Leveraging Design-Time Abstractions for Accelerator Design and Integration

Martha Massee Barker

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
under the Executive Committee
of the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2025

Abstract

Leveraging Design-Time Abstractions for Accelerator Design and Integration

Martha Massee Barker

With the end of Dennard scaling and the slowdown of Moore's law, the computing industry has turned to heterogeneous architectures to improve performance and energy efficiency. These systems combine general-purpose processors with domain-specific accelerators, delivering significant performance-per-watt improvement by executing critical compute kernels in hardware. However, designing and programming accelerators remains a specialized and complex task, requiring both domain-specific knowledge and hardware design expertise.

This dissertation supports the thesis that *design-time abstractions are essential to effective accelerator design and integration, and their careful selection can substantially simplify the implementation of key features*. To support this claim, this dissertation presents three contributions that leverage abstractions to improve the design and usability of accelerators.

First, it introduces a synthesized, hardware-only garbage collector for FPGA accelerators. The collector operates concurrently with the accelerator, exploiting idle memory cycles and incurring negligible performance overhead under an eager collection policy. Second, it presents a method for automatic partitioning and control of fine-grained power domains in dataflow circuits. By using activity signals inherent to the dataflow abstraction, the method partitions logic into sleepable and always-on power domains and synthesizes control signals with provable correctness guarantees. Finally, it proposes *Accel*, a novel MLIR dialect for compiler-directed task dispatch in chiplet-based accelerators. Our dialect captures task graph structure and scheduling constraints, enabling a pull-based runtime execution model that improves locality and reduces synchronization overhead. Together, these contributions show that thoughtfully chosen abstractions can enhance productivity and enable features that would otherwise be difficult to implement.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I am deeply grateful to my advisor, Martha Kim, for her support and guidance throughout my PhD. From the beginning, Martha has always made time for me—whether to discuss research, review a draft, or simply catch up over coffee. Her insights and questions consistently challenged me to think more deeply and precisely, yet she always delivered them with encouragement and kindness. Her mentorship extended well beyond research, and through her guidance I have grown as a teacher, a communicator, and a person.

I am also grateful to Stephen Edwards for his generous and thoughtful mentorship. His advice and feedback significantly strengthened my work, and his steady support leading up to submission deadlines was especially appreciated.

I would also like to thank the other members of my committee, Luca Carloni, Tony Gutierrez, and Mark Santolucito for their approachability, support, and valuable advice. I am truly grateful for their guidance throughout this process.

Graduate school would not have been the same without the friends I made at Columbia. Thank you to everyone who has sat in the fishbowl over the years—particularly Paolo, Richard, Emilio, Tom, John, Davide, Luca, Andrea, Jihye, Maico, Guy, Kuan-Lin, Joseph, and Soyoon—for your support, both technical and personal. Your friendship and camaraderie made the good times even better and the hard times easier. To Ryan and everyone at Canterbury, thank you for being a friendly and grounding spiritual home on campus.

I would like to thank the faculty and administrators at Columbia who have helped make it a welcoming and supportive department. In particular, I'm grateful to Simha Sethumadhavan and

Adam Cannon for their generosity with their time, guidance, and encouragement. Thank you as well to Carol Begg and Da'Shante Smith for helping me navigate the administrative hurdles of a PhD.

I owe special thanks to Dan Sorin, who introduced me to computer architecture as an undergraduate. He encouraged me to explore it further and generously let me audit an advanced graduate course despite my lack of prerequisites. His early mentorship played a major role in shaping my academic path.

Finally, I am incredibly thankful to my parents, Laura and Jack, and to my sister, Kathleen, for their constant love, support, and encouragement. I could not have done this without you.

Chapter 1: Introduction

With the end of Dennard scaling [1, 2] and the slowdown of Moore's law [3], the traditional avenues for improving processor performance, namely increasing clock frequency and transistor density, have become less effective. Multicore architectures, architectures with multiple simple cores, exploit parallelism to improve performance; however, the potential gain is limited by Amdahl's law [4]. As a result, computer architecture has pivoted towards heterogeneous computing to improve performance and energy efficiency [5]. Heterogeneous architectures combine general purpose processors and accelerators, custom hardware designed for specific compute tasks, onto the same system on chip (SoC). These accelerators execute critical compute kernels directly in hardware, avoiding the overheads of CPU control logic. As a result, they often deliver significantly greater energy efficiency and performance for their designated tasks. This performance-per-watt advantage has made heterogeneous computing an attractive solution across a wide range of domains, from mobile devices [6, 7, 8] and embedded systems [9, 10] to data centers [11, 12, 13, 14, 15] and high-performance computing platforms [16, 17].

Despite these advantages, heterogeneous architectures are substantially more complex than single-core or multicore architectures. This complexity arises from the need to manage a diverse set of processing units, each with its own design constraints, performance characteristics, and programming model. Designing heterogeneous systems involves a range of challenges that span the accelerator's lifetime from design through utilization.

In the early stages of design, architects must identify compute kernels for acceleration, requiring both domain knowledge and hardware design expertise. Once a kernel is identified, designing the accelerator itself presents challenges such as designing the compute logic, ensuring correctness, and meeting performance and energy goals. After the compute logic is in place, the accelerator must be integrated into the larger system which includes memory, communication, and host CPUs.

Finally, the runtime management of accelerators, deciding when and how to invoke them and ensuring their effective utilization, requires significant software development [18].

This dissertation focuses on two key challenges: reducing the design effort involved in accelerator development, and improving the integration of accelerators into software workflows. To address these challenges, this dissertation supports the thesis that *design-time abstractions are essential to effective accelerator design and integration, and their careful selection can substantially simplify the implementation of key features*.

This dissertation first provides two examples where using a dataflow design abstraction can facilitate accelerator design, specifically the design of memory management and power management in accelerators. The term dataflow appears in many contexts; in this dissertation it refers to a circuit representation comprising functional blocks, called actors, that send and receive data tokens according to a handshake protocol. Data tokens are transmitted only when the sender offers a signal downstream that the data is *valid* and the receiver indicates via an upstream signal that they are *ready* to receive the data. Circuit design using a dataflow representation confers several valuable properties: the circuits are highly parallel, and they are patient because the data transmission handshake decouples timing from function. This dissertation demonstrates two scenarios in which this dataflow representation facilitates hardware design: a hardware garbage collector for FPGA-based accelerators and automatic fine-grained power domain identification and control logic synthesis.

Chapter 3 presents the garbage collector. Garbage collection is a key feature of managed programming languages. It simplifies software development and eliminates certain memory bugs, namely memory leaks and use-after-free errors. Our garbage collector is naturally robust to variable memory access latency, which allows it to run in parallel with the application it serves, waiting patiently to execute its memory accesses during the application's idle cycles. This garbage collector works best when it runs continuously in the background to maintain enough free memory for the application to make progress.

Chapter 4 demonstrates how the dataflow representation facilitates automatic power domain insertion and control in accelerators. Power gating a circuit reduces power consumption by supplying power to only the active parts of the circuit and powering down the idle parts. Defining independent power domains and generating the necessary control signals can be challenging. Large domains are easy to control, but suboptimal because the entire domain needs to be on if any part of the domain is on. Small domains do not suffer this problem, but demand complicated control to ensure they are awake when needed. Typically both the domains and their control logic are manually defined. In contrast, we have designed and implemented a pass in an open source compiler [19] which automatically defines and controls fine-grained power domains.

Chapter 5 presents an IR to facilitate accelerator utilization, specifically task dispatch to chiplet accelerators. Chiplet accelerators, or accelerators comprised of many small integrated circuits working together, are typically represented to host software as a set of queues containing tasks to be dispatched to each accelerator intermingled with barriers and other synchronization operations. On the accelerator side, a command processor pulls from this queue and dispatches tasks to accelerators, respecting the synchronization operations as it encounters them. This push-based model is simple but can suffer from load imbalance and head-of-line blocking, resulting in wasted compute resources and delays. We designed a new dialect in MLIR, an open source compiler, that puts hints about the full topology of the task graph in the queue for the command processor. Specifically, the host-side compiler analyzes the graph and batches tasks so that the command processor runs them on the same physical chiplet, reducing delays caused by data movement while still maximizing compute resource usage.

1.1 Summary of Contributions

The contributions of this dissertation consists of three projects that demonstrate how design abstractions facilitate accelerator design and integration.

1.1.1 Synthesized Garbage Collection for FPGA Accelerators

Speed and ease of accelerator design is a growing need. High-level programming languages have provided significant gains in the software world, but lag behind in hardware development. We present a hardware implementation of a garbage collector that automates memory management, one of the major conveniences of modern software languages. Our garbage collector runs concurrently with the application it serves, using idle memory slots to operate with little to no impact on performance. To achieve this, our collector exploits rapid synchronization that is straightforward in hardware but difficult in software. This synchronization enables the collector to interleave with fine pockets of idleness on a per-cycle, per-heap basis.

Our collector typically incurred negligible overhead, only slowing the application to wait for collection to free memory when the heaps were so small that the collector could not keep pace with allocation. We also found that our concurrent collector performs best under an eager collection policy that collects garbage well before the application exhausts the available memory. Although this eager strategy performs more collection operations than strictly necessary, the application never pauses because our collector operates entirely in the background.

1.1.2 Automated Power Domain Insertion and Control

Energy efficiency has become a primary concern in circuit design. Dennard scaling has ended, so operating voltages have ceased shrinking along with transistor sizes, meaning running a circuit at full capacity violates heat dissipation limits and battery lifetimes. Power gating is a promising solution, but defining independent power domains and generating the necessary control signals can be challenging. Large domains are easy to control, but sub-optimal because the entire domain needs to be on if any part of the domain is on. Small domains avoid this problem but demand complicated control to ensure they are awake when needed.

We present algorithms, implemented in a high-level synthesis tool, that define and control fine-grained power domains. Starting from the dataflow representation used during synthesis, our algorithms partition dataflow actors into always-on and sleepable domains and automatically generate wake signals using detailed activity information naturally available in the dataflow circuitry. Our experiments generated partitioned circuits with between 48 and 541 fine grained power domains, far more than could be manually implemented. Additionally, these designs have a provable progress guarantee that domains will wake up and produce the same results as the ungated circuit, regardless of the delay in actually waking the power domain. We found that power domains sleep on average 60%-73% of the runtime, but that waking and sleeping interdependent dataflow actors can extend overall runtime.

1.1.3 Compiler-Directed Task Dispatch for Chiplet Accelerators

Chiplet-based accelerators are increasingly common in heterogeneous systems. However, conventional push-based task dispatch models where the host enqueues tasks into in-order queues struggle to support the fine-grained, asynchronous workloads prevalent in modern applications. These models introduce inefficiencies such as head-of-line blocking, limited flexibility, and poor resource utilization. We introduce *Accel*, a custom MLIR dialect designed to represent task graphs, dependencies, and scheduler directives. The dialect allows the compiler to encode readiness conditions, control dependencies, and fine-grained execution flow into a programmable intermediate representation. Our runtime scheduler interprets the directives and dispatches work to accelerators only when dependencies are resolved, significantly reducing overhead.

We prototype our system on an AMD Versal platform featuring AIE accelerators and Black-Parrot RISC-V packet processors and evaluate it using sparse triangular solver workloads. Compared to a traditional push-based model, our approach achieves up to 50% speedup over the push-based model. These results demonstrate that compiler-guided dispatch and programmable runtime scheduling are effective tools for enabling scalable, autonomous task graph execution in chiplet-based accelerators.

1.2 Dissertation Outline

In the remainder of this dissertation, Chapter 2 provides background information accelerator design and programming models, compiler multi-level lowering, and the dataflow representation used in the first two contributions. Chapter 3 presents a synthesized garbage collector for FPGA accelerators. Chapter 4 presents a methodology for automated power domain insertion and control in dataflow circuits. Chapter 5 presents an intermediate representation to facilitate compiler-driven task dispatch in chiplet accelerators. Finally, Chapter 6 summarizes the contributions of this dissertation and proposes future work.

Chapter 2: Background

This chapter lays the foundation for the remainder of the dissertation. It begins with an overview of the architectural challenges that gave rise to heterogeneous systems (Section 2.1). Then, Section 2.2 gives an overview of accelerators, beginning with a survey of their use in modern systems (Section 2.2.1), followed by a deeper discussion of accelerator design (Section 2.2.2) and programming models (Section 2.2.3). We then pivot in Section 2.3 to an overview of the dataflow abstraction used in Chapter 3 and Chapter 4. Finally, Section 2.4 provides an overview of multi-level IR lowering, the compiler methodology integral to Chapter 5.

2.1 The Rise of Heterogeneous Computing

For many years, Moore's Law [3] enabled rapid performance improvements by doubling the number of transistors on a chip approximately every two years. This growth in transistor density allowed for increasingly complex and powerful processors. Perhaps an even more significant factor was Dennard's observation in the 1970s - known as Dennard scaling - that as transistors shrank, their voltage and capacitance also shrank proportionally [1]. This meant that smaller transistors not only operated faster, but also consumed less power and that power density, defined as the amount of power dissipated per unit area, remained roughly constant even as transistor counts increased.

However, as transistors shrank, the physical assumptions underlying Dennard scaling began to break down [2]. Leakage current is the electrical current that flows through transistors even when they are inactive. While leakage current had been negligible for large transistors, it is more significant at smaller scales due to shorter channel lengths. This led to the so-called power wall, where power density constraints limit the performance gains achievable through increasing transistor count and clock frequency. Simply adding more transistors no longer translated into performance improvements, as the associated power and thermal costs became unsustainable.

This shift pushed computer architects to seek performance improvements in other ways, leading to the rise of multicore architectures. Instead of relying on faster individual cores, multicore processors integrate multiple CPU cores onto a single chip. These architectures improve performance by parallelizing workloads across multiple cores. However, Amdahl's Law imposes a fundamental limit on the speedup achievable through parallelization. The maximum speedup is limited by the portions of an application that must execute sequentially, and even for highly parallel workloads, the best case speedup scales linearly with the number of cores. Even for highly parallel applications, real-world performance improvements are not linear due to synchronization and communication overheads [4, 20, 21]. While multicore processors have delivered substantial performance benefits and remain a central part of modern computing, they are unlikely to drive future gains at the same pace.

In achieve improvements beyond multicore scaling, computer architects have increasingly turned to specialization to achieve performance and energy improvements [22]. Heterogeneous systems integrate general-purpose CPUs with specialized accelerators, customized hardware circuits designed to execute a specific algorithm with greater energy efficiency and performance than general-purpose processors.

Despite their advantages, heterogeneous architectures are more challenging to design and implement than single-core and multicore architectures. A key challenge is identifying the computational kernels that will benefit most from acceleration. Then comes the task of mapping algorithms to hardware, which requires both hardware design and domain-specific knowledge. Integrating accelerators into the system poses challenges such as communication, memory consistency, and power management which often must be explicitly handled by the accelerator. Heterogeneous architectures are also more challenging to design at the physical level to meet timing, area, and thermal constraints. Finally, the programming model for accelerators remains a significant challenge. Unlike CPUs, accelerators often require specialized frameworks, compilers, and APIs, which makes software development and debugging more complex.

2.2 Accelerators

There are a wide variety of accelerators, each offering different trade-offs in performance, energy, and flexibility. Application-specific integrated circuits (ASICs) are the most specialized accelerators, designed to execute a fixed function with maximum performance and power efficiency. However, this specialization comes with high design and fabrication costs [23]. Less specialized accelerators, such as graphics processing units (GPUs) and field-programmable gate arrays (FPGAs), offer more flexibility. GPUs contain hundreds to thousands of small cores designed to perform the same operation on a large volume of data in parallel. Originally developed for graphics rendering, they are now widely used across other domains such as scientific computing and machine learning. FPGAs, on the other hand, offer flexibility through their arrays of configurable logic blocks that can be programmed to implement a desired function. While their performance and energy efficiency generally exceed those of CPUs, they fall short of ASICs. Nonetheless, the reconfigurable nature of FPGAs allow them to adapt to evolving accelerator needs [24].

2.2.1 Survey of Accelerators

Accelerators are ubiquitous across the entire computing spectrum, from mobile devices to large-scale data centers. In the mobile domain, SoCs now routinely incorporate a wide range of accelerators to handle tasks such as graphics rendering [25], image and video processing [26], and signal processing [27]. At the other end of the spectrum, the rise of big data, machine learning, and cloud services has spurred the adoption of accelerators in data centers. These accelerators are used to handle large-scale data processing tasks such as machine learning training and inference [11] and data transformation [28]. One notable early example is Microsoft's use of FPGAs in its Azure data centers to accelerate search ranking and network processing tasks [12]. Microsoft's data centers demonstrate that integrating FPGA accelerators into server architectures can improve throughput and reduce latency for cloud services while offering reconfigurability for evolving workloads.

Machine learning has also been a major driver in the demand for accelerators [29, 30, 31].

Early machine learning models were run on general-purpose CPUs. However, general-purpose processors are not well-suited to the large number of matrix multiplication operations necessary for training and inference. In the late 2000s researchers started using GPUs to train machine learning models, as their architecture is optimized to perform the same operation across large data sets. In 2012 researchers at the University of Toronto demonstrated the benefit GPU acceleration for machine learning with AlexNet which used GPUs to train their deep convolutional neural network for the ImageNet competition [32]. As machine learning models and datasets grew larger, even GPUs were unable to meet performance and energy demands which prompted a shift towards ASICs for machine learning. Google developed the Tensor Processing Unit (TPU), an accelerator designed for machine learning inference and later extended to training, and deployed them in their data centers in 2015 [11]. The TPU is designed around systolic arrays of matrix multipliers which improve performance and energy efficiency over GPUs by reducing reads and writes to memory.

A growing area of accelerator design is the use of chiplets, which are small modular functional blocks that can be integrated into a single package [33, 34, 35]. Unlike traditional SoC designs which require fabricating all components on a single die, chiplet-based architectures enable designers to assemble systems from a variety of blocks. These chiplets can perform specialized tasks and be fabricated using different process nodes optimized for their specific functions. Chiplet accelerators enable better scalability, design reuse, and integration by allowing designers to mix and match different components. This type of modular design is also a cost-effective way to improve manufacturing yield and reduce design cost because smaller dies are less susceptible to defects.

2.2.2 Accelerator Design

While accelerators come in many forms, they are generally composed of a common set of components. At the core of every accelerator are one or more processing elements responsible for executing computations. In ASICs, this logic is hard-wired to perform a fixed function. In contrast, programmable accelerators such as GPUs contain many small cores with custom instruction sets, enabling flexibility across a broader range of workloads.

Unlike general-purpose CPUs, most accelerators do not rely on the system memory hierarchy. Instead, they use explicitly managed local memories known as scratchpads, which offer low-latency access and predictable performance [36]. GPUs are an exception, often incorporating a full memory hierarchy into the accelerator [37].

To communicate with the rest of the system, accelerators require dedicated communication infrastructure. In simpler configurations, accelerators are connected to the host via a shared bus. More complex systems use a network-on-chip (NoC) to connect many cores and accelerators via a packet-based communication network [38].

Finally, many accelerators include control logic to manage configuration, decode instructions, and coordinate the processing elements. This control logic can range from a simple state machine in fixed-function designs to sophisticated controllers that interpret and schedule instruction in programmable accelerators.

All of these components contribute to the overall complexity of accelerator design. Most accelerators are developed at the register-transfer level (RTL), where digital logic behavior is described using hardware description languages such as SystemVerilog [39] and VHDL ashenden2010designer. These languages, which are analogous to low-level assembly languages in software, offer fine-grained control over both hardware behavior and structure. This gives the developer the ability to tailor every aspect of the hardware; however, it is time-consuming and requires expertise in digital logic design.

An alternative to traditional RTL design is high-level synthesis (HLS), a methodology which compiles accelerator designs written in high-level languages into RTL [40]. HLS significantly reduces the development cost of accelerators by raising the level of abstraction. However, achieving high performance with HLS still often requires knowledge of the underlying hardware. Additionally, HLS tools typically support only a subset of language features, requiring developers to write code within the tool's synthesis capabilities. Several commercial HLS tools are available, including Cadence Stratus [41] and AMD Vitis HLS [42], which are widely used for both ASIC and FPGA development. HLS is also an active area of research with ongoing efforts to improve the performance and efficiency of synthesized accelerators and to broaden the range of high-level programs that can be effectively mapped to hardware [43, 44].

In addition to high-level synthesis, there is growing research interest in domain-specific languages specifically for hardware development. These languages aim to raise the abstraction level above RTL while avoiding the limitations associated with using existing software languages for hardware design [45, 46, 47]. However, these languages have limited uptake due to factors such as the steep learning curve and the need to adopt new and less mature toolchains.

2.2.3 Accelerator Programming Models

Heterogeneous architectures also introduce significant challenges for software developers to effectively utilize specialized components. In traditional single-core and multicore architectures, the instruction set architecture (ISA) abstracts away much of the underlying hardware complexity, allowing programmers to write code without needing detailed knowledge of the processor's implementation. In contrast, developers must understand the specific characteristics and capabilities of each device in a heterogeneous system to achieve good performance.

Programming heterogeneous systems presents several key challenges. One major issue is portability across architectures and the lack of unified abstractions. Different heterogeneous architectures may include different combinations of CPUs, GPUs, and other accelerators, each with unique capabilities and interfaces.

Another challenge involves task assignment and load balancing. Determining how to allocate tasks to available accelerators to maximize system efficiency is a complex problem. These decisions can be made statically at compile time or dynamically at runtime, and factors such as contention, resource availability, and data locality all affect performance. For example, assigning a task to an accelerator experiencing high traffic may degrade overall performance rather than improve it.

Finally, the memory hierarchy and data movement pose a challenge. Many accelerators do not utilize the system's memory hierarchy, but instead rely on their own local scratchpad memories. This requires software to manage data transfers to and from these memories, and to ensure consistency.

Accelerator programming models are heavily influenced by where the accelerator is integrated into the overall system. Tightly coupled accelerators are typically embedded within the processor datapath or close to the processor core, often sharing registers, caches, and other architectural resources [48]. These accelerators are usually accessed through ISA extensions, which define new instructions that invoke the accelerator's functionality as part of the standard instruction flow.

This level of integration enables low-latency invocation and fine-grained coordination between the processor and the accelerator. However, the developer must program the accelerator directly using assembly or the compiler must be aware of the extended ISA and support the generation of the new instructions. This may involve modifying the compiler's instruction selection and scheduling phases to ensure that the accelerator-specific instructions are emitted correctly.

For loosely coupled accelerators, those connected to the processor via the memory bus, much of the research and development in heterogeneous programming models has focused on GPUs. Most GPU programming models fall into two main categories: specialized languages designed for GPU programming and directive-based models that extend existing programming languages.

The first category includes low-level models such as CUDA [49] and OpenCL [50], and HIP [51]. These frameworks expose the underlying GPU hardware through APIs that allow developers to write parallel algorithms. They provide fine-grained control over thread organization, memory usage, and synchronization, enabling high-performance execution of compute kernels on GPUs. However, this performance comes at the cost of portability and ease of development because code must be written specifically for the target model.

The second category is directive-based models, which use special compiler directives, or keywords embedded in the source code, to guide parallel execution. Examples include as OpenMP [52] and OpenACC [53]. These models simplify GPU programming by allowing developers to incrementally adapt existing CPU code for GPU execution without requiring a complete rewrite. Developers identify parallel regions of code using the directives, and, in some models, also specify how tasks should be mapped to the device. While these models offer less fine-grained control, they are more accessible and portable compared to low-level approaches.

Unlike GPUs, which have well-established and broadly adopted APIs, many other classes of accelerators have much less mature programming models. These accelerators have diverse interfaces, which makes it difficult to develop unified programming abstractions. As a result, these accelerators are often programmed through domain-specific languages and frameworks that abstract away hardware details while targeting specific application domains. In the machine learning domain, for example, frameworks like PyTorch [54] and TensorFlow [**abadi**] allow developers to define neural networks in a Python-based environment. These models are then compiled using domain-specific compilers such as XLA (Accelerated Linear Algebra) [55], which optimize and lower machine learning computations into backend-specific code for CPUs, GPUs, and TPUs.

Outside of these narrowly focused domains, the challenge of writing portable, high-performance code across heterogeneous systems remains. This has motivated the development of portability frameworks like Kokkos [56], RAJA [57], and SYCL [58]. These frameworks provide higher-level abstractions for expressing parallelism and data movement in a hardware-agnostic way.

Portability frameworks are typically used in conjunction with a runtime system that is responsible for managing the dynamic aspects of parallel computation. For example, the runtime system may determine how to map tasks to accelerator devices, when and how to transfer data between memory spaces, and how to respond to changing workloads or system states. The framework and runtime together enable developers to effectively program accelerator-rich heterogeneous systems.

2.3 Dataflow Abstraction

Dataflow architectures were first introduced in the 1970s and 1980s as general-purpose ISAs to enable highly parallel computation by executing program parts whenever the necessary data values became available, rather than following a strict sequential order [59, 60]. Numerous variations were proposed, but these architectures struggled to compete with the more established von Neumann model in terms of performance. Additionally, they required specialized dataflow programming languages, which limited their accessibility and broader adoption [61, 62]. As a result, dataflow architectures failed to gain traction in mainstream computing.

As improvements in single-core performance began to slow in the mid-2000s, dataflow architectures received renewed attention as a potentially more scalable and more parallel alternative to superscalar designs [63, 64, 65, 66]. Despite promising research prototypes, multicore processors emerged as the more practical and widely adopted solution, offering better backward compatibility and easier integration into existing software ecosystems.

More recently, the dataflow abstraction has experienced a resurgence as an intermediate representation in HLS workflows to compile programs with complex control flow where traditional static scheduling falls short [67, 68, 69, 70]. These modern dataflow IRs use latency-insensitive design [71, 72] to implement dataflow networks as digital logic circuits.

A dataflow circuit is a network of functional units, called *actors*, connected via channels where data is transferred as tokens according to a latency-insensitive handshake protocol. The circuit implementation of each actor includes both functional logic and handshake logic to determine when it accepts and produces tokens. Each channel implements a latency-insensitive handshake as follows. Each *data* wire between actors is accompanied by a *valid* signal that is 1 if the upstream actor is offering a data token. A *ready* signal runs in the reverse direction indicating when the downstream actor is ready to consume a data token. In this protocol, a token is transferred between actors if and only if both *valid* and *ready* are true.

Each channel consists of a *data* signal, which carries the payload, an accompanying *valid* signal to indicate if the data is a valid token, and a *ready* signal running in the opposite direction to indicate if the downstream actor is ready to accept data. Tokens are transferred only when both *valid* and *ready* are true.

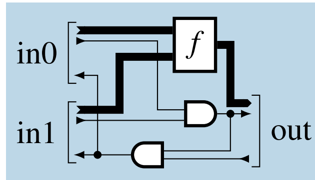

Figure 2.1: An example of a unit-rate actor. The thick lines show the functional wires, and the thin lines the handshake wires.

Figure 2.1 depicts an example of a unit-rate actor, which waits for a single token on each of its inputs before producing a single token on each of its outputs. The thick lines represent functional wires, and the thin lines represent handshake wires. The output is valid if both inputs are valid, and the inputs are ready if the output is ready and valid.

CIRCT [19], an open source hardware compiler, contains a library of actor implementations which is consistent with the actors described by Townsend et al. [70], but arbitrary additional actors can be designed. In addition to unit-rate actors, *mux* and *demux* actors consume a select token from their control input then either routes a token from the selected input to the output or routes a token from the input to the selected output, respectively. A *merge* actor selects a token from one of its inputs and routes it to the output. It has an additional output channel which emits a select token indicating the chosen input channel.

Dataflow actors are compositional, enabling complex circuits to be constructed from these simple components. To prevent combinational cycles within a dataflow network, two key rules are enforced. First, every cycle in the dataflow network must have at least one data and one control buffer. Second, no actor may have a combinational path from any of its ready signals to any of its valid signals. These constraints eliminate combinational cycles and ensure that dataflow circuits are deadlock-free.

Channels in the dataflow network are unbounded FIFOs, but practically they are implemented several ways for different functions. The simplest communication channel is a direct point-to-point connection. This connection is fast, but couples the actors' firing rules and potentially creates long combinational paths. The communication channel can also be a buffer that holds a single data token and decouples the firing of upstream and downstream actors. A *data* buffer breaks a combinational

```
int collatz(int a) {
  int b = 1;
  while (a != 1) {
    if (a % 2 == 0)
      a = a / 2;
    else
      a = 3 * a + 1;
    b = b + 1;
  }
  return b;
}
```

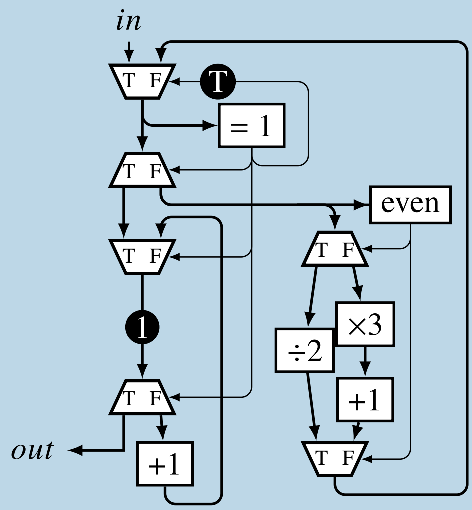

Figure 2.2: Dataflow network implementing the Collatz Conjecture algorithm. Each block represents a dataflow actor that sends and receives data according to a handshake protocol.

path in the data/valid network, and a *control* buffer breaks a path in the ready network. Lastly, a *fork* implements a channel with fanout that handles ready and valid signals without introducing combinational cycles.

As an example, we show a dataflow implementation of the Collatz Conjecture algorithm to determine the number of iterations it takes to turn its argument n into the value 1 by repeated applications of the function $f(n) = \begin{cases} n/2 & \text{if } n \text{ is even} \\ 3n+1 & \text{otherwise.} \end{cases}$ Figure 2.2 depicts a C implementation and its corresponding dataflow implementation. In the dataflow, in the first iteration, a mux passes the input through to an equality actor which determines if the Collatz sequence is complete and the number of iterations is output or to continue the algorithm by passing the token to a sequence of actors to check parity and update the number accordingly to pass back to start another iteration.

There are several key features of this abstraction that facilitate solutions to microarchitectural design challenges. The first is that dataflow circuits are data driven - work is only done when and where there is valid data. The second is that dataflow actors are patient, meaning the speed at which tokens arrive has no effect on function; computation simply waits until the required data is present [73]. Lastly, dataflow circuits are parallel because each block operates independently, triggering as soon as its inputs are ready, which allows actors with available data to continue working even if other actors are stalled.

2.4 Multi-level IR Lowering

Intermediate representations (IRs) are internal abstractions used by compilers to lower code from the source language to machine-level instructions [74]. Historically, compilers used a single IR to bridge between high-level source languages and low-level targets. By abstracting away the details of the source language and target device, IRs enable analysis and optimization. Intermediate representations can also provide a common abstraction for many different front and back ends, creating a centralized connection point. The LLVM compiler infrastructure [75], built on LLVM IR, is a widely adopted open-source compiler framework that supports optimizations across a broad range of source languages and targets.

However, the increasing complexity of software, especially when targeting heterogeneous systems, has driven a shift towards multi-level IR lowering in compilers [76, 77, 78]. Different optimizations are most effective at different levels of abstraction, and a single IR is often insufficient to capture the full range of transformations required for high-performance code generation. A high-level IR that closely resembles a domain-specific language can enable domain-aware optimizations, such as operation fusion - for instance combining a matrix multiplication and subsequent addition into a single fused operation. At the mid-level, more general-purpose compiler optimizations - such as loop unrolling, vectorization, and tiling - can be applied to improve performance across a wide range of architectures. Finally, a low-level IR that is closely tied to the target hardware can optimize details like memory layout, instruction scheduling, and resource allocation. This multi-level IR approach enables a structured and hierarchical compilation process, in which distinct IRs represent different levels of abstraction through which the code is progressively lowered.

The strength of this methodology is evident in the widespread adoption of the MLIR (multi-level intermediate representation) framework [76], despite its relatively recent introduction. MLIR is a modular and extensible compiler infrastructure designed to simplify the development of domain-specific compilers and support progressive lowering across multiple levels of abstraction. MLIR enables developers to define dialects, which are customized IRs with their own operations, types, and transformation passes. This flexibility allows developers to express domain semantics clearly while leveraging shared infrastructure for optimization, analysis, and code generation. It also encourages reuse, as developers can use their own custom dialects alongside existing ones to create complex compiler pipelines tailored to specific domains. The framework has been adopted across many domains and several major domain-specific compilers including TensorFlow [79] and IREE [80] are transitioning to MLIR.

Chapter 3: Synthesized Garbage Collection for FPGA Accelerators

This chapter focuses on memory management for accelerators, a feature typically managed by software. This chapter presents a synthesized, hardware-only garbage collector for FPGA accelerators that simplifies accelerator development by automating memory management. The collector leverages the dataflow abstraction described in Section 2.3, which enables highly parallel, latency-insensitive coordination between application logic and background collection tasks. By exploiting idle memory cycles and leveraging the rapid synchronization possible in hardware, the collector runs concurrently with the application and imposes negligible overhead. This work demonstrates how abstraction at the circuit level can facilitate the integration of runtime services into hardware while maintaining high performance expected of accelerators.

3.1 Introduction

FPGAs are attractive acceleration platforms with recent commercial and cloud deployments. However, programming FPGAs remains a formidable engineering effort, requiring intimate knowledge of hardware design and the application. High-level synthesis (HLS) can greatly reduce this effort by compiling programs specified in a higher-level language, although it rarely matches the quality of hand-engineered designs. Nevertheless, rapid design and deployment remain paramount concerns alongside performance, as evidenced by Google's use of HLS to design its video transcoding chip [81].

In modern programming languages, garbage collection simplifies the developer's task and eliminates bugs such as memory leaks and use-after-free errors. High-level synthesis reduces the effort of hardware design by compiling programs written in higher-level languages, but although HLS frameworks excel at optimizing arithmetic, loops, and regular memory accesses, they provide limited support for memory management.

In software, garbage collectors typically impose a 10-35% slowdown depending on the application [82, 83], which hardware acceleration can sometimes reduce [84, 83, 85, 86]. By definition, accelerators are faster than software and are therefore *even more sensitive* to interference from a garbage collector. A 1ms collection pause on a 100ms software task is just 1% of runtime, but the same pause on a 10ms accelerated task results in a 10% slowdown.

We present a synthesized, hardware-only automatic garbage collector that operates concurrently with the accelerator, imposing a negligible slowdown on the accelerator (e.g., under 1%) with sufficiently large heaps. Our collector integrates with FPGA accelerators synthesized from Haskell [87, 70] and operates on multiple, parallel heap memories. Each is implemented with FPGA BRAMs and can perform either an application or garbage collection operation in each clock cycle.

When free memory dwindles below a given fraction of the total heap size, our garbage collector initiates collection by taking a snapshot of all of the pointers in the program's possession. First, it performs a global memory fence, which waits for all in-flight memory operations to complete while prohibiting new ones. Then, the collector takes a single-cycle snapshot of all the "roots"—the pointers directly known to the application—and resumes normal application execution. In our hardware collector, this operation is vastly less intrusive than is typical in software implementations.

Our hardware collector synchronizes rapidly with the application, allowing it to safely run in the background. The collector uses memory cycles that the application would otherwise leave idle, much like simultaneous multithreading [88] fills idle execution slots with instructions from other threads. To keep the memory allocator available to the application during collection, the allocator and collector maintain a shared list of free objects to which the collector enqueues and the allocator dequeues. Maintaining such consistency is costly in software but is easy to do in hardware.

Our collector is integrated into an FPGA HLS flow that relies on an immutable memory model, i.e., the contents of an object are fixed at allocation time. While this simplifies the challenge of allowing the application to run during collection, our approach does not demand it. The techniques that other collectors [84, 89] use to manage mutations of the heap during collection could apply in our setting.

3.2 Garbage Collection Background

A garbage collector is an automated memory management tool that identifies and reclaims the memory of "dead" objects that are no longer accessible by the application, releasing application developers from having to manually free unused memory and the bugs that often accompany doing so. Jones [90] is the standard text on garbage collection.

Tracing and reference counting are the two main approaches. Ours is a tracing collector, which starts from the application's *roots*—the set of pointers held directly in variables (our roots reside in buffers throughout the microarchitecture)—then traverses the object graph to *mark* all objects reachable from roots. When marking is complete, the *mark and sweep* variant of a tracing collector does a linear scan to reclaim the storage of all unmarked objects.

By contrast, reference counting maintains a per-object count of references, recursively frees an object when its count reaches zero, and usually requires more bookkeeping.

Collectors often pause the application to avoid the confusion of a changing reference graph, a *stop-the-world* approach. Unfortunately, this often produces unacceptable pauses. *Concurrent collectors* avoid such pauses by allowing the application to continue during collection, but are challenging to implement efficiently because of the careful synchronization required between the application and collector.

3.3 Related Work

Bacon et al. [84, 89] present one of the few garbage collectors that, like ours, is hardware and coexists with an accelerator on an FPGA. Their collector is also concurrent and collects multiple inter-connected heaps. They observed zero mutator stall cycles with heaps between 1.1 and 1.7 times the minimum necessary heap size, but did so on memory traces from a Java application; it is less clear what their performance overhead would be for an accelerator. Our garbage collector is integrated with a Haskell-to-Hardware compiler and we have measured its overhead on six accelerators being virtually zero.

Termination detection is a key challenge of multi-heap garbage collectors. Bacon et al. detect termination by waiting for the maximum cycles required for a pointer to travel between heaps; we support undetermined delays between heaps. Furthermore, Bacon et al. connect every pair of heaps; we only connect heaps that reference each other. Together, we believe these differences make our approach more scalable.

Hardware accelerators for software garbage collection are less like our work. Maas et al. [83] propose a tracing garbage collection accelerator that employs small CPU modifications. Their accelerator achieves high memory bandwidth by using a bidirectional object layout to identify reference fields and decoupling marking and tracing. Unlike our collector, software collects roots and processes state such as the page-table base register and provides them to the accelerator. Jang et al. [85] accelerate key primitives in the ParallelScavenge GC algorithm on the near-memory logic layer of a 3D stacked DRAM. Tang et al. [91] accelerate three common GC functions that consume nearly 50% of collection time and add CPU instructions.

Researchers have proposed alternative hardware assists for GC. The hardware reference counting of Joao et al. [86] stores the reference count in object headers and caches updates to reduce memory accesses. Srisa-an et al. [92] store a limited reference count for each object in their Active Memory Processor. These works identify and reclaim some dead objects in hardware, reducing the frequency of software collections. GC co-processors move collection off of the main core. The Integrated Hardware Garbage Collector of Garcia et al. [93] is closely coupled with the processor to support modern languages in embedded systems. Meyer [94] develops an on-chip GC co-processor paired with a special purpose processor for embedded systems. Schmidt and Nilsen's Garbage Collected Memory module [95] is a near-memory co-processor accessed over the memory bus. In a CPU, hardware read barriers reduce the synchronization cost of concurrent collectors. Azul Systems' Click et al. [96] build a custom system for concurrent garbage collection including

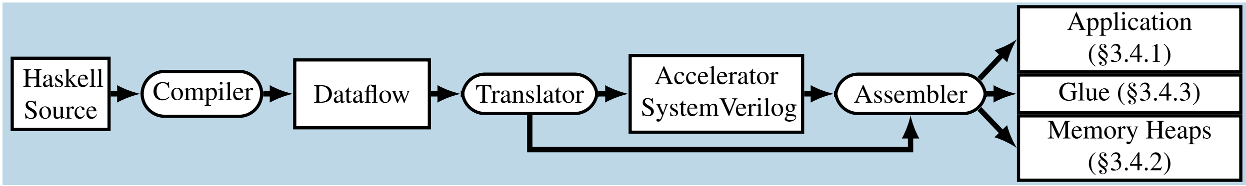

Figure 3.1: Accelerator synthesis toolchain. The Haskell program is compiled into SystemVerilog through a dataflow IR. The assembler synthesizes per-type garbage collected heaps (Section 3.4.2) connected to the application with glue logic (Section 3.4.3) according to type information and designer parameters.

a read barrier instruction. Meyer [97] implements both a hardware read barrier and handler directly in the pipeline of his GC co-processor.

3.4 Garbage Collector Design

Our garbage collector works in tandem with a hardware accelerator capable of taking atomic snapshots of the roots. We describe these structures below.

3.4.1 Accelerator Synthesis Flow

Figure 4.1 illustrates how we synthesize our accelerators with garbage collection. We extended the Haskell-to-Hardware compiler of Townsend et al. [87, 70], which targets algorithms with complicated control and irregular memory access patterns that rely on an immutable, garbage-collected heap.

The compiler synthesizes accelerators from Haskell programs by transforming them into a dataflow network, optimizing the network, and then synthesizing the network into SystemVerilog. Along the way, the compiler dismantles recursion and polymorphism [87] and adds buffers to the dataflow network to break what would become combinational cycles [70].

Dataflow nodes communicate data tokens that encode integers and Haskell's algebraic data types (tagged unions). The hardware employs latency-insensitive channels with back pressure [98]. Bit vectors hold data that typically includes a tag to indicate how to interpret the fields in the rest of the vector, typically consisting of integers and references to other objects encoded as

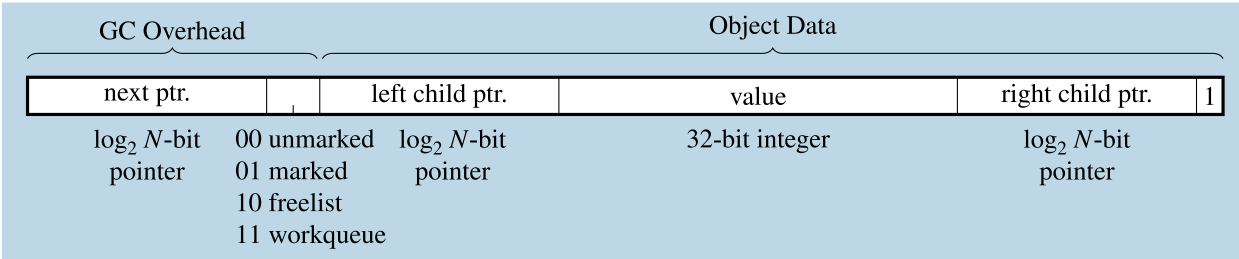

Figure 3.2: Layout of a binary tree node in memory. The application sees the Object Data bits: a value field, two child pointers sized according to the size of their heap, and a flag indicating the object is a branch or leaf. GC Overhead bits consist of a pointer field used to indicate the next object in the work or free list and two status bits indicating the object's garbage collection state.

pointers—per-type word addresses. Channels are strongly typed in that the compiler knows the possible interpretations of each vector, including any pointers.

Like Haskell's, our heap is immutable: an object's value is set only when the object is created. While the maximum live object count is the same for mutable and immutable memory, the immutable model results in more garbage because updating an object creates a new copy, keeping the live object count the same, but generating garbage out of the old copy. The dataflow network interacts with the heap through two primitives: *write* operates like C++'s *new* operator: it takes the data for a particular type of object (typically tag and field), allocates and stores it in memory, and returns a pointer to the object, whose data may be later retrieved by passing the pointer to *read*.

3.4.2 Heap Architecture

Our toolchain synthesizes an application-specific memory architecture in which each type has its own heap. The bit-width of each heap is set by the bit-width of the objects it stores and garbage collection metadata; the number of objects per heap is set by the designer. Figure 3.2 depicts the encoding of a binary tree node. The encoding, and hence the number of bits used for each object's fields, is automatically synthesized and is type- and application-specific.

Figure 3.3 shows the heaps we synthesize for an application that manipulates A's, B's, and C's. Each heap has a object storage array that is accessed by the application through the heap's read/write interface and implemented with pipelined FPGA BRAM with logic designed to fill the

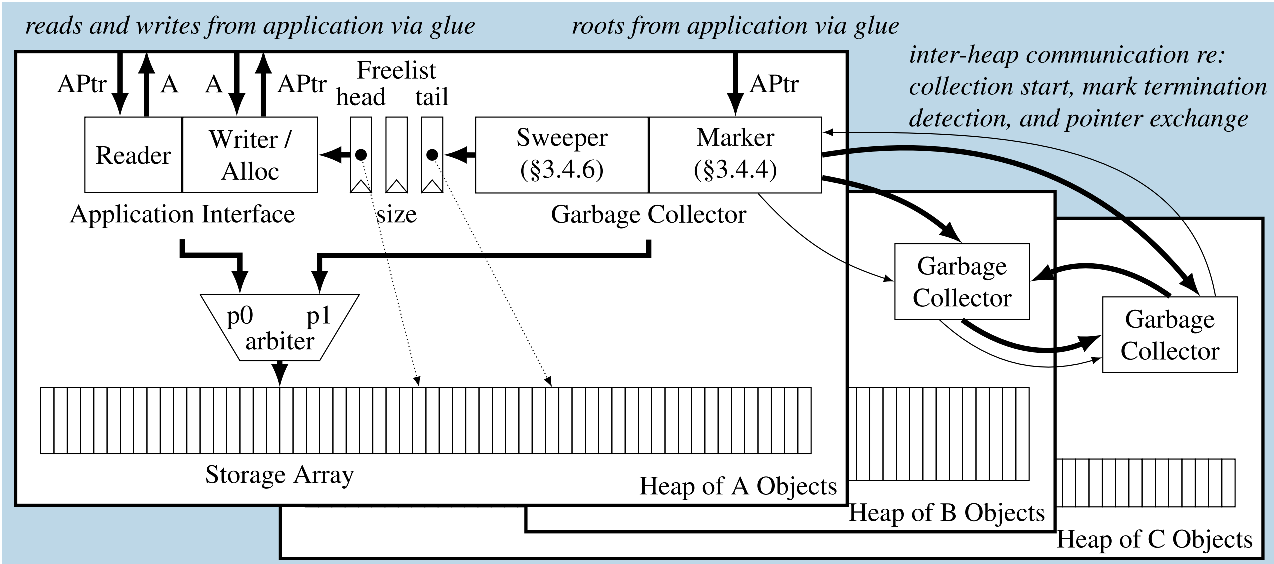

Figure 3.3: Parallel heap architecture. Each type in the application has a dedicated heap that operates independently from the other heaps *except* during garbage collection, when the heaps share pointers discovered during mark and communicate to detect mark completion.

pipe and tolerate its latency. In each cycle, an arbiter that favors the application grants access to either the application or the collector. Thus, while there is free memory, the application operates uninterrupted and the collector works in the background.

Objects that reference one another require special consideration in our multi-heap setting. Our static typing policy ensures we know at compile time which objects could potentially reference objects in other heaps. Consider an application with three types A, B, and C, where A objects contain B and C pointers. During garbage collection, when the marking process encounters an object of type A, marking will have to then mark the B and C objects it refers to, which are in separate heaps. Our heaps include such communication channels.

3.4.3 Root Collection

Garbage collection must capture a consistent snapshot of the state of the heap, notably all the roots. Software often takes a “stop-the-world” approach: the program is temporarily halted while the collector gathers roots from its state [90].

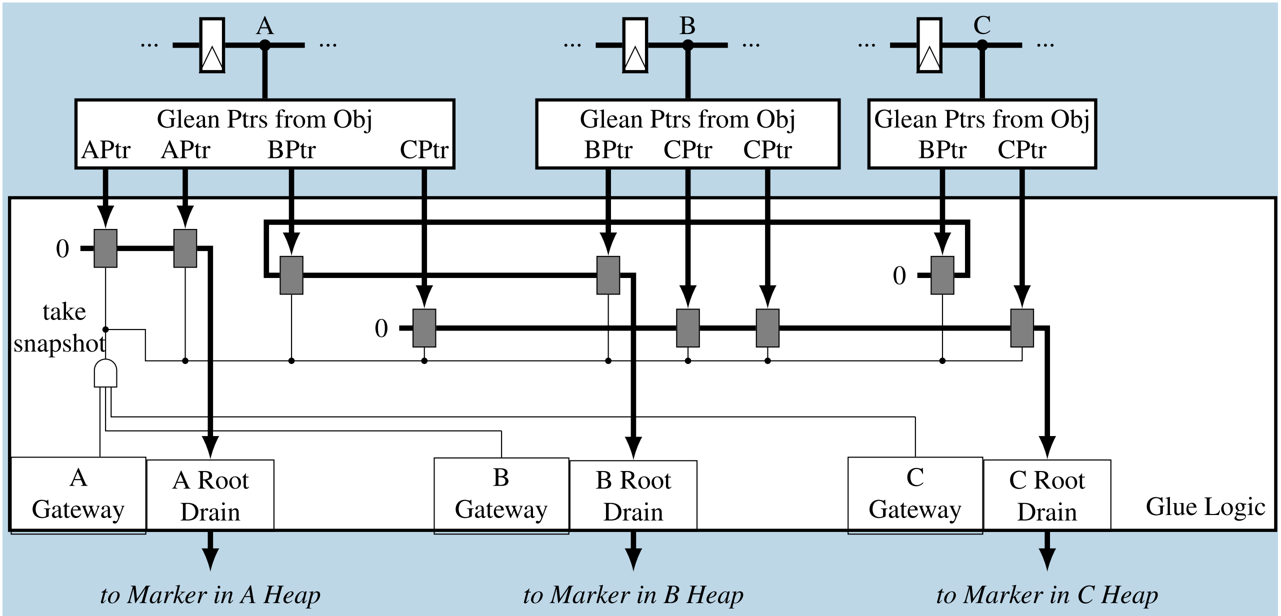

Figure 3.4: Single cycle root collection design. Residing in the glue logic between application and heaps, the root collection hardware waits for all pending memory requests to complete, snaps an image of all pointers in the application in a single cycle via parallel loads to multiple scan-chains, one per pointer type. These chains then drain the roots, in parallel, to their respective heap Markers. Collecting roots without stopping the application, apart for the brief wait to resolve pending memory requests, is a key source of performance gains. This example depicts three object types that refer to each other. Type A objects are tree-like and may refer to two other type-A objects, a B object, and a C object. Objects of types B and C are similarly referential.

We are able to gather roots quickly, due to the parallelism available in hardware. When collection has been triggered, the heaps block new memory operations until all in-flight ones have completed, then takes a single-cycle snapshot of all roots before letting the whole system proceed. The mark and sweep phases of the garbage collector are then performed in the background.

Each heap's gateway module maintains a count of in-flight reads and writes. When the free list in any heap dwindles below a threshold, the triggering heap signals its gateway module that it is time to garbage collect. The triggering heap also signals the others, which send the message to their own gateways. The gateways then apply back pressure on the request channels to stop incoming requests and wait until all pending requests complete. During this wait, nodes that are not memory reads or writes may continue to fire.

Once pending memory requests have completed, we capture all the roots in a single "snap" cycle using the machinery in Figure 3.4. Compile-time type information tells us exactly where roots can hide: every buffer that holds a type that may contain a "pointer." We augment each such buffer with an additional "root register" that can be compelled by a global "snap" signal to capture all pointers, if any, held in the buffer. Once the root registers are loaded, the hold on new memory operations is lifted and the application can continue normal execution.

All the root registers of each type connect into a shift register leading to that type's heap. Once the root registers have been loaded, the garbage collector begins draining these shift registers at its own pace.

3.4.4 Marking

Marking begins with the application roots. It appends roots to a work list linked via the "next ptr" field (see Figure 3.2). Each object also has a two-bit status field that indicates whether the object is marked, is in the work queue, or is free.

Marking traverses the object graph breadth-first by popping the top element from the work list then appending each of its unmarked children. The only departure from a standard mark algorithm is that pointers here can come from incoming roots, locally discovered pointers, or pointers unearthed by a mark process in another heap.

To avoid deadlock, each Marker adds discovered references to the appropriate work list before considering the mark complete and popping the next object from its work list. Thus, as long as each mark engine has sufficient internal buffering to hold the maximum number of discoverable references per object, the marker can always drain its internal buffers and make progress.

To maintain correctness during concurrent operation, objects allocated while the collector is marking are presumed live and allocated *marked*. This is a conservative assumption that ensures no live objects are incorrectly reclaimed. At worst, this enables the occasional object to survive one additional collection. When all mark engines have run out of objects to mark, the mark phase is over.

3.4.5 Mark Termination Detection

Termination detection in our distributed mark system is challenging. While a marker may "run out" of pointers and appear to be done, another remotely-discovered pointer may restart it. Termination occurs only once the last heap has "run out" of pointers and there are no inter-heap pointers in flight. We adapted our algorithm from Chandy-Lamport [99]: we connect the heaps in a simple ring around which they send "done check" messages. A cycle counter in each heap records the time when it "runs out" of work. When a heap has no further pointers to mark, it initiates a check by sending a positive done check message to the next heap. This message contains the initiating heap's ID as well as the time stamp (cycle count) when the heap ran out of pointers. When a heap receives an incoming done check, if it too is done marking and finished before the heap initiating the check, it propagates the original positive done check message to the next heap. If instead a heap still has pointers to mark, or has run out of pointers but did so after the heap that initiated the done check, it propagates a negative done check message. The cycle completes when the initiating heap receives a done check message with its own ID. If the message is still positive, then all types have finished and the check initiator was the last to finish, indicating the mark phase is complete.

3.4.6 Sweeping

To reclaim unmarked garbage, each heap performs a linear sweep of its address space. Unmarked objects are added to the free list, which is chained together using the "next ptr." field and marked objects are unmarked.

During the sweep, each heap maintains a pointer S to the current position of the linear sweep. At all times, objects in memory below S have already been swept and are either on the heap and unmarked or in the free list; objects above S will be kept if marked and reclaimed otherwise. We allow allocations to proceed during sweep by either allocating it marked or determining that it will not be swept. If the allocator removes an address A from the free list, there are three scenarios:

- If $S > A$, the sweeper has already passed the address to be allocated, so the allocation can proceed normally. The sweep is effectively over for these objects in this pass.
- If $S = A$, the sweeper and allocator have collided, but because application accesses have priority over collector accesses, the allocation will occur before the sweep so we mark the object so it will survive.
- If $S < A$, the sweeper has not yet reached A and would reclaim any object at A that was unmarked, so we allocate the object at A and set it marked, guaranteeing the object survives the sweeper when it does arrive.

During the sweep phase, the application continues to operate normally. The sweeper performs many memory operations since it touches every element of the heap, but does so in the background, without slowing the application.

3.5 Evaluation

Our evaluation quantifies the performance overhead of our collector by comparing its performance to an (unrealizable) ideal of execution without garbage collection.

We find our run-in-the-background collector imposes less than a 1% performance overhead when the heaps are "big enough" (often 2.5× the working set size), the collector is set to start collecting before the heap fills up, and the application does not fully use the available memory bandwidth. Our evaluation aims to characterize the design space of systems that are provisioned just enough to achieve such low overhead.

We measure performance as the total number of execution cycles, which we gathered from the cycle-accurate simulator Verilator. Our experiments indicate that the clock frequency does not change for all implementations of the same application. Application slowdown is the percentage of additional cycles taken beyond that necessary for the ideal collector (simulated by making the heaps so large as to avoid ever needing collection); a 100% slowdown means the application running with our collector took twice as many cycles as the application would with an ideal collector.

Table 3.1: Benchmark applications

Application	Size (LoC)	Heaps	Rate (Obj/Cyc)
Bandwidth	176*	5	varies
MergeSort	48	6	0.073
TreeSort	49	7	0.073
Reflect	49	6	0.085
RedBlack	124	6	0.079
Map_RedBlack	129	6	0.075
TSP	100	8	0.001
Clustering	382	19	0.071

*Lines of SystemVerilog code; all others are lines of Haskell code

3.5.1 Synthesis Results

Our accelerators and synthesized garbage collectors are heavily pipelined to maintain high clock frequencies regardless of size, which our experiments confirm. To evaluate operating frequency and area, we synthesized our collectors in Xilinx Vivado targeting a Zynq Ultrascale XCZU7CG-FBVB900-1LV-I FPGA with 230400 LUTs and 11Mb of BRAM.

Area: We synthesized the memory and garbage collector without an application, implemented a single heap containing linked-list type objects, and swept its capacity from 2^8 to 2^{17} objects. This increased LUT and register needs only modestly since only the size of the pointer objects and memory multiplexing logic needed to change. The largest collector required fewer than 1200 LUTs.

Frequency: The clock frequency of the synthesized garbage collector and memory with no application ranged from 270 MHz to 350 MHz across all the heaps we considered. The frequency decreased modestly for larger heaps, likely due to BRAM wiring. In any case, our collector never limited application frequency: we synthesized accelerators with and without GC and found working frequency unchanged.

3.5.2 Applications

We ran experiments on the eight applications listed in Table 3.1. Haskell naturally supports applications with irregular memory access patterns and recursion which our benchmarks exploit. We synthesized most applications from Haskell programs using the toolchain depicted in Figure 4.1; we manually coded *Bandwidth* in SystemVerilog.

Bandwidth is a micro-benchmark that generates high levels of memory traffic. It reads and writes from memory at specified rates while building and destroying list data structures.

MergeSort recursively sorts an integer list by separating it into even- and odd-indexed elements, sorting those, and then merging. *TreeSort* sorts a list by building and flattening a binary search tree. *Reflect* builds a binary search tree then flips the left and right children of each node. *RedBlack* and *Map_RedBlack* build balanced red-black trees then RedBlack swaps the left and right children of each tree node; Map_RedBlack rewrites the tree ten times, adding a constant to each tree node each pass. *TSP* implements a nearest-neighbor heuristic for the traveling salesman problem starting from a matrix of towns and the distances between them. This is computationally intensive, but generates little garbage, which demonstrates the unobtrusive nature of our collector on applications with a small memory footprint. *Clustering* implements K-means clustering. Used in a range of machine learning applications, it first constructs a K-d tree from a list of 2D points then performs the clustering algorithm.

3.5.3 Performance Overhead vs Heap Size

Our experimental results for *MergeSort* in Figure 3.5 show the performance overhead of our collector drops below 1% compared to an ideal collector as the heap sizes begin to exceed about twice the size of the maximum working set. We gathered this data with each heap sized randomly between the maximum working set for its type and an over-provisioned maximum. The horizontal axis of Figure 3.5 shows the heap size in objects compared to the maximum working set of the application, which we found empirically using memory access traces from each application. The heaps on the left side of the figure are just barely large enough to run the application; heap space increases towards the right. Unlike stop-the-world collectors, concurrent collectors require more heap space than the working set to accommodate allocations during collection.

On the far left of Figure 3.5, the garbage collection overhead is as high as 40%, which cor-

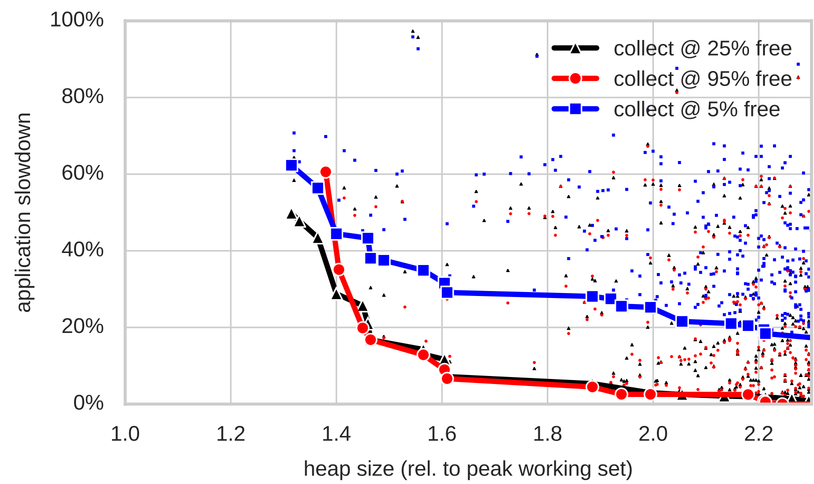

Figure 3.5: Garbage collector overhead in cycles on MergeSort as a function of heap size and collection threshold. An eager garbage collection trigger lets it run continuously in the background. Sufficient heap space prevents stalls by providing enough room for the collector to outpace allocations.

responds to the case where the heap is just barely large enough to accommodate the application. Here, the heap requires frequent garbage collection and the application is using memory at a rate faster than the collector recycles it. As heap capacity increases, the collection overhead drops as the application uses memory at a rate equal to or lower than the collection rate, indicating that the collector is able to scavenge unused memory bandwidth and still recycle garbage fast enough that the application can always allocate.

The overhead we report for each heap capacity value (horizontal position) represents the minimum observed overhead for that particular heap size—a Pareto frontier. An open problem in our technique is how best to allocate space among multiple heaps. We assume that an accelerator will be given a heap memory area budget, but that it will fall to the designer to allot it among the heaps, depending on the desired trade off between memory overhead and performance. Heap allocation is important because an under-provisioned heap can become a bottleneck by forcing more collections than if it had more capacity. Absent an effective algorithm for finding the best relative allocations, we show the best configurations from a randomized design space exploration.

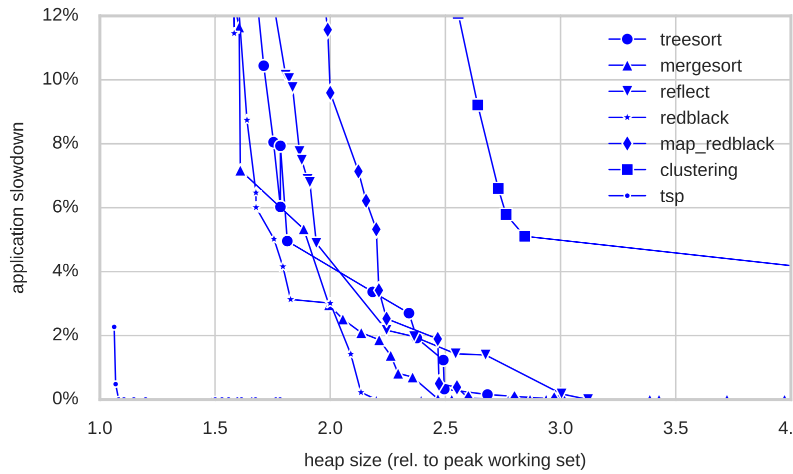

Figure 3.6: Garbage collector overhead in cycles as a function of heap size. Each application has a heap size where the collector can run fully in the background.

3.5.4 Performance Overhead vs Collection Threshold

Figure 3.5 also shows it is best for our collector to run continuously in the background rather than to minimize the number of collection events. We find that our collector slows the application less when it collects eagerly. Collection is triggered when the number of free objects in any heap drops the *collection threshold*. At a threshold of 5%, the collector waits until the heap is nearly full, whereas a 95% threshold leads to almost constant collection.

This result further demonstrates our collector's "run-in-the-background" character. This may seem counter-intuitive since a higher threshold triggers more collections, but overhead from the collector arises when the application has to wait for the collector to free additional objects. When garbage collection is eager, there is enough free memory for the application to continue making progress while the garbage collector works in the background. Triggering collection later, the application is more likely to consume all of the available heap space before the collector has reclaimed dead objects.

Our experiments show that above a certain size, the heaps are large enough for our concurrent collector to achieve idyllic behavior: collection is triggered early and often enough to avoid appli-

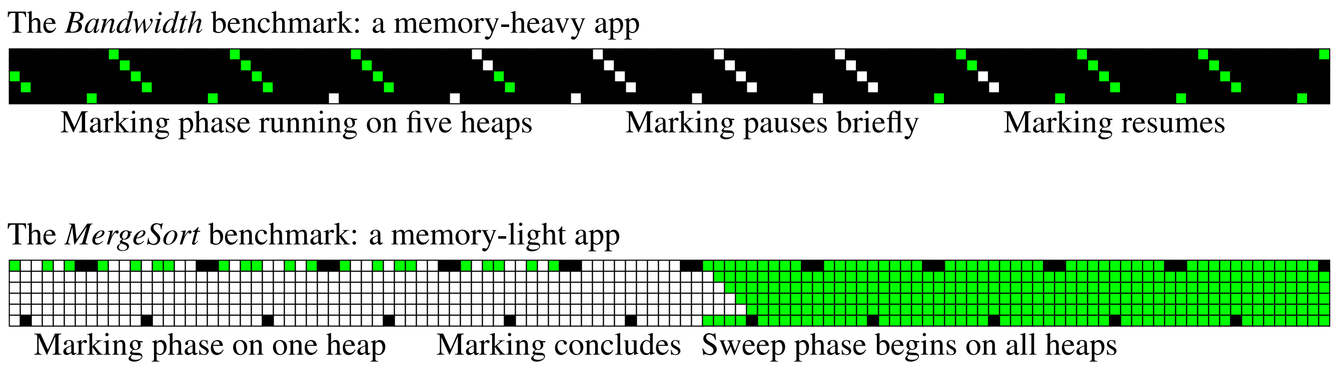

Figure 3.7: Memory accesses over time. (top) When the app memory utilization is high, GC proceeds more slowly, but it is still able to use gaps in application memory accesses to make progress. (bottom) When there are more memory slots available the collector uses however many it needs, from just a few to all.

cation pauses due to an out-of-memory situation and there is sufficient unused memory bandwidth to accommodate this larger-than-necessary number of collections. The overhead never drops to exactly zero in these experiments because there is a small cost for the memory barrier before the roots are captured, but this cost is negligible.

Figure 3.6 shows the overhead of our collector for various heap sizes of all the applications. Here, we selected a 25% collection threshold to balance eagerness and excessive collection. For all applications except *TSP* and *Clustering* our collector quickly drops to having negligible overhead.

Clustering is our largest and most complicated application with the heap divided among 19 data types. Its collection overhead does eventually become negligible, but not until 6.8× the maximum working set. The garbage collection overhead on *TSP* is near zero, even at the minimum viable heap size, because *TSP* is not memory-intensive due to the distance matrix, which remains live for the duration of the application dominating the working set. With little generated garbage, the application never has to pause for the collector to catch up.

3.5.5 Memory Bandwidth Overhead

To verify our hypothesis that the collector is filling otherwise unused memory cycles, we collected the cycle-by-cycle memory traces shown in Figure 3.7. Here, time runs from left to right and the rows in each trace represent different (parallel) heaps. Black squares represent application accesses, green squares represent collector accesses, and white squares represent unused memory cycles.

The top trace depicts our Bandwidth micro-benchmark, which makes heavy use of memory. The collector is near the beginning of the mark phase on all five heaps. It pauses briefly in the middle of this trace, possibly because of a "bubble" in the arriving roots, then resumes marking. The highly periodic pattern of the application's memory accesses is not interrupted during the bubble, confirming that our collector is able to share memory bandwidth with the application without interfering.

The bottom trace depicts our MergeSort benchmark at a point where the application is periodically using just two of the heaps and the collector is transitioning from marking to sweeping. The left side of the trace shows the mark phase finishing on the top heap; the other heaps have completed marking and are waiting to start sweeping. At about the halfway point, all heaps begin sweeping, which saturates the unused memory bandwidth. As with the memory-heavy benchmark, this change does not interfere with the application's progress (the black squares), which continue their periodic behavior.

The previous experiment shows the garbage collector is able to fill idle memory slots regardless of the application memory utilization, but that application memory utilization likely effects the garbage collection overhead as the garbage collector can only operate in the background if there are idle memory slots to use.

We varied the application memory bandwidth usage via the Bandwidth micro-benchmark and simulated collection triggered at 25% free space, the threshold with the best performance. Figure 3.8 shows the results. The thinnest line shows the GC overhead of the application using 50% of available memory slots (the smallest memory usage), and the line thickness increases as the memory usage increases.

The collector is sensitive to application memory utilization with the difference between the overhead at 50% utilization and 90% utilization ranging from about 35% to as much as 75% and

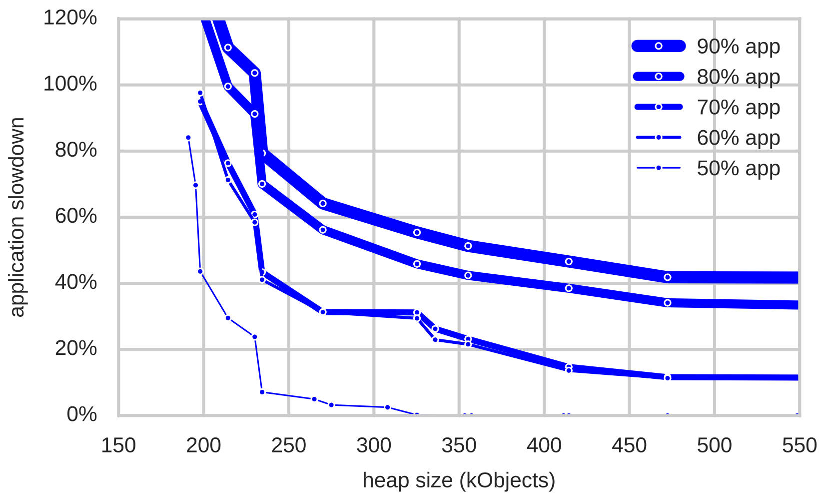

Figure 3.8: Collection overhead in cycles as a function of application memory utilization. As the application uses more memory slots there are not enough idle memory slots for the concurrent collector to operate in the background, so the overhead increases.

each successive utilization incurring a larger GC overhead. At the high end of the utilization range, the concurrent collector performs about the same as or even worse than a stop-the-world collector. When the application is using most of the memory slots, the concurrent collector cannot stay ahead of allocation. When the allocator inevitably runs out of memory, it must stop servicing allocation requests until the collector catches up. This does not necessarily stop the application because reads and non-memory operation can continue, but it may depending on the application and the length of the pause.

3.6 Conclusions

We presented a synthesized concurrent hardware garbage collector for FPGA-based accelerators. Our collector exploits rapid synchronization to fill otherwise idle memory slots on multiple, parallel memory heaps. We find that an eager collector that runs early and often paired with reasonably provisioned heaps allows the collector to run almost completely in the background. This brings the benefits of automated memory management to HLS, further reducing development time.

Chapter 4: Automated Power Domain Insertion and Control in Dataflow Circuits

Power gating is a well-established technique for reducing leakage power by turning off idle logic. However, applying it effectively requires identifying appropriate power domains and generating precise control logic—both challenging tasks that are typically done manually. This chapter presents a methodology for automatically identifying and controlling fine-grained power domains in dataflow circuits synthesized from high-level specifications. By leveraging the compositional structure and local activity knowledge of the dataflow abstraction, the approach partitions actors into always-on and sleepable regions, and synthesizes wake signals with provable correctness guarantees. This case study further supports the dissertation's thesis by showing how abstractions can reduce design effort but enable aggressive energy-saving optimizations that would be prohibitively complex to implement by hand.

4.1 Introduction

Leakage power—the power transistors consume when they are inactive—has become the dominant source of power consumption as transistors have gotten smaller [100]. Consequently, while more and more transistors can fit within the same area, they cannot all be powered on at the same time due to heat dissipation and battery life. Power gating, whereby idle parts of a chip are powered off while not in use, can be an effective tool to combat leakage. It has been shown that only about 10% of the logic gates in a typical circuit are active at any time [101, 102]. This exposes an opportunity to turn off idle gates saving up to 90% of leakage power.

These theoretical savings are unrealized in practice because they require small and fast power gates and identification and insertion of many small power domains. There has been progress on

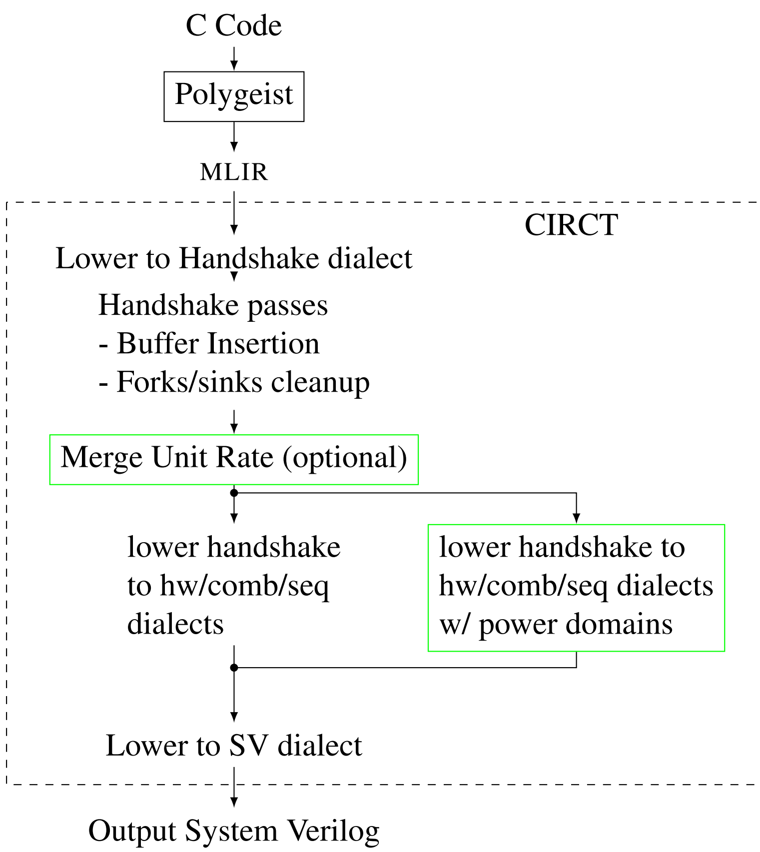

Figure 4.1: Compilation flow. The C program is lowered to the CIRCT Handshake dialect using existing tools and passes. In Handshake dialect we optionally merge unit rate actors before defining power domains and inserting control logic as part of the handshake to hardware lowering pass Section 4.3.

the first requirement: designing small and fast-acting power gates with a single cycle wake delay [103]. This work addresses the second challenge: how to identify and control many small power domains such that computation will not deadlock due to sleeping logic. We present algorithms to automatically insert and actuate fine-grained power domains. Our algorithms are implemented in the high-level synthesis (HLS) tool, CIRCT [19]. CIRCT is an open source hardware compiler which applies the LLVM/MLIR approach to compiler design. Specifically it offers a reusable infrastructure to define intermediate abstractions called dialects as it lowers MLIR to RTL. One such dialect, called Handshake, represents the circuit in dataflow form.

Dataflow circuits are a network of functional units called actors that communicate by transfer-ring data over channels according to a latency insensitive handshake. We leverage this representation to automatically identify power domains and insert power gate control logic to dynamically wake and sleep individual actors at runtime. The properties of dataflow that are most relevant to this power gating work are:

- *Precise knowledge of where there is activity.* Dataflow circuits communicate data tokens between actors using via latency insensitive channels. Because tokens flow only when *valid* and *ready* are both true, the handshake reveals exactly when and where tokens are flowing in a circuit. Because an actor needs to compute on specific combinations of input tokens, analysis of these signals can reveal whether an actor needs to compute and thus to be awake. The algorithms presented in Section 4.3.3 analyze these signals to generate the control signals that will sleep and wake the rest of the actor.

- *An identifiable control plane.* Analysis of these handshake signals also reveals which parts of the circuit control when tokens are produced and consumed and thus which parts of the circuit need to be always on to prevent deadlock. Because token transfers depend solely on the value of *valid* and *ready* signals, the control plane can be identified by tracing backward from these signals to find their transitive fanin as described in Section 4.3.1.

- *Patience.* Thanks to the latency-insensitive handshake described above, the latency required for each actor to compute does not effect correctness of the computation. This property becomes particularly valuable when power gating as a slow wakeup may impact performance but not correctness of a computation.

Using the properties of dataflow circuits, we are able to identify a set of states from which it is safe to remove and restore power to a domain without requiring costly state retention mechanisms. We are also able to identify the exact onset of idle and active periods without relying on software control or prediction heuristics. We prove that the resulting circuit is deadlock-free and produces the same results regardless of the latency of the power gate to wake or sleep each actor.

Our compilation flow, depicted in Figure 4.1, lowers C code to MLIR [76] using Polygeist [104]. Then we use several existing lowering passes to reach the Handshake dialect. Our passes are implemented as transforms within Handshake or integrated into the Handshake to HW lowering pass, which converts Handshake dialect to hw, comb, and seq dialects which together express digital logic. From the these dialects, our compilation flow converges with the existing path to produce RTL.

This work enables automated fine-grain power gating via the following contributions:

- an algorithm to automatically identify a power domain within each dataflow actor in a circuit;
- an algorithm to identify the set of states from which a power domain may sleep and reawaken in the reset state without changing the tokens produced by the actor;
- an algorithm to synthesize control logic to sleep and reawaken each power domain such that the token sequences are unchanged regardless of the time it takes the power domain to sleep or reawaken;
- and an automated flow, in the form of CIRCT passes, to power gate an entire circuit.

4.2 Power Gating Background

Power gating reduces leakage power by turning off transistors when they are inactive. A header or footer switch is inserted to disconnect transistors from the power supply. These switches can be inserted to control a single transistor or many transistors as a block. A region of a circuit controlled by a single switch is known as a *power domain*. While the concept of power gating is quite simple, correct implementation is challenging. Defining power domains requires identifying regions of correlated inactivity in the circuit. Defining control for a power domain requires identifying the beginning and end of idle periods in which it is both safe and beneficial to power down. Once the domains and control are defined, they require careful validation to preserve the circuit's function and avoid deadlock. These three steps are typically performed manually and therefore at a coarse

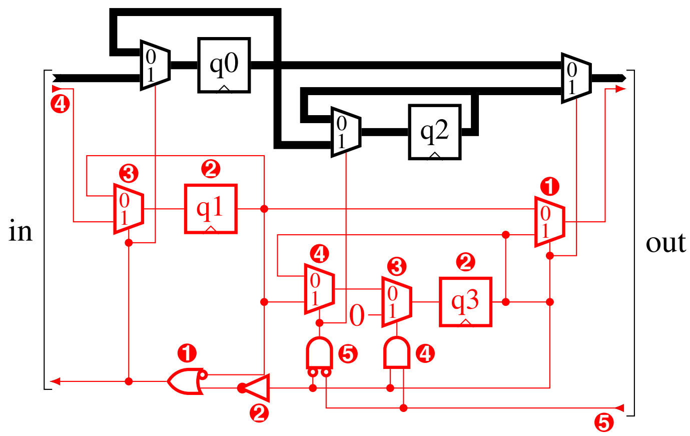

Figure 4.2: Partitioning a buffer into sleepable (data: black) and always-on (control: red) planes. The numbers indicate the order in which components are identified as being part of the control plane

granularity such as a functional unit or processor. Finally, there are physical design considerations, namely boundary isolation, state retention, gate design, and fluctuating voltage.

4.3 Power Gating Dataflow Circuits

Here, we present our procedure for power gating dataflow circuits. We start with an ungated dataflow circuit for each actor in a network, which we consider independently, split each into data circuitry that may be powered down and control circuitry that will always remain powered, identify the states in which the data circuitry may be power cycled without affecting future output tokens, synthesize a circuit that indicates when the data circuitry needs to be awake, and modify the *valid* and *ready* signals to account for any wake delay. Finally, we assemble the circuits for each power gated dataflow actor according to the original network topology. Below, we describe these steps in detail.

4.3.1 Partitioning into Control and Data Planes

Our first step is dividing the gates and registers of each actor's circuit into an always-on control plane and a data plane that we will eventually power gate. We will use the buffer circuit in Figure 4.2 from Edwards et al. [70] as a running example. The top of half of the circuit (thick wires indicate multi-bit buses) is a datapath that includes a register that holds buffered data (q0) and data when the downstream actor cannot accept it (q2). The handshake logic in the bottom half of the circuit includes two registers (q1, q3) that indicate whether the two datapath registers hold valid tokens and decides whether the buffer accepts and transmits data.

We define the control plane as all logic components that drive an actors outbound *valid* and *ready* signals. An actor's data plane is defined as all of the logic components that are not part of the control plane. To find the control plane, we start with the drivers of all *valid* and *ready* output signals in the actor's circuit and compute their transitive fanin, as shown in Figure 4.2. In this example, these two signals are driven by the OR gate and mux labeled 1. We then mark each (unmarked) component that drives any input of the newly marked components: registers q1 and q3 and the inverter. Then we mark the multiplexers labeled 3, the OR, mux, and *valid* input labeled 4, and finally the gates and *ready* input labeled 5. The algorithm stops because the drivers of these components are already marked.

4.3.2 Defining and Finding Idle States

To save power, we want to put the data plane to sleep as often as possible without changing the functional behavior of the actor. Sleeping in the reset state never loses data because our registers wake up in that state [105], but we can also sleep in states where discarding data will not affect the sequence of valid output tokens. We call these *idle states* and identify them as follows.

Our technique relies on knowing input- and output-don't-cares for our dataflow circuits: when a *valid* input is 0, we know that the corresponding *data* input will never affect the value of a valid output token (i.e., when a *valid* output is 1). As such, if we feed undefined data inputs to an actor with a *valid* input set to zero, we can conclude that these undefined values will never propagate to the outputs when a *valid* output is asserted. Our algorithm simply feeds such don't-care data into the circuit until it saturates; it follows that any undefined data in that state may be safely discarded, e.g., when the circuit is power-cycled.

We identify idle states—states whose data plane bits are either equal to those in the reset state or are don't-cares—by performing three-valued simulation and looking for a stable state. During the simulation, we model the contents of each register or wire as either a binary-valued vector or undefined (simply X).[1]

We attempt to find don't-care states in the data plane registers by feeding the actor don't-care inputs that we know will not propagate to the outputs. Throughout the simulation, we apply the same constant inputs: each *valid* input is set to 0, so that we can safely set each *data* input to X and know they will never propagate out of the actor as a valid token. Finally, we set each *ready* input to 1 to encourage the actor to "flush" any valid tokens it may hold.

We run the simulation until we find a state we have seen before, i.e., one in which the contents of the registers (either binary or X) match exactly. If it is a self-loop (i.e., a state in which the register values do not change) and the registers in the data plane are either the same as the reset state or X, we found the idle states. Otherwise, we conclude there are none.

This algorithm is sound because every state the algorithm identifies is an idle state, but it is not complete because it may not find every existing idle state. This approach is conservative, and ensures that we never power down in a state that is not actually an idle state.

Figure 4.3 illustrates how we find the idle states for the buffer in Figure 4.2. First, the four registers are reset to all 0s. From this state, we apply 0 to the *valid* input, 1 to the *ready* input and X to the *data* input. This only changes q0 to X because q3 = 0 sets the *ready* output to 1 and loads registers q0 and q1 from the *data* and *valid* inputs respectively (which have values X and 0). Furthermore, because q3 = 0 and the *ready* input is 1, registers q2 and q3 hold their values.

Next, we apply the same inputs and find the state remains unchanged because the four registers

[1] Components follow the usual three-valued rules: AND is 0 if any input is 0, 1 if all inputs are 1, and X otherwise; OR is 1 if any input is 1, 0 if all inputs are 0, and X otherwise; a multiplexer is X when its select input is X and passes one of its inputs otherwise; and each output of any other component is X if any input is X.

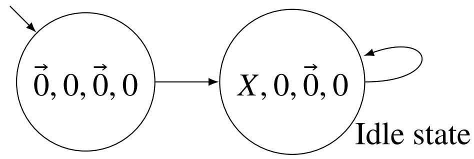

Figure 4.3: Finding the idle states for the buffer in Figure 4.2. In the reset state, registers q0, q1, q2, and q3 are all 0s ($\vec{0}$ denotes an all-zero vector); one step of the simulation sets q0 = X and another simulation step leaves the state unchanged, so we interpret any state with q1, q2, and q3 all zeros as an idle state.

are loaded in the same way. Because this strictly broadens the reset state of the data plane (q0 and q2 all zeros), we conclude we have an idle state when registers q1, q2, and q3 contain all zeros, meaning we can safely discard any data in q0.

Our algorithm has two other possible outcomes. If the stable state does not broaden the reset state of the data plane (e.g., some 0s became 1s), we conclude there are no idle states aside from the reset state. We also conclude this if the simulation reaches a state it saw earlier (e.g., when it is not a self-loop, such as you would find in a counter).

4.3.3 Computing the Wake Signal

At this point, we have identified the part of an actor that can be powered down (the data plane) and cases where we can safely discard the state held in that part (idle states). Next, we will synthesize logic that will indicate when an actor's data plane must be powered on.

An actor needs to be awake when it has work to do or must hold its state. This is true when

- the actor is not in an idle state, or
- the actor is about to transition from the idle state, or
- the actor is offering at least one valid output token.

The first and third conditions are easy to justify and test: if the actor is outside the idle state, the data plane is likely holding state that will matter and must remain powered. Similarly, if the actor is offering a token, it must hold that token's value until it is consumed and thus remain powered.

The second condition is easy to justify (if the actor is about to leave the idle state, it will start to hold data in the data plane, which requires it to be powered on), but harder to test. To synthesize the logic that tests this condition, we perform a partial brute-force simulation to determine which input patterns will bring the circuit out of an idle state then build a circuit that identifies these input patterns.

Together, these three conditions ensure that the actor will be awake when it has the inputs necessary to produce a valid output token now or in the future. The third condition is true if the actor can produce a valid token immediately. If instead the actor has work to do that will produce a future token, it will not be in the idle state, which is the first condition. The second condition ensures that even if the actor does not have data to produce a token in the future, it will wake up if the environment is indicating it is trying to deliver such data.

Our algorithm for determining when the actor may leave the idle state starts by eliminating *non-relevant* inputs, i.e., those that can never modify the idle state. Specifically, we calculate the transitive fanin from each register in the idle state that is not X, passing through both combinational components and registers. This gives relevant inputs: a subset of the actor's input signals that generally includes single-bit and vector values. Inputs that are not relevant may still flow into circuit registers that are X in the idle state, but doing so does not bring the circuit outside the idle state because the Xs indicate where the register contents do not matter.

Next, we simulate the circuit for one step from the idle state under every combination of relevant single-bit input values. This is potentially exponential, but is acceptable in practice because the number of such inputs is limited by the number of actor ports, not their width in bits. In particular, we do not consider all combinations of vector-valued inputs.

As before, we apply three-valued simulation, but its purpose is slightly different here. We set each register to its value in the idle state, including any X-valued registers to indicate they are not holding useful data. We also set all vector-valued inputs to X to conservatively model a variety of values.

If, starting from the idle state, applying a set of concrete 0s and 1s to relevant single-bit inputs and Xs to all others, we find a change in the contents of any register whose idle value was not X, we record that pattern of inputs as prompting a transition. There are three cases: when a register with a binary value changes to a different binary value, the difference means we left the idle state. When a binary value changes to an X, we conservatively assume a change has occurred. If a register started as X changes, it does not matter because the X indicated the idle state was not concerned with the contents of the register.

We perform this simulation from the idle state and test for all combinations of "relevant" single-bit inputs, which produces a Boolean function for *transition* (in truth-table form) of the relevant single-bit inputs. We synthesize logic for this function as part of the wake signal computation.

Finally, the *wake* signal is the logical OR of three conditions:

$$wake = \overline{idle} \cdot awake + transition + \sum_{o \in \text{outputs}} o\_valid$$

where *idle* is true when the circuit is in an idle state (i.e., all non-X data plane registers have their reset value) and not sleeping (*awake*; see the next section) and the third term is true if any output's *valid* signal is true. The original transition condition was "if the actor is in the idle state and is leaving," but Boolean simplification lets us drop the idle state from this term since it appears as the first term in *wake*.

For the buffer actor in Figure 4.2, our algorithm proceeds as follows. Earlier, we determined the idle state was $(\text{q0}, \text{q1}, \text{q2}, \text{q3}) = (X, 0, \vec{0}, 0)$, so we compute the transitive fanin of registers q1, q2, and q3. For this circuit, every input is relevant: { *in_valid*, *out_ready*, *in_data* }. For example, *in_data* is relevant because the fanin of q2 includes q0, whose fanin includes *in_data*.

Next, we simulate the circuit from the idle state $(X, 0, \vec{0}, 0)$ for each of the four combinations of the single-bit inputs; the in_data vector is set to X. This gives the following results:

in_valid	*out_ready*	*in_data*	**q0**	**q1**	**q2**	**q3**	*transition*
0	0	X	X	0	X	0	1
0	1	X	X	0	$\vec{0}$	0	0
1	0	X	X	1	X	0	1
1	1	X	X	1	$\vec{0}$	0	1

This function can be expressed as

$$transition = in\_valid + \overline{out\_ready}.$$

Because the idle state is $(X, 0, \vec{0}, 0)$,

$$idle = \overline{\mathrm{q1}} \cdot (\mathrm{q2} \stackrel{?}{=} \vec{0}) \cdot \overline{\mathrm{q3}}$$

where $\mathrm{q2} \stackrel{?}{=} \vec{0}$ is true when every bit in the q2 register is zero.

Finally, because this actor only has a single *valid* output,

$$wake = \overline{idle} \cdot awake + transition + out\_valid.$$

4.3.4 Instrumenting the Circuit

The previous steps have now given us enough information to safely add power gating to the actor circuit. Figure 4.4 illustrates how we do this for the buffer in Figure 4.2.

First, if the data plane is non-empty, we create a new power domain containing all the components in the data plane whose interface is all the wires that cross between the data and control planes. In our example, the data plane contains all the non-red components on the bottom of Figure 4.2, so we create the power domain shown in blue in Figure 4.4 whose interface includes the mux select signals, and the two data buses.

To the new power domain, we add a *wake* input signal that controls when the domain should

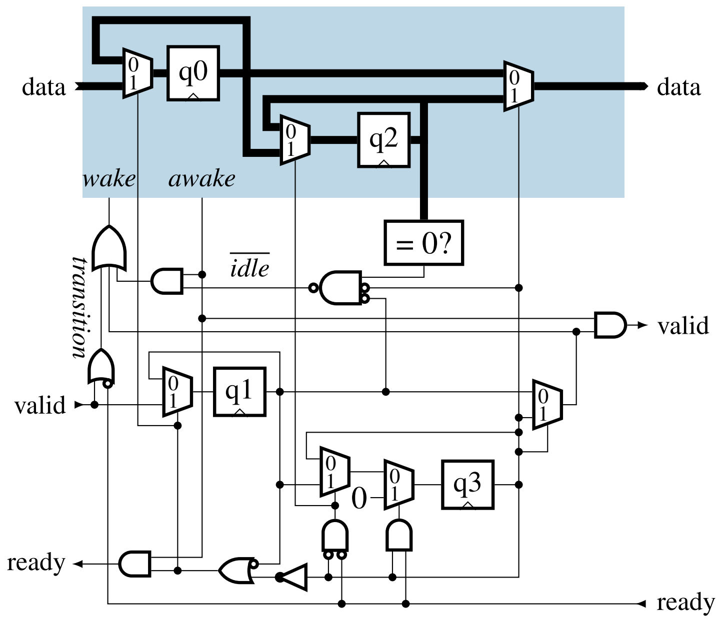

Figure 4.4: The result of instrumenting the buffer of Figure 4.2. The blue area indicates the new power domain, which is controlled by the *wake* signal and produces the *awake* signal to indicate when it has been powered on.

be powered on and an *awake* output that indicates the domain is powered on. The timing between these two signals may vary (as described in a later section); our instrumented actors respond to *awake*.

We add logic for generating the *wake* signal according to the rules in the previous section. In Figure 4.4, this is the OR gate driving *wake* that combines the *idle*, *transition*, and *valid* outputs. Note that the *transition* input computes $transition = in\_valid + \overline{out\_ready}$, the middle input is just the *valid* signal on the (single) output port, and the *idle* signal detects when the actor is in the idle state (the output of q1 and q3 must both be 0 and the output of q2 must be all zeros, which must be added to the interface of the new power domain), is inverted, and is ANDed with *awake*.

The final step is to gate all the *valid* and *ready* outputs with the *awake* signal to prevent the transfer of tokens until the data plane is powered up. In Figure 4.4, these are the two AND gates on the *ready* output in the lower left and the *valid* output on the right. The rest of the circuitry is unchanged from that of Figure 4.2.

This algorithm maintains the invariant that **the result of computation will be equivalent to the case of always on**. We have established that the sequence of valid output tokens will be unchanged by removing and restoring power the the sleepable domain (Section 4.3.1, Section 4.3.2) and that the sleepable domain will always wake when needed (Section 4.3.3), so the remaining danger is in tokens being dropped during the handshake. In dataflow, tokens are transferred when the producing actor offers a valid token and the receiving actor is ready to accept it. So for an actor to drop an input token, its outgoing *ready* signal must be high before it is ready to accept the data, i.e. when the data plane is powered down. This is not possible because the algorithm redefines all outgoing *ready* signals as the logical and of *awake* and the original value, so when the actor is not awake all *ready* signals will be zero and no tokens will be transferred. For an actor to drop a valid output token, it must be powered down after it has produced a valid output token and before the downstream actor has accepted it. This is not possible because when an actor is offering a valid output token the corresponding *valid* output signal will be true so *wake* will be true and the data domain will not be powered down until the token is transferred.

4.4 Evaluation

In this section, we characterize the dynamic activity of each sleepable power domain in a set of test circuits. We also evaluate the performance impact of power gating as a function of the physical wake time.

4.4.1 Test Circuit Characterization

We evaluate our techniques using kernels from the PolyBench benchmark suite listed Table 4.1 [104]. Using CIRCT, we lowered each kernel from C to SystemVerilog to generate three different test circuit configurations:

- a *baseline* circuit without any power domains or gating
- a *gated* circuit with power domains and control signals defined according to Section 4.3

		C Code Lines	Num. Actors	# Actor Types	# Power Domains	% Domains With Average Idle Period 0	0-10	10-100	100-1k	1k-10k	10k+
ADI	*Alt. Direction Implicit solver*	53	992	65	541	31	0	41	0	19	9
Durbin	*Toeplitz system solver*	28	285	51	138	36	1	38	0	20	5
FDTD 2D	*2D finite different time domain*	24	646	61	352	38	0	32	0	23	7
Floyd Warshall	*Dynamic prog. shortest path*	13	165	41	76	32	3	36	9	11	11
Heat 3D	*heat eq. over 3D data domain*	30	619	53	335	28	0	45	13	0	14
Jacobi 1D	*1D Jacobi stencil computation*	16	183	40	90	32	0	49	0	16	3
Jacobi 2D	*2D Jacobi stencil computation*	16	359	45	186	30	0	44	0	19	7
LU	*LU decomposition*	18	254	41	119	30	0	37	0	26	7
LU Decomp	*LU decomp. then forward subs.*	37	513	56	252	36	0	15	2	28	19
MM2	*2 matrix multiplications*	26	384	54	198	36	0	28	0	21	15
Nussinov	*Dynamic prog. for seq. alignment*	24	598	74	252	25	1	15	0	33	25
Seidel 2D	*2D Seidel stencil computation*	12	248	49	120	27	39	10	0	13	11
TriSolve	*Triangluar solver*	14	117	41	48	27	33	2	0	35	2

Table 4.1: 13 test circuits ranging from 117 to 992 actors, yielding 48 to 541 power domains. Average idle period for each power domain at right of table, shows the bulk of inactivity in the 10-1000 cycle range.

- a *merged-gated* circuit with adjacent unit rate actors merged into one actor prior defining power domains

Each of the test circuits listed in Table 4.1 have hundreds of actors, of which tens to hundreds are partitioned into sleepable and always-on domains. This is far more power domains than could be found and verified manually.

Table 4.1 also shows the breakdown of power domains by average idle period, ranging from 0 clock cycles (i.e., never idle) to over 10k clock cycles per idle period. We note that approximately one third of the power domains in each of our test circuits never become idle during simulation. This percentage exactly matches the percentage of buffers in the circuit, indicating that buffer actors are never powered off. This is partially due to the limited set of idle states found for buffers described in Section 4.3.2. It is also partially due to the nature of buffers which is to hold tokens, and if a dataflow circuit has buffers which rarely hold tokens then they are probably not necessary in the first place.

For the domains which do have idle periods, the numbers are promising, with the majority of idle periods lasting 10 cycles or more. For periods between 10 and 1000 cycles, fine-grained power gating can provide energy savings, provided these periods are successfully identified at runtime. The remaining idle periods, of 1000 cycles or longer, are also excellent targets for power gating, but given their length, they might have been identified via other methods.

The exceptions to these trends are Seidel2D and TriSolve, which have a large number of power domains with idle periods lasting less than 10 cycles. We hypothesize that these circuits have data paths where branch actors produce tokens on alternating outputs resulting in short idle and active periods.

4.4.2 Relative Wakefulness

To evaluate the impact of this fine-grained power gating on the circuit's power and performance, we simulate the gated circuit and the un-gated baseline in a cycle-accurate RTL simulator, Verilator. In each simulation of a power-gated circuit, we impose a wake delay: the number of cycles between when a power domain is told to wake up (*wake* = 1) and when it is ready to work (*awake* = 1). In these experiments, delays range from an idealized 0 cycles to 3 cycles.

First, we examine the relative active time of the gated and ungated versions, as seen in Wakefulness data on the left side of Table 4.2. These numbers report the percentage of power domains that are powered on in an average clock cycle. We observe that no more than 40% of the power domains are on in an average cycle. We further note that this activity level is essentially flat, regardless of the wake delay. This is because, when a dataflow actor is emitting a result, it must remain awake and continue offering its output tokens until the receiving actor is ready to accept them. Consequently, the length of each active period for each producer increases commensurate with the time to wake the consumer. However, while the rate of token transfers is slowed down by the wake delay, the sequence of valid tokens - and thus the order in which actors wake and sleep - is unchanged. This results in the same amount of wakefulness in each cycle despite the increased wake delay.

4.4.3 Relative Performance

Next, we compare the runtime, in clock cycles, of the power gated circuit against the ungated baseline, the results of which are shown in Table 4.2. The slowdowns are substantial because our

	Un-Merged Power Domains								Merged Power Domains							
	Wakefulness				Cycles				Wakefulness				Cycles			
Wake Delay:	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
ADI	0.34	0.34	0.33	0.33	1.00	2.29	3.61	4.95	0.37	0.37	0.37	0.37	1.00	1.97	2.99	4.03
Durbin	0.40	0.40	0.39	0.39	1.00	2.12	3.39	4.65	0.42	0.42	0.41	0.41	1.00	1.97	3.00	4.04
FDTD 2D	0.40	0.40	0.40	0.40	1.00	2.15	3.44	4.70	0.42	0.42	0.42	0.42	1.00	1.80	2.66	3.53
Floyd Warshall	0.39	0.39	0.38	0.38	1.00	1.92	2.93	3.93	0.42	0.42	0.41	0.41	1.00	1.75	2.49	3.24
Heat 3D	0.33	0.34	0.33	0.33	1.00	2.28	3.56	4.88	0.39	0.39	0.39	0.39	1.00	1.77	2.63	3.52
Jacobi 1D	0.39	0.39	0.38	0.38	1.00	2.06	3.13	4.24	0.40	0.40	0.39	0.39	1.00	1.87	2.81	3.73
Jacobi 2D	0.35	0.35	0.34	0.34	1.00	2.27	3.68	5.08	0.38	0.38	0.38	0.37	1.00	1.85	2.78	3.71
LU	0.36	0.36	0.36	0.35	1.00	1.92	2.82	3.75	0.39	0.39	0.39	0.38	1.00	1.73	2.45	3.19
LU Decomp	0.38	0.38	0.38	0.38	1.00	1.85	2.85	3.86	0.40	0.41	0.40	0.40	1.00	1.69	2.34	3.04
MM2	0.40	0.40	0.40	0.40	1.00	1.92	2.87	3.84	0.42	0.42	0.42	0.42	1.00	1.72	2.44	3.17
Nussinov	0.28	0.28	0.28	0.27	1.00	2.01	3.13	4.30	0.33	0.32	0.32	0.32	1.00	1.87	2.78	3.73
Seidel 2D	0.38	0.37	0.36	0.36	1.00	2.45	4.02	5.63	0.40	0.40	0.39	0.39	1.00	1.82	2.68	3.54
Seidel 2D*	0.45	0.46	0.46	0.46	1.00	1.17	1.35	1.53	0.43	0.43	0.42	0.42	1.00	1.33	1.69	2.03
TriSolve	0.37	0.37	0.36	0.35	1.00	1.80	2.68	3.61	0.38	0.38	0.37	0.37	1.00	1.67	2.35	3.02
TriSolve*	0.44	0.44	0.44	0.44	1.00	1.00	1.01	1.02	0.44	0.44	0.44	0.44	1.00	1.14	1.32	1.49

Table 4.2: Runtime in clock cycles and wakefulness of gated circuit relative to ungated baseline. The measurements on the left are of power gating applied to an unmodified circuit, and on the right power gating applied after merging unit-rate actors. For two test circuits marked with an asterisk, we eliminated all power domains with an average idle period of less than 10 cycles and reran simulation.

fine-grained power domains are power cycled very frequently and often have very short active periods as tokens ripple through the network.

However, while our method finds an abundance of power domains, it is not necessary to insert all of them. For example, we removed all power domains with an average idle period less than 10 cycles during simulation from the TriSolve and Seidel 2D test circuits. This increased the relative wakefulness of each circuit from 36% to 44% and from 37% to 46%, respectively, but significantly reduced the performance overhead to at most 1.02x the ungated baseline for TriSolve and 1.53x the ungated baseline for Seidel 2D.

We also observe that one could amortize the cost of multi cycle wake delays by combining actors into slightly larger power domains which are power cycled less frequently. We tested the effect of merging actors by merging adjacent unit rate actors into a single larger actor prior to the lowering pass where we apply our algorithms. We target unit rate actors, because they have particularly short active periods as they hold no state and thus remain awake only as long as there is new work being offered. Additionally, unit rate actors will not be ready to accept inputs until a downstream state-holding actor is also awake and ready to accept the outputs. When waking multiple adjacent unit rate actors, each actor must wake fully before the next begins waking, creating a multiplying factor to the wake delay.

The results of merging adjacent unit rate actors prior to power gating are shown in the right side of Table 4.2. The relative wakefulness of the circuit increases with merging, because the entire merged actor must be awake when any portion needs to be awake. We observe that this bump in activity is offset by faster overall execution. However, depending on the wake delay, the slowdowns remain significant. We believe there is opportunity to apply further merging, for example targeting the divergence and convergence points of disjoint execution paths and merging each path into a single power domain.

4.4.4 Synthesis Results

To evaluate the area overhead of our instrumented circuits, we used OpenLane [106] to perform synthesis and technology mapping for each of our test circuit configurations with the SkyWater SKY130 PDK. The instance area of the test circuits ranged from 73,857 to 842,267µm^2. The area overhead of all but two of our gated circuits relative to the baselane ranges from 4% to 40% with an average of 22% area overhead. The two outlying circuits, ADI and Durbin, have an area overhead of 204% and 222%, respectively. The baseline configuration of these two circuits were unable to complete the OpenLane flow, indicating that there may be factors in the original circuit design affecting the synthesis process.

4.5 Related Work

Manual power gating is difficult because designers must insert power-gating switches and verify the functional correctness and timing of each power domain. One approach to automate power gating uses information from a clock-gated netlist to partition a circuit into power domains [107, 108, 109]. These works find clock gating conditions based on don't cares in the circuit, then identify clock gated registers which can be grouped into power domains. Upasani et al. focus on the timing of circuits that are both clock and power gated and use a placement aware clustering heuristic to reduce timing overhead [110]. Udupi et al. derive temporary unobservabilities as part of a formal verification process which can be used for both clock and power gating [111]. Usami and Yoshioka propose a technique to reduce leakage power at runtime by identifying when state transitions do not occur [112]. All of these techniques require the design to already be clock gated, whereas our technique derives the same information directly from the structure of the dataflow circuit and does not require that previous step.

Another method of automating clock gating is to exploit architectural properties of finite state machines. N. Agarwal and Dimopoulos and Shin et al. use a FSM with data path (FSMD) representation in which the machine is partitioned with data storage and functional circuits [113, 114]. These FSMDs can be further partitioned into independent sub-FSMDs that can be powered off when the state transitions into a different sub-FSMD. Like our method, these works rely on the properties of a specific representation to define power domains, however under their scheme, only one power domain can be active at a time.

Leakage power is equally a concern in reconfigurable architectures which can be power gated at the LUT or tile granularity. Ishihara et al. and Bsoul and Wolton designed FPGAs for fine grained power gating where the power domains can be powered on and off at runtime using the routing fabric between blocks [115, 116]. Miniskar et al. presents a programmer directive based power gating scheme to turn off unused tiles in a CGRA architecture and Nayak et al. focus on reducing the area overhead of boundary protection in a power-gated CGRA [117, 118]. These methods depend on the specific device whereas our methodology is more general.

Like us, other researchers have used high-level information about the circuit to drive power gating. Fanni et al. use the dataflow intermediate output of the MPEG Reconfigurable Video Encoding standard to map multiple dataflow networks onto a power gated CGRA [119]. They identify common logic regions among the dataflow networks which once mapped to CGRA tiles can be powered down when none of the corresponding dataflow networks are active. Our work also derives power domain partitions using a dataflow representation, but we partition a single dataflow network and generate control signals from the dynamic activity in the network. A. Agarwal and Arvind leverage the rules-based design of Bluespec to generate fine-grained power domains and their control signals [120]. Like our dataflow representation, each rule has a firing condition which determines when the rule is active that can be used to determine power gating control signals. However, Bluespec scheduling does not allow rules to be activated over multiple cycles so their methodology requires that power domains are powered on within a clock cycle, limiting the potential clock frequency. Our methodology allows for multi-cycle actors and is robust to variable wake up delays.

4.6 Conclusions

We present a method for automatically defining and controlling fine-grained power domains in a dataflow circuits. Our compile-time algorithm partitions each dataflow actor into an always-on control plane and a sleepable data plane. We insert logic to dynamically control the sleep signal. The resulting circuits operate correctly under variable wake latencies.

Chapter 5: Compiler-Directed Task Dispatch for Chiplet Accelerators

The work in Chapter 3 and Chapter 4 demonstrates how a dataflow abstraction can simplify the implementation of complex hardware features. This chapter shifts focus to the software interface, showing how design abstractions facilitate task orchestration in heterogeneous architectures. We introduce a new MLIR [76] dialect, *Accel*, which provides an abstraction for compiler-directed, pull-based task scheduling on chiplet-based accelerators. By encoding task graphs and dependency hints, this abstraction enables the compiler to guide dynamic scheduling decisions at runtime, mitigating the inefficiencies inherent in static scheduling.

5.1 Introduction

As heterogeneous computing platforms become increasingly prevalent, accelerators such as GPUs, AI engines, and specialized domain-specific processors are being integrated more tightly into system-on-chip (SoC) and chiplet-based architectures. While these accelerators offer significant gains in performance and energy efficiency, they also introduce new challenges for software systems attempting to offload work effectively. Existing accelerator interfaces rely heavily on host-driven, push-based execution models, in which tasks are enqueued statically into FIFO structures and dispatched without awareness of task readiness or dependency structure. While effective for accelerators that execute long-running, coarse-grained tasks such as GPUs, this model often exacerbates head-of-line (HOL) blocking, load imbalance, and resource underutilization when faced with irregular or data-dependent computation.

These limitations are especially pronounced in task-graph-based programs, where execution order is determined by data dependencies between fine-grained tasks. Domains such as machine learning increasingly rely on asynchronous task graphs to expose parallelism and adapt to data-driven execution, with frameworks such as PyTorch using task graphs to represent the operations and data dependencies in neural networks. However, current accelerator dispatch models lack the flexibility needed to efficiently traverse and schedule such graphs.

In this work, we propose a compiler-directed, pull-based dispatch model that shifts the responsibility for task scheduling from the host CPU to the accelerator. Rather than pushing individual tasks into queues, the host transmits an entire task graph object, which describes the tasks, their dependencies, and compiler-inserted scheduling directives. The accelerator interprets this graph, dynamically managing execution based on task readiness and runtime feedback.

To support this execution model, we introduce *Accel*, a custom dialect in the MLIR framework that exposes the accelerator interface as a first-class programmable entity. By leveraging the compiler's global view of the program's control and data flow, our model enables the accelerator to dispatch only tasks that are ready and defer others until their dependencies are met.

5.2 Background on Accelerator Task Dispatch

The conventional task dispatch model, depicted on the left side of Figure 5.1, typically exposes accelerators to host software as one or more queues, each with a signaling mechanism referred to as a doorbell. A software scheduler running on the host CPU populates these queues with task packets as well as barriers and other synchronization operations. When the host enqueues a task, it will notify the accelerator device via the doorbell signal. Each task packet describes a discrete unit of work to be executed by one of the accelerator's processing elements and typically includes an operation, references to input data, and task-specific configuration parameters.

On the accelerator side, a dedicated command processor is responsible for draining these queues and dispatching tasks to its processing elements. The command processor functions as a packet processor, removing packets from the queue and processing them in order. It dispatches tasks to processing elements, stalls on barriers, and signals the host when tasks complete. The command processor relies entirely on the enqueued synchronization operations to ensure dependencies are resolved before dispatch.

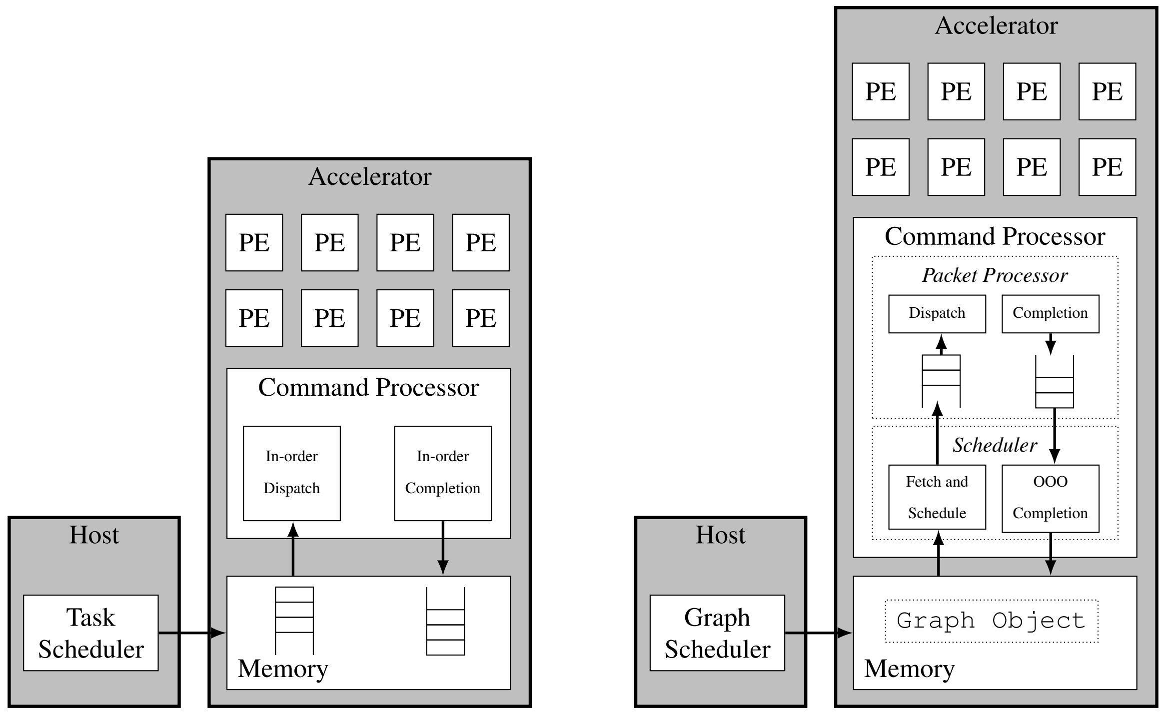

Figure 5.1: On the left is the traditional push-based dispatch architecture where the host schedules tasks for the accelerator command processor to dequeue and dispatch in order. On the right is our pull-based dispatch architecture with an extended command processor which traverses and schedules a task graph using compiler-inserted hints.

While the queue-based interface provides a clean abstraction over low-level hardware and simplifies software-hardware interaction, static scheduling of tasks to queues leads to inefficiency. Since task packets are processed in order, a barrier at the front of the queue prevents all other tasks in the queue from being dispatched, even ones that are ready to execute. One way to mitigate this head-of-line blocking is to schedule tasks to multiple queues to enable more precise barrier placement. In systems with multiple queues, dispatch typically uses simple, fixed algorithms such as round-robin to pull tasks from multiple queues. However, the benefits are limited by the number of queues and may introduce load imbalance depending on how queues are mapped to accelerators. These inefficiencies become especially pronounced with short-running tasks that have many dependencies, such as those in task graphs.

5.3 Pull-Based Dispatch Model

In place of push-based dispatch, we propose pull-based dispatch which shifts task scheduling responsibilities from the host processor to the accelerator's command processor, as shown on the right in Figure 5.1. Rather than enqueueing individual tasks, the host CPU dispatches an entire task graph object to the accelerator. This graph encodes tasks, their dependencies, and compiler-inserted scheduling directives that enable the command processor to reason about task readiness.

To support this execution model, the accelerator command processor architecture is significantly extended. While it still includes a packet processor to dequeue and dispatch tasks to accelerator processing elements, it is augmented with a scheduler CPU that interprets the task graph and populates the dispatch queues. This scheduler maintains two task pools: a ready pool, containing tasks whose dependencies have been resolved, and a sleeping pool, containing tasks awaiting the completion of others. Guided by compiler-inserted scheduling hints and dependency information embedded in the task graph, the scheduler interprets task readiness and pushes only ready tasks to the dispatch queues. Task completion signals from the processing elements enable the scheduler to dynamically update the task graph state.

This dynamic scheduling can be enhanced by the compiler's global view of the task graph. With full visibility into a program's control flow and data flow, the compiler can statically determine task dependencies and insert instructions to guide scheduling at runtime. This logic is encoded in a custom MLIR dialect, *Accel*, which introduces operations for defining task graphs, tasks, and scheduling metadata. These operations are eventually lowered into command processor bytecode, where they assist the scheduler in interpreting the graph structure.

This pull-based model is more scalable and adaptable than traditional host-driven scheduling. By combining static compiler analysis with dynamic runtime feedback, the command processor is able to interpret complete task graphs. Tasks are dispatched as they become ready, enabling improved accelerator utilization.

5.4 *Accel* Dialect

The *Accel* dialect provides a high-level, programmable abstraction for task graphs, dependencies, and scheduling behavior. Unlike traditional compute-focused dialects (such as MLIR's GPU or AIE dialects), *Accel* is concerned purely with task orchestration. It expresses the structure and control of computation, not the computation itself.

The design of *Accel* is motivated by two key goals. First, we aim to represent task graphs in a way that is both hardware-agnostic and semantically rich, enabling the compiler to encode dependency and scheduling logic in a form that the command processor can interpret and execute. Second, we aim to bridge the gap between high-level domain-specific IRs and low-level device dispatch code, allowing fine-grained compiler optimizations and flexible lowering to diverse hardware targets.

5.4.1 Dialect Structure

At the core of the dialect are three fundamental types: a graph, a task, and a fiber. A *graph* represents a full task graph with potentially hundreds or thousands of nodes. Each *task* corresponds to a unit of computation with a well-defined set of dependencies and a unique identifier. A *fiber* is a subset of a task graph that is scheduled together and executed serially.

To define these structures, the dialect introduces a set of graph construction operations that encode dependency information, describe data movement between tasks, and dispatch the graph to the accelerator.

5.4.2 Compilation Flow

Accel is designed to compose seamlessly with other MLIR dialects, as shown in Figure 5.2. It sits below high-level domain-specific dialects such as those for linear algebra, image processing, or machine learning, allowing compiler passes to first perform domain-aware optimizations before lowering into the accel dialect for task orchestration. From there, the IR can be further lowered

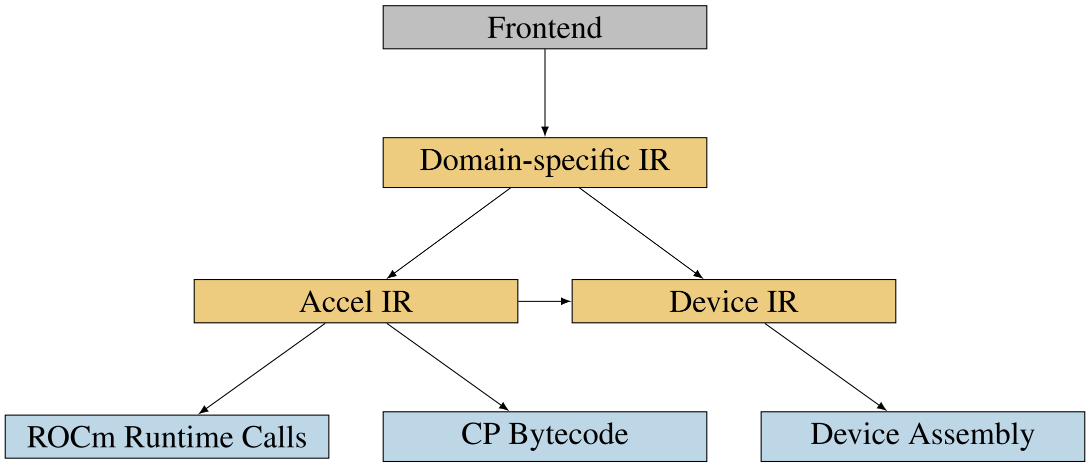

Figure 5.2: Compilation flow for *Accel*, a task graph IR for generic accelerator interfaces that can be lowered command processor bytecode, calls into runtime libraries, op hardware-specific dialects.

to command processor bytecode, calls into runtime libraries, or hardware-specific dialects for devices like GPUs or AIE arrays. This flexibility means that *Accel* is not restricted to any specific architecture or device.

5.4.3 Fiber Abstraction and Partitioning

As task graphs grow in size and complexity, the overhead associated with scheduling and dispatching individual tasks becomes non-trivial. Fine-grained scheduling, while precise, can introduce significant runtime overhead, particularly in systems where each task incurs setup latency and memory movement.

Fibers serve two primary goals: scalability and data locality. From a scalability standpoint, grouping tasks into fibers reduces the total number of schedulable entities that the command processor scheduler must manage. Rather than tracking and dispatching thousands of individual tasks, the command processor scheduler operates on a smaller number of fibers, each carrying its own internal dispatch logic. This abstraction aligns well with resource-constrained command processors or simple scheduler CPUs, enabling them to scale execution without becoming a bottleneck.

From a locality perspective, fibers allow the compiler to co-locate interdependent tasks on the same device or compute tile. This reduces the need for inter-device communication and enables reuse of data already present in local memory. For example, in a sparse triangular solver workload, many tasks operate on overlapping rows of a matrix. Grouping these into a single fiber allows intermediate results to remain in on-chip memory, avoiding costly DRAM transfers.

While fibers are dynamically scheduled by the command processor, the tasks in each fiber are statically scheduled by the compiler to run sequentially on the same processing element. Each fiber contains a sequence of operations that specify the schedule using the following *Accel* operations:

- `InitCountOp` sets the number of unsatisfied dependencies before a fiber becomes ready.
- `DispatchTaskOp` triggers task execution.
- `BarrierOp` manage inter-fiber synchronization.
- `WakeFiberOp` signals readiness of dependent fibers.
- `FiberTerminatorOp` indicates completion.

We adopt a wait-count-based synchronization mechanism for the initial fiber wake-up. Each fiber begins with a defined number of dependencies, which is decremented as tasks or external signals complete. When the count reaches zero, the fiber is ready for execution. The command processor scheduler moves the fiber into the ready pool and executes the instruction stream of *Accel* ops.

During execution, tasks in a fiber may depend on other fibers. To support flexible synchronization, we introduce a lightweight synchronization barrier. Unlike traditional barriers, which stall all execution until resolved, our `BarrierOp` affects only the specific fiber in which it resides. When a barrier op is encountered, the fiber returns to the sleep pool, and the barrier becomes its wake condition. This allows other fibers and tasks to continue executing in parallel, improving overall system throughput. Additionally, the `WakeFiberOp` enables fibers to be woken up early if specific conditions are met, such as the completion of just the initial tasks they depend on, rather than waiting for coarse-grained dependencies across the entire graph.

```
%f1 = accel.scheduled_fiber
{
  accel.init_count 1
  accel.dispatch_task %t4
  accel.dispatch_task %t8
  accel.dispatch_task %t9
  accel.barrier %t10
  accel.dispatch_task %t13
  accel.terminate_fiber
}
```

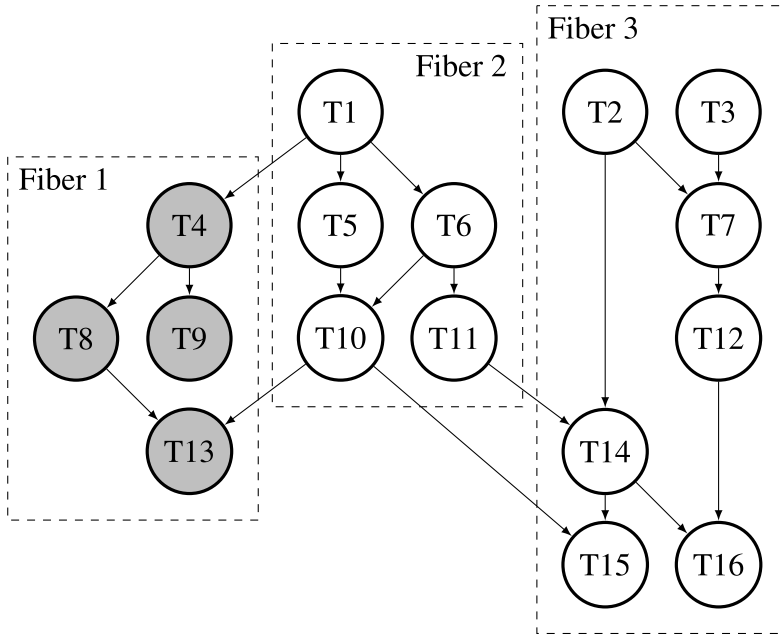

Figure 5.3: On the right is one possible partitioning of a task graph into fibers. On the left is the scheduled fiber instruction stream that corresponds to Fiber 1 (shaded).

A partitioning pass during compilation constructs fibers primarily for data locality using a depth-first policy that creates long chains of dependent tasks that can be executed on the same tile. Once the task graph is partitioned, a compiler pass transforms each fiber into a scheduled fiber that encodes its execution plan as a sequential stream of operations. The scheduler interprets this stream like a program, maintaining a per-fiber program counter and executing operations in order. If a wait or barrier condition is encountered, the fiber is moved to the sleeping pool; when the condition is resolved, it is returned to the ready pool. This model retains the dynamic, data-driven execution semantics of our pull-based scheduler while significantly reducing runtime control decisions by limiting the number of entities that must be scheduled.

An example of a scheduled fiber is shown in Figure 5.3. The `InitCountOp` indicates the number of tasks that must complete before the fiber is ready, in this case only task T1. Once task T1 is complete, the command processor scheduler will notify its waiters, including this fiber, and decrement their counters. When the count goes to zero, the fiber will be moved to the ready pool where it can dispatch task T4 and then tasks T8 and T9. At that point, the fiber encounters an external dependency, and the barrier op will prompt the scheduler to move the fiber into the sleeping pool until task T10 is complete. Once task T10 is complete, the fiber will move back into the ready pool and dispatch task T13.

5.5 Evaluation

To evaluate the effectiveness of our compiler-directed, pull-based dispatch model, we conducted a series of experiments using sparse triangular solver benchmarks running on a prototype system. These experiments assess how our model improves task dispatch latency, total execution time, and system scalability compared to traditional push-based scheduling. We also examine the impact of partitioning a task graph into fibers and evaluate execution strategies that exploit data locality.

5.5.1 Prototype System

We built a prototype targeting the AMD Versal platform [121]. This prototype system brings together programmable command processors, high-performance accelerator arrays, and a fully integrated compiler and runtime stack to demonstrate end-to-end task graph execution.

Our prototype is deployed on the AMD Versal VCK190, a heterogeneous system-on-chip that integrates general-purpose processors, reconfigurable logic, and a dense array of AI Engine (AIE) accelerator tiles. These AIE tiles are designed for high-throughput, low-latency computation and are ideal for task-parallel workloads like sparse triangular solvers.

The platform's control logic is split between an ARM Cortex-A72 scheduler CPU embedded in the Versal processing subsystem and eight BlackParrot RISC-V processors [122] synthesized in the programmable logic fabric. The BlackParrot cores serve as packet processors, executing bytecode generated from *Accel* and managing communication between the scheduler and the accelerator devices. Each BlackParrot core maintains its own instruction memory and supports task dispatch operations, queue management, and signal handling. A shared hardware queue descriptor memory region stores task queues, doorbell signals, and shared flags. This memory region is exposed to both the host and the accelerator as a prefetchable PCIe base address register, enabling efficient user-space interaction with the queueing system.

At the top of the software stack, the MLIR compiler incorporates the *Accel* dialect and corresponding lowering passes. Task graphs are transformed into *Accel* ops containing scheduling directives and fiber operations. This representation is then compiled into CP-compatible bytecode and linked with the application.

We extend the AMD ROCm runtime (ROCr) [123], an open-source software stack for GPU accelerator computation, to support arbitrary accelerators by abstracting devices as generic agents with queues, signals, and memory regions. Our implementation generalizes the ROCr agent model to include the AIE array and the command processor complex. This makes our system fully compatible with the HSA programming model [124] while allowing for dynamic, asynchronous task execution.

At the kernel level, we implement a custom Linux device driver for the AIE and CP complex. The driver handles device memory allocation, queue registration, and agent creation. Once the initial queues and contexts are established, all task scheduling occurs in user space, reducing overhead and latency.

During execution, the scheduler CPU runs a compact runtime loop that manages the state of all active fibers. Each fiber has an associated program counter that tracks the next operation to be interpreted. When a fiber is moved to the ready pool, the scheduler fetches the next instruction, executes it (e.g., dispatching a task), and either continues or places the fiber back into the sleeping pool depending on the instruction type.

The BlackParrot packet processors are responsible for enqueuing tasks into hardware queues, triggering dispatches to the AIE tiles, and handling task completions. Once an accelerator task completes, a signal is sent to the scheduler CPU, which updates the appropriate wait counts or barriers and determines if any fibers are now ready to execute.

Crucially, task and fiber execution is data-driven: a fiber only progresses when its wait conditions have been met. This model supports dynamic task graph traversal without requiring the host to maintain complex scheduling logic, reducing the overhead typically seen in push-based dispatch systems.

5.5.2 Applications

We use a set of task graphs derived by applying the sparse triangular solver algorithm to a variety of input matrices resulting in a set of task graphs with different sizes, shapes, and dependencies [125]. The input matrices are drawn from the SuiteSparse Matrix Collection [126, 127], and are symmetric and sparse, with between 2,000 and 6,000 rows. The graphs are constructed using a tiling approach that groups non-zero entries into manageable subgraphs based on a target task capacity, producing task graphs with 300 to 4,000 tasks.

Each task in the graph represents a single unit of work and includes dependencies based on data requirements. These dependencies form a directed acyclic graph (DAG), which is then used to generate the *Accel* ops for compilation and dispatch.

5.5.3 Push vs. Pull Dispatch

We first compare our pull-based model to the conventional push-based model, which statically populates hardware queues regardless of task readiness. The results are shown in Figure 5.4. Across the full set of SpTS graphs, we observe a speedup of between 4% and nearly 50% in total execution time using the pull-based model. While all graphs show some speedup, those with less parallelism experience diminishing returns as the number of queues increases. These gains stem primarily from improved queue utilization and reduced waiting time for ready tasks.

5.5.4 Fiber Execution and Locality-Aware Scheduling

Next, we evaluated the use of locality-based fibers compared to the baseline pull-based model. We compare two execution strategies: a basic round-robin scheduler without fibers and a locality-aware scheduler that places all tasks in a fiber on the same AIE tile. The goal is to reduce memory

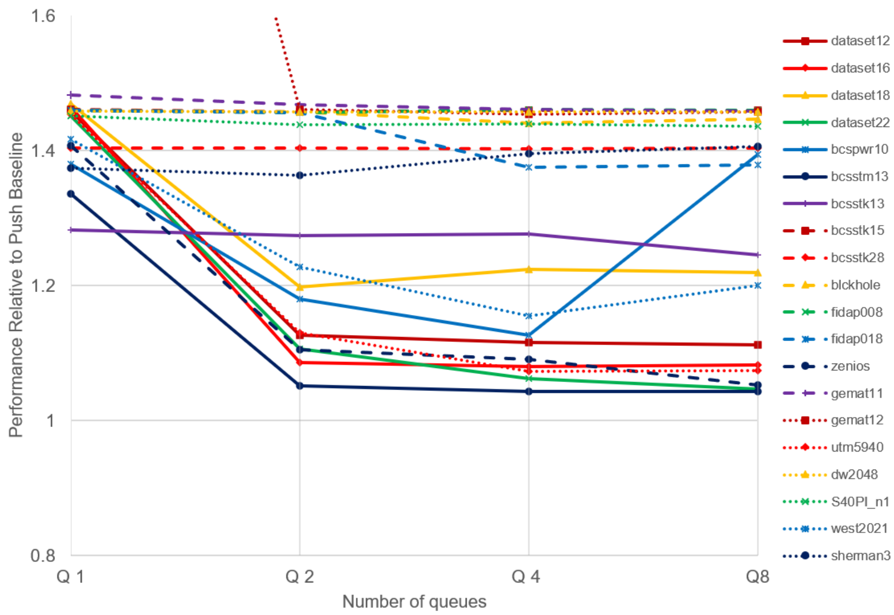

Figure 5.4: Performance of pull-based dispatch relative to baseline push-based dispatch. All graphs show some amount of speedup, but the performance impact of the number of queues depends on the paralellism in the graph.

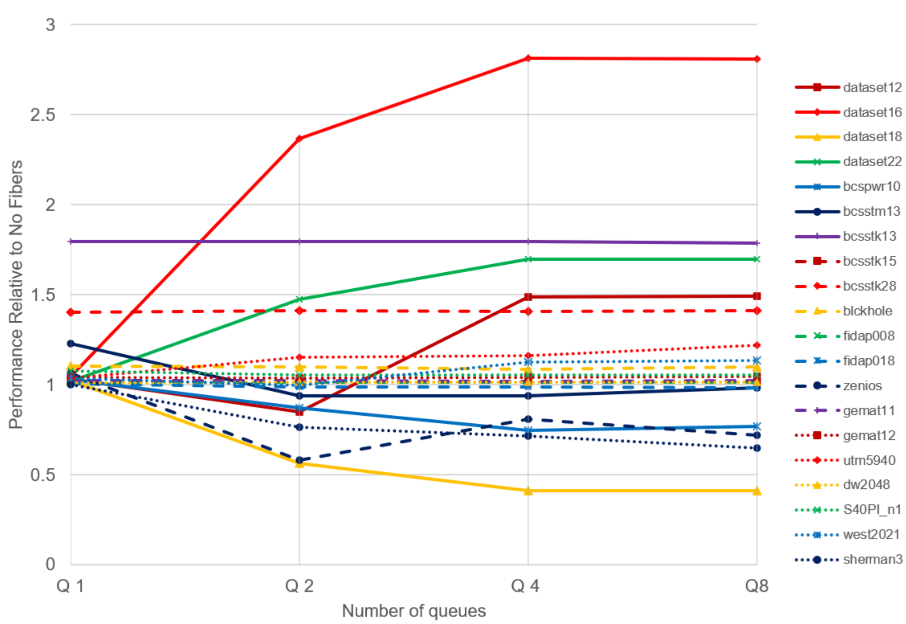

Figure 5.5: Performance of pull-based dispatch with fibers relative to pull-based dispatch without fibers.

transfers and exploit on-tile data reuse.

The results in Figure 5.5 show that for most graphs locality-aware scheduling improves execution time compared to the round-robin baseline. The performance gain is driven by two main factors: a reduction in the number of instructions executed by the command processor and a reduction in data movement between tiles.

The performance of locality-based schedule is closely correlated to the parallelism in each task graph. Graphs with many data dependencies exhibit a tradeoff between limiting the number of data transfers and serializing otherwise parallel tasks.

While fiber-based scheduling offers strong performance in many cases, we observed that overly aggressive partitioning can lead to unintended serialization, reducing parallelism and increasing critical path length. In some graphs, especially those with wide, highly parallel structures, locality-aware grouping reduced concurrency to the point where performance degraded relative to the flat, per-task model.

Given that the effectiveness of fiber scheduling is strongly correlated with task granularity and hardware mapping, the workload has a significant impact on performance In workloads like sparse triangular solve, where individual tasks are relatively small and data movement dominates execution time, locality-based execution with fibers can significantly reduce memory traffic and improve performance. However, in compute-bound or highly parallel tasks, the benefit may be less pronounced, and serialization penalties may outweigh locality gains.

5.6 Related Work

Prior work targeting task-graph dispatch primarily focuses non-chiplet accelerators. Mosayebi et al. studied queue management in PIM accelerators to identify bottlenecks in large-scale graph workloads [128]. Their approach uses improved scheduling heuristics for PIM cores to reduce waiting time and increase utilization by optimizing the scheduling of work queues. Tzeng et al. explored dynamic task management for irregular workloads on GPUs [129]. They find that task donation and task stealing result in significantly less idleness than both centralized and distributed

FIFO queues. We use a different mechanism, but also find that dynamic scheduling improves performance.

Several MLIR dialects target parallel execution and heterogeneous systems, but they focus on compute rather than task dispatch and graph-level orchestration. The MLIR GPU dialect provides abstractions for launching compute kernels on GPUs. It models thread hierarchies and synchronization within a kernel, but assumes a push-based dispatch model [130]. Similarly, the AIE dialect, designed for AMD's AI Engines, is closely tied to the AIE tile architecture [131]. These dialects are designed to optimize computation for specific device architectures and do not represent task graphs or scheduling hints as *Accel* does. The MLIR Async dialect models asynchronous control flow using composable async regions [132]. It is integrated with a coroutine-style execution model and does not represent the topology of a task graph. It also does not facilitate dynamic task dispatch by the command processor.

In the machine learning domain, Katel et al. present an MLIR-based code generation pipeline for NVIDIA tensor cores, using several MLIR dialects including GPU [133]. However, their focus is on kernel generation and optimization, and they use the conventional task dispatch model. Martinez et al. define the HDNN dialect as an abstraction for deep neural networks [134]. HDNN emphasizes device portability and programming productivity and can be lowered to CPU, GPU, or TPU backends by changing the device specifier, which differs in focus from *Accel* .

Beyond MLIR, frameworks like Legion [135], Kokkos [56], and HPVM [136] support task graph programming and dynamic scheduling. These frameworks operate primarily with runtime libraries at a higher level than our pull-based model, but they could be supported by our model at the hardware level. Wei et al. recently introduced Cohort, a software-oriented framework for accelerator management based on shared-memory queues [137]. However, they focus on single-producer, single-consumer queueing models.

5.7 Conclusions

We present a method for offloading entire task graphs to chiplet-based accelerators. Leveraging *Accel*, we tightly integrate compiler-driven scheduling with a programmable command processor, enabling fine-grained control over task orchestration. Our prototype, implemented on the Versal AIE platform, demonstrates how this co-designed approach can reduce execution time, improve resource utilization, and adapt flexibly across diverse architectures. By combining hardware programmability, a rich intermediate representation, and a custom runtime scheduler, we offload complex orchestration logic directly onto the accelerator. This architecture enables a pull-based scheduling model that outperforms traditional push-based dispatch, particularly in scenarios involving sparse and irregular workloads.

Chapter 6: Conclusions

The increasing use of heterogeneous architectures to meet performance and energy efficiency demands has introduced new challenges in accelerator design and system integration. Compared to traditional single-core and multicore architectures, heterogeneous architectures require more specialized hardware design and more complex system-level coordination and programming models. This dissertation has argued that design-time abstractions are a powerful mechanism for managing this complexity. In three case studies, this work has demonstrated that carefully selected abstractions can reduce development effort while enabling features that would otherwise be difficult to implement.

Chapter 3 and Chapter 4 showed how a dataflow abstraction can simplify the design and synthesis of complex hardware features. In Chapter 3, we presented a synthesized hardware garbage collector for FPGA accelerators. By leveraging the latency-insensitive and parallel nature of dataflow circuits, the collector operates concurrently with the application, exploiting idle memory cycles with negligible performance overhead. This contribution demonstrates that automatic memory management, typically done in software, can be efficiently realized in hardware using a dataflow abstraction.

Chapter 4 demonstrated how the dataflow abstraction can also be used to facilitate power gating for accelerators. Here, the precise knowledge of activity inherent to dataflow circuits allowed us to identify fine-grained power domains and synthesize control logic in dataflow circuits. The resulting power-gated accelerator operates correctly even with variable wake latencies and ensures that power domains wake when needed. This method makes fine-grained power gating practical for synthesized accelerators.

Chapter 5 shifted focus to accelerator interfaces and presented a method for dispatching task graphs on chiplet-based accelerators. Unlike traditional push-based dispatch, where the software host statically schedules tasks in queues, our pull-based dispatch model enables the accelerator to dynamically schedule tasks at runtime. To support this execution model, we introduce *Accel*, a new MLIR dialect designed to represent task graphs, dependencies, and scheduling hints. This combination of static compiler analysis with dynamic runtime feedback avoids the bottlenecks of push-based dispatch.

Collectively, these contributions support the thesis that *design-time abstractions are essential to effective accelerator design and integration, and their careful selection can substantially simplify the implementation of key features*. The abstractions in this dissertation enable automated garbage collection and fine-grained power gating for synthesized accelerators as well as dynamic task dispatch for chiplet-based accelerators, demonstrating how abstractions facilitate the design and integration of accelerators.

References

[1] R. H. Dennard, F. H. Gaensslen, H.-N. Yu, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted mosfet's with very small physical dimensions," *IEEE Journal of solid-state circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[2] M. Bohr, "A 30 year retrospective on dennard's mosfet scaling paper," *IEEE Solid-State Circuits Society Newsletter*, vol. 12, no. 1, pp. 11–13, 2007.

[3] G. E. Moore, "Cramming more components onto integrated circuits," *Proceedings of the IEEE*, vol. 86, no. 1, pp. 82–85, 1998.

[4] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, pp. 33–38, 2008.

[5] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," in *2014 IEEE international solid-state circuits conference digest of technical papers (ISSCC)*, IEEE, 2014, pp. 10–14.

[6] H. Mair *et al.*, "23.3 a highly integrated smartphone soc featuring a 2.5 ghz octa-core cpu with advanced high-performance and low-power techniques," in *2015 IEEE International Solid-State Circuits Conference-(ISSCC) Digest of Technical Papers*, IEEE, 2015, pp. 1–3.

[7] J. Pyo *et al.*, "23.1 20nm high-k metal-gate heterogeneous 64b quad-core cpus and hexa-core gpu for high-performance and energy-efficient mobile application processor," in *2015 IEEE International Solid-State Circuits Conference-(ISSCC) Digest of Technical Papers*, IEEE, 2015, pp. 1–3.

[8] N. D. Lane *et al.*, "Deepx: A software accelerator for low-power deep learning inference on mobile devices," in *2016 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, IEEE, 2016, pp. 1–12.

[9] J. Park *et al.*, "A 646gops/w multi-classifier many-core processor with cortex-like architecture for super-resolution recognition," in *2013 IEEE International Solid-State Circuits Conference Digest of Technical Papers*, IEEE, 2013, pp. 168–169.

[10] K. H. Lee and N. Verma, "A low-power processor with configurable embedded machine-learning accelerators for high-order and adaptive analysis of medical-sensor signals," *IEEE Journal of Solid-State Circuits*, vol. 48, no. 7, pp. 1625–1637, 2013.

[11] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th annual international symposium on computer architecture*, 2017, pp. 1–12.

[12] A. M. Caulfield *et al.*, "A cloud-scale acceleration architecture," in *2016 49th Annual IEEE/ACM international symposium on microarchitecture (MICRO)*, IEEE, 2016, pp. 1–13.

[13] A. Putnam, "Fpgas in the datacenter: Combining the worlds of hardware and software development," in *Proceedings of the Great Lakes Symposium on VLSI 2017*, 2017, pp. 5–5.

[14] J. Fowers *et al.*, "A configurable cloud-scale dnn processor for real-time ai," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2018, pp. 1–14.

[15] D. Firestone *et al.*, "Azure accelerated networking:{smartnics} in the public cloud," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018, pp. 51–66.

[16] T. Endo, S. Matsuoka, A. Nukada, and N. Maruyama, "Linpack evaluation on a supercomputer with heterogeneous accelerators," in *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, 2010, pp. 1–8.

[17] K. Li *et al.*, "A vector systolic accelerator for multi-precision floating-point high-performance computing," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 69, no. 10, pp. 4123–4127, 2022.

[18] H. Vo, Y. Lee, A. Waterman, and K. Asanovic, "A case for os-friendly hardware accelerators," in *Workshop on the Interaction between Operating System and Computer Architecture (WIVOSCA), at the International Symposium on Computer Architecture (ISCA)*, Citeseer, 2013.

[19] CIRCT. "Circuit IR Compilers and Tools." (), (visited on 09/21/2024).

[20] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA '11, San Jose, California, USA: Association for Computing Machinery, 2011, 365–376, ISBN: 9781450304726.

[21] M. B. Taylor, "Is dark silicon useful? harnessing the four horsemen of the coming dark silicon apocalypse," in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC '12, San Francisco, California: Association for Computing Machinery, 2012, 1131–1136, ISBN: 9781450311991.

[22] S. Borkar and A. A. Chien, "The future of microprocessors," *Commun. ACM*, vol. 54, no. 5, 67–77, 2011.

[23] M. Khazraee, L. Zhang, L. Vega, and M. B. Taylor, "Moonwalk: Nre optimization in asic clouds," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17, Xi'an, China: Association for Computing Machinery, 2017, 511–526, ISBN: 9781450344654.

[24] I. Kuon and J. Rose, "Measuring the gap between fpgas and asics," in *Proceedings of the 2006 ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays*, ser. FPGA '06, Monterey, California, USA: Association for Computing Machinery, 2006, 21–30, ISBN: 1595932925.

[25] T. Akenine-Möller and J. Ström, "Graphics for the masses: A hardware rasterization architecture for mobile phones," in *ACM SIGGRAPH 2003 Papers*, ser. SIGGRAPH '03, San Diego, California: Association for Computing Machinery, 2003, 801–808, ISBN: 1581137095.

[26] H. Le *et al.*, "Mobilecodec: Neural inter-frame video compression on mobile devices," in *Proceedings of the 13th ACM Multimedia Systems Conference*, ser. MMSys '22, Athlone, Ireland: Association for Computing Machinery, 2022, 324–330, ISBN: 9781450392839.

[27] K. Tsoumanis, S. Xydis, G. Zervakis, and K. Pekmestzi, "Flexible dsp accelerator architecture exploiting carry-save arithmetic," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 1, pp. 368–372, 2016.

[28] A. Pourhabibi *et al.*, "Optimus prime: Accelerating data transformation in servers," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20, Lausanne, Switzerland: Association for Computing Machinery, 2020, 1203–1216, ISBN: 9781450371025.

[29] J. Dean, D. Patterson, and C. Young, "A new golden age in computer architecture: Empowering the machine-learning revolution," *IEEE Micro*, vol. 38, no. 2, pp. 21–29, 2018.

[30] A. Parashar *et al.*, "Scnn: An accelerator for compressed-sparse convolutional neural networks," *SIGARCH Comput. Archit. News*, vol. 45, no. 2, 27–40, Jun. 2017.

[31] S. Han *et al.*, "Eie: Efficient inference engine on compressed deep neural network," *SIGARCH Comput. Archit. News*, vol. 44, no. 3, 243–254, Jun. 2016.

[32] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, 2012.

[33] M. M. Kim, M. Mehrara, M. Oskin, and T. Austin, "Architectural implications of brick and mortar silicon manufacturing," in *Proceedings of the 34th Annual International Symposium*

on Computer Architecture, ser. ISCA '07, San Diego, California, USA: Association for Computing Machinery, 2007, 244–253, ISBN: 9781595937063.

[34] S. Naffziger *et al.*, "Pioneering chiplet technology and design for the amd epyc™ and ryzen™ processor families : Industrial product," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 57–70.

[35] Y. Li, A. Louri, and A. Karanth, "Spacx: Silicon photonics-based scalable chiplet accelerator for dnn inference," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022, pp. 831–845.

[36] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad memory: Design alternative for cache on-chip memory in embedded systems," in *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, ser. CODES '02, Estes Park, Colorado: Association for Computing Machinery, 2002, 73–78, ISBN: 1581135424.

[37] X. Mei and X. Chu, "Dissecting gpu memory hierarchy through microbenchmarking," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 1, pp. 72–86, 2017.

[38] L. Benini and G. De Micheli, "Networks on chips: A new soc paradigm," *Computer*, vol. 35, no. 1, pp. 70–78, 2002.

[39] S. Sutherland, S. Davidmann, and P. Flake, *SystemVerilog for Design Second Edition: A Guide to Using SystemVerilog for Hardware Design and Modeling*. Springer Science & Business Media, 2006.

[40] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach, "An introduction to high-level synthesis," *IEEE Design Test of Computers*, vol. 26, no. 4, pp. 8–17, 2009.

[41] Cadence, *Stratus high-level synthesis*, `https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html`.

[42] AMD, *Vitis hls*, `https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vitis/vitis-hls.html`.

[43] R. Nane *et al.*, "A survey and evaluation of fpga high-level synthesis tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, 2016.

[44] S. Lahti, P. Sjövall, J. Vanne, and T. D. Hämäläinen, "Are we there yet? a study on the state of high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 5, pp. 898–911, 2019.

[45] R. Nikhil, "Bluespec system verilog: Efficient, correct rtl from high level specifications," in *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE '04.*, 2004, pp. 69–70.

[46] J. Bachrach *et al.*, "Chisel: Constructing hardware in a scala embedded language," in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC '12, San Francisco, California: Association for Computing Machinery, 2012, 1216–1225, ISBN: 9781450311991.

[47] R. Nigam, P. H. Azevedo de Amorim, and A. Sampson, "Modular hardware design with timeline types," *Proc. ACM Program. Lang.*, vol. 7, no. PLDI, Jun. 2023.

[48] Y. S. Shao and D. Brooks, *Research infrastructures for hardware accelerators*. Springer Nature, 2022.

[49] D. Luebke, "Cuda: Scalable parallel programming for high-performance scientific computing," in *2008 5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro*, 2008, pp. 836–838.

[50] A. Munshi, "The opencl specification," in *2009 IEEE Hot Chips 21 Symposium (HCS)*, 2009, pp. 1–314.

[51] P. Bauman *et al.*, "Introduction to amd gpu programming with hip," *Presentation at Oak Ridge National Laboratory. Online at: https://www. olcf. ornl. gov/calendar/intro-to-amd-gpu-programming-with-hip*, 2019.

[52] L. Dagum and R. Menon, "Openmp: An industry standard api for shared-memory programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[53] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "Openacc — first experiences with real-world applications," in *Euro-Par 2012 Parallel Processing*, C. Kaklamanis, T. Papatheodorou, and P. G. Spirakis, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 859–870, ISBN: 978-3-642-32820-6.

[54] A. Paszke *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32, Curran Associates, Inc., 2019.

[55] A. Sabne, *Xla : Compiling machine learning for peak performance*, 2020.

[56] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of parallel and distributed computing*, vol. 74, no. 12, pp. 3202–3216, 2014.

[57] D. A. Beckingsale *et al.*, "Raja: Portable performance for large-scale scientific applications," in *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2019, pp. 71–81.

[58] R. Keryell, R. Reyes, and L. Howes, "Khronos sycl for opencl: A tutorial," in *Proceedings of the 3rd International Workshop on OpenCL*, ser. IWOCL '15, Palo Alto, California: Association for Computing Machinery, 2015, ISBN: 9781450334846.

[59] J. B. Dennis and D. P. Misunas, "A preliminary architecture for a basic data-flow processor," *SIGARCH Comput. Archit. News*, vol. 3, no. 4, 126–132, 1974.

[60] C. Arvind and D. Culler, "Dataflow architectures," *Annual review of computer science*, vol. 1, no. 1, pp. 225–253, 1986.

[61] J. B. Dennis, "First version of a data flow procedure language," in *Programming Symposium, Proceedings Colloque Sur La Programmation*, Berlin, Heidelberg: Springer-Verlag, 1974, 362–376, ISBN: 3540068597.

[62] G. Kahn, "The semantics of a simple language for parallel programming," *Information processing*, vol. 74, pp. 471–475, 1974.

[63] S. Swanson *et al.*, "The wavescalar architecture," *ACM Trans. Comput. Syst.*, vol. 25, no. 2, 2007.

[64] M. Budiu, P. Artigas, and S. Goldstein, "Dataflow: A complement to superscalar," in *IEEE International Symposium on Performance Analysis of Systems and Software, 2005. ISPASS 2005.*, 2005, pp. 177–186.

[65] M. B. Taylor *et al.*, "Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ilp and streams," in *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ser. ISCA '04, München, Germany: IEEE Computer Society, 2004, p. 2, ISBN: 0769521436.

[66] K. Sankaralingam *et al.*, "Exploiting ilp, tlp, and dlp with the polymorphous trips architecture," in *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ser. ISCA '03, San Diego, California: Association for Computing Machinery, 2003, 422–433, ISBN: 0769519458.

[67] J. Cortadella, M. Kishinevsky, and B. Grundmann, "Synthesis of synchronous elastic architectures," in *2006 43rd ACM/IEEE Design Automation Conference*, 2006, pp. 657–662.

[68] L. Josipović, R. Ghosal, and P. Ienne, "Dynamically scheduled high-level synthesis," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '18, Monterey, CALIFORNIA, USA: Association for Computing Machinery, 2018, 127–136, ISBN: 9781450356145.

[69] R. Li, L. Berkley, Y. Yang, and R. Manohar, "Fluid: An asynchronous high-level synthesis tool for complex program structures," in *2021 27th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, 2021, pp. 1–8.

[70] S. A. Edwards, R. Townsend, M. Barker, and M. A. Kim, "Compositional dataflow circuits," *ACM Trans. Embed. Comput. Syst.*, vol. 18, no. 1, Jan. 2019.

[71] L. P. Carloni, "From latency-insensitive design to communication-based system-level design," *Proceedings of the IEEE*, vol. 103, no. 11, pp. 2133–2151, 2015.

[72] L. Carloni, K. McMillan, A. Saldanha, and A. Sangiovanni-Vincentelli, "A methodology for correct-by-construction latency insensitive design," in *1999 IEEE/ACM International Conference on Computer-Aided Design. Digest of Technical Papers (Cat. No.99CH37051)*, 1999, pp. 309–315.

[73] L. Carloni, K. McMillan, and A. Sangiovanni-Vincentelli, "Theory of latency-insensitive design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 9, pp. 1059–1076, 2001.

[74] J. Stanier and D. Watson, "Intermediate representations in imperative compilers: A survey," *ACM Comput. Surv.*, vol. 45, no. 3, Jul. 2013.

[75] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, 2004, pp. 75–86.

[76] C. Lattner *et al.*, "MLIR: Scaling compiler infrastructure for domain specific computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2021, pp. 2–14.

[77] T. Gysi *et al.*, "Domain-specific multi-level ir rewriting for gpu: The open earth compiler for gpu-accelerated climate simulation," *ACM Trans. Archit. Code Optim.*, vol. 18, no. 4, Sep. 2021.

[78] D. Klopp, S. Erdweg, and A. Pacak, "A typed multi-level datalog ir and its compiler framework," *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA2, Oct. 2024.

[79] J. Pienaar, *Mlir in tensorflow ecosystem*, Compilers For Machine Learning (C4ML) 2020, San Diego, CA, USA, 2020.

[80] H.-I. C. Liu, M. Brehler, M. Ravishankar, N. Vasilache, B. Vanik, and S. Laurenzo, "Tinyiree: An ml execution environment for embedded systems from compilation to deployment," *IEEE Micro*, vol. 42, no. 5, pp. 9–16, 2022.

[81] P. Ranganathan *et al.*, "Warehouse-scale video acceleration: Co-design and deployment in the wild," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 600–615.

[82] T. Cao, S. M. Blackburn, T. Gao, and K. S. McKinley, "The yin and yang of power and performance for asymmetric hardware and managed software," in *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2012, pp. 225–236.

[83] M. Maas, K. Asanović, and J. Kubiatowicz, "A hardware accelerator for tracing garbage collection," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2018, pp. 138–151.

[84] D. F. Bacon, P. Cheng, and S. Shukla, "And then there were none: A stall-free real-time garbage collector for reconfigurable hardware," *ACM SIGPLAN Notices*, vol. 47, no. 6, pp. 23–34, 2012.

[85] J. Jang *et al.*, "Charon: Specialized near-memory processing architecture for clearing dead objects in memory," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 726–739.

[86] J. A. Joao, O. Mutlu, and Y. N. Patt, "Flexible reference-counting-based hardware acceleration for garbage collection," in *Proceedings of the 36th annual international symposium on computer architecture*, 2009, pp. 418–428.

[87] R. Townsend, M. A. Kim, and S. A. Edwards, "From functional programs to pipelined dataflow circuits," in *Proceedings of the 26th International Conference on Compiler Construction*, ser. CC 2017, Austin, TX, USA: Association for Computing Machinery, 2017, 76–86, ISBN: 9781450352338.

[88] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *Proceedings of the 22nd annual international symposium on Computer architecture*, 1995, pp. 392–403.

[89] D. F. Bacon, P. Cheng, and S. Shukla, "Parallel real-time garbage collection of multiple heaps in reconfigurable hardware," *ACM SIGPLAN Notices*, vol. 49, no. 11, pp. 117–127, 2014.

[90] R. Jones, A. Hosking, and E. Moss, *The garbage collection handbook: the art of automatic memory management*. CRC Press, 2016.

[91] J. Tang, S. Liu, Z. Gu, X.-F. Li, and J.-L. Gaudiot, "Achieving middleware execution efficiency: Hardware-assisted garbage collection operations," *The Journal of Supercomputing*, vol. 59, no. 3, pp. 1101–1119, 2012.

[92] W. Srisa-an, C.-T. Lo, and J.-M. Chang, "Active memory processor: A hardware garbage collector for real-time java embedded devices," *IEEE Transactions on Mobile Computing*, vol. 2, no. 2, pp. 89–101, 2003.

[93] A. A. García, D. May, and E. Nutting, "Integrated hardware garbage collection," *ACM Trans. Embed. Comput. Syst.*, vol. 20, no. 5, 2021.

[94] M. Meyer, "An on-chip garbage collection coprocessor for embedded real-time systems," in *11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'05)*, IEEE, 2005, pp. 517–524.

[95] W. J. Schmidt and K. D. Nilsen, "Performance of a hardware-assisted real-time garbage collector," *ACM SIGPLAN Notices*, vol. 29, no. 11, pp. 76–85, 1994.

[96] C. Click, G. Tene, and M. Wolf, "The pauseless gc algorithm," in *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, 2005, pp. 46–56.

[97] M. Meyer, "A true hardware read barrier," in *Proceedings of the 5th international symposium on Memory management*, 2006, pp. 3–16.

[98] B. Cao, K. A. Ross, M. A. Kim, and S. A. Edwards, "Implementing latency-insensitive dataflow blocks," in *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, 2015, pp. 179–187.

[99] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 3, no. 1, pp. 63–75, 1985.

[100] N. Kim *et al.*, "Leakage current: Moore's law meets static power," *Computer*, vol. 36, no. 12, pp. 68–75, 2003.

[101] B. Geuskens and K. Rose, *Modeling microprocessor performance*. Springer Science & Business Media, 2012.

[102] R. Joseph and M. Martonosi, "Run-time power estimation in high performance microprocessors," in *Proceedings of the 2001 international symposium on Low power electronics and design*, 2001, pp. 135–140.

[103] J. P. Cerqueira and M. Seok, "Temporarily fine-grained sleep technique for near- and subthreshold parallel architectures," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 1, pp. 189–197, 2017.

[104] W. S. Moses, L. Chelini, R. Zhao, and O. Zinenko, "Polygeist: Raising c to polyhedral mlir," in *Proceedings of the ACM International Conference on Parallel Architectures and*

Compilation Techniques, ser. PACT '21, Virtual Event: Association for Computing Machinery, 2021.

[105] Y. Shin, J. Seomun, K.-M. Choi, and T. Sakurai, "Power gating: Circuits, design methodologies, and best practice for standard-cell vlsi designs," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 15, no. 4, Oct. 2010.

[106] M. Shalan and T. Edwards, "Building openlane: A 130nm openroad-based tapeout- proven flow : Invited paper," in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2020, pp. 1–6.

[107] L. Bolzani, A. Calimera, A. Macii, E. Macii, and M. Poncino, "Enabling concurrent clock and power gating in an industrial design flow," in *2009 Design, Automation & Test in Europe Conference & Exhibition*, 2009, pp. 334–339.

[108] M. Hamada *et al.*, "An automated runtime power-gating scheme," in *2007 25th International Conference on Computer Design*, 2007, pp. 382–387.

[109] J. Cong, B. Liu, R. Majumdar, and Z. Zhang, "Behavior-level observability analysis for operation gating in low-power behavioral synthesis," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 16, no. 1, 2010.

[110] G. Upasani, A. Calimera, A. Macii, E. Macii, and M. Poncino, "Reducing timing overhead in simultaneously clock-gated and power-gated designs by placement-aware clustering," in *Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation*, J. Monteiro and R. van Leuken, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 227–236, ISBN: 978-3-642-11802-9.

[111] S. Udupi, J. Urdahl, D. Stoffel, and W. Kunz, "Exploiting hardware unobservability for low-power design and safety analysis in formal verification-driven design flows," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 6, pp. 1262–1275, 2019.

[112] K. Usami and H. Yoshioka, "A scheme to reduce active leakage power by detecting state transitions," in *The 2004 47th Midwest Symposium on Circuits and Systems, 2004. MWSCAS '04.*, vol. 1, 2004, pp. I–493.

[113] N. Agarwal and N. Dimopoulos, "Fsmd partitioning for low power using ilp," in *2008 IEEE Computer Society Annual Symposium on VLSI*, 2008, pp. 63–68.

[114] C.-H. Shin, M.-H. Oh, S. N. Kim, and S. W. Kim, "Fine-grained power gating of datapath using fsm," in *2011 IEEE 2nd International Conference on Networked Embedded Systems for Enterprise Applications*, 2011, pp. 1–5.

[115] S. Ishihara, M. Hariyama, and M. Kameyama, "A low-power fpga based on autonomous fine-grain power gating," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 19, no. 8, pp. 1394–1406, 2011.

[116] A. A. M. Bsoul and S. J. E. Wilton, "An fpga architecture supporting dynamically controlled power gating," in *2010 International Conference on Field-Programmable Technology*, 2010, pp. 1–8.

[117] N. R. Miniskar, R. R. Patil, R. N. Gadde, Y.-c. R. Cho, S. Kim, and S. H. Lee, "Intra mode power saving methodology for cgra-based reconfigurable processor architectures," in *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2016, pp. 714–717.

[118] A. Nayak *et al.*, "Improving energy efficiency of cgras with low-overhead fine-grained power domains," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 16, no. 2, Apr. 2023.

[119] T. Fanni, C. Sau, L. Raffo, and F. Palumbo, "Automated power gating methodology for dataflow-based reconfigurable systems," in *Proceedings of the 12th ACM International Conference on Computing Frontiers*, ser. CF '15, Ischia, Italy: Association for Computing Machinery, 2015, ISBN: 9781450333580.

[120] A. Agarwal and Arvind, "Leveraging rule-based designs for automatic power domain partitioning," in *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2013, pp. 326–333.

[121] B. Gaide, D. Gaitonde, C. Ravishankar, and T. Bauer, "Xilinx adaptive compute acceleration platform: Versaltm architecture," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19, Seaside, CA, USA: Association for Computing Machinery, 2019, 84–93, ISBN: 9781450361378.

[122] D. Petrisko *et al.*, "Blackparrot: An agile open-source risc-v multicore for accelerator socs," *IEEE Micro*, vol. 40, no. 4, pp. 93–102, 2020.

[123] AMD, *Rocm platform runtime*, `https://github.com/RadeonOpenCompute/ROCR-Runtime`.

[124] H. Foundation, *Hsa platform system architecture specification*, `https://hsafoundation.com/standards/`, Accessed: 2025.

[125] M. Naumov, "Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the gpu," *NVIDIA Corp., Westford, MA, USA, Tech. Rep. NVR-2011*, vol. 1, 2011.

[126] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, Dec. 2011.

[127] S. P. Kolodziej *et al.*, "The suitesparse matrix collection website interface," *Journal of Open Source Software*, vol. 4, no. 35, p. 1244, 2019.

[128] M. A. Mosayebi, A. M. Hasani, and M. Dehyadegari, "Enhanced graph processing in pim accelerators with improved queue management," *Microelectronics Journal*, vol. 94, p. 104 637, 2019.

[129] S. Tzeng, A. Patney, and J. D. Owens, "Task management for irregular-parallel workloads on the gpu," *High Performance Graphics*, 2010.

[130] MLIR, *Gpu dialect*, `https://mlir.llvm.org/docs/Dialects/GPU/`, Accessed: 2025.

[131] J. Zhuang *et al.*, "Aries: An agile mlir-based compilation flow for reconfigurable devices with ai engines," in *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2025, pp. 92–102.

[132] MLIR, *Async dialect*, `https://mlir.llvm.org/docs/Dialects/AsyncDialect/`, Accessed: 2025.

[133] N. Katel, V. Khandelwal, and U. Bondhugula, "Mlir-based code generation for gpu tensor cores," in *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, 2022, pp. 117–128.

[134] P. A. Martínez, G. Bernabé, and J. M. García, "Hdnn: A cross-platform mlir dialect for deep neural networks," *The Journal of Supercomputing*, vol. 78, no. 11, pp. 13 814–13 830, 2022.

[135] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, IEEE, 2012, pp. 1–11.

[136] A. Ejjeh *et al.*, "Hpvm: Hardware-agnostic programming for heterogeneous parallel systems," *IEEE Micro*, vol. 42, no. 5, pp. 108–117, 2022.

[137] T. Wei, N. Turtayeva, M. Orenes-Vera, O. Lonkar, and J. Balkind, "Cohort: Software-oriented acceleration for heterogeneous socs," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023, Vancouver, BC, Canada: Association for Computing Machinery, 2023, 105–117, ISBN: 9781450399180.