Using Reconfigurable Hardware to Fight Dark Silicon

by
Isaiah Leonard

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 19, 2015

Contents

List of Figures

List of Tables

Abstract

After four decades of exponential improvement, computing performance has hit a wall. Power constraints and physical limitations have led to the rise of *dark silicon*: most modern processors can run only 3% of their transistors at maximum frequency. Pairing reconfigurable hardware with a general purpose processor (GPP) to support partnered computation can yield performance and efficiency improvements. Because software processors waste roughly 90% of their energy on instruction interpretation, customized hardware can significantly decrease the energy utilization of targeted computations. In this work, we explore the possibility of using a GPP tightly coupled to a dynamically reconfigurable co-processor to provide efficient, general purpose, partnered computation.

1. Introduction

Computing performance has hit a wall [23]. While the number of transistors that can fit in a given area continues to increase, power constraints have led to the rise of *dark silicon* as many of those transistors must be left unused. Chip manufacturers have responded to this voltage scaling crisis by producing multi-core chips and enabling so-called "turbo modes" that power down most of the processor in order to run a small fraction of its transistors (about 3% on modern cores) at their maximum clock frequency without overheating. While using multiple cores to compute in parallel is one path to increased performance under these power constraints, we currently lack languages that would make parallel code easy to write as well as programmers who are well-versed in the principles and pitfalls of parallel programming [6]. Even the best efforts at increasing the parallelization of computation are unlikely to yield orders-of-magnitude improvement in the near future, as they must wring parallelism out of explicitly sequential code. Thus, as a direct result of the power utilization wall imposed by the breakdown of voltage scaling, chip space is becoming a less valuable resource while power is growing more important.

There are several ways of trying to exploit this power-space tradeoff. One of the more promising paths is pairing a general purpose processor (GPP[1]) with a set of tightly-coupled heterogeneous units, each of which is designed to increase the energy-efficiency of executing a target piece of code. UCSD's *conservation cores* (c-cores) project is a particularly effective and well-researched implementation of this heterogeneous core-GPP coupling [23]. However, the system that the UCSD researchers describe requires that chips be printed in a conventional manner with a set of inflexible application specific integrated circuits (ASICs) on board. The mostly static nature of these conservation cores would make them less useful for many users (who may not use the applications that the generated cores target) and shorten their utility-lifespan, as the target codebase changes and invalidates the hardware designed for the outdated software.

Field programmable gate arrays (FPGAs) represent a compromise between GPPs and ASICs. While FPGAs are generally less space efficient and less computationally powerful

[1] A table of acronyms can be found on page 35.

than both GPPs and ASICs, they are more flexible than ASICs and, when compiled to target a specific set of code, can provide roughly 10× performance and power gains over GPPs [3]. Unlike earlier generations of FPGAs, systems like the Xilinx Zynq, which pairs an ARM processor with an FPGA on a single chip, are fast, reconfigurable, and tightly-coupled enough to be used for real-time partnered computation [27]. Therefore, Zynq and other hybrid FPGA-GPP chips are an ideal platform for the conservation cores concept, as they could yield significant performance benefits without locking users into a set of immutable circuits.

In the future, every processor could be a hybrid, pairing a GPP with a reconfigurable FPGA co-processor. Ideally, the GPP would execute as little code as possible, with most computation performed on the more efficient FPGA. Further, one can imagine that smart-phone processors will be coupled to an FPGA, and software updates and application downloads will include not only application code, but also configurations for FPGA regions that can efficiently execute that code. It would even be possible for a phone to tailor its FPGA configurations to its owner's usage patterns by periodically running self-analysis and recompiling its hardware.

This project represents the first small steps towards that future. In it, we describe a runtime reconfigurable datapath with a simple user interface that is implemented on the Zynq's programmable logic. While FPGA hardware has made significant advances and is now highly capable, the software and documentation supporting them remain byzantine [27], making FPGAs and the new Zynq hardware in particular difficult to use. Therefore, in this work we aim to give a proof-of-concept and roadmap for a coarser-grained FPGA co-processor that exposes FPGA efficiency and flexibility while hiding its complexity.

The remainder of this work is organized as follows. Chapter 2 talks about previous research on energy efficiency; comparisons between GPPs, ASICs, and FPGAs; automated circuit design and analysis; heterogeneous co-processor systems; and systems on the Zynq platform. Chapter 3 presents a typical Zynq project workflow by walking through the creation and testing of a circuit that computes Hailstone sequence lengths. The results of the tests hint at the performance capabilities of the Zynq system. In Chapter 4, we discuss the design and goals for the runtime reconfigurable datapath, and we present experimental performance results in Chapter 5. Chapter 6 outlines some of the options for further work to enhance the capability of an FPGA co-processor. Finally, Chapter 7 is a brief summary.

2. Previous Work

Previous research related to this work can be grouped into four main categories. First, there are a few papers that describe the increasing importance of energy efficiency compared to chip space caused by the breakdown of traditional voltage scaling. Second, there is a large body of research focused on the relative efficiency of computation performed on GPPs, FPGAs, and ASICs. Third, there have been a number of attempts to efficiently automate circuit design, synthesis, and analysis, using design space exploration and various tuning heuristics to manage the trade-off between compilation time and circuit quality. Finally, many researchers have designed and tested systems that integrate GPPs with specialized, heterogeneous hardware accelerators using both compile-time code analysis and runtime computation partitioning. We consider those contributions here.

2.1 Dark Silicon and the Importance of Energy Efficiency

In a pair of important papers, Taylor of UCSD describes the utilization wall imposed by physical limitations on the amount of voltage required to avoid gate leakage [21, 22]. Previously, voltage scaled down with the size of the transistors, meaning that from one generation to the next, chip makers could add more transistors without increasing power requirements. In 2005, however, process size finally decreased to the point that transistors became too small to allow linear voltage scaling without gate leakage, so the voltage required for each transistor has remained roughly constant. Therefore, more power is required per unit area, so not all of the transistors can be operating under the original power budget. The result of this utilization wall is *dark silicon*, and an era in which industry "progress is measured by improvements in transistor energy efficiency" instead of transistor speed and count. Indeed, given the realities of dark silicon, chip area is decreasingly valuable while energy efficiency is a vital design parameter. Taylor describes four possible responses to dark silicon: (1) shrinking chip sizes to increase silicon utilization; (2) dynamically managing chip-wide power usage by under-clocking parts of the chip to compute as much as possible while staying within the power budget; (3) using the dark silicon to house a set of specialized

co-processors each of which can much more efficiently (in terms of time, power, or both) perform necessary computations; or (4) hoping for a significant advance in semiconductor technology that would make the voltage-scaling wall irrelevant. Taylor is most optimistic about the use of specialized co-processors and emphasizes the importance of making careful decisions about how to best use "extra" chip space for more efficient computation.

2.2 Efficiency Comparisons Between GPPs, FPGAs, and ASICs

In 2010, Hameed et al. analyze the inefficiencies of GPPs compared to ASICs [8]. Most notably, they find that GPPs spend over 90% of their computation energy on overhead operations (such as instruction fetch and decode) that can be completely eliminated by specialized hardware. As a result, ASICs are generally around 500 times more energy efficient than GPPs. Beyond that computational overhead, ASICs are still over 50 times more efficient than GPPs because they can exploit parallelism, group complex instructions that are often used together, and make use of specialized hardware specifically designed for a target application. Overall, without custom hardware and instructions, computation energy is still dominated by overhead, with less than 10% going to the functional units. The authors' "inescapable conclusion is that truly efficient designs will require application-specialized hardware."

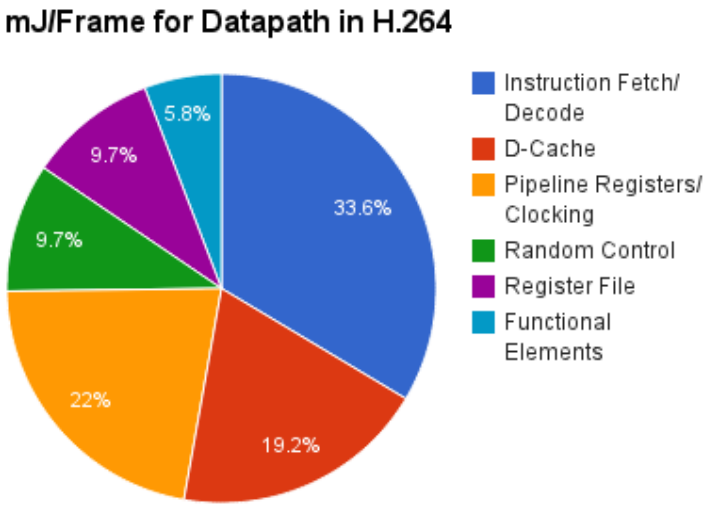

Figure 2.1: Chart of GPP computation energy. Roughly 5% of computation energy goes to the functional units, with most being spent on computational overhead. Data from Hameed et al. [8]

Amdahl's Law is an important checkpoint on claims of performance improvements when using parallel techniques. Hill and Marty of University of Wisconsin-Madison refine the

thinking behind Amdahl's Law in speedup formulae tuned for multicore processors that are symmetric, asymmetric, or dynamic [10]. Symmetric processors dedicate the same resources to each multicore (e.g. 16 identical 4-resource multicores on a 64-resource chip), asymmetric processors have differently sized cores (e.g. one 16-resource core and 24 2-resource cores), and dynamic processors can be run-time reconfigured to run as one powerful core for sequential processing or as a set of possibly asymmetric smaller cores for parallel processing. Though these multicores do not have application specific hardware, asymmetric multicores represent a similar approach: trying to break up the larger computation into pieces that can be handled by optimally-sized sections of hardware. While the authors emphasize that even in the multicore era, sequential speedups are crucially important, and targeting those bottlenecks can still provide valuable efficiency gains, they find that orders-of-magnitude gains come only when large amounts of potential parallelism (>95%) can be exploited. Another key conclusion is that even when some decisions or configurations may seem locally inefficient, they can be globally efficient if they reduce total chip idle time. Overall, dynamic multicore designs outperform asymmetric multicores, and asymmetric multicore designs perform better than symmetric ones.

Kuon and Rose use detailed experimental measurements to examine the area, energy, and delay gaps between FPGAs and ASICs [12]. Overall, they find FPGA logic requires roughly 35× more area (mostly due to routing), 14× more power, and a 3-4× longer critical path. Most modern FPGAs have some hard logic blocks (arithmetic logic units (ALUs), random access memory (RAM) blocks, etc.), which can reduce those gaps to around 18× for area, and 8× for power, depending on how much of the application hardware can be mapped to those hard blocks. The cost of those improvements is the loss of flexibility on the FPGA, as fabric space must be dedicated to the hard blocks, whether or not they are used.

Chung et al. provide a general survey of the benefits and trade-offs associated with integrating unconventional cores (u-cores) with GPPs [3]. The work focuses on energy efficiency, rather than pure performance or speed, noting that power consumption and I/O bandwidth are the two main performance bottlenecks for modern computation. The authors evaluate three different types of u-cores (ASICs, FPGAs, and GPGPUs) paired with GPPs, modeling a situation in which sequential operations are performed on the GPP and parallel operations are mapped to the u-core. They find that u-cores almost always improve the energy efficiency of computation, especially as the amount of parallelism in the target application increases. In addition, the benefits of u-core use are heavily dependent on off-chip bandwidth trends, as bandwidth ceilings quickly become a limiting factor for extremely fast custom logic; if I/O and bandwidth improvements continue to lag behind logic improvements, FPGAs will be able to offer performance that is closer to that of ASICs

than their relative computational efficiency would suggest. The authors' results show that ASICs offer both the highest performance and most power efficiency, but that ASICs are expensive to develop and designed around a single set of target applications. FPGAs, on the other hand, can provide significant efficiency gains over GPPs alone while offering more flexibility (and thus a longer utility life-span) than ASICs.

2.3 Automated Circuit Design, Synthesis, and Analysis

In their 1999 paper, Aditya et al. describe a system that automates the design of application-specific very long instruction word (VLIW) and explicitly parallel instruction computing (EPIC) processors and instruction sets to efficiently implement an input set of target code [2]. More than simply outputting one feasible design, however, the authors' program-in-chip-out (PICO) system performs extensive design-space exploration to handle the "intelligent, quantitative, cost-performance trade-offs" for customized architecture. The PICO system has three main components: (1) a design-space explorer, which uses various heuristics and search strategies to find close-to-optimal architectures; (2) an architecture synthesis sub-system which takes the resulting abstract architecture and devises a pseudo-optimal concrete architecture and compiler target; and (3) a retargetable compiler which generates the best possible instruction set and application code for the given application and processor design passed to it. PICO uses the three subsystems in a cyclical fashion, as the compiler's output and execution time estimates are used by the design-space explorer as input to the search heuristics looking for optimal architectures. The authors' preliminary tests show that processors with heterogeneous functional units are almost always more efficient in terms of size and performance than processors which have only homogeneous functional units.

PRML, a partially reconfigurable modeling language, enables designers to efficiently explore the design space for a given application, using different area and performance metrics to tailor the tradeoffs to their workload [11]. In many ways, PRML seems to be a modern iteration of the automated synthesis and design space exploration discussed by Aditya et al. While those authors were trying to generate efficient designs for VLIW processors, however, Kumar and Gordon-Ross direct their automation and design space exploration towards partially reconfigurable FPGAs. One of the biggest points of emphasis is the explicit tradeoff analysis in PRML's architecture. There are always going to be tradeoffs between runtime and quality of output for the modeling, and area and performance in the final design. Making those tradeoffs easy to manage can only help designers. The basic internal tradeoff analysis uses a generated, partially reconfigurable architecture and compares area, performance, and inter-region communication overheads, and the system gives the user a set of tuning parameters to adjust which of those tradeoffs the automated

design prioritizes.

Sankar and Rose examine the trade-off between FPGA compile time and routing quality [20]. Longer compilation times represent a substantial impediment to many of the primary applications for FPGAs including rapid prototyping, emulation, and custom computation. Using techniques such as recursive clustering of hardware blocks to build up a design and then recursive routing and placement of those blocks starting with the full design, the authors find that designs within one-third of optimal can be achieved in less than 10% of the compilation time. Any real-time compilation is computation-intensive and likely to produce relatively low quality circuits. However, perhaps a more important application is the use of fast routing and placement to estimate the quality of a more optimal design. The authors report that with under 10 seconds of runtime, their system can compile designs whose wire lengths can be used to estimate, with roughly 5% error, optimal circuit wire lengths.

More recently, in 2010 Lavin et al. of Brigham Young University created RapidSmith, a system for automated FPGA design that aims to significantly speed up FPGA compilation times [13, 14, 15, 16]. Their principal improvement comes from the creation and use of pre-compiled and internally routed macro-blocks which can be combined quickly without requiring further low-level verification. Experimental results indicate that the circuits designed in this manner provide performance only 25% worse than that of hand-tuned circuits, while development time is reduced by 67%. Further, the researchers replace the default Xilinx placer and router, which handle the hard macros very poorly, with HMFlow [14], which yields a 10-50× speedup compared to the Xilinx tools in configuration time. However, the circuits that come out of HMFlow are limited to 50% of the resources on the FPGA; they are also 2-4× slower than those produced by the default tools because not re-compiling internal routing necessarily limits optimization.

2.4 Systems with Heterogeneous Co-processors

Several early projects in this area targeted very small reconfigurable accelerators that act only as a single functional unit.

Razdan and Smith describe using programmable hardware units to accompany a GPP for a broad range of applications [19]. The programmable functional units (PFUs) developed by the authors implement a set of combinational functions within a single instruction cycle time. A new reduced instruction set computing (RISC) instruction selects from up to 2048 possible PFU configurations and then returns the appropriate result. At a higher level, the authors' 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 all reconfiguration overhead) and develop a toolchain that will automatically implement that reconfiguration.

The CHIMAERA system is similar to Razdan and Smith's, but the hardware is slightly more powerful: Ye et al. use compile-time analysis and the addition of one new instruction to the instruction set to map sequences of operations to one reconfigurable functional unit [28]. The authors find that the greatest performance gains come not from simple shift operations (which are quite easily mapped to the reconfigurable unit), but rather from reconfigurable unit operations that replace branching logic by implementing speculative execution in hardware.

Hauser and Wawrzynek's GARP system pairs a MIPS processor with an FPGA co-processor [9]. Though the hardware available at the time was much less capable than modern systems (the FPGAs were too small to encode entire programs and could only hold one configured circuit at a time), the main issues that concern the authors remain relevant: finding pieces of code that will be more efficient on an FPGA, handling the I/O, and achieving optimal runtime allocation of computation. Simulations suggest that for certain applications, FPGAs were much more efficient than the most powerful GPPs of the period.

More recently, in a 2005 paper Clark et al. discuss a framework to integrate customized instructions for a configurable compute accelerator (CCA) into the instruction set for a GPP [4]. The authors' main innovation is a compiler that integrates a user-specified CCA configuration into compile-time analysis, allocating appropriate code blocks to the CCA. While most of the analysis is done at compile-time, the framework also uses run-time analysis to create and branch to the code necessary for the CCA. Determining when using the CCA will improve efficiency given the overhead involved and overlapping code subgraphs that could otherwise be mapped to the CCA is crucially important and extremely difficult.

Quinn et al. describe the Dynamo system, a successful implementation of runtime partitioning for platforms that have a mix of GPPs and FPGAs; it handles the assignment, compilation, and execution, leaving only pipeline selection (i.e. designing efficient circuits and source code) to the user [18]. The system automatically selects the most efficient combination of hardware and software and generates source code to implement that partitioning. Dynamo uses pre-compiled hardware implementations so that it can work dynamically and efficiently—having to generate those designs at runtime would slow down processing. Along with the hardware implementation specifications themselves, performance profiles are pre-computed, allowing the system to accurately decide whether hardware or software would be more efficient for a given step. The researchers conclude that every phase of processing has a crossover point between smaller image sizes (which are more efficiently processed in software) and larger ones (for which the speedup of the FPGA outweighs the overhead).

Similarly, shorter pipelines should be handled entirely in software, while longer pipelines should mostly be handled by the FPGA with only simple computational steps remaining in software.

In 2010, Venkatesh et al. from UCSD explored the benefits of pairing specialized conservation cores (c-cores) with GPPs, significantly reducing the power consumption for frequently executed, energy intensive code [23]. The authors' primary focus is not increased performance, but rather similar performance with lower energy consumption: "If a given computation can be made to consume less power at the same level of performance, other computations can be run in parallel without violating the power budget." The voltage-scaling utilization wall has made energy efficiency, rather than pure computational speed, the primary bottleneck. Because they target energy efficiency, rather than computation speed, c-cores can be used for a much broader set of applications or code blocks, even when there is no available parallelism to exploit. Even when there is not parallelism to be exploited (or when acceleration via increased parallelism is not the goal), using c-cores simply to make computation more energy efficient is worthwhile because it allows more of the chip to be functional at any given time. The c-core system, as described, contains multiple tiles, each of which contains one GPP and several heterogeneous c-cores as shown in Figure 2.2. The researchers' toolchain generates c-cores by profiling a target workload, selecting "hot" code regions, and automatically synthesizing c-core hardware to implement the code block. The compiler must be extended to use available c-cores on applicable code segments, while executing other code on the GPP. The authors also discuss "patchable" c-cores, which have limited flexibility (increasing their lifetime and the amount of code they can execute) but incur significant (2×) area and power consumption overhead. Simulations suggest that a system using only 18 c-cores can have a 16× improvement in energy consumption for code that can be executed on the c-cores, resulting in a reduction of nearly 50% for the energy consumption of the target applications as a whole.

GreenDroid is an evolution of the c-cores paradigm targeting the Android platform [7]. Android is a good target platform because it has a relatively well-defined set of basic applications, short hardware update cycle that mitigates the effect of out of date c-cores, and (as with any mobile platform) a heightened emphasis on power savings. The experimental results show an 11× energy savings through the use of hundreds of different c-cores (compared to only 18 in the authors' initial research [23]) that are able to cover roughly 95% of the test Android-based workload. The automated toolchain generates a system design with 16 tiles, each of which has a general purpose central processing unit (CPU), a large L1 cache (used for communication with the c-cores), I/O ports, and between 8 and 15 c-cores.

Quasi-specific (QS) cores are a more general implementation of c-cores: while each c-core targets only one specific piece of code, QS cores are designed to handle similar

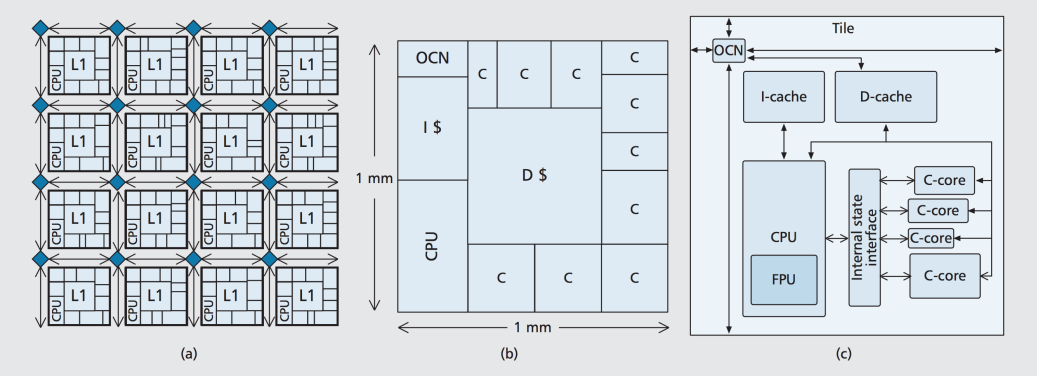

Figure 2.2: A diagram showing (a) a c-core chip with multiple tiles, (b) the physical floorplan for an individual tile, and (c) the logical layout of the interconnects within a tile. Figure from Goulding-Hotta et al. [7]

(graphically isomorphic) pieces of code across hot-spotted applications [24]. To generate QS-cores, Venkatesh et al. add a phase in their processor design methodology that searches for similar code patterns across the input applications (matching based on the semantics of the code, not its text). The researchers use design-space exploration to find optimal QS-core configurations given the input code and chip space requirements. For example, when designing cores to handle *find, insert, and delete* operations on a few basic data structures, four QS-cores are able to cover the same number of code segments as 11 c-cores. This reduces not only the total area required for the heterogeneous cores, but also the amount and complexity of the interconnects. Preliminary results indicate that the QS-cores are 18 times more energy efficienct than GPPs and require 66% less specialized logic than comparable ASICs.

2.4.1 Systems on the Zynq Platform

First released in 2011, the Xilinx Zynq system consists of two standard ARM processors (the processing system (PS)) that are tightly coupled to an FPGA fabric (the programmable logic (PL)), significantly lowering the communication overhead between GPPs and FPGA co-processors [27].

In 2013, Dobai and Sekanina used the Zynq system on chip (SoC) to implement an evolvable, reconfigurable system [5]. The paper focuses on evolutionary algorithms that iteratively generate possible configurations for a circuit, test those configurations, evaluate their performance against a fitness function, and then tweak them. The Zynq's tight coupling of ARM processors and the programmable logic allows for faster data transfers and more adaptive reconfiguration. For this application, the experimental results show that the Zynq is over 400× more efficient than an ARM processor alone, and over 100×

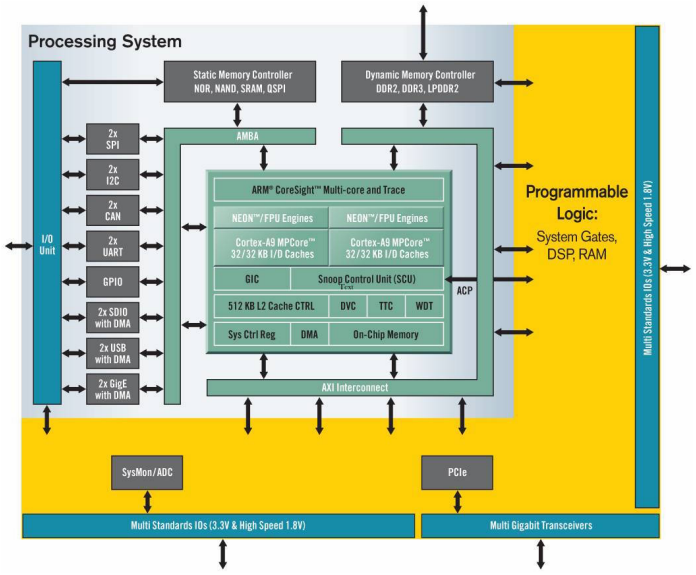

Figure 2.3: The Zynq SoC. Illustration from Xilinx.

more efficient than a high-performance desktop processor. Two different reconfiguration methodologies are compared: virtual reconfigurable circuits (VRC) which incur significant space overhead but require less reconfiguration and dynamic partial reconfiguration (DPR) acceleration which has no area overhead (since circuits are reconfigured more aggressively) but does incur a significant reconfiguration overhead. Which of those two methods is more efficient depends largely on how many reconfigurations are expected and how much of the PL can be utilized concurrently by the application.

Pendlum et al. look at using a Zynq system to perform cognitive radio (CR) analysis by partitioning computation between the Zynq's ARM processors and its PL [17]. CR algorithms have two main components: spectrum sensing (monitoring the activity on the licensed wavelengths) and spectrum decision (the dynamic, fast selection of which spectrum to use). Spectrum sensing, which has much data parallelism, maps well to hardware implementation, while spectrum sensing, which is mostly sequential, is generally more suited for a traditional processor. Although the authors achieve significantly better performance when executing the entire computation on the PL than when allocating the sensing to the PL and the decision to the PS, achieving dynamic partitioning on the Zynq platform represents a significant accomplishment.

In a pair of 2014 papers, Vipin and Fahmy present systems for automatically and efficiently managing partial reconfiguration (PR) on the Xilinx Zynq platform [25, 26]. A Zynq-like system is ideal for PR because the complex reconfiguration algorithms can be processed on the ARM PS, while the data-intensive computation can be done on the con-

figured PL. While PR obviously incurs space and communication overhead (since some configurations will use their region sub-optimally), it enables much greater overall efficiency because the rest of the FPGA can function while one region is being reconfigured. The authors' open-source, automated CoPR system allows the user to define a set of configuration specifications (to be automatically translated into bitstreams for partially reconfigurable regions (PRRs)) and adaptation specifications (defining the algorithm that decides when and how the chip should be reconfigured) in C code, ignoring the physical implementation that the system designs. ZyCAP, an open-source PR controller, uses the ARM processors on the Zynq to dynamically control the reconfiguration of the PL using a set of small accelerators. The goal is to have the processor only explicitly manage very high-level reconfiguration management, leaving lower-level mechanisms for other hardware. This strategy allows the PS to be independently operational most of the time, instead of being occupied with the reconfiguration and management of the PL. By leveraging the different reconfiguration and bitstream input options, the authors manage to achieve near maximum throughput for the PR management, improving over 3× (and up to 20×) compared to standard implementations. Experimental results show that only as processing times get very large (amortizing the reconfiguration costs) do the standard implementations provide comparable application performance.

2.5 Summary

As discussed in [21, 23] and others, the end of Dennard voltage scaling means that computer engineers must focus on using less valuable chip area to improve computational efficiency. Taken as a whole, the research comparing different kinds of heterogeneous processors suggests that while ASICs have a considerable performance advantage compared to FPGAs, FPGAs can be significantly more efficient than general purpose processors (and nearly as efficient as ASICs when bandwidth is a limiting factor) while retaining enough flexibility to be used for a wide variety of applications. Previous efforts to automate circuit design,

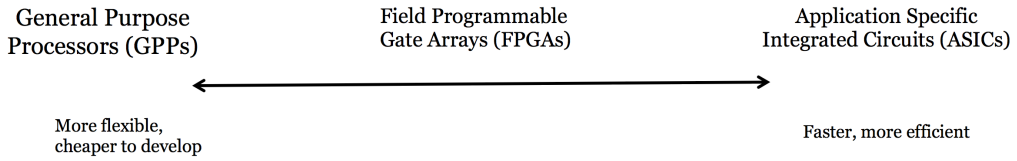

Figure 2.4: Processor Spectrum

synthesis, and analysis show that automated systems can effectively generate near-optimal circuit designs. Finally, previous systems that combine heterogeneous accelerators with

GPPs make it clear that such a design can offer significant performance and efficiency gains compared to a GPP alone. The c-core and QS-core research [23, 24] suggests that a system integrating automatically generated ASICs with a modified compiler and runtime controller can significantly reduce power consumption while remaining mostly hidden from the end user. While the patchability of c-cores does lengthen their life-cycle and allow them to adapt to small code changes, an FPGA-based solution could simply update its library of c-cores to match changes in the target code. Though the Zynq platform is relatively new and there have been few published papers discussing its use, [5, 17, 25] all present successful systems on the Zynq; moreover, they confirm that by tightly coupling its FPGA to the ARM processors and enabling PR, Zynq significantly mitigates the communication and reconfiguration overhead that are so significant for co-processors [18, 19, 28]. Overall, this previous work suggests that a c-core-like system implemented on a Zynq processor would yield significant energy efficiency gains compared to a GPP, allow for more long-term flexibility than an ASIC, and incur much less overhead than other FPGA systems.

3. The Zynq Workflow

Given the findings of the studies discussed in Chapter 2, FPGAs can be significantly more efficient than GPPs. Moreover, because of its tightly-coupled PS and PL, the Zynq seems to be an ideal platform on which to implement a reconfigurable co-processor. Before working towards a more general FPGA co-processor, we wanted to demonstrate the performance benefits of using the Zynq's PL for a sample application that we had written. In this chapter, we describe the process of creating and testing a circuit to show both the Zynq's capabilities and the difficulty of working with the platform.

3.1 Target Application: Hailstone Sequence Length Computation

The computation of Hailstone sequences, which come out of the Collatz Conjecture, is computationally intensive, but it has a relatively simple control flow making it a good target for a hardware implementation. The Hailstone sequence length computation is an interesting number-theory calculation for a variety of reasons, all of which make its computation characteristics important. Still, it is representative of a wide variety of scientific applications.

3.1.1 Hailstone Sequences

Hailstone sequences have a very simple definition as defined in Algorithm 1. As an example,

Algorithm 1 Hailstone Sequence

Require: Input n > 0.

1: **repeat**
2: **if** n is odd **then**
3: $n \leftarrow 3n + 1$
4: **else**
5: $n \leftarrow \frac{n}{2}$
6: **end if**
7: **until** n $= 1$

the Hailstone sequence for 3 is shown below.

$$3 \rightarrow 10 \rightarrow 5 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$$

Given a Hailstone sequence, we can define its length as the number of iterations required to reach 1. For example, the length of the Hailstone sequence for 3 is 7, because it takes seven iterations of the main loop to get from 3 to 1. It is not known whether all positive integers generate Hailstone sequences with finite length; the Collatz Conjecture holds that they do.

Note that when n is odd in Algorithm 1, multiplying it by 3 and adding 1 will always produce an even number. Simply dividing that result by two before returning to the top of the loop results in more efficient computation. This improvement is shown in Algorithm 2.

Algorithm 2 Hailstone Sequence V 2.0

Require: Input n > 0.

1: **repeat**
2: **if** n is odd **then**
3: $n \leftarrow \frac{3n+1}{2}$
4: **else**
5: $n \leftarrow \frac{n}{2}$
6: **end if**
7: **until** n $= 1$

3.2 Describing the Circuit

The first step towards using the Zynq PL is describing a circuit that will compute the target function. Xilinx supports the two most used hardware languages, Verilog and very high speed integrated circuit (VHSIC) Hardware Description Language (VHDL). We chose to use VHDL for this project. While VHDL code may appear similar to code from traditional software programming languages, it differs in important ways. Software programming languages such as C compile into sequential assembly language instructions that must still be interpreted by the CPU at runtime. On the other hand, compiling and synthesizing VHDL code creates a definition file for an inherently parallel hardware circuit that can be realized in an ASIC or instantiated on an FPGA.

To compare the performance of the Hailstone length computation in hardware and software, we implemented Algorithm 2 in both C and VHDL. These two implementations are shown below.

```
int hailstone(long input) {
  int count = 0;
  long long cur = input;
```

```
  while(1) {
    if (cur % 2L) {       //"L" indicates a long constant
      if (cur == 1L) {
        return count;
      } else {
        cur = ((cur * 3L) + 1L) / 2L;
        count += 2;
      }
    } else {
      cur = cur / 2L;
      count++;
    }
  }
}
```

Listing 3.1: C Implementation

```
if val1(0) = '1' then
  --number is odd
  if val1 = 1 then
    iter_count   <= std_logic_vector(count);
    state   <= done;
    output_ready <= '1';
    output_ready_internal <= '1';
  else
    val2  := shift_left(val1, 1) + 1;
    val2  := val2 + val1;
    val1  := shift_right(val2, 1);
    count := count + 2;
    state <= computing;
  end if;
else
  --number is even
  val1  := shift_right(val1, 1);
  count := count + 1;
  state <= computing;
end if;
```

Listing 3.2: VHDL Implementation

These two implementations look similar, but there are two important differences. First, the VHDL code utilizes state variables and synchronization signals to coordinate the computation. Second, while the C compiles into assembly code that must be interpreted, but once the VHDL has been compiled and flashed to the Zynq's PL, there are no more instructions or interpretation, simply wires, gates, and other hardware structures that execute the computation.

3.3 Using the Xilinx Vivado Toolchain

3.3.1 Simulation

Having written a VHDL file that describes the Hailstone length computation circuit, we next used a simulator to ensure that its operation is logically correct. Testing a VHDL circuit is significantly more complex than testing a software application, as a VHDL testbench must mock a hardware environment attached to the circuit being tested. Instead of simply calling a function with a few different test inputs, a programmer must define and provide a clocking mechanism and interfaces between the circuit being tested and the testing unit. Xilinx's Vivado software suite includes a simulator, but the simulator provided in our version was unreliable and frequently crashed, so we used an older Xilinx simulator from 2011. One final issue with hardware simulation is that while it can mock the logic of a circuit, it cannot test how the logic will be implemented in hardware, or even whether it can be implemented at all. For example, one can simulate a VHDL circuit with a loop whose exit condition is parametrized at runtime (e.g. $i < n$ where n is an input parameter), but such a loop cannot be directly synthesized.

3.3.2 Creating and Synthesizing Circuit IP

The next step is creating a new "intellectual property" (IP) block, Xilinx's term for the individual hardware blocks that can be implemented on their FPGAs. Vivado provides an IP creation tool that helps with this work by automating some of the AXI—the interconnects which handle communication between the PS and PL—parameters. Even so, the process is quite involved, as one must import code, instantiate it in a poorly documented AXI template, and then package the IP core.

To use the newly created IP core, it must then be integrated into a block diagram, which effectively lays out a plan for the entire Zynq system. To do so, we have to import some Xilinx IP (such as the IP for the ARM PS) and connect the different pieces of IP correctly. The result is a block diagram like that shown in Figure 3.1.

Once the block diagram is complete and all interconnects have been managed correctly, it can be synthesized and implemented. Unlike software compilation, which usually takes at most a few seconds, Vivado's synthesis and implementation generally take between 3 and 5 minutes to run. This means that testing and updating an IP core is extremely time consuming, as even minor changes require a significant amount of time to implement.

Synthesis and implementation output many reports, some of which detail the resource utilization of the IP cores. The Hailstone length computation circuit uses only a small fraction of the PL's resources, less than 4% of its look-up tables (LUTs) and 1% of its

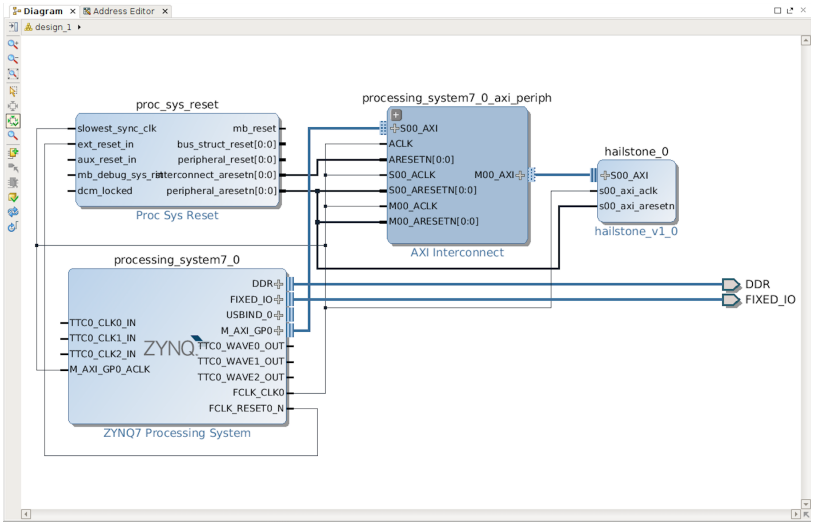

Figure 3.1: Vivado block diagram for the Hailstone circuit

registers. [1]

3.3.3 Testing in the Xilinx SDK

The result of implementation is a netlist that Vivado can translate into a bitstream. To use this bitstream, and test the circuit, we use the Xilinx SDK. With the Zynq properly connected to a computer (it requires two USB cables and the appropriate placement of a few jumpers), we can write the bitstream to the PL. At this point, our circuit is in place on the Zynq's PL, waiting for input.

Testing is also more complicated than with a normal software application. The circuit cannot simply be called like a normal C function. Instead, one must use either flags or interrupts to communicate between the PS and PL. In this implementation, we used a custom scheme to implement communication between PS and PL with a set of flags contained in some of the AXI registers. When the PS wants to compute the Hailstone sequence length of an integer, it writes the input value to the `INPUT` register and sets the `INPUT_READY` register. The circuit reads that `INPUT_READY` is high, consumes the input, and raises its `INPUT_CONSUMED` flag. When it has finished computing the Hailstone sequence length, the circuit writes the result to the `OUTPUT` register and sets its `OUTPUT_READY` flag high. Once the PS reads that `OUTPUT_READY` is high, it reads the output and resets `INPUT_READY`; the circuit then resets the `OUTPUT_READY` flag. At this point, the PS can initiate a new computation.

Obviously this interface is much more difficult to implement than a simple function call,

[1] Even that estimate may be high. We believe much of the circuitry is one-time overhead that is computation-independent.

but it is necessary to synchronize the computation between the independently clocked PS and PL. Thus, while the main goal of the Hailstone test was to compare the performance of software and hardware implementations of the Hailstone sequence length computation, a secondary target was evaluating the speed of the Zynq's interconnects between the ARM processors and the PL. There is overhead involved in synchronizing and communicating between those two chip components, and a useful co-processor design must yield performance improvement despite that overhead.

3.4 Summary

To compare the software and hardware implementations, we timed how long it took each to compute the Hailstone sequence length for each of the first n integers, with n ranging from 1,000 to 10,000,000. To test the hardware implementation we used the communication scheme described above in Subsection 3.3.3 to iteratively supply each input value, wait for the circuit to signal that it had finished, and then read the circuit's output. To test the software implementation, we called the `hailstone(long input)` function. For each iteration count, we timed both implementations three times and averaged the results.

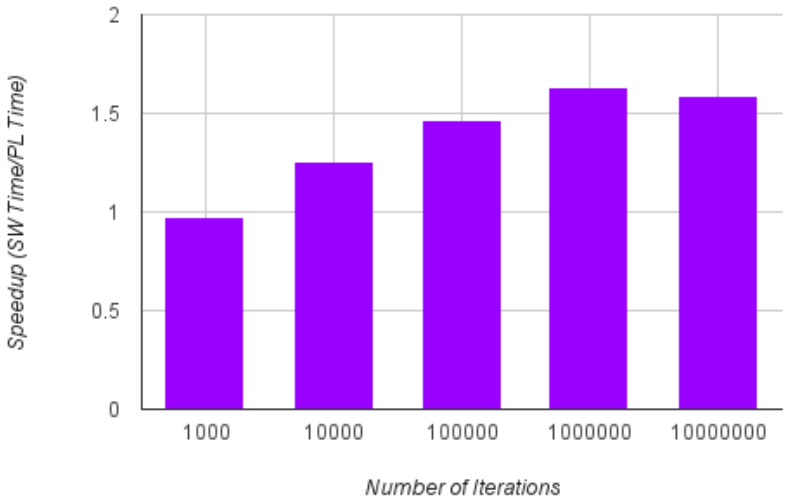

Figure 3.2: Speedup of hardware Hailstone implementation over software

Despite the overhead involved in communicating between the PS and PL, and the inherent lack of parallelism in this computation, the hardware implementation was significantly more efficient. As shown in Figure 3.2, the hardware implementation was faster for all trials

of more than 1000 iterations; for the longer-running trials it was roughly 50% faster than the software implementation.

This is an extremely promising result, as it demonstrates that hardware computation can be faster and more efficient than software even for straight-line code that does not exploit any parallelism. Further, as discussed in Subsection 3.3.2, the synthesized circuit for this computation required an almost negligible amount of the PL's resources, around 4% of its LUTs, suggesting that even a more parallel (and efficient) solution would easily fit into a small corner of the PL fabric. While we have no direct method of evaluating the energy consumption of the calculation, we expect those improvements are even more significant.

4. Reconfigurable Datapath Design

In this section, we discuss first the concept and benefits of an abstracted FPGA-based co-processor and then the reconfigurable datapath that we have designed as a first step towards that goal.

4.1 Concept and Goals

The results of the Hailstone sequence length computation discussed in Chapter 3 and the research discussed in Chapter 2 show that an FPGA-based co-processor can provide significant performance improvement over a GPP alone. However, FPGA flexibility and efficiency comes at a significant workflow-cost, as they are extremely difficult to work with. The chips themselves are very complex (see, for example, Xilinx's 1863 page manual for the Zynq [27]); the learning curves for VHDL and Verilog are steep; and the software supporting FPGAs is clumsy, expensive, industry-specific, slow, and poorly documented. Indeed, a large portion of current FPGA research, such as Lavin et al.'s RapidSmith focuses on simplifying the workflow for computing on an FPGA [13]. As a result, while some hardware and system designers may appreciate the flexibility that an FPGA provides, average users need a higher level of abstraction. They could benefit from some of an FPGA's capabilities without being exposed to the full intricacy of the hardware itself. Effectively, we need to make the FPGA's fabric coarser, giving the user access to flexible, easily managed modules that can be placed on the FPGA fabric. As a first step, we have created a reconfigurable datapath that encapsulates what is effectively a microcode controlled vector processor managed at runtime in C.

4.2 Circuit Design and Implementation

The basic model for the reconfigurable datapath is a state machine that performs computation on two (or three, for multiply-add) elements on every clock cycle. Using this model, a diagram of which is shown below in Figure 4.1, makes it easy for the user to succinctly supply all of the necessary state for a given computation and for the circuit to keep track of all necessary state until the computation has been completed. Comparison of a user-defined iteration limit and an internal iteration count determines when a computation is complete. In the rest of this section we describe the main components of the state machine and its operation: data transfer, data storage, array pointers and increments, and available operations.

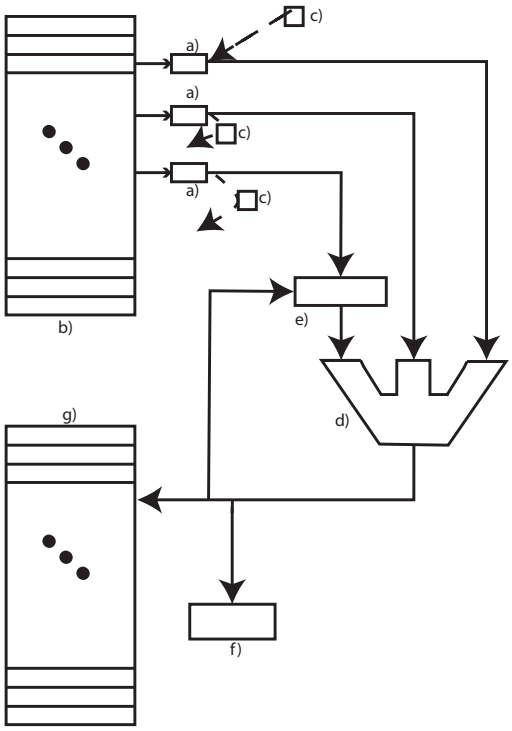

Figure 4.1: Logical diagram of the reconfigurable datapath. On each iteration, three vector offsets (a) index elements from the tri-ported RAM (b); each pointer is then incremented by its vector stride (c). The ALU (d) operates on those elements and the result is stored in the output RAM (g) and added to the accumulator (f). Execution continues until the iteration count (e) reaches the user-provided limit.

4.2.1 Data Transfer

For this project, we used the Zynq's AXI4-Lite interface to implement data transfer between the PS and the datapath on the PL. The implementation uses some of shared AXI registers for data transfer and others as a system of flags similar to that of the Hailstone circuit described in Section 3.2. When the PS is ready to transfer input data to the PL, it raises an `INPUT_READY` flag, loads the first value into the transfer register, and increments a `PS_IO_INDEX` register. On the next clock cycle, the circuit reads that the processor `INPUT_READY` flag is set high and that the value of the `PS_IO_INDEX` is greater than the `PL_IO_INDEX` (initialized to zero). The circuit will therefore read in the value from the transfer register, write that value to a block RAM on the PL, and increment the `PL_IO_INDEX`. When the processor sees that the `PL_IO_INDEX` has been incremented and is now equal to the `PS_IO_INDEX`, it can put a new value into the transfer register and increment the `PS_IO_INDEX` to transfer the next data element. This process continues until the processor has supplied all of its input, at which point it will lower the `INPUT_READY` flag and reset the `PS_IO_INDEX`. This mechanism allows us to synchronize computation between the independently clocked PS and PL.

After computation has completed, a similar operation is used to transfer output from the PL to the PS, with the circuit placing output values in the data transfer register and the processor reading those values until all of the output has been consumed.

4.2.2 Data Storage

On the PL, input data is stored in a tri-ported memory that make use of the Zynq's dedicated block RAM (BRAM) [27]. Using the dedicated "hard" RAM blocks as opposed to distributed RAM implemented on the fabric's LUTs results in significantly better performance, efficiency, and space utilization as discussed in [12]. The input is stored in tri-ported RAM so that during operation, three input values can be read and used on every clock cycle. Output data is stored in a separate BRAM. One value is written to this BRAM on each datapath cycle during computation. After computation is complete, one value is read from this BRAM and placed into the data transfer register on each cycle.

4.2.3 Vector Offsets and Strides

The user supplies three integer vector offsets (A, B, and C) and a stride length for each. On each cycle of the datapath, those offsets are used as indices into the tri-ported RAMs to read the operands for that cycle. Each offset is then incremented by its user-specified vector stride so that the next value can be supplied for the following cycle. This setup allows both standard array traversal and constant operands (by setting the stride length to 0).

4.2.4 Operations

On each datapath cycle, up to three operands are read from the tri-ported RAM, the user-specified operation is performed, and the result is written to the output RAM and added to an accumulator. There are five available operations which the user can choose: add, subtract, multiply, multiply-add, and "black box." The first four of those operations are performed on the Zynq's dedicated DSP48 ALUs. As with the BRAMs, using the dedicated hardware is more efficient than ALUs implemented on the general FPGA fabric.

The "black box" option allows the user to supply a pre-compiled and synthesized circuit that computes some function of up to three operands. For example, the Hailstone sequence length circuit discussed in Section 3.1 could be supplied as the black box function, and each datapath cycle would then compute the length of the Hailstone sequence for the value at vector offset A in the input RAM. This feature lets the user piggyback his computation on our infrastructure. Most people likely will not want or be able to synthesize their own circuits, but for those that do, this feature allows them to easily use those circuits without also designing and implementing an interface between the PS and their circuit.

4.3 User Interface

In the following sections, we will give a basic outline of the C interface for the datapath. We will then show how one can compute vector magnitude using that interface.

4.3.1 C Interface

The user interface for the datapath is a simple C struct, `alu_computation`. The basic parts of the interface are shown below in listing 4.1. `baseaddr_p` is a pointer to the base AXI4 register that the processor and circuit share. That is, `baseaddr_p` points to the lowest register value through which the processor communicates with the datapath. `arr_pointer` is a pointer to the PS input vector. `arr_size` is the number of input elements. `a_base_index, b_base_index, c_base_index` and `a_inc, b_inc, c_inc` are the three vector offsets and their vector strides respectively. `operation_code` is a function code indicating the user's choice of operation. `limit` defines how many iterations the datapath should perform - the logical vector length. The result of computation is stored in an `alu_computation_result` struct that contains both the total accumulated value of the computation and a pointer to the result array.

```
typedef struct alu_computation alu_computation;
typedef struct alu_computation_result alu_computation_result;

struct alu_computation {
```

```
    uint32_t *baseaddr_p;           //shared register base address
    int *arr_pointer;               //input vector base address
    int arr_size;                   //input vector size
    int a_base_index, b_base_index, c_base_index;
                                    //initial vector offsets
    int a_inc, b_inc, c_inc;        //vector strides
    int operation_code;             //function code
    int limit;                      //iteration limit
};

struct alu_computation_result {
    double accumulated_value;       //final accumulator value
    int *result_array;              //pointer to result array
};

//create a new alu_computation struct
alu_computation *create_alu_computation();

//execute the computation defined by c and return the result struct
alu_computation_result *alu_compute(alu_computation * c);
```

Listing 4.1: C interface header file for reconfigurable datapath

To perform a computation, the user simply creates an `alu_computation` struct by calling `create_alu_computation`, fills in the struct members, and then calls `alu_compute`, passing in the pointer to the filled-in struct. `alu_compute` transfers the data to the PL, sets up the computation, retrieves the results, and places them in an `alu_computation_result` struct.

4.3.2 Example Usage: Vector Magnitude

The code in Listing 4.2 shows how the datapath can be used to easily calculate the magnitude of the vector containing the digits 0 through 9. The user first creates and fills in her input array in lines 1-6. She then creates the `alu_computation` and fills in its members appropriately (`operation_code` 2 indicates multiplication). The `a_base_index` and `b_base_index` vector offsets are both initialized to 0 and their strides to 1 because computing vector magnitude involves walking through the vector one element at a time and multiplying each element by itself. She then simply calls `alu_compute` to get her `alu_computation_result`. The result's `accumulated_value` contains the sum of the squares of each element, so the user simply computes the square root of that value to get the vector's magnitude.

```
1 int arr_len = 10;
2 int arr [arr_len];
3 int i = 0;
4 for(; i < arr_len; i++) {
5   arr[i] = i;
6 }
7 alu_computation *c = create_alu_computation();
8 c->baseaddr_p = baseaddr_p;
9 c->arr_pointer = arr;
```

```
10 c->arr_size = arr_len;
11 c->a_base_index = 0;
12 c->b_base_index = 0;
13 c->c_base_index = 0;
14 c->a_inc = 1;
15 c->b_inc = 1;
16 c->c_inc = 0;
17 c->operation_code = 2;
18 c->limit = arr_len;
19
20 alu_computation_result * res = alu_compute(c);
21
22 printf("Vector magnitude: %f\n", sqrt(res->accumulated_value));
```

Listing 4.2: Example usage of C struct to calculate vector magnitude

4.4 Summary

The runtime reconfigurable datapath described in this section represents a first step towards a larger scale FPGA-based co-processor. The "black box" feature in particular provides a prototype for a future system in which a central FPGA hub handles communication between the processor and several hardware circuits; those circuits could even be a collection of c-cores, creating a system like that described in Section 2.5.

Perhaps most important, the FPGA's reconfigurability makes it possible to modify, improve, or even discard this design without any significant financial or technical ramifications: unlike an ASIC, an FPGA does not lock its users into the hardware design that comes out of the fabrication plant.

5. Reconfigurable Datapath Performance

The current implementation of the datapath as described in the previous chapter is merely a proof-of-concept. Knowing this, we decided not to optimize its performance, and instead focused on ensuring that we had created a solid foundation and clear road-map for future work—many possible improvements are outlined in Chapter 6, which follows. Nevertheless, we wanted to test the performance of the datapath as currently implemented to confirm that with fine-tuning and optimization, we can provide simple user interfaces for efficient, partnered computation on a hybrid GPP-FPGA platform.

We chose vector dot product as our test computation for two main reasons. First, it is a widely used and understood computation that is therefore relevant to many different applications. More importantly, dot product's simplicity makes interpretation of the datapath's performance straightforward. In its current implementation, the datapath does not exploit any parallelism by, for example, computing the product for multiple indices concurrently, so its performance reflects only the relative efficiency of computing in hardware as opposed to software.

To test performance on vector dot product, we computed the dot product of integer vectors with 1000 elements. We computed 100 dot products per trial and then took the average time per dot product operation (total time divided by 100). We ran 10 trials for each implementation and selected the fastest times from those 10. For the hardware implementation, we ran two sets of trials: one that timed the entire computation, including data transfer, and one that included only the time required to compute the dot product. The latter was measured as the time between the PS raising its `COMPUTE_READY` flag and the circuit raising its `OUTPUT_READY` flag.

The results are shown below in Table 5.1. While the entire datapath computation, including data transfer, was orders of magnitude (roughly 25×) slower than the software implementation, the datapath computation was over 33% faster than the software implementation. This is an extremely promising result. Clearly, data transfer is always going to

Table 5.1: Vector Product Results

	Software	Datapath Total	Datapath Computation Only
Time Per Dot Product	0.054 ms	1.36 ms	0.039 ms

outweigh the dot product computation itself given that each element is only operated on once, though that overhead can be mitigated by techniques discussed in Subsection 6.1.1. That the datapath is faster than software implies that even for generic, serial applications, a hardware circuit should be more efficient than software code. Indeed, the increased performance must come from the elimination of the instruction fetch and decode overhead discussed in Section 2.2: by eliminating the 90% of GPP computation that is almost pure overhead, even a sub-optimal hardware solution can be more efficient than a software implementation. Just as the c-cores researchers suggested, specialized circuitry and hardware computation are efficient even when they do not exploit parallelism.

6. Future Work

The reconfigurable datapath described in this thesis represents just a first step towards a complete FPGA-based co-processor system. In this section, we will discuss some of the ways we can move closer to that goal, first with enhancements to the reconfigurable datapath as currently implemented and then by integrating some of the previously studied techniques discussed in Chapter 2. Because the datapath has a very small footprint on the PL, as shown in Figure 6.1 and Table 6.1, even improvements that would consume significantly more PL resources are quite feasible.

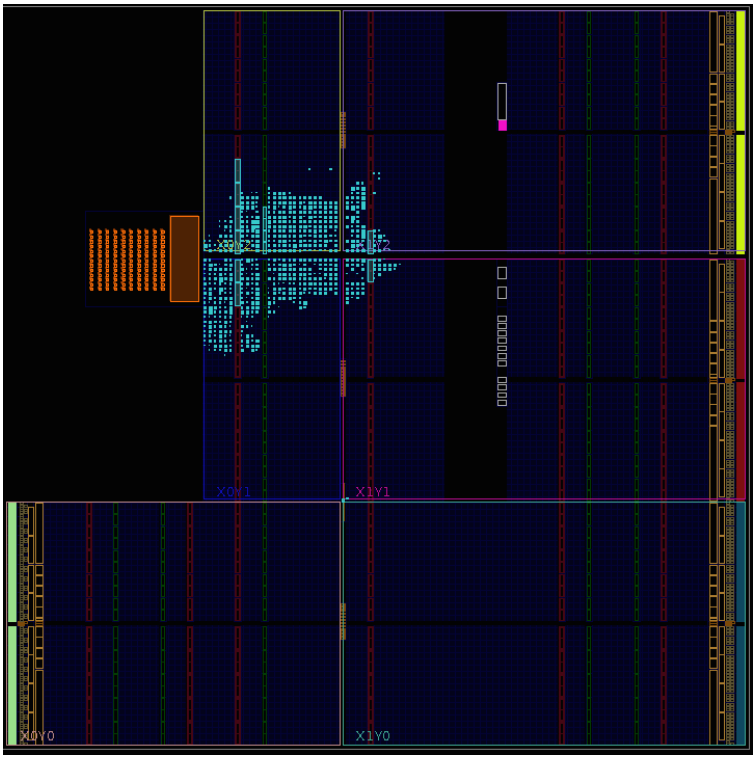

Figure 6.1: Reconfigurable Datapath Footprint on the Zynq PL. Leaf nodes used by the datapath are highlighted in blue.

Table 6.1: Reconfigurable Datapath Resource Utilization

Resource	LUTs	Registers	BRAM	DSP ALUs
Utilization	4%	2%	6%	2%

6.1 Upgrades to the Reconfigurable Datapath

6.1.1 Improving Data Transfer

Improving the data transfer technique used by the reconfigurable datapath would significantly decrease one of its primary sources of overhead. Currently, data transfer works as described in Subsection 4.2.1: data values are read in by the PL one at a time from a data transfer register. While this overhead may be negligible for long-running computations, for shorter computations, or computations which use more data than can be stored in the datapath at one time, a more diverse set of data transfer methods would be beneficial. In this area, there are a few main options that could be implemented, possibly in tandem.

- Pipelining the data transfer would allow computation to begin before all of the data had been read in. Pipelining would mitigate the overhead of data transfer by ensuring that after the first set of data had been read in, useful work (and not just data transfer) would be done during all future datapath cycles. Unfortunately, pipelining is not simply a matter of starting computation on the second datapath cycle, since doing so might result in invalid memory reads if, for example, one of the vector offsets were pointing to an index that had not yet been read in.

- The Zynq SoC includes several types of AXI interconnects between the PS and PL, each of which has its own set of advantages and disadvantages. Integrating some of these alternative transfer methods might be beneficial in certain circumstances. For example, the AXI Streaming FIFO would reduce overhead for large transfers. In certain circumstances, direct memory access might render transferring input data unnecessary. Adding these (and other) capabilities to the datapath would not be simple, however, as each is only useful for particular applications. Moreover, managing all of the different possible input streams would be quite complex, and would add significant space overhead to the datapath. I/O automation and appropriate I/O abstractions are very important areas of study.

6.1.2 Computation Pipelining and Multiple ALUs

Pipelining computation and introducing multiple ALUs would reduce the computation time at the cost of increased resource usage. One of the major advantages that ASICs and FPGAs

have compared to GPPs is their ability to exploit data parallelism, but the datapath, as currently implemented, does not do so. More PL area or BRAMs would be required for data storage (since more RAM reads would happen on every cycle) and more of the DSP48 units would be taken up. This, in turn, limits the resources available for both black box computations and other circuits on the PL.

6.1.3 Broadening the Capabilities

One more way to make the datapath more capable would be to increase its scope. This could take a few different forms, a couple of which are detailed below.

- Adding support for data elements of other sizes and types (as opposed to just 32-bit integers) would allow the datapath to be used for a much broader set of computation-intensive tasks, such as computing edit distance between strings. As with the other improvements, this would be quite complex, requiring many different run-time options and a much more robust black box integration.

- Transforming the datapath into a communication hub responsible for transferring data and signals between the PS and many heterogeneous circuits on the PL would move us further towards a c-core like system. Indeed, with a truly functional communication hub between the PS and PL-based circuits, one could even implement the c-cores themselves on the PL and use them to efficiently handle the segments of code that they were designed to compute.

6.2 Integration with a Compiler and Kernel

For an FPGA-based co-processor to gain widespread use, it must be integrated into both compilers and the system kernel so that programmers do not have to explicitly invoke its use. Indeed, while an enhanced version of our datapath might represent a useful tool for efficiency-focused developers who were aware of it, only automatic, implied invocation from the compiler or runtime machine will lead to significant efficiency improvements across most applications. As a simple example, Python's list comprehensions are an ideal target for a co-processor capable of efficiently mapping computations across a vector of input data, but we cannot expect every Python developer to explicitly invoke a co-processor. Indeed, doing so would create code that was not portable, as systems without the co-processor would be unable to execute the modified code. Instead, the Python interpreter should be modified to invoke the co-processor if it is available on the interpreter's host system. Previous research in this area [4, 18, 23] suggests that even relatively complex decisions, such as whether or

not the co-processor should be invoked for a specific instance of a computation that it can handle, can be made at runtime to further maximize efficiency.

Integration with a kernel would also improve co-processor I/O by allowing the co-processor to use memory mapping and interact with the general purpose processor through a device driver. Though the Zynq platform is relatively new, there are already a few device driver IP cores that help manage the interface between a Linux operating system on the PS and user logic on the PL [1].

6.3 Partial Reconfiguration

Partial reconfiguration refers to shutting down and reconfiguring only a portion of an FPGA while the remainder stays active and running. In general, this is obviously preferable to shutting down the entire FPGA if not all of it must be reconfigured, though it does introduce the difficulty of managing reconfigurable regions and some waste of FPGA resources. Moreover, for PR to be truly efficient, circuits that could occupy the same region must have similar resource requirements and interconnect configurations.

PR would allow an FPGA-based co-processor to dynamically swap in circuits designed to efficiently handle whatever code was currently running. For example, if someone opened an application on her phone, and that application had an associated circuit designed to make it run more efficiently, the co-processor could instantiate that circuit on its programmable logic without shutting down completely and interrupting other ongoing computations such as cell signal processing. The research of Vipin and Fahmy indicates that relatively efficient PR can be achieved on the Zynq [25].

6.4 Automated Circuit Design and Synthesis

Section 2.3 reviewed previous research on automated circuit design, synthesis, and analysis. Integrating that automation into a c-core like system would be hugely beneficial. Circuits for frequently used code could be automatically generated, installed into a circuit library, and then controlled by a runtime partial reconfiguration manager. Periodically, the processor could evaluate the performance of those circuits, improve their designs, and recompile them into the library. This sort of evolutionary algorithm would, over time, tailor each co-processor to its users without any explicit input from the users themselves. Further, it would mean that all applications, and not only those whose developers had designed an accompanying accelerator, might benefit from dedicated circuits on the co-processor.

6.5 Future Concerns

While an FPGA-based co-processor could greatly improve computational efficiency and help fight against the trend of dark silicon, it would also introduce some new concerns. Most prominent among these is hardware security. Software viruses, malware, and exploits are a constant worry, and a hardware virus or maliciously designed circuit could be equally harmful. Without the proper protections in place, malicious developers could design circuits to interrupt or steal information from other circuits on the FPGA, or to render the co-processor wholly unusable by attacking communication interconnects. Guarding against such attacks requires a new set of security checks that can identify harmful circuit designs and enforce boundaries between circuits which are not designed to co-operate.

7. Conclusion

Using reconfigurable hardware to build a partnered computation processor is the most promising way to break through stagnating performance in the face of dark silicon.

The datapath we implemented clearly is not capable of yielding tangible benefits for real-world applications, but our results confirm what earlier research suggested: general purpose processors waste too much of their execution on computational overhead. Indeed, both experiments discussed in this work evaluated the performance of using hardware circuits to execute serial algorithms—we did not even exploit any of the inherent parallelism in computing vector dot product—and in both cases the hardware computation was faster and more efficient than a software implementation. Given that the individual hardware components on the Zynq PL (the ALUs, BRAMs, clocks, etc.) are all likely slower than their ARM counterparts in the PS, the efficiency gains must have come from eliminating the overhead of instruction fetch, decode, and interpretation. As the c-cores researchers stated, specialized circuitry can improve computational efficiency even in the absence of parallelism, and almost all of the code that a CPU executes could be executed more efficiently by an appropriate hardware circuit.

Our datapath and the future work discussed in Chapter 6 provide a proof-of-concept and roadmap for a general purpose, partnered computation processor. Following that vision by coupling software and hardware computation, we can increase efficiency and performance in the era of dark silicon.

Acronyms

ALU arithmetic logic units (ALUs) are fundamental processor components that perform arithmetic or bitwise logical functions on binary integer inputs.

ARM advanced RISC machine (ARM) is a set of RISC architectures developed by ARM Holdings. ARM is currently the dominant RISC architecture, and ARM processors are used in most mobile devices.

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

AXI Advanced eXtensible Interface (AXI) is a set of micro-controller buses used in the Zynq SoC to handle data transfer between the PS and IP cores on the PL.

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

CISC complex instruction set computing (CISC) is a CPU design strategy in which each instruction can execute multiple low-level operations. Intel's x86 is the most common modern CISC architecture.

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

EPIC explicitly parallel instruction computing (EPIC) is a hybrid between RISC and VLIW architectures in which the compiler controls parallel instruction execution by executing multiple software instructions in parallel. In VLIW systems, CPU hardware has to manage the scheduling of parallel execution.

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

GPGPU general purpose computing on graphics processing units (GPGPU) refers to using a GPU to execute computation normally performed by the CPU. GPGPU can have better performance than computation on a CPU for applications with large amounts of data parallelism, such as vector processing.

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

GPU graphics processor units (GPUs) are integrated circuits designed to efficiently process images for output to a display. GPUs exploit data parallelism to achieve much higher performance than CPUs for typical image processing algorithms.

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.

MIPS microprocessor without interlocked pipeline stages (MIPS) is a RISC instruction set first introduced by MIPS Technologies in 1981.

PL programmable logic (PL) refers to the FPGA portion of the Zynq SoC.

PR partial reconfiguration (PR) refers to shutting down and reconfiguring only a portion of an FPGA while the remainder stays powered and running.

PS processing system (PS) refers to the ARM GPPs in the Zynq SoC.

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

RISC reduced instruction set computing (RISC) is a CPU design strategy that uses smaller set of optimized instructions so that instructions can be executed in fewer clock cycles. ARM and MIPS are the most prominent RISC architectures.

SoC system on chip (SoC) integrated circuits combine all the components of a computer onto a single chip. SoCs consume less power than traditional computers and are therefore used in mobile devices and embedded systems.

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.

VLIW very long instruction word (VLIW) processor architectures allow programmers to specify instructions to execute in parallel.

Bibliography

[1] Xillybus: An fpga ip core for easy dma over pcie with windows and linux. URL `http://xillybus.com/`.

[2] S. Aditya, B.R. Rau, and V. Kathail. Automatic architectural synthesis of VLIW and EPIC processors. In *System Synthesis, 1999. Proceedings. 12th International Symposium on*, pages 107–113, Nov 1999. doi: 10.1109/ISSS.1999.814268.

[3] Eric S. Chung, Peter A. Milder, James C. Hoe, and Ken Mai. Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPGPUs? In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 225–236, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4299-7. doi: 10.1109/MICRO.2010.36. URL `http://dx.doi.org/10.1109/MICRO.2010.36`.

[4] Nathan Clark, Jason Blome, Michael Chu, Scott Mahlke, Stuart Biles, and Krisztian Flautner. An architecture framework for transparent instruction set customization in embedded processors. *SIGARCH Comput. Archit. News*, 33(2): 272–283, May 2005. ISSN 0163-5964. doi: 10.1145/1080695.1069993. URL `http://doi.acm.org/10.1145/1080695.1069993`.

[5] R. Dobai and L. Sekanina. Towards evolvable systems based on the Xilinx Zynq platform. In *Evolvable Systems (ICES), 2013 IEEE International Conference on*, pages 89–95, April 2013. doi: 10.1109/ICES.2013.6613287.

[6] S.H. Fuller and L.I. Millett. Computing performance: Game over or next level? *Computer*, 44(1):31–38, Jan 2011. ISSN 0018-9162. doi: 10.1109/MC.2011.15.

[7] N. Goulding-Hotta, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, P. Huang, M. Arora, S. Nath, V. Bhatt, J. Babb, S. Swanson, and M. Taylor. The GreenDroid mobile application processor: An architecture for silicon's dark future. *Micro, IEEE*, 31(2):86–95, March 2011. ISSN 0272-1732. doi: 10.1109/MM.2011.18.

[8] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. Understanding sources of inefficiency in general-purpose chips. *SIGARCH Comput. Archit. News*, 38(3):37–47, June 2010. ISSN 0163-5964. doi: 10.1145/1816038.1815968. URL `http://doi.acm.org/10.1145/1816038.1815968`.

[9] John R Hauser and John Wawrzynek. Garp: A MIPS processor with a reconfigurable coprocessor. In *Field-Programmable Custom Computing Machines, 1997. Proceedings., The 5th Annual IEEE Symposium on*, pages 12–21. IEEE, 1997.

[10] Mark D Hill and Michael R Marty. Amdahl's Law in the multicore era. *IEEE Computer*, 41(7):33–38, 2008.

[11] R. Kumar and A Gordon-Ross. PRML: A modeling language for rapid design exploration of partially reconfigurable FPGAs. In *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*, pages 117–120, April 2013. doi: 10.1109/FCCM.2013.24.

[12] Ian Kuon and Jonathan Rose. Measuring the gap between FPGAs and ASICs. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(2):203–215, 2007.

[13] C. Lavin, M. Padilla, P. Lundrigan, B. Nelson, and B. Hutchings. Rapid prototyping tools for FPGA designs: RapidSmith. In *Field-Programmable Technology (FPT), 2010 International Conference on*, pages 353–356, Dec 2010. doi: 10.1109/FPT.2010.5681429.

[14] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings. Hmflow: Accelerating FPGA compilation with hard macros for rapid prototyping. In *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, pages 117–124, May 2011. doi: 10.1109/FCCM.2011.17.

[15] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings. Rapidsmith: Do-it-yourself CAD tools for Xilinx FPGAs. In *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, pages 349–355, Sept 2011. doi: 10.1109/FPL.2011.69.

[16] Christopher Lavin, Marc Padilla, Subhrashankha Ghosh, Brent Nelson, Brad Hutchings, and Michael Wirthlin. Using hard macros to reduce FPGA compilation time. In *Proceedings of the 2010 International Conference on Field Programmable Logic and Applications*, FPL '10, pages 438–441, Washington, DC, USA, 2010. IEEE

Computer Society. ISBN 978-0-7695-4179-2. doi: 10.1109/FPL.2010.90. URL http://dx.doi.org/10.1109/FPL.2010.90.

[17] Jonathon Pendlum, Miriam Leeser, and Kaushik Chowdhury. Reducing processing latency with a heterogeneous FPGA-processor framework. In *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, pages 17–20, May 2014. doi: 10.1109/FCCM.2014.13.

[18] Heather Quinn, Miriam Leeser, and Laurie Smith King. Dynamo: a runtime partitioning system for FPGA-based HW/SW image processing systems. *Journal of Real-Time Image Processing*, 2(4):179–190, 2007.

[19] Rahul Razdan and Michael D. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, MICRO 27, pages 172–180, New York, NY, USA, 1994. ACM. ISBN 0-89791-707-3. doi: 10.1145/192724.192749. URL http://doi.acm.org/10.1145/192724.192749.

[20] Yaska Sankar and Jonathan Rose. Trading quality for compile time: Ultra-fast placement for FPGAs. In *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays*, FPGA '99, pages 157–166, New York, NY, USA, 1999. ACM. ISBN 1-58113-088-0. doi: 10.1145/296399.296449. URL http://doi.acm.org/10.1145/296399.296449.

[21] M.B. Taylor. Is dark silicon useful? Harnessing the four horsemen of the coming dark silicon apocalypse. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 1131–1136, June 2012.

[22] Michael Taylor. A landscape of the new dark silicon design regime. *IEEE Micro*, 33(5):8–19, September 2013. ISSN 0272-1732. doi: 10.1109/MM.2013.90. URL http://dx.doi.org/10.1109/MM.2013.90.

[23] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation Cores: Reducing the energy of mature computations. *SIGARCH Comput. Archit. News*, 38(1):205–218, March 2010. ISSN 0163-5964. doi: 10.1145/1735970.1736044. URL http://doi.acm.org/10.1145/1735970.1736044.

[24] Ganesh Venkatesh, Jack Sampson, Nathan Goulding-Hotta, Sravanthi Kota Venkata, Michael Bedford Taylor, and Steven Swanson. QsCores: Trading dark silicon for

scalable energy efficiency with quasi-specific cores. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 163–174, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1053-6. doi: 10.1145/2155620.2155640. URL `http://doi.acm.org/10.1145/2155620.2155640`.

[25] K. Vipin and S.A Fahmy. Zycap: Efficient partial reconfiguration management on the Xilinx Zynq. *Embedded Systems Letters, IEEE*, 6(3):41–44, Sept 2014. ISSN 1943-0663. doi: 10.1109/LES.2014.2314390.

[26] Kizheppatt Vipin and Suhaib A Fahmy. Automated partial reconfiguration design for adaptive systems with CoPR for Zynq. In *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, pages 202–205, May 2014. doi: 10.1109/FCCM.2014.63.

[27] UG585. *Zynq-7000 All Programmable SoC*. Xilinx, v1.8.1 edition, Sept 2014.

[28] Zhi Alex Ye, Andreas Moshovos, Scott Hauck, and Prithviraj Banerjee. CHIMAERA: A high-performance architecture with a tightly-coupled reconfigurable functional unit. *SIGARCH Comput. Archit. News*, 28(2):225–235, May 2000. ISSN 0163-5964. doi: 10.1145/342001.339687. URL `http://doi.acm.org/10.1145/342001.339687`.