, and copy the result back to the host for verification. The host code is also in charge of generating the software programs for the socket instances, and controlling the socket manager to dispatch the programs to every socket in the system. Once the configuration of all sockets is done, the socket manager ignites the socket executions, and waits until their completion.

We are only interested in comparing benchmark executions of SPADES versus Vitis implementations, therefore neither host-device memory transfer time nor FPGA reconfiguration

Table 7.6: Performance Comparison (SPADES / Vitis / SPADEStd12)

Benchmark	Problem size	Exec. time (us)		
		Socket	Vitis	SPADEStd12
matmul	1024x1024	11319	17239	12092
cholesky	4096x4096	451742	480184	470773
jacobi-2d	8192x8192	117124	136188	119768
spmv	8192	12093	11953	13370
linear (mvm)	4096, 4096x4096	3327	3206	3916
conv3d	192x13x13, 384x192x3x3	899	1234	678

time is factored into our results. We also did not factor in the controller software configuration time in the performance results. Therefore, the system performance here could be understood as the execution time of the SPADES design, including communication between the sockets with the FPGA DRAM or among themselves or with socket manager, in addition to their computations, starting as early as when the socket manager sends wake-up calls to the sockets, until the socket manager receives the done messages from all the sockets.

Since our approach could only scale up to 9 *socket_m*, we are also keen on investigating how it fares against a full 12-socket implementations by the top-down SPADES flow without socket floorplanning (SPADEStd12). Table 7.6 shows that SPADES incurs virtually little to no performance loss to Vitis or SPADEStd12.

We attribute the results to several factors as follows. (1) Although our cycle counts are worse than Vitis HLS cores' due to the control overhead of Socket CC, it is well-compensated by the improvement in clock frequency owing to our physical design choices and optimizations. (2) To mitigate the control overhead, we employ a task queue for asynchronous execution of the custom logic and the DMAs, thus effectively overlapping the work done by the controller (such as address generation, scalar computation, branches, and loops) with the DMAs and the custom logic. (3) More sockets (or cores) do not necessarily lead to better performance, either due to lower achievable frequency or saturated memory bandwidth.

The last point could be strengthened by examining the results shown in Figure 7.7. Here, we plot the system performance at various numbers of SPADES sockets. The results are normalized against single-socket performance. Therefore, this graph demonstrates the speedup of multi-socket execution over the case of a single socket under a fixed problem size in each benchmark (strong scaling). We could observe that *matmul* performance grows fastest as we have more sockets, while the improvements of *linear*, *cholesky* are somewhat slower. The remaining benchmarks, *jacobi-2d* and *spmv*, saturate the performance before reaching 9 sockets. In principle, adding more sockets increases memory bandwidth usage, and since the external DRAM memory bandwidth is limited, when the system bandwidth demand reaches the maximum, the socket's memory latency becomes worse. In consequence, for compute-dominant benchmarks in which the compute latency outweighs the memory latency as in
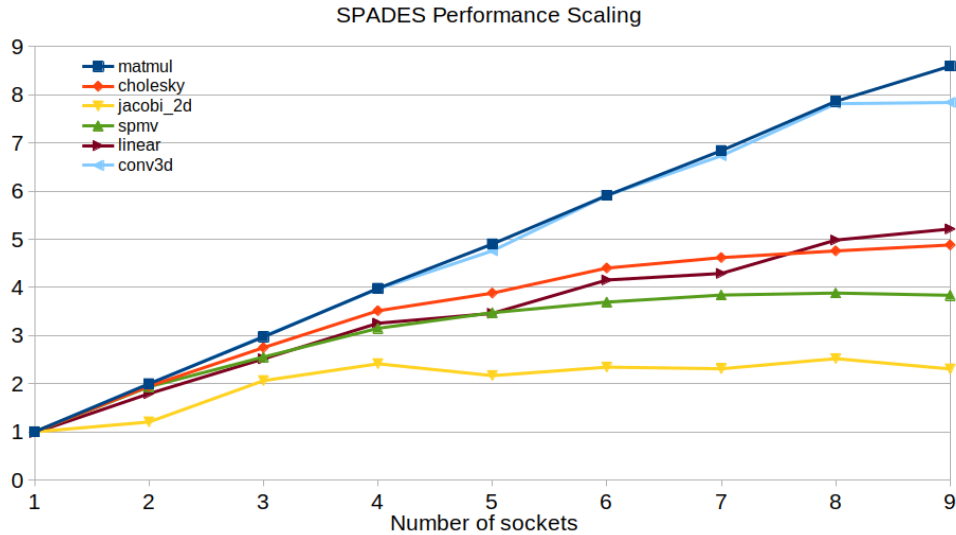
Figure 7.7: SPADES System Performance scaling concerning the number of sockets in the design

the case of *matmul*, having more sockets will lead to a better performance. Additionally, higher FMax helps the performance due to the improved compute latency. In contrast, for memory-dominant benchmarks, neither utilizing more sockets nor better FMax helps the performance when they saturate the external memory bandwidth. In that case, one could attempt to redesign the benchmarks at the core level, either by reducing extra memory requests, or more effective overlapping computation and communication to hide the memory latency (e.g., by working with a larger on-chip tile size).

7.7 Mapping complex applications

This section presents some preliminary results on using SPADES to map a complex application. The application is a small Convolutional Neural Network (CNN) consisting of two *conv3d* layers followed by three *linear* layers. The specification of the layers is given in Table 7.7. All layers are designed with 32-bit floating-point datatype. We utilize the same *conv3d* and *linear* sockets discussed in the previous sections to generate a complete design for this application. A design contains a certain number of sockets for *conv3d* and some for *linear*. For example, out of 9 sockets, we can allocate 4 to accelerate *conv3d* layers, and 5 for *linear* layers. Different configurations will lead to different performance results. For example, if the *conv3d* layers are more dominant in the CNN, utilizing more *conv3d* sockets will result in better performance. In each layer of the CNN, relevant sockets will execute it in parallel by dividing the workload among those sockets. The socket will need to synchronize
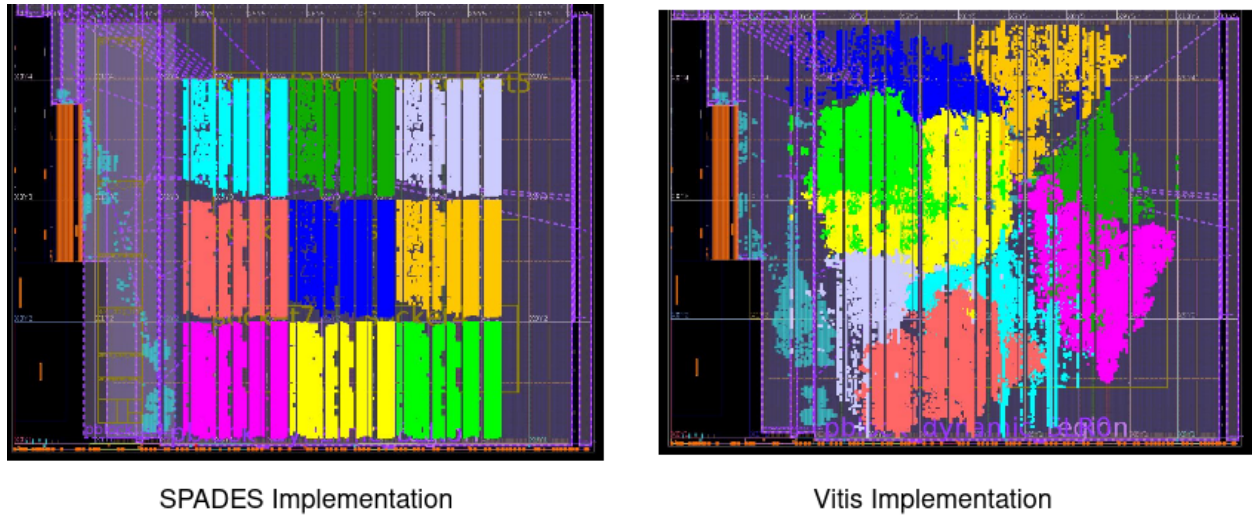
Figure 7.8: SPADES and Vitis Implementations for Configuration {4x *conv3d*, 5x *linear*}

before moving on to execute the next layer since we need to ensure that the output feature map result from the current layer is fully written back to the DRAM. The *linear* sockets remain inactive during the execution of the *conv3d* sockets and become active when the next layer is *linear*. There needs to be some coordination between the sockets to transfer the flow of execution. We can have a master socket for each type. A master socket synchronizes with sockets of the same types, and with the other master counterpart. At the design assembly step, we need to specify the NoC connectivity accordingly regarding these control paths between the sockets. The coordination logic is done through the controller software in each socket. We use a similar approach when creating a design for this application in Vitis. The design contains Vitis HLS cores for *conv3d* and *linear*. However, the synchronization mechanism between the Vitis cores is more difficult to accomplish, since their designs do not adopt any NSUs for receiving synchronizing requests from a NoC client. In this case, we coordinate their executions in the host code. The overhead of coordination through the host pales in comparison to the execution time of a CNN layer. As an example, Figure 7.8 demonstrates the implementation results of SPADES and Vitis for a particular design configuration.

Table 7.8 presents the performance comparison between SPADES and Vitis for different configurations of SPADES sockets (or Vitis cores). SPADES designs consistently produce better performance. This is attributed to the higher achievable FMax in SPADES implementation. Table 7.9 shows the FMax of the SPADES and Vitis designs in different configurations. We can observe that the FMax values of SPADES designs are similar in all configurations; in fact, it is the FMax of a *conv3d* socket. On the other hand, the FMax of Vitis designs varies from one configuration to another. Therefore, one can predict the final

Table 7.7: Small CNN layer specification

Layer	Input feature map dimension	Weight dimension	Output feature map dimension
conv3d0	192x13x13	384x192x3x3	384x13x13
conv3d1	384x13x13	256x384x3x3	256x13x13
linear0	45056	45056x4096	4096
linear1	4096	4096x4096	4096
linear2	4096	4096x1000	1000

Table 7.8: Performance Comparison

Configuration	SPADES exec. time (us)	Vitis exec. time (us)	SPADES Improvement
4x conv3d, 5x linear	85871	91155	5.8%
3x conv3d, 6x linear	72412	80690	10.3%
5x conv3d, 4x linear	98618	102164	3.5%

Table 7.9: FMax Comparison

Configuration	SPADES FMax (MHz)	Vitis FMax (MHz)	SPADES Improvement
4x conv3d, 5x linear	427	321	33%
3x conv3d, 6x linear	427	307	39.1%
5x conv3d, 4x linear	427	340	25.6%

FMax of a SPADES design that contains heterogeneous sockets by the slowest socket (i.e., the one whose FMax value is the lowest).

Next, we examine the compile time taken by SPADES and Vitis to generate designs for this application under different configurations as shown in Table 7.10. SPADES significantly outperforms Vitis in compile time with up to 42x speedup. Note that we reuse the physical implementations of *conv3d* and *linear* sockets to generate a different socket composition per configuration. Therefore, the SPADES compile time is essentially the time spent on design assembly (Step3) and NoC routing. This is possible because our backend flow supports reusability at the *physical netlist* level. On the other hand, although Vitis does reuse the synthesized results of the *conv3d* and *linear* HLS cores, it still has to perform placement and routing of those cores altogether, which results in a longer compile time.

This demonstration of mapping a complex application highlights the three primary fea-

Table 7.10: Compile time Comparison

Configuration	SPADES compile time (s)	Vitis compile time (s)	SPADES Speedup
4x conv3d, 5x linear	227	8098	35.7x
3x conv3d, 6x linear	223	7084	31.8x
5x conv3d, 4x linear	229	9525	42x

tures of SPADES: modularity, composability, and reusability. Consequently, they lead to a better achievable FMax (and performance) in addition to shorter compile time.

Chapter 8

Conclusions

8.1 Concluding Remarks

Modern FPGAs have been growing larger than ever with the introduction of heterogeneous hard blocks and multi-die programmable fabrics. At the same time, emerging applications have become increasingly demanding in compute and memory bandwidth. However, the top-down compilation approach employed by traditional vendor tools are no longer able to cope with the programmable device and design scalability. Therefore, we need to rethink and redesign the *tool flow* to manage such complexity effectively.

This work is among many that aim to address such challenges. We present an end-to-end tool flow from application to placed-and-routed netlist targeting a NoC-embedded FPGA architecture. Co-design with the target architecture, our flow exploits the application domain's modularity and reusability in tandem with hardware/software partitioning techniques to design and implement a system of parallel, distributed ensemble of sockets (SPADES). We achieve a 7.1x speedup in compile time in comparison to the standard vendor tool flow on several benchmarks from different application domains. Particularly, our flow finishes the compilation in 30 minutes or less instead of 2.5-3 hours. Our flow also yields better achievable frequency, and slightly more resource utilization for FF and LUT. More importantly, we demonstrate that SPADES produces working and functionally correct board-level executions, thus implying its practicality. Our system-level performance is comparable to Vitis multi-core designs across different benchmarks. The experimental platform is AMD Xilinx Versal VCK5000 data center card.

8.2 Future Explorations

The front-end application mapping and optimization are currently done manually. In particular, we need to extract the communication and control logic from the application, since those would be managed by the controller software running on the socket's softcore. Next, we also have to determine the appropriate amount of on-chip RAM blocks and their

connectivity with the compute logic and the communication part. We then design the compute logic and optimize its latency by doing loop optimizations such as unrolling and pipelining. The control software plays a key role in the performance of a socket; this is the place in which we schedule the communication and computation. An effective design would attempt to overlap the communication and computation latency as well as the control overhead. Hence, future work involves automation of this process to relieve the burden on users. We think the MLIR (Multi-level Intermediate Representation) compiler framework [54] could facilitate the front-end development of this work. MLIR offers multiple abstraction levels through operations organized in different *Dialects*. The dialects could be as high to the application level (such as neural network graphs, affine loop nests, etc.), or as close to modeling specific hardware targets (such as NVIDIA GPUs, AMD AI Engines [68, 52], etc.). The advantage of using MLIR is that one could have different abstraction representations (operations) coexist in a single IR, thus enabling holistic optimizations that require views from different levels. In our SPADES model, we have the entities such as the control unit, memory logic, and compute unit where each entails distinct code or design generation within a socket. If we look across the system, there are multiple sockets that may or may not share similar designs. The multi-level, multi-entity of our model fits nicely with the MLIR framework. We could leverage the framework to build code generation capability, as well as make use of higher-level dialects to capture the structure and design intent to facilitate the extraction of control, memory, and compute logic.

The compile time of Socket CL is one major area for improvement if we would like to further reduce the total compile time. In that regard, optimizing the place and route runtime of Socket CL likely leads to better compile time, since they are the dominant factors. We could build a library of pre-compiled Socket CL netlists to effectively eliminate the place and route time. For operations that are frequently used such as matrix multiplication, Fast Fourier Transforms, etc., one could design and pre-implement well-optimized Socket CLs for those operations. Therefore, when composing an application, we could select from the library which operations needed by the application. This practice is heavily adopted in the software world where we usually find libraries of highly optimized linear algebra kernels such as BLAS, and in some cases, the libraries could be distributed in binary form which does not require recompilation. But that does not mean hardware vendors never provide any user libraries. However, the libraries are usually at RTL level, thus requiring further compilation. Our tool flow, on the other hand, could compose designs at the netlist level, thereby improving the reusability of the libraries. Portability, on the other hand, is not guaranteed if we try to use the netlist on a different FPGA architecture (or even the same architecture, but with incompatible programmable fabric). Portability in FPGA design flows requires further research.

The partitioning of the socket floorplan to Socket CC and Socket CL unfortunately leads to some unused fabric regions, particularly at the joint boundary of the two partitions. Our initial goal is devising a partition that ensures straightforward re-combination in a later step, then concern about the QoR next. The weakness of the approach, besides wasting some fabric resources as mentioned above, is reducing the routing resource available to Socket CL,

thus rendering routing congestion for some benchmarks. The FF bridges, mainly to reduce the timing paths between the partitions, also incur additional delays in the communication of the partitions. We plan to investigate a better partitioning approach. We think it may benefit to use a custom placer and router in RapidWright (such as RWRoute) to assist us with this task. The custom tools would give us more flexibility, especially when dealing with the potential routing conflicts between Socket CC and Socket CL.

Although we utilize multiple clock buffers, one for each socket instance, in our full design, the clock buffers are all driven by the same clock generator. This happens by the design of the shell, not the limitation of our tool flow. It is generally not an issue. But imagine a situation in which there are two sockets, A and B in the design of which A could only meet a lower timing target than B. Since their clocks are driven by the same source, A and B will be clocked at the frequency that A could meet, thereby degrading the performance of B. In this case, having A and B operating on different frequencies would likely yield a better system performance. This raises the question of whether we would need to handle the clock mismatch (clock-domain crossing) between different sockets. It is not necessary, since a socket primarily interfaces with the embedded NoC. The NoC runs on its own frequency and has its clock-domain crossing logic to handle the clock rate difference with the PL modules. Another way to look at this is that a socket is well isolated by the NoC. Thus, an interesting future work would be redesigning the shell to utilize different clock generators to drive the sockets' clock buffers to enable more heterogeneous clock frequency configurations for the ensemble of sockets.

The design flow that we adopt in this work is a reconfigurable flow by vendor design. Concretely, there is a fixed base platform shell and the reconfigurable user-logic partition. Our user-logic partition contains the socket manager from the static logic and the instances in socket regions. Thus, everytime when we download a bitstream to the card, we basically reconfigure the whole user partition. In some cases, it might be advantageous to perform reconfiguration at a smaller scale, such as socket level, especially if we only would like to change the implementation of one or a few sockets in the system. This would require us to further divide the user partition to smaller reconfigurable regions for partial reconfiguration.

In this work, we omit the runtime of bitstream generation from the compile time. On complex devices, generating a bitstream could require tens of minutes to hours to finish. This is an area for future improvement. We could apply the same principles as employed in our tool flow so far here: modularity and reusability. Regarding modularity, we could generate multiple bitstreams instead of a monolithic one, each for a different socket. We could also reuse a bitstream if there are multiple sockets with similar designs; we then rely on custom backend tool such as RapidWright to update a socket's bitstream according to its relocation referenced to the original netlist.

We also do not consider the *link_design* step in the total compile time. This step essentially combines our user logic with the vendor shell design checkpoint. Depending on the target device, the shell design checkpoint could be enormous, and might involve heavy disk and memory usage for the vendor software (Vivado). In our flow, this step usually takes around 10-15 minutes. Spending such an amount of time without doing any actual useful

work is mind-boggling. Unfortunately, the vendor tool mandates this step to produce a valid implementation for NoC routing, QoR reports (timing, area utilization, DRC) and bitstream generation. However, at this point, there is no particular information from the shell that is useful or important to the SPADES design. Therefore, we could attempt to create an abstract, minimal "stub" shell that mimics the interface of the real shell while allowing legal link result and faster link time with the SPADES user logic.

Lastly, although we take advantage of the hardened NoC available in our target device, our approach could also work with FPGA platforms that do not have a NoC. We could virtualize an FPGA with a soft NoC. To ensure a better QoR, we might have to redesign our data movement logic. For example, we could use a simpler communication interface than AXI to reduce area utilization. We could also make use of direct connections between sockets to reduce the traffic jam on the NoC.

8.3   Reflections and Lessons Learned

Best practices in digital design never get old. Modularity, composability, and reusability are the guiding design principles that I have used throughout the development of SPADES. Modularity concerns breaking a problem into smaller sub-problems, or designing submodules from a complex, large module. Composability ensures ease of integration of those submodules. Reusability facilitates instantiating several similar submodules to solve a problem. Imagine working on a complex hardware design project that involve multiple teams working on different submodules. These principles help guide the development of each submodule in isolation and parallel with each other, thereby minimizing the chance of arising conflicts and improving productivity. One could update a submodule without impacting the rest of the design. Yet, in my opinion, it is very challenging to come up with a perfect partitioning technique when modularizing an application. Some submodules could be very large, while other trivial. Often, it is more manageable to redesign an application itself to fit into a framework (or model) that has a better support for modularity and composability, than trying to divide up an existing design or implementation of the application. This leads to the SPADES model that I have introduced in this thesis.

I also find that it is more effective to build a design flow by thinking from the bottom up. That is to take into account the target device's architectural characteristics, and devise methods built around them that aim to utilize these features. The next step is to find an appropriate high-level representation which closely captures the structure of the low-level device characteristics to facilitate efficient mapping. It is the structural correlation between the design hierarchy and the device organization that I would like to maintain throughout the design flow. Approaching the problem in a top-down manner might risk losing the design hierarchy due to a design flattening carried out by the higher-level tooling optimizations. I would argue that maintaining the design hierarchy is the key to achieve faster compile time as well as better performance in most cases. This point is corroborated by the results present in this thesis. It remains to be seen whether the model is also applicable to a

wider range of applications. One may think that this approach is restrictive in ways that limit the exploration of the design space or risk under-utilizing the resource of the target device. However, I think it is important to make a tradeoff between specialty (this flow) and generality (standard flow) in some situations, especially if productivity is of utmost concern. At the same time, sacrificing tooling generality does not always lead to losing performance. My experience is that the placer and router are effective when dealing with small and moderate-sized netlists. Yet, they might yield sub-optimal results as the input designs get increasingly complex. This is because Place and Route are NP-compute problems, and they need to rely on certain heuristics to reach a solution within a reasonable time. By organizing the device floorplan into smaller sections such as socket regions, and then invoking the tools to solve P&R at a smaller scale, there is an opportunity for QoR improvement.

Real devices might present unforeseeable engineering problems to a research model or prototype. I think it is important that the research prototype is working on a real hardware. That would demonstrate the practicality of the tool flow, and allow us to collect data on much bigger problem sizes than doing RTL simulations. Addressing these engineering challenges might be daunting and tedious, yet they give us valuable insights on how to build future architectures and tools more effectively. Specifically, in this work, we need to floorplan the device to maximize the opportunity for netlist relocatability. The more compatible fabric regions we can get, the better flexibility we can achieve in terms of deciding the placement of a socket. Note that although SPADES does compile a socket to a predetermined region (floorplan), the final placement of the socket on the fabric is decided later when SPADES performs design assembly. In other words, there is a virtual placement that results in a location-agnostic socket implementation. Therefore, future FPGA fabrics should be built with ample regularity. This will give us more freedom in choosing the placement locations.

In addition, SPADES routes the clock and reset signals once the sockets are assembled, since they are compiled separately. The clocks need to be routed from the clock buffers that are usually located at the edge of the device to the socket regions, and the resets are routed from another region (static logic). While the global routing of these signals does not create any major bottleneck in this work, it will be more effective if there is an immediate access point to the clock and reset from within a socket region. For example, one of the NoC endpoints (NMU or NSU) of a socket region can supply a clock and reset signal to the fabric cells in the region, and the mechanism of controlling the NoC endpoint's clock and reset can be done via another NoC client. Thus, we can avoid spending the fabric routing resource to route these global signals for each socket region.

Another potential future device improvement that will work well with SPADES is augmenting the NoC endpoints with more capability. Recall that Socket CC incurs some overhead in control due to the soft CPU core, and its functionality is fixed. We can make Socket CC a hardened block — either separate or together with a NoC endpoint to produce a "smart NoC" hard block. A smart NoC will have a scalar control processor, a DMA engine, and an LSU. This will lead to better resource utilization of the fabric if we implement Socket CC using ASIC technology. An ASIC version of Socket CC is also likely able to operate at a higher frequency, thereby improving performance efficiency. A custom logic can interface

with Socket CC hard block via its LSU ports using a similar approach as presented in this work. However, the custom logic region should have more routing resource in this case to mitigate the routing congestion issue.

To further extend the idea of smart NoC, one could also adopt this flow for rapid prototyping and designing ASIC accelerators. An ASIC template that consists of bare socket floorplans in which each has one or several smart NoCs could be provided initially. Each accelerator is designed separately that fit in one or many socket regions, and the tool will handle the composition of the accelerators based on a design specification. The accelerators could communicate with one another and the memory subsystems through the smart NoCs. By designing the accelerators in isolation and relying on a template for composition and communication infrastructure, it will fasten the design cycle. It would be interesting to conduct a study on evaluating the QoR of designs generated by such flow against the standard top-down ASIC flow similar to what we have demonstrated for FPGA designs.

Bibliography

[1]   *ABC.* https://people.eecs.berkeley.edu/~alanmi/abc/abc.htm.

[2]   Mohamed S. Abdelfattah and Vaughn Betz. "The Case for Embedded Networks on Chip on Field-Programmable Gate Arrays". In: *IEEE Micro* 34.1 (2014), pp. 80–89. DOI: 10.1109/MM.2013.131.

[3]   Mohamed Saleh Abdelfattah, David Han, Andrew Bitar, Roberto Dicecco, Shane O'Connell, Nitika Shanker, Joseph Chu, Ian Prins, Joshua Fender, Andrew C. Ling, and Gordon R. Chiu. "DLA: Compiler and FPGA Overlay for Neural Network Inference Acceleration". In: *2018 28th International Conference on Field Programmable Logic and Applications (FPL)* (2018), pp. 411–4117. URL: https://api.semanticscholar.org/CorpusID:49864686.

[4]   *Achronix FPGA 2D-NoC.* https://www.achronix.com/revolutionary-new-2d-network-chip.

[5]   *AMD Vitis.* https://www.xilinx.com/products/design-tools/vitis.html.

[6]   *ARM AMBA AXI Protocol.* https://developer.arm.com/documentation/ihi0022/b.

[7]   *Autonomous Driving and Advanced Driver Assistance Systems (ADAS).* https://www.intel.com/content/www/us/en/automotive/products/programmable/applications.html.

[8]   Utku Aydonat, Shane O'Connell, Davor Capalija, Andrew C. Ling, and Gordon R. Chiu. "An OpenCL™ Deep Learning Accelerator on Arria 10". In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays.* FPGA '17. Monterey, California, USA: Association for Computing Machinery, 2017, pp. 55–64. ISBN: 9781450343541. DOI: 10.1145/3020078.3021738. URL: https://doi.org/10.1145/3020078.3021738.

[9]   Benjamin L.C. Barzen, Arya Reais-Parsi, Eddie Hung, Minwoo Kang, Alan Mishchenko, Jonathan W. Greene, and John Wawrzynek. "Narrowing the Synthesis Gap: Academic FPGA Synthesis is Catching Up With the Industry". In: *2023 Design, Automation and Test in Europe Conference and Exhibition (DATE).* 2023, pp. 1–6. DOI: 10.23919/DATE56975.2023.10137310.

[10]   *Cadence: Emulation and Prototyping.* https://www.cadence.com/en_US/home/
        tools/system-design-and-verification/emulation-and-prototyping.html.

[11]   Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz
        Czajkowski, Stephen D. Brown, and Jason H. Anderson. "LegUp: An Open-Source
        High-Level Synthesis Tool for FPGA-Based Processor/Accelerator Systems". In: 13.2
        (2013). ISSN: 1539-9087. DOI: 10.1145/2514740. URL: https://doi.org/10.1145/
        2514740.

[12]   Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fow-
        ers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim,
        Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods,
        Sitaram Lanka, Derek Chiou, and Doug Burger. "A cloud-scale acceleration archi-
        tecture". In: *2016 49th Annual IEEE/ACM International Symposium on Microarchi-
        tecture (MICRO)*. 2016, pp. 1–13. DOI: 10.1109/MICRO.2016.7783710.

[13]   Yao Chen, Jiong He, Xiaofan Zhang, Cong Hao, and Deming Chen. "Cloud-DNN:
        An Open Framework for Mapping DNN Models to Cloud FPGAs". In: *Proceedings of
        the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays.*
        FPGA '19. Seaside, CA, USA: Association for Computing Machinery, 2019, pp. 73–82.
        ISBN: 9781450361378. DOI: 10.1145/3289602.3293915. URL: https://doi.org/10.
        1145/3289602.3293915.

[14]   Shaoyi Cheng, Qijing Huang, and John Wawrzynek. "Synthesis of program binaries
        into FPGA accelerators with runtime dependence validation". In: *2017 International
        Conference on Field Programmable Technology (ICFPT)*. 2017, pp. 96–103. DOI: 10.
        1109/FPT.2017.8280126.

[15]   Shaoyi Cheng and John Wawrzynek. "High Level Synthesis with a Dataflow Architec-
        tural Template". In: (June 2016).

[16]   Yuze Chi, Jason Cong, Peng Wei, and Peipei Zhou. "SODA: Stencil with Optimized
        Dataflow Architecture". In: *2018 IEEE/ACM International Conference on Computer-
        Aided Design (ICCAD)*. 2018, pp. 1–8. DOI: 10.1145/3240765.3240850.

[17]   Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield,
        Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Maleen
        Abeydeera, Logan Adams, Hari Angepat, Christian Boehn, Derek Chiou, Oren Firestein,
        Alessandro Forin, Kang Su Gatlin, Mahdi Ghandi, Stephen Heil, Kyle Holohan, Ah-
        mad El Husseini, Tamas Juhasz, Kara Kagi, Ratna K. Kovvuri, Sitaram Lanka, Friedel
        van Megen, Dima Mukhortov, Prerak Patel, Brandon Perez, Amanda Rapsang, Steven
        Reinhardt, Bita Rouhani, Adam Sapek, Raja Seera, Sangeetha Shekar, Balaji Sridha-
        ran, Gabriel Weisz, Lisa Woods, Phillip Yi Xiao, Dan Zhang, Ritchie Zhao, and Doug
        Burger. "Serving DNNs in Real Time at Datacenter Scale with Project Brainwave".
        In: *IEEE Micro* 38.2 (2018), pp. 8–20. DOI: 10.1109/MM.2018.022071131.

[18] Eric S. Chung, James C. Hoe, and Ken Mai. "CoRAM: An in-Fabric Memory Architecture for FPGA-Based Computing". In: FPGA '11. Monterey, CA, USA: Association for Computing Machinery, 2011, pp. 97–106. ISBN: 9781450305549. DOI: 10.1145/1950413.1950435. URL: https://doi.org/10.1145/1950413.1950435.

[19] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. "High-Level Synthesis for FPGAs: From Prototyping to Deployment". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30.4 (2011), pp. 473–491. DOI: 10.1109/TCAD.2011.2110592.

[20] Philippe Coussy and Adam Morawiec. *High-Level Synthesis: From Algorithm to Digital Circuits.* June 2008.

[21] W.J. Dally and B. Towles. "Route packets, not wires: on-chip interconnection networks". In: *Proceedings of the 38th Design Automation Conference.* 2001, pp. 684–689.

[22] André DeHon. "Balancing Interconnect and Computation in a Reconfigurable Computing Array (or, Why You Don't Really Want 100% LUT Utilization)". In: *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays.* FPGA '99. Monterey, California, USA: Association for Computing Machinery, 1999, pp. 69–78. ISBN: 1581130880. DOI: 10.1145/296399.296431. URL: https://doi.org/10.1145/296399.296431.

[23] R.H. Dennard, F.H. Gaensslen, Hwa-Nien Yu, V.L. Rideout, E. Bassous, and A.R. LeBlanc. "Design of ion-implanted MOSFET's with very small physical dimensions". In: *IEEE Journal of Solid-State Circuits* 9.5 (1974), pp. 256–268. DOI: 10.1109/JSSC.1974.1050511.

[24] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. "A Configurable Cloud-Scale DNN Processor for Real-Time AI". In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA).* 2018, pp. 1–14. DOI: 10.1109/ISCA.2018.00012.

[25] Brian Gaide, Dinesh Gaitonde, Chirag Ravishankar, and Trevor Bauer. "Xilinx Adaptive Compute Acceleration Platform: VersalTM Architecture". In: *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays.* FPGA '19. Seaside, CA, USA: Association for Computing Machinery, 2019, pp. 84–93. ISBN: 9781450361378. DOI: 10.1145/3289602.3293906. URL: https://doi.org/10.1145/3289602.3293906.

[26] Licheng Guo, Yuze Chi, Jie Wang, Jason Lau, Weikang Qiao, Ecenur Ustun, Zhiru Zhang, and Jason Cong. "AutoBridge: Coupling Coarse-Grained Floorplanning and Pipelining for High-Frequency HLS Design on Multi-Die FPGAs". In: *Proceedings of the 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays.*

FPGA '21. Virtual Event, USA: Association for Computing Machinery, 2021, pp. 81–92. ISBN: 9781450382182. DOI: 10.1145/3431920.3439289. URL: https://doi.org/10.1145/3431920.3439289.

[27] Licheng Guo, Pongstorn Maidee, Yun Zhou, Chris Lavin, Jie Wang, Yuze Chi, Weikang Qiao, Alireza Kaviani, Zhiru Zhang, and Jason Cong. "RapidStream: Parallel Physical Implementation of FPGA HLS Designs". In: *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '22. Virtual Event, USA: Association for Computing Machinery, 2022, pp. 1–12. ISBN: 9781450391498. DOI: 10.1145/3490422.3502361. URL: https://doi.org/10.1145/3490422.3502361.

[28] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, Huazhong Yang, and William (Bill) J. Dally. "ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA". In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '17. Monterey, California, USA: Association for Computing Machinery, 2017, pp. 75–84. ISBN: 9781450343541. DOI: 10.1145/3020078.3021745. URL: https://doi.org/10.1145/3020078.3021745.

[29] Yuwei Hu, Yixiao Du, Ecenur Ustun, and Zhiru Zhang. "GraphLily: Accelerating Graph Linear Algebra on HBM-Equipped FPGAs". In: *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 2021, pp. 1–9. DOI: 10.1109/ICCAD51958.2021.9643582.

[30] Sitao Huang, Kun Wu, Hyunmin Jeong, Chengyue Wang, Deming Chen, and Wen-Mei Hwu. "PyLog: An Algorithm-Centric Python-Based FPGA Programming and Synthesis Flow". In: *IEEE Transactions on Computers* 70.12 (2021), pp. 2015–2028. DOI: 10.1109/TC.2021.3123465.

[31] Eddie Hung. "Mind the (synthesis) gap: Examining where academic FPGA tools lag behind industry". In: *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*. 2015, pp. 1–4. DOI: 10.1109/FPL.2015.7294007.

[32] *Intel FPGA System Interconnect*. https://www.intel.com/content/www/us/en/docs/programmable/683711/21-2/system-interconnect-39578.html.

[33] *Intel High-level Synthesis*. https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html.

[34] *Intel Quartus*. https://www.intel.com/content/www/us/en/products/details/fpga/development-tools/quartus-prime.html.

[35] *Intel Stratix 10*. https://www.intel.com/content/www/us/en/products/sku/210290/intel-stratix-10-gx-10m-fpga/specifications.html.

[36] Abhishek Kumar Jain, Xiangwei Li, Suhaib A. Fahmy, and Douglas L. Maskell. "Adapting the DySER Architecture with DSP Blocks as an Overlay for the Xilinx Zynq". In: *SIGARCH Comput. Archit. News* 43.4 (2016), pp. 28–33. ISSN: 0163-5964. DOI: 10.1145/2927964.2927970. URL: https://doi.org/10.1145/2927964.2927970.

[37]   Abhishek Kumar Jain, Douglas L. Maskell, and Suhaib A. Fahmy. "Coarse Grained
       FPGA Overlay for Rapid Just-In-Time Accelerator Compilation". In: *IEEE Transac-
       tions on Parallel and Distributed Systems* 33.6 (2022), pp. 1478–1490. DOI: `10.1109/
       TPDS.2021.3116859`.

[38]   Abhishek Kumar Jain, Chirag Ravishankar, Hossein Omidian, Sharan Kumar, Maithilee
       Kulkarni, Aashish Tripathi, and Dinesh Gaitonde. "Modular and Lean Architecture
       with Elasticity for Sparse Matrix Vector Multiplication on FPGAs". In: *2023 IEEE
       31st Annual International Symposium on Field-Programmable Custom Computing Ma-
       chines (FCCM)*. 2023, pp. 133–143. DOI: `10.1109/FCCM57271.2023.00023`.

[39]   Lana Josipović, Radhika Ghosal, and Paolo Ienne. "Dynamically Scheduled High-Level
       Synthesis". In: *Proceedings of the 2018 ACM/SIGDA International Symposium on
       Field-Programmable Gate Arrays*. FPGA '18. Monterey, CALIFORNIA, USA: Associ-
       ation for Computing Machinery, 2018, pp. 127–136. ISBN: 9781450356145. URL: `https:
       //doi.org/10.1145/3174243.3174264`.

[40]   Nachiket Kapre and Jan Gray. "Hoplite: Building austere overlay NoCs for FPGAs".
       In: *2015 25th International Conference on Field Programmable Logic and Applications
       (FPL)*. 2015, pp. 1–8. DOI: `10.1109/FPL.2015.7293956`.

[41]   Nachiket Kapre and Tushar Krishna. "FastTrack: Leveraging Heterogeneous FPGA
       Wires to Design Low-Cost High-Performance Soft NoCs". In: *2018 ACM/IEEE 45th
       Annual International Symposium on Computer Architecture (ISCA)*. 2018, pp. 739–
       751. DOI: `10.1109/ISCA.2018.00067`.

[42]   Nachiket Kapre, Nikil Mehta, Michael deLorimier, Raphael Rubin, Henry Barnor,
       Michael J. Wilson, Michael Wrighton, and Andre DeHon. "Packet Switched vs. Time
       Multiplexed FPGA Overlay Networks". In: *2006 14th Annual IEEE Symposium on
       Field-Programmable Custom Computing Machines*. 2006, pp. 205–216. DOI: `10.1109/
       FCCM.2006.55`.

[43]   R. Kastner, J. Matai, and S. Neuendorffer. "Parallel Programming for FPGAs". In:
       ArXiv e-prints (May 2018). arXiv: `1805.03648`.

[44]   Chris Lavin and Alireza Kaviani. "RapidWright: Enabling Custom Crafted Imple-
       mentations for FPGAs". In: *2018 IEEE 26th Annual International Symposium on
       Field-Programmable Custom Computing Machines (FCCM)*. 2018, pp. 133–140. DOI:
       `10.1109/FCCM.2018.00030`.

[45]   Ilia Lebedev, Christopher Fletcher, Shaoyi Cheng, James Martin, Austin Doupnik,
       Daniel Burke, Mingjie Lin, and John Wawrzynek. "Exploring Many-Core Design Tem-
       plates for FPGAs and ASICs". In: *Int. J. Reconfig. Comput.* 2012 (2012). ISSN: 1687-
       7195. URL: `https://doi.org/10.1155/2012/439141`.

[46] Cheng Liu, Ho-Cheung Ng, and Hayden Kwok-Hay So. "QuickDough: A rapid FPGA loop accelerator design framework using soft CGRA overlay". In: *2015 International Conference on Field Programmable Technology (FPT)*. 2015, pp. 56–63. DOI: `10.1109/FPT.2015.7393130`.

[47] Leo Liu, Jay Weng, and Nachiket Kapre. "RapidRoute: Fast Assembly of Communication Structures for FPGA Overlays". In: *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2019, pp. 61–64. DOI: `10.1109/FCCM.2019.00018`.

[48] Sihao Liu, Jian Weng, Dylan Kupsh, Atefeh Sohrabizadeh, Zhengrong Wang, Licheng Guo, Jiuyang Liu, Maxim Zhulin, Rishabh Mani, Lucheng Zhang, Jason Cong, and Tony Nowatzki. "OverGen: Improving FPGA Usability through Domain-specific Overlay Generation". In: *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2022, pp. 35–56. DOI: `10.1109/MICRO56248.2022.00018`.

[49] Xinheng Liu, Yao Chen, Tan Nguyen, Swathi Gurumani, Kyle Rupnow, and Deming Chen. "High Level Synthesis of Complex Applications: An H.264 Video Decoder". In: FPGA '16. Monterey, California, USA: Association for Computing Machinery, 2016, pp. 224–233. ISBN: 9781450338561. DOI: `10.1145/2847263.2847274`. URL: `https://doi.org/10.1145/2847263.2847274`.

[50] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. "Optimizing Loop Operation and Dataflow in FPGA Acceleration of Deep Convolutional Neural Networks". In: FPGA '17. Monterey, California, USA: Association for Computing Machinery, 2017, pp. 45–54. ISBN: 9781450343541. DOI: `10.1145/3020078.3021736`. URL: `https://doi.org/10.1145/3020078.3021736`.

[51] Dan McNamara. *FPGA vs ASIC: 5G changes the equation.* `https://www.xilinx.com/publications/reports/mobile-experts-market-report.pdf`.

[52] *MLIR AIE.* `https://github.com/Xilinx/mlir-aie`.

[53] G.E. Moore. "Cramming More Components Onto Integrated Circuits". In: *Proceedings of the IEEE* 86.1 (1998), pp. 82–85. DOI: `10.1109/JPROC.1998.658762`.

[54] *Multi-level Intermediate Representation.* `https://mlir.llvm.org/`.

[55] Kevin E. Murray, Oleg Petelin, Sheng Zhong, Jia Min Wang, Mohamed Eldafrawy, Jean-Philippe Legault, Eugene Sha, Aaron G. Graham, Jean Wu, Matthew J. P. Walker, Hanqing Zeng, Panagiotis Patros, Jason Luu, Kenneth B. Kent, and Vaughn Betz. "VTR 8: High-Performance CAD and Customizable FPGA Architecture Modelling". In: *ACM Trans. Reconfigurable Technol. Syst.* 13.2 (2020). ISSN: 1936-7406. DOI: `10.1145/3388617`. URL: `https://doi.org/10.1145/3388617`.

[56] Marie Nguyen and James C. Hoe. "Time-Shared Execution of Realtime Computer Vision Pipelines by Dynamic Partial Reconfiguration". In: *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. 2018, pp. 230–2304. DOI: `10.1109/FPL.2018.00046`.

[57] Tan Nguyen, Zachary Blair, Stephen Neuendorffer, and John Wawrzynek. "SPADES: A Productive Design Flow for Versal Programmable Logic". In: *2023 33rd International Conference on Field-Programmable Logic and Applications (FPL)*. 2023, pp. 65–71. DOI: 10.1109/FPL60245.2023.00017.

[58] Papakonstantinou, Alexandros and Gururaj, Karthik and Stratton, John A. and Chen, Deming and Cong, Jason and Hwu, Wen-Mei W. "FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs". In: *2009 IEEE 7th Symposium on Application Specific Processors*. 2009, pp. 35–42. DOI: 10.1109/SASP.2009.5226333.

[59] Dongjoon Park, Yuanlong Xiao, and André DeHon. "Fast and Flexible FPGA Development using Hierarchical Partial Reconfiguration". In: *2022 International Conference on Field-Programmable Technology (ICFPT)*. 2022, pp. 1–10. DOI: 10.1109/ICFPT56656.2022.9974201.

[60] Kyle Rupnow, Yun Liang, Yinan Li, Dongbo Min, Minh Do, and Deming Chen. "High level synthesis of stereo matching: Productivity, performance, and software constraints". In: *2011 International Conference on Field-Programmable Technology*. 2011, pp. 1–8. DOI: 10.1109/FPT.2011.6132716.

[61] David Shah, Eddie Hung, Clifford Wolf, Serge Bazanski, Dan Gisselquist, and Miodrag Milanovic. "Yosys+nextpnr: An Open Source Framework from Verilog to Bitstream for Commercial FPGAs". In: *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2019, pp. 1–4. DOI: 10.1109/FCCM.2019.00010.

[62] Linghao Song, Yuze Chi, Atefeh Sohrabizadeh, Young-kyu Choi, Jason Lau, and Jason Cong. "Sextans: A Streaming Accelerator for General-Purpose Sparse-Matrix Dense-Matrix Multiplication". In: *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '22. Virtual Event, USA: Association for Computing Machinery, 2022, pp. 65–77. ISBN: 9781450391498. DOI: 10.1145/3490422.3502357. URL: https://doi.org/10.1145/3490422.3502357.

[63] Srivatsan Srinivasan, Andrew Boutros, Fatemehsadat Mahmoudi, and Vaughn Betz. "Placement Optimization for NoC-Enhanced FPGAs". In: *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2023, pp. 41–51. DOI: 10.1109/FCCM57271.2023.00014.

[64] Ian Swarbrick, Dinesh Gaitonde, Sagheer Ahmad, Brian Gaide, and Ygal Arbel. "Network-on-Chip Programmable Platform in VersalTM ACAP Architecture". In: *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 212–221. ISBN: 9781450361378. DOI: 10.1145/3289602.3293908.

[65] *The LLVM Compiler Infrastructure*. https://llvm.org/.

[66] Frederick Tombs, Alireza Mellat, and Nachiket Kapre. "Mocarabe: High-Performance Time-Multiplexed Overlays for FPGAs". In: *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2021. DOI: `10.1109/FCCM51124.2021.00021`.

[67] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. "FINN: A Framework for Fast, Scalable Binarized Neural Network Inference". In: FPGA '17. Monterey, California, USA: Association for Computing Machinery, 2017, pp. 65–74. ISBN: 9781450343541. DOI: `10.1145/3020078.3021744`. URL: `https://doi.org/10.1145/3020078.3021744`.

[68] *Versal AI Engine*. `https://docs.xilinx.com/r/en-US/ug1273-versal-acap-design/AI-Engine`.

[69] *Versal Network-on-chip*. `https://docs.xilinx.com/r/en-US/pg313-network-on-chip`.

[70] *Vitis High-level Synthesis*. `https://www.xilinx.com/products/design-tools/vitis/vitis-hls.html`.

[71] *Vivado Dynamic Function Exchange*. `https://docs.xilinx.com/r/en-US/ug909-vivado-partial-reconfiguration/Introduction-to-Dynamic-Function-eXchange`.

[72] *Vivado: Launching Multiple Runs*. `https://docs.xilinx.com/r/en-US/ug904-vivado-implementation/Launching-Multiple-Runs`.

[73] Yuanlong Xiao, Syed Tousif Ahmed, and André DeHon. "Fast Linking of Separately-Compiled FPGA Blocks without a NoC". In: *2020 International Conference on Field-Programmable Technology (ICFPT)*. 2020, pp. 196–205. DOI: `10.1109/ICFPT51103.2020.00035`.

[74] Yuanlong Xiao, Aditya Hota, Dongjoon Park, and André DeHon. "HiPR: High-level Partial Reconfiguration for Fast Incremental FPGA Compilation". In: *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*. 2022, pp. 70–78. DOI: `10.1109/FPL57034.2022.00022`.

[75] Yuanlong Xiao, Eric Micallef, Andrew Butt, Matthew Hofmann, Marc Alston, Matthew Goldsmith, Andrew Merczynski-Hait, and André DeHon. "PLD: Fast FPGA Compilation to Make Reconfigurable Acceleration Compatible with Modern Incremental Refinement Software Development". In: *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '22. Lausanne, Switzerland: Association for Computing Machinery, 2022, pp. 933–945. ISBN: 9781450392051. DOI: `10.1145/3503222.3507740`. URL: `https://doi.org/10.1145/3503222.3507740`.

[76] Yuanlong Xiao, Dongjoon Park, Andrew Butt, Hans Giesen, Zhaoyang Han, Rui Ding, Nevo Magnezi, Raphael Rubin, and Andre DeHon. "Reducing FPGA Compile Time with Separate Compilation for FPGA Building Blocks". In: *2019 International Conference on Field-Programmable Technology (ICFPT)*. 2019, pp. 153–161. DOI: `10.1109/ICFPT47387.2019.00026`.

[77] *Xilinx 7-Series CLB*. `https://docs.xilinx.com/v/u/en-US/ug474_7Series_CLB`.

[78] *Xilinx Deep Learning Processor Unit*. `https://docs.xilinx.com/r/1.2-English/ug1414-vitis-ai/Deep-Learning-Processor-Unit-DPU`.

[79] *Xilinx Microblaze*. `https://www.xilinx.com/products/design-tools/microblaze.html`.

[80] *Xilinx UltraScale+ CLB*. `https://docs.xilinx.com/v/u/en-US/ug574-ultrascale-clb`.

[81] *Xilinx Versal CLB*. `https://docs.xilinx.com/r/en-US/am005-versal-clb`.

[82] *Xilinx VU19P*. `https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus-vu19p.html`.

[83] Yifan Yang, Qijing Huang, Bichen Wu, Tianjun Zhang, Liang Ma, Giulio Gambardella, Michaela Blott, Luciano Lavagno, Kees Vissers, John Wawrzynek, and Kurt Keutzer. "Synetgy: Algorithm-Hardware Co-Design for ConvNet Accelerators on Embedded FPGAs". In: FPGA '19. Seaside, CA, USA: Association for Computing Machinery, 2019, pp. 23–32. ISBN: 9781450361378. DOI: `10.1145/3289602.3293902`. URL: `https://doi.org/10.1145/3289602.3293902`.

[84] Hanchen Ye, Cong Hao, Jianyi Cheng, Hyunmin Jeong, Jack Huang, Stephen Neuendorffer, and Deming Chen. "ScaleHLS: A New Scalable High-Level Synthesis Framework on Multi-Level Intermediate Representation". In: *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 2022, pp. 741–755. DOI: `10.1109/HPCA53966.2022.00060`.

[85] Chi Zhang and Viktor Prasanna. "Frequency Domain Acceleration of Convolutional Neural Networks on CPU-FPGA Shared Memory System". In: FPGA '17. Monterey, California, USA: Association for Computing Machinery, 2017, pp. 35–44. DOI: `10.1145/3020078.3021727`.

[86] Jialiang Zhang and Jing Li. "Improving the Performance of OpenCL-Based FPGA Accelerator for Convolutional Neural Network". In: FPGA '17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 25–34. DOI: `10.1145/3020078.3021698`.

[87] Jieru Zhao, Liang Feng, Sharad Sinha, Wei Zhang, Yun Liang, and Bingsheng He. "Performance Modeling and Directives Optimization for High-Level Synthesis on FPGA". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.7 (2020), pp. 1428–1441. DOI: `10.1109/TCAD.2019.2912916`.

[88]   Ritchie Zhao, Weinan Song, Wentao Zhang, Tianwei Xing, Jeng-Hau Lin, Mani Srivastava, Rajesh Gupta, and Zhiru Zhang. "Accelerating Binarized Convolutional Neural Networks with Software-Programmable FPGAs". In: FPGA '17. Monterey, California, USA: Association for Computing Machinery, 2017, pp. 15–24. ISBN: 9781450343541. DOI: 10.1145/3020078.3021741. URL: https://doi.org/10.1145/3020078.3021741.

Appendix A

Controller Software

A.1   Matrix Multiply

```
1  for (w = SOCKET_ID; w < (N / BHA) * (N / BWB); w += NUM_CORES) {
2    i = w / (N / BWB);
3    j = w % (N / BWB);
4
5    // Setup DMA/LSU configuration for reading C to local_c
6    LSU0_RAM_START_IDX = 2 * BWA + 1;
7    LSU0_RAM_BLOCK_FACTOR = 1;
8    LSU0_RAM_CYCLIC_FACTOR = 4;
9    LSU0_EXT_MEM_OFFSET = &C(i + 0, j + 0);
10   LSU0_SEG_STRIDE = N;
11   LSU0_SEG_COUNT = BHA;
12   LSU0_LEN = BWB;
13   LSU0_MODE = 1; // read
14   // Start LSU0 operation
15   TQ_LSU0_START();
16   TQ_LSU0_DONE();
17
18   // Setup DMA/LSU configuration for reading A to local_a
19   LSU0_RAM_BLOCK_FACTOR = 2; // for dual-ported reads
20   LSU0_RAM_CYCLIC_FACTOR = BWA / 2;
21   LSU0_SEG_STRIDE = N;
22   LSU0_SEG_COUNT = BHA;
23   LSU0_LEN = BWA;
24   LSU0_MODE = 1; // read
25
26   // Setup DMA/LSU configuration for reading B to local_b
27   LSU1_RAM_BLOCK_FACTOR = BWB * 2; // for dual-ported reads
28   LSU1_RAM_CYCLIC_FACTOR = BWA / 2;
29   LSU1_SEG_STRIDE = N;
30   LSU1_SEG_COUNT = BHB;
31   LSU1_LEN = BWB;
32   LSU1_MODE = 1; // read
```

```
33
34   for (k = 0; k < N / BWA; k++) {
35     KRN_LEN = BWA;
36     LSU0_EXT_MEM_OFFSET = A(i + 0, k + 0);
37     LSU1_EXT_MEM_OFFSET = B(k + 0, i + 0);
38
39     // switching between ping and pong RAM blocks
40     if (pp == 0) {
41       KRN_PP = 2;
42       LSU0_RAM_START_IDX = 0;
43       LSU1_RAM_START_IDX = BWA;
44       pp = 1;
45     } else {
46       KRN_PP = 1;
47       LSU0_RAM_START_IDX = BWA / 2;
48       LSU1_RAM_START_IDX = BWA + (BWA / 2);
49       pp = 0;
50     }
51
52     // Start LSU0, LSU1, and CL operations concurrently
53     TQ_LSU0_START();
54     TQ_LSU1_START();
55     if (k != 0) {
56       // Do not start CL on the first iteration
57       KRN_START = 1;
58       TQ_CL_START();
59       TQ_CL_DONE();
60     }
61     TQ_LSU0_DONE();
62     TQ_LSU1_DONE();
63   }
64
65   // trailing CL run
66   KRN_PP = (pp == 0) ? 2 : 1
67   KRN_START = 1;
68   TQ_CL_START();
69   TQ_CL_DONE();
70
71   // Setup DMA/LSU configuration for writing C from local_c
72   LSU0_RAM_START_IDX = 2 * BWA + 1;
73   LSU0_RAM_BLOCK_FACTOR = 1;
74   LSU0_RAM_CYCLIC_FACTOR = 4;
75   LSU0_EXT_MEM_OFFSET = &C(i + 0, j + 0);
76   LSU0_SEG_STRIDE = N;
77   LSU0_SEG_COUNT = BHA;
78   LSU0_LEN = BWB;
79   LSU0_MODE = 2; // write
80   TQ_LSU0_START();
81   TQ_LSU0_DONE();
82 }
```

```
83
84 // Closing code
85 // wait until all tasks complete
86 while (TQ_EMPTY_N == 1);
87 // notify socket manager
88 CTRL_MAXI_SOCKET_OFFSET = SOCKET_MANAGER_NOC_ADDR + STATUS_COMPLETE_OFFSET
       (CORE_ID);
89 CTRL_MAXI_WRITE = 1;
90 while (CTRL_MAXI_WRITE_DONE == 0);
91 // assert socket_done signal
92 CPU_STATUS = 1;
```

Listing A.1: Optimized controller software for benchmark matmul