SOAR: a Self-Optimizing Adaptive SoC on FPGAs

Minwoo Kang

Professor Duane A. Bailey, First Reader.
Professor Kelly Shaw, Second Reader.

A thesis submitted in partial fulfillment of the requirements for the
Degree of Bachelor of Arts with Honors in Computer Science

WILLIAMS COLLEGE
Williamstown, Massachusetts

May 9, 2020

Abstract

With the power utilization wall imposed on modern microprocessors, architectural research has turned to the use of specialized hardware as a source of continued performance scaling. As a result, System-on-a-Chip (SoC) architectures containing hardware accelerators have gained significant traction over the past years. Unfortunately, each accelerator has a specific set of tasks it can execute, and accelerator-based systems are often incapable of out-performing general-purpose processors outside their target application domains. In this work, we propose SOAR, a reconfigurable architecture that extends the range of hardware acceleration by adaptively tailoring its configuration of accelerators to the running workload. The SOAR hardware is built as a 64-bit RISC-V SoC with a collection of RoCC accelerators attached to a scalar, in-order Rocket core. Evaluation of the design implemented on a Xilinx Artix-7 FPGA under RISC-V Linux demonstrates that our RoCC accelerators can return speed-ups up to 10× and energy-efficiency improvements up to 170× over equivalent software implementations. Along with SOAR hardware, we also introduce a software infrastructure that infers dynamic power demands and accelerator utilization frequencies to determine the optimal configuration of on-chip accelerators. We propose the use of dynamic dispatch for switching function execution between using hardware and using software library calls. Finally, we explain how our proposed framework can be customized to enhance either energy-efficiency or performance, or any arbitrary optimization goal.

Acknowledgements

I express my deepest gratitude towards Professor Duane Bailey, for his remarkable dedication, clear advice and continued support throughout our journey investigating FPGAs, RISC-V, unclear documentations and perplexing hardware malfunctions. Without Duane's care and guidance, completing this thesis would not have been possible.

I would also like to thank Professor Kelly Shaw for her mentorship, heartfelt encouragements, and careful reviews. Kelly has been wonderful, both as an instructor and as a research advisor, and my understanding of computer architecture has been greatly enriched by learning from her.

Special thanks to Maddie Burbage, for her work on the combinatorial sequence generation accelerator, and Daniel Yu, for discussions on previous research related to this work.

Contents

1. Introduction

Today, we are living in a *new golden age* for computer architecture [1]. While the slowdown of Moore's law and the end of Dennardian scaling have imposed a *power utilization wall* [2] on modern processors, a number of exciting, novel approaches to architectural design have sprung up in response to the persisting demands for better processors in this era of *dark silicon* [3, 4]. A noticeable trend is the shift of focus from processor performance to energy-efficiency; since a simple scaling of on-chip transistor counts no longer returns improvements in performance, many chip architects have begun to explore the possibilities of trading chip space for power.

As a result, recent literature has highlighted the use of accelerators, which are specialized hardware only capable of performing a certain set of tasks but with greater performance and/or energy-efficiency than general-purpose processors (GPPs). A popular strategy has thus been to design heterogeneous System-on-a-Chip (SoC) architectures that couple a set of these accelerators with general-purpose cores—a particular example is UCSD's *conservation-core (c-core)* system that achieves notable gains in energy-savings through its collection of energy-efficient application specific integrated circuits (ASICs) [2]. However, a natural drawback to this approach is that each accelerator has a limited range of applications it can improve. On the other hand, real-world users are likely to demand processors to perform optimally for a variety of workloads. In response, architects may imagine a system with as many accelerators as possible, but realistically there are strict constraints on on-chip space and power. Hence, it is extremely important to carefully decide which collection of accelerators the architecture should contain.

Reconfigurable fabric, such as Field Programmable Gate Arrays (FPGAs), may provide a partial solution to such limitations of using accelerators. Although FPGAs are less space- and performance-efficient than ASICs, they offer the ability to dynamically reconfigure the system according to the given workload. Instead of hard-wiring a list of operations the SoC can accelerate, a reconfigurable implementation will be able to modify such a list on-the-fly. We even further suspect that, in many cases, the improvements from being able to tailor the hardware to the running

program can possibly outweigh the relative overhead from using FPGAs. It is also worth mentioning that FPGAs, unlike ASICs, offer the flexibility to patch the accelerator designs according to version updates of the target application. Therefore, FPGAs arguably have the potential for being ideal platforms for implementing accelerator-based SoCs.

However, the remaining problem is how to decide which set of accelerators the system should configure on the chip, at each given moment in time. In answering this question, we can perhaps leverage our knowledge on how the O/S decides which set of pages should be kept in main memory. The concept of a *working set* can be applied to the configuration of accelerators—if the system can reason about which accelerators will be most useful during a certain time frame, then the reconfigurable hardware can be directed to configure itself according to such reasoning. Furthermore, to make decisions towards optimizing energy-efficiency, this automated system could require information about the accelerator utilization rates and the time-varying on-chip power demands.

In this project, we aim to develop a framework for reconfigurable SoCs to adaptively decide their optimal set of accelerators. Our architecture will not only optimize efficiency by judiciously power-gating (i.e., turning off) less utilized accelerators, but also by reconfiguring the set of on-chip hardware units. Decisions about accelerator utilization will be made by a software infrastructure and be based on measurements of accelerator usage statistics and dynamic power demands. From the programmer's perspective, this work provides an additional abstraction layer between the operating system and the underlying hardware that exposes the parellelism of the specialized hardware for the user to capitalize on, while relieving the burden of the programmer having to decide on the details of tailoring the hardware configuration to the user-specific workload. We believe such autonomous *personalization of hardware* will allow users to maximize the efficiencies of dark silicon.

This work is organized as follows. Chapter 2 discusses the previous research on dynamic reconfigurable architectures; heterogeneous systems that dynamically decide whether code should be executed in hardware or in software; RISC-V SoC architectures; and power-adaptive systems. Chapter 3 presents the workflow of designing and implementing hardware on FPGAs, and Chapter 4 details our proposed self-optimizing, adaptive architecture. Chapter 5 explains the software framework through which our system power adaptively adjust its use of on-chip accelerators. In Chapter 6, we report the methodology and results of performance and power consumption measurements. Chapter 7 discusses future directions this work could pursue. Finally, Chapter 8 is a brief summary.

2. Previous Work

Previous research related to this work can be grouped into four main categories. First, there exists a large body of research on dynamically reconfigurable architectures, often implemented on Field Programmable Gate Arrays (FPGAs), with capabilities to tailor parts of the hardware to varying computational loads. Second, there has been a set of research projects that discuss frameworks for judiciously orchestrating code execution on heterogenous hardware components to achieve better energy-efficiency. Third, an increasing number of projects have been utilizing the RISC-V open-source hardware ecosystem to implement System-on-a-Chip (SoC) designs with special-purpose hardware as co-processors. Finally, many works have demonstrated architectures that involve Power Management Units (PMUs) evaluating power demands on-the-fly and accordingly making adjustments to the system. We consider these contributions in this chapter.

2.1 Dynamically Reconfigurable Architectures

Several research projects have explored architectures that include a reconfigurable hardware unit as a co-processor that enables the system to harness power efficiency by dynamically configuring the reconfigurable data path based on compile-time and/or run-time analysis.

The PRogrammable Instruction Set Computers (PRISC) microarchitecture developed by Razdan and Smith is one of the earlier attempts on coupling programmable hardware units with a General-Purpose Processor (GPP) [5]. The programmable functional units (PFUs) are designed to execute a set of combinational functions that can be computed within a single instruction cycle time. A custom reduced instruction set computing (RISC) instruction selects from up to 2048 possible PFU configurations and then returns the results of the computation. At a high level, the approach is very applicable to more modern hardware: they use compile-time analysis to find pieces of code which can be implemented more efficiently on programmable hardware (taking into account the reconfiguration overhead) and develop a toolchain that will automatically implement

that reconfiguration.

The GARP architecture proposed by Callahan et al. couples a MIPS processor with a reconfigurable array that functions as a compute co-processor [6]. This paper first identifies key concerns with using reconfigurable hardware in general: long reconfiguration times and low data bandwidths. Though the capabilities of FPGAs have vastly improved over the years since GARP was introduced, similar issues still apply today, given how FPGA-based architectures are less effective when executing tasks with low compute-to-memory-bandwidth ratios. Callahan et al. attempt to work around the limitations of limited data bandwidth by implementing a 2D *GARP* array of Configurable Logic Blocks (CLBs) that are interconnected with programmable wiring. The programmable wires allow the system to use the same data paths for multiple purposes, for loading configuration data and also for synchronizing register values between the host and GARP array when the array is idle. The same data paths are then used for memory accessing when the GARP array is active. The actual GARP architecture was simulated using a cycle-accurate simulator and results indicate that the GARP model was able to out-perform the most powerful CPUs at that time.

The Chimaera system by Ye et al. is similar to GARP in that it also includes a small, reconfigurable functional unit (RFU) tightly coupled with a general-purpose processor [7]. Like GARP, this system includes a Chimaera C compiler that identifies sequences of source codes that can be mapped to a single RFU operation. The paper introduces compiler optimizations for such mapping, including instruction combination, control localization and SIMD-like parallelized execution of sub-word operations. Furthermore, the RFU contains a separate register file, an execution control unit and a configuration control and caching unit that allow the actual functional unit (the reconfigurable array) to hold multiple RFU configurations at a time, reducing the burden of having to load configurations from off-chip frequently. It is also notable that the RFU architecture is more complex than GARP's and is much closer to accelerator designs we have nowadays with dedicated control and memory management units. The authors report results from timing experiments that the Chimaera architecture returns an average of 21% performance improvements under their most pessimistic latency model.

2.2 Dynamic Code Execution in Heterogeneous Architectures

Here, we discuss previously proposed energy-efficient heterogeneous architectures that demonstrate frameworks for making decisions about whether to execute a given line of code in hardware or in

software.

In 2010, Venkatesh et al. suggested an architecture pairing general-purpose host processors with specialized conservation cores (c-cores) that focus on energy efficiency rather than performance improvements [2]. Their ASIC implementation demonstrates a system that consists of multiple tiles, each with a unique set of c-cores as shown in Figure 1; further proposed is an automatic synthesis toolchain that will generate such cores from selecting "hot" regions in the target application's code. An interesting aspect of this work is the idea of generating energy conserving cores instead of performance-improving accelerators. Using c-cores allows the architecture to target a wider range of workloads that may not have explicit parallel structures in the code but can still benefit from improving the efficiencies related to dark silicon. More importantly, this work also presents a generally-applicable framework for heterogeneous SoCs that involves: (1) exposing the availability of c-cores to the compiler so that specific instructions will make use of the special-purpose hardware; and (2) utilizing a *fall-back* system such that the general-purpose processors (GPPs) will execute the code if no suitable c-core is available. The authors further address that they incorporate "patchability" into the c-core system with hopes of compensating for the fact their ASIC implementations cannot be updated post-fabrication in case the target application had undergone version updates. Although the additional hardware logic and control required for patching incurs a 2× overhead in area and power consumption, simulation results report that the c-core enabled architecture can return up to a 50% reduction in energy consumption.

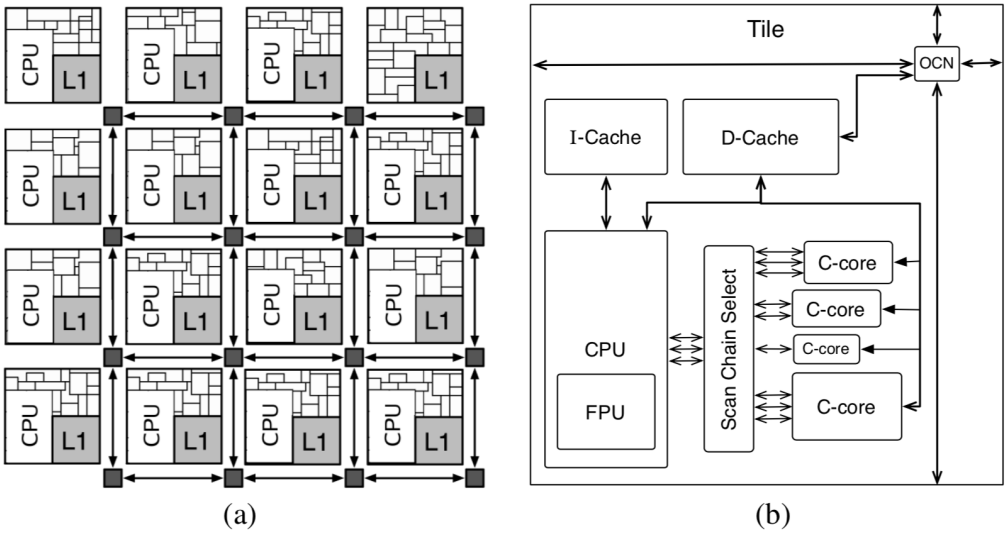

Figure 2.1: High-level diagram of the *c-core* enabled system. (a) The system consists of multiple tiles. (b) Each tile consists of a general-purpose core and a set of *c-cores*. Figure from Venkatesh et al. [2].

The same team from UCSD applied the c-core model to the Android platform. Swanson and Taylor explain that their motivation for developing GreenDroid arises from the fact that the Android mobile phone software stack spends 95 percent of its time executing just 43,000 static instructions; since c-cores are best applied to workloads that repeatedly execute the same functions, Android applications are great targets for the c-core architecture [8]. The "cold" code that the toolchain does not generate c-cores for are instead executed by the host CPU. The experimental results based on a 45nm research prototype, containing about 100 c-cores, report up to 11× energy-savings while maintaining similar, if not improved, levels of performance.

Quasi-specific Cores (QsCores) are generalized versions of c-cores that are capable of executing a set of general-purpose computations instead of a single, specific piece of code [9]. QsCores offer a major improvement in using ASICs as co-processors, since a relatively small number of these cores are able to support a potentially large fraction of the target application's code. In other words, in a QsCore-enabled system, a significant portion of the code can be executed via specialized cores without having to execute them on the less-efficient host CPU. To auto-generate such cores, Venkatesh et al. augment their synthesis toolchain from [2], so that the compiling system profiles hotspots in the application code and further identifies similar code patterns across the hotspots. The system also explores the trade-offs between computational powers of individual QsCores and the area required for building each core. As a result, what is generated is an optimal set of QsCores for a given input workload and on-chip area constraints. The QsCore-enabled system requires 50% fewer specialized cores than a comparable c-core (fully-specialized) system and also provides up to 25 × energy-efficiency compared to general-purpose processors over a diverse set of workloads.

2.3 RISC-V Accelerator-Based SoCs

The RISC-V Instruction Set Architecture (ISA) is an open-source ISA [10] that is increasingly becoming adopted by both academic researchers and industry manufacturers. A notable implementation of this ISA is Berkeley's Rocket Chip SoC Generator—a number of recent publications have based their architectures on either the scalar, in-order Rocket core or the speculative Berkeley Out-of-Order Machine (BOOM)[11]. In this section, we describe previous research contributions that share similar design patterns of using the Rocket Chip generator platform and attaching special-purpose hardware units as co-processors.

In 2016, Lee et al. developed the Hwacha vector-processor to work alongside the scalar Rocket core [12]. The proposed vector-processor optimizes its efficiency through packing vector arithmetic

operations into separate blocks of instructions; the vector issue unit then fetches and decodes vector instructions, and breaks them down into smaller operations to be executed in parallel by a systolic pipeline. The scalar host processor and the vector accelerator have independent instruction caches but share a 32KB data cache. It is further notable that Hwacha is integrated into the host's demand-paged virtual memory environment, and at the same time, also achieves a peak energy-efficiency of 16.7 double-precision GFLOPS/W.

Mao provides an example of attaching a hardware accelerator to the Rocket core platform via the Rocket Custom Co-processor (RoCC) interface [13] as shown in Figure 2.2. More specifically, this work explores the design of a memory to memory copy accelerator that is—like the Hwacha vector-processor—a Direct Memory Access (DMA) engine but is also aware of the virtual memory of the host. Mao's accelerator is effective because it has access to the host's page table walker, can perform its own address translation with its translation lookaside buffer (there is hence no need for page-pinning), and can also communicate with the host CPU in cases of page faults. Specifying such complicated control and communication between the host and the accelerator is made easy through the RoCC interface. Furthermore, using the RoCC interface allows users to easily invoke the hardware accelerator through custom memcpy() instructions that extend the RISC-V ISA.

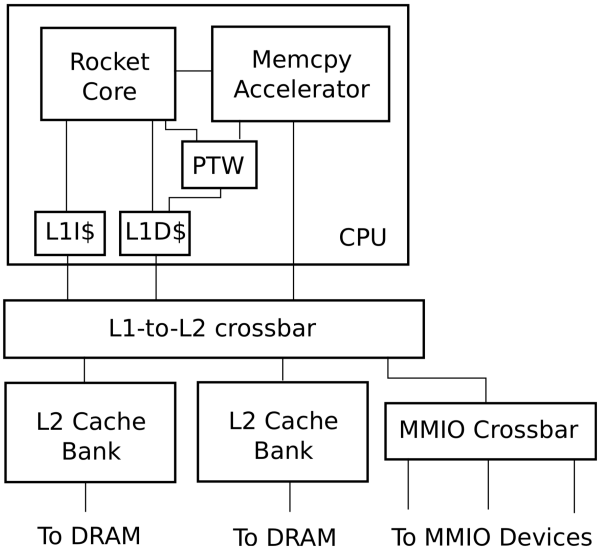

Figure 2.2: High-level diagram of the RISC-V Rocket architecture with a memcpy accelerator as a RoCC co-processor. Figure from Mao [13].

Koenig et al. also develops a RoCC accelerator for exact dot product computations [14]. In

this design, the authors ultimately find that connecting the accelerator to the host's L2 data cache with a 128-bit interface (compared to interfacing with the 64-bit data path to the L1 cache) allows significantly better performance, especially when the input vectors are large enough to exceed the size allowed in the L1 cache. Results compare the cycles-per-element (CPE) of their accelerator against comparable solutions using software libraries, and the authors find that the accelerator fully saturates the data cache bandwidth and achieves CPEs comparable to the Intel MKL, which is one of the fastest software implementations of dot products built with kernels leveraging x86 Advanced Vector Extension (AVX) instructions.

With the Celerity 511-core architecture, Davidson et al. showcase that it is possible to build an extremely high-performant architecture based on the RISC-V Rocket ecosystem [15]. As shown in Figure 2.3, the Celerity SoC consists of three architectural tiers: (1) the general-purpose tier that has five Linux-capable Rocket cores; (2) the specialization tier built with a Binarized Neural Network (BNN) accelerator; and (3) the massively-parallel tier that has a 496-core tiled, many-core array. Each of the tiles in the massively-parallel tier contains a low-power, 5-stage, in-order RISC-V Vanilla-5 core and a simple router that connects the tile to the mesh interconnection network. An interesting aspect of this architecture is the implementation of a mesh Network-on-a-Chip (NoC) of the many-core array: the authors propose an extension to the load-reserved, store-conditional (LR-SC) atomic instructions that they call LR load-on-broken-reserve (LR-LBR), which puts the core pipeline into a low-power state until the next remote store occurs. This custom atomic instruction enables a tight producer-consumer synchronization in the many-core network, allowing Celerity to more efficiently utilize the massively-parallel tier. Also described is the use of High-level Synthesis (HLS) to generate BNN accelerators from initial C++ implementations. Such example demonstrates how recent improvements in HLS tools enable architects to produce specialized accelerator architectures without the design burden of having to detail the hardware at the circuit level, using a hardware description language (HDL). Like the memory copy unit and dot product accelerator mentioned above, Celerity interfaces its accelerators with the general-purpose tier using the RoCC interface. Combining all three tiers of hardware, the Celerity architecture is reported to improve performance-per-watt by more than 100× compared to a mobile GPU.

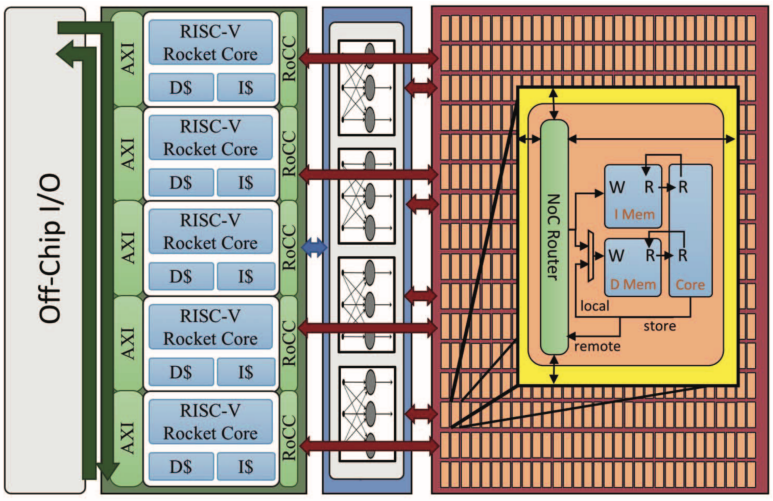

Figure 2.3: High-level diagram of the Celerity architecture. The section in green refers to the general-purpose tier; section in blue refers to the specialization tier; and section in red refers to the massively-parallel. tier. Figure from Davidson et al. [15].

2.4 Power-Adaptive Architectures

Historically, Dynamic Voltage and Frequency Scaling (DVFS) has been a popular power management mechanism incorporated into a wide variety of power-aware architectures. Here, we discuss relatively recent research on systems that use PMUs to adaptively adjust their behavior beyond the scope of traditional DVFS.

Eldridge et al. from IBM Research designed VELOUR, a RISC-V heterogeneous system with a machine learning accelerator and a power and resiliency management unit (PRIME) [16] for low voltage operations. As shown in Figure 2.4, the VELOUR architecture includes Critical Path Monitor (CPM) sensors that feed PRIME with real-time data on on-chip power demands; also included are performance counters that detect power proxy events. PRIME further includes state machines that monitor other metrics of power usage: a power-history-based voltage droop predictor (VDP) and a power management unit (PMU), with which PRIME predicts if at any point in time the host Rocket core or one its accelerators will perform incorrect operations. In this case, PRIME takes action to throttle the core for a pre-computed length of duty cycle that it is based on statistics collected by the VDP and PMU. The authors further demonstrate a framework for testing system behavior under low voltage situations—CHIFFRE is a Chisel / FIRRTL based fault injection tool that allows pre-ASIC experimentation. Another notable aspect of this work is that the entire architecture was built in the RISC-V open-source environment and that both PRIME

and CHIFFRE were implemented using the RoCC interface.

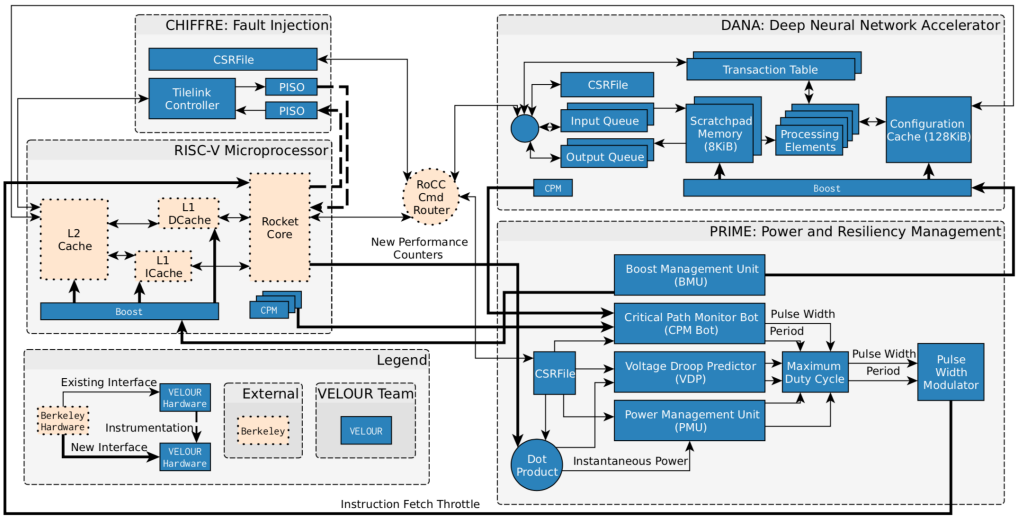

Figure 2.4: High-level diagram of the VELOUR architecture. The systems consists of a RISC-V Rocket core, PRIME, CHIFFRE and a neural network hardware accelerator. Figure from Eldridge et al. [16].

s

Keller et al. from UC Berkeley developed a RISC-V based architecture with an integrated and programmable PMU that allows the implementation of a variety of power management algorithms [17]. The authors designed such a PMU with a 32-bit 3-stage in-order processor (Z-scale) with a 8KB scratchpad to which the power-management programs were written; the minimal design of the 3-stage processor allows sufficient compute capabilities for processing relatively complex algorithms while incurring small costs to on-chip area (0.06 mm^2 when implemented in 28 nm FD-SOI). Furthermore, the switched-capacitor DCDC (SC-DCDC) system used for adaptive clock generation also provides toggle signals that can be compared against a fixed reference frequency to provide a simple, non-invasive measurement of core-power usage. PMU cores access these power measurements and use them to coordinate fine-grained voltage scaling (AVS) without the involvement of the host processor. As a result, the PMU can adaptively switch the core voltage by detecting changes in the workload within a 1μs time frame and reduce energy consumption by almost 40%.

2.5 Summary

Optimizing energy-efficiency through the use of specialized hardware has become a standard design practice in this new era of computer architecture research. Previous architectures that couple

reconfigurable units with general-purpose cores [6, 18] clearly demonstrate that a system with dynamic customization capabilities can significantly reduce power consumptios. Using reconfigurable units, if not more general implementations of specialized units, allows systems to deliver enhanced efficiencies, not only over a wider range of workloads, but also in a more scalable manner since they help reduce the area required to implement the application-specific units. Research on c-core and Qs-Core systems [2, 8, 9] also suggests that a framework of executing code in specialized hardware whenever possible—and falling back to executing on the host CPU when hardware is unavailable—is a reasonable approach for such heterogeneous SoC designs. In particular, the RISC-V ecosystem has already produced a volume of research on special-purpose hardware that is open-source and available to anyone [12–14, 16], and the number of such contributions are expected to continuously grow in the near future. Previous works on power-adaptive designs further hint at the opportunities of achieving even better power-efficiencies with the help of integrated, intelligent and self-aware Power Management Units [16, 17]. Overall, previous literature suggests that a reconfigurable SoC that can dynamically configure a range of special-purpose hardware according to on-chip power demands will be able to attain notable improvements in energy-savings; perhaps the best platform for implementing this architecture would be the RISC-V ecosystem which provides a wealth of open-source hardware development tools, along with a range of readily-available accelerator designs.

3. Hardware Implementation Workflow

The Field Programmable Gate Array (FPGA) is the most suitable computing platform for implementing our dynamic, self-optimizing architecture. FPGAs not only offer the reconfigurablilty needed for run-time customization of hardware but also are becoming increasingly available, thanks to the improvements in manufacturing technologies. Today's commercial-grade FPGAs feature impressive logic cell capacities, large enough to implement digital systems as complex as Linux-bootable processors. Shown in Figure 3.1 is one of such FPGA product, the Xilinx Artix-7 Nexys-A7 board, which we chose to use for this project.

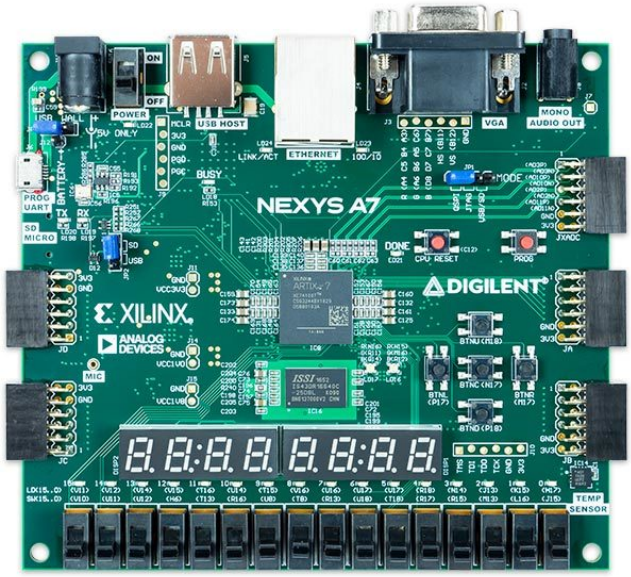

Figure 3.1: Xilinx's Nexys-A7 board.

Concerns with using FPGAs, however, arise from the fact that most commercial FPGA architectures are both complex and proprietary and that programming them requires the use of

vendor-specific electronic design automation (EDA) tools. For example, to use Xilinx products there is no other choice but to study and utilize Xilinx's Vivado Design Suite software. Such challenges of working with FPGAs are in fact one of the biggest hindrances to their wide-spread use, especially among application developers who could otherwise immensely benefit from accelerating software through specialized hardware on FPGAs. In this chapter, therefore, we first sketch further details of a typical FPGA architecture to better understand its reconfigurability; then, we walk through the process of creating and implementing a high-level digital design on Artix-7 FPGAs, based on concrete examples.

3.1 Field Programmable Gate Array Architecture

Here, we provide an introduction to the internal architecture of an FPGA. Historically originating from Programmable Logic Devices (PLDs), FPGAs have over the years evolved into a powerful reconfigurable fabric consisting of two main components: configurable logic blocks (CLBs) and programmable interconnects. The user is able to configure (and reconfigure) the fabric by specifying both the function of each block and the interconnection between the logic blocks. A typical FPGA architecture is designed as a 2D array of logic blocks. As shown in Figure 3.2, logic blocks in such mesh-styled structure often resemble remotely placed "islands" and are thus commonly referred to as logic islands.

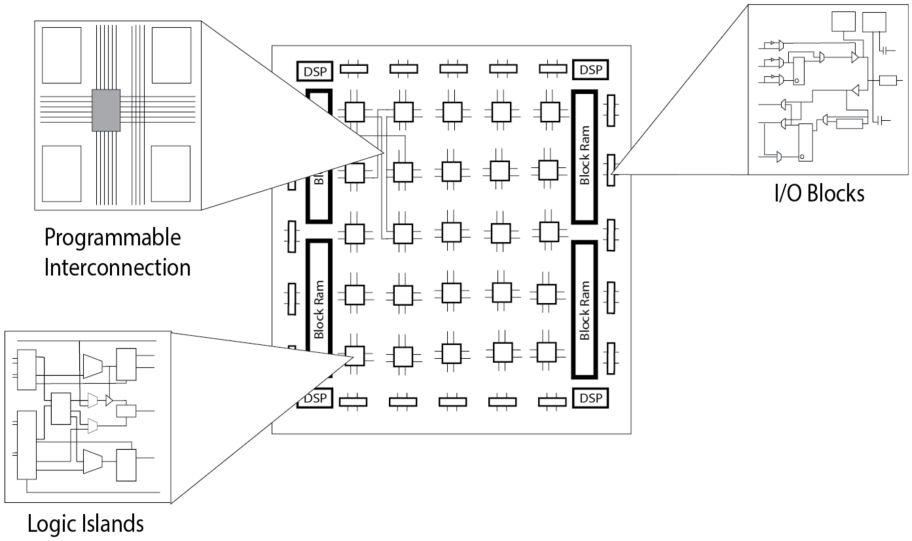

Figure 3.2: Internal architecture of a typical island-style/mesh-style FPGA. Logic islands equivalently refer to CLBs.

In FPGAs, combinational logic is implemented through logic blocks, which are in turn constructed with multiple instances of look-up-tables (LUTs). In contrast to ASICs or other traditional CMOS Integrated Circuits (ICs) where combinational logic is implemented through hard-wiring physical logic gates, FPGAs simulate the same behavior through LUTs that can implement arbitrary combinational functions. An n-bit LUT is in other words a *general-purpose logic gate* that can implement any deterministic function of n inputs. Besides LUTs, a typical logic block will also include FFs to optionally store the outputs of the logic function and thus support a pipelined circuit design. CLBs further include latches that are set by the configuration bit-stream given by the user. Bits stored in those latches support the reconfigurability of the logic blocks and hence the overall FPGA. Specific details on how CLBs are organized vary greatly depending on the FPGA vendor and even among different products from the same vendor. As an example, in the Xilinx Artix-7 FPGAs we choose to use, individual CLBs consist of elementary logic units called *logic slices* that contain four 6-input LUTs and eight FFs.

The other key component of an FPGA is the programmable interconnect that sets the communication channels between arbitrary CLBs. Since the FPGA itself can be thought of as a two-dimensional grid of logic blocks, the routing architecture consists of bundles of wires that run across the chip vertically and horizontally. Placement of the logic units and routing are mostly determined at the hardware compile-time when the user generates the configuration bit-stream; the goal of the EDA software is to determine the the placement of logic blocks that will minimize wiring. Most FPGA vendors such as Xilinx and Intel/Altera, again use latches to store the configuration of the interconnects. These latches are also configured by the user-defined bit-stream.

Hence, FPGAs are largely a homogenous grid built with logic blocks and interconnects: thanks to this regular structure, manufacturing FPGAs are less complex than fabricating a modern CPU of the same size. As a result, the logic capacities have grown explosively over the past decades and the FPGAs today are powerful enough to implement a wide range of digital circuits. However, to further improve performance and usability, most FPGA vendors also include other IC components in their products, such as block RAMs (BRAMs), digital signal processing (DSP) slices, I/O controllers, Peripheral Component Interconnect express (PCIe) and more. BRAMs store larger amounts data than what logic blocks can support and thus help the FPGA to handle data-intense computations; DSP slices have dedicated multipliers and adders to efficiently handle signal processing loads; many commercial FPGAs also include various types of high performance I/O components to optimize data transport to and from other on-chip devices, such as CPUs, GPUs, add-on SSDs and ethernet ports.

3.2 RISC-V Hardware Design Workflow

Having described the basics of FPGAs, we move on to discussing the workflow of implementing custom digital logic on an Artix-7 FPGA. A typical implementation workflow is illustrated in Figure 3.3.

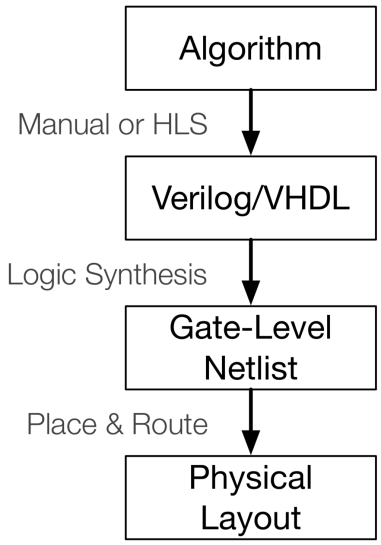

Figure 3.3: A typical FPGA synthesis flow. Figure from [19].

In most cases, one begins with a high-level concept or function in mind—in the context of building accelerators, this would be the target algorithm the user wishes to accelerate using hardware. The next step is to describe the corresponding circuit at the Register Transfer Level (RTL) using a hardware description language (HDL). The two most commonly used HDLs are Verilog and Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL), both of which date back to the 1980s. Unlike software programming languages like C that are compiled into assembly language instructions for the CPU to execute at run-time, these HDLs are compiled and synthesized to a definition file describing the hardware itself. Finally, logic synthesis and place-and-route tools, often part of commercial EDA software, are used to take the input RTL to produce a gate-level netlist and then either a physical circuit layout or a FPGA configuration bitstream.

The first immediate challenge is to effectively produce a RTL implementation of the desired algorithm. Manual implementation in Verilog or VHDL is highly unreliable and inefficient since the designer must directly and entirely describe the hardware with low-level circuit blocks such as adders, multiplexers and registers. The design process is painfully slow even for experienced

hardware developers, let alone application programmers with little to no knowledge about circuit-level designs. Therefore, many research and commercial efforts have produced high-level synthesis (HLS) tools that will auto-generate low-level RTL code from high-level algorithm descriptions.

Among the many available HLS tools, we choose to use Chisel, a parameterizable hardware design language that is embedded in Scala [20]. There are two important reasons behind our choice: (1) Chisel is a highly efficient language that enables agile development of complex circuit generators and offers a wealth of sample code as templates for custom designs, and (2) Chisel is part of an ecosystem that includes many helpful resources for hardware development, most notably the RISC-V Instruction Set Architecture (ISA) [10] and the Rocket Chip SoC Generator [11]. Since we are in need of an open-source processor for our own architecture, we decided to use the in-order, scalar Rocket core as our host CPU, and given that Rocket itself is written in Chisel, we implemented the rest of our design in the same language. In the remainder of this section, we describe the process of specifying a circuit for a target application in Chisel, using Chisel to generate hardware-synthesizable Verilog code, and finally testing the Verilog output via simulation tools.

3.2.1 Target Application: Bresenham's Circle Drawing Algorithm

Bresenham's circle drawing algorithm is a low-cost method to accurately represent a continuous circle on a pixelated screen, without the use of any floating-point operations. It is an example of a computationally intensive task with a relatively straight-forward control flow, making it an ideal candidate for constructing an example hardware accelerator. Based on the symmetry of a circle centered at the origin, we only need to compute the pixel coordinates on the first octant as described in Algorithm 1.

Algorithm 1 Bresenham's Circle Drawing Algorithm

Require: Radius $r > 0$

$e \leftarrow -r, \quad x \leftarrow r, \quad y \leftarrow 0$

$a[] \leftarrow$ Array of computed coordinates

while $x \geq y$ **do**

$a[y] \leftarrow x$

$e \leftarrow e + (2y + 1), \quad y \leftarrow y + 1$

if $e \geq 0$ **then**

$e \leftarrow e - (1 - 2x), \quad x \leftarrow x - 1$

end if

end while

3.2.2 Describing the Circuit in Chisel

To build a hardware unit for the circle drawing algorithm, we first need to specify the circuit in Chisel. Here, we list implementations of the algorithm both in C and in Chisel to highlight the differences.

```c
int swCircle(int r, int a[]) // Inputs
{
  // Local Variables
  int e = -r;
  int x = r;
  int y = 0;

  // Computatation
  while (x >= y) {
    a[y] = x;
    e = e+2*y+1;
    y++;
    if (e >= 0) {
      e = e+1-2*x;
      x--;
    }
  }
  return y; // Output
}
```

Listing 3.1: C Implementation

```
class Circle extends Module {
  val io = IO(new Bundle {
    // Inputs
    val r         = Input(UInt(8.W))
    val outputEnable = Input(Bool())
    // Outputs
    val x = Output(UInt(8.W))
    val y = Output(UInt(8.W))
  })

  // Local Variables
  val e  = RegInit(~io.r+1.U)
  val tempx  = RegInit(io.r)
  val tempy  = RegInit(0.U(8.W))
  val e2 = e + tempy+ tempy + 1.U
  val e3 = e + 2.U -tempx-tempx+tempy+tempy
  val done = tempx <= tempy || ~io.outputEnable || (tempx-1.U < tempy+1.U)

  // Computation
  tempy := Mux(~done, (tempy + 1.U),tempy)
  tempx := Mux(~done && e2(7)===0.U, (tempx -1.U), tempx)
  e := Mux(done, e, Mux(e2(7)===0.U, e3,e2))
  io.x := tempx
  io.y := tempy
}

// Generate hardware to later run computations
object CircleDriver extends App {
  chisel3.Driver.execute(args, () => new Circle)
}
```

Listing 3.2: Chisel Implementation

The greatest difference between the two versions arises from the fact that the C code is a direct software implementation of the *computation* itself, whereas the Chisel code is a specification of a *hardware artifact* that will later be invoked to perform the computations. While the C program is called every time we run the algorithm, the Chisel code is executed only once at compile-time–from then on, the same hardware unit is re-used with different input values, instead of having to make modifications.

Despite the differences, however, the C and Chisel code listed in Listing 3.1 and 3.2 exhibit remarkable similarities in their structures. Both implementations include code blocks that describe

the computations for the circle drawing algorithm, detail what the input and outputs are, and declare local variables to aid the computation. Such similarities demonstrate the clear advantages of using a HDL embedded in a software programming language like Scala. Chisel offers many of Scala's modern language features, including support for object-oriented and functional programming, code-reuse and hardware description via composable, parameterizable modules. For example, `Mux` is a higher-level abstraction of multiplexers in Chisel which can be fully parameterized and used as if it were a software construct. Compiling Chisel, the multiplexer object is automatically transformed down into register and wire-level units that Verilog is capable of understanding.

3.2.3 Compiling into Verilog

Based on our Chisel code in Listing 3.2, we can generate a Verilog output using the Flexible Intermediate Representation for RTL (FIRRTL) compiler that is built into the Chisel framework[21]. Chisel itself contains a front-end compiler that generates the specified circuit into intermediate data structures, which are in turn simplified, optimized and verified by the back-end FIRRTL framework. Through this chain of circuit-level transformations, FIRRTL emits the final circuit design in Verilog. The combination of Chisel and FIRRTL greatly eases the workflow of digital hardware design as the generation of RTL code from a Scala-embedded language is entirely automated and sufficiently optimized. Listing 3.3 gives a snippet of the Verilog code that is produced from the Chisel implementation of the circle drawing algorithm we saw earlier.

```
1  module Circle( // @[:@3.2]
2    // Inputs
3    input         clock, // @[:@4.4]
4    input         reset, // @[:@5.4]
5    input  [7:0] io_r, // @[:@6.4]
6    input         io_outputEnable, // @[:@6.4]
7    // Outputs
8    output [7:0] io_x, // @[:@6.4]
9    output [7:0] io_y // @[:@6.4]
10 );
11   // Local Variables
12   wire [7:0] _T_13; // @[circle.scala 18:20:@8.4]
13   wire [8:0] _T_15; // @[circle.scala 18:25:@9.4]
14   wire [7:0] _T_16; // @[circle.scala 18:25:@10.4]
15   reg [7:0] e; // @[circle.scala 18:19:@11.4]
16   reg [31:0] _RAND_0;
17   reg [7:0] tempx; // @[circle.scala 19:23:@12.4]
```

```
18    reg [31:0] _RAND_1;
19    reg [7:0] tempy; // @[circle.scala 20:23:@13.4]
20    reg [31:0] _RAND_2;
21     ...
22    assign _T_13 = ~ io_r; // @[circle.scala 18:20:@8.4]
23    assign _T_15 = _T_13 + 8'h1; // @[circle.scala 18:25:@9.4]
24    assign _T_16 = _T_13 + 8'h1; // @[circle.scala 18:25:@10.4]
25    assign _T_21 = e + tempy; // @[circle.scala 21:14:@14.4]
26    assign _T_22 = e + tempy; // @[circle.scala 21:14:@15.4]
27     ...
28    // Computation
29    always @(posedge clock) begin
30      if (reset) begin
31        e <= _T_16;
32      end else begin
33        if (!(done)) begin
34          if (_T_58) begin
35            e <= e3;
36          end else begin
37            e <= e2;
38          end
39        end
40      end
41      if (reset) begin
42        tempx <= io_r;
43      end else begin
44        if (_T_59) begin
45          tempx <= _T_45;
46        end
47      end
48      if (reset) begin
49        tempy <= 8'h0;
50      end else begin
51        if (_T_50) begin
52          tempy <= _T_48;
53        end
54      end
55    end
56  endmodule
```

Listing 3.3: Parts of the Verilog output generated by Chisel/FIRRTL

3.2.4 Simulating the Design

The final step of the hardware design workflow is to test and verify the circuit. Testing the logical fidelity of a hardware design is in many ways more complex than testing software since the testbench must accurately simulate the entire hardware environment attached to the circuit at hand. Fortunately, the RISC-V ecosystem offers several highly effective tools for pre-silicon hardware simulation and perhaps the most powerful is FireSim, an open-source FPGA-accelerated cycle-accurate RTL simulator [22]. FireSim goes beyond the scope of many software RTL simulation tools, by offering standard I/O models for DRAM, disk and Ethernet network connections and allowing the user to simulate not only one but also clusters of SoCs by harnessing multiple FPGA instances available through cloud computing services. More recently, FireSim has further integrated a profiling and modeling framework that optimizes the hardware-software stack as a whole, taking into account the operating system, application software, RTL implementations and network links simultaneously [23]. Hardware developers interested in high-performance RTL simulation should refer to these tools.

However, in our case, the RTL design is relatively small in scale. There is no need for a cloud-based FPGA-accelerated platform for simulation. Instead, we opt to use another software RTL simulation tool integrated into the RISC-V system stack called *Verilator*. This tool produces a cycle-accurate behavioral model in C++ based on the input Verilog file and offers more reliable performance than other event-driven RTL simulators. With Verilator, users can simply run assembly instruction tests or benchmarks to verify and estimate the performance of the given circuit. Test scripts can be written in C and their binaries are compiled using the RISC-V cross-compiler to be then run by Verilator against its behavioral model. An example is shown in Listing 3.4 of a snippet of the test script written for a Rocket-attached accelerator for the circle drawing algorithm.The software implementation of the algorithm and the hardware-invoking function calls can be written side-by-side and run consecutively in the same binary to test any differences in the computation results.

```
1 static inline void hwCircle_init(int r)
2 {
3   ROCC_INSTRUCTION_S(0, r, 0);
4 }
5
6 static inline unsigned long hwCircle_iter()
7 {
8   unsigned long value;
```

```
  ROCC_INSTRUCTION_D(0, value, 1);
  return value;
}

int hwCircle(int radius, int x[])
{
  int y = 0;
  int xVal;
  hwCircle_init(radius);
  x[y++] = radius;
  while (xVal = hwCircle_iter()) {
    x[y++] = xVal;
  }
  return y;
}
```

Listing 3.4: Parts of the Verilator test script

3.3 FPGA Implementation Workflow

Having completed and verified the RTL circuit, we finally run the Vivado Design Suite to implement the design onto a physical FPGA board. Once we input the Chisel-generated Verilog file, Vivado creates a new *intellectual property* (IP) block, which is the term used by Xilinx for individual hardware blocks to be placed on the FPGA. To implement the full-scale SOAR SoC, we must input all generated Verilog files, not only for our circle drawing accelerator but also for the host Rocket core and its associated peripherals. Vivado attaches the accelerator IP as an add-on to the Rocket IP, and the full design is merged together before it is finally embedded onto a physical FPGA.

Based on the layout specified in Verilog, the EDA software automatically verifies the wiring and clock synchronization across all the elements in the circuit. Once Vivado has verified all interconnects, the circuit can be synthesized and implemented. Hardware synthesis usually takes a significantly greater amount of time compared to an equivalent software compilation: in the case of synthesizing a full-scale SoC with a host CPU, the process can take up to 30 minutes. This implies that any updates to the hardware design post-synthesis are extremely time consuming, and it is important to step-by-step verify the circuit using RTL simulations. As part of the synthesis and implementation process, Vivado outputs several reports on timing, power consumption and area utilization estimates. Figure 3.4 shows an implemented circuit diagram of the Rocket host core containing a circle drawing algorithm accelerator as a co-processor on an Artix-A7 (XC7A100T-

1CSG324C) FPGA; Vivado reports that the Rocket core system occupies 75% of the available LUTs, 25% of the FFs and 61% of the BRAMs, while the accelerator utilizes 1.6% of the available LUTs, 0.4% of the FFs and 0.5% of the BRAMs.

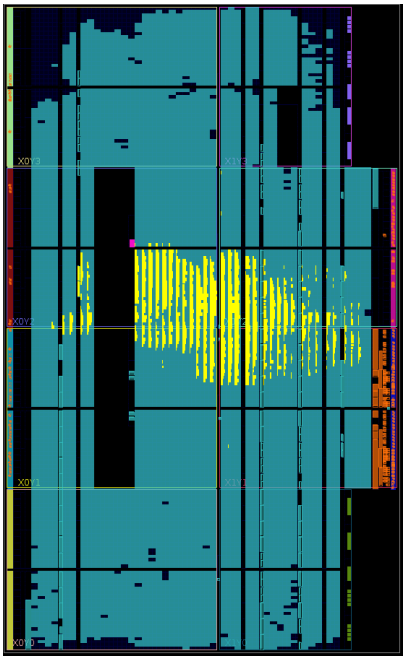

Figure 3.4: Implemented circuit diagram for the Rocket core; the area marked in yellow indicates the logic units that are being used for Rocket.

3.4 Summary

Through the example of Bresenham's circle drawing algorithm, we see that the process of programing a high-level algorithm on an FPGA is a fairly involved process. The complexity of this process often discourages researchers from exploring the potentials of accelerator-rich SoCs that can significantly improve system performance and energy-efficiency in various application domains. However, we also witness that the advent of highly efficient open-source tools, both for HLS and hardware simulation, is bringing about significant changes to the hardware development landscape. We predict that these tools will attract more developers to design and implement their own sets of accelerators and that open-source accelerator designs will continue to become increasingly available to everyone.

4. SOAR Architecture

4.1 Concept and Goals

Research discussed in Chapter 2 demonstrates that accelerator-rich SoCs can offer significant improvements in energy-efficiency and/or performance compared to GPPs. Indeed, there are clear advantages to utilizing domain-specific hardware units, and in this work, we consider a heterogeneous SoC that attaches several accelerators to a general-purpose CPU. However, there are several major drawbacks in current implementations of such systems.

- Accelerator-based architectures have a *limited range of functions* they can improve. For example, an SoC with a neural net accelerator will be tailored towards high performance machine learning inference, but will exhibit equal or worse performance when executing any other workloads. Due to the very nature of domain-specific hardware, there exists a strict restriction on the generality of tasks that each accelerator can perform. Nonetheless, there still exists a demand for improved performance over general-purpose workloads, and the question is how to optimize performance over a wide range of tasks with accelerators.

- Most architectures so far have employed a *static policy* for using accelerators. Mainly, the system identifies instructions or tasks that can be run using hardware and will simply execute the function in hardware whenever possible. However, we anticipate situations where the system should be optimized towards power efficiency, whereas in other cases the system should focus on performance and throughput. Therefore, the hardware may need to adjust its setting for using accelerators, depending on the user-defined optimization goals and specific workloads the system is executing at the moment.

- The process of using accelerators has been *complex*, often requiring the user to understand the underlying hardware architecture. And because most systems have used widely varying CPU-accelerator interfaces, it is difficult to imagine a single architecture with a freely in-

terchangable, customizable list of accelerators. Instead, it would be advantageous to have a system that will automatically control and streamline the process of invoking individual accelerators, such that users can simply run the same software that they would normally run on a GPP and still receive performance gains and/or energy savings.

In response to the challenges listed above, we propose an architecture that is *reconfigurable, adaptive* and *easy-to-use*.

- We propose building our system on reconfigurable fabric, which will allow our architecture to dynamically customize its list of physically available accelerators. We devise a system that is capable of swapping out a less-utilized accelerator for a unit that is in greater demand, based on an internal record of which accelerator utilization statistics. Such a scheme will allow the hardware to support improvements in performance and efficiency over a wide range of tasks, as long as there exists an accelerator design for each potential workload.
- We propose a framework to power-adaptively utilize accelerators. Our architecture will monitor the system's dynamic power usage and accelerator utilization rates. Based on a simple heuristic function, the system will determine at each moment which accelerators should be invoked (instead of executing corresponding software functions) and which accelerators should be configured on-chip, with the goal of optimizing overall efficiency.
- We propose an architecture that is fully-automated at the hardware and system software levels, such that optimal use of accelerators can be achieved without user intervention. Furthermore, we devise a modular framework for attaching accelerators to our system, so that any accelerator design that fits our basic requirements can be easily *plugged-in and run*.

Our proposed architecture includes both the underlying hardware and a software infrastructure. The remainder of this chapter specifies the details of our hardware implementation on FPGAs; the following chapter discusses the SOAR software, how it monitors the hardware performance and executes self-aware decisions about the hardware control and reconfiguration.

4.2 SoC Design and Implementation

The overall hardware system architecture is shown in Figure 4.1. The SoC includes several components including the open-source Rocket core, a set of homegrown RoCC accelerators, L1 and L2 caches, and finally, memory and I/O connection peripherals specified by the Rocket Chip generator [11] and the lowRISC SoC extension [24].

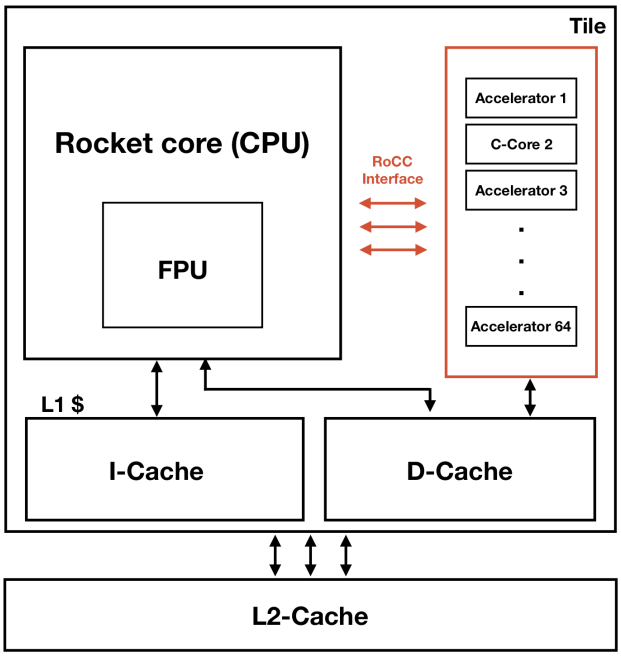

Figure 4.1: Diagram of the SOAR SoC architecture

4.2.1 Rocket Core and the RoCC Accelerator Interface

The *Rocket* core is a 5-stage, in-order scalar processor developed at UC Berkeley that has been released as open-source. The core contains a memory management unit that supports page-based virtual memory with a translation lookaside buffer (TLB) and page table walker (PTW), a non-blocking data cache, branch prediction unit, and an optional floating point unit (FPU). Rocket is an implementation of the RISC-V ISA (RV64G) and supports machine, supervisor and user-privilege levels to run RISC-V Debian Linux. Rocket's pipeline diagram is shown in Figure 4.2.

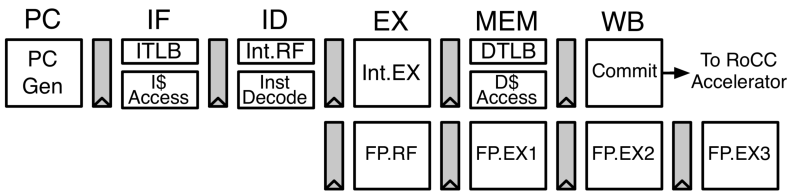

Figure 4.2: Pipeline Diagram for the Rocket core

Rocket inherently supports the addition of custom accelerators via the Rocket Custom Coprocessor (RoCC) interface. As shown in Figure 4.3, the default RoCC interface consists of three signal groups: command control (CC), register-mode and memory-mode signals. CC signals support coordination between the host and accelerators, such as through indicating whether an accelerator is currently busy (in the process of a memory request), whether a privileged process is running on the core, or if the core should be interrupted.

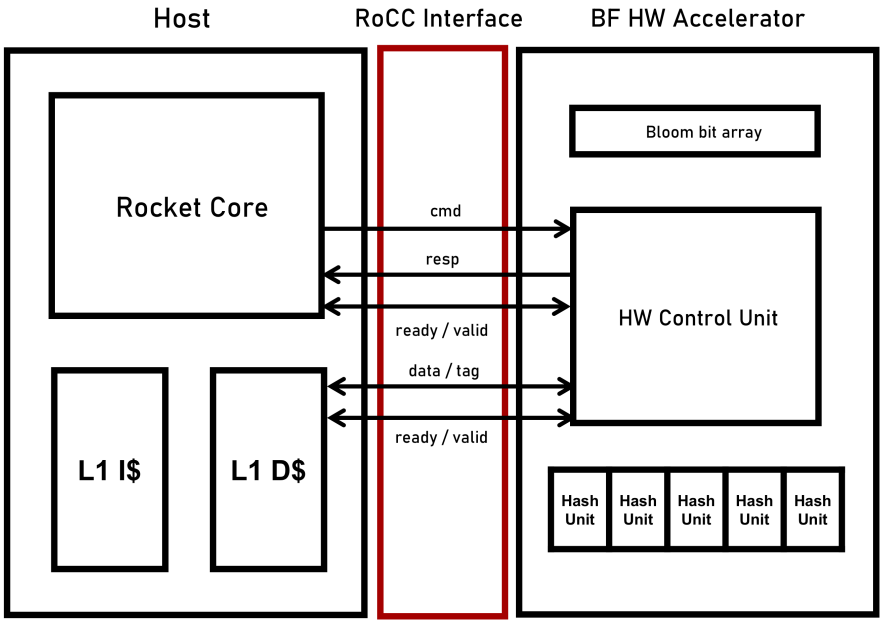

Figure 4.3: RoCC interface, example shown with Bloom filter accelerator

Register-mode signals can be subdivided into Command and Response signals. Command signals are used to invoke accelerators and are directly driven by custom RoCC instructions. The format of RoCC instructions are shown in Figure 4.4: the `function` bits specify which accelerator function is being called; `source 1` and `source 2` hold the source register IDs; instruction flags indicate whether source and/or destination registers are set to be used; `destination` is the destination register ID; and finally, the custom opcode is for differentiating between instructions for different accelerators, in case there are multiple accelerators present. Rocket decodes the instruction, parses each piece of information mentioned above and issues Command signals which further include source register data and read-valid signals, indicating whether the Command was valid and the accelerator is *ready* to receive a new Command. In turn, Response signals specify the response from the accelerator back to Rocket and contain the destination register ID and data, as well as another set of ready-valid signals for synchronization control. In the context of the Rocket pipeline, communication over Register-mode signals occurs at the very end of the pipeline as noted in Figure 4.2.

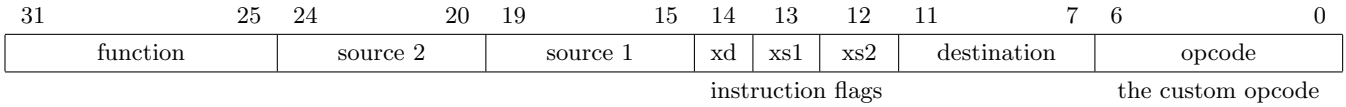

Figure 4.4: RoCC Custom Instruction

Memory-mode signals are used by the RoCC accelerator to access the shared L1 data cache. Memory request signals include the memory address, write data, and an opcode specifying whether it is a load or a store operation; response signals similarly consist of the memory address and the data response to the request. As in the case of Register-mode signals, Memory-mode signals ensure coordination between the cache and accelerator via ready-valid signals. In the extended RoCC interface, it is further possible to establish communications between accelerators and the L2 cache, PTW, Control Status Register (CSR) and FPU. Such extended interface may be desirable for complex, large-scale RoCC accelerator designs; however, for reasons discussed later in this Chapter, we opt to use the default RoCC interface in this work.

4.2.2 lowRISC extension of the Rocket Chip generator

The Rocket Chip generator is the overarching Chisel framework that produces RTL designs of Rocket-based platforms. Based in Chisel, the Rocket Chip generator utilizes a collection of parameterized chip-building libraries to construct custom SoCs that include, not only the Rocket core and the RoCC interface, but also supporting hardware sub-components including the L1 data and

instruction caches, L2 caches, memory interface networks and AMBA-compatible I/O connection peripherals. A block diagram of a system produced by this framework is shown in Figure 4.5.

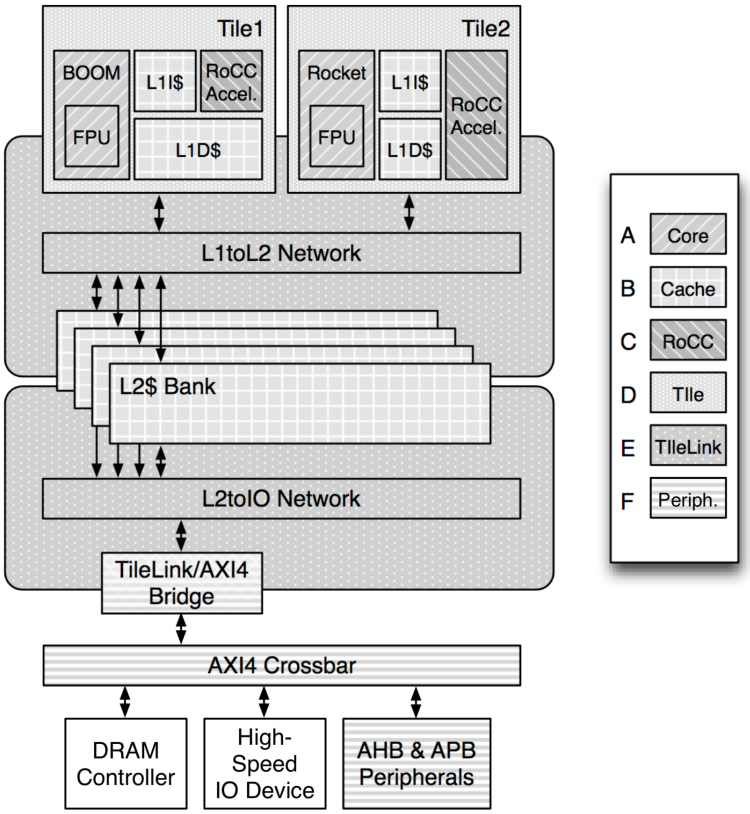

Figure 4.5: Diagram of the full Rocket SoC produced by the Rocket Chip generator. Figure from [11].

lowRISC further extends the Rocket Chip generator with open-source hardware peripherals for Ethernet connection, Universal Asynchronous Receiver/Transmitter (UART) connection, FPGA board-specific display and keyboard connections, and Linux boot-loading through a SD-card reader [24]. The implementation of the lowRISC system on a Nexys-A7 board is capable of booting RISC-V Linux kernels with a clock rate of 50 MHz. The SOAR architecture builds upon the lowRISC system by attaching a set of RoCC accelerators to the hardware and introducing an additional software system described in Chapter 5. Working with lowRISC presented several challenges, including that the up-to-date lowRISC SoC release is built on an outdated version of the Rocket chip. Nonetheless, basing our architecture on lowRISC has proven to be immensely helpful since we were able to implement the architecture on an physical FPGA, instead of as a software simulation instance.

4.2.3 SOAR Accelerators

This section describes a partial collection of homegrown RoCC accelerators we implemented for SOAR. Indeed, there exist several approaches to designing accelerators: previous research mostly demonstrates application-level accelerators, such as neural net processing units, since these large-scale accelerators can carry out involved computations and return significant performance gains; other works have devised toolchains that profile target software applications and auto-generate hardware units. In this work, we instead focus on building accelerators for fine-grained kernels and functions, such as hashing and string operations. There are four major reasons behind our design choice:

- Fine-grained accelerators are more appropriate for our adaptive, customizable architecture. It is more optimal to tailor the on-chip hardware with a large number of small units than with a few, large accelerators.
- The current implementation of the Rocket Chip generator does not support context-switching in RoCC accelerators. Explicit page pinning is required for large-scale accelerators, otherwise the hardware units may loose context during their command executions. Developing RoCC accelerators with such capabilities [13] is outside the scope of this work.
- The current implementation of RoCC does not allow Rocket and another co-processor to execute instructions simultaneously. That is, whenever a RoCC command is issued to invoke an accelerator, the host CPU itself is stalled until the RoCC interface returns response signals indicating the command has been completed. Building a more efficient, parallelized host-accelerator interface would be an interesting direction future research can pursue; more details are explained in Chapter 7.
- Small accelerators are easier to implement and verify. This approach follows RISC-V's overarching emphasis on the *Agile hardware development* methodology and allows a small team of developers to produce a number of effective accelerators over a short period of time. We further hope that releasing this work as open-source will motivate various parties to also partake in developing their own list of compatible accelerators; focusing on small units will make such design efforts more manageable and will allow future iterations of SOAR to benefit from having a wider pool of available accelerator designs.

The remainder of this section provides information about some of the functions we developed

RoCC accelerators for. Functions listed here are Bloom filter operations, combinatorial sequence generation, string copy and string comparison; not included are square-root computations, sorting and Secure Hashing Algorithms (SHA-2). For the sake of brevity, we only list the C implementations in this chapter; corresponding Chisel code can be found in Appendix B.

Bloom filter operations

Bloom filters (BFs) are space-efficient, probabilistic data structures that offer constant-time membership querying. Due to their efficiencies, Bloom filters have been widely adapted in various application domains, from database systems to networks and security. In this work, we specifically consider Bloom filters for string matching on large text files. We consider the two main operations, mapping (inserting) an element to the Bloom filter and testing (querying) a word had previously been mapped to the filter. As an example, Listing 4.1 shows the implementation of the `map` function in C.

```
/*
 * Map word to Bloom filter.
 * Places 1 in filter at indices that given word maps to.
 */
void mapToBloom(unsigned char *filter, char *word)
{
    long *hashes = (long *)calloc(K_NUM_HASH, sizeof(long));
    hash(hashes, word);

    // set the bits at the hash value indices to 1
    for (int i = 0; i < K_NUM_HASH; i++)
    {
        filter[hashes[i]] = 1;
    }
}

void hash(long *hashes, char *word)
{
    unsigned long x = hashstring(word);
    unsigned long y = x >> 4;

    for (int i = 0; i < K_NUM_HASH; i++)
    {
        x = (x + y) % M_NUM_BITS; // ith hash value
        y = (y + i) % M_NUM_BITS; // displacement
```

```
26          hashes[i] = x;
27
28  }
```

Listing 4.1: Bloom filter map operation implemented in C

Combinatorial sequence generation

Combinatorial sequence generation is a key operation for hardware-level randomization and cryptographic applications. Most notably, loop-less generation of bit patterns is an effective way to produce interesting combinatorial sequences. In this work, we implement accelerators to speed up the algorithms using successor rules, implemented by Knuth [25] and Stevens and Williams [26]. As an example, Listing 4.2 depicts the C implementation of Knuth's fixed-weight binary string generation algorithm.

```
1  int nextWeightedCombination(long n, unsigned long last, unsigned int *out) {
2      unsigned long next, temp;
3      next = last & (last + 1);
4      temp = next ^ (next - 1);
5
6      next = temp + 1;
7      temp = temp & last;
8
9      next = (next & last) - 1;
10     next = (next < 0x8000000000000000)? next : 0;
11     next = last + temp - next;
12
13     if(next / (1L << n) != 0) { return -1; }
14     *out = next % (1L << n);
15     return 1;
16 }
```

Listing 4.2: Knuth's fixed weight binary string generation implemented in C

String Copy

String copy is a fundamental operation that copies a character string from one memory address to another. Due to its such prevalent use, this function is an ideal candidate for one of our target accelerators for optimizing performance against general-purpose workloads. Listing 4.3. lists a

typical implementation of `strcpy` in C.

```
1
2 char * strcpy(char *strDest, const char *strSrc)
3 {
4     char *strTemp = strDest;
5     while(*strDest++ = *strSrc++);
6     return strTemp;
7 }
```

Listing 4.3: `strcpy` implemented in C

String Comparison

String comparison is another typical operation frequently used in various workloads. The function compares two character strings and returns the lexicographic difference. Listing 4.4. shows a typical C implementation of `strcmp`.

```
1  int swstrcmp(const char* s1, const char* s2)
2  {
3    unsigned char c1, c2;
4
5    do {
6      c1 = *s1++;
7      c2 = *s2++;
8    } while (c1 != 0 && c1 == c2);
9
10   return c1 - c2;
11 }
```

Listing 4.4: `strcmp` implemented in C

4.3 Summary

The SOAR hardware architecture consists of a single host Rocket core, a list of RoCC accelerators and various peripherals supporting the memory hierarchy and I/O connections. The RoCC interface provides an effective and manageable way to design custom hardware units that is compatible with

Rocket. As long as the design meets the basic requirements for the interface, any accelerator can be interchangably attached to Rocket, thus be utilized as part of the SOAR architecture. The full SOAR architecture configured on Xilinx Aritx-7 FPGA is able to boot RISC-V Debian Linux from the SD-card and can communicate with other machines over Ethernet connection. Now that we have specified the hardware infrastructure implemented on reconfigurable hardware, we describe the software framework for adaptive and easy-to-use control of accelerators in the following chapter.

5. Framework for Adaptive Utilization of Accelerators

In this chapter, we describe the software stack that lies on top of the hardware described in Chapter 4, an additional layer that will enable SOAR to make self-aware decisions about how to optimally use its accelerators. The SOAR software handles the invocation of function calls, both in hardware and in software, gauges dynamic power demands, and determines which set of accelerators should be configured and activated at each time. Ideally, all of these operations should be carried out in hardware to further reduce performance and power overhead. Hence, in upcoming iterations of SOAR, we expect to integrate our support software into the underlying hardware architecture itself. More details on future directions are found in Chapter 7.

5.1 Support for Dynamic Dispatch

Given a list of accelerator-implemented functions, we need to be able to execute each function both in hardware and in software. As proposed, SOAR does not keep a hard-wired list of accelerators that are utilized each and every time. Instead, the system invokes an accelerator only when the unit is both configured and activated, which occurs once SOAR has determined it is most advantageous to do so. In any other case, whenever the functions is called, SOAR will execute the function via a software library call. This seamless transitioning between HW-SW execution modes through dynamic dispatch allows our system to benefit from efficient hardware-software collaboration.

The SOAR software first implements a top-level wrapper function for each accelerator; the wrapper then uses simple `if-else` control logic to invoke either the hardware or software implementation. Decisions about dynamic dispatch are based on a metadata array that is described below in Table 5.1.

Name	Data Type	Description	Config Type
`index`	`int`	Index in the *entire* list of accelerator functions	Static
`customId`	`int`	Index in the *current* HW configuration	Dynamic (CT)
`hw_avail`	`int`	Whether accelerator is currently configured on-chip	Dynamic (CT)
`hw_on`	`int`	Whether accelerator is currently decided to be used	Dynamic (RT)
`hw_fun`	`void *`	Pointer to hardware function call	Static
`sw_fun`	`void *`	Pointer to software function call	Static
`speed_reward`	`int`	Pre-calculated performance gains	Static
`power_reward`	`int`	Pre-calculated power savings	Static
`count`	`int`	Combined HW-SW function invocation count	Dynamic (RT)

Table 5.1: Fields in each accelerator metadata. CT indicates hardware compile-time; RT stands for run-time.

Each element in the array corresponds to a packet of metadata for each accelerator. The first parameter, `index`, marks the location of the accelerator within the entire list of accelerators. On the other hand, `customId` notes the location in the currently configured array of on-chip accelerators. `hw_avail` is the parameter indicating whether the accelerator is currently configured on-chip, and `hw_on` indicates whether the accelerator should be used or not. `hw_fun` and `sw_fun` are function pointers to the hardware and software implementations. While the hardware activation variables are dynamically configured, the function implementations themselves are fixed at development-stage. Hardware methods are implemented using in-line assembly instructions invoking custom RoCC instructions, and the software methods are the corresponding C implementations described in Section 4.2.2. Finally, the speed and power rewards are pre-computed performance and power-efficiency results, averaged over multiple runs, and the `count` variable records the number of times an accelerator function has been called, either in hardware or in software.

5.2 Determining the Working Set of Accelerators

As discussed, the run-time selection between HW-SW execution depends on the metadata parameter `hw_on`. To build a fully adaptive system, however, we also require that this parameter to be updated at run-time, according to changes in the running workload and power budget. Therefore, SOAR periodically executes a self-evaluation process to determine what is the current *working set* of accelerators. Similar to its original meaning in operating systems, we define a working set of accelerators to be the "optimal collection of activated accelerators referenced by a running process

during a given time interval."

Calculating this working set is completed in two steps: first, the system gauges the run-time power consumption and the frequency at which each accelerator function has been invoked, either in hardware or in software. Next, based on the measured data and pre-calculated `speed_reward` and `power_reward` value, the system determines (1) whether each on-chip available accelerator should be activated or not and (2) whether any of the configured accelerators should be exchanged for a more useful unit. The mechanism for comparing the use of different accelerators is based on the user-defined heuristic function. Hence, SOAR can be personalized to work towards either optimized performance or energy-efficiency—or any other goal the user sets.

Note that the process of determining the working set is itself an expensive operation that cannot be executed frequently. While the dispatch mechanism is invoked every time an accelerator function is called by the running workload, the SOAR updates the accelerator activation parameters (`hw_on`) only once every minute (60 seconds) and reasons about hardware reconfiguration every five minutes (300 seconds). Given a large enough time frame, we expect that the overhead of computing the working set will be amortized over hundreds of millions of optimized operations SOAR will execute during the same time frame.

5.2.1 Power and Usage Monitoring

Monitoring dynamic power consumption is key to making self-aware decisions about improving energy-efficiency. If the system is already drawing too much power, SOAR will attempt to configure and activate accelerators with higher power rewards. Ideally, we would measure the accurate power demands of the FPGA via direct hardware instrumentation tools, yet we have not had the opportunity to fully explore this possibility in this work. We hope to explore direct hardware-based power instrumentation in the future, as we discuss more in Chapter 7. For this version of SOAR, we instead use the values from the Linux `loadavg` kernel as a proxy for power consumption.

The Linux load average kernel calculates the current number of processes either in the running (R) or waiting (D) states. In a single threaded system, a low value from `loadavg` reasonably corresponds to a situation where the system is mostly idle, and while the operating system is idle, RISC-V Linux directs the processor to enter its power-saving mode. In this state, the processor is stalled with the `wfi` (wait for interrupt) instruction and most of the core is gated off, hence saving dynamic power that would have otherwise been dissipated through switching transistors. In our current implementation of the SOAR software, the `loadavg` kernel is invoked every minute, interrupting the current workload; the system records the readings to the metadata array and feeds information to the heuristic function. We have also explored the possibility of directly counting

the number of cycles spent on executing `wfi` instructions at the hardware level, through adding an extra counter to Rocket CSR. This method would have allowed us to measure the frequency of processor stalling without having to undergo kernel swaps every minute. Unfortunately, `wfi` cycle counting has been unsuccessful when tested on our physical FPGA implementation, so we decide to use the Linux kernel for power measurements.

As mentioned earlier when discussing Table 5.1, the SOAR software keeps a counter for each accelerator in the metadata array. Unlike power monitoring, updating the counters do not incur any kernel invocations and can be executed without much overhead. Therefore, SOAR updates the counters each and every time an accelerator function is execute. Also, in order to respond to situations where the workload characteristics drastically change over time, all counters are reset to zero every five minutes.

5.2.2 Heuristic-based Customization

SOAR operates with a user-given heuristic to determine the working set of accelerators. We purposefully design our system to be *goal-agnostic*, such that the use of accelerators can be tailored towards any goal defined by the user. Here, however, we consider the specific case of optimizing performance under a specific power budget as an example to illustrate how the heuristic function might be designed.

Since most accelerators improve performance (`speed_reward`), SOAR will default to activating (`hw_on`) all accelerators that are currently configured on-chip (`hw_avail`). Readings from the power monitor will check if the defined power budget is being met: if so, `hw_on` and `hw_avail` will not be changed; otherwise, SOAR will de-activate any accelerators with negative power rewards (`power_reward`). As mentioned above, re-evaluation of accelerator activation takes place every 60 seconds.

Over 300 second intervals, SOAR also determines the potential need for hardware reconfiguration. Suppose the on-chip area limit is up to two accelerators at a time. Reviewing the `count` parameters in the metadata array, SOAR will list the two most highly demanded accelerator functions. If those two accelerators differ from what is currently configured on-chip, then SOAR computes the product of speed reward and function invocation frequency for each accelerator and compares the products. If a non-configured unit scores higher than a current on-chip unit, then SOAR prints to `stdout` how the hardware should be reconfigured. Ideally, partial reconfiguration will take place when such need is noticed, yet this process has not been included as part of our current implementation. Potential improvements on the SOAR design are fully discussed in Chapter 7.

5.3 Summary

The SOAR software infrastructure is designed to support two major operations: (1) switching between hardware and software function execution via dynamic dispatch and (2) adaptively adjusting the system accelerator usage policies via determining the working set of accelerators. Recording accelerator configuration parameters in a metadata array has allowed our system to be scalable and manageable, since we only need to extend the metadata array to integrate additional RoCC accelerators into the architecture. We also highlight the fact that this library does not require any inputs from the user besides the heuristic for comparing the utilities of accelerators. Furthermore, the heuristic function can be specified in any form, allowing SOAR to customize the system towards arbitrary optimization goals. Operations carried out by the SOAR software does introduce performance and power consumption costs, yet in the end, we expect that the returns from optimized use of accelerators can significantly outweigh such overheads.

6. Evaluation

In this chapter, we evaluate the performance and energy-efficiency of our proposed architecture. We describe the methodology, tools and benchmarks that were used in our measurements and report the final results.

6.1 Methodology

To demonstrate the effectiveness of our proposed system, we measure the speed-up and power savings of the full SOAR architecture relative to the default Rocket implementation. The difference between the two versions is that SOAR additionally contains a set of on-chip RoCC accelerators and it makes use of those accelerators through the software system described in Chapter 7. The rest of the two designs are identical, including that we implement both RTL on Xilinx Artix-7 FPGA, as part of the Digilent Nexys-A7 board.

The benchmarks we use are the following accelerator functions: Bloom filter operations, binary string generation, `strcpy` and `strcmp`. When calling these functions, SOAR decides between executing in software or invoking one of the hardware accelerators, whereas the default implementation always runs in software. We compare the performance and power consumption between these two cases. Though it is reasonable to conduct the measurements *bare-metal*, we specifically choose to measure the system efficiencies while running RISC-V Debian Linux, since we expect that most real-world users will be making use of SOAR along with a reasonable operating system.

6.2 Hardware Utilization

Before discussing performance and power efficiency, we briefly report how SOAR utilizes FPGA hardware components, compared to the default Rocket core. As mentioned, the only difference between the two hardware architectures is that SOAR has several additional RoCC accelerators. The total number of accelerators units included in SOAR can vary as long as the FPGA has enough

hardware resources to implement the full RTL design. The example shown in Table 6.1 lists the hardware utilization of a SOAR architecture that has been configured with the four accelerators described in Chapter 4. The table shows that the hardware resources required for RoCC accelerators are far less than what is required for the rest of the host CPU. The full Linux-booting system, including IO and memory peripherals, currently occupies about 88% of total LUTs available on Aritx-7 XC7A100T-1CSG324C. Given that each of accelerators require 1 6% of the LUTs, we can conclude that most commercial FPGAs are able to support a SOAR implementation at least 6 to 12 fine-grained RoCC accelerators.

	LUTs	**FFs**	**BRAMs**	**DSP48**
Total Available	63400	126800	135	240
Rocket	39998 (63.1%)	23754 (18.7%)	65 (48.1%)	15 (6.25%)
Bloom	3626 (5.72%)	267 (0.21%)	0	0
Combinations	3225 (5.09%)	400 (0.32%)	0	0
Strcpy	221 (0.35%)	99 (0.08%)	0	0
Strcmp	137 (0.22%)	34 (0.03%)	0	0

Table 6.1: Summary of SOAR hardware utilization

6.3 Performance Results

Performance was measured in physical hardware time via the `gettimeofday` system call. Measuring with actual time is a much more accurate way to gauge system performance than counting the number of instruction cycles spent on . Although each standard RISC-V instruction is designed to consume a single clock cycle, custom RoCC instructions, on the other hand, may take multiple cycles to execute. `gettimeofday` provides a direct and manageable way to measure physical time, and by looping through the benchmark functions millions of times, we bring up the total measurement time interval up to several minutes. This approach secures greater accuracy in our measurements.

Table 6.2 summarizes the average operations per second when executing each of the functions supported by the combinatorial sequence generation accelerator, both in hardware and in software. For 16-bit binary strings, we see that the fixed-weight and ranged combination generation are accelerated using hardware; however, for general combinations, we see that running the function on hardware is actually *slower* than running in software. General combinations is thus an interesting

example of a case where using hardware may potentially improve power savings but at the cost of negatively impacting performance.

	Software	**Hardware**	**Relative Speed-up**
Fixed-Weight Combinations	39.36	147.12	3.74×
General Combinations	5.91	3.67	0.62×
Ranged Combinations	4.67	48.32	10.4×

Table 6.2: Executed operations per second for each function of the combinatorial sequence generation accelerator, when generating 16-bit-wide binary sequences.

6.4 Energy-Efficiency Results

Power measurements of an FPGA is itself a challenging task. While we have been unable to find appropriate instrumentation tools to specifically measure power consumption of the FPGA chip, we have measured the gross power consumption rate of the entire FPGA board using the P3 P4400 Kill-A-Watt Electricity Usage Monitor. The Kill-A-Watt device provides up to three significant figures on the average power consumption rates in kilowatts per hour. For accurate measurements, we executed benchmark workloads in an infinite loop, so that the FPGA can run the workload overnight–after about 24 hours, we terminated the running process and recorded the power consumption rate readings.

	Software	**Hardware**
Ops per Second	4.67	48.32
Power Consumption	0.04 KWH	0.025 KWH
Energy-Delay Product	510 μJ$\cdot s$	3.0 μJ$\cdot s$

Table 6.3: Energy-efficiency measurement results for the number-theoretic sequence generation accelerator, when generating ranged combinations

In general, we suspect that using hardware helps reduce energy consumption. Unfortunately, we have only been able to test our hypothesis against a specific case: we measured the power-

efficiencies of ranged combination generations using hardware and in software. The results are shown in Table 6.3. We notice that the hardware is capable of speeding up the operation by 10× while consuming less power. The energy-delay product (EDP), a measure of energy-efficiency frequently used in characterizing microarchitecture, is defined as

$$EDP = P_{avg} \times (\Delta T)^2 = \text{Avg Power Consumption (Watts)} \times \left(\frac{\text{Elapsed Time (s)}}{\text{Elapsed Loop Count}} \right)^2.$$

EDP calculations indicate that hardware generation of ranged combinations is about 170× more efficient than the software implementation.

6.5 Summary

Our evaluation of SOAR hardware utilization suggests that the architecture can support 6 to 12 RoCC accelerators at the same time, even when implemented on a moderate, commercial-grade FPGA. Performance and power-efficiency measurements, albeit incomplete and require further analyses, indicate that most of our accelerators significantly improve performance, if not also reduce dynamic power consumption, compared to equivalent software implementations. In future research endeavors, we look forward to seeing further evaluations of the heuristic-based reconfiguration of accelerators and its impact on SOAR performance and energy-efficiency.

7. Future Work

The SOAR architecture described in the previous chapters is a first step towards a fully self-optimizing SoC that can adaptively tailor its hardware to the running program and time-varying efficiency requirements. As mentioned several times, there are many directions in which future works may improve our current implementation. Here, we discuss some of the potential research questions.

7.1 Improvements on the Hardware Architecture

7.1.1 Diversifying the Collection of Accelerators

An immediate extension of this work is to develop an SoC with more RoCC accelerators. One of the greatest advantages of the SOAR design is that the system can operate with a large pool of accelerators, while the system itself will automatically choose the optimal set of units to configure and activate on-chip. In other words, with more accelerators available, SOAR can deliver better efficiencies over diverse applications. As mentioned in Chapters 3 and 4, developing a small, fine-grained accelerator like our examples is not a daunting task thanks to the wealth of open-source RISC-V hardware development tools. For instance, both Chisel and the Rocket Chip generator are specifically designed to ease the burden of designing RoCC accelerators. We thus anticipate that many exciting accelerators can be designed and deployed onto SOAR in the near future.

7.1.2 Further Investigating the Host-Accelerator Memory Interface

The default RoCC interface allows the accelerators to share the L1 data cache with Rocket. At the moment, our accelerators can successfully execute memory requests and responses during RTL simulations via Verilator, but performing memory operations on the physical FPGA implementation results in a complete system failure. We currently do not understand where the error comes from, yet we suspect that the synchronization over the RoCC memory interface is not properly configured

on hardware. Along with debugging, we also hope to explore if there are more efficient methods for transferring data between Rocket and accelerators, even perhaps between the accelerators. Memory coherence and consistency for such heterogeneous SoCs are interesting open-ended research topics.

7.1.3 Replacing the Software Infrastructure with Integrated Hardware Units

In Chapter 5, we mentioned how the SOAR software infrastructure can incur significant overheads. Operations performed the software, including monitoring power consumption and determining the working set of accelerators, are all tasks that involve the use of Linux kernels, which are bound to interrupt the running program; however, it is also necessary to continue executing these functions in order to maintain the functionality of the SOAR design. We bring here again the notion that hardware is often more efficient than software in executing the same set of tasks. Using hardware is even more advantageous when we have full knowledge of which specific operations the circuit should handle. Therefore, to truly optimize SOAR, one may consider replacing the current SOAR software framework with corresponding hardware units, integrating them into the architecture described in Chapter 4.

For example, power monitoring may be improved by using an actual printed circuit board (PCB) attachment that can directly measure the analog power consumption rates of the FPGA. Readings can be fed back to the FPGA through an analog-to-digital converter. Similarly, accelerator utilization monitoring can be done using custom Control Status Registers (CSRs) within Rocket. Signals from the Rocket instruction decoder can be designed to trigger specific registers to count each time a RoCC instruction has been executed. As such, we can imagine various ways to implement hardware designs to carry out the operations currently handled in software.

7.1.4 Designing a Custom Compiler

Another way to improve SOAR is through adding a `gcc`-extended compiler that can identify function calls in the source code and compile into SOAR-specific instructions. In the current implementation, we have only been working with C implementation code that we wrote alongside the accelerators—to use the SOAR software framework, accelerator wrapper functions must be manually placed in the source code. Designing a custom compiler will be a meaningful project in that it will spare the burden of users during the application development process and will also allow users to bring any C source code to run on SOAR.

7.1.5 Incorporating Partial Reconfiguration into SOAR

Partial Reconfiguration (PR) can be a powerful addition to the SOAR system. Offered by higher-end commercial FPGAs, PR allows the hardware to re-configure a particular portion of itself at run-time, therefore being able to run the rest of the implemented design while a part of the chip is getting updated. Applying PR to SOAR will allow the system to execute the recommended changes to the on-chip configuration of accelerators at run-time, instead of at some later compile-time. This offers a higher-degree of dynamic customization and will bring the architecture closer to a fully self-optimizing hardware architecture.

7.2 Improvements on Performance and Power Measurements

In this work, we have analyzed how our set of RoCC accelerators offer speed-ups and energy savings. Experimental results show that wisely using these accelerators can significantly improve the efficiencies of running various applications. The next step in understanding the promise of SOAR is to analyze how the self-aware optimization process can further contribute to improving system efficiency, over a wide range of benchmarks. An example would be an experiment where we run a mixture of well-known benchmarks such as the applications in SPECint and track (1) how SOAR determines the working set of accelerators and suggests hardware to be reconfigured and (2) whether SOAR improves energy-efficiency more than a SoC with a fixed-set of accelerators.

8. Conclusion

Power-efficiency has become a key design constraint, as the prevalence of *dark silicon* continues to grow in modern processors. In response to tightening on-chip power budgets, computer architects have pointed at specialization of hardware, through the use of DSAs such as accelerators, to extend the scaling of system performance. This work is a continuation of such endeavors towards building a high-performance, power-efficient SoC based on accelerators. However, the SOAR system differentiates itself from previous SoC designs in that it explores how to perform optimally not only over a few specialized tasks, but also for a wide range of general-purpose workloads. SOAR presents an automated framework for adaptively controlling the use of accelerators and tailoring the hardware to the running program.

In this work, we have implemented a full-scale, RISC-V SoC based on the lowRISC extension of the Rocket Chip generator. Using Chisel HDL, we explored an approach for designing modular accelerators that allows less-experienced hardware developers to easily generate special-purpose units. We implemented several of such homegrown accelerators and attached them to our SoC through the RoCC interface. The entire architecture was implemented on a Xilinx Artix-7 FPGA and is capable of invoking individual accelerators while booting RISC-V Linux. Finally, we introduced a software infrastructure based on dynamic dispatch to seamlessly switch code execution between hardware and software. We demonstrated a flexible framework that allows any heuristic function to be applied to determining the working set of accelerators.

Experiments described in Chapter 6 indicate that our collection of RoCC accelerators can significantly improve performance and energy-efficiency. Attaching more accelerators to SOAR will allow SOAR to choose from a larger pool of potential hardware units and become even more efficient. Unfortunately, we have not been able to include efficiency evaluations of the heuristic-based working set determination as part of this thesis. By completing such analyses and exploring the various topics mentioned in Chapter 7, we expect future iterations of SOAR to become the fully reconfigurable, adaptive and easy-to-use system we have proposed in this work.

Appendix A. Acronyms

ASIC application specific integrated circuits (ASICs) are integrated circuits designed for a particular application. They generally cannot be used for other purposes.

BRAM block RAMs that are placed on an FPGA fabric to provide more efficient memory storage for FPGA circuits.

CPU the central processing unit (CPU) performs the instructions of a computer program.

DSP Digital Signal Processor are hardware units on Xilinx FPGAs responsible for fast and efficient multiplication and addition.

FPGA field programmable gate arrays (FPGAs) are integrated circuits that can be reconfigured after they have been manufactured.

GPP general purpose processors (GPPs) are processors that can be used for a wide variety of programs and applications.

IP intellectual property (IP) cores are how Xilinx defines the individual hardware blocks that can be implemented on their FPGAs.

LUT look-up tables (LUTs) replace computation with indexing into a stored data array. Most computation on FPGAs is implemented through select lines that index into LUTs to encode boolean logic functions.

RAM random access memory (RAM) is a type of data storage in which read and write times are independent of order of access.

VHDL VHSIC Hardware Description Language (VHDL) is a hardware description language used to define digital systems and integrated circuits. VHDL and Verilog are the two most widely used hardware description languages.

Appendix B.
Chisel Code for RoCC Accelerators

```
// RoCC Bloom filter accelerator
class BloomAccel (implicit p: Parameters) extends LazyRoCC {
  override lazy val module = new BloomAccelImp(this)
}

class BloomAccelImp(outer: BloomAccel) extends LazyRoCCModule(outer) with
    HasCoreParameters {
  // accelerator memory
  val bloom_bit_array = RegInit(Vec(Seq.fill(20000)(0.U(1.W))))
  val miss_counter = RegInit(0.U(64.W))
  // val busy = RegInit(Bool(false))

  val cmd = Queue(io.cmd)
  val funct = cmd.bits.inst.funct
  val hashed_string = cmd.bits.rs1

  // decode RoCC custom function
  val doInit = funct === UInt(0)
  val doMap = funct === UInt(1)
  val doTest = funct === UInt(2)


  // Hash computation
  val x0  = Wire(UInt())
  val y0  = Wire(UInt())

  val x1  = Wire(UInt())
  val y1  = Wire(UInt())
```

```
28
29    val x2  = Wire(UInt())
30    val y2  = Wire(UInt())
31
32    val x3  = Wire(UInt())
33    val y3  = Wire(UInt())
34
35    val x4  = Wire(UInt())
36    val y4  = Wire(UInt())
37
38    val x5  = Wire(UInt())
39    val y5  = Wire(UInt())
40
41    x0 := hashed_string
42    y0 := hashed_string >> 4
43
44    x1 := (x0 + y0) % 20000.U(64.W)
45    y1 := (y0 + 0.U(64.W)) % 20000.U(64.W)
46 n
47    x2 := (x1 + y1) % 20000.U(64.W)
48    y2 := (y1 + 1.U(64.W)) % 20000.U(64.W)
49
50    x3 := (x2 + y2) % 20000.U(64.W)
51    y3 := (y2 + 2.U(64.W)) % 20000.U(64.W)
52
53    x4 := (x3 + y3) % 20000.U(64.W)
54    y4 := (y3 + 3.U(64.W)) % 20000.U(64.W)
55
56    x5 := (x4 + y4) % 20000.U(64.W)
57    y5 := (y4 + 4.U(64.W)) % 20000.U(64.W)
58
59    val found1 = Wire(UInt())
60    val found2 = Wire(UInt())
61    val found3 = Wire(UInt())
62    val found4 = Wire(UInt())
63    val found5 = Wire(UInt())
64
65    found1 := bloom_bit_array(x1)
66    found2 := bloom_bit_array(x2)
67    found3 := bloom_bit_array(x3)
68    found4 := bloom_bit_array(x4)
69    found5 := bloom_bit_array(x5)
```

```
70
71  // Custom function behaviors
72  when (cmd.fire()) {
73    when (doInit) {
74      bloom_bit_array := Reg(init = Vec.fill(20000)(0.U(1.W)))
75      miss_counter := RegInit(0.U(64.W))
76      // fresh := Bool(true)
77    }
78    when (doMap) {
79      bloom_bit_array(x1) := 1.U(1.W)
80      bloom_bit_array(x2) := 1.U(1.W)
81      bloom_bit_array(x3) := 1.U(1.W)
82      bloom_bit_array(x4) := 1.U(1.W)
83      bloom_bit_array(x5) := 1.U(1.W)
84    }
85    when (doTest) {
86      miss_counter := miss_counter + ~(found1 & found2 & found3 & found4 & found5)
87    }
88  }
89
90  // PROCESSOR RESPONSE INTERFACE
91  // Control for communicate accelerator response back to host processor
92  val doResp = cmd.bits.inst.xd
93  val stallResp = doResp && !io.resp.ready
94
95  cmd.ready := !stallResp
96    // Command resolved if no stalls AND not issuing a load that will need a
     request
97  io.resp.valid := cmd.valid && doResp
98    // Valid response if valid command, need a response, and no stalls
99  io.resp.bits.rd := cmd.bits.inst.rd
100   // Write to specified destination register address
101 // io.resp.bits.data := bloom_bit_array(7081.U(64.W))*1000.U(64.W) +
     bloom_bit_array(9951.U(64.W))*100.U(64.W)
102 io.resp.bits.data := miss_counter
103 // io.resp.bits.data := Mux(doMap, debug, miss_counter)
104   // Send out
105 io.busy := cmd.valid
106   // Be busy when have pending memory requests or committed possibility of
     pending requests
107 io.interrupt := Bool(false)
108   // Set this true to trigger an interrupt on the processor (not the case for
```

```
    our current simplified implementation)
}
```

Listing 8.1: RoCC Bloom filter accelerator implemented in Chisel

```

// RoCC accelerator for number-theoretic seqeunce generation
class Combinations(implicit p: Parameters) extends LazyRoCC {
    override lazy val module = new CombinationsImp(this)
}

//Main accelerator class, directs instruction inputs to functions for computation
class CombinationsImp(outer: Combinations)(implicit p: Parameters) extends
    LazyRoCCModule(outer) with HasCoreParameters{
    //Accelerator states: idle, busy (accessing memory), resp (sending response)
    val s_idle :: s_busy :: s_resp :: Nil = Enum(Bits(), 3)
    val state = Reg(init = s_idle) //State idle until handling an instruction
    val tryStore = state === s_busy

    //Instruction inputs
    val length = Reg(init = io.cmd.bits.rs1) //Length of binary string
    val previous = Reg(init = io.cmd.bits.rs2) //Previous binary string
    val rd = Reg(init = io.cmd.bits.inst.rd) //Output location
    val function = Reg(init = io.cmd.bits.inst.funct) //Specific operation
    val currentAddress = Reg(UInt(64.W)) //The address to use for memory stores
    //Always-updated versions of the inputs
    val fastLength = Mux(io.cmd.fire(), io.cmd.bits.rs1, length)
    val fastPrevious = Mux(io.cmd.fire(), io.cmd.bits.rs2, previous)

    //Answers for each function: FixedWeight, General, Ranged, then memory
    versions of each (functions 0-6)
    val outputs = Array(nextCombination.fixedWeight(fastLength(5,0), fastPrevious)
    , nextCombination.generalCombinations(fastLength(5,0), fastPrevious),
    nextCombination.rangedCombinations(fastLength(5,0), fastPrevious, fastLength
    (11,6), fastLength(17,12)))

    //Command and response states
    io.cmd.ready := state === s_idle
    io.resp.valid := state === s_resp

    //Accelerator response data
    val summedReturns = Reg(init = 0.U(64.W))
    //For a 3-bit function code, bit 2 sets whether memory is used or not, and
```

```
    bits 1 and 0 set which combination to use
34  val lookups = Array(0.U->outputs(0),1.U->outputs(1), 2.U->outputs(2),
35      4.U->summedReturns, 5.U->summedReturns, 6.U->summedReturns)
36  io.resp.bits.data := MuxLookup(function, outputs(0), lookups)
37  io.resp.bits.rd := rd
38
39
40  //State control
41  //Setup for processing commands
42  when(io.cmd.fire()) {
43      //Capture inputs
44    length := io.cmd.bits.rs1
45    rd := io.cmd.bits.inst.rd
46    function := io.cmd.bits.inst.funct
47
48      //Whether it's a memory-using instruction or not (bit 2 set in the
    function code)
49    when(io.cmd.bits.inst.funct(2)===1.U) {
50      state := s_busy
51      summedReturns := 0.U
52      currentAddress := io.cmd.bits.rs2
53    } .otherwise {
54          previous := io.cmd.bits.rs2
55        state := s_resp
56    }
57  }
58
59  //When done with an instruction
60  when(io.resp.fire()) {
61      state := s_idle
62  }
63
64
65  //Memory-access state
66  val memAccesses = Reg(init = 0.U(4.W)) //Whether all memory requests have been
     resolved
67  val accessesChange = Wire(UInt(4.W))
68  accessesChange := (io.mem.req.fire() & 1.U(4.W)) - (io.mem.resp.valid & 1.U(4.
    W))//The latest amount of memory accesses either started or finished
69
70  //Source of new combination data
71  val nextCombinations = Array(memoryAccess.cycleCombinations(fastLength, io.mem
```

```
    .req.fire(), io.cmd.fire(), 0), memoryAccess.cycleCombinations(fastLength, io.
    mem.req.fire(), io.cmd.fire(), 1), memoryAccess.cycleCombinations(fastLength,
    io.mem.req.fire(), io.cmd.fire(), 2))
72  val memLookups = Array(0.U -> nextCombinations(0), 1.U -> nextCombinations(1),
     2.U -> nextCombinations(2))
73  val combinationStream =  Wire(UInt(64.W))
74  combinationStream := MuxLookup(function(1,0), nextCombinations(0), memLookups)
75
76  //Request and response controls
77  //When a request is sent, set up next cycle's response data
78  when(io.mem.req.fire()) {
79      memAccesses := memAccesses + accessesChange
80      currentAddress := currentAddress + 8.U
81      printf("next: %x address %x mem: %x\n", combinationStream, currentAddress,
     memAccesses)
82  }
83
84  //When a response is received, save response data
85  when(io.mem.resp.valid) {
86      memAccesses := memAccesses + accessesChange
87      summedReturns := summedReturns + io.mem.resp.bits.data
88      printf("tag: %x addr: %x mem %x\n", io.mem.resp.bits.tag, io.mem.resp.bits
    .addr, memAccesses)
89  }
90
91
92  //Controls for accessing memory
93  val cycleOver = combinationStream === nextCombination.doneSignal
94  val finished = cycleOver //&& memAccesses === 0.U
95
96  //Switch out of memory mode when finished
97  when(tryStore && finished) {
98    state := s_resp
99  }
100
101 //Memory request interface
102 io.mem.req.valid := tryStore && !cycleOver
103 io.busy := tryStore
104 io.mem.req.bits.addr := currentAddress
105 io.mem.req.bits.tag :=  combinationStream(5,0) //Change for out-of-order
106 io.mem.req.bits.cmd := 1.U
107 io.mem.req.bits.data := combinationStream //combinationStream
```

```
108 //    io.mem.req.bits.size := log2Ceil(8).U
109 //    io.mem.req.bits.signed := Bool(false)
110     io.mem.req.bits.phys := Bool(false)
111 
112     //Always false
113     io.interrupt := Bool(false)
114 }
115 
116 
117 //Generates binary string combinations based on input constraints and saves them
    to memory.
118 object memoryAccess {
119     //Depending on the type for cycleCombinations, a different combination pattern
     will be used.
120     def cycleCombinations(constraints: UInt, getNext: Bool, reset: Bool, kind: Int
    ) : UInt = {
121         val initial = Wire(UInt(64.W)) //The first value of the cycle
122         if(kind == 1) { //The general cycle starts and ends with all 1s
123             initial := (1.U << constraints(5,0)) - 1.U
124         } else { //The other cycles start with lower 1s filled according to
    allowed weights
125             initial := (1.U << constraints(11,6)) - 1.U
126         }
127 
128         val nextSent = Reg(UInt(64.W)) //The value currently saved for storing to
    memory
129         val result = kind match { //Calculate next value as the last is being
    stored
130             case 0 => nextCombination.fixedWeight(constraints(5,0), nextSent)
131             case 1 => nextCombination.generalCombinations(constraints(5,0),
    nextSent)
132             case 2 => nextCombination.rangedCombinations(constraints(5,0),
    nextSent, constraints(11,6), constraints(17,12))
133         }
134 
135         //Cycle by one when next value requested
136         when(getNext) {
137         nextSent := result
138       }
139 
140         //Start new cycle of the requested length when a reset is requested
141       when(reset) {
```

```
142         nextSent := initial
143       }
144       nextSent
145     }
146 }
147 
148 //These methods generate the next combination for a certain function with the
    given parameters
149 object nextCombination {
150     def doneSignal = UInt("hffffffff") //Signal to return upon a finished cycle
    (64 bit -1)
151 
152     //Generates a fixed-weight binary string based on a previous string of the
    same
153     //weight and length. Binary strings up to length 32 will work.
154     def fixedWeight(length: UInt, previous: UInt) : UInt = {
155         //Calculations to generate the next combination (From Knuth's algorithm
    for Williams' 'cool' ordering)
156         //Mask up to the right-most '10' of bits
157         val trimmed = previous & (previous + 1.U) //Remove trailing 1s
158         val trailed = trimmed ^ (trimmed - 1.U) //Make a mask for the right-most
    '10' onwards (if no '10', mask everything)
159         val indexTrailed = trailed & previous //Mask the previous string
160 
161         //Create a duplicate of the mask if the bit before the last '10' is a 1
162         val indexShift = trailed + 1.U //Set the bit to the left of the mask (or
    nothing set if mask is of everything)
163         val subtracted = (indexShift & previous) - 1.U //If the bit masked by
    indexShift is 1, subtracted is the mask, if it is 0, subtracted has all bits
    set
164         val fixed = Mux(subtracted.asSInt < 0.S, 0.U, subtracted) //If no '10', or
     the bit at indexShift isn't 1, fixed is 0, otherwise it is subtracted
165 
166         //Rotate masked bits to get the result, return if the cycle isn't over yet
167         val result = previous + indexTrailed - fixed //Rotate the right side of
    the string starting from indexShift, or the whole string if indexShift not set
168         val stopper = 1.U(1.W) << length(5,0) //Set the bit to the left of the
    binary string
169 
170         //Fill result with all 1s if finished
171         Mux(result >> length =/= 0.U, doneSignal, result % stopper) //The end of
    the cycle has been reached if the bit at stopper is set in the new string
```

```
172     }
173
174     //Generates the next binary string of a certain length based on the cool-er
    ordering
175     def generalCombinations(length: UInt, previous: UInt) : UInt = {
176         //Calculations to generate the next combination (Algorithm by Maddie to
    generate Stevens' and Williams' 'cooler' orderings)
177         //Mask up to the right-most '01' before the end of the string
178         val trimmed = previous(31,1) | (previous(31,1) - 1.U) //Remove trailing 0s
179         val trailed = trimmed ^ (trimmed + 1.U) //Make a mask for the right-most
    01 onwards
180         val mask = Wire(UInt(32.W)) //Shift the mask to a 32 bit wire instead of
    31
181         mask := (trailed << 1.U) + 1.U
182
183         //Find the last spot in the mask, to use for rotating the 0th bit
184         val lastTemp = Wire(UInt(32.W))
185         lastTemp :=  trailed + 1.U //If there is a valid 01, this is the last bit
186         val lastLimit = 1.U << (length(5,0) - 1.U) //Otherwise use the final bit
187         val lastPosition = Mux(lastTemp > lastLimit || lastTemp === 0.U, lastLimit
    , lastTemp) //Choose which bit position to use
188
189         val cap = 1.U << length(5,0) //One bit beyond the width of the string
190         val first = Mux(mask < cap, 1.U & previous, 1.U & ~previous) //Flip the
    first bit if there is no valid 01
191         val shifted = (previous & mask) >> 1.U //Shift the masked region
192         val rotated = Mux(first === 1.U, shifted | lastPosition, shifted) //Move
    the first bit to the end of the shifting
193         val result = rotated | (~mask & previous) //Combine the rotated and non-
    rotated parts of the string
194
195         Mux(result === (cap - 1.U), doneSignal, result) //If finished, the result
    is all 1s
196     }
197
198     //Generates the next binary string within a weight range, based on cool-est
    ordering
199     def rangedCombinations(length: UInt, previous: UInt, minWeight: UInt,
    maxWeight: UInt) : UInt = {
200         //Calculations to generate the next combination (Algorithm by Maddie to
    generate Stevens' and Williams' 'coolest' orderings)
201         //Mask up to the right-most '01' before the end of the string
```

```
202         val trimmed = previous(31,1) | (previous(31,1) - 1.U) //Remove trailing 1s
203         val trailed = trimmed ^ (trimmed + 1.U) //Make a mask for the right-most
    01 onwards
204         val mask = Wire(UInt(32.W)) //Shift the mask to a 32 bit wire instead of a
     31 bit wire
205         mask := (trailed << 1.U) + 1.U
206
207         //Find the last spot in the mask, used for rotating the 0th bit
208         val lastTemp = Wire(UInt(32.W))
209         lastTemp :=  trailed + 1.U //If there is a valid '01', this is the last
    bit
210         val lastLimit = 1.U << (length(5,0) - 1.U) //Otherwise use the string's
    final bit
211         val lastPosition = Mux(lastTemp > lastLimit || lastTemp === 0.U, lastLimit
    , lastTemp) //Choose which bit position to use
212
213         val count = Wire(UInt(32.W))
214       count := PopCount(previous(31,0)) //Count the number of set bits in the
    string, which should be within the weight constraints
215
216         val cap = 1.U << length(5,0) //Set a bit one beyond the string's width
217         val flipped = 1.U & ~previous //Take the complement of the 0th bit
218         val valid = Mux(flipped === 0.U, count > minWeight, count < maxWeight) //
    Check if still a valid weight with that bit changed
219         val first = Mux(mask < cap || !valid, 1.U & previous, flipped) //Flip the
    first bit if there is no valid 01
220         val shifted = (previous & mask) >> 1.U //Shift the masked region
221
222         //Flip the bit while rotating if no 01 and new string is valid
223         val rotated = Mux(first === 1.U, shifted | lastPosition, shifted) //Move
    the first bit
224         val result = rotated | (~mask & previous) //Add the first bit to the final
     result
225
226         Mux(result === (1.U << minWeight) - 1.U, doneSignal, result) //Return -1
    if finished
227     }
228 }
```

Listing 8.2: RoCC accelerator for binary bit pattern generation implemented in Chisel

```
1 // RoCC strcpy accelerator
2
```

```
/**
 * Lazy module for accelerator for comparing two strings.
 */
class Strcpy(implicit p: Parameters) extends LazyRoCC {
   override lazy val module = new StrcpyImp(this)
}

/**
 * This accelerator compares two strings; the result is the byte difference
 * the first differing bytes, or 0.  This implementation assumes that strings
 * are resident in virtual memory.  Calls to this instruction may require mapping
 * and/or locking pages to physical memory.
 */
class StrcpyImp(outer: Strcpy)(implicit p: Parameters)
   extends LazyRoCCModule(outer) with HasCoreParameters
{
   // states of the state machine
  val idleState :: readState :: writeState :: doneState :: Nil = Enum(Bits(),4)
   val state = Reg(init = idleState) // idle -> compare* -> done -> idle
   val srcPtr = RegInit(0.U(64.W))  // pointer into the string
   val dstPtr = RegInit(0.U(64.W))  // pointer into the string
   val strVal = Reg(UInt(8.W))      // the character read
   val request = Reg(init=false.B)  // true iff request is in transit to mem

   // back pressure to core: only ready when not computing
   io.cmd.ready := (state === idleState)   // when is this accelerator ready?
   // when accelerator uses mem; perhaps reduced to just compare state?
   io.busy := (state =/= idleState)
   // request validity is stored in "request"
   io.mem.req.valid := request

   /**
    * when we make memory requests, we're reading a single byte from memory
    * (or cache) at a time.  These bytes are unsigned (if they'd been signed
    * then the data field would be sign-extended).  The addresses we use are
    * virtual addresses; we depend on the caching system to perform the page
    * table walking for us (this is important because a translation failure may
    * require re-play of the request).
    */
   // static portions of the request
   val R = 0.U
   val W = 1.U
```

```
45    val cmd = Reg(init = R)
46    val data = Reg(init = 0.U(8.W))
47    val ptr = Reg(init = 0.U(64.W))
48    val size = Reg(init = 0.U(4.W))
49
50    val done = (state === writeState) && (!request) && (data === 0.U)
51
52    io.mem.req.bits.cmd := cmd   // R/W
53    io.mem.req.bits.size := 0.U  // log2(n); n is one byte
54    io.mem.req.bits.signed := false.B  // value is unsigned
55    io.mem.req.bits.data := data  // write data
56    io.mem.req.bits.phys := false.B  // pointers are virtual addresses
57    io.mem.req.bits.addr := ptr   // R/W address
58    io.mem.req.bits.tag := 0.U   // identify the source string (A=0, B=1)
59
60    /**
61     * The state machine.
62     * This machine starts in idle.  When a command is received, the request
63     * to memory is made and is held valid for the ready cycle and one more.
64     * Response must be collected when valid.
65     */
66    switch (state) {
67      is (idleState) {
68         when (io.cmd.fire()) {
69            srcPtr := io.cmd.bits.rs1 // first String
70            dstPtr := io.cmd.bits.rs2 // second string
71      // now, set up initial request: read from source string
72            request := true.B
73      cmd := R
74      ptr := io.cmd.bits.rs1
75      data := 0.U
76            state := readState                // move to first stage request
77         }
78      }
79      is (readState) {
80        // request is now *not* valid; await response is valid
81        when (RegNext(io.mem.req.fire())) {
82          request := false.B
83        }
84        // on "rising edge" of response:
85        when (io.mem.resp.valid && !RegNext(io.mem.resp.valid)) {          //
   memory as response data
```

```
86            srcPtr := srcPtr+1.U  // move source along
87            // set up write request to destination
88            cmd := W
89        ptr := dstPtr
90            data := io.mem.resp.bits.data
91            val ch = io.mem.resp.bits.data(7,0)
92            printf("Writing '%c' to %x\n",ch,dstPtr);
93            state := writeState
94            request := true.B
95          }
96        }
97        is (writeState) {
98          // request is now *not* valid; await response is valid
99          when (RegNext(io.mem.req.fire())) {
100           request := false.B
101         }
102         when (!request) {
103           dstPtr := dstPtr+1.U
104           // (possibly) set up next read:
105           cmd := R
106       ptr := srcPtr
107           data := 0.U
108           state := Mux(done,idleState,readState)
109           request := !done  // if we're not done, we're reading
110         }
111       }
112     }
113     // combinational circuitry for response building
114     io.resp.valid := false.B
115     io.interrupt := false.B
116 }
```

Listing 8.3: RoCC `strcpy` accelerator implemented in Chisel

```
1 // RoCC strcmp accelerator
2
3 /**
4  * Lazy module for accelerator for comparing two strings.
5  */
6 class FullStrcmp(implicit p: Parameters) extends LazyRoCC {
7    override lazy val module = new FullStrcmpImp(this)
8 }
9
```

```
/**
 * This accelerator compares two strings; the result is the byte difference
 * the first differing bytes, or 0.  This implementation assumes that strings
 * are resident in virtual memory.  Calls to this instruction may require mapping
 * and/or locking pages to physical memory.
 */
class FullStrcmpImp(outer: FullStrcmp)(implicit p: Parameters)
   extends LazyRoCCModule(outer) with HasCoreParameters
{
   // states of the state machine
   val idleState :: compareState :: doneState :: Nil = Enum(Bits(),3)
   val state = Reg(init = idleState) // idle -> compare* -> done -> idle
   val aPtr = RegInit(0.U(64.W))  // pointer into the string
   val bPtr = RegInit(0.U(64.W))  // pointer into the string
   val aVal = Reg(UInt(8.W))      // the character read
   val bVal = Reg(UInt(8.W))      // the character read
   val diff = aVal - io.mem.resp.bits.data  // computed difference of a and b
   val done = ((aVal === 0.U) ||  // finish predicate
               (io.mem.resp.bits.data === 0.U) || (diff =/= 0.U))
   val source = Reg(init=0.U(1.W))  // which (A=0 or B=1) string is the source
   val destReg = Reg(init=io.cmd.bits.inst.rd)  // register to write to
   val request = Reg(init=false.B)  // true iff request is in transit to mem

   // back pressure to core: only ready when not computing
   io.cmd.ready := (state === idleState)   // when is this accelerator ready?
   // when accelerator uses mem; perhaps reduced to just compare state?
   io.busy := (state =/= idleState)
   // request validity is stored in "request"
   io.mem.req.valid := request

   /**
    * when we make memory requests, we're reading a single byte from memory
    * (or cache) at a time.  These bytes are unsigned (if they'd been signed
    * then the data field would be sign-extended).  The addresses we use are
    * virtual addresses; we depend on the caching system to perform the page
    * table walking for us (this is important because a translation failure may
    * require re-play of the request).
    */
   // static portions of the request
   io.mem.req.bits.cmd := 0.U   // read
   io.mem.req.bits.size := 0.U  // log2(n); n is one byte
   io.mem.req.bits.signed := false.B  // value is unsigned
```

```
52    io.mem.req.bits.data := Bits(0)  // data written (never used)
53    io.mem.req.bits.phys := false.B  // pointers are virtual addresses
54    io.mem.req.bits.addr := Mux(source === 0.U, aPtr, bPtr)  //
55    io.mem.req.bits.tag := source   // identify the source string (A=0, B=1)
56
57    /**
58     * The state machine.
59     * This machine starts in idle.  When a command is received, the request
60     * to memory is made and is held valid for the ready cycle and one more.
61     * Response must be collected when valid.
62     */
63    switch (state) {
64      is (idleState) {
65          when (io.cmd.fire()) {
66             destReg := io.cmd.bits.inst.rd  // get destination register
67             aPtr := io.cmd.bits.rs1 // first String
68             bPtr := io.cmd.bits.rs2 // second string
69             request := true.B
70             state := compareState                // move to first stage request
71          }
72      }
73      is (compareState) {
74        // request is now *not* valid; await response is valid
75        when (RegNext(io.mem.req.fire())) {
76          request := false.B
77        }
78        when (io.mem.resp.valid && !RegNext(io.mem.resp.valid)) {          //
    memory as response data
79           request := true.B
80           when (source === 0.U) {
81              aVal := io.mem.resp.bits.data
82              aPtr := aPtr+1.U
83           } .otherwise {
84              bVal := io.mem.resp.bits.data
85              bPtr := bPtr+1.U
86              when (done) {
87                 request := false.B;
88                 state := doneState
89              }
90           }
91           source := ~source
92        }
```

```
93      }
94      is (doneState) {
95        when (io.resp.fire()) {
96          state := idleState
97        }
98      }
99    }
100   // combinational circuitry for response building
101   io.resp.bits.rd := destReg
102   io.resp.bits.data := diff
103   io.resp.valid := state === doneState
104   io.interrupt := false.B
105 }
```

Listing 8.4: RoCC `strcmp` accelerator implemented in Chisel

Bibliography

[1] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Commun. ACM*, vol. 62, no. 2, pp. 48–60, Jan. 2019, ISSN: 0001-0782. DOI: 10.1145/3282307. [Online]. Available: http://doi.acm.org/10.1145/3282307.

[2] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation cores: Reducing the energy of mature computations," *SIGPLAN Not.*, vol. 45, no. 3, pp. 205–218, Mar. 2010, ISSN: 0362-1340. DOI: 10.1145/1735971.1736044.

[3] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, Jun. 2011, pp. 365–376.

[4] M. B. Taylor, "Is dark silicon useful? Harnessing the four horsemen of the coming dark silicon apocalypse," in *DAC Design Automation Conference 2012*, Jun. 2012, pp. 1131–1136.

[5] R. Razdan and M. D. Smith, "A high-performance microarchitecture with hardware-programmable functional units," in *Proceedings of the 27th Annual International Symposium on Microarchitecture*, ser. MICRO 27, San Jose, California, USA: Association for Computing Machinery, 1994, pp. 172–180, ISBN: 0897917073. DOI: 10.1145/192724.192749. [Online]. Available: https://doi.org/10.1145/192724.192749.

[6] T. J. Callahan, J. R. Hauser, and J. Wawrzynek, "The Garp architecture and C compiler," *Computer*, vol. 33, no. 4, pp. 62–69, Apr. 2000, ISSN: 1558-0814. DOI: 10.1109/2.839323.

[7] S. Yehia, S. Girbal, H. Berry, and O. Temam, "Reconciling specialization and flexibility through compound circuits," in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, Feb. 2009, pp. 277–288. DOI: 10.1109/HPCA.2009.4798263.

[8] S. Swanson and M. B. Taylor, "Greendroid: Exploring the next evolution in smartphone application processors," *IEEE Communications Magazine*, vol. 49, no. 4, pp. 112–119, Apr. 2011. DOI: 10.1109/MCOM.2011.5741155.

[9] G. Venkatesh, J. Sampson, N. Goulding-Hotta, S. K. Venkata, M. B. Taylor, and S. Swanson, "QSCORES: Trading dark silicon for scalable energy efficiency with quasi-specific cores," in *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec. 2011, pp. 163–174.

[10] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-54, May 2014. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html.

[11] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The Rocket Chip Generator," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, Apr. 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html.

[12] Y. Lee, A. Waterman, R. Avizienis, H. Cook, C. Sun, V. Stojanović, and K. Asanović, "A 45nm 1.3GHz 16.7 double-precision GFLOPS/W RISC-V processor with vector accelerators," in *ESSCIRC 2014 - 40th European Solid State Circuits Conference (ESSCIRC)*, Sep. 2014, pp. 199–202. DOI: 10.1109/ESSCIRC.2014.6942056.

[13] H. Mao, "Hardware acceleration for memory to memory copies," Master's thesis, EECS Department, University of California, Berkeley, Jan. 2017. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-2.html.

[14] J. Koenig, D. Biancolin, J. Bachrach, and K. Asanovic, "A hardware accelerator for computing an exact dot product," in *2017 IEEE 24th Symposium on Computer Arithmetic (ARITH)*, Jul. 2017, pp. 114–121. DOI: 10.1109/ARITH.2017.38.

[15] S. Davidson, S. Xie, C. Torng, K. Al-Hawai, A. Rovinski, T. Ajayi, L. Vega, C. Zhao, R. Zhao, S. Dai, A. Amarnath, B. Veluri, P. Gao, A. Rao, G. Liu, R. K. Gupta, Z. Zhang, R. Dreslinski, C. Batten, and M. B. Taylor, "The Celerity open-source 511-core RISC-V tiered accelerator fabric: Fast architectures and design methodologies for fast chips," *IEEE Micro*, vol. 38, no. 2, pp. 30–41, Mar. 2018. DOI: 10.1109/MM.2018.022071133.

[16] S. Eldridge, T. J. Watson, V. Verma, R. S. Joshi, and P. Bose, "A low voltage RISC-V heterogeneous system boosted SRAMs, machine learning, and fault injection on VELOUR," in *First Workshop on Computer Architecture Research with RISC-V (CARRV 2017) at the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50)*, Oct. 2017.

[17] B. Keller, M. Cochet, B. Zimmer, J. Kwak, A. Puggelli, Y. Lee, M. Blagojević, S. Bailey, P. Chiu, P. Dabbelt, C. Schmidt, E. Alon, K. Asanović, and B. Nikolić, "A RISC-V processor SoC with integrated power management at submicrosecond timescales in 28 nm FD-SOI," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 7, pp. 1863–1875, Jul. 2017. DOI: 10.1109/JSSC.2017.2690859.

[18] Z. Ye, A. Moshovos, S. Hauck, and P. Banerjee, "CHIMAERA: A High-Performance Architecture with a Tightly-Coupled Reconfigurable Functional Unit," Apr. 2000.

[19] S. Y. Shao, "Design and modeling of specialized architectures," PhD thesis, Graduate School of Arts and Sciences, Harvard University, 2016. [Online]. Available: https://dash.harvard.edu/handle/1/33493560.

[20] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a Scala embedded language," in *DAC Design Automation Conference 2012*, Jun. 2012, pp. 1212–1221. DOI: 10.1145/2228360.2228584.

[21] P. S. Li, A. M. Izraelevitz, and J. Bachrach, "Specification for the FIRRTL language," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-9, Feb. 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-9.html.

[22] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra, Q. Huang, K. Kovacs, B. Nikolic, R. Katz, J. Bachrach, and K. Asanović, "FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA '18, Los Angeles, California: IEEE Press, 2018, pp. 29–42, ISBN: 978-1-5386-5984-7. DOI: 10.1109/ISCA.2018.00014. [Online]. Available: https://doi.org/10.1109/ISCA.2018.00014.

[23] S. Karandikar, A. Ou, A. Amid, H. Mao, R. Katz, B. Nikolić, and K. Asanović, "FirePerf: FPGA-accelerated full-system hardware/software performance profiling and co-design," Mar. 2020, pp. 715–731. DOI: 10.1145/3373376.3378455.

[24] lowRISC, *The lowRISC chip*, version 0.6 refresh. [Online]. Available: `https://github.com/lowRISC/lowrisc-chip`.

[25] D. Knuth, *The Art of Computer Programming, Volume 4, Fascicle 3*. Pearson Education, 2005, ISBN: 0201853949.

[26] B. Stevens and A. Williams, "The coolest way to generate binary strings," *Theory of Computing Systems*, vol. 54, no. 4, pp. 551–557, May 2014.