Programmable, Energy-minimal Computer Architectures

Graham Gobieski

CMU-CS-22-145

August 2022

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Nathan Beckmann, Co-Chair
Brandon Lucia, Co-Chair
Todd Mowry
Kenneth Mai
Tony Nowatzki, External

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored by an Apple PhD Fellowship, the National Science Foundation under award numbers CNS-1526342 and CCF-1815882, and the U.S. Army under award number W911NF1820218. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Abstract

Ultra-low-power (ULP) sensor devices are increasingly being deployed for a variety of use-cases that require sophisticated processing of sensed data. Regardless of the deployment, energy-efficiency is critical; for battery-powered devices, energy-efficiency determines device lifetime, while for energy-harvesting devices, energy-efficiency determines performance by dictating the frequency of recharging. Unfortunately, existing devices pay a severe energy tax for their programmability, wasting energy in instruction-fetch/decode, pipeline-control and data supply. Further, offloading computation from an edge-device to the cloud is not practical as communication costs an order-of-magnitude more energy than local compute. The solution is to redesign the ULP sensor system stack to increase the energy-efficiency of on-board compute and enable sophisticated processing of sensed data. This thesis proposes such a stack — from software to silicon — that leverages new execution models to reduce the tax of programmability and achieve extreme energy-efficiency. Specifically it contributes 1) Sonic, a software framework that enables machine inference on intermittently-operating, energy-harvesting devices, 2) Manic, a vector-dataflow co-processor (and corresponding silicon prototype), 3) Snafu, an ULP coarse-grain-reconfigurable-array (CGRA) generation framework and architecture, and 4) RipTide, a co-designed dataflow compiler and energy-minimal CGRA. Sonic was the first demonstration of machine inference on a commercial, intermittently-operating device, but also exposed the flaws of such devices. Manic fixed these problems by combining vector execution to amortize instruction fetch with dataflow execution to minimize data supply energy by forwarding intermediate values directly from producers to consumers. Snafu extended Manic's vector-dataflow to further reduce energy by minimizing the toggling of shared pipeline resources. Its generated CGRAs implement spatial-vector-dataflow execution that lays out computation across a fabric of PEs, keeping each PE configured in the same way throughout kernel execution. Finally, RipTide improves overall system efficiency by compiling and offloading to its CGRA, programs written in C with complex control-flow and irregular memory accesses. Together these contributions form the basis of a new ULP sensor system stack that is > 2 orders-of-magnitude more efficient than existing systems, enabling new emerging applications that require intelligence "beyond-the-edge."

Acknowledgements

There are many people to thank for helping me throughout my PhD. First my research advisors, Professors Brandon Lucia and Nathan Beckmann. Brandon and Nathan have taught me how to be a good researcher. They helped me find interesting problems and guided me in my pursuit of solutions. They have complementary advising styles that has a made us a formidable team. Brandon always has a deep understanding of the big picture, which has kept my research grounded. Nathan's detail-oriented approach has made my research stronger and more polished. The success I have enjoyed during my PhD would not have been possible without them.

I am grateful to my close collaborators, including Oguz Aatli, Souradip Ghosh, Professor Kenneth Mai, Danny Bankman, Professor Todd Mowry, and Professor Tony Nowatzki. Oguz, Ken, and Danny were pivotal in our successful tape-out of Manic, guiding me throughout the tape-out process and helping me verify, debug, and optimize our design. Tony, Todd, and Souradip were central to the development of RipTide. Bouncing ideas off of Tony and Todd during weekly research meetings, helped me refine ideas and improved the design of RipTide. Souradip's contributions to RipTide's compiler were critical to the project's success and I thoroughly enjoyed our work together.

I have been fortunate to receive an Apple PhD Fellowship in AI/ML and spend a summer at Apple working on machine learning architecture. Jaewon Shin, my manager, helped me navigate research in industry and encouraged me to be independent and pursue difficult but relevant problems.

I want to thank additional CMU colleagues and administrative staff. I thank Harsh Desai, Alexei Colin, and Emily Ruppel for answering my questions on microelectrics and for their support of my work on intermittent computing. I also want to thank the other PhD students in Brandon's and Nathan's groups for providing extremely useful feedback on my research during practice talks and weekly lunch meetings. Finally, I am grateful to Deborah Cavlovich for helping me navigate the logistics of the PhD.

There are several friends I want to highlight that provided advice and support throughout my graduate studies. Michael Rudow, Jack Kosaian, Matt Butrovich, Han Zhang, and Guilio Zhou are fellow CS PhD students and friends. They not only provided an outlet for discussing my PhD, but also helped me experience Pittsburgh. I am also grateful to George Yu. He provided me valuable advice and a different perspective during our biweekly discussions. He has also been the very best travel companion, willing to meet in whatever corner of the world.

Last but not least, I want to thank my parents and my brother. My parents, Beth and John, encouraged me to pursue a PhD and helped me through the lows and the highs of the process. My brother, Reid, is my best friend and has always been there in whatever situation. I am the man that I am today because of them.

Contents

Chapter 1

Introduction

Ultra-low-power (ULP) sensor devices are increasingly being deployed for a variety of use-cases from in-body health sensing to civil infrastructure monitoring to tiny chip-scale satellites [255]. These devices are composed of low-power sensors attached to an ULP microcontroller, which communicates to other edge devices or to the cloud via a low-power radio, all powered by a small battery or from energy harvested from the environment. The sensors on these devices are increasingly capable — ranging from high-definition image sensors [169] to multi-sensor arrays [131] — producing a growing volume of data. To make sense of the data, sophisticated processing, like machine learning (ML) inference using a deep neural network (DNN) or digital signal processing (DSP), is required. But sophisticated processing requires resources that existing ULP devices lack. In particular, ULP sensor systems are extremely energy-constrained — batteries limit device lifetimes and energy harvested from the environment is extremely scarce. One solution is to offload processing, but communication uses much more energy than local compute. Thus, the energy-efficiency of onboard compute is the key determinant of application success. However, existing, general-purposes systems suffer fundamental inefficiencies. And highly-specialized systems (i.e. ASICs) compromise on programmability, a requirement as applications emerge in the domain. So how can we achieve extreme energy-efficiency to enable sophisticated computations on ultra-low-power systems without sacrificing programmability? The answer — a new system stack — is the objective of this work.

1.1. CHALLENGES

There are a multitude of challenges, detailed below, to supporting sophisticated processing on ULP sensor devices. These devices are severely resource- and energy-constrained, complicating application development and limiting the lifetimes of deployed devices. There is a need for extreme energy-efficiency without sacrificing programmability. This is the central theme to the this thesis.

Offloading computation does not scale

Offloading computation from a ULP device to a more powerful nearby computer (e.g., at the "edge" or cloud) is one approach to the increased processing sophistication and sensed data volume of applications in the ULP domain. The more data a sensor produces, though, the more data the device must communicate. Unfortunately, transmitting data compromises security, clogs networks, and takes much more energy per byte than sensing, storing, or locally computing on those data [84, 145]. While a high-powered device like a smartphone, with a high-bandwidth, long-range radio, can afford to offload data to the edge or cloud, this is not practical for power-, energy-, and bandwidth-limited sensor devices [74, 84].

Local compute reduces cost of communication

Since offloading is infeasible, the alternative is to process data *locally* on the sensor node itself. Our work, SONIC, demonstrates how systems can use commodity off-the-shelf microcontrollers (COTS MCU) to filter sensed data locally so that only meaningful data (as defined by the application) are transmitted. Processing data locally minimizes the high energy cost of communication, reducing energy by $\approx 20\times$ compared to a design that always offloads, but makes the application highly sensitive to the energy-efficiency of computation.

Energy-efficiency is critical to end-to-end system performance

Energy efficiency is the primary determinant of end-to-end system performance in ULP embedded systems. For battery-powered devices [56, 203], energy efficiency determines device lifetime: once a single-charge battery has been depleted the device is dead and it is impractical to replace the battery on millions (or more [219]) deployed devices. Even rechargeable batteries are limited in the number of recharge cycles, and a simple data-logging application can wear out the battery in just a few years [113, 172]. For energy-harvesting devices [48, 101, 103, 249, 256], energy efficiency determines device performance. These devices store energy in a capacitor and spend most of their time powered off, waiting for the capacitor to recharge. Greater energy efficiency leads to less time waiting and more time doing useful work [71].

Existing devices are energy-inefficient

However, ULP COTS MCUs used in many deeply embedded sensor nodes (e.g., TI MSP430, ARM M0+ & M4+) are energy-inefficient. These MCUs are general-purpose, programmable devices that support a variety of applications. But this generality comes at at a high power, energy, and performance cost.

Programmability is expensive in three main ways [24, 92, 107]. First, *instruction supply* consumes significant energy: in the best case, the energy of an instruction cache hit, and in the worst case, the energy of a main memory read and instruction cache fill. Lacking sophisticated microarchitectural features such as superscalar and out-of-order execution pipelines [110, 233], the energy overhead of instruction supply constitutes a significant fraction of total operating energy. Second, data supply through *register file (RF) access* also consumes significant energy. And third, *pipeline-control* can burn significant energy as resources are reconfigured from cycle-to-cycle to run different operations. Together, instruction fetch/decode, data supply, and pipeline-control can consume at least 54.4% of the average execution energy across a variety of representative workloads for ULP devices.

Specialization *can* limit programmability

To combat the energy costs of generality, some recent work has turned to microarchitectural specialization, making a system energy-efficient at the expense of generality and programmability [39–41, 75, 143, 237]. Specialization customizes a system's control and datapath to accommodate a particular workload (e.g., deep neural networks [39, 40]), eliminating inessential inefficiencies like instruction supply, RF access, and pipeline-control. The downsides of specialization are its high non-recurring engineering cost and its inability to support a wide range of applications. Given the emerging nature of applications (e.g., due to new machine learning algorithms [116]) in the ULP domain, specialization is premature. New architecture must be highly programmable, while at the same time being extremely energy-efficient.

Existing execution models are flawed

Programmability and energy efficiency might seem to be wholly incompatible, considering the gap between fixed-function ASIC designs and COTS scalar designs.

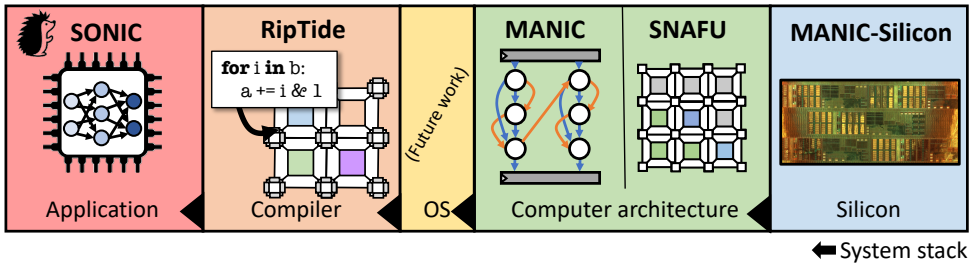


Figure 1.1: This thesis contributes a new energy-efficient system stack: SONIC is a software framework that enables intermittent inference, RIPTIDE compiles C-code to ULP CGRAs, MANIC is an ULP vector-dataflow co-processor, SNAFU generates ULP CGRAs, and MANIC-SILICON is a silicon prototype of MANIC.

However, the trade-off is not as pronounced as these designs would lead one to believe. Their execution models are at the extremes; the fixed-function execution model of ASIC designs limits programmability, while the scalar execution model of COTS MCUs wastes significant energy. There is room in the middle for alternative execution models that balance programmability and energy-efficiency. For example, one starting point is vector execution, which slightly reduces programmability, but amortizes instruction supply energy improving overall energy-efficiency. Developing new execution models, therefore, is critical to resolving the tension between programmability and energy-efficiency.

Choosing the right programming interface

Programmability needs to come in the correct form. It is more than just configurability. A design that's highly configurable, but difficult to program from a high-level language will have limited adoption. Legacy software needs to be supported out-of-the-box with minimal changes so that developers can quickly adopt new hardware. However, there are downsides (e.g., incomplete information on memory-ordering and parallelism) to sticking with established programming interfaces. Choosing the right programming interface and building a compiler to target new hardware is as important as the hardware itself.

1.2. OBJECTIVE OF THIS WORK

This work proposes a complete system stack that leverages new execution models to maximize energy-efficiency without significantly sacrificing programmability. This approach enables new applications in the ULP domain as improved energy efficiency makes sophisticated workloads practical, while maintaining support for programmability allows for iteration, development of new algorithms, and quick deployment. Our work fills out the stack – from software to compilation to computer architecture and to silicon implementation. Together these works support the following thesis:

> High energy-efficiency can be achieved across the system stack from software to silicon without significantly compromising on programmability by leveraging new execution models to reduce instruction fetch/decode, pipeline-control, and data supply energies.

The following contributions form the basis for the thesis and the new system stack.

[Software] SONIC is the first demonstration of DNN inference on an energy-harvesting device (Ch. 3):

SONIC is the software-component of the new ULP sensor system stack. It is an intermittence-aware software system with specialized support for DNN inference. SONIC runs optimized networks found using GENESIS, a tool that

automatically compresses networks to balance inference accuracy and energy. SONIC introduces loop continuation, a new technique that dramatically reduces the cost of guaranteeing correct intermittent execution for loop-heavy code like DNN inference. Across three neural networks on a commercially available MCU, SONIC reduces inference energy by 6.9× over the prior state-of-the-art.

[Architecture] MANIC is an energy-efficient vector-dataflow co-processor (Ch. 4): MANIC contributes new computer architecture to the ULP system stack. It is an efficient vector-dataflow architecture for ultra-low-power embedded systems, achieving high energy-efficiency without sacrificing programmability and generality. MANIC introduces *vector-dataflow execution*, allowing it to exploit the dataflows in a sequence of vector instructions and amortize instruction fetch and decode over a whole vector of operations. By forwarding values from producers to consumers, MANIC avoids costly vector register file reads. By carefully scheduling code and avoiding dead register writes, MANIC avoids costly vector register writes. On average, MANIC is 3.4× more energy efficient than a scalar baseline and 12% more energy-efficient than a vector baseline.

[Silicon] MANIC-SILICON proves the energy-efficiency of MANIC (Sec. 4.3):

MANIC-SILICON represents the silicon component of the new ULP sensor system stack. It is a prototype of the MANIC architecture that demonstrates the energy-efficiency of the design. MANIC-SILICON is a complete, standalone system possessing a RISC-V scalar core, the MANIC co-processor, a data cache, an instruction cache, and main memory composed of 64KB of SRAM and 256KB of non-volatile embedded MRAM. The design is implemented in Intel 22FFL high-threshold-voltage process and achieves a max efficiency of 256 MOPS/mW drawing just 19μW at 4MHz.

[Architecture] SNAFU generates ULP CGRAs (Ch. 5):

SNAFU is another new computer architecture in the ULP system stack. SNAFU generates ULP coarse-grain reconfigurable arrays (CGRAs) that implement spatial-vector-dataflow execution, building on MANIC's vector-dataflow execution model. In spatial-vector-dataflow execution, a dataflow graph (DFG) is mapped spatially across the fabric of processing elements, applying the same DFG to many input data values, and routing intermediate values directly from producers to consumers. This minimizes instruction and data-movement energy, just like MANIC, and reduces pipeline-control energy by eliminating unnecessary switching activity as operations do not share execution hardware. SNAFU uses 41% less energy and runs 4.4× faster than MANIC.

[Architecture & Compilation] RIPTIDE is an energy-minimal dataflow compiler & CGRA architecture (Ch. 6):

RIPTIDE rounds out the system stack, providing an ULP CGRA *architecture* and co-designed *compiler* that compiles high-level C-code to the new hardware. It proposes a new set of program primitives that support arbitrary control-flow, irregular memory accesses, common program idioms, and memory ordering without needing to tag values to save energy; this requires careful analysis by the compiler to guarantee correctness, particularly to enforce control-flow and memory ordering. To save even more energy, RIPTIDE offloads control-flow operations into the on-chip network. RIPTIDE observes that these operations

are simple but prevalent, so it is wasteful to assign them to processing elements. Instead, RipTide reuses existing hardware in the on-chip network to directly implement control-flows operations. Compared to Snafu, RipTide uses 25% less energy and is also 17% faster without requiring hand-coded assembly.

Together Sonic, Manic, Manic-Silicon, Snafu, and RipTide form the new energy-efficient system stack. This stack is capable of running C programs within 2.3× of the energy of ASICs, enabling sophisticated applications involving DNN inference to last 5 to 10 years on a single AA battery. This combination of programmbility and state-of-the-art energy efficiency represents a paradigm-shift for ULP sensor systems. Deployed devices do not need to be frequently replaced or recharged, reducing maintenance costs and carbon footprints. At the same time, applications can be updated in the field, allowing the device to adapt to environmental changes and improve onboard processing. In short, this new system stack will facilitate development of emerging applications in the ULP sensor domain.

1.3. OUTLINE

This thesis is divided into eight chapters, including this introduction. The second chapter provides useful background on embedded systems, intermittent computing, vector architectures, dataflow architectures, and coarse-grain reconfigurable arrays. The next four chapters describe Sonic, Manic & Manic-Silicon, Snafu, and RipTide in detail. The order of these chapters follows the timeline of development. This structure will make clear the connection between each work: Sonic, despite being the first to demonstrate DNN inference on an energy-harvesting device, exposed the drawbacks of existing ULP devices' scalar execution model; Manic solved these drawbacks with vector-dataflow execution; Snafu improved on Manic with spatial-vector-dataflow execution; and RipTide built on Snafu to further improve general-purpose programmability and overall system energy-efficiency. Finally, the last two chapters discuss exciting future research directions in ULP computer architecture and CGRAs and then conclude.

Chapter 2

Background

This chapter discusses relevant prior work that this thesis builds on. The chapter is split into discussions on 1) low-powered embedded devices, 2) edge inference, 3) related efficient programmable architectures, and 4) coarse-grain reconfigurable arrays. The first includes a general overview of ULP sensor devices and an introduction to intermittent computing. The second discusses algorithmic improvements and hardware accelerators for efficient neural network inference. The third describes relevant vector machine and dataflow architectures. And the fourth expands on a specific class of spatial-dataflow architectures, called coarse-grain reconfigurable arrays.

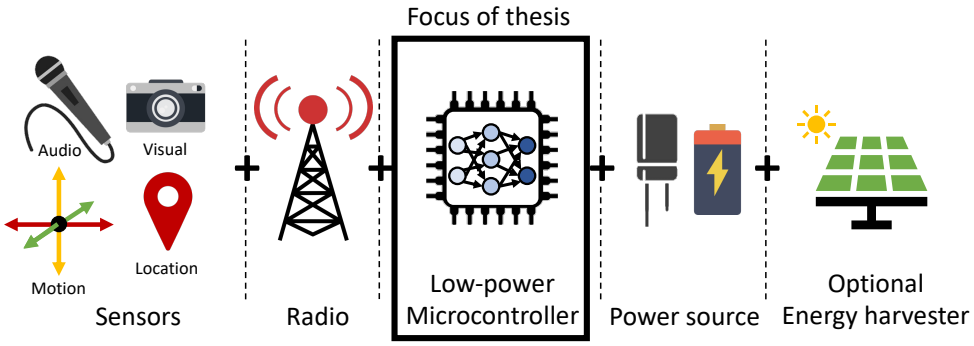


Figure 2.1: Typical ULP device is composed of 1) series of sensors (e.g., camera, microphone, GPS, accelerometer, etc.), 2) low-power radio (e.g., LoRaWAN or BLE), 3) low-power microcontroller, the focus of this thesis, 4) power source like a battery or capacitor and 5) an optional energy harvester like a solar cell.

2.1. LOW-POWER EMBEDDED DEVICES

Low-power embedded sensor devices as shown in Fig. 2.1 are composed of: 1) low-powered sensors (e.g. HiMax HM01B0 camera [3]), 2) a low-powered radio (e.g. LoRaWAN [11] or BLE [229]), 3) an ULP microcontroller (e.g. ARM M0 [1] or TI-MSP430 [110], 4) a power source like a coin-cell battery or capacitor, and 5) an optional energy harvester (e.g. solar cell or RF harvester [197]). Energy dictates the viability of these devices. For battery-powered devices energy dictates lifetime, and for energy-harvesting devices that operate intermittently, energy dictates performance by controlling the time spent waiting for energy to be collected. On top of this, the MCUs of these are devices are simple, severely resource-constrained and energy-inefficient.

2.1.1 Device operation

ULP sensor devices operate on a duty cycle: periodically sensors collect data from the environment, the microcontroller processes the data, and then data is transmitted via the low-power radio. This interval of data collection is dictated by application requirements (i.e. how often a particular event will take place), the required lifetime for the device (particularly for battery-powered devices), and energy-consumption of

the different components of the device. Application requirements have to be carefully balanced with the energy consumption of the MCU, sensors, and radio.

Battery-backed systems: Some devices rely on batteries [56,113,203]. These devices can quickly deplete their battery even if the application is simply logging data. Battery lifetimes can be improved by adding a energy harvester, like a solar cell or radio-frequency harvester, that can recharge the battery. However, rechargeable batteries have limited recharge cycles and may not be able to cope with extreme environmental conditions (e.g. too hot/cold temperatures).

Capacitor-backed systems: Instead a capacitor can be used to buffer energy from an energy harvester and power the device. Capacitors have an effective lifetime (> 10 years) that often exceeds the lengths of application deployments. But they do not offer the same energy density as batteries. In a capacitor-backed system, the application often must wait for energy to be collected in the capacitor by the energy-harvester. This makes application performance dependent on the availability of energy in the environment. Energy availability is not constant — input power can vary with environmental conditions. For example, weather and time-of-day significantly (by several orders of magnitude) impact the amount of energy a solar cell can harvest. These operating conditions complicate execution and affect the device's ability to quickly react to changing environmental conditions.

2.1.2 Intermittent execution model

Systems that harvest energy and store that energy in a hardware buffer (e.g. capacitor) usually operate intermittently. This is because device operating power usually exceeds power harvested from the environment. To operate despite this weak input power, a device slowly accumulates energy in a hardware buffer and operates when the buffer is full. The device drains the buffer as it operates, then it turns off and waits for the buffer to fill again.

Software executes in the *intermittent execution model* on these energy-harvesting devices [36, 114, 148, 158, 159, 200]. In intermittent execution, software progresses in bursts, resetting at frequent power failures. Existing devices [110, 230] mix volatile state (e.g. registers and SRAM) and non-volatile memory (e.g. FRAM). A power failure clears volatile state while non-volatile memory persists. Repeated power failures impede progress [200], and may leave memory inconsistent due to partially or repeatedly applied non-volatile memory updates [148]. These progress and consistency issues lead to incorrect behavior that deviates from any continuously-powered execution [46]. Specifically, write-after-read (WAR) dependences lead to inconsistent memory and differing control-flow as re-execution can expose the value from the latter write to the read, which is not possible in continuously-powered execution.

Prior work addressed progress and memory consistency using software checkpoints [105, 148, 235], non-volatile processors (NVPs) [149, 150], and programming models based around atomic tasks [47, 104, 151, 151]. Non-volatile processors require technology process changes and constantly pay a tax for their non-volatility (i.e., non-volatile elements cost more energy than volatile elements). Software-checkpointing and task-based runtime systems, on the other hand, can be deployed on existing devices with limited impact on energy since the amount of volatile state is small.

Checkpointing systems

Checkpoint-based systems insert checkpoints into programs using compiler, runtime, and hardware support. Just-in-time (JIT) checkpointing is the most popular strategy. In JIT checkpointing, hardware monitors the voltage of the capacitor and when the voltage dips below a predetermined threshold indicating a power failure is

imminent, triggers an interrupt that checkpoints program state. The interrupt writes back the volatile state of the program, including program counter and stack, to non-volatile memory. Then when power resumes, the runtime system restores the volatile state and jumps back to the place in execution where power failed. This system is often transparent to the programmer, but may complicate (or even lead to incorrect) use of peripherals [220] and interrupts.

Task-based runtime systems

An alternative to checkpointing are task-based runtime systems. These systems avoid frequent checkpoints by restarting from a task's start after power failure, at which point all register and stack state must be re-initialized. To ensure memory consistency, tasks ensure that the effect of a partial task execution is not visible to a subsequent re-execution. Specifically, data that are read then written (i.e., a WAR dependence) may expose the result of an interrupted task. Task-based systems avoid "the WAR problem" with redo-logging [151] and static data duplication [47].

Task-based systems guarantee correct execution, but at a significant run-time cost. Redo-logging and static duplication both increase memory and compute in proportion to the amount of data written. Transitioning from one task to the next takes time, so short tasks that transition frequently suffer poor performance. Long tasks better amortize transition costs, but re-execute more work after a power failure. Worse, a task that is too long faces *non-termination* if the energy it requires exceeds the energy that the device can buffer. The programmer, therefore, needs to be careful in splitting a program into atomic tasks.

2.1.3 COTS ULP Devices

In addition to being energy-constrained, ULP sensor devices are also severely resource constrained. ARM's Cortex M0 [1] or TI's MSP430 [110] are the most commonly used processors in existing ULP sensor systems [49,100,102,103,206,230]. Such MCUs' frequency is typically 1–16MHz, leaving a substantial performance gap compared to, e.g. a full-fledged, 2GHz Xeon-based system. The MCU usually also houses all the memory available to the system, including embedded SRAM, which is volatile, and embedded non-volatile memory (e.g. FRAM). Embedded memories are small and capacity varies by device. A typical MSP430 low-power MCU includes just 1–4KB of SRAM and 32–256KB of FRAM. While continuously powered (i.e., wired) embedded systems may interface with larger memories via a serial bus (I^2C or SPI), most ULP sensor devices do not due to their high access energy and latency. The typical operating power of an COTS ULP device is around 3–5mW.

Architecture

COTS ULP devices achieve ULP operation by being simple. They have an 3 to 5 stage, in-order scalar core, which may lack instruction and/or data caches. Some MCUs also come with a vector co-processor such as TI's Low Energy Accelerator (LEA) [111] or support for vector extensions like Arm's Neon [17] vector ISA. Additionally, some MCUs also include DMA engines and accelerators for tasks like AES encryption [110]. Despite these additional features, the energy consumption of the scalar core (and it's memory accesses) dominates total MCU energy. This is because the scalar execution model pays a high price for general-purpose programmability, constantly refetching and redecoding the same instructions and communicating intermediates via a centralized register file. The purpose of this thesis is to reduce this tax for general-purpose programmability. Chapters Ch. 4, Ch. 5, and Ch. 6 propose new computer architectures that maximize efficiency while maintaining a high-degree of programmability.

2.2. EDGE INFERENCE

As ULP sensor devices become pervasive they will increasingly need to make intelligent decisions. Deep neural network (DNN) inference is the state-of-the-art approach for such intelligence. They are the standard for applications ranging from understanding speech to image recognition [129, 215, 223]. With their accuracy, however, comes a high computational cost. Neural networks require millions or billions of parameters and operations. This makes deploying neural networks onto resource-constrained, energy-harvesting devices difficult. Fortunately there has been much work on reducing network footprint and improving the performance and energy-efficiency of inference.

2.2.1 Algorithmic improvements to NN inference

Since DNNs are robust to noise, algorithmic optimizations can be made that reduce NN memory footprint and increase inference performance without significantly impacting accuracy. Inference does not need full-precision floating point [66, 95] and near-zero weights can often be "pruned" [29, 96, 168, 171] without losing much accuracy. Layers can also be factored or split into several smaller, less-computationally intense layers [45, 222, 224]. Finally, networks can be redesigned [108, 207, 227] from the ground up to minimize storage and computation. These networks leverage smaller convolutional filters, but make up for accuracy degradation by being wider or deeper. This is worthwhile because there is a quadratic relationship between convolutional filter size (i.e., side length) and computational cost.

These algorithmic improvements sometimes come with modifications to the training regime to further increase accuracy. Networks can be fine-tuned during the final stages (final 20-30% of epochs) of training to adapt to algorithmic changes. During fine-tuning, the forward direction (i.e., inference) adopts the algorithmic change (e.g. reduced precision or pruning), while the backward direction remains the same. This allows the network to adjust its weights to the algorithmic change.

More exotic training regimes have also been explored to enforce additional properties to help inference performance. Binary networks [142] learn binary (-1, +1) weights that simplify multiplication to a single `and` operation. Structured sparsity can also be enforced [67, 171]. Pruning can lead to extremely sparse layers without much structure which can lead to irregular memory accesses. By enforcing structure on this sparsity during training, irregular memory accesses can be reduced.

Ch. 3 discusses the application of several algorithmic changes to compress neural networks to fit into device memory. It describes a tool, called GENESIS, that prunes, factors, and reduces the precision of neural networks, fine-tuning network weights to improve accuracy.

2.2.2 Inference accelerators

The computer architecture community has also responded to the need for efficient DNN inference. Some architectures focus on dense computations [39–41], others on sparse computations [75, 95, 130, 257], and still others on CNN acceleration [14, 15, 73, 190, 201, 218]. Industry has followed this trend, embracing custom silicon for DNNs [117]. The key to the efficiency of these architectures is to maximize data reuse and optimize data movement. For example, Eyeriss [40] introduces row-stationary dataflow to maximize reuse on a spatial fabric, while MAERI [130] designs a new on-chip network specialized for several DNN dataflows (including hard-to-accelerate LSTMs).

The circuits community has also responded, taping out extremely specialized accelerators with orders of magnitude higher energy-efficiency. These designs use low-level VLSI techniques (e.g. sub-threshold computing [78]), custom analog or mixed-signal circuits [27], or exploit emerging technologies (e.g. ReRAM crossbars [252, 253]) to

achieve TOPS/W. Unfortunately this extreme energy efficiency comes at the expense of programmability (and perhaps reliability, in the case of analog circuits). In fact, some accelerators specialize to the particular NN architecture [27].

The work described in Ch. 4, Ch. 5, and Ch. 6 take a different approach and focus on entirely different power domain (mW v. μW). Energy-efficiency is achieved without sacrificing programmability. This makes these designs applicable to many different applications, future-proof to algorithmic developments, and reduces the upfront non-recurring engineering cost.

2.3. EFFICIENT PROGRAMMABLE ARCHITECTURES

There is a long history of computer architectures that increase performance and/or improve energy-efficiency while maintaining programmability. Specifically, prior work on vector and dataflow architectures inform the work of this thesis and specifically Manic, Snafu, and RipTide. These architectures exploit structure in the program (among data and control-flow) to change the execution model to reduce or even eliminate instruction and data supply energies without sacrificing general-purpose programmability.

2.3.1 Vector architectures

Vector machines exploit data parallelism to amortize instruction supply energy (i.e., fetch, decode, and issue) across many operations. Early vector machines targeted super computing [55], but most commercially available architectures (e.g. AVX [79] and GPUs [53]) today support vectors. These vector designs target performance and operate at a power budget orders-of-magnitude higher than that contributed by this thesis. Nonetheless, all vector designs require large vector register files (VRF) exacerbating register file access cost, especially in designs that require a VRF with many ports. Thus, reducing VRF cost and complexity has been a primary focus of prior vector designs [18, 128].

T0 [18,246] is a vector architecture with reconfigurable pipelines. Software controls datapaths to chain operations, eliminating VRF access within a chain. However, microarchitectural details of the datapath are exposed to software, requiring major software changes and recompilation.

CODE [128] reduces VRF cost by distributing the VRF among heterogeneous functional units. This design is transparent to software because CODE renames operands at instruction issue to use registers near an appropriate functional unit. Distribution lets CODE reduce VRF ports, but requires a routing network to send values between functional units.

AVA [133] is a high-performance out-of-order vector processor that adapts the vector length and the number of vector registers stored in the VRF to the application. Some applications require long vectors, while others want moderate sizes. AVA trades additional vector length for fewer vector registers, spilling registers to memory as needed and then smartly pre-fetching them. Although AVA reduces energy and area, its primary goal is performance (by supporting variable-sized vector lengths).

Finally, there has been much work on reducing the cost and improving the scalability of GPU register files. This includes virtualizing the RF [115, 238] so that physical registers can be shared, compressing registers [139] to maximize utilization of the physical RF, and coalescing RF reads and writes [19] to minimize RF accesses. Relative to the large (100s of KB), many-ported (32+ ports) register files of GPUs, these optimizations require minimal additional hardware, but do not scale to ULP domain where energy efficiency is higher priority than performance.

2.3.2 Dataflow architectures

Dataflow architectures, like vector machines, also have a long history [68–70, 177] that includes changes to the programming and execution model to eliminate control and data movement overheads. In particular, dataflow is prevalent today as part out-of-order (OoO) execution engines, where restricted dataflow improves performance and reduces RF pressure [20, 33, 124, 210, 212]. However, pure dataflow architectures have not found the same commercial success, but spatial-dataflow architectures still show great promise for improving energy efficiency and performance.

Dennis proposed the first dataflow architecture in 1975, introducing a small set of primitives to implement arbitrary control-flow by conditionally routing values to consumers [70]. In 1990, the MIT tagged-token dataflow machine showed how to practically implement dataflow in hardware [178]. The design relies on the use of a domain-specific language, called Id [176], to describe program dataflow and cap resource use to avoid resource-based deadlocks from unbounded parallelism [188].

Later, Wavescalar [221] and Trips [208] identified dataflow locality as the key determinant of sequential code performance. Wavescalar compiles C/C++ programs to WaveCache, a grid of simple compute units and memory that co-locates computation with data. Wavescalar handles arbitrary control-flow by tagging data (to distinguish instances of a value across loop iterations) and enforcing memory ordering by converting memory dependences to data dependences during compilation. Despite preserving dataflow locality, Wavescalar was not designed to minimize energy — the architecture relies on expensive tag-token matching and still fetches instructions, constantly reconfiguring compute pipelines.

Trips extracts hyperblocks (multiple basic blocks without backedges) from programs and executes these hyperblocks in dataflow-fashion across a mesh of processing elements. Trips heavily relies on speculation to increase performance; in-flight operations number in the hundreds or even thousands. This amount of speculation inevitably wastes energy by discarding mispeculated work. Further, Trips constantly reconfigures PE pipelines for each hyperblock, toggling control and data signals.

In contrast, ELM [23] is a dataflow architecture specifically designed for low-power, embedded operation. ELM uses restricted SIMD execution and operand forwarding to provide dataflow-like execution. ELM's complex register file hierarchy and forwarding mechanism are software-controlled, exposing microarchitectural details to the programmer and requiring significant changes to the compiler toolchain.

2.4. COARSE-GRAIN RECONFIGURABLE ARRAYS

Since Wavescalar, Trips, and ELM there has been a resurgence of spatial-dataflow architectures, called coarse-grain reconfigurable arrays (CGRAs), because they offer better energy efficiency and performance with lower hardware complexity. A CGRA architecture [26, 44, 52, 60, 76, 86, 88, 119, 154, 161, 162, 164, 174, 175, 182, 189, 192, 198, 204, 209, 216, 225, 226, 240, 241, 247, 248] is a spatial array of processing elements (PEs) connected by an on-chip interconnect (NoC). A PE in a CGRA consumes inputs and produces outputs consumed by another PE, forming a pipeline corresponding to program dataflow. CGRA efficiency derives from avoiding control and data-movement overheads. A CGRA reduces instruction overheads by mapping operations to a PE, avoiding the need for instruction fetch and decode and simplifying control. A CGRA mitigates data movement overheads by avoiding large register files, instead moving operands through a NoC directly from producer PE to consumer PEs.

A wide variety of CGRA architectures target different domains. CGRAs exist as standalone cores [162, 208, 221, 242], co-processors [50, 51, 86, 90, 97, 154, 183, 225],

components of a processor pipeline [88, 137, 144] or memory hierarchy [146], or as co-processors [52, 58–60, 174, 175, 182, 192, 198, 204, 216, 239, 240, 248, 251]. These contexts expose a wide range of hardware design choices, including PE operation set, PE complexity, and NoC topology. Further, PEs may be homogeneous or heterogeneous; the latter is more area- and energy-efficient, but creates a combinatorally large design space [26]. PEs typically include functional units for arithmetic, logic, and memory access, but can also include specialized functionality [58–60, 81, 204, 239, 247, 251]. Later chapters (Ch. 5 and Ch. 6) of this thesis will address CGRA design decisions in detail to maximize energy-efficiency.

2.4.1 Types of CGRAs

CGRA designs can be categorized in four ways, as identified by [248]. They distiguished by how they schedule operations and whether PE resources are shared. Specifically the four types of CGRAs are: systolic (statically scheduled & dedicated PEs), shared-systolic (statically scheduled & PEs shared), tagged-dataflow (dynamically scheduled & PEs shared), and ordered-dataflow (dynamically scheduled & dedicated PEs).

Systolic: Systolic designs [52, 86, 162, 182, 193] rely on the compiler to schedule operations in space and time. These designs achieve high performance and energy-efficiency by eliminating dynamic control, but make compilation challenging. To maximize utilization, the compiler must reason about operation latencies, but these latencies might not be available at compilation time and might not be fixed (e.g. memory access latency depends on where data exists in the hierarchy). This limits the applications that can be mapped to these designs.

Shared-systolic: Shared-systolic designs [119, 154, 161, 164, 216] ([248] refers to these as "CGRAs") add additional layer of complexity to systolic designs. To maximize utilization of available hardware, the compiler generates schedules where operations can time-multiplex on the same PE resources. This makes scheduling even more challenging v. systolic designs.

Tagged-dataflow: Tagged-dataflow designs [177, 189, 208, 221, 240] dynamically schedule operations in time (and potentially in space) and time-multiplex PE resources. These designs tag data tokens and then match tags to distinguish between multiple instances of a single value, dynamically firing/enabling an operation when tags of inputs match. This tagging mechanism maximizes performance, allowing speculation and the re-ordering of operations and even entire loop iterations. However, this sort of dynamic tracking of values requires a high power budget (100s of mW v. ≈1mW of ULP domain) and comes at a significant energy cost.

Ordered-dataflow: Ordered-dataflow designs [81, 88, 198] are the final category of CGRAs. They do not share PE resources and dynamically schedule operations in time, but disallow the re-ordering of tokens. This makes compilation easier (v. systolic and shared-systolic) since the compiler need not reason about operation latencies and is cheaper to implement in hardware (v. tagged-dataflow) since there is no need for tag matching (because tokens arrive in order). These benefits make ordered-dataflow a good choice for the ULP domain and are the reasons why SNAFU (Ch. 5) and RIPTIDE (Ch. 6) implement it.

2.4.2 Low-power CGRAs

The CAD and circuit communities [62, 119, 123, 187] have also contributed low-power CGRA designs. These designs are usually systolic or shared-systolic and operate at 10s of mW (still order of magnitude more than ULP domain). They use VLSI techniques to reduce power (e.g. low-voltage design, fine-grain clock/power gating)

that are complementary to the CGRA designs (SNAFU and RIPTIDE) in this thesis. This thesis focuses on *architecture* that minimizes energy and maintains flexibility. In fact, Ch. 5 describes SNAFU, which has a goal of letting designers generate ULP CGRAs at reduced VLSI effort.

2.4.3 Compilation

Compiling for CGRAs is challenging. Some architectures require domain-specific languages [126, 176], while others place constraints on the types of programs that can be compiled (i.e., no outer loops, or memory ordering is not enforced). This makes compilation tractable, but also limits the applications supported.

Compilation for CGRAs is similar to hardware synthesis. The compiler must find a layout of operations that fits within fabric resources with valid routes between all producers and consumers. In performance-focused CGRAs, the compiler must also reason about timing to maximize utilization and minimize initiation interval (minimum interval between subsequent loop iterations). With this vast search space, optimization-based methods often do not converge in a reasonable time [180, 185]. Most CGRA compilers use heuristics [26, 153, 180, 185, 191, 226, 245, 247] that can fail or produce poor mappings. Recent work proposed graph convolutional networks as a solution [160]. This thesis will (Ch. 5 and Ch. 6) describe integer-linear programming and SAT-based approaches to mapping a program to PE resources. Mapping is tractable because the compiler does not need to reason about timing or utilization for the proposed architectures (SNAFU and RIPTIDE).

Dataflow control-flow models

Compilation is also affected by the underlying control-flow model. There are three competing control-flow models for dataflow execution: predication, selection (ϕ), and steering (ϕ^{-1}). Each has benefits and drawbacks. Steering is the best choice for an energy-constrained context because it avoids routing values to the not-taken branch paths, but it can require extra routing by the compiler.

Predication routes values unnecessarily: Predication is popular, especially in GPU and vector architectures [79, 99], converting conditional code to straightline code to simplify execution. In predication, only one side of a branch fully executes while the other side partially executes, passing through results from the enabled side to downstream consumers. Predication simplifies control flow in CGRAs because tokens arrive on every path, simplifying operand ordering. But predication has a performance and energy cost because values flow unnecessarily through the not-taken path. SNAFU (Ch. 5) uses predication for control flow, supporting simple, affine loops.

Selection (ϕ) burns energy on paths not taken: Selection executes both sides of a branch fully, sending results to a mux that chooses between results using the branch decider. Selection maximizes performance because the branch condition and each side of the branch execute speculatively in parallel. However, selection wastes energy by throwing away work.

Steering (ϕ^{-1}) is most energy efficient: Steering was proposed in the original dataflow paper [70] and has been used notably in a few dataflow architectures [34, 90, 162, 183, 221]. Steering routes values to only the taken path of a branch based on the branch's decider. Steering serializes execution of the taken path on the branch decider, but avoids executing any operations on the not-taken path. RIPTIDE (Ch. 6) implements steering to minimize energy, as steering never fires unneeded operations.

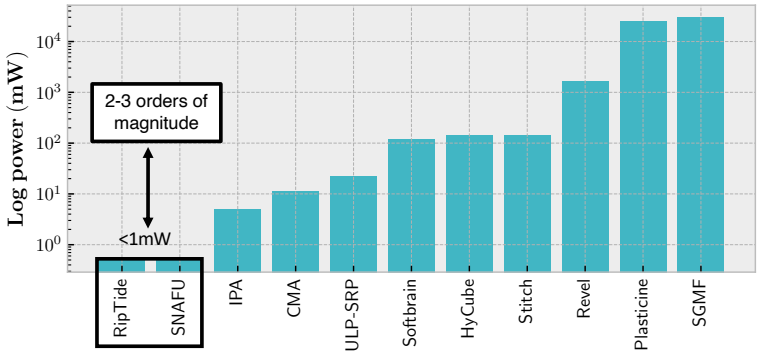


Figure 2.2: This thesis presents work (SNAFU and RIPTIDE) that targets the ULP domain, 2–3 orders of magnitude less than prior work.

	Ultra-low-power CGRAs			High-performance CGRAs				
	SRP [123]	**CMA [187]**	**IPA [62]**	**HyCube [243]**	**Revel [248]**	**SGMF [240]**	**SNAFU [81]**	**RIPTIDE [83]**
Fabric size	3×3	8×10	4×4	4×4	5×5	8×8 & 32 mem	N×N (6×6 evaluated)	N×N (6×6 evaluated)
NoC	Neighbors only	Neighbors only	Neighbors only	Static, bufferless, multi-hop	Static & dynamic NoCs (2×)	Dynamic routing	Static, bufferless, multi-hop	Static, bufferless, multi-hop
PE assignment	Static	Static	Static	Static	Static *or* dynamic	Dynamic	Static	Static
Time-share PEs?	Yes	Yes	Yes	Yes	Yes	Yes	No	No
Scheduling/PE firing	Static	Static	Static	Static	Static *or* dynamic	Dynamic	Dynamic	Dynamic
Heterogeneous PEs?	No	No	No	No	Yes	Yes	Yes	Yes
Program support	VLIW/Simple loops[1]	VLIW/Simple loops[1]	Simple loops[1]	Simple loops[1]	Loops that fit patterns	Arbitrary	Vectorizeable loops	Arbitrary
Power	22 mW	11 mW	3–5 mW	140 mW	1.66 W	20 W	<1 mW	<1 mW
MOPS/mW (approx.)	30–100 MIPS/mW	100–200 MIPS/mW	140 MIPS/mW	26 (system)	12/16 (fabric/system)	—	134/97 (fabric/system)	254/117 (fabric/system)

[1] Simple loops are singly-nested and have few loop-carried dependence.

Table 2.1: Architectural comparison of prior work to SNAFU and RIPTIDE.

2.4.4 Compare & contrast different CGRA designs

Fig. 2.2 and Table 2.1 summarize the CGRA design space, comparing different CGRA designs to work of this thesis, SNAFU and RIPTIDE. In particular, the figures highlight what makes the ULP sensor domain different. Specifically, most prior CGRAs target much higher power domains (0.1-1W [119, 182, 225, 247, 248] or even up to 100 W [198, 240]), and their design decisions do not translate well to the ULP domain. The few CGRAs targeting ULP operation (≈1 mW) [62, 123, 187] are not flexible and leave energy savings on the table. They places restrictions on programs, e.g., supporting only simple, single-nested loops.

In contrast, this thesis presents SNAFU and RIPTIDE, which are designed from the ground-up to minimize energy (even at the expense of area and performance) while maximizing flexibility. SNAFU and RIPTIDE, like DSAGEN [247], generate CGRAs, but instead of targeting performance, they target ULP operation. They minimize PE energy by statically assigning operations to specific PEs and, unlike prior low-power CGRAs [62, 119, 123, 187, 225], minimize switching by not sharing PEs between operations. Likewise, to minimize NoC energy, SNAFU & RIPTIDE implement a statically configured, bufferless, multi-hop NoC, similar to HyCube [119]. This NoC is a contrast with prior ULP CGRAs [62, 123, 187] that restrict communication to a PE's immediate neighbors. Unlike many prior CGRAs that are statically scheduled, SNAFU & RIPTIDE implement dynamic dataflow firing to support variable latency FUs. Dynamic dataflow firing is essential to SNAFU & RIPTIDE's flexibility and ability to support arbitrary, heterogeneous PEs in a single fabric. SNAFU & RIPTIDE avoid expensive tag-token matching [88, 198] by disallowing out-of-order execution, unlike high-performance designs [177, 189, 221, 240]. Lastly, RIPTIDE targets more than simple, vectorizable loops [62, 119, 123, 187], compiling programs directly from C and

supporting deeply-nested loops and irregular memory accesses. The end result is that Snafu & RipTide are flexible and general-purpose, while still achieving extremely low operating power and high energy-efficiency.

Chapter 3

SONIC: Deploying DNNs on intermittent embedded devices[1][2]

The maturation of energy-harvesting technology and the recent emergence of viable intermittent computing models creates the opportunity to build sophisticated battery-less systems with most of the computing, sensing, and communicating capabilities of existing battery-powered systems. Many future IoT applications require frequent decision making, e.g., when to trigger a battery-draining camera, and these decisions must be taken locally, as it is often impractically expensive to communicate with other devices. Future IoT applications will require *local* inference on raw sensor data, and their performance will be determined by inference accuracy. Using energy numbers from recent state-of-the-art systems, we show that such local inference can improve end-to-end application performance by 480× or more.

Recently, deep neural networks (DNNs) [129, 215, 223] have made large strides in inference accuracy. DNNs enable sophisticated inference using limited, noisy inputs, relying on rich models learned from many examples. Unfortunately, while DNNs are much more accurate than traditional alternatives [89, 163], they are also more computationally demanding.

Typical neural networks use tens of millions of weights and require billions of compute operations [129, 215, 223]. These networks target high-powered, throughput-optimized processors like GPUs or Google's TPU, which executes up to 9 trillion operations per second while drawing around 40 watts of power [117]. Even a small DNN (e.g., LeNet [134]) has over a million weights and millions of operations. The most efficient DNN accelerators optimize for performance as well as energy efficiency and consume hundreds of mW [39, 40, 75, 95].

Challenges: In stark contrast to these high-performance systems, energy-harvesting devices use simple microcontrollers (MCUs) built for extreme low-power operation. DNN inference on these devices is unexplored, and several challenges must be overcome to enable emerging IoT applications on energy-harvesting systems built from commodity components. Most importantly, energy-harvesting systems operate *intermittently* as power becomes available, complicating the development of efficient, correct software. The operating period depends on the properties of the power system, but is short—typically around 100,000 instructions. As a result, *existing DNN inference implementations do not tolerate intermittent operation.*

Recent work proposed software systems that guarantee correct execution on intermittent power for arbitrary programs [47, 104, 105, 148, 151, 235]. These systems add significant runtime overheads to ensure correctness, slowing down DNN inference by

[1] [82] G. Gobieski, N. Beckmann, and B. Lucia, "Intermittent deep neural network inference," in SysML, 2018.

[2] [84] G. Gobieski, B. Lucia, and N. Beckmann, "Intelligence beyond the edge: Inference on intermittent embedded systems," in ASPLOS, 2019.

on average 10× in our experiments. What these systems have missed is the opportunity to *exploit the structure of the computation to lower the cost of guaranteeing correctness.* This missed opportunity is especially costly for highly structured and loop-heavy computations like DNN inference.

Our approach and contributions: We present the *first demonstration of intermittent DNN inference* on real-world neural networks running on a widely available energy-harvesting system. We make the following contributions:

- We first analyze where energy is spent in an energy-harvesting system and show that inference accuracy largely determines IoT application performance (Sec. 3.1). This motivates using DNNs despite their added cost over simpler but less accurate inference techniques.
- Building on this analysis, we present GENESIS, a tool that automatically compresses networks to maximize IoT application performance (Sec. 3.3). GENESIS uses known compression techniques [29, 45, 96, 168]; our contribution is that GENESIS optimally balances inference energy v. accuracy.
- We design and implement SONIC, a software system for DNN inference with specialized support for intermittent execution (Sec. 3.4). To ensure correctness at low overhead, SONIC introduces *loop continuation*, which exploits the regular structure of DNN inference to selectively violate task-based abstractions from prior work [151], allowing direct modification of non-volatile memory. Loop continuation is safe because SONIC ensures loop iterations are idempotent through *loop-ordered buffering* (for convolutional layers) and *sparse undo-logging* (for sparse fully-connected layers). These techniques let SONIC resume from where it left off after a power failure, eliminating task transitions and wasted work that plague prior task-based systems.
- Finally, we build TAILS to show how to incorporate hardware acceleration into SONIC (Sec. 3.5). TAILS uses hardware available in some microcontrollers to accelerate matrix multiplication and convolution. TAILS automatically calibrates its parallelism to ensure correctness with intermittent power.

We evaluate SONIC & TAILS on a TI MSP430 microcontroller [110] using an RF-energy harvester [7, 8] (Secs. Sec. 3.6 & Sec. 3.7). On three real-world DNNs [109, 134, 205], SONIC improves inference efficiency by 6.9× on average over Alpaca [151], a state-of-the-art intermittent system. TAILS exploits DMA and SIMD to further improve efficiency by 12.2× on average.

We conclude with a discussion of the limitations of current energy-harvesting MCUs and the need for new ULP architectures (Sec. 3.8).

3.1. MOTIVATION FOR INTERMITTENT INFERENCE

Many attractive IoT applications will be impractical without intelligence "beyond the edge." Communication is too expensive on these devices for solutions like cloud offloading to be practical. Instead, energy-harvesting devices must decide *locally* how to spend their energy, e.g., when to communicate sensor readings or when to activate an expensive sensor, such as a high-resolution camera.

This section makes the case for inference on energy-harvesting, intermittently operating devices. We show how communication dominates energy, even with state-of-the-art low-power networking, making cloud offloading impractical. We analyze where energy is spent and show that, to a first order, *inference accuracy determines system performance*, motivating the use of DNNs in these applications. Using this analysis we will later compare different DNN configurations and find one that maximizes application performance (Sec. 3.3).

3.1.1 The need for inference beyond the edge

Many applications today offload most computation to the cloud by sending input data to the cloud and waiting for a response. Unfortunately, communication is not free. In fact, on energy-harvesting devices, communication costs orders-of-magnitude more energy than local computation and sensing. These high costs mean that *it is inefficient and impractical for energy-harvesting devices to offload inference to the edge or cloud*, even on today's most efficient network architectures. For example, the recent OpenChirp network architecture lets sensors send data over long distances with extremely low power consumption. To send an eight-byte packet, a terrestrial sensor draws 120mA for around 800ms [74]. Using the recent Capybara energy-harvesting power system [49], such a sensor would require a *900mF* capacitor bank to send a single eight-byte packet. This large capacitor array imposes an effective duty cycle on the device, because the device must idle while charging before it can transmit. A Capybara sensor node with its 2cm × 2cm solar array in direct sunlight (an optimistic setup) would take around 120 seconds to charge a 900mF capacitor bank [49]. Hence, sending a single 28 × 28 image with 1B per pixel (e.g., one MNIST image [135]) to the cloud for inference would take *over an hour*.

In contrast, our full-system Sonic prototype performs inference locally in just 10 seconds operating on weak, harvested RF energy—an improvement of more than 360×. Sonic & Tails thus open the door to entirely new classes of inference-driven applications on energy-harvesting devices.

3.1.2 Why accuracy matters

We now consider an example application to show how inference accuracy determines end-to-end application performance. This analysis motivates the use of state-of-the-art inference techniques, namely DNNs, over less accurate but cheaper techniques like support-vector machines.

To reach these conclusions, we employ a high-level analytical model, where energy in the system is divided between sensing, communication, and inference. (Sensing includes all associated local processing, e.g., to set up the sensor and post-process readings.) We use local inference to filter sensor readings so that only the "interesting" sensor readings are communicated. Our figure of merit is the number of interesting sensor readings that can be sent in a fixed amount of harvested energy (which is also a good proxy for execution time). We denote this as IMpJ, or interesting messages per Joule. Though this metric does not capture the interesting readings that are *not* communicated due to inference error (i.e., false negatives), our analysis demonstrates the need for high accuracy, and hence false negatives are uncommon.

This simple model captures many interesting applications of inference beyond the edge: e.g., wildlife monitoring, disaster recovery, wearables, military, etc. For concreteness, we consider a wildlife-monitoring application where sensors with small cameras are deployed across a wide area with OpenChirp connectivity. These sensors monitor a local population of, say, hedgehogs and send pictures over radio when they are detected. The goal is to capture as many images of hedgehogs as possible, and images without have no value.

Baseline without inference: Our baseline system does not support local inference, so it must communicate every image. Communication is expensive, so this baseline system does not perform well. Suppose sensing costs E_{sense} energy, communicating one sensor reading costs E_{comm} energy, and interesting events occur at a base rate of p (see Table 3.1). Then the baseline system spends $E_{\text{sense}} + E_{\text{comm}}$ energy per event,

Parameter	Description
IMpJ	Our figure of merit, the number of "interesting" messages sent per Joule of harvested energy.
p	Base rate (probability) of "interesting" events.
t_p	True positive rate in inference.
t_n	True negative rate in inference.
E_{sense}	Energy cost of sensing (e.g., taking a photo).
E_{comm}	Energy cost of communicating one sensor reading.
E_{infer}	Energy cost of a inference on one sensor reading.

Table 3.1: Description of each parameter in our energy model.

only p of which are worth communicating, and its IMpJ is:

$$\text{Baseline} = \frac{p}{E_{\text{sense}} + E_{\text{comm}}} \tag{3.1}$$

Ideal: Although impossible to build, an ideal system would communicate only the interesting sensor readings, i.e., a fraction p of all events. Hence, its IMpJ is:

$$\text{Ideal} = \frac{p}{E_{\text{sense}} + p\ E_{\text{comm}}} \tag{3.2}$$

Local inference: Finally, we consider a realistic system with local, imperfect inference. In addition to sensing energy E_{sense}, each sensor reading requires E_{infer} energy to decide whether it is worth communicating. Suppose inference has a true positive rate of t_p and a true negative rate of t_n. Since communication is very expensive, performance suffers from incorrectly communicated, uninteresting sensor readings at a rate of: $(1-p)\,(1-t_n)$. Its IMpJ is:

$$\text{Inference} = \frac{p\ t_p}{(E_{\text{sense}} + E_{\text{infer}}) + (p\ t_p + (1-p)\,(1-t_n))\ \ E_{\text{comm}}} \tag{3.3}$$

Case study: Wildlife monitoring: We now apply this model to the earlier wildlife monitoring example. Hedgehogs are reclusive creatures, so "interesting" photos are rare, say $p = 0.05$. Low-power cameras allow images to be taken at low energy, e.g., $E_{\text{sense}} \approx 10\text{mJ}$ [170]. As we saw above, communicating an image is expensive, taking $E_{\text{comm}} \approx 23{,}000\text{mJ}$ over OpenChirp [74]. Finally, we consider two systems with local inference: a naïve baseline implemented using prior task-based intermittence support (specifically Tile-8 in Sec. 3.4.2) and SONIC & TAILS, our proposed technique. Their inference energies are gathered from our prototype (Sec. 3.6), taking $E_{\text{infer,naïve}} \approx 198\text{mJ}$ and $E_{\text{infer,TAILS}} \approx 26\text{mJ}$, respectively.

Fig. 3.1 shows each system's IMpJ after plugging these numbers into the model. For simplicity, the figure assumes that true positive and negative rates are equal, termed "accuracy". Since communication dominates the energy budget, local inference enables large end-to-end benefits on the order of $1/p = 20\times$. However, for these gains to be realized in practice, inference must be accurate, and the benefits quickly deteriorate as inference accuracy declines. Qualitatively similar results are obtained when p varies, though the magnitude of benefit changes (increasing with smaller p).

This system is dominated by the energy of sending results. Inference is relatively inexpensive, so naïve local inference and SONIC & TAILS perform similarly (though SONIC & TAILS outperforms Naïve by up to 14%). To see the benefits of efficient inference, we must first address the system's communication bottleneck.

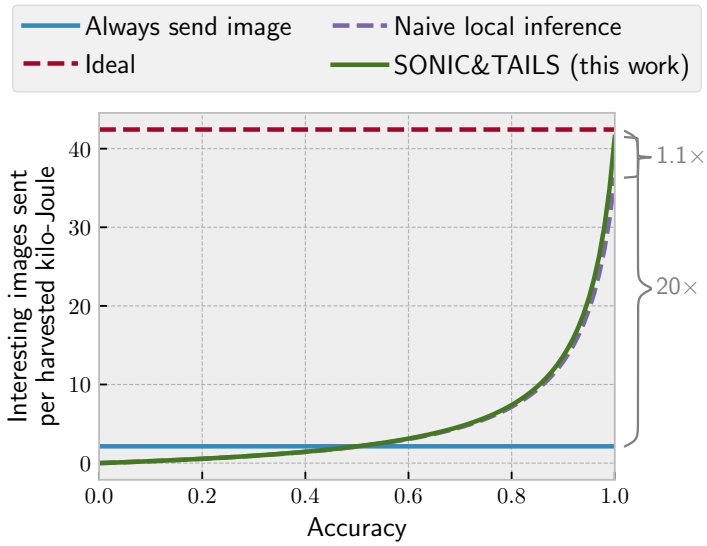


Figure 3.1: Inference accuracy determines end-to-end system performance in an example wildlife monitoring application. Interesting events are rare and communication is expensive; local inference ensures that energy is only spent on interesting events.

Sending only inference results: Depending on the application, even larger end-to-end improvements are possible by sending only the *result* of inference rather than the full sensor reading. For instance, in this wildlife monitoring example, the energy-harvesting device could send a single packet when hedgehogs were detected, rather than the full image. The effect is to significantly decrease E_{comm} for the systems with local inference, mitigating the system's bottleneck. In our wildlife monitoring example, E_{comm} decreases by 98×.

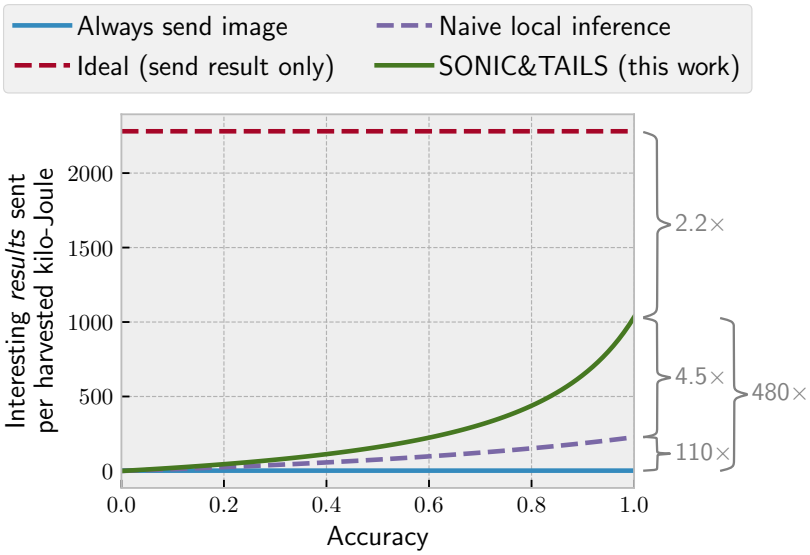


Figure 3.2: Local inference (i.e., Naive and SONIC & TAILS) lets energy-harvesting devices communicate only *results* of inference, enabling dramatic increases in end-to-end system performance.

Fig. 3.2 shows end-to-end performance when only sending inference results. Local inference allows dramatic reductions in communication energy: SONIC & TAILS can detect and communicate 480× more events than the baseline system without local inference. These reductions also mean that inference is a non-negligible energy cost,

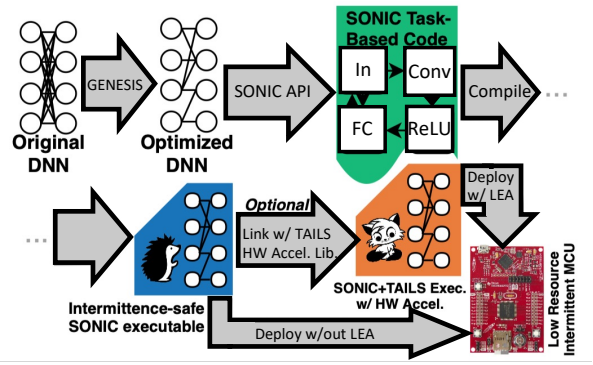


Figure 3.3: Overview of implementing a DNN application using SONIC & TAILS. GENESIS first compresses the network to optimize interesting messages sent per Joule (IMpJ). SONIC & TAILS then ensure correct intermittent execution at high performance.

and *SONIC & TAILS outperform naïve local inference by* 4.6×. Finally, the gap between Ideal and SONIC & TAILS is 2.2×. This gap is difficult to close further on current hardware, but will be addressed in later chapters (Ch. 4, Ch. 5, and Ch. 6).

3.2. SYSTEM OVERVIEW

This thesis describes the first system for performing DNN inference efficiently on intermittently-operating, energy-harvesting devices. Fig. 3.3 shows the new system components in this work and how they produce an efficient, intermittence-safe executable starting from a high-level DNN model description. There are three main components to the system: GENESIS, SONIC, and TAILS.

GENESIS (generating energy-aware networks for efficiensy on intermittent systems) is a tool that automatically optimizes a DNN, starting from a programmer's high-level description of the network. GENESIS attempts to compress each layer of the network using well-known separation and pruning techniques. GENESIS's goal is to *find a network that optimizes IMpJ* while meeting resource constraints. As Fig. 3.3 shows, GENESIS's input is a network description and its output is an optimally compressed network. Sec. 3.3 describes GENESIS.

SONIC (software-only neural intermittent computing) is an intermittence-safe, task-based API and runtime system that includes specialized support for DNN inference that *safely "breaks the rules" of existing task-based systems to improve performance*. SONIC is compatible with existing task-based frameworks [47, 151], allowing seamless integration into larger applications. Sec. 3.4 describes SONIC in detail.

TAILS (tile-accelerated intermittent LEA support) is an alternative to the SONIC runtime library that leverages hardware vector acceleration, specifically targeting the TI Low Energy Accelerator (LEA) [111]. To use TAILS, the programmer need only link their compiled binary to the TAILS-enabled runtime system. This runtime includes all of SONIC's optimizations and a suite of hardware-accelerated vector operations, such as convolutions. Sec. 3.5 describes TAILS in detail.

Starting with a high-level network description, a programmer can use GENESIS, SONIC, and TAILS to build an efficient, intermittent DNN-enabled application that meets resource constraints, is robust to intermittent operation, and leverages widely available hardware acceleration.

3.3. OPTIMAL DNN COMPRESSION WITH GENESIS

The first challenge to overcome in SONIC & TAILS is fitting neural networks into the resource constraints of energy-harvesting systems. In particular, the limited memory

Network	Layer	Uncompressed Size	Compression Technique	Compressed Size	Compression	Accuracy
Image classification (MNIST)	Conv	$20 \times 1 \times 5 \times 5$	HOOI	3×1D Conv	11.4×	99.00%
	Conv	$100 \times 20 \times 5 \times 5$	Pruning	1253	39.9×	
	FC	200×1600	Pruning, SVD	5456	109×	
	FC	500×200	Pruning, SVD	1892	—	
	FC	10×500	—	—	—	
Human activity recognition (HAR)	Conv	$98 \times 3 \times 1 \times 12$	HOOI	3×1D Conv	2.25×	88.0%
	FC	192×2450	Pruning, SVD	10804	58.1×	
	FC	256×192	Pruning, SVD	—	—	
	FC	6×256	—	—	—	
Google keyword spotting (OkG)	Conv	$186 \times 1 \times 98 \times 8$	HOOI, Pruning	3×1D Conv	7.3x	84.0%
	FC	96×1674	Pruning, SVD	16362	11.8×	
	FC	128×96	Pruning, SVD	2070	—	
	FC	32×128	SVD	4096	2×	
	FC	128×32	SVD	4096	—	
	FC	128×12	—	—	—	

Table 3.2: Neural networks used in this paper.

capacity of current microcontrollers imposes a hard constraint on networks. We have developed a tool called GENESIS that automatically explores different configurations of a baseline neural network, applying separation and pruning techniques (Sec. 2.2.1) to reduce the network's resource requirements. GENESIS improves upon these known techniques by optimally balancing inference energy and true positive/negative rates to maximize IMpJ, building on the the model in Sec. 3.1.

3.3.1 Neural networks under consideration

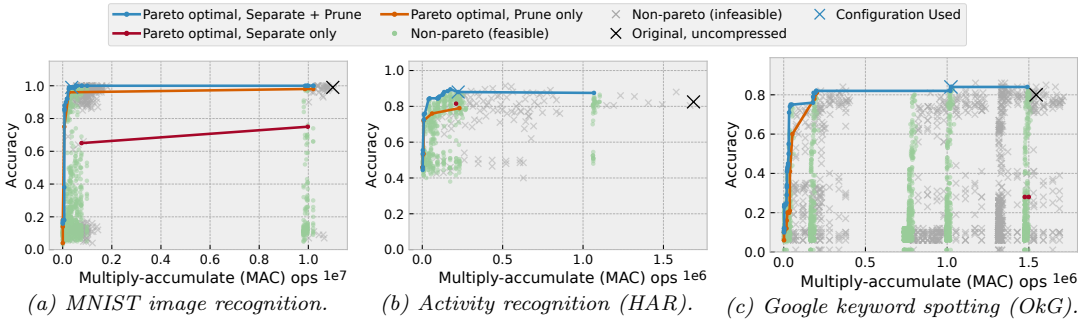
We consider three networks, summarized in Table 3.2. To represent image-based applications (e.g., wildlife monitoring and disaster recovery), we consider MNIST [135]. We consider MNIST instead of ImageNet because ImageNet's large images do not fit in a resource-constrained device's memory. To represent wearable applications, we consider human activity recognition (HAR). HAR classifies activities using accelerometer data [109]. To represent audio applications, we consider Google keyword spotting (OkG) [205], which classifies words in audio snippets.

We also evaluated binary neural networks and several SVM models and found that they perform poorly on current energy-harvesting MCUs. A 99%-accurate binary network for MNIST required 4.4MB of weights [54], exceeding the device's scant memory, and compressing this to 360KB lost nearly 10% accuracy [16]. Likewise, no SVM model that fit on the device was competitive with the DNN models [136]: measured by IMpJ, SVM under-performed by 2× on MNIST and by 8× on HAR, and we could not find an SVM model for OkG that performed anywhere close to the DNN.

3.3.2 Fitting networks on energy-harvesting systems

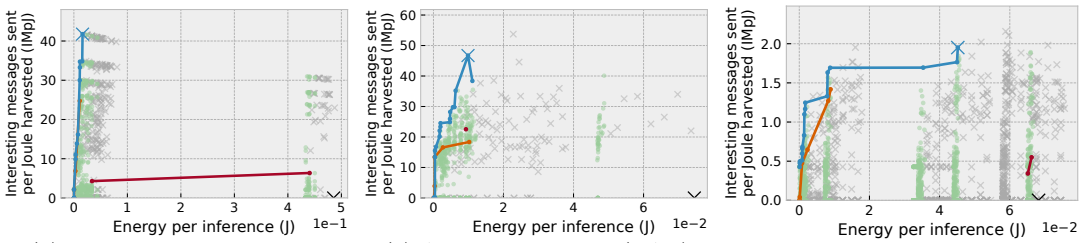
GENESIS evaluates many compressed configurations of a network and builds a Pareto frontier. Compression has trade-offs in four dimensions, difficult to capture with a pareto curve; these include true negative rate, true positive rate, memory size (i.e., parameters), and compute/energy (i.e., operations). Fully-connected layers typically dominate memory, whereas convolutional layers dominate compute. GENESIS compresses both.

GENESIS compresses each layer using two known techniques: separation and pruning. Separation (or rank decomposition) splits an $m \times n$ fully-connected layer into two $m \times k$ and $k \times n$ matrix multiplications, or an $m \times n \times k$ convolutional filter into three $m \times 1 \times 1$, $1 \times n \times 1$, and $1 \times 1 \times k$, filters [29, 45]. GENESIS separates layers



(a) MNIST image recognition. *(b) Activity recognition (HAR).* *(c) Google keyword spotting (OkG).*

Figure 3.4: GENESIS explores the inference accuracy-cost tradeoff for different neural network configurations.



(a) MNIST image recognition. *(b) Activity recognition (HAR).* *(c) Google keyword spotting (OkG).*

Figure 3.5: GENESIS uses our end-to-end application performance model (Eq. 3.3) to select the best feasible network configuration.

using the Tucker tensor decomposition, using the high-order orthogonal iteration algorithm [64, 65, 234]. Pruning involves removing parameters below a given threshold, since they have small impact on results [96, 168].

GENESIS sweeps parameters for both separation and pruning across each layer of the network, re-training the network after compression to improve accuracy. GENESIS relies on the Ray Tune black box optimizer with the Median Stopping Rule to explore the configuration space [87, 166]. Fig. 3.4 shows the results for the networks in Table 3.2. Each marker on the figure represents one compressed configuration, shown by inference accuracy on the y-axis and inference energy on the x-axis. Feasible configurations (i.e., ones that fit in our device's small memory; see Sec. 3.6) are shown as green circles and infeasible configurations are grey $\times$s. Note that the original configuration (large $\times$) is infeasible for all three networks, meaning that they cannot be naïvely ported to the device because their parameters would not fit in memory.

Fig. 3.4 also shows the Pareto frontier for each compression technique. Generally, pruning is more effective than separation, but the techniques are complementary.

3.3.3 Choosing a neural network configuration

GENESIS estimates a configuration's IMpJ using the model from Sec. 3.1, specifically Eq. 3.3. The user specifies E_{sense} and E_{comm} for their application as well as per-compute-operation energy cost. From these parameters, GENESIS estimates E_{infer} for each configuration, and uses the inference accuracy from the prior training step to estimate application performance. The user can specify which class in the training set is "interesting," letting GENESIS compute true positive t_p and negative t_n rates for the specific application.

Fig. 3.5 shows the results by mapping each point in Fig. 3.4 through the model. For these results, we use E_{sense} from Sec. 3.1, per-operation energy from our SONIC & TAILS prototype in Sec. 3.6, and estimate E_{comm} from input size assuming OpenChirp networking [74].

GENESIS chooses the feasible configuration that maximizes estimated end-to-end performance (i.e., IMpJ). Fig. 3.5 shows that this choice is non-trivial. True positive,

true negative, and inference energy affect end-to-end application performance in ways that are difficult to predict. Simply choosing the most accurate configuration, as the twisty blue curve suggests in Fig. 3.5, is insufficient since it may waste too much energy or underperform other configurations on true positive or true negative rates.

3.4. EFFICIENT INTERMITTENT INFERENCE WITH SONIC

SONIC is the first software system optimized for inference on resource-constrained, intermittently operating devices. SONIC supports operations common to most DNN computations, exposing them to the programmer through a simple API. SONIC's functionality is implemented as a group of *tasks* supported by the SONIC runtime system, which is a modified version of the Alpaca runtime system [151]. These tasks implement DNN functionality, and the SONIC runtime system guarantees correct intermittent operation.

Specializing intermittence support for DNN inference yields large benefits. Prior task-based intermittent execution models [47, 151] can degrade performance by up to 19× and by 10× on average (Sec. 3.7). SONIC dramatically reduces these overheads to just 25%–75% over a standard baseline of DNN inference that does not tolerate intermittent operation.

SONIC achieves these gains by eliminating the three major sources of overhead in prior task-based systems: redo-logging, task transitions, and wasted work (Sec. 2.1.2). Our key technique is *loop continuation*, which selectively violates the task abstraction for loop index variables. Loop continuation lets SONIC directly modify loop indices without frequent and expensive saving and restoring. By writing loop indices directly to non-volatile memory, SONIC checkpoints its progress after each loop iteration, eliminating expensive task transitions and wasting work upon power failure.

Loop continuation is safe because SONIC ensures that each loop iteration is idempotent. SONIC ensures idempotence in convolutional and fully-connected layers through *loop-ordered buffering* and *sparse undo-logging*. These two techniques ensure idempotence without statically privatizing or dynamically checkpointing data, avoiding the overheads imposed by prior task-based systems.

3.4.1 The SONIC API

The SONIC API lets the programmer describe a DNN's structure through common linear algebra primitives. Just as a programmer chains tasks together in a task-based intermittent programming model [47, 104, 151], the programmer chains SONIC's tasks together to represent the control and data flow of a DNN inference pipeline. SONIC's API exposes functionality that the programmer invokes like any other task in their program (specifically, a *modular task group* [47, 151]). Though SONIC "breaks the rules" of a typical task-based intermittent system, the programmer does not need to reason about these differences when they are writing a program using the SONIC API. The program-level behavioral guarantee that SONIC provides is the same as the one underlying other task-based intermittent execution models: a SONIC task will execute atomically despite power interruptions by ensuring that repeated, interrupted attempts to execute are idempotent.

3.4.2 The SONIC runtime implementation

DNN inference is dominated by loops within each layer of the neural network. SONIC optimizes DNN inference by ensuring that these loops execute correctly on intermittent power while adding much less overhead than prior task-based systems.

Loops in task-based systems: A typical task-based intermittent system sees two kinds of loops: *short loops* and *long loops*. All iterations of a *short loop* fit in a single task and will complete without consuming more energy than the device can

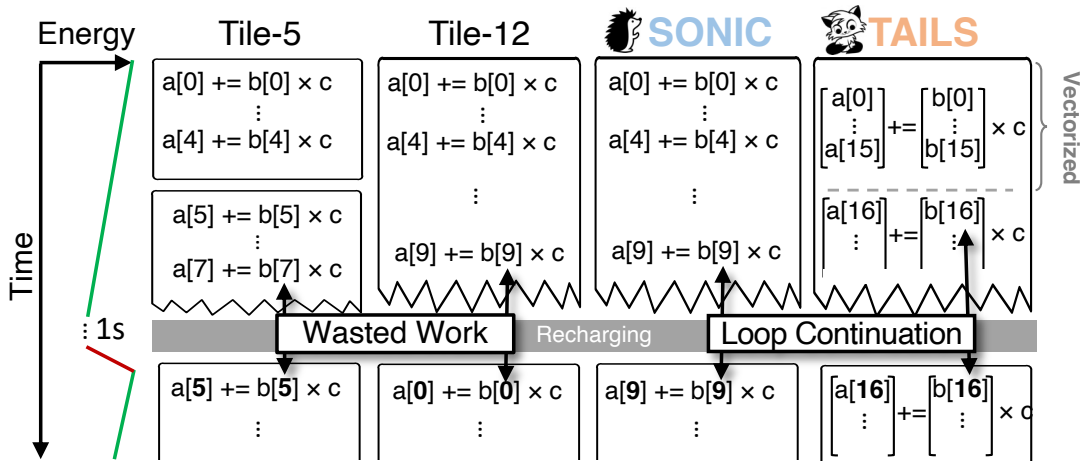


Figure 3.6: Executing a loop using two fixed task-tilings and with SONIC's loop continuation mechanism. Loop continuation avoids the re-execution and non-termination costs of task-tiling. TAILS uses SIMD to perform more work in a fixed energy budget (Sec. 3.5).

buffer. A short loop maintains control state in volatile memory and these variables clear on power failure. When power resumes, the task restarts and completes. Data manipulated by a short loop are usually non-volatile (i.e., "task-shared" [151]) and if read and updated, they must be backed up (either statically or dynamically) to ensure they remain consistent. The problem with short loops is that they always restart from the beginning, wastefully repeating loop work that was already done. In contrast, a *long loop* with many iterations does not fit in a single task; a long loop demands more energy than the device can buffer and may never terminate. A programmer must split loop iterations across tasks, requiring a task transition on each iteration and requiring control state and data to be non-volatile and backed up. The problem with long loops is that may not terminate and, when split across tasks, impose hefty privatization and task transition overheads.

Task-tiling is a simple way to split a loop's iterations into tasks. A task-tiled loop executes a fixed number of iterations per task. Task-tiling amortizes task transitioning overhead, but risks executing more iterations in a single task than the device's energy buffer can support, causing non-termination. Figure 3.6 shows the intermittent execution (energy trace on left) of a loop computing a dot product using two fixed tile sizes of five (Tile-5) and twelve (Tile-12). Tile-5 wastes work when four iterations complete before a failure. Tile-12 prevents forward progress because the device buffers insufficient energy to complete twelve iterations.

Loop continuation

SONIC's *loop continuation* is an intermittence-safe optimization that avoids wasted work, unnecessary data privatization, and task transition overheads in tasks containing long-running loop nests. Loop continuation works by directly modifying loop control variables and memory manipulated in a loop nest, rather than splitting a long-running loop across tasks. Loop continuation permits loops of arbitrary iteration count within a single task, with neither non-termination nor excessive state management overhead. Loop continuation stores a loop's control variables and data manipulated directly in non-volatile memory *without backing either up.* When a loop continuation task restarts, its (volatile) local variables are reinitialized at the task's start. The loop control variables, however, retain their state and the loop continues from the last attempted iteration.

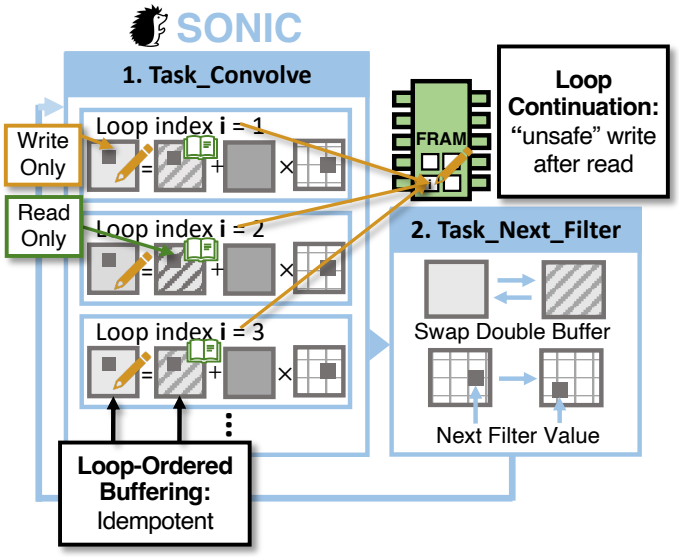


Figure 3.7: SONIC uses *loop continuation* and *loop-ordered buffering* to reduce overheads of correct intermittent execution. *Loop continuation* maximizes the amount of computation done per task by allowing computation to pick up where it left off before power failure.

Fig. 3.7 shows how loop continuation works by storing the loop control state for Task_Convolve in non-volatile memory. SONIC ensures that the loop's control variable `i` is correct by updating it at the end of the iteration and *not resetting it upon re-execution.* A power failure during or after the update to the control variable may require the body of the loop nest to repeat a single iteration, but it never skips an iteration.

Figure 3.6 shows SONIC executing using loop continuation. Despite the power interruption, execution resumes on the ninth loop iteration, rather than restarting the entire loop nest or every fifth iteration like Tile-5 does.

Idempotence tricks

Normally, restarting from the middle of a loop nest could leave manipulated data partially updated and possibly inconsistent. However, loop continuation is safe because SONIC's runtime system ensures each loop iteration is idempotent using either *loop-ordered buffering* or *sparse undo-logging.* SONIC never requires an operation in an iteration to read a value produced by another operation in the same iteration. Thus, an iteration that repeatedly re-executes due to power interruption will always see correct values.

Loop-ordered buffering: Loop-ordered buffering is a double-buffering mechanism used in convolutional layers (and dense fully-connected layers) that ensures each loop iteration is idempotent without expensive redo-logging (cf., [151]). Since the MSP430 devices do not possess sophisticated caching mechanisms, *rather than optimizing for reuse and data locality, SONIC optimizes the number of items needed to commit.* By re-ordering the loops in DNN inference and double-buffering partial activations as needed, SONIC is able to *completely eliminate* commits within a loop iteration.

Evaluating a sparse or dense convolution requires SONIC to apply a filter to a layer's entire input activation matrix. SONIC orders loop iterations to apply each element of the filter to each element of the input activation (i.e., multiplying them) before moving on to the next element of the filter. For idempotence, SONIC writes the partially accumulated value to an intermediate output buffer, rather than applying

updates to the input matrix in-place. After applying a single filter element to each entry in the input and storing the partial result in the intermediate buffer, SONIC swaps the input buffer with the intermediate buffer and moves on to the next filter value.

Since SONIC never reads and then writes to the same memory locations within an iteration, it avoids the WAR problem described in Sec. 2.1.2 and loop iterations are thus idempotent. Fig. 3.7 shows how under loop-ordered buffering, SONIC never reads and writes to the same matrix buffer while computing a partial result in Task_Convolve. After finishing this task, SONIC transitions to Task_Next_Filter, which swaps the buffer pointers and gets the next value to apply from the filter.

Sparse undo-logging: While loop-ordered buffering is sufficient to ensure each loop iteration is idempotent, it is sometimes unnecessarily wasteful. The problem arises because loop-ordered buffering swaps between buffers after every task, so it must copy data between buffers in case it is read in the future—even if the data has not been modified. This copying is wasteful on sparse fully-connected layers, where most filter weights are pruned and thus few activations are modified in a single iteration. With loop-ordered buffering, SONIC ends up spending most of its time and energy copying unmodified activations between buffers.

To eliminate this inefficiency, SONIC introduces *sparse undo-logging* which ensures idempotence through undo-logging instead of double buffering. To ensure atomicity, sparse undo-logging tracks its progress through the loop via two index variables, the *read* and *write* indices. When applying a filter, SONIC first copies the original, unmodified activation into a canonical memory location, and then increments the read index. SONIC then computes the modified activation and writes it back to the original activation buffer (there is no separate output buffer). Then it increments the write index and proceeds to the next iteration. This two-phase approach guarantees correct execution, since sparse undo-logging resumes computing the output value from the buffered original value if power fails in the middle of an update.

Sparse undo-logging ensures that the work per task grows with the number of modifications made, not the size of the output buffer (unlike loop-ordered buffering). However, sparse undo-logging doubles the number of memory writes per modified element, so it is inefficient on dense layers where most data are modified. In those cases, loop-ordered buffering is significantly more efficient. We therefore only use sparse undo-logging in sparse fully-connected layers. Finally, unlike prior task-based systems such as Alpaca, sparse undo-logging ensures idempotence with *constant* space overhead and *no* task transition between iterations.

Related work: Prior work in persistent memory [77] uses techniques similar to our sparse undo-logging. This work is in the high-performance domain, and therefore focuses on cache locality and scheduling cache flushes and barriers. In contrast, our prototype has no caches, and we exploit this fact in loop-ordered buffering to rearrange loops in a way that would destroy cache performance on conventional systems. Moreover, SONIC is more selective than [77], only using undo-logging in sparse fully-connected layers where it outperforms double buffering.

3.5. HARDWARE ACCELERATION WITH TAILS

TAILS improves on SONIC by incorporating widely available hardware acceleration to perform inference even more efficiently. A programmer may optionally link their SONIC application to the TAILS runtime system, enabling the application to use direct-memory access (DMA) hardware to optimize block data movement and to execute operation in parallel using a simple vector accelerator like the TI Low-Energy Accelerator (LEA) [111]. LEA supports finite-impulse-response discrete-time

convolution (FIR DTC), which directly implements the convolutions needed in DNN inference.

Tails's runtime system enables the effective use of LEA in an intermittent system by *adaptively binding hardware parameters at run time to maximize operational throughput without exceeding the device's energy buffer.* Our Tails prototype adaptively determines the DMA block size and LEA vector width based on the number of operations that successfully complete using the device's fixed energy buffer. After calibrating these parameters, Tails uses them to configure available hardware units and execute inference thereafter.

3.5.1 Automatic one-time calibration

Before its first execution, a Tails application runs a short, recursive calibration routine to determine DMA block size and LEA vector size. The routine determines the maximum vector size that it is possible to DMA into LEA's operating buffer, process using FIR DTC, and DMA back to non-volatile memory without exceeding the device's energy buffer and impeding progress. If a tile size does not complete before power fails, the calibration task re-executes, halving the tile size. Calibration ends when a FIR DTC completes and Tails uses that tile size for subsequent computations.

3.5.2 Accelerating inference with LEA

Once Tails determines its tile size, the application runs, using DMA and LEA to compute dense and sparse convolutions and dense matrix multiplications. LEA has limitations: it only supports dense operations and can only read from the device's small 4KB SRAM (not the 256KB FRAM). Tails uses DMA to move inputs into SRAM, invokes LEA, and DMAs the results back to FRAM. Dense layers are natively supported: fully-connected layers use LEA's vector MAC operation, and convolutions use LEA's one-dimensional FIR DTC operation. To support two- and three-dimensional convolutions, Tails iteratively applies one-dimensional convolutions and accumulates those convolutions' results. Tails uses loop-ordered buffering to ensure that updates to the partially accumulated values are idempotent (Fig. 3.4.2).

Sparse operations require more effort. Tails uses LEA for sparse convolutions by first making filters dense (padding with zeros). Making the filters dense is inexpensive because each filter is reused many times, amortizing its creation cost. However, this does mean that LEA performs unnecessary work, which sometimes hurts performance. For this reason, we use LEA's dot-product operation instead of FIR-DTC for $1 \times p \times 1$ factored convolutional layers.

Finally, sparse fully-connected layers are inefficient on LEA because filters do not get reuse. We found that Tails spent most of its time on padding filters, and, despite significant effort, we were unable to accelerate sparse fully-connected layers with LEA. For this reason, Tails performs sparse fully-connected layers in software exactly like Sonic.

3.6. METHODOLOGY

We implement Sonic and Tails on the TI-MSP430FR5994 [110] at 16MHz in the setup in Fig. 3.8. The board is connected to a Powercast P2210B [7] harvester 1m away from a 3W Powercaster transmitter [8]. We ran all configurations on continuous power and on intermittent power with three different capacitor sizes: 1mF, 50mF, and 100µF.

Running code on the device: We compile with MSPGCC 6.4 and use TI's MSP-Driverlib for DMA and TI's DSPLib for LEA. We use GCC instead of Alpaca's LLVM backend because LLVM lacks support for 20-bit addressing and produces slower code for MSP430 than GCC.

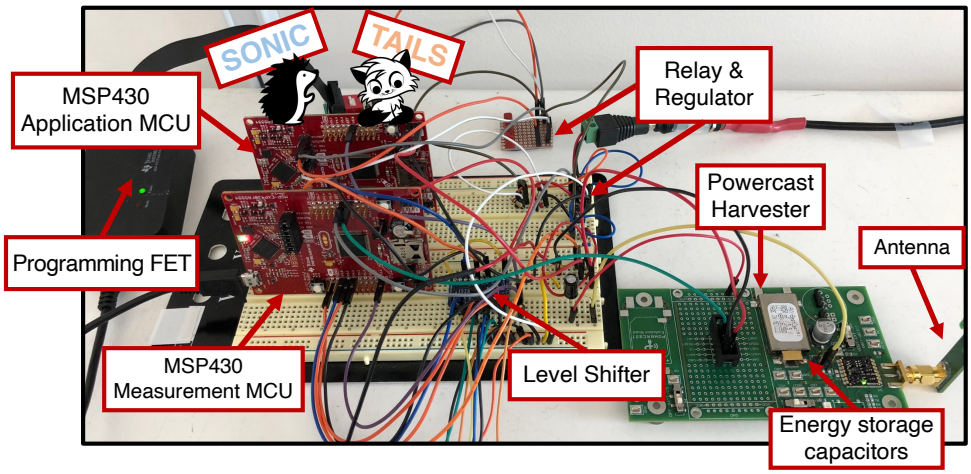


Figure 3.8: Diagram of the measurement setup.

Measurement: We use a second MSP430FR5994 to measure intermittent executions. GPIO pins on the measurement MCU connect through a level-shifter to the intermittent device, allowing it to count reboots and signal when to start and stop timing. We automate measurement with a combination of software and hardware that compiles a configuration binary, flashes the binary to the device, and communicates with the measurement MCU to collect results. The system actuates a relay to switch between continuous power for reprogramming and intermittent power for testing.

Measuring energy: By counting the number of charge cycles between GPIO pulses, we can determine the amount of energy consumed in different code regions. For a more fine-grained approach, we built a suite of microbenchmarks to count how many times a particular operation (e.g., a load from FRAM) can run in single charge cycle. We then profile how many times each operation is invoked during inference and scale by per-operation energy to get a detailed energy breakdown.

Baselines for comparison: We compare SONIC & TAILS to four DNN inference implementations. The first implementation is a standard, baseline implementation that does not tolerate intermittent operation (it does not terminate). The other three implementations are based on Alpaca [151] and split up loops by tiling iterations, as in Fig. 3.6.

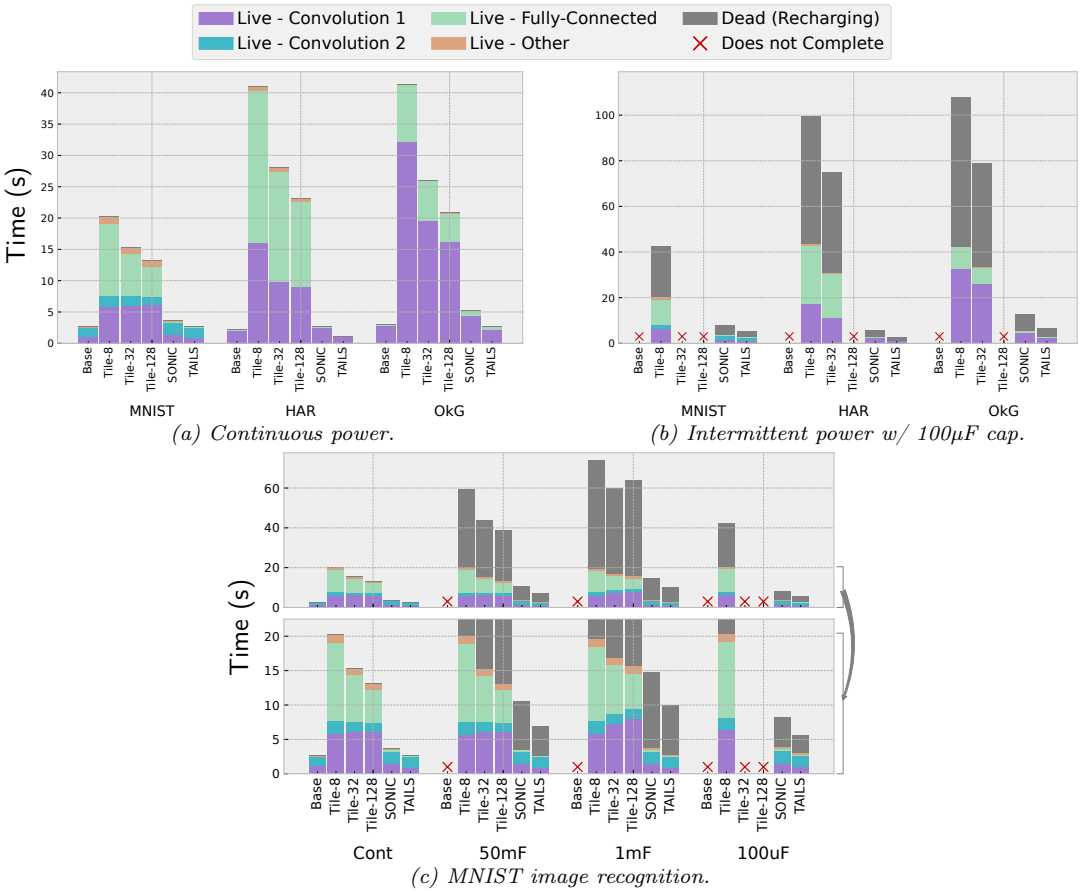
3.7. EVALUATION

We now evaluate our prototype to demonstrate that: *(i)* SONIC & TAILS guarantee correct intermittent execution; *(ii)* SONIC & TAILS greatly reduce inference energy and time over the state-of-the-art; and *(iii)* SONIC & TAILS perform well across a variety of networks without any hand-tuning.

3.7.1 SONIC & TAILS accelerates intermittent inference

Fig. 3.9 show the inference time for the three networks we consider (Table 3.2). For each network, we evaluated six implementations running on four different power systems. We break inference time into: dead time spent recharging; live time spent on each convolution layer (which dominates); live time spent on the fully-connected layers; and everything else.

First, notice that SONIC & TAILS guarantees correct execution for every network on every power system. This is not true of the naïve baseline, which does not run correctly on intermittent power, or of most tilings for prior task-based intermittent systems. The only other implementation that reliably executes correctly is Tile-8, since its tiling is small enough to always complete within a single charge cycle. The other tilings fail



(a) Continuous power.

(b) Intermittent power w/ 100µF cap.

(c) MNIST image recognition.

Figure 3.9: Fig. 3.9a: Three networks on continuous power, where SONIC & TAILS add dramatically lower overheads than prior task-based systems. Fig. 3.9b: Three networks on intermittent power (100µF capacitor), where the baseline and most tiled implementations do not complete. Fig. 3.9c: The MNIST network across all four power systems. SONIC & TAILS always completes and has consistently good performance; HAR and OkG show similar results.

on some configurations: Tile-32 fails on MNIST with a 100µF capacitor, and Tile-128 fails on all networks at 100µF.

SONIC & TAILS guarantee correct execution at much lower overheads than Tile-8. Averaging across networks, Tile-8 is gmean 13.4× slower than the naïve baseline on continuous power, whereas SONIC is 1.45× slower and TAILS is actually 1.2× *faster* than the baseline. That is to say, SONIC improves performance on average by 6.9× over tiled Alpaca [151], and TAILS improves it by 12.2×. Moreover, execution time is consistent across capacitor sizes for SONIC & TAILS.

Larger tile sizes amortize overheads somewhat, but since they do not complete on all networks or capacitor sizes, they are an unattractive implementation choice. SONIC & TAILS guarantee correct intermittent execution across all capacitor sizes, while also being faster than the largest tilings: even compared to Tile-128, SONIC is on average 5.2× faster on continuous power and TAILS is 9.2× faster.

Both DMA and LEA improve TAILS's efficiency. We tested configurations where DMA and LEA are emulated by software and found that LEA consistently improved performance by 1.4×, while DMA improved it by 14% on average.

Ultimately, these results indicate that inference is viable on commodity energy-harvesting devices, and SONIC & TAILS significantly reduce overheads over the state-of-the-art.

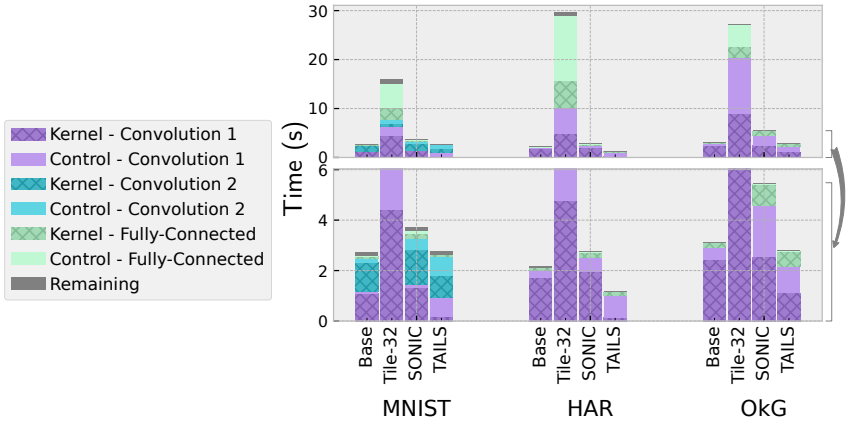


Figure 3.10: Proportions of time spent computing the kernel of a layer. SONIC & TAILS add small overheads over a naïve baseline, unlike prior task-based systems (Tile-32).

3.7.2 Loop continuation nearly eliminate intermittence overheads

Fig. 3.10 shows that the overheads of SONIC & TAILS come mainly from control required to support intermittence. The darker-hatched regions of the bars represent the proportion of time spent computing a layer's kernel (i.e., the main loop), while the lighter regions represent control overheads (i.e., task transitions and setup/teardown). Most of the difference in performance between the baseline and SONIC is attributable to the lighter, control regions. This suggests that SONIC imposes small overhead over the naïve baseline, which accumulates values in registers and avoids memory writes (but does not tolerate intermittence).

TAILS's overhead also comes from control; TAILS significantly accelerates kernels. TAILS's control overhead is large due to LEA's fixed-point representation, which forces TAILS to bit-shift activations before invoking FIR-DTC. Moreover, LEA does not have a left-shift operation (it does have a right-shift), so these shifts must be done in software. These shifts account for most of the control time in Fig. 3.10.

Fig. 3.10 also shows the time breakdown for Tile-32. Unlike SONIC & TAILS, Tile-32 spends significantly more time in both control and the kernel. This is because Alpaca uses redo-logging on all written values to ensure idempotence, so every write requires dynamic buffering (kernel time) and committing when the task completes (control time). SONIC & TAILS effectively eliminate redo-logging, avoiding these overheads.

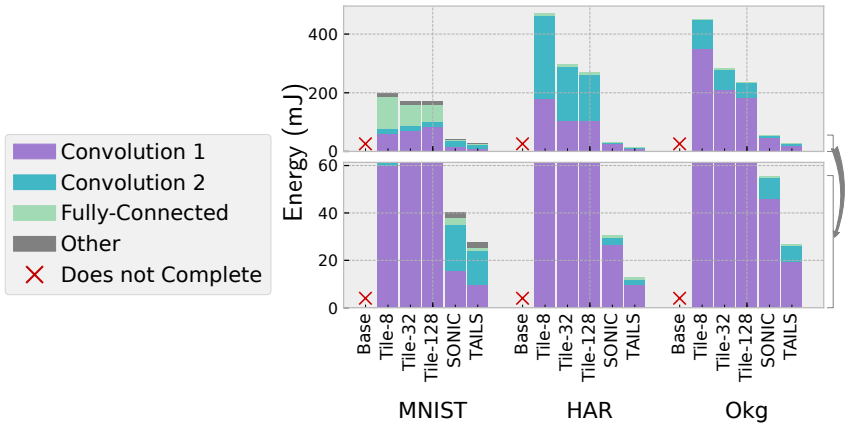


Figure 3.11: Energy of three neural networks with a 1mF capacitor. SONIC & TAILS require substantially less energy than the state-of-the-art.

3.7.3 SONIC & TAILS use much less energy than tiling

Energy-harvesting systems spend a majority of their time powered off recharging, so execution time is largely determined by energy efficiency. Fig. 3.11 shows that SONIC & TAILS achieve high performance because they require less energy than other schemes. Inference energy is in direct proportion to the dead time spent recharging in Fig. 3.9. Since dead time dominates inference time, SONIC & TAILS get similar improvements in inference energy as they do in terms of inference time.

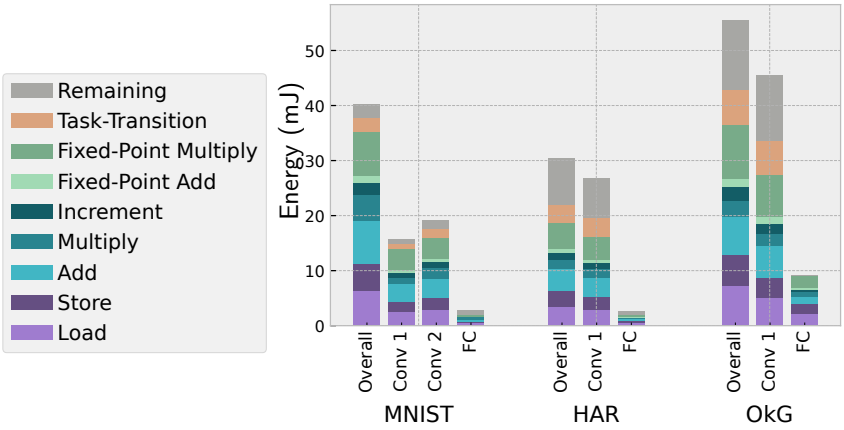


Figure 3.12: Energy profile of SONIC broken down by operation and layer. Multiplication, control, and memory accesses represent significant overheads.

3.7.4 Where does SONIC's energy go?

Fig. 3.12 further characterizes SONIC by showing the proportion of energy spent on different operations. The blue regions represent memory operations, the orange regions are control instructions, the green regions are arithmetic instructions within the kernels, the purple regions are the task-transition overhead, and the grey regions are the remaining, unaccounted-for energy. The control instructions account for 26% of SONIC's energy, and a further 14% of system energy comes from FRAM writes to loop indices. Ideally, these overheads would be amortized across many kernel operations, but doing this requires a more efficient architecture.

3.8. DISCUSSION

This chapter argued that intelligence "beyond the edge" will enable new classes of IoT applications and presented SONIC, the software component to the new ULP sensor system stack. SONIC specializes intermittence support for DNN inference to guarantee correct execution, regardless of power system, while reducing overheads by up to 6.9× and 12.2×, respectively, over the prior state-of-the-art.

However, our experience in building SONIC & TAILS also showed that there is a large opportunity to accelerate inference on ULP sensor devices. But, current microcontrollers for energy-harvesting systems are poorly suited to efficient inference. They are sequential, single-cycle processors, and so spend very little of their energy on "useful work" [107]. For example, by deducting the energy of `nop` instructions from Fig. 3.12, we estimate that SONIC spends at least 40% of its energy on instruction fetch and decode. This cost is a waste in highly structured computations like DNN inference, where overheads easily amortize over many operations.

LEA should bridge this efficiency gap, but unfortunately LEA has many limitations. Invoking LEA is expensive. Each LEA invocation should therefore do as much work as possible, but LEA's parallelism is limited by its small (4KB) SRAM buffer. This small buffer also forces frequent DMA between SRAM and FRAM, which

cannot be overlapped with LEA execution and does not support strided accesses or scatter-gather. LEA also has surprising gaps in its support: it does not support vector left-shift or scalar multiply, forcing TAILS to fall back to software. In software, integer multiplication is a memory-mapped peripheral that takes four instructions and nine cycles. All told, these limitations cause SONIC & TAILS to spend much more energy than necessary. There is ample room to improve inference efficiency via a better architecture – the subject of the next three chapters. Ch. 4 discusses MANIC a ULP vector-dataflow co-processor, Ch. 5 describes SNAFU, a ULP CGRA generation framework and architecture, and Ch. 6 presents RIPTIDE, a dataflow compiler and ULP CGRA. These architectures achieve much higher energy-efficiency v. existing scalar MCUs (like the MSP430) by leveraging vector and dataflow execution that minimize instruction and data supply energies.

Chapter 4

MANIC: An energy-efficient, vector-dataflow co-processor[1]

Tiny, pervasively deployed, ultra-low-power sensor systems enable important new applications in environmental sensing, in- and on-body medical implants, civil infrastructure monitors, and even tiny chip-scale satellites. These applications require a new ULP sensor system stack. Ch. 3 contributed a software component to this new stack, describing how these devices can be made "intelligent" with on-device machine inference using SONIC. However, SONIC also demonstrated that existing systems suffer fundamental inefficiencies that demand new, extremely energy-efficient computer architectures.

Sensing workloads are increasingly sophisticated: Sensor devices are collecting increasingly more data as sensor capability has matured. This increase in sensed data volume requires more sophisticated processing. But as Ch. 3 argued, offloading work to a more powerful edge device or to the cloud is impractical as transmitting data takes much more energy per byte than computing locally. Under these constraints, application performance becomes dependent on the energy-efficiency of computation.

Energy-efficiency is only half the story, though. A computation-heavy sensor system also needs to be highly programmable to support a wide variety of applications. But, programmability and energy-efficiency are in tension, since programmability often carries a significant energy penalty. Our goal is to design a highly programmable architecture that *hides microarchitectural complexity while eliminating the energy costs of programmability.*

Existing low-power architectures fall short: While Ch. 3 showed that it is possible to run sophisticated processing on existing devices, ULP COTS MCUs (e.g., TI MSP430, ARM M0+ & M4+) nonetheless fail to meet the criteria for effective sensor nodes. Their scalar execution models pay a high price for general purpose programmability (see the *COTS MCU* dot in Fig. 4.1), wasting energy fetching & decoding instructions, controlling execution pipeline resources, and supplying data.

Programming pitfalls of architectural specialization: Specialization of a system's control or datapath is one way to reduce the tax of general-purpose programmability, eliminating inessential hardware structures and functions for a particular application. But this comes at the expense of flexibility and generality (see the *ASIC* dot in Fig. 4.1), making a highly specialized design susceptible to quick obsolescence.

Existing programmable, efficient designs are insufficient: In contrast to specialization, another approach to programmable energy-efficiency is to target a conventional vector architecture (such as NVidia's Jetson TX2 [186], ARM NEON [17],

[1] [85] G. Gobieski, A. Nagi, N. Serafin, M. M. Isgenc, N. Beckmann, and B. Lucia, "Manic: A vector-dataflow architecture for ultra-low-power embedded systems," in MICRO, 2019.

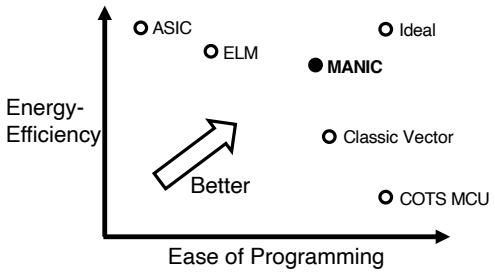


Figure 4.1: MANIC seeks to improve energy efficiency without compromising programmability.

or TI LEA [111]), amortizing the cost of instruction supply across a large number of compute operations. Unfortunately, vector architectures exacerbate the energy costs of RF access, especially in high-throughput designs with multi-ported vector register files (VRFs) [18, 128, 194], and so remain far from the energy-efficiency of fully specialized designs [92] (see the *classic vector* dot in Fig. 4.1).

The ELM architecture stands out among prior efforts as an architecture that targets ultra-low-power operation, operates with extremely high energy-efficiency, and retains general-purpose programmability [23, 24]. The key to ELM's efficiency is an *operand forwarding* network that avoids latching intermediate results and a distributed RF that provides sufficient register storage, while avoiding unfavorable RF energy scaling. Unfortunately, despite these successes, ELM faces fundamental limitations that prevent its widespread adoption. ELM makes significant changes to the architecture and microarchitecture of the system, requiring a full re-write of software to target its exotic, software-managed RF hierarchy and instruction-register design. This programming task requires expert-level assembly hand-coding, as compilers for ELM are unlikely to be simple or efficient; e.g., ELM itself cites a nearly 2× drop in performance when moving from hand-coded assembly to compiler-generated assembly [23]. While ELM supports general-purpose programs, it does so with a high programmability cost and substantial changes to software development tools (as shown in Fig. 4.1).

Our design and contributions: This chapter presents MANIC: an efficient vector-dataflow architecture for ultra-low-power embedded systems. As depicted in Fig. 4.1, MANIC is closest to the *Ideal* design, achieving high energy-efficiency while remaining general-purpose and simple to program. MANIC is simple to program because it exposes a standard vector ISA interface based on the RISC-V vector extension [202].

MANIC achieves high energy-efficiency by eliminating the two main costs of programmability through its vector-dataflow design. First, **vector** execution amortizes instruction supply energy over a large number of operations. Second, MANIC addresses the high cost of VRF accesses through its **dataflow** component by forwarding operands directly between vector operations. MANIC transparently buffers vector outputs in a small forwarding buffer and, at instruction issue, renames vector operands to directly access the forwarding buffer, *eliminating read accesses to the VRF*. Additionally, MANIC extends the vector ISA with **kill annotations** that denote the last use of a vector register, *eliminating write accesses to the VRF*. The vector-dataflow architecture is efficient because MANIC amortizes the energy of tracking dataflow across many vector operations. MANIC thus eliminates a large fraction of VRF accesses (90.1% on average in our experiments) with simple microarchitectural changes that leave the basic vector architecture intact.

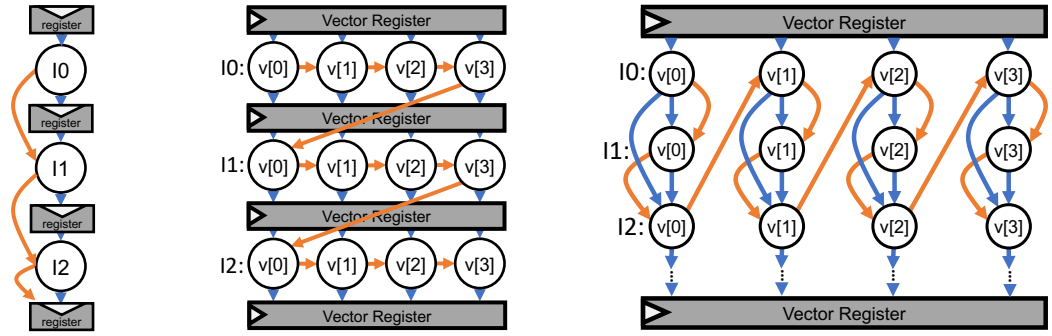
Finally, we have designed and implemented a code scheduling algorithm that exploits MANIC's operand forwarding to minimize VRF energy, while being *microarchitecturally agnostic*. In other words, it is *not* necessary to expose the details of the

pipeline architecture or size of forwarding buffers to minimize VRF energy—a single code schedule is near-optimal across a range of microarchitectural design points.

To evaluate MANIC, we taped-out a test-chip in Intel's 22nm high-threshold voltage process that included MANIC, a scalar design, and a vector design. We measured the energy of the various designs in the test-chip across a collection of programs appropriate to the deeply embedded domain. MANIC reduces energy by 3.4× v. the scalar design and by 12% v. the vector design.

4.1. VECTOR-DATAFLOW EXECUTION

MANIC implements the *vector-dataflow* execution model. There are two main goals of vector-dataflow execution (Fig. 4.1). The first goal is to provide general-purpose programmability. The second goal is to do this while operating efficiently by minimizing instruction and data supply overheads. Vector-dataflow achieves this through three features: *(i)* vector execution, *(ii)* dataflow instruction fusion, and *(iii)* register kill points.



(a) Scalar *(b) Vector* *(c) Vector-dataflow*

Figure 4.2: Different execution models. Orange arrows represent control flow, blue arrows represent dataflow. MANIC relies on vector-dataflow execution, avoiding register accesses by forwarding and renaming.

4.1.1 Vector execution

The first main feature of MANIC's execution model is vector execution. Vector instructions specify an operation that applies to an entire vector of input operands (as in ample prior work discussed in Sec. 2.3.1). The key advantage of vector operation for an ultra-low-power design is that control overheads imposed by each instruction — instruction cache access, fetch, decode, and issue — amortize over the many operands in the vector of inputs. Vector operation dramatically reduces the cost of instruction supply and control, which is a primary energy cost of general-purpose programmability. Vector operation is thus a key ingredient in MANIC's energy-efficiency.

Fig. 4.2 illustrates the difference between scalar execution and vector execution. Fig. 4.2a executes a sequence of instructions in a scalar fashion. Blue arrows show dataflow and orange arrows show control flow. Instructions proceed in sequence and write to and read from the register file to produce and consume outputs and operands. Fig. 4.2b executes the same sequence of instructions in a vector execution. The execution performs the vector instruction's operation on each element of the vector in sequence, consuming operands from and producing outputs to the register for each operation over the entire vector. Control proceeds *horizontally* across each of the vector's elements for a single vector instruction before control transfers *vertically* to the next vector instruction. Vector execution amortizes the control overhead of a scalar execution because a single instruction corresponds to an entire vector worth of operations.

4.1.2 Dataflow instruction fusion

The second main feature of MANIC's execution model is *dataflow instruction fusion*. Dataflow instruction fusion identifies windows of contiguous, dependent vector instructions. Dataflow instruction fusion eliminates register file reads by directly forwarding values between instructions within the window. Comparing to a typical vector machine illustrates the benefit of dataflow instruction fusion. In a typical vector machine, instructions execute independently and each operation performs two vector register file reads and one vector register file write. Accessing the vector register file has an extremely high energy cost that scales poorly with the number of access ports [24,128]. With dataflow instruction fusion, each instruction that receives a forwarded input avoids accessing the expensive vector register file to fetch its input operands. Avoiding these reads reduces the total energy cost of executing a window of vector instructions.

Fig. 4.2c illustrates the difference between vector execution and vector-dataflow execution in MANIC. Vector-dataflow first identifies data dependencies among a sequence of vector instructions in a fixed-size instruction window. After identifying dependences between instructions in the window, MANIC creates an efficient dataflow forwarding path between dependent instructions (using the forwarding mechanism described in Sec. 4.2). Fig. 4.2c shows a window of dependent operations made up of instructions `I0`, `I1`, and `I2`. Execution begins with the first vector instruction in the window (`I0`) and the first element of the vector (`v[0]`). However, unlike a typical vector execution, control transfers *vertically* first, next applying the second vector instruction to the first vector element. The orange arcs illustrate vertical execution of `I0`, then `I1`, then `I2` to the vector inputs represented by `v[0]`. After vertically executing an operation for each instruction in the window for `v[0]`, the orange arcs show that control steps horizontally, executing the same window of operations on the next element of the vector, `v[1]`. The blue arrows illustrate the dataflow forwarding captured by vertical execution in a window of vector-dataflow execution. The blue arrow from `I0` to `I2` shows that the value produced by `I0` is forwarded directly to `I2` without storing the intermediate result in the vector register file.

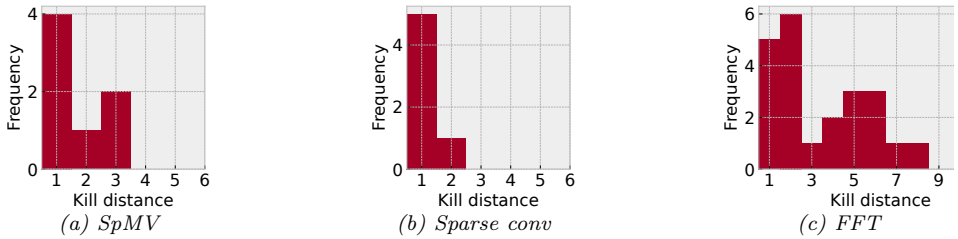
4.1.3 Vector register kill points

The third main feature of MANIC's execution model is its use of *vector register kill points*. A vector register is *dead* at a particular instruction if no subsequent instruction uses the value in that register. Hence, a dead value need not be written to the vector register file. The instruction at which a vector register becomes dead is the *kill point* for that register. Though MANIC forwards values between dependent instructions without going through the vector register file, MANIC normally must write each operand back to the vector register file because the operand may be used in a later window.

However, if a program explicitly informs MANIC of each register's kill points, then MANIC can eliminate register file writes associated with those registers. We propose to tag each of an instruction's operands with an optional *kill bit* that indicates that the register is dead at that instruction, and its value need not be written back to the vector register file. Kill bits do not affect programmability because they are optional, a compiler analysis to identify dead registers is simple, and kill bits do not expose microarchitectural details, such as the size of MANIC's instruction window.

4.1.4 Applications benefit from vector-dataflow

We studied the core compute kernels in a wide variety of sensor node applications and found abundant opportunities for vector-dataflow execution. Regardless of MANIC's window size, an application has more exploitable vector dataflows if its



(a) SpMV *(b) Sparse conv* *(c) FFT*

Figure 4.3: Histograms of kill distances for three different applications. Distances skew left, suggesting values are consumed for the last time shortly after being produced.

sequences of dependent instructions tend to be shorter. The length of a dependent instruction sequence is characterized by the distance (or number of instructions) between a when register's value is produced and when that register is killed (the kill point). We deem this the *kill distance*. Shorter kill distances require fewer resources for forwarding in a window and make a window of any size more effective.

We statically measured the distribution of kill distances for all registers in the inner loops of three kernels. The histograms shown in Fig. 4.3 suggest that kill distances tend to be short and that a reasonably small (and thus implementable) window size would capture dependencies for these kernels.

4.1.5 Synchronization and memory consistency

In MANIC, the vector unit runs as a loosely-coupled co-processor with the scalar core. As a result, MANIC must synchronize vector and scalar execution to ensure a consistent memory state. A typical sequentially consistent model would require frequent stalls in the scalar core to disambiguate memory and, worse, would limit the opportunity for forwarding in the vector unit. These issues could be avoided with microarchitectural speculation, including load-store disambiguation and mis-speculation recovery mechanisms, but we judge such mechanisms too expensive for ultra-low-power applications. Moreover, in practice, the scalar core and the vector unit rarely touch the same memory during compute-intensive program phases, so the mechanisms would be largely unused.

Instead, we add a new `vfence` instruction that handles both synchronization and memory consistency. `vfence` stalls the scalar core until the vector unit completes execution with its current window of vector-dataflow operations. MANIC's use of `vfence` operations is very similar to memory fences for concurrency in x86, ARM, and other widely commercially available processors [80]. Properly used, `vfence` operations cause the scalar and vector cores' executions to be sequentially consistent. In practice, this often means inserting a `vfence` at the end of the kernel.

As with any system relying on fences, the programmer is responsible for their correct use (i.e., avoiding data races). Relying on the programmer to avoid data races is practical since compilers struggle with alias analysis, reasonable because `vfences` are rare, and consistent with common practice in architectures and high-level programming languages [28, 112].

4.2. MANIC ARCHITECTURE

MANIC is a processor microarchitecture that implements the vector-dataflow execution model to improve energy efficiency while maintaining programmability and generality. MANIC's hardware/software interface is a recent revision of the RISC-V ISA vector extension [202]. MANIC adds a vector unit with a single lane to a simple, in-order scalar processor core. The vector unit has a few simple additions to support vector-dataflow execution: instruction windowing hardware and a renaming

mechanism together implement forwarding between dependent instructions. With no modifications to the ISA, MANIC runs programs efficiently. With a minor ISA change, MANIC further improves efficiency by conveying register kill annotations; the microarchitecture uses these annotations to kill registers instead of incurring the cost of writing them to the vector register file.

4.2.1 Vector ISA

The software interface to MANIC's vector execution engine is the RISC-V ISA vector extension [202] and RISC-V code[1] will run efficiently on a MANIC system with only minor modifications to add `vfence` instructions for synchronization and memory consistency.

A programmer may further optionally recompile their code using our custom MANIC compiler to use minor ISA changes that support code scheduling and vector register kill annotations. We emphasize that these compiler-based features do not require programming changes, do not expose microarchitectural details, and are optional to the effective use of MANIC.

MANIC implements the RISC-V V vector extension. RISC-V V does not specify a fixed number of vector registers, but its register name encoding includes five bits for vector register names. We implement 16 vector registers, requiring four bits to name, and leaving a single bit in the register name unused. We use the extra bit in a register's name to convey kill annotations from the compiler to the microarchitecture. If either of an instruction's input registers has its high-order bit set, the encoded instruction indicates to MANIC that the register dies at the instruction. To support code scheduling, MANIC's optional compiler support runs the dataflow code scheduling algorithm (described in Sec. 4.2.5). After scheduling, the compiler analyzes definitions and uses of each register and adds a kill annotation to a killed register's name in the instruction at which it dies.

4.2.2 Microarchitecture

MANIC's microarchitecture is split along the two phases of execution: Decode & Rename and Execute. During the Decode & Rename phase, MANIC buffers a window of decoded instructions ("Insn Buffer") and identifies dataflow between them, renaming operands to point to the "Forwarding Buffer" as dataflow allows. Then during the Execute phase, MANIC cycles through the instruction buffer ("VIssue"), determines the source of each instruction operand ("VGate"), executes the operation ("VExecute" and "VMemory"), and performs a write back if necessary ("VWriteback"). Fig. 4.4 shows the two phases of execution as well as the different microarchitectural components of each.

Decode & Rename

Decode & Rename is responsible for creating a window of instructions to execute according to vector-dataflow. The Decode & Rename logic activates once per window of instructions, identifying, preparing, and issuing for execution a window of dependent instructions over an entire vector of inputs. The logic analyzes a short sequence of instructions that has the same number of instructions as the instruction buffer can hold. Decode & Rename identifies dataflow between instructions by comparing the names of their input and output operands. If two instructions are dependent — the output of one of the instructions is the input of another — MANIC should forward the output value directly from its producer to the input of the consumer, avoiding the register file. MANIC's rename logic implements forwarding by renaming the instructions' register operands to refer to a free location in MANIC's forwarding

[1] Specifically, RISC-V E extension, which uses 16 architectural registers.

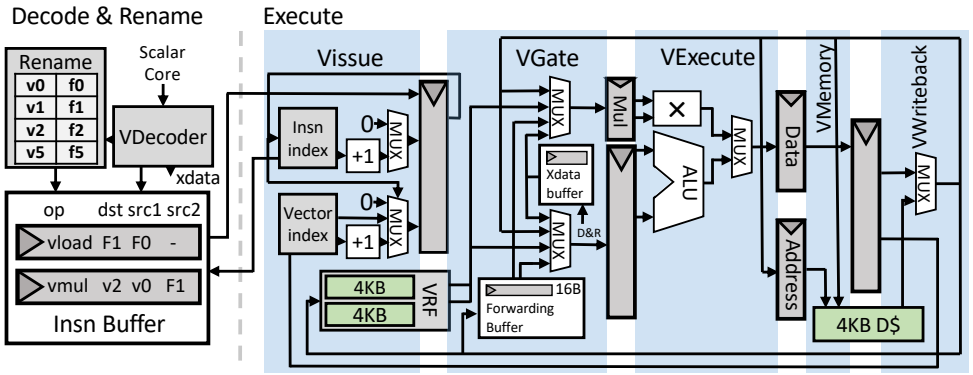


Figure 4.4: A block diagram of MANIC's microarchitectural components. MANIC operates in two phases, Decode & Rename and execute. Decode & rename fills in the instruction buffer, identifying opportunities for dataflow forwarding. Execute cycles through the instruction buffer to compute across instructions (vertical) and then across vectors (horizontal).

buffer, instead of to the register file. The Decode & Rename logic records the renaming in MANIC's renaming table, which is a fixed-size, directly-indexed table, with one entry for each register that can be renamed in a window of instructions. After Decode & Rename identifies dependent operations and performs renaming for the window, it dispatches the window of operations for execution.

Instruction buffer: MANIC uses its instruction buffer to store a decoded window of instructions that have had their register operands renamed by the Decode & Rename logic. Each entry of the buffer tells the Execute phase where to read and write an operand. For input operands, the instruction buffer controls whether to fetch an operand from the vector register file, from the Xdata buffer (data buffered from the scalar core like base address or stride), or from MANIC's forwarding buffer (in the case of an operand being forwarded between instructions in the window). Likewise, for output operands, the instruction buffer controls whether to write an output operand to the vector register file, to the forwarding buffer, or to both.

Limits to window size: There are several classes of instructions that limit window size. These include stores, permutations, and reductions. Permutations and reductions require interactions between elements in a vector, which creates a *horizontal* dependence between operations on different vector elements. MANIC does not support forwarding for such operations because of the complexity of the dependence tracking that they introduce. Instead, these operations execute one element at a time, ultimately writing to the vector register file.

A store also ends the decoding and renaming of a window. A store may write to a memory location that a later operation loads from. Such a through-memory dependence is unknown until execution time. Consequently, MANIC conservatively assumes that the address of any store may alias with the address of any load or store in the window (i.e., in a later vector element). A store ends the construction of a window to avoid the need for dynamic memory disambiguation to detect and avoid the effect of such aliasing. We evaluated adding a *non-aliasing store* instruction that would allow MANIC to forward past stores, but this instruction improved energy-efficiency by less than 0.5% in our applications. This is because store instructions often naturally close windows (e.g. a `vfence` follows the store to ensure correctness). Thus, given the added programming complexity for minimum benefit, we conclude that such an instruction is unnecessary.

Xdata buffer: Some instructions like vector loads and stores require extra information (e.g. base address and stride) available from the scalar register file when the instruction is decoded. Due to the loosely coupled nature of MANIC, this extra information must be buffered alongside the vector instruction. Since not all vector instructions require values from the scalar register file, MANIC includes a separate buffer, called the xdata buffer, to hold this extra information. Entries in the instruction buffer contain indices into the xdata buffer as needed. During execution, MANIC uses these indices to read information from the xdata buffer and execute accordingly.

Structural hazards: There are two structural hazards that cause MANIC to stop buffering additional instructions, stall the scalar core, and start vector execution. The first hazard occurs when the instruction buffer is full and another vector instruction is waiting to be buffered. The second hazard occurs when the xdata buffer is full and a decoded vector instruction requires a slot. The prevalence of each hazard depends on the size of the buffers associated with each. The first hazard is most common, while the second is rare.

Execute

The Execute phase begins once a `vfence` instruction is reached or there is a structural hazard. MANIC has a five-stage execution pipeline consisting of: VIssue, VGate, VExecute, VMemory, and VWriteback. VIssue tracks execution progress, maintains a pointer into the instruction buffer, reads decoded instructions, and (only if necessary) initiates VRF reads; VGate determines the source for each operand (the VRF, Xdata buffer, Forwarding buffer, or bypass paths) and steers operands to the multiplier or ALU; VExecute computes the ALU and multiplier results; VMemory issues loads and stores; and VWriteback writes results to the Forwarding Buffer or VRF, as appropriate.

VIssue: VIssue determines what the execution pipeline should execute next and initiates VRF reads if they are necessary (no dataflow identified during Decode & Rename). It maintains an instruction pointer into the instruction buffer for the current instruction as well as counter representing the completed vector length. Together these track the progress of execution. Execution proceeds first vertically through the entire window. VIssue bumps the instruction pointer for each entry in the instruction buffer, reconfiguring the pipeline according to each instruction. Then execution proceeds horizontally; VIssue resets the instruction pointer to the top of the instruction buffer and increments the completed vector length counter. When the completed vector length counter matches the vector length of the computation and the instruction pointer is at the end of the instruction buffer, execution is finished.

VGate: VGate chooses the source for each operand. Possible sources include the VRF, the Xdata buffer, bypass paths or the Forwarding buffer. VGate reduces switching activity in VExecute by steering operands to dedicated input registers for the ALU or multiplier to, e.g., prevent a VADD from toggling the multiplier. This is important because, unlike conventional vector execution, the active instruction changes every cycle in MANIC, increasing activity on control and data signals.

Forwarding buffer: The forwarding buffer stores intermediate values as MANIC's execution unit forwards them to dependent instructions in the instruction window. The buffer has a single read port, single write port, and an single entry (32b×16) for each vector register. It is simple (1r1w) and small (64B in total), which corresponds to a very low static power and access energy compared to the very high static power and access energy of the vector register file. By accessing the forwarding buffer instead of

accessing the vector register file, an instruction with one or more forwarded operands consumes less energy than one that executes without MANIC.

Efficient reductions: RISC-V V contains reduction instructions like `vredsum v1 v2`, which adds up all elements of `v2` and writes the sum into the first element of `v1`. MANIC supports these operations efficiently without accessing the VRF by accumulating partial results in a single 32b reduction register. This is possible because windows close on reductions so there will only ever be a single reduction per window. The Decode & Rename logic recognizes a reduction, and remaps the second source operand and the destination to point to the reduction register. During the Execute phase, MANIC will then use the partial result in the reduction register as one source for the reduction (e.g., sum) and overwrite it with the new value as it is produced. This optimization re-purposes MANIC's existing dataflow mechanisms to save an entire vector-length of VRF reads and writes for reductions.

4.2.3 Memory system

MANIC's memory subsystem includes an instruction cache (icache) and a data cache (dcache). This departs from the designs of many commercial microcontrollers in the ultra-low-power computing domain, which do not have dcaches and have extremely small icaches on the order of 64 bytes [110]. However, we find that even small or moderately sized dcaches (512B) are effective in minimizing the number of accesses to main memory. We measured miss curves for the different application we consider; for each application there is an extreme drop-off in the number of misses for even small cache sizes, and with a 512B cache the curves are basically flat. Since the energy of an access to main memory dwarfs an access to the dcache, the dcache offers a significant reduction in energy.

Caching and intermittence: In the intermittent computing domain, improperly managed caches may lead to memory corruption because dirty data may be lost when power fails. As such, MANIC assumes a hardware-software JIT-checkpointing mechanism (like [25, 114, 152]) for protecting the caches and any dirty data. Checkpointing energy for cached data is virtually negligible because caches are very small relative to the operating period.

4.2.4 Putting it together with an example

We illustrate the operation of the Decode & Rename logic, renaming table, instruction buffer, and forwarding buffer with an example of MANIC's operation, shown in Fig. 4.5. The figure starts with vector-aware assembly code that MANIC transforms into vector-dataflow operations by populating the renaming table and instruction buffer with information about the dataflow. Vector assembly instructions pass into MANIC's microarchitectural mechanisms as they decode and later execute.

Decoding instructions and renaming operands: The figure shows a three-instruction program and illustrates how the Decode & Rename logic populates the instruction buffer and remaps registers for each instruction.

- `vload`: The rename logic records the load in the instruction window and, since the instruction is a vector load and requires a base address, also inserts the base address (`&a` forwarded from the scalar register file) into the xdata buffer. In addition, the logic writes an empty renaming entry to `v0` in the renaming table along with the index of the instruction in the instruction buffer. An empty renaming entry at execution time signifies a vector register write. However, during Decode & Rename, an empty entry may be filled by an instruction added to the instruction window later during the same Decode & Rename phase.

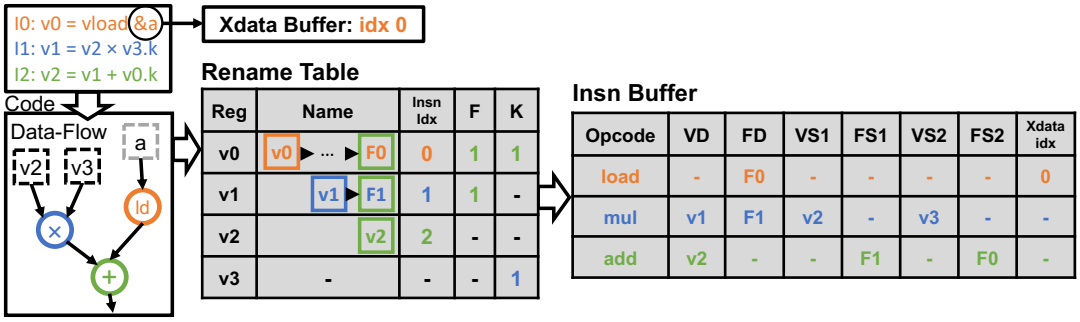


Figure 4.5: MANIC's Decode & Rename logic constructs windows of instructions with dataflow. The rename table keeps track of registers and names, updating the instruction buffer when new opportunities for forwarding are identified.

- `vmul`: The multiply instruction consumes two register operands that are not in the renaming table and, at execution time, will issue two vector register file reads. As with the load, the rename logic records the multiply's output register with an empty entry in the renaming table as well as the index of the multiply in the instruction buffer.

- `vadd`: The add's inputs are `v0` and `v1` with the kill annotation indicating that the instruction kills register `v0`. The rename logic looks up each input operand in the renaming table and, finding both have valid entries, identifies this instruction as the target for forwarding. The rename logic remaps `v0` to refer to the first entry of the forwarding buffer and `v1` to the second position. The load instruction in the instruction buffer (found by the saved index in the renaming table) is updated and will store its result in `F0` instead of `v0`. Similarly, the multiply instruction is also updated and will store its result in `F1`, but since `v1` is not killed, it will still be written-back to the register file. The add instruction then will fetch its input operands from `F0` and `F1` instead of the vector register file. The kill annotations associated with `v3` and `v0` follow the re-written instructions into the instruction window, enabling their use during execution to avoid register file writes.

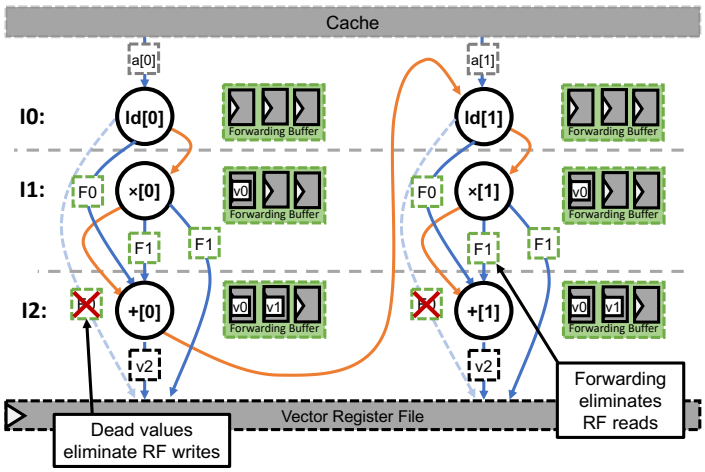


Figure 4.6: MANIC's microarchitecture components execute a window of instructions using forwarding according to dataflow across an entire vector of input.

Executing a window of instructions: After Decode & Rename, the window of instructions is ready to execute. Fig. 4.6 shows (via the orange control-flow arcs) how MANIC executes the entire window vertically for a single vector element before moving on to execute the entire window for the second vector element. The blue dataflow arcs show how MANIC forwards values between dependent instructions using its forwarding buffer. The green squares marked with "F" names are forwarded values. The figure also shows how MANIC uses a kill annotation at runtime. The registers with kill annotations (`v0` and `v3`) need not be written to the vector register file when the window completes executing, sparing the execution two vector register writes required by a typical vector execution.

4.2.5 Microarchitecture-agnostic dataflow scheduling

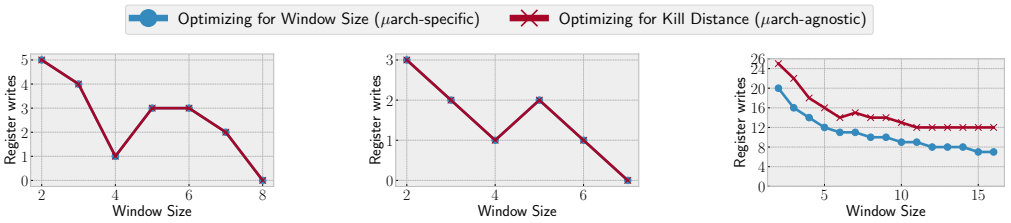
MANIC's final feature is *microarchitecture-agnostic dataflow scheduling*. This feature is optional compiler support that re-orders vector instructions to make dependent operations as close as possible to one another. If dependent operations are closer together in an instruction sequence, then it is more likely that they will appear together in one of MANIC's vector-dataflow windows. By re-ordering operations to appear close together in a window, MANIC creates more opportunities to forward values from a producer instruction to its consumer, eliminating more vector register file accesses.

MANIC's dataflow scheduler does not compromise programmability or generality. The programmer need not understand the microarchitecture to reap the benefits of the dataflow scheduler. The dataflow scheduler minimizes the forwarding distance between dependent instructions, rather than targeting a particular window size. While not always optimal for a given window size, this microarchitec-ture-agnostic optimization prevents the compiler from being brittle or dependent on the microarchitectural parameters of a particular system.

To minimize forwarding distance between dependent instructions, MANIC's dataflow code scheduler uses *sum kill distance*. A vector register's kill distance is the number of instructions between when an instruction defines the register and when the value in the register is used for the last time (i.e., the register dies). The sum kill distance is the sum of all registers' kill distances across the entire program. To remain agnostic to the window size of particular MANIC implementation, the code scheduler minimizes the sum kill distance (which is equivalent to minimizing average kill distance). Sum kill distance is a proxy for the number of register writes in a program because if a register does not die during a window's execution, the system must write its value back to the register file. When sequences of dependent instructions are closer together, their intermediate values die more quickly, because registers need not remain live waiting for unrelated instructions to execute. A larger window accommodates dependence chains that include longer kill distances.

We implement dataflow code scheduling using brute force (exhaustive) search for small kernels containing fewer than 12 vector operations. For larger kernels (e.g., FFT), we implement dataflow code scheduling via simulated annealing that randomly mutates instruction schedules, while preserving dependences, to produce a new valid schedule, accepting this new schedule with some probability.

Fig. 4.7 shows that the microarchitecture-agnostic minimization of the sum kill distance closely approximates a microarchitecture-specific approach that optimizes for a particular window size. The plot shows the number of register writes made by one iteration of the kernel's inner loop for a given window size using code optimized by the two different optimization criteria. The blue line shows the number of register writes of a *microarchitecture-specific* schedule, where window size is exposed to the compiler. The red line shows the number of writes for our *microarchitecture-agnostic*



(a) Sparse matrix multiply *(b) Sparse convolution* *(c) Fast-fourier transform*

Figure 4.7: Code scheduling is microarchitecturally agnostic – minimizing the sum of kill distances is good proxy for minimizing register writes for specific window size.

schedule based on sum kill distance. The two curves generally agree, suggesting that minimizing sum kill distance eliminates register writes with similar efficacy as when window size is exposed explicitly to the compiler. For the FFT kernel, the instruction window is broken by stores and permutations (Sec. 4.2.2), causing additional vector register file writes. This is a limitation of optimizing only for sum kill distance and could be fixed by taking into account these operations during optimization.

4.3. MANIC-SILICON

To evaluate MANIC, we taped-out a silicon prototype. The chip, MANIC-SILICON, has two objectives: 1) be a viable replacement for existing MCUs in ULP sensor deployments and 2) validate vector-dataflow execution against competing execution models. Towards the first objective, MANIC-SILICON meets the criteria for remote ULP sensor deployments; specifically, the testchip operates at extreme low-power, supports general-purpose programs, can run standalone (without an external FPGA or MCU driving control), has I^2C and GPIO to communicate with sensors and integrates a non-volatile main memory, a requirement for devices that may suffer (frequent) power failures. Towards the second objective, the MANIC-SILICON testchip includes several independent designs that implement different execution models for comparison, including scalar, vector, and vector-dataflow.

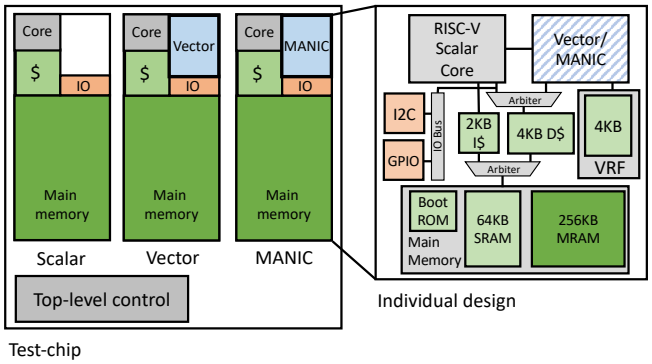


Figure 4.8: Block diagrams of test-chip and a single design. The chip includes scalar, vector, and MANIC designs. Each design is isolated (dedicated power rails) and can run standalone.

4.3.1 Chip design

MANIC-SILICON includes three different designs, shown in Fig. 4.8. They include a scalar design, an optimized vector design, and MANIC. Logic at the top-level controls which design is active and arbitrates chip IO among the designs. The designs share a common core consisting of: a RISC-V (RV32emi) microcontroller, a 2-KB instruction cache, a 4-KB data cache, a module that handles I^2C communication and programming, a module that handles GPIO communication, a module that tracks

statistics about device operation, and main memory composed of 64KB of SRAM, 1KB of ROM, and 256KB of embedded MRAM (eMRAM). The scalar design adds no additional components to this common core. The vector design adds a simple, single-lane vector co-processor. The co-processor has a three-stage pipeline (VIssue, VExecute, VWriteback) that computes a vector operation by iterating over vector elements. The MANIC design adds the MANIC co-processor. Each design serves as comparison point of execution model: the scalar design is similar to existing commodity devices, while the vector design implement vector execution, providing the closest competition to MANIC's vector-dataflow execution.

Why eMRAM?: Each of the designs on MANIC-SILICON include 256KB of embedded MRAM. This is a requirement for remote deployments where energy is sparse and a device may suffer frequent power failures. MANIC-SILICON is also one of the first demonstrations of eMRAM integrated into a complete system. eMRAM provides non-volatility at lower costs than competing NVM technologies. Compared to flash, eMRAM has word-level addressibility, higher write endurance, lower read and write latencies, and lower read and write energies. Also since it can be fabricated in the same process as logic the use of eMRAM avoids expensive off-chip IO.

Power isolation: MANIC-SILICON was designed so that power and energy could be measured of each individual design separately. MANIC-SILICON has separate power domains for IO, SRAM, eMRAM, and logic. Further, there is a separate SRAM and logic power domains for each design and the eMRAM modules can be power-gated on disabled designs, isolating the module of the active design. This separation allows for near-complete isolation of an active design, which is critical to accurately determining the energy consumption of that design.

4.3.2 Verification and bring-up of MANIC-SILICON

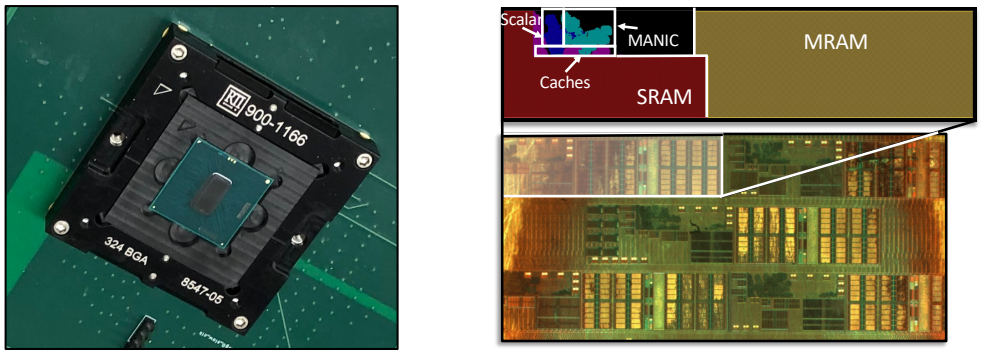
Verification of chip design is the most important step in the tape-out process. For MANIC-SILICON, it involved three items: 1) integration of design-for-test (DFT) structures to allow for easy debugging, 2) safeguards in the event certain features did not work and 3) comprehensive unit and integration testing at each level of hardware abstraction (e.g pre- and post- synthesis and post-place-and-route). The following paragraphs describe the design-for-test strategy and chip bring-up in more detail.

Design-for-test: DFT is a design methodology that makes a design robust to feature failure. For MANIC-SILICON, we developed a series of standard DFT modules with scan-chain interfaces that wrap selectively-chosen registers, SRAM macros, and the eMRAM macro. This allows each of these structures to be read and written to directly while interacting minimally with on-chip logic. There are two important examples of note in MANIC-SILICON. First, entire features can be disabled and signals routed around them; e.g. the instruction and data caches can both be disabled. Second, there are five mechanisms for programming the chip. The first two mechanisms rely on the ROM-based bootloader. If eMRAM is enabled program code and state is stored in the eMRAM otherwise its stored in the SRAM. The second two mechanisms use the scan-chain interfaces to the SRAM and eMRAM macros. Program code and state can be written directly to these macros and the bootloader directed to start execution immediately (without flashing) from either the SRAM or eMRAM. Finally, in the worst case scenario there is also a mechanism for feeding the RISC-V core directly with instructions and data for loads. These built-in safeguards are not only important for debugging, but also protect chip operation from a number of problematic scenarios when features might have failed.

Programming and communication with the chip: To program and communicate with the MANIC-SILICON prototype, we co-developed an Arduino-based programmer with MANIC-SILICON's bootloader using an FPGA implementation of MANIC-SILICON. The Arduino-based programmer converts serial commands to I^2C commands, passing data to and from the computer to the testchip. Programming is initiated by the testchip which asks the programmer for the application size. The programmer then communicates with the computer to get the application's binary size and responds to the testchip over I^2C. Then the testchip repeatedly asks the programmer for bytes of the application binary. The programmer receives this data from the computer and responds to the testchip. There is a handshaking protocol between both the programmer and the computer as well as the programmer and the testchip. This ensures data is not lost during flashing/programming. Once all data has been transferred, the testchip jumps to the starting address in main memory. The programmer continues to be connected, handling further communication (primarily printing to console) between the testchip and the computer.

Tuning the eMRAM: Besides programming, the other important item for chip bring-up is the tuning of the embedded MRAM macro. Due to manufacturing variability, configuration of the eMRAM is different chip-to-chip. As such there are a number of different settings and parameters to get the eMRAM macro reading and writing correct data as well as minimizing write latency. We built a tool to quickly sweep the configuration space (using the scan-chain interface with the eMRAM) to determine the optimal settings for a particular instance of the macro.

4.4. METHODOLOGY



(a) Package. *(b) Electron micrograph.*

Figure 4.9: Fig. 4.9a shows the MANIC test-chip. Fig. 4.9b shows an electron micrograph of delidded chip and highlights components of MANIC design.

MANIC-SILICON was fabricated in a 22nm bulk finFET process using high-threshold voltages standard cells. Fig. 4.9 shows the die photo of the 4mm × 8mm testchip; the MANIC design has an area of $0.57mm^2$. MANIC-SILICON is optimized to run with a 4MHz to 50MHz clock from an on-die clock generator at 0.4V to 1.0V logic, 0.4V to 1.0V SRAM, and 1.10V MRAM. All results are reported at the minimum energy point with the clock at 4MHz and logic and SRAM at 0.4V.

Benchmarks: We evaluate each design (scalar, vector, and MANIC) of MANIC-SILICON across ten benchmarks with random 32b inputs. For the vector design and MANIC, we vectorize a plain-C implementation of each benchmark. We further optimize the benchmarks for MANIC by inserting kill annotations and optimizing the code schedule for sum of kill distance.

Measuring energy: Energy consumption is the primary metric of interest. We measure energy by measuring current and the time it takes complete each benchmark. To

	2017 [122]	2018 [118]	2019 [32]	2020 [199]	This work
Architecture	Scalar & Vector	Scalar	Scalar	Scalar w/ SIMD ext.	Scalar & Vector-dataflow
ISA	RISC-V	Thumb-2	Thumb-2	Thumb-2	RISC-V
Process (nm)	28 FD-SOI	14 Tri-gate	28 FD-SOI	65 LP	22 bulk FF
Core Area (mm^2)	1.07	6.25	0.675	6	0.57
Voltage (V)	0.48-1.0	0.4-1.0	0.4	0.4-0.75	0.4-1.05 Core 1.1 MRAM
Frequency (MHz)	20-797	0.2-950	40-80	0.8-38	4-48.9
Memory (KB)	56 SRAM	64+64+384 SRAM	32+32 SRAM	128 ROM, 16+4 SRAM	64 SRAM, 256 MRAM
Power Budget (mW)	1-200	1-20	1	1-4	0.019-2 w/o MRAM 1-2 w/ MRAM
Average Power (µW)	50000	80	144	47	19.1 w/o MRAM @ 4MHz[1] 1.7mW w/ MRAM @ 49MHz[1]
Peak Efficiency (MOPS/mW)[2]	41.8 MFlops/mW	Not reported	97	Not reported	256 w/o MRAM 33.2 w/ MRAM
Best Active Energy (pJ/Cycle)	Not reported	6.2	3	10.9	3.7 w/o MRAM 29 w/ MRAM

[1] Over all benchmarks [2] 32b operations

Table 4.1: MANIC v. prior work. MANIC is 2.6× more efficient than prior work, achieving 256 MOPS/mW (@19μW & 4MHz).

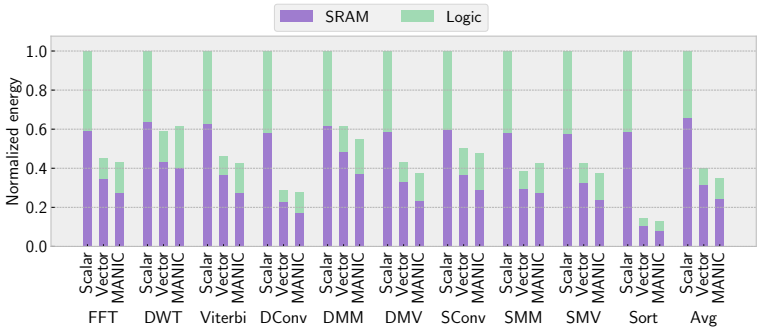
measure current for logic and SRAM, a digital multimeter (Agilent 34405a, Agilent 34410a) is wired in series with the DC power supply (Agilent E4638A). For MRAM, a source meter unit (Keithley 2401) is used. Timing information (i.e. program start and program end) are transmitted from the chip via I^2C to an Arduino and then to a computer.

4.5. EVALUATION

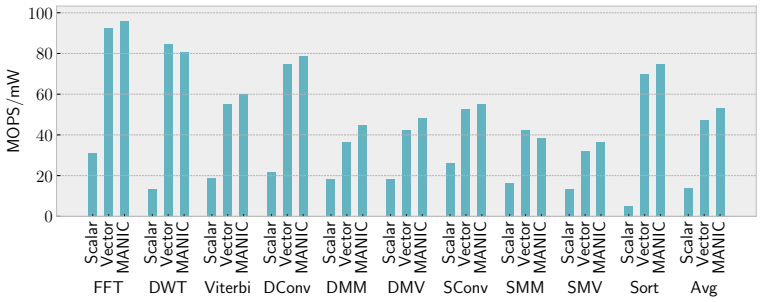
We now evaluate MANIC-SILICON and specifically the MANIC design to 1) demonstrate it's viability as an ULP MCU, 2) to show that it is more energy-efficient than prior state-of-the-art designs, and 3) to validate the vector-dataflow execution model against scalar and vector models. MANIC draws just 19.1μW (@ 4MHz), uses 2.6× less energy than the prior state-of-the-art MCU, and improves efficiency by 3.4× v. scalar and by 12% v. vector.

MANIC is energy-efficient: Table 4.1 compares MANIC with prior work [32, 118, 122, 199]. MANIC was designed for energy-minimal, low-power operation: MANIC consumes 19µW at 4MHz, significantly lower than prior work. MANIC is more energy-efficient than prior work (by 2.6×), with a peak efficiency of 256 MOPS/mW (vector increment, 32b ops) and 3.7pJ/cycle at 0.4V, 4MHz, room temperature, and MRAM disabled. With random inputs, which cause unrealistic, near worst-case toggling of data lines, MANIC gets 45 MOPs/mW on dense matrix-matrix multiplication (DMM).

Vector-dataflow uses less energy than scalar & vector: Fig. 4.10 shows the energy (normalized to scalar) and energy efficiency of the scalar, vector, and MANIC designs. On average, MANIC's vector-dataflow execution reduces energy (and likewise increases energy-efficiency) by 3.4× v. scalar and by 12% v. vector design. FFT and Sort are particularly good benchmarks for MANIC v. the scalar baseline. MANIC achieves 92MOPS/mW and 75MOPS/mW on FFT and Sort respectively v. just 32MOPS/mW and 5MOPS/mW for the scalar design. This shows the benefits of vector execution; in particular, the vectorized implementations of FFT and Sort use different algorithms (for FFT, vectorized FFT v. Cooley-Tukey and for Sort, radix sort v. merge sort) than the scalar baseline such that they can take advantage of longer vectors.



(a) Energy (normalized to scalar).



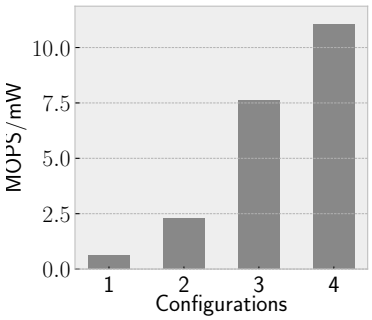
(b) Energy efficiency.

Figure 4.10: Energy (normalized to scalar) and energy-efficiency of scalar, vector, and MANIC designs across ten benchmarks.

Why is MANIC not even more efficient?: Although MANIC does reduce energy on average v. the vector baseline, the reduction is only 12%. This is because, while MANIC effectively reduces VRF accesses (purple decreases in Fig. 4.10b), it reconfigures the execution pipeline every cycle as it iterates through entries in the instruction buffer, executing a single vector element across the instruction window. This toggles control and data signals, which burns energy (green increases in Fig. 4.10b) and cancels out some of the gains from reducing VRF accesses. Sec. 4.6 will go into more detail and later chapters (Ch. 5 and Ch. 6) will present work that specifically addresses this problem.

Size (KB)	256
Area (mm^2)	0.31
Voltage (V)	1.1
Leakage (µW)	663
32b Read Latency @ 50 MHz (ns)	170
32b Write Latency @ 50 MHz (µs)	8.4
32b Read Energy (pJ)	437
32b Write Energy (nJ)	29.7
Read Energy (pJ/bit)	13.7
Write Energy (pJ/bit)	929

(a) MRAM characterization.



(b) MOPS/mW for DMM.

1: Running from MRAM, DCache enabled, 48.9 MHz, 0.64V Core
2: Running from MRAM, DCache disabled, 231 MHz, 1.0V Core
3: Running from SRAM, MRAM enabled, DCache enabled, 48.9 MHz, 0.64V Core
4: Running from SRAM, MRAM enabled, DCache disabled, 166 MHz, 1.0V Core

Figure 4.11: MRAM characterization & case study on DMM.

MRAM characterization: Fig. 4.11 characterizes the embedded MRAM and presents a case study of designs with MRAM enabled. MRAM leakage is 663µW, reads take

170ns and 13.7pJ/bit, while writes take 8.4µs and 929pJ/bit. Write latency is independent of clock frequency. A case study of DMM puts these numbers into context. The figure includes several system configurations: 1) Manic running out of MRAM with the DCache enabled @49MHz, 2) Manic running out of MRAM as fast as possible @231MHz (this necessitates the DCache being disabled), 3) Manic running from SRAM, DCache enabled, and MRAM enabled @49MHz, and 4) Manic running as fast as possible @166MHz (w/o DCache) and MRAM enabled. Configuration 4 achieves max efficiency with 11MOPS/mW and configuration 2 achieves max efficiency for running from MRAM with 2.3MOPS/mW. As found in prior low-power systems, MRAM's high static power is a significant challenge for energy efficiency. There are possible architectural (e.g. caching) and VLSI techniques (e.g. fine-grain power gating) that could address this challenge as part of future work.

4.6. DISCUSSION

Towards the new ULP sensor system stack, Manic contributes new ULP computer architecture and Manic-Silicon contributes a silicon prototype. There are four key takeaways from our experience building these systems. First, Manic-Silicon emphasizes even more the inefficiency of existing commodity MCUs. Manic draws two orders of magnitude less power than an MSP430 (19μW v. 3-5mW) and achieves much better performance (2.5×) v. a similar scalar design. Second, Manic-Silicon validates the vector-dataflow execution model by demonstrating real energy improvements v. scalar and vector baselines.

Third, taping-out Manic-Silicon demonstrated the importance of building real systems. Initially we built Manic in RTL, but without compiled memories. Instead we counted memory accesses and used Destiny [196] to estimate read and write energies of SRAMs of various sizes. This overestimated the improvement that eliminating VRF accesses would yield (38% of simulated system v. 12% for real system).

Finally fourth, Manic-Silicon exposed low-level effects of vector-dataflow execution that would have be missed by high-level models. Specifically, Manic only narrowly outperforms the vector baseline because Manic's implementation of vector-dataflow execution leads to higher switching activity of shared pipeline resources. In the vector baseline, pipeline resources remain configured in the same way throughout execution of a vector instruction. Further data operands tend to be similar across vector elements of the same instruction. Now compare this to Manic's implementation of vector-dataflow execution. Manic iterates over the entries in the instruction buffer reconfiguring its execution pipeline every cycle as it computes a single element across the window of instructions. Not only does this toggle control signals, but it can also lead to additional toggling of data signals as operands of different instructions might not be similar. For example, one instruction could be operating on benchmark data, while another operates on addresses; addresses are not similar (bits set) to benchmark data so executing these operations back-to-back may toggle many data signals.

Manic's high switching activity is solved by Snafu (Ch. 5) and RipTide (Ch. 6). RipTide and Snafu are ULP CGRAs that leverage spatial dataflow to minimize switching activity by dedicating resources to each operation for the duration of a kernel's execution.

Chapter 5

SNAFU generates ULP CGRAs[1]

The opportunity for tiny, ULP devices is enormous [147], but enabling sophisticated processing on these devices remains challenging. Prior chapters have presented progress on this challenge, contributing to a new ULP sensor system stack. Ch. 3 described software to enable machine inference on commodity MCUs and Ch. 4 proposed MANIC and MANIC-SILICON, a new computer architecture and silicon prototype, respectively. MANIC is an energy-efficient vector-dataflow co-processor that is a big improvement over COTS devices. However, it stills fall short due to high switching activity in the shared execution pipeline, a significant inefficiency at ULP-scale. Eliminating these overheads can reduce energy by nearly half, proving that, despite their low operating power, MANIC and other existing programmable ULP designs [61, 98, 173, 244] are not energy-minimal. Alternatively, custom ASICs would achieve energy-minimality, but come at high upfront cost [211] and with severely limited application scope, risking quick obsolesce. Thus, there is still a need for new architectures that achieve ULP (<1 mW), *energy-minimal* operation while maintaining a high degree of design flexibility and ease of programmability.

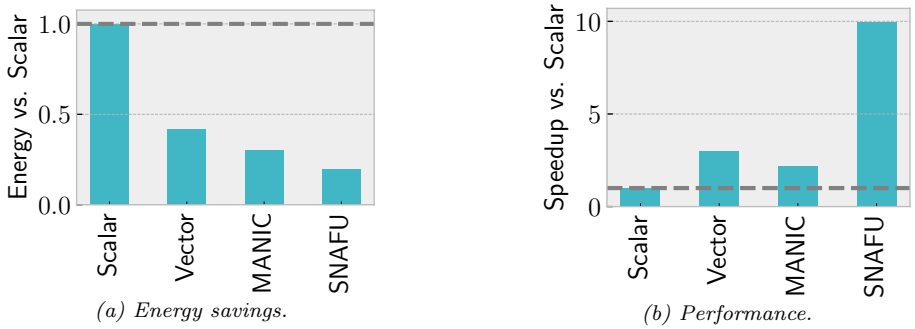
Ultra-low-power CGRAs are the answer: This chapter presents SNAFU,[2] a framework to generate ULP, energy-minimal coarse-grain reconfigurable arrays (CGRAs). SNAFU CGRAs execute in a *spatial vector-dataflow* fashion, mapping a dataflow graph (DFG) spatially across a fabric of processing elements (PEs), applying the same DFG to many input data values, and routing intermediate values directly from producers to consumers. The insight is that spatial vector-dataflow minimizes instruction and data-movement energy, just like MANIC, but also eliminates unnecessary switching activity because operations do not share execution hardware.

The major difference from most prior CGRAs [76, 86, 88, 119, 162, 182, 198, 208, 225, 240, 241, 247, 248] is the extreme design point — SNAFU operates at *orders-of-magnitude lower energy and power budget*, demanding an exclusive focus on energy-minimal design. SNAFU is designed from the ground up to minimize energy, even at the cost of area or performance. For example, SNAFU schedules only one operation per PE, which minimizes switching activity (energy) but increases the number of PEs needed (area). As a result of such design choices, SNAFU comes within 2.6× of ASIC energy efficiency while remaining fully programmable.

SNAFU generates ULP CGRAs from a high-level description of available PEs and the fabric topology. SNAFU defines a standard PE interface that lets designers *"bring your own function unit"* and easily integrate it into a ULP CGRA, along with a library of common PEs. The SNAFU framework schedules operation execution and routes intermediate values to dependent operations while consuming minimal energy.

[1] [81] G. Gobieski, A. O. Atli, K. Mai, B. Lucia, and N. Beckmann, "SNAFU: an ultra-low-power, energy-minimal CGRA-generation framework and architecture," in 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2021, pp. 1027–1040.

[2] Simple Network of Arbitrary Functional Units.



(a) Energy savings. *(b) Performance.*

Figure 5.1: SNAFU-ARCH's energy and performance normalized to a scalar baseline. On average, SNAFU uses 81% less energy and is 9.9× faster, or 41% less energy and 4.4× faster than MANIC.

SNAFU is easy to use: it includes a compiler that maps vectorized C-code to efficient CGRA bitstreams, and it reduces design effort of tape-out via top-down synthesis of CGRAs.

Contributions: This chapter contributes the following:

- We present SNAFU, the first flexible CGRA-generator for ULP, energy-minimal systems. SNAFU makes it easy to integrate new functional units, compile programs to energy-efficient bitstreams, and produce tape-out-ready hardware.
- We discuss the key design choices in SNAFU that minimize energy: scheduling at most one operation per PE; asynchronous dataflow without tag-token matching; statically routed, bufferless, multi-hop NoC; and producer-side buffering of intermediate values.
- We describe SNAFU-ARCH, a complete ULP system-on-chip with a CGRA fabric, RISC-V scalar core, and memory. We implement SNAFU-ARCH in an industrial sub-28 nm FinFET process with compiled memories. SNAFU-ARCH operates at <1 mW at 50 MHz. SNAFU-ARCH reduces energy by 81% v. a scalar core and 41% v. MANIC; and improves performance by 9.9× v. a scalar core and 4.4× v. MANIC.
- Finally, we quantify the cost of programmability through three comprehensive case studies that compare SNAFU-ARCH against fixed-function ASIC designs. We find that programmability comes at relatively low cost: on average, SNAFU-ARCH takes 2.6× more energy and 2.1× more time than an ASIC for the same workload. We break down SNAFU-ARCH's energy in detail, showing that it is possible to close the gap further while retaining significant general-purpose programmability. These results call into question the need for extreme specialization in most ULP deployments.

5.1. OVERVIEW

SNAFU is a framework for generating energy-minimal, ULP CGRAs and compiling applications to run efficiently on them. SNAFU-ARCH is a complete ULP system featuring a CGRA generated by SNAFU, a scalar core, and memory.

SNAFU is a flexible ULP CGRA generator: SNAFU is a general and flexible framework for converting a high-level description of a CGRA to valid RTL and ultimately to ULP hardware. Fig. 5.2 shows SNAFU's workflow. SNAFU takes two inputs: a library of processing elements (PEs) and a high-level description of the CGRA topology. SNAFU lets designers customize the ULP CGRA via a *"bring your own functional unit"* approach, defining a generic PE interface that makes it easy to add custom logic to a generated CGRA.

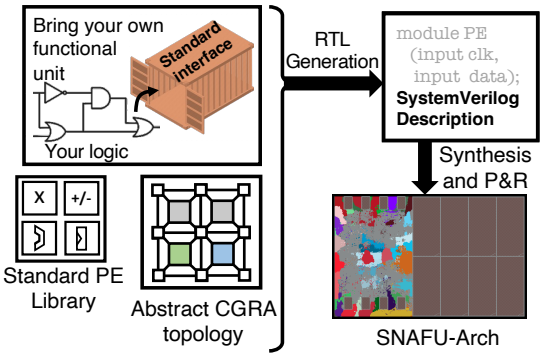


Figure 5.2: Overview of SNAFU. SNAFU is a flexible framework for generating ULP CGRAs. It takes a *bring-your-own functional unit* approach, allowing the designer to easily integrate custom logic tailored for specific domains.

With these inputs, SNAFU generates complete RTL for the CGRA. This RTL includes a statically routed, bufferless, multi-hop on-chip network parameterized by the topology description. It also includes hardware to handle variable-latency timing and asynchronous dataflow firing. Finally, SNAFU simplifies hardware generation by supporting top-down synthesis, making it easy to go from a high-level CGRA description to a placed-and-routed ULP design ready for tape out.

SNAFU-ARCH is a complete ULP, CGRA-based system: SNAFU-ARCH is a specific, complete system implementation that includes a CGRA generated by SNAFU. The CGRA is a 6×6 mesh topology composed of PEs from SNAFU's standard PE library. SNAFU-ARCH integrates the CGRA fabric with a scalar core and 256 KB of on-chip SRAM main memory. The resulting system executes vectorized, RISC-V programs [202] with the generality of software and extremely low power consumption (≈300 µW). Compared to the RISC-V scalar core, SNAFU-ARCH uses 81% less energy for equal work and is 9.9× faster. Compared to MANIC (a state-of-the-art general-purpose ULP design), SNAFU-ARCH uses 41% less energy and is 4.4× faster. Compared to hand-coded ASICs, SNAFU-ARCH uses 2.6× more energy and is 2.1× slower.

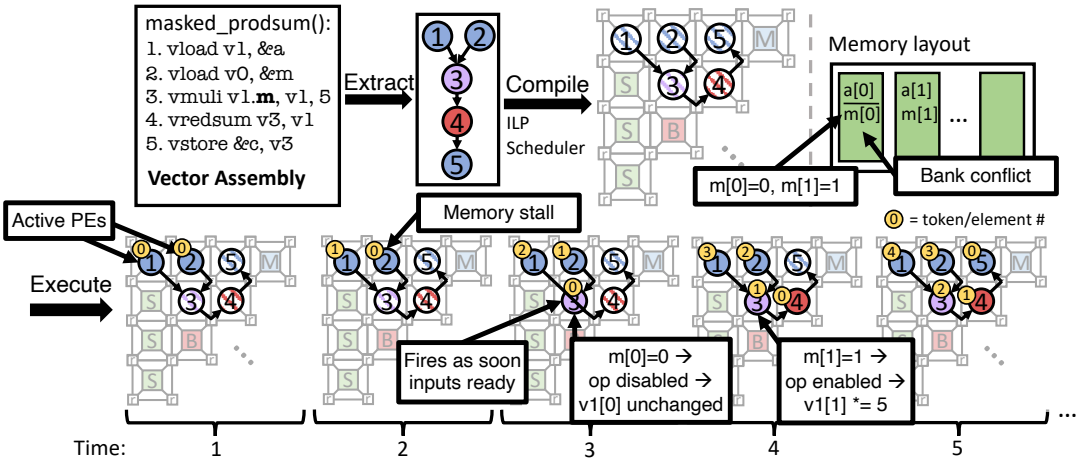


Figure 5.3: An example execution on a SNAFU CGRA fabric. The DFG is extracted from vectorized C-code, compiled to a bitstream, and then executed according to asynchronous dataflow firing.

Example of SNAFU in action: Fig. 5.3 shows the workflow to take a simple vectorized kernel and execute it on an ULP CGRA generated by SNAFU. This kernel

multiplies values at address `&a` by 5 for the elements where the mask `m` is set, sums the result, and stores it to address `&c`. SNAFU's compiler extracts the dataflow from the kernel source code and generates a bitstream to configure the CGRA fabric. The scalar core configures the CGRA fabric and kicks off fabric execution using three new instructions (`vcfg`, `vtfr`, `vfence`), after which the CGRA runs autonomously in SIMD fashion over arbitrarily many input data values. The fabric executes the kernel using asynchronous dataflow firing:

① In the first timestep, the two memory PEs (that load `a[0]` and `m[0]`) are enabled and issue loads. The rest of the fabric is idle because it has no valid input values.

② The load for `a[0]` completes, but `m[0]` cannot due to a bank conflict. This causes a stall, which is handled transparently by SNAFU's scheduling logic and bufferless NoC. Meanwhile, the load of `a[1]` begins.

③ As soon as the load for `m[0]` completes, the multiply operation can fire because both of its inputs have arrived. But `m[0] == 0`, meaning the multiply is disabled, so `a[0]` passes through transparently. The load of `a[1]` completes, and loads for `a[2]` and `m[1]` begin.

④ When the predicated multiply completes, its result is consumed by the fourth PE, which keeps a partial sum of the products. The preceding PEs continue executing in pipelined fashion, multiplying `a[1]` × `5` (since `m[1] == 1`) and loading `a[3]` and `m[2]`.

⑤ Finally, a value arrives at the fifth PE, and is stored back to memory in `c[0]`. Execution continues in this fashion until all elements of `a` and `m` have been processed and a final result has been stored back to memory...

The next three sections describe SNAFU. Sec. 5.2 describes the SNAFU ULP CGRA-generator. Sec. 5.3 describes how SNAFU minimizes energy. And Sec. 5.4 describes SNAFU-ARCH.

5.2. DESIGNING SNAFU TO MAXIMIZE FLEXIBILITY

SNAFU is designed to generate CGRAs that minimize energy, maximize extensibility, and simplify programming. For the architect, SNAFU automates synthesis from the top down and provides a *"bring your own functional unit"* interface, allowing easy integration of custom FUs into a CGRA. For the application programmer, SNAFU is designed to efficiently support SIMD execution of vectorized RISC-V C-code, using a custom compiler that targets the generated CGRA.

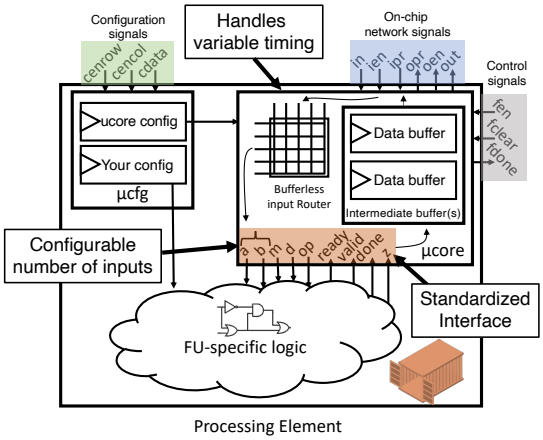


Figure 5.4: SNAFU provides a standardized interface that makes integrating new types of PEs trivial. SNAFU handles variable-latency logic, PE-specific configuration, and the sending and receiving of values from the on-chip network.

5.2.1 Bring your own functional unit (BYOFU)

SNAFU has a generic PE microarchitecture that exposes a standard interface, enabling easy integration of custom functional units (FUs) into the CGRA. If a custom FU implements SNAFU's interface, then SNAFU generates hardware to automatically handle configuring the FU, tracking FU and overall CGRA progress, and moderating its communication with other PEs. There are few limitations on the sort of the logic that SNAFU can integrate. SNAFU's interface is designed to support variable latency and currently supports up to four inputs, but could be easily extended for more. The PE can have any number of additional ports and contain any amount of internal state.

Fig. 5.4 shows the microarchitecture of a generic SNAFU processing element, comprising two components: µcore and µcfg. The µcore handles progress tracking, predicated execution, and communication. The standard FU interface (highlighted orange) connects the µcore to the custom FU logic. The µcfg handles (re-)configuration of both the µcore and FUs.

Communication: The µcore handles communication between the processing element and the NoC, decoupling the NoC from the FU. The µcore is made up of an input router, logic that tracks when operands are ready, and a few buffers for intermediate values. The input router handles incoming connections, notifying the internal µcore logic of the availability of valid data and predicates. The intermediate buffers hold output data produced by the FU. Before an FU (that produces output) fires, the µcore first allocates space in the intermediate buffers. Then, when the FU completes, its output data is written to the allotted space, unless the predicate value is not set, in which case a fallback value is passed through (see below). Finally, the buffer is freed when all consumers have finished using the value. These intermediate buffers are the *only data buffering in the fabric*, outside of internal FU state. The NoC, which forwards data to dependent PEs, is entirely bufferless.

The FU interface: SNAFU uses a standard FU interface for interaction between a PE's µcore and FU. The interface has four control signals and several data signals. The four controls signals are `op`, `ready`, `valid`, and `done`; the µcore drives `op` and the FU is responsible for driving the latter three. `op` tells the FU that input operands are ready to be consumed. `ready` indicates that the FU can consume new operands. `valid` and `done` are related: `valid` says that the FU has data ready to send over the network, and `done` says the FU has completed execution. The remaining signals are data: incoming operands (`a`, `b`), predicate operands (`m`, `d`), and the FU's output (`z`). The FU-µcore interface allows the µcore to handle variable-latency logic, making the FU's outputs available only when the FU completes an operation. The µcore raises back-pressure in the network when output from an FU is not ready and stalls the FU (by keeping `op` low) when input operands are not ready or there are no unallocated intermediate buffers. When the FU asserts both `valid` and `done`, the µcore forwards the value produced by the FU to dependent PEs via its NoC router.

Progress tracking and fabric control: The fabric has a top-level controller that interfaces with each µcore via three 1-bit signals. The first enables the µcore to begin execution, the second resets the µcore, and the third tells the controller when the PE has finished processing all input. The µcore keeps track of the progress of the FU by monitoring the `done` signal, counting how many elements the FU has processed. When the number of completed elements matches the length of the computation, the µcore signals the controller that it is done.

Predication: SNAFU supports conditional execution through built-in support for vector predication. The µcore delivers not only the predicate `m`, but also a fallback value

`d` — for when the predicate is false — to the FU. When the predicate is true, the FU executes normally; when it is false, the FU is still triggered so that it can update internal state (e.g., memory index for a strided load), but the fallback value is passed through.

Configuration services: The µcfg handles processing element configuration, setting up a PE's dataflow routes and providing custom FU configuration state. Router configuration maps inputs (`a`, `b`, `m`, `d`) to a router port. The µcfg forwards custom FU configuration directly to the FU, which SNAFU assumes handles its own internal configuration. The µcfg module contains a configuration cache that can hold up to six different configurations. The cached configurations reduce memory accesses and allow for fast switching between configurations. This improves both energy-efficiency and performance. It also benefits applications with dataflow graphs too large to fit onto the fabric. These applications split their dataflow graph into multiple subgraphs. The CGRA executes them one at a time, efficiently switching between them via the configuration cache. Note, however, that even with the configuration cache, each fabric configuration is intended to be re-used across many input values before switching, unlike prior CGRAs that multiplex configurations cycle-by-cycle (Sec. 2.4).

5.2.2 SNAFU's PE standard library

SNAFU includes a library of PEs that we developed using the BYOFU custom FU interface. The library includes four types of PEs: a basic ALU, multiplier, memory (load/store) unit, and scratchpad unit.

Arithmetic PEs: There are two arithmetic PEs: the basic ALU and the multiplier. The basic ALU performs bitwise operations, comparisons, additions, subtractions, and fixed-point clip operations. The multiplier performs 32-bit signed multiplication. Both units are equipped with the ability to accumulate partial results, like PE #4 (`vredsum`) in Fig. 5.3.

Memory PEs: The memory PEs generate addresses and issue loads and stores to global memory. The PE operates in two different modes, supporting strided access and indirect access. The memory PE also includes a "row buffer," which eliminates many subword accesses on accesses to a recently-loaded word.

Scratchpad PEs: A scratchpad holds intermediate values produced by the CGRA. The scratchpad is especially useful for holding data communicated between consecutive configurations of a CGRA, e.g., when the entire dataflow graph is too large for the CGRA. The PE connects to a 1 KB SRAM memory that supports stride-one and indirect accesses. Indirect access is used to implement permutation, allowing data to be written or read in a specified, permuted order.

5.2.3 Generating a CGRA fabric

Generating RTL: Given a collection of processing elements, SNAFU automatically generates a complete energy-minimal CGRA fabric. SNAFU ingests a high-level description of the CGRA topology and generates valid RTL. This high-level description includes a list of the processing elements, their types, and an adjacency matrix that encodes the NoC topology. With this high-level description, SNAFU generates an RTL header file. The file is used to parameterize a general RTL description of a generic, energy-minimal CGRA fabric, which can then be fed through standard CAD tools.

NoC and router topology: SNAFU generates a NoC using a parameterized bufferless router model. The router can have any input and output radix and gracefully handles network back-pressure. Connections between inputs and outputs are configured

statically for each configuration. Routers are mux-based because modern CAD tools optimize muxes well.

Top-down synthesis streamlines CAD flow: Following RTL generation, SNAFU fabrics can be synthesized through standard CAD tools from the top down without manual intervention. Top-down synthesis is important because SNAFU's bufferless, multi-hop NoC introduces combinational loops that normally require a labor-intensive, bottom-up approach to generate correct hardware. Industry CAD tools have difficulty analyzing and breaking combinational loops (i.e., by adding buffers to disable the loops). SNAFU leverages prior work on synthesizing FPGAs (which face the problem with combinational loops in their bufferless NoCs) from the top down to automate this process [140, 165]. SNAFU partitions connections between routers and PEs and uses timing case analysis to eliminate inconsequential timing arcs. SNAFU is the first framework for top-down synthesis of a CGRA, eliminating the manual effort of bottom-up synthesis.

5.2.4 Compilation

The final component is a compiler that targets the generated CGRA fabric. Fig. 5.3 shows the compilation flow from vectorized code to valid CGRA configuration bitstream. The compiler first extracts the dataflow graph from the vectorized C code. SNAFU asks the system designer (not the application programmer) to provide a mapping from RISC-V vector ISA instruction to a PE type, including the mapping of an operation's inputs and output onto an FU's inputs and output. This mapping lets SNAFU's compiler seamlessly support new types of PEs.

Integer linear program (ILP) scheduler: The compiler uses an integer linear program (ILP) (see Sec. A.1) formulation to schedule operations onto the PEs of a CGRA. The scheduler takes as input the extracted dataflow graph, the abstract instruction→PE map, and a description of the CGRA's network topology. The scheduler's ILP constraint formulation builds on prior work on scheduling code onto a CGRA [185]. The scheduler searches for subgraph isomorphisms between the extracted dataflow graph and the CGRA topology, minimizing the distance between spatially scheduled operations. At the same time, the ILP adheres to the mappings in the abstract instruction→PE map and does not map multiple dataflow nodes or edges to a single PE or route. To handle PEs that are shared across multiple fabric configurations (e.g., scratchpads holding intermediate data), programmers can annotate code with instruction *affinity*, which maps a particular instruction to a particular PE.

Scalability: Prior work has found scheduling onto a CGRA fabric to be extremely challenging and even intractable [63, 127, 181, 247], limiting compiler scalability to small kernels. However, this is not the case for SNAFU's compiler because SNAFU's hardware makes compilation much easier: SNAFU supports asynchronous dataflow firing and does not time-multiplex PEs or routes. Together, these properties mean that the compiler need not reason about operation timing, making the search space much smaller and simplying its constraints. As a result, SNAFU's compiler can find an optimal solution in seconds even for the most complex kernels that we have evaluated.

Current limitations: If a kernel is too large to fit onto the CGRA or there is resource mismatch between the kernel and the fabric, the tool relies on the programmer to manually split the vectorized code into several smaller kernels that can be individually scheduled. This is a limitation of the current implementation, but not fundamental; a future version of the compiler could automate this process.

5.3. DESIGNING SNAFU TO MINIMIZE ENERGY

SNAFU's design departs from prior CGRAs because it is designed from the ground-up to *minimize energy*. This difference is essential for emerging ULP applications, and it motivates several key features of SNAFU's CGRA architecture. This section explores these differences and explains how they allow SNAFU to minimize energy.

5.3.1 Spatial vector-dataflow execution

MANIC's vector-dataflow execution: Vector-dataflow execution, introduced in Ch. 4, amortizes instruction fetch, decode, and control (vector) and forwards intermediate values between instructions (dataflow). MANIC's vector-dataflow implementation parks intermediate values in a small "forwarding buffer," instead of the large vector register file (VRF).

MANIC reduces energy and adds negligible area, but its savings are limited by two low-level effects that only become apparent in a complete implementation. First, compiled SRAMs are cheaper and scale better than suggested by high-level architectural models [196, 214]; i.e., MANIC's savings from reducing VRF accesses are smaller than estimated. Second, MANIC multiplexes all instructions onto a shared execution pipeline, causing high switching activity in the pipeline logic and registers as control and data signals toggle cycle-to-cycle. Both effects limit MANIC's energy savings.

How SNAFU reduces energy: SNAFU reduces energy by implementing *spatial* vector-dataflow execution. Like vector-dataflow, SNAFU's CGRA amortizes a single fabric configuration across many computations (vector), and routes intermediate values directly between operations (dataflow). But SNAFU *spatially* implements vector-dataflow: SNAFU *buffers intermediate values locally* in each PE (v. MANIC's shared forwarding buffer) and *each PE performs a single operation* (v. MANIC's shared pipeline). Note that this design is also a contrast with some prior CGRAs, which share PEs among multiple operations to increase performance and utilization.

As a result, SNAFU reduces both effects that limit MANIC's energy savings. We estimate that the reduction in switching activity accounts for the majority of the 41% of energy savings that SNAFU achieves v. MANIC. The downside is that SNAFU takes significantly more area than MANIC. This tradeoff is worthwhile because ULP systems are tiny and most area is memory and I/O (see Sec. 5.6). SNAFU's leakage power is negligible despite its larger area because we use a high-threshold-voltage process.

5.3.2 Asynchronous dataflow firing without tag-token matching

The rest of this section discusses how SNAFU differs from prior CGRAs, starting with its dynamic dataflow firing.

Execution in prior CGRAs: Prior CGRAs have explored both static and dynamic strategies to assign operations to PEs and to schedule operations [248]. Static assignment and scheduling is most energy-efficient, whereas fully dynamic designs require expensive tag-matching hardware to associate operands with their operation. A static design is feasible when all operation latencies are known and a compiler can find an efficient global schedule. Static designs are thus common in CGRAs that do not directly interact with a memory hierarchy [88, 119, 182].

How SNAFU reduces energy: SNAFU is designed to easily integrate new FUs with unknown or variable latency. E.g., a memory PE may introduce variable latency due to bank conflicts. A fully static design is thus not well-suited to SNAFU, but SNAFU cannot afford full tag-token matching either.

SNAFU's solution is a hybrid CGRA with static PE assignment and dynamic scheduling. ("Ordered dataflow" in the taxonomy of prior work [248].) Each PE uses local, asynchronous dataflow firing to tolerate variable latency. SNAFU avoids

tag-matching by enforcing that values arrive in-order. This design lets SNAFU integrate arbitrary FUs with little energy or area overhead, adding just ≈ 2% system energy to SNAFU-ARCH. The cost of this design is some loss in performance v. a fully dynamic CGRA. Moreover, asynchronous firing simplifies the compiler, as discussed above, because it is not responsible for operation timing.

5.3.3 Statically routed, bufferless on-chip network

NoCs in prior CGRAs: The on-chip network (NoC) can consume a large fraction of energy in high-performance CGRAs, e.g., more than 25% of fabric energy [119,182]. Buffers in NoC routers are a major energy sink, and dynamic, packet-switched routers cause high switching activity. Prior ULP CGRAs avoid this cost with highly restrictive NoCs that limit flexibility [62,123,187].

How SNAFU reduces energy: SNAFU includes a statically-configured, bufferless, multi-hop on-chip network designed for high routability at minimal energy. Static circuit-switching eliminates expensive lookup tables and flow-control mechanisms, and prior work showed that such static routing does not degrade performance [119]. The network is bufferless (a PE buffers values it produces; see below), eliminating the NoC's primary energy sink (half of NoC energy or more [167]). As a result, SNAFU's NoC takes just ≈ 6% of system energy.

5.3.4 Minimizing buffers in the fabric

Buffering of intermediate values in prior CGRAs: Prior CGRAs maximize performance by forwarding values to dependent PEs and buffering them in large FIFOs, freeing a producer PE to start its next operation as early as possible. If a dependent PE is not ready, the NoC or dependent PE may buffer values. This approach maximizes parallelism, but duplicates intermediate values unnecessarily.

How SNAFU reduces energy: SNAFU includes minimal in-fabric buffering at the producer PE, with none in the NoC. Buffering at the producer PE means each value is buffered exactly once, and overwritten only when all dependent PEs are finished using it. In SNAFU-ARCH, producer-side buffering saves ≈ 7% of system energy v. consumer-side buffering. The cost is that a producer PE may stall if a dependent PE is not ready. SNAFU minimizes the number of buffers at each PE; using just four buffers per PE by default (Sec. 5.6 evaluates SNAFU's sensitivity to the number of buffers per PE).

5.4. SNAFU-ARCH: A COMPLETE ULP SYSTEM W/ CGRA

SNAFU-ARCH is a complete ULP system that includes a CGRA fabric generated by SNAFU integrated with a scalar RISC-V core and memory.

5.4.1 Architectural overview

Fig. 5.5 shows an overview of the architecture of SNAFU-ARCH. There are three primary components: a RISC-V scalar core, a banked memory, and the SNAFU fabric. The SNAFU fabric is tightly coupled to the scalar core. It is a 6×6 mesh possessing 12 memory PEs, 12 basic-ALU PEs, 8 scratchpad PEs, and 4 multiplier PEs. The RTL for the fabric is generated using SNAFU and the mesh topology shown. The memory PEs connect to the banked memory, while the scratchpad PEs each connect to 1 KB outside the fabric.

The RISC-V scalar core implements the E, M, I, and C extensions and issues control signals to the SNAFU fabric. The banked memory has eight 32 KB memory banks (256 KB total). In total there are 15 ports to the banked memory: thirteen from the SNAFU fabric and two from the scalar core. The twelve memory PEs account for the majority of the ports from the fabric. The final port from the fabric allows

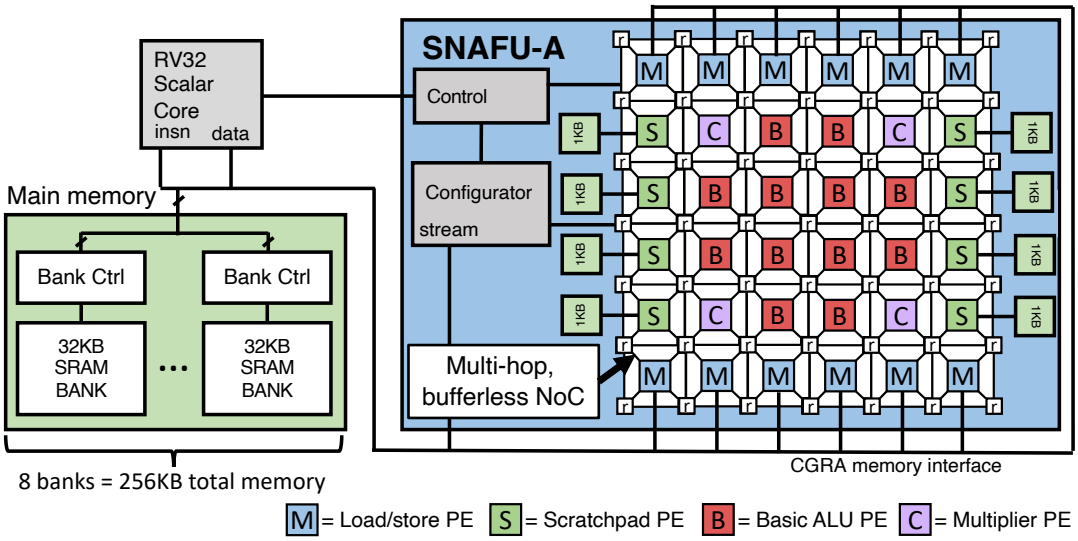


Figure 5.5: Architectural diagram of SNAFU-ARCH. SNAFU-ARCH possesses a RISC-V scalar core tightly coupled with the SNAFU fabric. Both are attached to a unified 256 KB banked memory.

the SNAFU configurator to load configuration bitstreams from memory. Each bank of the main memory can execute a single memory request at a time; its bank controller arbitrates requests using a round-robin policy to maintain fairness.

5.4.2 Example of SNAFU-ARCH in action

SNAFU-ARCH adds three instructions to the scalar core to interface with the CGRA fabric, summarized in Table 5.1. We explain how they work through the following example.

The SNAFU fabric operates in three states: idle, configuration, and execution. During the idle phase the scalar core is running and the fabric is not. When the scalar core reaches a `vcfg` instruction, the fabric transitions to the configuration state. The scalar core passes a vector length and a bitstream address (from the register file) to the fabric configurator (see Fig. 5.5). The configurator checks to see if this configuration is still in the fabric's configuration cache (Sec. 5.2.1). If it is, the configurator broadcasts a control signal to all PEs and routers to load the cached configuration; otherwise, it loads the configuration header from memory. The header tells the configurator which routers and which PEs are active in the configuration. Then the configurator issues a series of loads to read in configuration bits for the enabled PEs and routers.

Once this has completed, the configurator stalls until the scalar core either reaches a `vtfr` instruction or a `vfence` instruction. `vtfr` lets the scalar core pass a register value to the fabric configurator, which then passes that value to a specific PE (encoded in the instruction). This allows PEs to be further parameterized at runtime from the scalar core. `vfence` indicates that configuration is done, so the scalar core stalls and

Instruction	Purpose
`vcfg <len> <addr>`	Load a new fabric configuration and set vector length.
`vtfr <val> <pe>`	Communicate scalar value to fabric.
`vfence`	Start fabric execution and wait.

Table 5.1: New instructions to interface with CGRA.

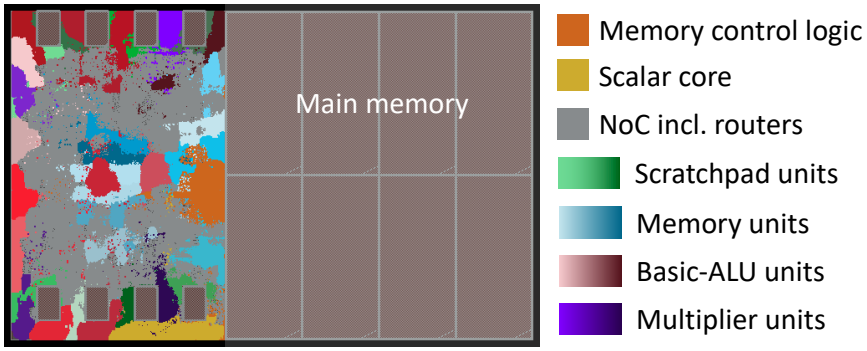


Figure 5.6: Layout of SNAFU-ARCH. Memory PEs are blue, scratchpad PEs are green, basic-ALU PEs are red, multiplier PEs are purple, the scalar core is yellow, and main-memory control logic is orange. The remaining grey regions contain routers and wires.

the fabric transitions to execution. Execution proceeds until all PEs signal that they have completed their work (Sec. 5.2.1). Finally, the scalar core resumes execution from the `vfence`, and the fabric transitions back into the idle state.

5.5. EXPERIMENTAL METHODOLOGY

We implemented SNAFU-ARCH as well as three baselines entirely in RTL and synthesized each system using an industrial sub-28 nm FinFET process with compiled memories. We evaluated the systems using post-synthesis timing, power, and energy models. Additionally, we placed and routed SNAFU-ARCH (see Fig. 5.6) to validate top-down synthesis.

Software-hardware stack: We developed a complete software and hardware stack for SNAFU. We implemented SNAFU-ARCH, its 256 KB banked memory, and its five-stage pipelined RISC-V scalar core in RTL and verified correctness by running full applications in simulation using both Verilator [217] and Cadence Xcelium RTL simulator [10]. We synthesized the design using Cadence Genus [2] and an industrial sub-28 nm, high-threshold voltage FinFET PDK with compiled memories. Next, we placed and routed the design using Cadence Innovus [4]; Fig. 5.6 shows the layout. We also developed SNAFU's compiler that converts vectorized C-code to an optimal fabric configuration and injects the bitstream into the application binary. Finally, we simulated full applications post-synthesis, annotated switching, and used Cadence Joules [5] to estimate power.

	Parameter	Values
	Frequency	50 MHz
	Main memory	256 KB
	Scalar register #	16
Vector	Vector register #	16
Vector	Vector length	16/32/64
Vector	Window size (for MANIC)	8
SNAFU-ARCH	Fabric dimensions	6×6
SNAFU-ARCH	Memory PE #	12
SNAFU-ARCH	Basic-ALU PE #	12
SNAFU-ARCH	Multiplier PE #	4
SNAFU-ARCH	Scratchpad PE #	8

Table 5.2: Microarchitectural parameters.

Name	Description	Small	Medium	Large
FFT	2D Fast-fourier transform	16×16	32×32	64×64
DWT	2D Discrete wavelet transform	16×16	32×32	64×64
Viterbi	Viterbi decoder	256	1024	4096
Sort	Radix sort	256	512	1024
SMM	Sparse matrix-matrix	16×16	32×32	64×64
DMM	Dense matrix-matrix	16×16	32×32	64×64
SMV	Sparse matrix-dense vector	32×32	64×64	128×128
DMV	Dense matrix-dense vector	32×32	64×64	128×128
Sconv	Sparse 2D convolution	16×16,	32×32,	64×64,
	filter:	3×3	5×5	5×5
Dconv	Dense 2D convolution	16×16,	32×32,	64×64,
	filter:	3×3	5×5	5×5

Table 5.3: Benchmarks and their input sizes.

Baselines: We compare SNAFU-ARCH against three baseline systems: *(i)* a RISC-V scalar core with a standard five-stage pipeline[1], *(ii)* a vector baseline that implements the RISC-V V vector extension, and *(iii)* MANIC [85], the prior state-of-the-art in general-purpose ULP design. The scalar core is representative of typical ULP microcontrollers like the TI MSP430 [110]. Each baseline is implemented entirely in RTL using the same design flow. Table 5.2 shows their microarchitectural parameters. The vector baseline and MANIC both have a single vector lane, which minimizes energy at the cost of performance.

Benchmarks: We evaluate SNAFU-ARCH, MANIC, and the vector baseline across ten benchmarks on three different input sizes, shown in Table 5.3. We use random inputs, generated offline. For MANIC and the vector baseline, each benchmark has been vectorized from a corresponding plain-C implementation. For SNAFU-ARCH, these vectorized benchmarks are fed into our compiler to produce CGRA-configuration bitstreams. In cases where the benchmarks contain a permutation, the kernel is manually split and pieces are individually fed into our compiler.

Metrics: We evaluate SNAFU-ARCH and the baselines primarily on their energy efficiency and secondarily on their performance. We measure the full execution of each benchmark after initializing the system, and we measure efficiency by the energy to execute the complete benchmark normalized to either the scalar baseline or SNAFU-ARCH. We measure performance by execution time (cycles) or speedup normalized to either the scalar baseline or SNAFU-ARCH.

5.6. EVALUATION

We now evaluate SNAFU-ARCH to show: (1) SNAFU-ARCH is significantly more energy-efficient than the state-of-the-art. (2) Secondarily, SNAFU-ARCH significantly improves performance over the state-of-the-art. (3) SNAFU-ARCH is an optimal design point across input, configuration cache, and intermediate-buffer sizes. (4) SNAFU is easily extended with new PEs to improve efficiency. (5) Significant opportunities remain to improve efficiency in the compiler.

5.6.1 Main results

Fig. 5.7 shows that SNAFU-ARCH is much more energy-efficient and performant v. all baselines. The figure shows average energy and speedup of SNAFU-ARCH normalized to the scalar baseline. SNAFU-ARCH uses 81%, 57%, and 41% less energy than the scalar, vector, and MANIC baselines, respectively. SNAFU-ARCH is also highly performant; it is 9.9×, 3.2×, and 4.4× faster than the respective baselines.

1) SNAFU-ARCH saves significant energy: Fig. 5.7 shows detailed results for all ten benchmarks. Fig. 5.7a breaks down execution energy between memory, the scalar core, vector/CGRA, and remaining (other). SNAFU-ARCH outperforms all baselines on each benchmark. This is primarily because SNAFU-ARCH implements spatial vector-dataflow execution. Vector, MANIC, and SNAFU-ARCH all benefit from vector execution (SIMD), significantly improving energy-efficiency and performance compared to the scalar baseline by eliminating much of the overhead of instruction fetch and decode. However, SNAFU-ARCH benefits even more from vector execution because, once SNAFU-ARCH's fabric is configured, it can be re-used across an unlimited amount of data (unlike the limited vector length in the vector baseline and MANIC).

Moreover, only SNAFU-ARCH takes advantage of spatial dataflow. The vector baseline writes all intermediate results to the vector register file, which is quite costly. MANIC eliminates a majority of these VRF accesses (saving 27% energy compared to

[1]This is a different design than the scalar baseline in Ch. 6

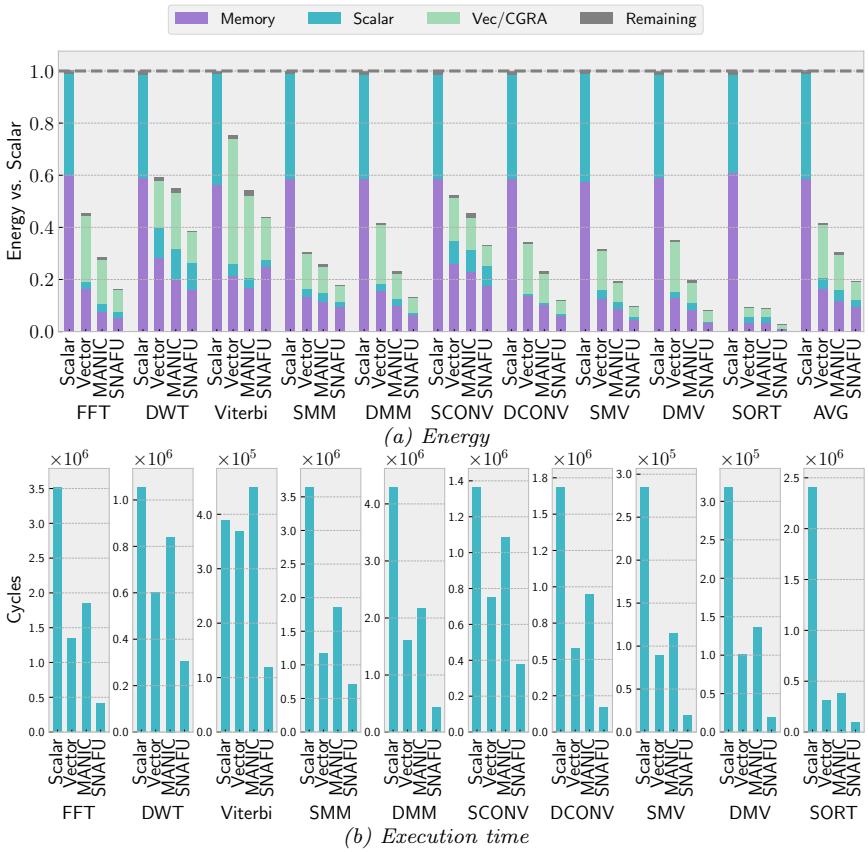


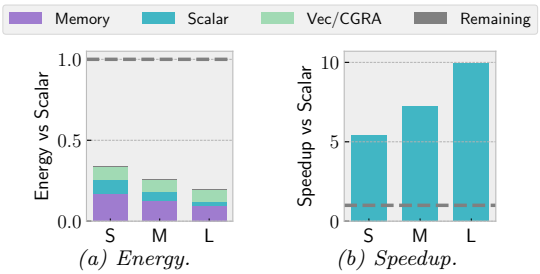
Figure 5.7: Energy and execution time, normalized to the scalar baseline, across ten applications on large inputs. On average, SNAFU-ARCH uses 81%, 57%, 41% less energy and is 9.9×, 3.2×, and 4.4× faster than the scalar design, vector baseline, and MANIC, respectively.

the vector baseline) by buffering intermediate values in a less expensive "forwarding buffer." However, MANIC shares a single execution pipeline across all instructions, which significantly increases switching activity. SNAFU-ARCH, on the other hand, executes a dataflow graph spatially. Each PE only handles a single operation and routes are configured statically. This leads to significantly less dynamic energy because intermediate values are directly communicated between dependent operations and there is minimal switching in PEs.

2) SNAFU-ARCH also greatly improves performance: Fig. 5.7b shows the execution time (in cycles) of all benchmarks and systems. Across the board, SNAFU-ARCH is faster — from 3.2× to 9.9× on average, depending on the baseline. SNAFU-ARCH achieves this high-performance by exploiting instruction-level parallelism in each kernel, which is naturally achieved by SNAFU's asynchronous dataflow-firing at each PE.

3) SNAFU-ARCH is ultra-low-power and has a small footprint: The SNAFU-ARCH fabric operates between 120 µW and 324 µW, depending on the workload. This operating power domain is two to five orders-of-magnitude less than most prior CGRA designs. Leakage power is also insignificant (<3%) because SNAFU-ARCH uses a high-threshold-voltage process.

In addition, SNAFU-ARCH is tiny. The entire design in Fig. 5.6, including compiled memories, is substantially less than 1 mm^2. Note, however, that SNAFU-ARCH saves energy at the cost of area: SNAFU-ARCH occupies 1.8× more area than MANIC and 1.7× more than the vector baseline. Given SNAFU-ARCH's tiny size, we judge this to be a good tradeoff.



(a) Energy. *(b) Speedup.*

Figure 5.8: SNAFU-ARCH v. the scalar design across three input sizes – small (S), medium (M) and large (L).

Benchmark analysis: SNAFU-ARCH is especially energy-efficient on dense linear algebra kernels and sort. SNAFU-ARCH uses on average 49% less energy on average for DMM, DMV, and DConv v. 35% less on average for SMM, SMV, and SConv. This is because the dense linear algebra kernels take full advantage of coalescing in the memory PEs and generally have fewer bank conflicts, reducing energy and increasing performance (5.8× v. 3.8×).

Sort is another interesting benchmark because MANIC barely outperforms the vector baseline, while SNAFU-ARCH reduces energy by 72%. (The scalar baseline performs terribly due to a lack of a good branch predictor.) This gap in energy can be attributed to the unlimited vector length of SNAFU-ARCH: the vector length of both vector and MANIC baselines is 64, but the input size to Sort is 1024. SNAFU-ARCH is able to sort the entire vector with minimal re-configuration and buffering of intermediate values.

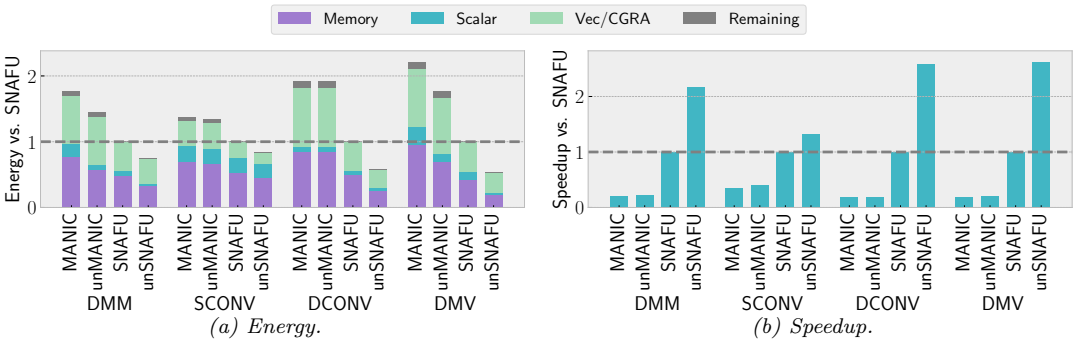
5.6.2 Sensitivity studies

We characterize SNAFU-ARCH by running applications on three different input sizes. Further, we find the optimal configuration of SNAFU-ARCH by sweeping the size of the configuration cache and the number of intermediate buffers.

Energy-efficiency and performance improve on larger workloads: Fig. 5.8a shows SNAFU-ARCH's energy across three different input sizes: small (S), medium (M), and large (L). For most applications, SNAFU-ARCH's benefits increase with input size. (But SNAFU-ARCH is faster and more efficient at all input sizes.) As input size increases, SNAFU-ARCH generally widens the gap in energy-efficiency with the scalar baseline, from 67% to 81%. SNAFU-ARCH also improves v. the vector baseline from 39% to 57% and v. MANIC from 37% to 41% (not shown). The primary reason for this improvement is that, with larger input sizes, SNAFU-ARCH can more effectively amortize the overhead of (re)configuration.

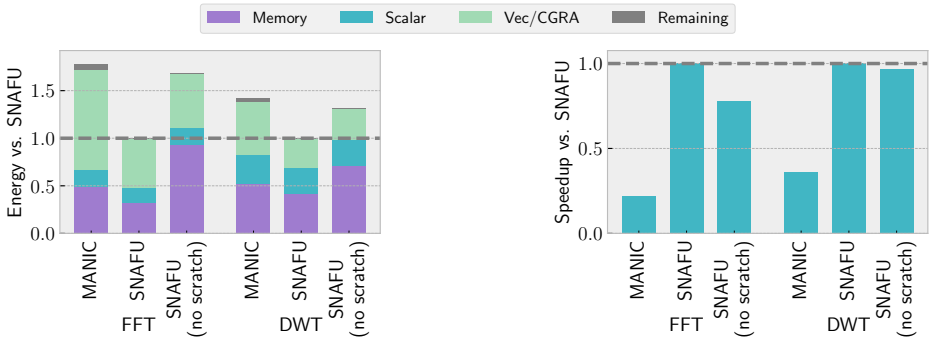
This trend is even more pronounced in the performance data. Fig. 5.8b shows the speedup of SNAFU-ARCH normalized to the scalar baseline. SNAFU-ARCH is 9.9×, 3.2×, 4.4× faster than the scalar baseline, vector baseline, and MANIC on the large input size and 5.4×, 2.4×, and 3.4× faster on the small input.

SNAFU's optimal parameterization: We considered designs with different configuration cache sizes (1, 2, 4, 6, and 8) and different numbers of intermediate buffers (1, 2, 4, and 8). For all applications except FFT, DWT, and Viterbi, configuration-cache size makes little difference. FFT, DWT, and Viterbi realize an average 10% energy savings with a size of six entries. This is because these applications have up to six phases of computation, and each phase requires a different fabric configuration. Similarly, most applications are insensitive to the number of intermediate buffers. With too few buffers, PEs stall due to lack of buffer space. Two buffers is enough to eliminate most of these stalls, and four buffers is optimal.



(a) Energy. *(b) Speedup.*

Figure 5.9: Energy and speedup of MANIC, unMANIC (w/ loop unrolling), SNAFU-ARCH, and unSNAFU-ARCH (w/ loop unrolling), normalized to SNAFU-ARCH. unSNAFU-ARCH uses 31% less energy and is 2.2× faster SNAFU-ARCH; MANIC benefits much less.



(a) Energy. *(b) Speedup.*

Figure 5.10: Energy and speedup of MANIC, SNAFU-ARCH w/ and w/out scratchpads, normalized to SNAFU-ARCH. Scratchpads improve energy-efficiency by 34% and performance by 13%.

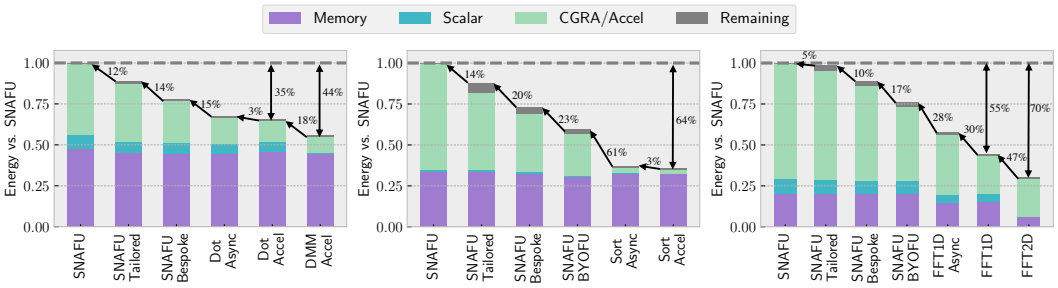
5.6.3 Case studies

We conduct two case studies *(i)* to show that there are opportunities to further improve performance and energy efficiency with only software changes; and *(ii)* to demonstrate the flexibility of SNAFU's BYOFU approach.

Loop-unrolling leads to significantly improved energy and performance: We show the potential of further compiler optimizations through a case study on loop unrolling. Fig. 5.9 shows the normalized energy and speedup of MANIC and SNAFU-ARCH with and without loop unrolling on four different applications. With loop unrolling, SNAFU-ARCH executes four iterations of an inner loop in parallel (v. one iteration without loop unrolling). On average, with loop unrolling, SNAFU-ARCH's energy efficiency improves by 85%, 71%, 62%, and 33% v. scalar, vector, MANIC, and SNAFU-ARCH without unrolling. The performance results are even more significant: with loop unrolling, SNAFU-ARCH's speedup improves to 19×, 7.5×, 11×, and 2.2× v. the same set of baselines. These results make it clear that SNAFU-ARCH can effectively exploit instruction-level parallelism and that there is an opportunity for the compiler to further improve efficiency.

SNAFU makes it easy to add new FUs: Initially, SNAFU-ARCH did not have scratchpad PEs. However, FFT and DWT produce permuted results that must be persisted between re-configurations of the fabric. Without scratchpad units, these values were being communicated through memory.

Leveraging SNAFU's standard PE interface, we were able to quickly add scratchpad PEs to SNAFU-ARCH with minimal effort — we just made SNAFU aware of the new



(a) DMM energy. *(b) Sort energy.* *(c) FFT energy.*

Figure 5.11: Quantifying the cost of SNAFU's programmability on three benchmarks. Each subfigure shows, from left-to-right, the effect of gradually removing SNAFU-ARCH's programmability and increasing its specialization, culminating in a hand-coded ASIC. Overall, SNAFU-ARCH's energy is within 2.6× on average of the ASICs', and can be easily specialized to narrow the gap at low upfront design cost.

PE, without *any* changes to SNAFU's framework. Fig. 5.10 shows the normalized energy and speedup of SNAFU-ARCH with and without scratchpads for FFT and DWT. Persisting intermediate values to main memory is quite expensive: without scratchpads, SNAFU-ARCH consumes 54% more energy and is 16% slower on average. The flexibility of SNAFU allowed us to easily optimize the SNAFU-ARCH fabric for FFT and DWT at low effort, without affecting other benchmarks. The next section explores the implications for programmability and specialization.

5.7. THE COST OF PROGRAMMABILITY

With the mainstream acceptance of architectural specialization, the architecture community faces an ongoing debate over the ideal form and degree of specialization. Some results suggest a need for drastic specialization to compete with ASICs [92, 213, 228]; whereas others argue that programmable designs can compete by adopting a non-von Neumann execution model [179, 184].

We contribute to this debate by performing an apples-to-apples comparison of a programmable design (SNAFU-ARCH) against three hand-coded ASICs, demonstrating that the cost of programmability is low in the ULP domain. We compare end-to-end systems in the same technology and design flow using an industrial PDK with compiled memories. Our results thus avoid pitfalls of prior studies based on simulations, analytical models, or extrapolations from different technologies.

We find that, on average, SNAFU-ARCH uses 2.6× more energy and 2.1× more time than an ASIC implementation. Breaking down the sources of inefficiency in SNAFU-ARCH, we find that SNAFU lets designers trade off programmability and efficiency via simple, incremental design changes. SNAFU makes *selective* specialization easy, letting the architect focus their design effort where it yields the most efficiency gains.

SNAFU is within* 2.6× *of ASIC energy: Fig. 5.11 shows the energy of DMM, Sort, and FFT on large inputs. The leftmost bars in Fig. 5.11 represent SNAFU-ARCH and the rightmost bars represent a fixed-function, statically scheduled ASIC implementation of the same algorithm. SNAFU-ARCH uses as little as 1.8× and on average 2.6× more energy than the ASICs. To explain the gap, we now consider intermediate designs that build up from the ASIC back to SNAFU-ARCH.

SNAFU-ARCH, inner loops, & Amdahl's Law: SNAFU maps only inner loops to its fabric and runs outer loops on the scalar core, limiting its benefits to the fraction of dynamic execution spent in inner loops. To make a fair comparison, we built ASICs for DMM and FFT that accelerate only the inner-loop of the kernel (DOT-ACCEL

and FFT1D-Accel), just like Snafu-Arch. These designs add 33% energy to run outer loops on the scalar core, reducing the energy gap to 2.2× (v. 2.5× for these benchmarks previously). A future version of Snafu could eliminate this extra scalar energy by mapping outer loops to its fabric [248].

Asynchronous dataflow firing adds minimal overhead: Next, we add asynchronous dataflow firing to the ASIC designs (∗-Async bars). Comparing Async designs to the ASICs shows that asynchronous dataflow firing adds little energy overhead, just 3% in DMM and Sort. The 30% overhead in FFT-Async is inessential and could be optimized away: Snafu's current implementation of asynchronous dataflow firing adds an unnecessary pipeline stage when reading scratchpad memories in the ASIC designs.

Closing the gap with negligible design effort: Next, we consider variants of Snafu-Arch to break down the cost of software programmability. Snafu-Bespoke hardwires the fabric configuration, eliminating unused logic during synthesis (like prior work [42]), removing Snafu-Arch's software programmability. Snafu-Bespoke uses 54% more energy than the Async designs. The gap in energy with Async can be attributed to Snafu's logic for predicated execution and the operation set that Snafu implements, which is not well suited to every application. For instance, Sort-Accel can select bits directly, whereas Snafu must do a `vshift` and `vand`.

BYOFU makes it easy to specialize where it matters: Snafu's flexible, "bring your own functional unit" design (Sec. 5.2.1) makes it easy to add missing operations to improve efficiency. To illustrate this, Sort-Byofu and FFT-Byofu improve Snafu-Bespoke's efficiency by adding specialized PEs to the fabric. For Sort, we add a PE that fuses `vshift` and `vand` operations. For FFT, we size scratchpads properly for their data. In both cases, the energy savings are significant. Snafu-Bespoke uses 20% more energy than the Byofu designs, and the Byofu designs come within 44% of the Async ASIC designs. These savings come with much lower design effort than a full ASIC, and we expect the gap would narrow further if more Byofu PEs were added to the fabric.

The cost of software programmability is low: Next, Snafu-Tailored specializes the CGRA to eliminate extraneous PEs, routers, and NoC links, but is not hardwired like Snafu-Bespoke or Snafu-Byofu — i.e., Snafu-Tailored is where the design becomes programmable in software. Snafu-Tailored uses only 15% more energy than Snafu-Bespoke, illustrating that the cost of software programmability is low.

The original Snafu-Arch design uses just 10% more energy than Snafu-Tailored. This gap includes the cost of PEs, routers, and links that are not needed by every application, but may be used by some. This gap is also small, suggesting that *general-purpose* programmability also has a low cost.

The above comparisons yield three major takeaways. The big picture is that ***the total cost of Snafu's programmability is 2–3×*** in energy and time v. a fully specialized ASIC. While significant, this gap is much smaller than the 25× (or larger) gap found in some prior studies [92].[2] Whether further specialization is worthwhile will depend on the application; for many applications, a 2–3× gap is good enough [211].

Moreover, for applications that benefit from more specialization, ***Snafu allows for selective specialization with incremental design effort*** to trade off efficiency

[2]The smaller gap comes from comparing to a programmable design that exploits *both* vector and dataflow techniques to improve efficiency [179, 184].

and programmability. Fig. 5.11 shows that by tailoring the fabric topology, hardwiring configuration state, or partially specializing PEs, designers can get within 2× of ASIC efficiency at a small fraction of ASIC design effort. Designers can incrementally scale their design effort to the degree of specialization appropriate for their application.

Finally, digging deeper, we can better understand the cost of programmability as separate costs incurred at design time (i.e., hardware implementation quality) and run time (i.e., overhead for running software). With current synthesis tools, ***software programmability itself is surprisingly cheap, but carries a hidden design-time cost:*** the gap between SNAFU and SNAFU-BESPOKE is just 27%, whereas the gap between SNAFU-BESPOKE and ASICs is 2.1×. Even when runtime reconfigurability is removed from a design, synthesis tools cannot produce circuits similar to a hand-coded ASIC because they do not understand the intent of RTL. Barring a breakthrough in synthesis, this challenge will remain. SNAFU provides a path forward: BYOFU lets a designer specialize for critical operations, enabling programmable designs to compete with ASICs at a fraction of the design effort and without forfeiting programmability.

5.8. DISCUSSION

This chapter added to the computer architecture component of the new ULP sensor system stack. It presented SNAFU, a framework for generating ultra-low-power CGRAs that maximizes flexibility while minimizing energy. SNAFU takes a *bring your own functional unit* approach, allowing easy integration of custom logic, and it minimizes energy by aggressively favoring efficiency over performance throughout the design. We used SNAFU to generate SNAFU-ARCH, a complete ULP CGRA that uses 41% less energy and is 4.4× faster than MANIC, the prior state-of-the-art general-purpose ULP system. Moreover, SNAFU-ARCH is competitive with ASICs and can be incrementally specialized to trade off efficiency and programmability.

This success of SNAFU makes it clear that the more a program is offloaded to SNAFU, the more efficient it will be. However, while SNAFU is highly programmable, offloading an entire program may be impractical or impossible. This is because SNAFU requires hand-coded vector assembly and only targets vectorizable inner-loops. It has no support for outer-loops or more irregular control-flow patterns. What is required is RIPTIDE (Ch. 6), the final component of the new ULP system stack. RIPTIDE, dicussed next, is a dataflow compiler co-designed with a new CGRA fabric that compiles programs written in higher-level languages (like C) and supports general-purpose control-flow.

Chapter 6

RipTide: a programmable, energy-minimal dataflow compiler and architecture[1]

Computing at the extreme edge calls for ultra-low-power (<1 mW), *extremely energy-efficient*, and *general-purpose* processing. Prior chapters have built out a new ULP sensor system stack that fits these requirements. Ch. 3 contributed Sonic, a software system, to enable inference on intermittent, energy-harvesting devices. Ch. 4 contributed Manic and Manic-Silicon, a new vector-dataflow computer architecture and corresponding silicon prototype. Ch. 5 contributed Snafu, a ULP CGRA architecture that implemented spatial-vector-dataflow execution, achieving energy efficiency competitive with ASICs while remaining programmable by software.

So what is missing from the stack? Well, Amdahl's Law tells us that to achieve significant end-to-end benefits, CGRAs must benefit the vast majority of program execution. CGRAs must support a wide variety of program patterns at minimal programmer effort, and they must provide a complete compiler and hardware stack that makes it easy to convert arbitrary application code to an efficient CGRA configuration. Unfortunately, prior CGRAs, including Snafu, struggle to support common programming idioms efficiently, leaving significant energy savings on the table.

On the hardware side, many prior CGRAs support only simple, regular control flow, e.g., inner loops with streaming memory accesses and no data-dependent control [81, 182, 198]. To support complex control flow and maximize performance, other CGRAs employ expensive hardware mechanisms, e.g., associative tags to distinguish loop iterations, large buffers to avoid deadlock, and dynamic NoC routing [177,188,221,240]. In either case, energy is wasted: from extra instructions needed to implement control flow unsupported by the CGRA fabric, or from inefficiency in the CGRA microarchitecture itself.

On the compiler side, mapping large computations onto a CGRA fabric is a perennial challenge. Heuristic compilation methods often fail to find a valid mapping [181,195], and optimization-based methods lead to prohibitively long compilation times [43, 181]. Moreover, computations with irregular control flow are significantly more challenging to compile due to their large number of control operations, which significantly increase the size of the dataflow graph. To avoid these issues, some CGRAs (including Snafu) require hand-coded vector assembly, restricting programs to primitives that map well onto a CGRA. Vector assembly sidesteps irregular control, but makes programming cumbersome [81, 182, 254].

[1] [83]G. Gobieski, S. Ghosh, M. Heule, T. Mowry, N. Beckmann, and B. Lucia, "RipTide: A programmable, energy-minimal dataflow compiler and architecture," in MICRO, 2022.

RIPTIDE'S APPROACH AND CONTRIBUTIONS

RipTide is a co-designed CGRA compiler and architecture that supports arbitrary control flow and memory access patterns without expensive hardware mechanisms. Unlike prior low-power CGRAs, ***RipTide can execute arbitrary code***, limited only by fabric size and routing. ***RipTide saves energy by offloading more code onto the CGRA***, where it executes with an order-of-magnitude less energy than a von Neumann core. In particular, RipTide supports deeply nested loops with data-dependent control flow and aliasing memory accesses, as commonly found in, e.g., sparse linear algebra. These benefits are realized via the following contributions:

RipTide's instruction set architecture supports complex control while minimizing energy: RipTide adopts a *steering* control paradigm [35, 70, 221], in which values are only routed to where they are actually needed. To support arbitrary nested control without tags, RipTide introduces new control-flow primitives, such as the *carry gate*, which selects between tokens from inner and outer loops. RipTide also optimizes the common case by introducing operators for common programming idioms, such as its *stream generator* that generates an affine sequence for, e.g., streaming memory accesses.

RipTide's (almost) free lunch: offloading control flow into the on-chip network: RipTide implements its new control flow primitives without wasting energy or PEs by leveraging existing NoC switches. The insight is that a NoC switch already contains essentially all of the logic needed for steering control flow, and, with a few trivial additions, it can implement a wide range of control primitives. Mapping control-flow into the NoC frees PEs for arithmetic and memory operations, so that RipTide can support deeply nested loops with complex control flow on a small CGRA fabric.

RipTide compiles C programs to an efficient CGRA configuration: RipTide is easy to program: it compiles functions written in a high-level language (currently, C) and employs novel analyses to safely parallelize operations. We observe that, with steering control flow and no program counter, conventional transitive reduction analysis fails to enforce all memory orderings, so we introduce *path-sensitive transitive reduction* to infer orderings correctly. RipTide implements arbitrary control flow without associative tags by enforcing strict ordering among values, leveraging its new control operators. RipTide maps programs onto the CGRA by formulating place-and-route as a SAT instance or integer linear program. The SAT formulation finds configurations quickly (< 3 min), while the ILP formulation yields configurations that use 4.3% less energy.

Summary of results: We implement a complete RipTide system in RTL and synthesize it in an industrial sub-28nm FinFET process with compiled memories. Including core and memory, RipTide's area is just $\approx 0.5\text{mm}^2$. Across ten benchmarks, ranging from linear algebra to graph search, RipTide reduces energy by 25% v. Snafu, the state-of-the-art energy-minimal design, and improves performance by 17% (Fig. 6.1). At nominal voltage with random inputs, RipTide achieves 180 MOPS/mW (including main memory) on `dmm`. RipTide consumes just 2.4× more energy than equivalent ASICs for `dmm`, `sort`, and `fft`, and RipTide achieves these benefits on software written in C.

We identify several methodological challenges in measuring CGRA efficiency. The choice of metric can skew reported efficiency by more than 10× — e.g., RipTide achieves 1970 fabric MIPS/mW on `dmm`, which is often reported as MOPS/mW in

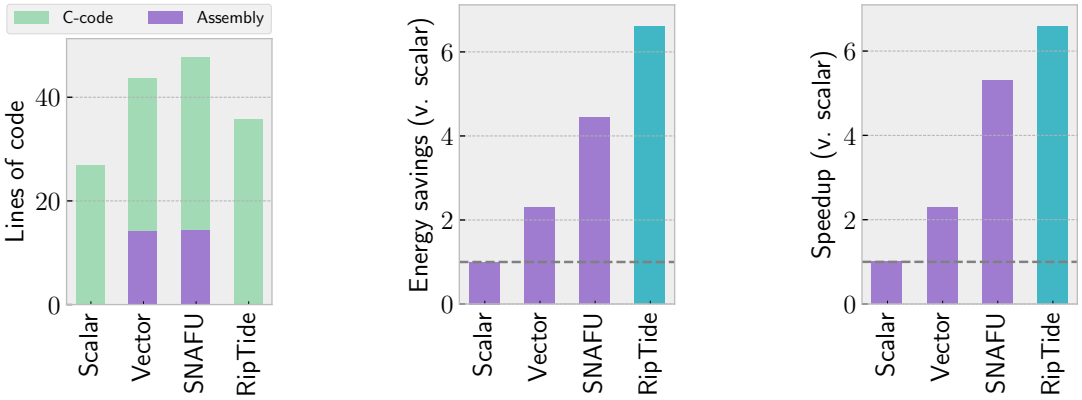


Figure 6.1: RipTide improves energy efficiency and performance on average across ten benchmarks over the state of the art, while compiling programs from high-level C (v. vector assembly in SNAFU).

prior work. Surveying prior work, we find that RipTide is 2.4× more energy-efficient than prior, performance-oriented CGRAs with comparable data [232, 243].

Broader implications on architecture: We perform an in-depth case study of `dmm`, comparing RipTide to an ASIC implemented in the same design flow. RipTide is competitive on energy and performance, but consumes significantly more area than the ASIC. ASICs thus offer an area advantage over CGRAs, but this advantage disappears in SoC designs with a large number of ASIC blocks. Given the large advantages gained by software programmability, we argue that energy-minimal CGRAs like RipTide have a compelling edge over ASICs for the majority of computations.

Road map: Secs. 6.1, 6.2, and 6.3 present RipTide's architecture, compiler, and microarchitecture, respectively. Secs. 6.4 and 6.5 evaluate RipTide, and Sec. 6.6 concludes by discussing RipTide's broader implications.

6.1. RIPTIDE INSTRUCTION SET ARCHITECTURE

RipTide provides a rich set of control-flow operators to support complex programs. Its ISA, shown in Table 6.1, has six categories of operators: arithmetic, multiplier, memory, control flow, synchronization, and streams. (Multiplication is split from other arithmetic because, to save area, only some PEs can perform multiplication.) We now highlight the control-flow, synchronization, and stream operators.

6.1.1 Control-flow operators

RipTide's operators are illustrated in Fig. 6.2. Whenever a value is read, it is implied that the operator waits until a valid token arrives for that value over the NoC. Tokens are buffered at the inputs if they are not consumed or discarded.

Operator(s)	Category	Symbol(s)	Semantics
Basic binary ops	Arithmetic	$+$, $-$, $<<$, $!=$, etc.	*a* `op` *b*
Multiply, clip	Multiplier	$*$, `clip`	*a* `op` *b*
Load	Memory	`ld`	`ld` *base*, *idx*(, *dep*)
Store	Memory	`st`	`st` *base*, *idx*, *val*(, *dep*)
Select	Control Flow	`sel`	*cond* `?` *val*0 `:` *val*1
Steer, carry, invariant	Control Flow	(`T` | `F`), `C`, `I`	See Fig. 6.2
Merge, order	Synchronization	`M`, `O`	See Fig. 6.2
Stream	Stream	`STR`	See Fig. 6.2

Table 6.1: RipTide's instruction set architecture (ISA).

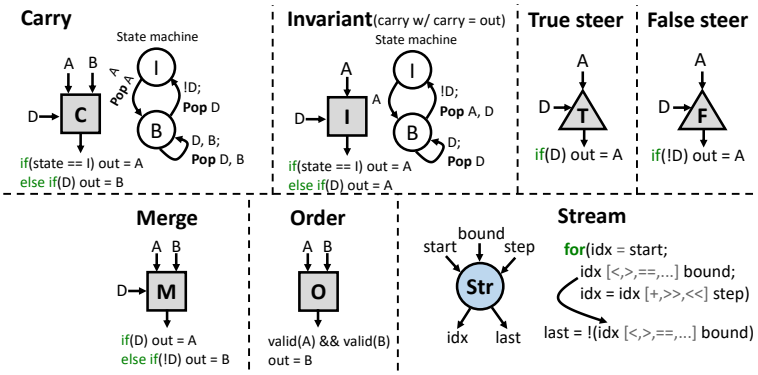


Figure 6.2: Semantics of control-flow operators in RipTide.

Steer: Steers (ϕ^{-1}) come in two flavors — True and False — and take two inputs: a decider, `D` and a data input, `A`. If `D` matches the flavor, then the gate passes `A` through; otherwise, `A` is discarded. Steers are necessary to implement conditional execution, as they gate the inputs to disabled branches.

Carry: Carry represents a loop-carried dependency and takes a decider, `D`, and two data values, `A` and `B`. Carry has the internal state machine shown in Fig. 6.2. In the *Initial* state, it waits for `A`, and then passes it through and transitions to the *Block* state. While in *Block*, if `D` is True, the operator passes through `B`. It transitions back to *Initial* when `D` is False, and begins waiting for the next `A` value (if not already buffered at the input).

Carry operators keep tokens ordered in loops, eliminating the need to tag tokens. All backedges are routed through a carry operator in RipTide. By not consuming `A` while in *Block*, carry operators prevent outer loops from spawning a new inner-loop instance before the previous one has finished. (Iterations from one inner-loop may be pipelined if they are independent, but entire instances of the inner loop will be serialized.)

Invariant: The invariant operator is a slight variation of carry. It represents a loop invariant and can be implemented as a carry with a self-edge back to `B`. Invariants are used to generate a new loop-invariant token for each loop iteration.

6.1.2 Synchronization operators

Merge: The merge operator enforces cross-iteration ordering by making sure that tokens from different loop iterations appear in the same order, regardless of the control path taken within by each loop iteration. The operator takes three inputs: a decider, `D`, and two data inputs, `A` and `B`. Merge is essentially a mux that passes through either `A` or `B`, depending on `D`. But note that only the value passed through is consumed.

Order: The order operator is used to enforce memory ordering by guaranteeing that multiple preceding operations have executed. It takes two inputs, `A` and `B`, and fires as soon as both arrive, passing `B` through.

6.1.3 Stream operators

Streams generate a sequence of data values, which are produced by evaluating an affine function across a range of inputs. These operators are used in loops governed by affine induction variables. A stream takes three inputs: `start`, `step`, and `bound`. It initially sets its internal `idx` to `start`, and then begins iterating a specified arithmetic operator `f` as `idx' = f(idx, step)`.

A stream operator produces two output tokens per iteration: `idx` itself, and a control signal `last`. `last` is False until `idx` reaches `bound`, whereupon it is True and

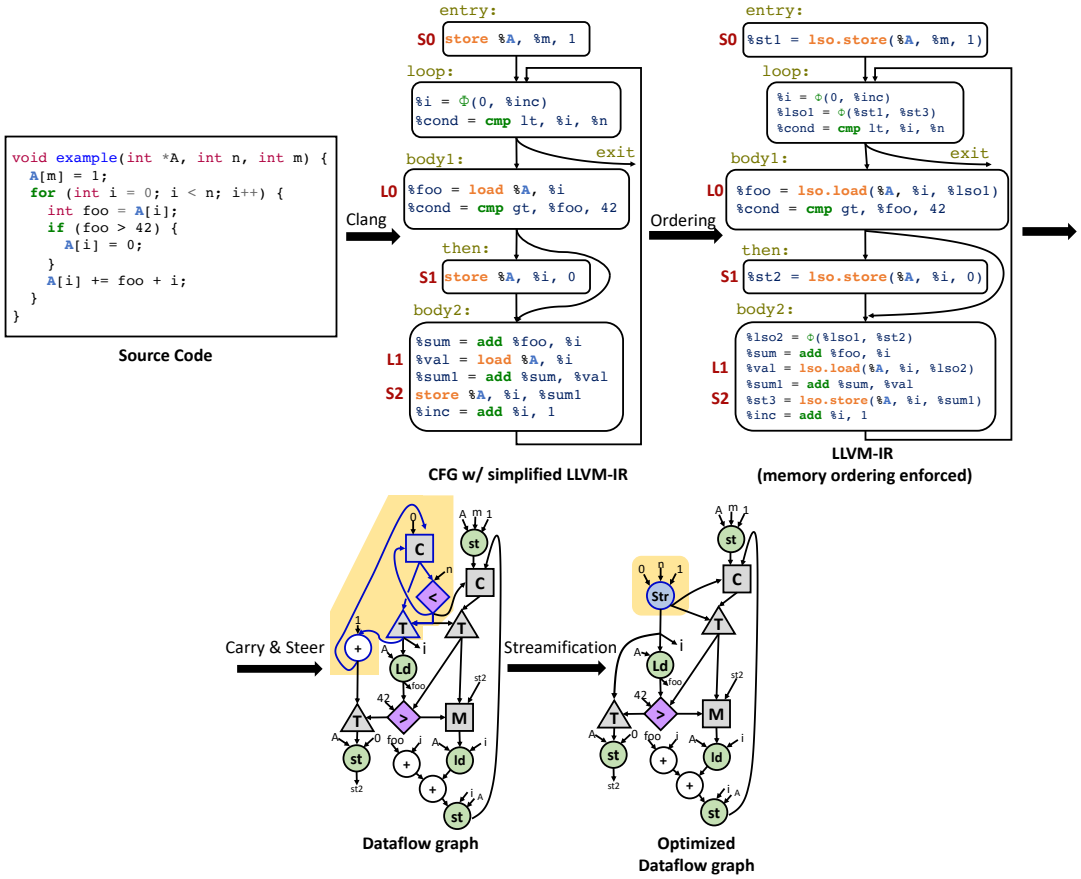


Figure 6.3: RipTide's frontend and middle-end components. The frontend compiles C code to LLVM-IR using `clang`. The middle-end produces an optimized dataflow graph (DFG) that enforces memory ordering and RipTide's control paradigm.

the stream stops iterating. `last` is used by downstream control logic to, e.g., control a carry operator for outer loops.

6.2. RIPTIDE COMPILER

RipTide compiles, optimizes, and maps high-level C code to RipTide's CGRA fabric. Its compiler has a frontend, middle-end, and backend. The frontend uses clang to compile C to LLVM's intermediate representation (IR). The middle-end manipulates the LLVM IR to insert control-flow operators from Sec. 6.1 and enforce memory ordering; then it translates the IR to a dataflow graph (DFG) representation and optimizes the DFG by transforming and fusing subgraphs, reducing operator count by 27% (Fig. 6.3). The backend takes the DFG as input and maps operators onto the CGRA, producing a configuration bitstream in minutes.

6.2.1 Memory-ordering analysis

RipTide maps sequential code onto a CGRA fabric in which many operations, including memory, may execute in parallel. For correctness, some memory operations must execute in a particular order. RipTide's middle-end computes required orderings between memory operations present in the IR and adds control-flow operations to enforce those orderings.

Constructing a memory-operation ordering graph: The first step to enforcing memory ordering is to construct an ordering graph (OG) that encodes dependences

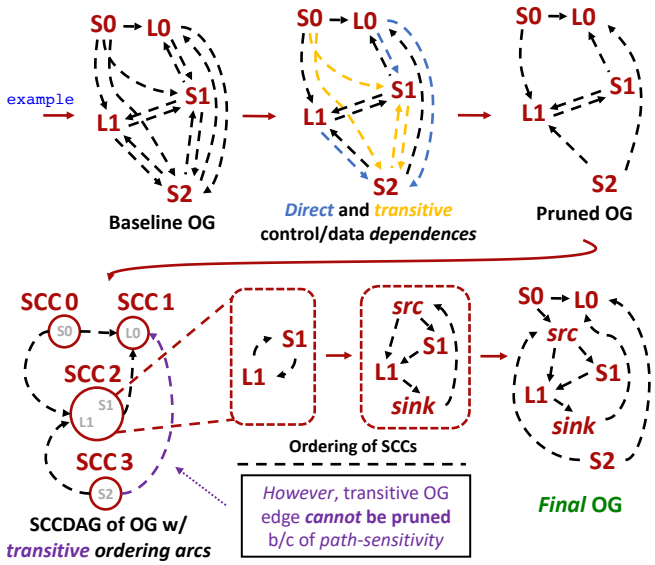


Figure 6.4: RipTide's middle-end enforces memory ordering. For `example`, an ordering graph (OG) that is iteratively pruned and reduced.

between memory operations. RipTide uses alias analysis to identify memory operations that may or must access the same memory location (i.e., alias), adding an arc between the operations in the OG accordingly. RipTide makes no assumptions on the alias analysis and need not consider self-dependences because repeated instances of the same memory operation are always ordered on its CGRA fabric. Fig. 6.4 shows a basic, unoptimized OG in the top left for an `example` function.

Pruning the ordering graph: The OG as computed can be greatly simplified. Prior work has simplified the OG with improved alias analysis [106] and by leveraging new control-flow primitives [38, 156]. These efforts are orthogonal to RipTide. RipTide simplifies the OG by eliminating redundant ordering arcs that are already enforced by data and control dependences. RipTide finds data dependences by walking LLVM's definition-use (def-use) chain from source to destination and removes *ordering* arcs for dependent operations [236]. For instance, in `example`'s CFG from Fig. 6.3, **S2** is data-dependent on **L1**, so there need not be an ordering arc in the OG. This is reflected in the blue-outlined arc from **L1** to **S2** that is pruned in the OG in Fig. 6.4. Similarly, control dependences order some memory operations if the execution of the destination is control-dependent on the source. RipTide analyzes the CFG to identify control dependences between memory operations and removes those orderings from the OG. In `example`'s CFG from Fig. 6.3, the arc from **L0** to **S1** in Fig. 6.4 is pruned using this analysis.

Transitive memory-ordering analysis: Two dependent memory operations are transitively ordered if there is a path (of ordering arcs) in the OG from source to destination. RipTide finds and eliminates redundant arcs that are transitively ordered by other control- and data-dependence orderings. This reduces the number of operations required to enforce ordering by 18% v. unoptimized ordering.

To simplify its OG, RipTide uses transitive reduction (TR) [13], which prior work deployed to simplify ordering relation graphs for parallel execution of loops [156, 157]. We apply TR to the OG, which converts a (potentially cyclic) ordering graph into an acyclic graph of strongly connected components (the SCCDAG). Traditional TR eliminates arcs between SCCs, removes all arcs within each SCC, and adds arcs to each SCC to form a simple cycle through all vertices.

We modify the algorithm in two ways to make it work for RipTide's OG. First, arcs in the inserted cycle must be compatible with program order instead of being arbitrary. Second, the inserted arcs must respect proper loop nesting, avoiding arcs directly from the inner to outer loop. To handle these arcs, we add synthetic loop entry and exit nodes to each loop (shown as `src` and `sink` nodes at the bottom of Fig. 6.4). Any arc inserted that links an inner loop node to an outer loop node instead uses the inner loop's exit as its destination. Symmetrically, an arc inserted that links an outer loop node to an inner loop node has the inner loop's entry as its destination. With these two changes, the SCCDAG is usable for TR.

However, we observe that applying existing TR analysis to the OG in RipTide fails to preserve required ordering operations. The problem is that a source and destination may be ordered along one (transitive) path, and ordering along another (direct) path may be removed as redundant. Execution along the transitive path enforces ordering, but along the direct path does not, which is incorrect. Fig. 6.4 shows a scenario where path-sensitivity is critical. The path, **SCC3**(**S2**)→**SCC1**(**L0**), should not be eliminated in TR because the alternative path, **SCC3**(**S2**)→**SCC2**(**L1**)→**SCC1**(**L0**), does not capture the direct control-flow path from **S2** to **L0** via the backedge of the loop. This problem arises due to RipTide's steering control and lack of a program counter to order memory operations.

To correctly apply TR to remove redundant ordering arcs, RipTide introduces *path-sensitive* TR, which confirms that a transitive ordering path subsumes all possible control-flow paths before removing any ordering arc from the OG. With this constraint in place, RipTide can safely use transitive reduction.

Enforcing ordering constraints: Memory operators in RipTide produce a control token on completion and can optionally consume a control token (*dep* in Table 6.1) to enforce memory ordering. The middle-end encodes ordering arcs as defs and uses of data values in the IR (as seen in the IR transform of loads and stores in Fig. 6.3) before lowering them as dependences in the DFG. For a memory operator that must receive multiple control signals, the middle-end inserts order operations (Sec. 6.1) to consolidate those signals.

6.2.2 Control-flow operator insertion

The compiler lowers its IR to use RipTide's control paradigm by inserting RipTide control-flow operators into the DFG.

Steer: The compiler uses the control dependence graph (CDG) [57] to insert steers. For each consumer of a value, the compiler walks the CDG from the producer to the consumer and inserts a steer operator at each node along the CDG traversal if it has not already been inserted by a different traversal. The steer's control input is the decider of the basic block that the steer depends on, and its data input is the value or the output of an earlier inserted steer.

Carry and invariant: For loops, the compiler inserts a carry operator for loop-carried dependences and an invariant operator for loop-invariant values into the loop header. A carry's data input comes from the loop backedge that produces the value. An invariant's data input comes from values defined outside the loop. These operators should produce a token only if the next iteration of the loop is certain to execute; to ensure this behavior, the compiler sets their control signal to the decider of the block at the loop exit.

Merge: If two iterations of a loop may take different control-flow paths that converge at a single join point in the loop body, either may produce a token to the join point first. But for correctness, the one from the earlier iteration must produce the first

token. The compiler inserts a merge operator at a join point in the CFG to ensure that tokens flow to the join point in iteration order. The control signal `D` for the merge operator is the decider of nearest common dominator of the join point's predecessor basic blocks. Since the earlier iteration sends its control signal first and RipTide does not reorder tokens, the merge operator effectively blocks the later iteration until the earlier iteration resolves.

6.2.3 Stream fusion

RipTide performs target-specific operator fusion on the DFG to reduce required operations and routes by combining value *stream generators* with loop control logic and address computation logic. RipTide supports streams and applies them for the common case of a loop with an affine loop governing induction variable (LGIV). A stream makes loop logic efficient by fusing the LGIV update and the loop exit condition into a single operator. In the DFG, loop iteration logic is represented by the exit condition, an update operator, the carry for the LGIV's value, and the steer that gates the LGIV in a loop iteration. The middle-end fuses these operators into a single stream operator and sets the stream's initial, step, and bound values. Fig. 6.3 shows stream compilation, where the operators for loop iteration logic (outlined in blue in the DFG) are fused into a stream operator. RipTide applies induction-variable analysis [12, 21] to find affine LGIVs. RipTide also identifies address computations, maps these to an affine stream if possible, and fuses the stream into the memory operator.

6.2.4 Mapping DFGs to hardware

RipTide's backend takes a DFG and a CGRA topology description and generates scalar code to invoke RipTide and a bitstream to configure the RipTide fabric. This involves finding a mapping of DFG nodes and edges to PEs, control-flow modules (Sec. 6.3.4), and links. Mapping can be difficult, and there is much prior work on heuristic methods that trade mapping quality for compilation speed [22, 93, 94, 120, 121, 138, 141, 250, 258]. RipTide has two advantages v. this prior work. First, RipTide does not time-multiplex operations, so it only needs to schedule operations in space, not time. Prior compilers unroll loops to reason about operation timing and identify the initiation interval, increasing program size. Second, RipTide targets energy efficiency, not performance. Rather than optimize for initiation interval, it need only focus on finding a valid solution, since leakage is insignificant.

RipTide provides two complementary mappers: one based on boolean satisfiability (SAT) and another based on integer linear programming (ILP) that minimizes the average routing distance. The SAT-based mapper runs quickly, taking < 3 min for our most complex benchmark, whereas the ILP-based mapper yields 4.3% avg. energy savings v. SAT (Sec. 6.5.3).

Problem description: The constraints of the ILP and SAT formulations are similar (see Appendix A for a complete, formal description). The formulations ensure that every DFG vertex is mapped to a hardware node, that every edge is mapped to a continuous route of hardware links, and that the inputs and outputs of a vertex match the incoming and outgoing links of a hardware node. They further disallow the mapping of multiple DFG vertices to a single hardware node, the sharing of hardware links by multiple edges with different source vertices, and the mapping of a DFG edge through a control-flow module when a DFG vertex is mapped to that module. Together these are the necessary constraints to produce not only a valid mapping, but also a good mapping (SAT is close to ILP in terms of energy).

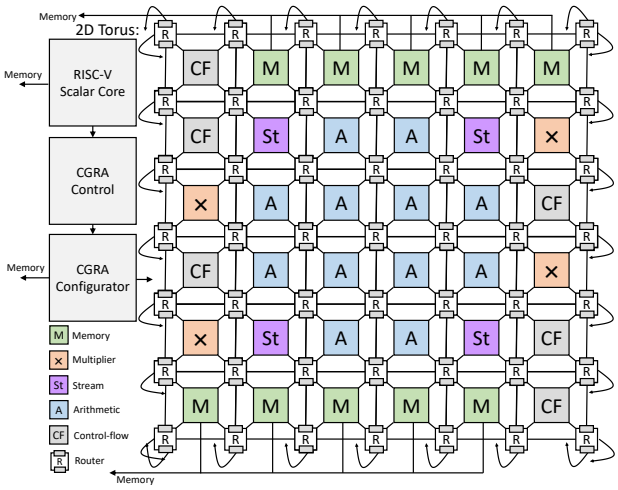


Figure 6.5: RipTide's ULP CGRA fabric.

6.3. RIPTIDE MICROARCHITECTURE

RipTide is an energy-minimal coarse-grained reconfigurable array (Fig. 6.5). The 6×6 fabric contains heterogeneous PEs connected via a bufferless, 2D-torus NoC. A complete RipTide system contains a CGRA fabric, a RISC-V scalar core, and a 256KB (8×32KB banks) SRAM main memory.

6.3.1 Tagless dataflow scheduling

RipTide implements asynchronous dataflow firing via *ordered dataflow* (Sec. 2.4). By adding ordering operators where control may diverge, RipTide ensures that tokens always match on arrival at a PE, obviating the need for tags. Tagless, asynchronous firing has a low hardware cost (one bit per input plus control logic), and it lets RipTide tolerate variable operation latency (e.g., due to bank conflicts) while eliminating the need for the compiler to reason about operation timing.

6.3.2 Processing elements

RipTide's PEs perform all arithmetic and memory operations in the fabric. Fig. 6.6 shows the microarchitecture of a PE. The PE includes a functional unit (FU) and the µcore. The µcore interfaces with the NoC, buffers output values, and interfaces with top-level fabric control for PE configuration.

Functional units: The µcore exposes a generic interface using a latency-insensitive ready/valid protocol to make it easy to add new operators. Inputs arrive on `in_data` when `in_valid` is high, and are consumed when `fu_ready` is high. The FU reserves space in the output channel by raising `fu_alloc` (e.g., for pipelined, multi-cycle operations), and output arrives on `fu_data` when `fu_valid` is high. `out_ready` supplies back pressure from downstream PEs. The remaining signals deal with top-level configuration and control.

Communication: The µcore decouples NoC communication from FU computation. The µcore tracks which inputs are valid, raises backpressure on input ports when its FU is not ready, buffers intermediate results in output channels, and sends results over the NoC. Decoupling simplifies the FU.

Configuration: The µcore handles PE and FU configuration, storing configuration state in a two-entry *configuration cache* that enables single-cycle reconfiguration. Additionally, the µcore enables the fabric to overlap reconfiguration of some PEs while others finish computation on an old configuration.

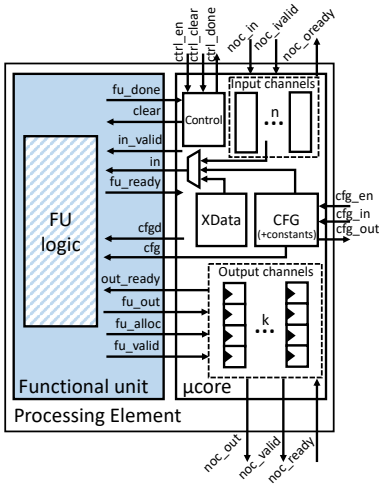


Figure 6.6: PE microarchitecture

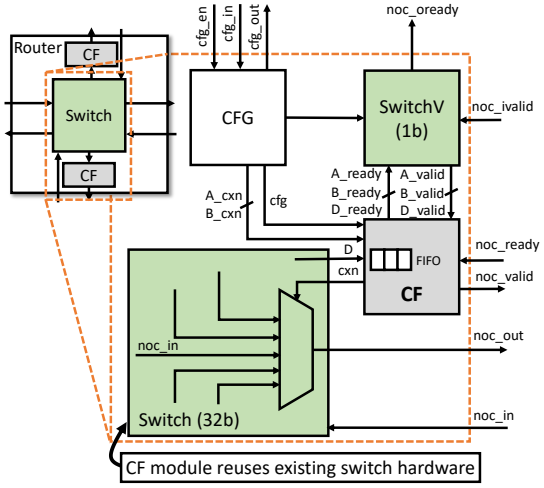


Figure 6.7: Router microarchitecture

PE types: RipTide includes a heterogeneous set of PEs:

- *Memory PEs* issue loads and stores to memory and have a "row buffer" that coalesces non-aliasing subword loads.
- *Arithmetic PEs* implement basic ALU operations, e.g., compare, bitwise logic, add, subtract, shift, etc.
- *Multiplier PEs* implement multiply, multiply + shift, multiply + fixed-point clip, and multiply + accumulate.
- *Control-flow PEs* implement steer, invariant, carry, merge, and order (Sec. 6.1) — but most of these are actually implemented in RipTide's NoC (see below).
- *Stream PEs* implement common affine iterators (Sec. 6.1).

6.3.3 Bufferless NoC

RipTide connects PEs via a statically configured, multi-hop, bufferless on-chip network with routers. Instead of buffering values in the NoC, PEs buffer values in their output channel. NoC buffers are a primary energy sink in prior CGRAs [81, 119], and RipTide completely eliminates them. Similarly, RipTide's NoC is statically routed to eliminate routing look-up tables and flow-control mechanisms.

6.3.4 Control flow in the NoC

Control-flow operators are simple to implement (often a single multiplexer), but there are many of them. Mapping each to a PE wastes energy and area, and can make mapping to the CGRA infeasible. Among our ten benchmarks, 46% of operations are control flow, and eight benchmarks do not map if each control-flow operator requires a dedicated PE.

We observe that much of the logic required to implement control flow is already plentiful in the NoC. Each NoC switch is a crossbar that can be re-purposed to mux values for control. Thus, to implement each control-flow operator, RipTide manipulates a switch's routing and ready/valid signals to provide the desired functionality.

RipTide's router microarchitecture is shown in Fig. 6.7. The router shares routing configuration and its data and valid crossbars with the baseline NoC. RipTide adds a control-flow module (CFM) at a configurable number of output ports (in our case, two output ports). The CFM determines when to send data to the output port and manipulates inputs to the data switch to select which data is sent.

Control-flow module: The CFM takes eight inputs and produces five outputs that control router configuration and dataflow through the network. The inputs are:

- `cfg`: configuration of the CFM (i.e., opcode);

```
cxn = A_cxn
forever:
    A_ready = D_ready = 0
    if A_valid && D_valid: # wait for A and D
        # if D is true, pass through A;
        # else discard A
        noc_valid = D
        A_ready = D_ready = noc_ready || !D
        if D: wait for noc_ready
```

(a) Steer (True flavor).

```
forever:
    # begin in Initial state
    if A_valid:
        cxn = A_cxn          # pass through A
        noc_valid = A_valid
        D_ready = A_ready = noc_ready
        B_ready = xxx        # don't care
    wait for noc_ready
    # transition to Block state
    do until D_valid && !D:
        cxn = B_cxn          # pass through B
        noc_valid = B_valid
        D_ready = B_ready = noc_ready
        A_ready = false      # hold A at input
        wait for noc_ready
```

(b) Carry.

Figure 6.8: Implementing control flow using NoC control signals.

- `A_valid`, `B_valid`, `D_valid`: whether inputs are valid;
- `D`: value of the decider;
- `A_cxn` and `B_cxn`: input ports for `A` and `B`; and
- `noc_ready`: backpressure signal from the output port.

From this, the CFM produces outputs:

- `A_ready`, `B_ready`, and `D_ready`: upstream backpressure signals that allow the CFM to block upstream producers until all signals required are valid;
- `noc_valid`: the valid signal for the CF's output; and
- `cxn`: which port (`A_cxn` or `B_cxn`) to route to the output port on the data switch.

Supported operations: The CFM can be configured for routing or for the control operators in Sec. 6.1. Routing, e.g., `out = A`, is simple: just set `cxn = A_cxn`, `noc_valid = A_valid`, and `A_ready = noc_valid`.

Other operators are more involved, but each requires only a small state machine. Fig. 6.8 is pseudocode for steer and carry operators (Sec. 6.1). A steer forwards `A` if `D` is true; otherwise, it discards `A`. To implement steer, the CFM waits for `A` and `D` to be valid. If `D` is true, then `noc_valid` is raised, and the `noc_ready` signal propagates upstream to `A` and `D` and the CFM waits for `noc_ready`, i.e., for the value to be consumed. If `D` is false, then `noc_valid` is kept low, and `A_ready` and `D_ready` are raised to discard these tokens.

Carry is more complicated. Carry begins in *Initial* state, waiting for a valid `A` token. It forwards the token and transitions to *Blocked* state, where it forwards `B` until it sees a false `D` token. See the pseudocode in Fig. 6.8b for details.

Control flow in the NoC adds small hardware overheads: Implementing control flow in the NoC is far more energy- and area- efficient than in a PE, saving an estimated 40% energy and 22% area v. CGRA with all CF operations mapped to PEs (All PEs in Fig. 6.15). The CFM deals only with narrow control signals and the 1b decider value `D`. It does not need to touch full data signals at all; these are left to the pre-existing data switch. Importantly, this means that the CFM adds no data buffers. Instead, the CFM simply raises the `*_ready` signals to park values in the upstream output channels until they are no longer needed.

By contrast, implementing control flow in a PE requires full data-width muxes and, if an entire PE is dedicated to control, an output channel to hold the results. Nevertheless, RipTide is sometimes forced to allocate a PE for control flow. Specifically, if a control-flow operator takes a constant or software-supplied value that is not -1, 0, or 1, it currently requires µcore support.

Buffering of decider values: The CFM provides a small amount of buffering for decider values. This is because loop deciders often have high fanout, which means that the next iteration of a loop is likely blocked by one or more downstream consumers. To remove this limitation, RipTide provides a small amount of downstream buffering for 1b decider signals, improving performance with minimal impact on area. The CFM uses run-length encoding to buffer up to eight decider values with just 3b of additional state, yielding up to 3.8× performance (on `dmm`) at an area of cost of $< 1\%$.

6.4. EXPERIMENTAL METHODOLOGY

We evaluate a complete RipTide system: the compiler built using LLVM and the microarchitecture fully implemented in RTL in an industrial, sub-28nm FinFET process.

Compiler: RipTide's compiler passes extend LLVM 12.0 [132] and we compile workloads with `-Oz` to optimize code size. RipTide's compiler middle-end uses LLVM's flow-insensitive alias analyses for memory ordering. We evaluate both RipTide's SAT and ILP mappers (see Sec. 6.2.4), but unless otherwise specified we use the ILP mapper. The ILP mapper uses CVXPY [72] and Gurobi 9.5 [91]. The SAT mapper uses CaDiCal [31] to rewrite and simplify the problem's clauses and then uses a new parallel SAT solver, developed concurrently and based on YalSAT [30], to find a valid mapping.

Hardware: RipTide is implemented completely in RTL, including the 6×6 CGRA, RISC-V (RV32EMC) scalar core, and 256KB SRAM main memory. We use Cadence Xcelium to verify correctness and measure performance. We synthesize RipTide using Cadence Genus and a high-threshold-voltage, FinFET PDK with compiled memories. To estimate power, we simulate full benchmarks post-synthesis and use Cadence Joules to estimate power from annotated switching activities.

Baselines: The evaluation compares to several baselines—scalar, vector, Snafu, and three ASICs—also implemented entirely in RTL, using the same design flow. All baselines and RipTide use the same scalar core and main memory. The scalar baseline is a simple, six-stage microcontroller[1]. The vector baseline adds a single-lane coprocessor [85]. Snafu is the state-of-the-art energy-minimal CGRA.

Benchmarks: We evaluate ten workloads important to the ULP domain on random inputs. For the vector baseline, we vectorized all code by hand (except `dfs`, which does not vectorize well). Snafu uses the vectorized code to generate its bitstreams. For RipTide, we compile and run the plain C implementation of each benchmark. The exceptions are `sort`, for which we use merge sort on the scalar core and radix sort for RipTide (because it maps entirely onto the fabric), and `dmm`, for which, where explicitly noted, we tune its C implementation to maximize efficiency.

6.5. EVALUATION

We evaluate RipTide to show that it is easy to program in a high-level language and uses 25% less energy than the state-of-the-art energy-minimal design, while improving performance by 17% on average and up to 2.5×. Moreover, control flow in the NoC is essential for large workloads and reduces energy by up to 2.3×.

[1] This is a more energy-efficient and higher performance design than the scalar baseline in Ch. 5

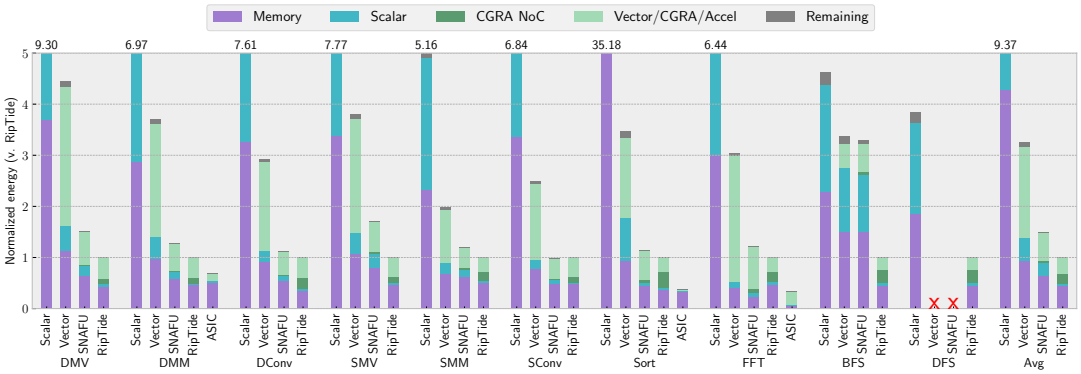


Figure 6.9: Energy (v. RipTide) of scalar, vector, Snafu, RipTide across ten benchmarks. RipTide uses 25% less energy than Snafu.

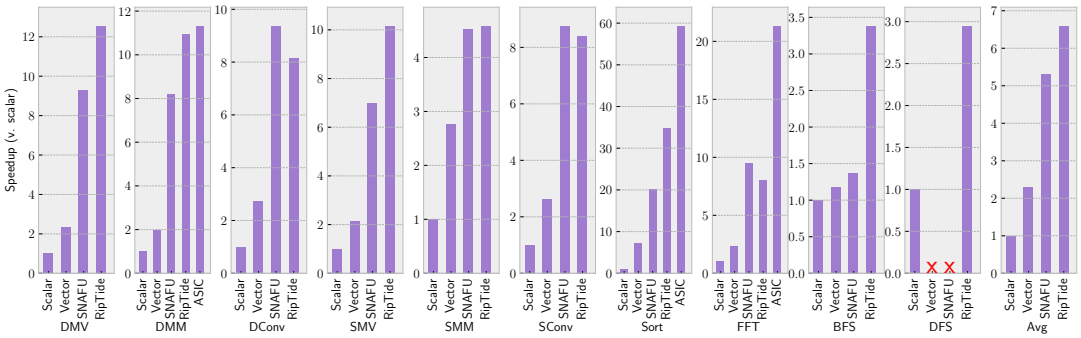


Figure 6.10: Speedup (v. scalar) of scalar, vector, Snafu, and RipTide across ten benchmarks. RipTide is 17% faster than Snafu.

6.5.1 Main results

RipTide compiles high-level code to its fabric: RipTide compiles, schedules, and runs ten applications on its 6×6 fabric. For all but `fft`, RipTide offloads the entire benchmark onto the fabric, including outer loops. For `fft`, a 6×6 fabric does not have enough arithmetic or multiplier PEs, so we split `fft` into two separate functions. Further, RipTide maps and runs `dfs`, which is *not possible* for the vector and Snafu baselines (×s in the figures).

RipTide saves energy: Fig. 6.9 presents energy of the scalar, vector, Snafu, and ASICs normalized to RipTide. RipTide reduces energy by 6.6× v. scalar, 3.1× v. vector, and 25% v. Snafu. RipTide uses less energy across the board. Fig. 6.9 breaks energy into memory, scalar, vector/CGRA, and CGRA NoC. RipTide saves energy v. scalar and vector because it does not fetch instructions, re-uses its configuration across many inputs, and forwards operands directly from producers to consumers. RipTide uses less energy than Snafu by reducing scalar computation: RipTide runs outer loops on the fabric, but Snafu runs them on the scalar core. RipTide's scalar core fetches 86% fewer instructions than Snafu's, eliminating pipeline control, register-file access, and instruction fetch — as seen by RipTide's lower memory energy in Fig. 6.9.

The only benchmark for which memory energy increases v. Snafu is `fft`. Snafu uses scratchpads in the fabric for `fft`, which reduces main memory energy. Even without scratchpads, RipTide shows an overall energy reduction. (RipTide currently lacks a programming interface for scratchpads, but can easily support them in hardware.)

`sconv` shows how control-flow costs in RipTide move from scalar core to the fabric (e.g., steer, carry). While RipTide reduces scalar energy, it adds fabric energy

(v. Snafu) to support outer loops. Scalar execution is a small fraction of overall energy for `sconv`, so RipTide provides no benefit on this benchmark. Moreover, comparing fabric energy for Snafu and RipTide on `sconv` shows that RipTide's microarchitectural additions cost little energy.

RipTide runs C programs with near-ASIC efficiency: Fig. 6.9 also compares RipTide to hand-coded, fixed-function ASICs for `dmm`, `sort`, and `fft`. RipTide uses 2.4× more energy on average than the ASICs while compiling programs directly from C. RipTide compares especially favorably to `dmm`, using 46% more energy. The data show that the cost of RipTide's programmability is low.

RipTide is faster than prior energy-minimal CGRAs: Fig. 6.10 shows performance normalized to scalar. RipTide is 6.2×, 3.4×, and 17% faster than scalar, vector, and Snafu. RipTide does especially well on `bfs`, with a 2.5× speedup v. Snafu. The benefit comes from RipTide's ability to run `bfs`'s irregular outer loop on the fabric, whereas Snafu is bottlenecked on the scalar core because its fabric runs only inner loops. The only benchmarks where RipTide underperforms Snafu are `dconv`, `sconv`, and `fft`. The difference in performance comes down to implementation: we tailor applications to each architecture to minimize energy, not maximize performance. On Snafu, we re-order loops to maximize vector length, minimizing scalar work but adding some memory accesses; on RipTide, we can avoid these memory accesses by accumulating intermediate results in the fabric.

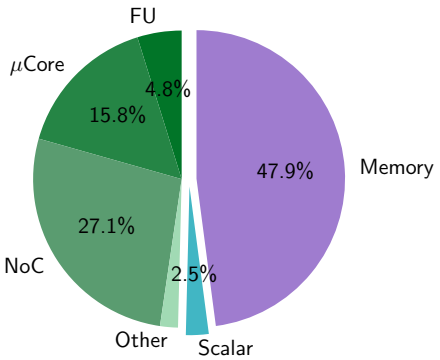


Figure 6.11: Area breakdown for a complete RipTide system. Area is dominated by the CGRA fabric and main memory (SRAM + arbitration logic), eaching taking about half of the system. The scalar core is just 2.5% of system area. The NoC takes 54% of CGRA area, and PEs (μcore + FUs) take 42%.

RipTide is tiny and has extremely low power consumption: The complete RipTide system (CGRA, memory, and scalar core) is ≈ 0.5mm^2. Fig. 6.11 breaks down system area among its components. RipTide operates between 320μW and 910μW, with negligible leakage (< 3%) due to RipTide's high-threshold-voltage process. Overall, the complete system, including memory, achieves 180 MOPS/mW running a hand-tuned C implementation of `dmm` that unrolls twice along the output column dimension. Without tuning, RipTide achieves 141 MOPS/mW on `dmm`.

6.5.2 RipTide v. prior low-power CGRAs

Table 6.2 compares RipTide against several recent CGRAs. We compare designs across their general-purpose programmability, architectural parameters, and reported performance, power, and efficiency. RipTide supports a broader range of programs and is more energy-efficient than prior CGRAs.

Making a fair comparison: Table 6.2 gives absolute numbers for different designs and does not re-scale them to normalize the node. These numbers are our best effort

	HyCube testchip* [243]	HM-HyCube REVAMP◇ [26]	UE-CGRA† [231, 232]	SNAFU◇ [81]	RIPTIDE◇ *(this work)*	
Irregular loops	✗	✗	✓	✗	✓	
Loop nesting	✗	✗	✗	✗	✓	
Memory ordering	✗	✗	✗	✗	✓	
Variable-latency ops	✗	✗	✗	✗	✓	
Node	40LP	22	TSMC 28	sub-28nm	sub-28nm	
Fabric dimensions	4×4	6×6	8×8	6×6	6×6	
Fabric area (mm^2)	—	0.2	0.25	0.27	0.25	
Frequency (MHz)	488	100	750	50	50	
Memory size (KBs)	4	64	64	256	256	
Benchmark	`fft`	Linear algebra	`fft`	`fft`	`fft`	`dmm`‡
Fabric power (mW)	—	8.4	16.7	0.54	0.24	0.50
System power (mW)	140	—	—	0.74	0.52	0.91
Performance (MOPS)	5380	—	625	71	62	164
Fabric efficiency (MOPS/mW)	—	103	38	134	254	328
System efficiency (MOPS/mW)	26	—	—	97	117	180

* Silicon implementation. † Post-P&R simulation. ◇ Post-synthesis simulation. ‡ Hand-tuned C software.

Table 6.2: Comparison of RIPTIDE to other low-power CGRAs. RIPTIDE supports a broader set of programs while improving energy efficiency.

at accurately characterizing prior designs v. RIPTIDE. Few prior CGRAs admit meaningful comparison, however, because prior work reports performance, power, and efficiency inconsistently.

Concrete numbers for energy efficiency are hard to come by. Many prior CGRAs focus on performance [198, 240] or mapping [121, 138, 250] and report metrics (e.g., initiation interval) that are not the focus of RIPTIDE. Others report relative results [174, 232] or use high-level models [174, 248] that make quantitative comparison difficult.

Differences in measurement methodology also make it challenging to compare reported results. Prior CGRAs often report *total* operation count, including, e.g., loads, stores, and loop control, or are unclear about which operations are counted [62, 123, 187]. These numbers, though often reported as MOPS [62, 123], are closer to MIPS as defined for traditional CPUs. Table 6.2 counts only essential arithmetic operations,[2] and we have verified with the authors of other designs in Table 6.2 that they count MOPS the same way. Finally, many prior CGRAs report power for the fabric only, excluding, e.g., the core and memory [26, 125, 187]. We report both fabric and full-system power, and focus on the latter. *Full-system MOPS/mW is the most important metric for the applications targeted by RIPTIDE.*

RIPTIDE is more programmable than prior CGRAs: Table 6.2 highlights a number of programming features supported by RIPTIDE that are unsupported by prior CGRAs. In addition to making RIPTIDE easier to program, these features improve energy efficiency by allowing RIPTIDE to offload a larger fraction of a program onto the efficient CGRA fabric.

RIPTIDE is more energy-efficient than prior CGRAs: RIPTIDE is the most energy-efficient CGRA by a significant margin. Scaled to 22nm, both the HyCube testchip [243] and UE-CGRA [231] achieve roughly 48 full-system MOPS/mW on `fft`. RIPTIDE achieves 117 full-system MOPS/mW, which is 2.4× better, even including RIPTIDE's larger memory and despite `fft` being the only kernel to not fit entirely on RIPTIDE's fabric. On `dmm` with loop unrolling, efficiency improves to 180 full-system MOPS/mW.

[2] Specifically, $2n^3$ ops for `dmm` and $10n \log_2 n + O(n)$ ops for `fft`.

Using a different measurement methodology changes the absolute results dramatically, highlighting the challenge of making apples-to-apples comparisons between CGRAs. If we count *all* operations, instead of only essential arithmetic (i.e., MIPS), RipTide achieves 400 MIPS/mW on `fft`. If we measure only the fabric, RipTide achieves 254 MOPS/mW and 859 MIPS/mW — increasing reported efficiency by 7.3× v. full-system MOPS/mW.

Measuring only the fabric, a recent version of HyCube generated by REVAMP [26] achieves 103 fabric-only MOPS/mW, averaging across several linear algebra benchmarks. (A tuned, heterogeneous fabric achieves 172MOPS/mW.) RipTide achieves 328 fabric-only MOPS/mW on unrolled `dmm`. Meaningful comparisons thus require a detailed understanding of what is being measured.

RipTide's area is similar to prior CGRAs: RipTide is somewhat larger than prior CGRAs (7000 µm^2/PE for RipTide, v. 5500 µm^2/PE for HyCube [26][3] and 3900 µm^2/PE for UE-CGRA). Differences in NoC design help explain these discrepancies. UE-CGRA has no routers, instead routing values through PEs. HyCube's NoC accounts for 24% of fabric area, with each PE containing a 4×4 crossbar switch. RipTide's NoC accounts for 54% (0.14µm^2) of fabric area, but offers more connectivity (more links and 8×8 switches) and capability (dynamic flow control and control-flow operators). Snafu's 2D-mesh NoC is 0.11µm^2 and uses the same switch design as RipTide, showing that RipTide's 2D-torus topology and CFMs add modest overhead (< 0.03µm^2).

RipTide targets a different design point: As currently evaluated, RipTide is much slower than prior CGRAs. We evaluate RipTide at 50 MHz, v. 100s of MHz for prior designs. RipTide has significant slack at 50 MHz and could run much faster. We have not yet pushed frequency further due to RipTide's bufferless NoC and top-down synthesis flow, which requires additional tooling to estimate worst-case critical path. (A similar problem arises in FPGAs.) Frequency in the 10s of MHz is common in ULP microcontrollers.

Nevertheless, lower frequency means that RipTide's raw performance is well below prior CGRAs: on `fft`, 62 MOPS for RipTide v. 5,380 MOPS for HyCube and 625 MOPS for UE-CGRA. Factoring out frequency, RipTide achieves 1.24 ops/cycle v. 11 ops/cycle for HyCube and 0.83 ops/cycle for UE-CGRA. RipTide's lower ops/cycle is partly by design: RipTide trades performance for efficiency by mapping a single operation to each PE, whereas HyCube maps multiple operations per PE to maximize utilization. This tradeoff makes sense for RipTide because it targets applications that are limited by energy, not performance.

Combining RipTide's low frequency and high energy efficiency yields extremely low power consumption. RipTide draws 2–3 orders-of-magnitude less power than HyCube and UE-CGRA. Only Snafu and RipTide draw less than 1 mW — and this is the entire system, including the 256KB main memory.

6.5.3 Compiler characterization

RipTide's compiler effectively optimizes dataflow graphs, reducing operation count by 27% while enforcing memory ordering v. an unoptimized DFG without ordering. The compiler also reduces programmer effort: RipTide compiles from C with no hand-coded assembly, requiring just 8.7 added LoC on average over the original C (mostly for wrappers). Lastly the compiler is fast — the SAT mapper finds a solution to each benchmark in < 3 min and uses only 4.7% more energy than the ILP mapper.

[3] HM-HyCube generated using REVAMP [26]. The HyCube testchip [243] area includes I/O pads, etc., and is not directly comparable.

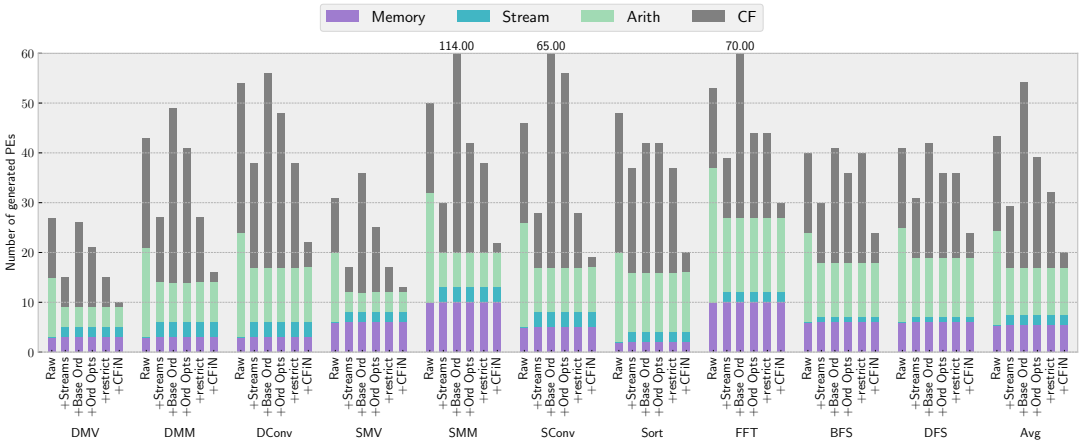


Figure 6.12: Operator counts for ten different benchmarks. Starting with an unoptimized, unordered baseline (`Raw`), compiler optimizations reduce operator counts while enforcing memory ordering, making it feasible to map benchmarks to hardware.

RipTide's compiler reduces operation count: Reducing operation count is important because operations consume PEs in RipTide's fabric. Fig. 6.12 shows operation counts by type with different optimizations applied. The first bar is an unoptimized DFG mapped to RipTide. This DFG requires many PEs to map to hardware and may yield incorrect results because it does not enforce memory ordering. The second bar adds streams, operator fusion, and redundant control-flow elimination, reducing operation count by 33%. The third bar adds unoptimized memory ordering, which *increases* operation count by 82% to ensure correctness. Mapping this graph to hardware is challenging due to its size. The fourth bar applies RipTide's ordering optimizations (Sec. 6.2), reducing operation count (v. the third bar) by 18%. The fifth bar adds programmer-inserted annotations on pointers (C's `restrict` keyword) to better inform LLVM's alias analysis, reducing operation count by 16%. The last bar removes control-flow operations that map to RipTide's NoC, reducing the number of operations on PEs by 35%, demonstrating the benefit of RipTide's control flow in the NoC. Between RipTide's compiler optimizations and implementation of control flow in the the NoC, RipTide reduces operations mapped to PEs by 52% (first v. last bar) while enforcing memory ordering.

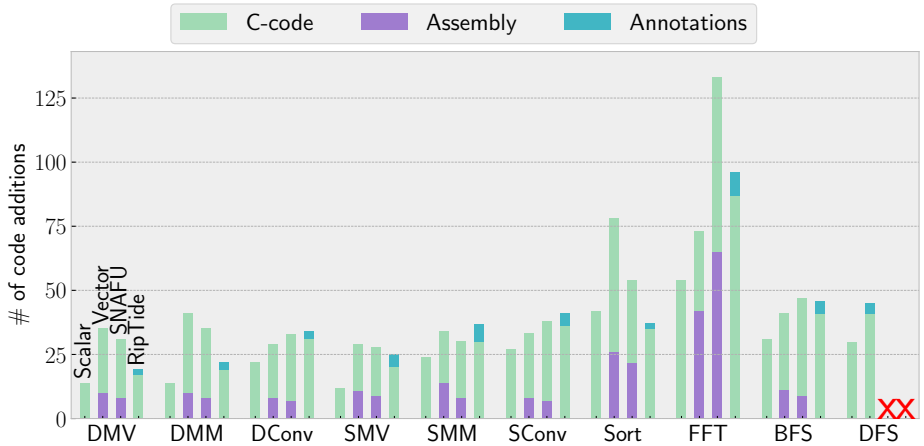
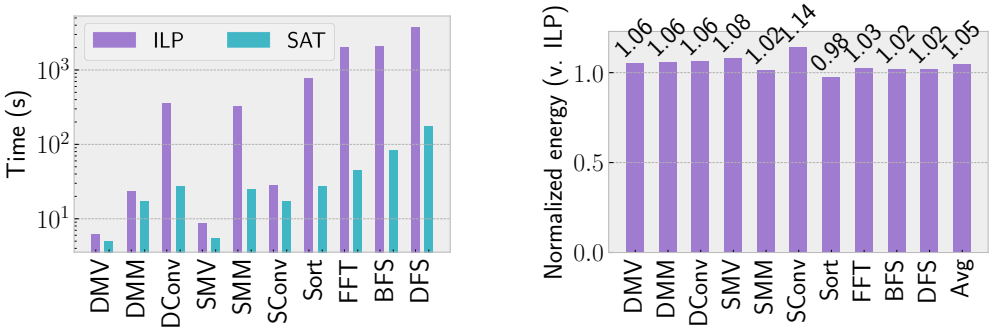


Figure 6.13: The number of code additions for ten benchmarks running on scalar, vector, Snafu, and RipTide. RipTide requires no hand-coded assembly unlike vector and Snafu.

RipTide reduces programmer effort: Fig. 6.13 counts code additions, including lines of code (LoC) in C, assembly, and `restrict` annotations. RipTide has no hand-written assembly, compiling directly from C, while 32% and 27% of the LoC for the vector and Snafu baselines are hand-written assembly. On average, vector adds 17 LoC v. scalar, Snafu adds 21 LoC v. scalar, and RipTide adds just 8.7 lines. Annotations in RipTide represent a small fraction of the overall LoC, just 11.2% and, on average, the programmer adds 4.5 annotations per benchmark.



(a) Compilation time. *(b) Normalized energy.*

Figure 6.14: Compilation time (16 threads, Intel i9-9900K) and normalized energy (v. ILP) of SAT and ILP mappers. SAT is 15.1× faster than ILP, but uses 4.7% more energy.

ILP v. SAT: Fig. 6.14a shows the end-to-end compilation times for RipTide using its SAT and ILP mappers. Fig. 6.14b compares the energies of the resulting mappings. SAT is 15.1× faster than ILP on average, finding solutions to most benchmarks in under a minute. Rapid compilation makes SAT appropriate for iterative software development. On the other hand, ILP produces mappings that use 4.3% less energy on average, making it ideal for final optimization prior to deployment.

The consistently narrow energy gap between SAT and ILP suggests that good solutions are dense in RipTide; i.e., any valid mapping found by SAT is close to the optimal energy from ILP. RipTide does not time-multiplex PEs, so mapping affects energy largely through routing distance. But the loss in routing distance is constrained by routability (i.e., any valid mapping will tend to place dependent operations close to one another), and the energy impact of routing distance in RipTide is reduced by its bufferless NoC. These observations help to explain why SAT performs well in RipTide.

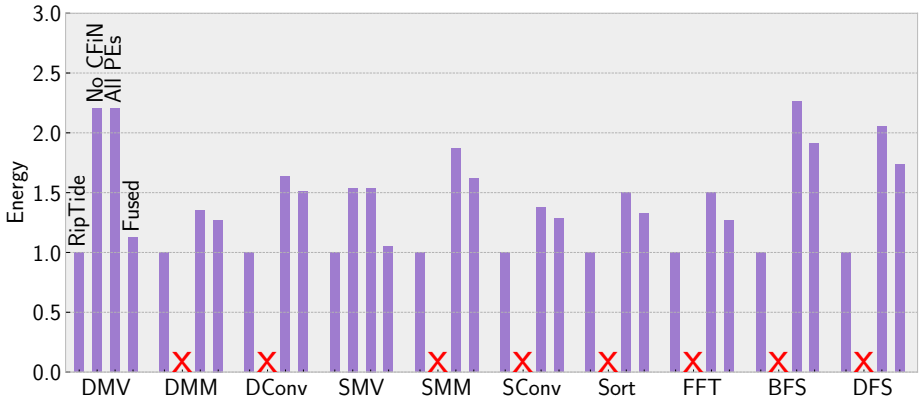


Figure 6.15: Control flow in the NoC saves energy. RipTide uses 45%, 40%, and 27% less energy than RipTide w/ No CFiN, a fabric where all CF ops are PEs ("All PEs"), and a fabric that fuses CF ops into PEs ("Fused").

6.5.4 Control flow in the NoC saves energy & area

Fig. 6.15 quantifies the benefits of implementing control flow in the NoC. From left to right, the plot shows energy on:

- RipTide: Control flow implemented in the NoC (CFiN).
- No CFiN: Control flow mapped to PEs on a 6×6 fabric.
- All PEs: Control flow mapped to PEs, and fabric size increased as necessary to fit each benchmark.
- Fused: One control-flow operator fused into each PE, and fabric sized increased as necessary to fit each benchmark.

We synthesize the first two configurations to estimate energy, while for the latter two configurations we extrapolate energy using area and power estimates for control-flow modules and PEs derived from the synthesized configurations.

RipTide uses the least energy: 45% less than No CFiN, 40% less than All PEs, and 27% less than Fused. RipTide's energy benefit stems from CFiN avoiding the overhead of a full PE. Other configurations also have unique problems. No-CFiN is possible only for `dmv` and `smv`, which are small enough to map to the same RipTide fabric; other workloads have too many control-flow operators to map. The All PEs and Fused configurations add many control-flow PEs, wasting energy and area: RipTide is 22% and 17% smaller than All PEs and Fused, respectively.

6.6. CONCLUSION & ARCHITECTURAL IMPLICATIONS

This chapter presented RipTide, the final piece in the new ULP system stack. RipTide's dataflow compiler makes it easy for a programmer to access the energy-efficiency of it's co-designed CGRA. RipTide provides a rich set of control-flow operators, letting it support arbitrary control flow and memory access on the CGRA fabric. It implements these primitives without tagged tokens and offloads most control operations into its programmable on-chip network, saving energy and hardware resources. RipTide compiles applications written in C while using 25% less energy v. Snafu and 6.6× less v. a von Neumann core.

So where is the trade-off? Fig. 6.9 and Fig. 6.10 compare the energy and performance for `dmm` on RipTide v. an equivalent ASIC. RipTide does not compromise much on energy or performance — coming within 46% and 3%, respectively — but it is not a free lunch. There is a high area cost for RipTide's programmability: RipTide is 57× larger than the ASIC.[4] The question is, is RipTide's programmability worth the extra area?

RipTide area is inflated partly because of low utilization on PEs that perform outer loops. RipTide only supports one operation per PE, so entire PEs are consumed even if an operation fires rarely. Ch. 7 will discuss future designs that address this constraint by allowing limited time-multiplexing, either at a fine [248] or coarse [174] granularity.

Regardless, the area difference shows potentially large cost savings from ASICs, so long as a computation is performed frequently enough to overcome ASICs' upfront design and verification costs. Standardized, pervasive tasks like JPEG compression and wireless communication protocols are good candidates for ASICs. But if the computation is prone to change or used infrequently, then this cost advantage rapidly disappears.

[4] This is without including main memory, which is half of chip area. Also, `dmm` is an extreme case; e.g., RipTide is 9× larger than `fft`. On `fft`, the ASIC yields larger improvements in energy (saving 67%) and performance (by 62%) v. RipTide. This is because RipTide has too few resources to offload the entire `fft` kernel and the ASIC uses scratchpads for twiddle factors.

"Garden of ASICs": Some have proposed that, with increasing transistor budgets and stagnating power budgets, processors should embrace extreme heterogeneity and assemble a large number of distinct ASICs [228,237]. The "garden of ASICs" approach lets architects do something with extra transistors, but it significantly increases system design and verification cost. Moreover, it creates herculean challenges in system integration, as there is no standard programming interface for ASICs, obsolescence is monotonic and likely inevitable, and programs must be somehow partitioned between ASICs and cores with accompanying data-coordination issues.

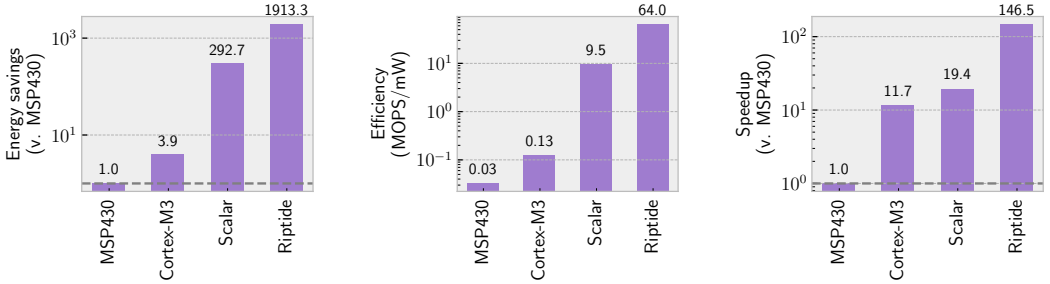
RipTide suggests an alternative approach: Rather than spend area on ASICs that will idle most of the time, instead build an energy-minimal, programmable dataflow fabric. The two designs take similar area with a few dozen ASICs. And the dataflow fabric is cheaper to design, more broadly applicable, and easier to use — programs can be simply compiled for a different target. Finally, as a general-purpose design, programmable dataflow fabrics can create a self-sustaining ecosystem that aggregates optimizations and achieves sufficient scale to justify cutting-edge silicon. All told, while dataflow fabrics like RipTide are not a replacement for ASICs, they will play an important role in improving the efficiency of general-purpose processing as designs are increasingly constrained by energy instead of area, and they will reduce the demand for specialized hardware to accelerate the majority of applications.

Chapter 7

Future work

This thesis has contributed a new ULP system stack that opens up many future research directions. In particular, the success of SNAFU and RIPTIDE makes it possible to consider research questions besides those to do with energy efficiency. This chapter will discuss future work on improving area efficiency, performance, and compilation in the context of CGRAs.

7.1. QUANTIFYING THE PROGRESS MADE



(a) Energy savings. *(b) Efficiency.* *(c) Speedup.*

Figure 7.1: Energy savings, efficiency, and speedup v. MSP430 for neural network inference. RIPTIDE saves 1676× energy and is 235× faster v. the MSP430.

Fig. 7.1 quantifies the significant progress that this thesis has made towards a new ULP, energy-minimal sensor system stack. We compare an MSP430, Cortex-M3 (STM32L15RE [9]), and our scalar design to RIPTIDE running neural network inference (the network is a derivative of LeNet [134]). For the MSP430 and the Cortex-M3, we run the network on-device to measure latency and estimate power using a digital mulitimeter. For our scalar design and RIPTIDE, we implement each using a high-threshold voltage, industrial-grade process and use a post-synthesis annotated switching model to estimate power (same methodology as in Sec. 6.4). Fig. 7.1a shows energy savings: our scalar design is already an aggressive baseline, saving 292× energy v. MSP430 and Cortex-M3; RIPTIDE improves even further, saving a massive 1913×. Even adjusting for process technology, RIPTIDE saves ≈ 323× energy v. MSP430, suggesting that a significant portion of energy savings comes from the architecture and not from process technology scaling. Fig. 7.1b and Fig. 7.1c show similar trends in efficiency and speedup. RIPTIDE achieves $64 MOPS/mW$, 6.7× greater than our scalar design, which is already 73× better than the Cortex-M3. It is also 146×, 12.5×, and 7.6× faster than the MSP430, Cortex-M3, and our scalar design, respectively. Together the energy savings, efficiency, and speedup of RIPTIDE v. MSP430, Cortex-M3, and our scalar design represent significant progress and make RIPTIDE an ideal platform for new applications in the ULP domain.

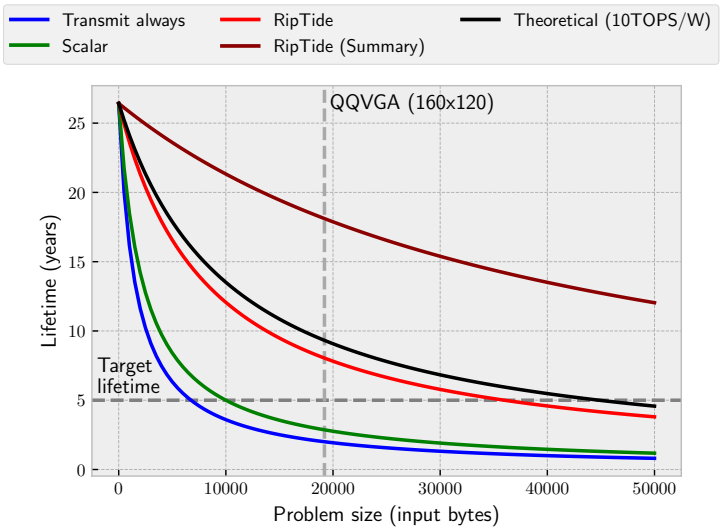


Figure 7.2: Device lifetime as a function of problem size. A RipTide-based system smartly discarding uninteresting data achieves similar lifetime to a hypothetical system with a 10TOPS/W processor. RipTide (Summary), which only transmits a summary of captured data, increases lifetime even more.

7.1.1 Is compute energy efficiency still a bottleneck?

With the progress that has been made, the natural question is whether the energy efficiency of compute is still the bottleneck in RipTide-based ULP sensor systems. To answer this question, we modeled device lifetime of such systems as a function of problem size with goal of achieving a five year lifetime processing a QQVGA (160×120) frame once every five seconds. Device lifetime is directly related to energy efficiency (in battery-powered systems), and problem size is a proxy for the amount of compute required and the amount of data that needs to be transmitted. Fig. 7.2 shows two RipTide-based systems (red) as well as a system that transmits all data collected (blue), a system similar to Sonic that relies on a scalar MCU (green), and a hypothetical system that achieves 10TOPS/W (black). Each system is composed of 1) a single AA battery, 2) a sensor (similar to the HM01B0 ULP camera), 3) an ULP MCU (e.g., scalar, RipTide, or hypothetical), and 4) a bluetooth (BLE) radio. Besides the transmit-all configuration, each system is modeled to run a neural network similar to those in Sonic to smartly discard uninteresting data. Assuming interesting events are somewhat rare, this means these systems transmit infrequently, approximately every 20 minutes (v. 5s interval of sensor readings). Finally, RipTide (Summary) models an application that only sends a short summary of captured data; this could be the result of classification (i.e., class label) or it could be a fragment of data deemed interesting.

RipTide-based systems achieve the five-year lifetime target while processing QQVGA frames, while the transmit-all system and the Sonic-like, scalar-MCU system fall well short. In fact, the RipTide-based system (lighter red) actually gets quite close to the hypothetical system that achieves a much higher efficiency of 10 TOPS/W. This suggests that the energy efficiency of compute is no longer the bottleneck — rather, the energy efficiency of the radio is now more important. RipTide (Summary) (dark red) supports this conclusion; efficient onboard compute that summarizes and/or compresses sensor data, minimizes communication energy and extends lifetime even more. Thus, RipTide has solved the problem of compute efficiency for a variety of applications, opening the door to exploring to new problems.

7.2. FUTURE RESEARCH DIRECTIONS

Many open problems remain in ULP sensor systems and specifically power-constrained CGRAs. They can be classified into three categories: increasing area efficiency, scaling performance, and expanding programmability. Progress on each of these fronts will be important to drive adoption.

7.2.1 Area

RipTide's programmability and energy efficiency come at the cost of area. RipTide is larger than several ASICs (e.g., `dmm`, `fft`, and `sort`) combined. Thus, improving area efficiency is an important goal for future designs, especially to lower fabrication costs and make even larger, more-capable fabrics competitive.

Time-multiplexing: Area efficiency can be improved by increasing resource utilization through time-multiplexing. RipTide only supports a single operation per PE, so resources are underutilized when operations fire rarely. This is especially true for operations in outer loops. Mapping and time-multiplexing multiple operations on the same PE, will significantly improve utilization. This increases area efficiency, makes it possible to map larger programs onto smaller fabrics, and potentially decreases compilation times by simplifying mapping constraints.

There are two possible approaches: fine-grain [248] and coarse-grain [174]. In the fine-grain approach, multiple operations share a single PE and incoming operands trigger reconfiguration of the PE. In the coarse-grain approach, a kernel is split into multiple subkernels with the CGRA switching between them when progress is stalled on the active subkernel. The key to amortizing the cost of coarse-grain reconfiguration is for the CGRA to buffer several inputs so when reconfiguration occurs, multiple instances of the subkernel can be initiated.

There is a role for the compiler in both the fine- and coarse- grained approaches. For fine-grain multiplexing, the compiler needs to determine which operations should multiplex. This may mean reasoning about the program's critical path, only time-multiplexing operations off this path. For coarse-grain multiplexing, the compiler needs to determine where to split a kernel, while considering live-in and live-out values and the cost of reconfiguration.

Alternative control-flow models: Area efficiency can also be improved by reducing the resources needed by a program. In particular, supporting arbitrary control flow requires allocating a significant amount of resources (especially routing) to control-flow operations. Alternative control-flow models, therefore, may improve area efficiency if they require fewer operations. RipTide's steering-based (ϕ^{-1}) control-flow model requires steering gates for every incoming value in untaken branches. Instead, there may be situations where selection (ϕ) or even predication use less resources and/or achieve better performance. It is also possible to mix these control-flow models to minimize area, maximize resource utilization, or even increase performance.

Optimizing the topology: Finally, an additional way to improve area efficiency is to optimize the CGRA topology and resource mix so that they better match the requirements of applications [26, 155]. For example, a CGRA fabric could be specialized [247] for a set of applications, like linear algebra kernels, to significantly reduce area ($> 2\times$). This entails merging the computation graphs of a set of applications to form a minimally sized common graph that can be used to generate CGRA hardware. The common graph, however, is only as representative as the set of applications are. Thus, there is a trade-off between area and programmability that will require collaboration between hardware designers and application programmers to strike the right balance.

7.2.2 Performance

Performance is another dimension future designs could seek to improve. CGRAs like RipTide already outperform (in-order) scalar and vector designs by unlocking a large amount instruction-level parallelism. But there are other ways to improve performance, including thread-level parallelism, speculation, caching (and other memory hierarchy optimizations) and compiler optimizations (e.g., loop unrolling, etc.). Scaling up the performance of RipTide-like designs could make them competitive in different computing domains like wearables or mobile phones.

Parallelization

Exploiting thread-level parallelism is a natural next step to boost performance for power-constrained CGRAs. Multiple threads could be mapped to the same fabric, running in parallel on different PEs or time-multiplexing onto the same PEs. Multiple threads could also be mapped to separate fabrics. RipTide is tiny ($< 0.5mm^2$) compared to even a wearable CPU and could be replicated hundreds of times to support many threads in parallel. This design offers two significant benefits over vector processors/GPUs. First, threads would not run in lockstep. This means threads would better exploit instruction-level parallelism and would be able to diverge without underutilizing resources. Second, the design could scale better v. vector processors/GPUs, since it does not rely on a monolithic register file. With that said, there are many interesting challenges with this design. Questions remain regarding thread scheduling and synchronization, memory-hierarchy design, and the programming interface, to name a few.

Programming the memory

As performance is scaled with more threads, memory becomes a bottleneck. New memory hierarchies and programming models need to be developed to improve data placement, which will minimize data movement and maximize utilization of available memory bandwidth.

Memory hierarchy: Future CGRA memory hierarchies will be composed of both caches and scratchpads. Caches simplify the programmer's job, but raise the question of coherence. Coherence is especially challenging since CGRAs (potentially running many different threads) may have tens or even hundreds of memory accessors. Scratchpads offer an alternative, but complicate the job of the programmer. The CGRA needs to a provide a rich set of synchronization primitives so that code is correct and performant.

Programming models: Memory management can be simplified with novel programming models. Using higher-level abstractions (e.g., functional languages or DSLs) might allow the compiler to generate code to manage a hierarchy of scratchpads. These interfaces and/or programmer annotations might also allow the compiler to co-locate data with computation by suggesting efficient ways of partitioning data. This is especially important for future CGRA fabrics that might contains hundreds of PEs. Partitioning memory to physically locate computation close to data improves performance and energy-efficiency by minimizing data movement.

Speculation

Speculation is another lever future CGRA designs could pull to improve performance. The key is to dynamically eliminate sequentializing dependencies without costing significant resources or energy on misspeculations.

Loop speculation: Speculating on loop-carried dependencies would improve performance by minimizing initiation interval. It would also increase resource utilization as

there would be more work in flight. Further, RipTide already maintains the order of loop iterations so misspeculations could be handled by restoring fabric state from a checkpoint of an earlier loop iteration. However, memory accesses become more expensive: speculative stores would need to be buffered and subsequent loads checked against these buffered stores.

Memory speculation: Another form of speculation that would boost performance is memory speculation. It is difficult for the compiler to prove memory operations will not alias, resulting in extra dependencies between operations. These dependencies, like loop-carried dependencies in loop speculation, sequentialize execution and may lengthen critical paths. However, many memory operations, at runtime, will not access the same locations, so enforcing the dependencies between them wastes time and resources. Instead, a transactional memory system that speculates on whether memory operations will alias could make sense. Detecting true aliases and recovering after an aborted transaction, though, are challenging especially in CGRAs where, unlike e.g., OoO cores, there may be no centralized control to initiate a flush.

Code refactoring

The programmer and the compiler can also play a role in increasing performance, by applying known code transformations in the new context of CGRAs, to refactor code. These transformations include loop-unrolling to increase instruction-level parallelism (which has been explored for CGRAs [119]), automatic loop parallelization and vectorization, and code-motion to flatten dependence chains and reduce loop initiation intervals. RipTide provides evidence that these transformations can be effective; RipTide with hand-coded loop-unrolling on `dmm` is 1.86× faster and is 29% more efficient than without.

The CGRA context does change the way these transformations are applied v. traditional compilers targeting von Neumann machines, as there is a strict limit to the number of operations that can be mapped onto the fabric. There will be a need for an accurate cost model with a feedback loop between the middle-end optimizer of a compiler and its backend-mapper (that solves for the mapping to CGRA hardware) to inform which of the transformations are undertaken.

7.2.3 Compilation

Improving compilation for CGRAs is another important topic for future work. While RipTide made significant progress on this, it has a few limitations, including no support for methods and single-entry-multi-exit loops, that could be addressed in future iterations of its compiler. There are also some more fundamental questions regarding how programs are represented, how to accelerate the mapping of a program to hardware, and the correct programming interface.

Choosing the right IR

CGRA compilation, like VLIW compilation, is dependent on the available hardware resources (i.e. composition of the CGRA fabric). This means that if there are several devices with different CGRA fabrics, a program needs to be recompiled for each. Further low-level hardware details need to be exposed to the compiler so that it can correctly optimize and map a program. This complicates not only compiler development, but also increases compilation complexity, reduces portability, and makes virtualization (which is especially important in the modern datacenter computing context) challenging since the programmer must compile for a specific fabric.

Instead, compilation could be split into two phases, one phase that is device-agnostic and another that optimizes for a specific device. This is similar to the approach taken for Nvidia GPUs; first CUDA programs are compiled to PTX [6]

(device-agnostic byte-code) and then the PTX is optimized on the host machine for the target device. For CGRAs, compilation would be split between dataflow compilation and mapping. Dataflow compilation converts a program to an intermediate representation (IR) that can be targeted to many different CGRAs with potentially different ISAs. Then, mapping schedules and optimizes the IR to a specific fabric and its ISA. The key is developing an IR for a program that captures the right amount of information to facilitate this mapping. Too simple, and the mapper may not have enough information to perform low-level, device-specific optimizations. Too complex, and the mapper will be complicated, making mapping slow and development costly. And at the same time, the IR should maintain backwards compatibility, while being extensible as hardware matures and gains new capabilities.

Scalability

Mapping a program to a specific CGRA fabric is challenging, since it reduces to finding a colored subgraph isomorphism between a program's computation graph and CGRA hardware resources. For relatively small programs (small number of operations), RipTide's ILP- and SAT- based mappers do well to find valid mappings, but as program size increases mapping times increase superlinearly. For example, while DMV (12 operations) takes $\approx 10s$ to map onto a 6×6 fabric, DFS (50 operations) takes just under three minutes ($\approx 18\times$ more time for $4.1\times$ more operations) with SAT and more than an hour using ILP. New encoding methods and ways of solving these constraint-based problem are required.

Heuristic-based approaches have also showed promise for CGRAs [22, 121, 138, 141, 250, 258] and are widely used for place & route for ASIC design and FPGAs [37]. However, they may require an overprovisioning of resources — routes in particular — and may be suboptimal v. constraint-based formulations. This might be a necessary tradeoff, though, as a boost in compilation performance can enable future compilers to use mapping in a feedback loop to choose which optimization (e.g., loop unrolling, loop vectorization, etc.) passes to apply to maximize/minimize different application objectives. Further, depending on the optimization, the suboptimality of heuristic-based mapping may be moot.

Changing the programming interface

The programming interface also affects compilation by dictating the amount and type of information supplied to the compiler. Program annotations or domain specific languages (DSLs) can better expose program structure, enabling the compiler to reduce memory dependences (by simplifying alias analysis), increase parallelization, or pipeline loops to name a few optimizations. Different programming interfaces may also simplify the programmer's job by providing higher-level primitives.

However, rather than redesigning the entire system for a specific interface or DSL, future interfaces should be built on top of systems with general-purpose program support like what RipTide provides. Compilers can use the general-purpose support as a crutch when programs only partially fit a future DSL's model. This increases the applicability of DSLs and prevents one-off designs that require significant hardware changes when application requirements change.

Chapter 8

Conclusion

This thesis has presented a new ULP sensor system stack that will enable future applications of "beyond-the-edge" intelligence. The overarching goal has been to reduce energy at each level of the stack without sacrificing programmability. From software to silicon, we have contributed the following systems to meeting this goal:

- SONIC is a machine inference software runtime system for intermittently operating, energy-harvesting devices. It leverages the regular structure of inference to reduce the costs of guaranteeing correct execution under frequent power failures. SONIC was the first to demonstrate inference on intermittent, energy-harvesting devices and showed the importance of accurate, local inference. It also exposed the inefficiencies of existing commercial devices and stressed the need for new architectures.

- MANIC was our first response to the need for new architectures. MANIC developed vector-dataflow execution to amortize the cost of instruction fetch (vector execution) and minimize data supply energy (VRF accesses) by forwarding intermediates directly from producers to consumers (dataflow execution). MANIC-SILICON showed the benefits of the vector-dataflow execution in a real testchip prototype, but also exposed the limitations of the implementation, which wasted energy reconfiguring shared pipeline resources from cycle-to-cycle.

- SNAFU generates ULP CGRAs that address MANIC's limitations. SNAFU implements spatial-vector-dataflow execution, where each PE is assigned a single operation for the duration of a kernel's execution, to eliminate the cost of reconfiguring shared pipeline resources. It also implements a bufferless NoC to reduce data supply energy and supports asynchronous dataflow without expensive tag-token matching. At the same time, SNAFU maximizes flexibility by taking a "bring-your-own-functional" unit approach that allows designers to easily integrate custom operations. SNAFU is competitive with ASIC designs while maintaining a high-degree of programmability. Through iterative and selective specialization, SNAFU can further close the gap to ASIC designs.

- RIPTIDE observes that, the more computation offloaded to an ULP CGRA fabric, the more efficient the overall system. It develops a dataflow compiler and ULP CGRA architecture that target programs written in C to reduce programmer effort (v. systems like SNAFU that require hand-coded assembly) in offloading computation. It introduces a general-purpose control-flow model to support a wide variety of program idioms, including irregular memory accesses and deeply-nested loops. Control-flow operations are conveniently and efficiently implemented using existing resources in RIPTIDE's on-chip network. RIPTIDE achieves the best of both worlds — it narrows the gap to ASIC designs even more

than Snafu, while improving on programmability. Compared to COTS MCUs, RipTide demonstrates the significant progress made by this thesis, achieving 2–3 orders of magnitude better energy and performance.

These contributions form the basis of a new energy-minimal, ULP sensor system stack. They show that extreme energy-efficiency can be achieved without significantly compromising on programmability. This is the power of rethinking the entire stack; optimizing at multiple levels at once, reduces energy, increases performance, and ultimately enables new applications.

Appendix A

Constraint-based scheduling

A.1. SNAFU'S MAPPER

Input	Explanation
V	Set of DFG vertices
E	Set of DFG edges
N	Set of hardware nodes (PEs)
R	Set of hardware routers
L	Set of hardware links
$C_{vn}(V,N) = 1$ if v can map to n	Vertex-node compatibility matrix
$H_{ln}(L,N) = 1$ if l originates from n	Link-to-node matrix
$H_{nl}(N,L) = 1$ if l comes from n	Node-to-link matrix
$H_{lr}(L,R) = 1$ if l originates from r	Link-to-router matrix
$H_{rl}(R,L) = 1$ if l comes from r	Router-to-link matrix
Variable	**Explanation**
$M_{vn}(V,N) = 1$ if v is mapped to n	Vertex-to-node matrix
$M_{el}(E,L) = 1$ if e is mapped to l	Edge-to-link matrix

Table A.1: Inputs & variables of SNAFU's ILP formulation.

Table A.1 lists the inputs and variables of the SNAFU's ILP formulation for mapping. The goal of the mapper is to solve for M_{vn} and M_{el}, which map a DFG's vertices to hardware PEs (hardware nodes) and a DFG's edges to hardware links, respectively. Matrix C_{vn} captures the compatibility of a DFG's vertex-to-hardware node (i.e., a memory operation must be mapped to a memory PE). Matrices H_{nl}, H_{ln}, H_{rl}, H_{lr}, describe the topology of the CGRA fabric by specifying the connectedness of links to hardware nodes and routers.

Objective: ***minimize*** $\sum_{e\in E, l\in L} M_{el}(e,l)$ *subject to*

Constraint	Explanation
$\forall v \in V, n \in N, M_{vn}(v,n) \leq C_{vn}(v,n)$	Vertices are mapped to compatible nodes
$\forall v \in V, \sum_{n\in N} M_{vn}(v,n) = 1$	Every vertex must be mapped to a node
$\forall n \in N, \sum_{v\in V} M_{vn}(v,n) \leq 1$	No node can be used by more than one vertex
$\forall e \in E, r \in R, \sum_{l\in L} M_{el}(e,l) H_{lr}(l,r) = \sum_{l\in L} M_{el}(e,l) H_{rl}(r,l)$	Flow into a router must equal the flow out
$\forall e \in E, n \in N, \sum_{l\in L} M_{e,l}(e) H_{nl}(n,l) = M_{vn}(src(e),n)$	If a vertex is mapped to a node, then the output edges are mapped to outgoing links
$\forall e \in E, n \in N, \sum_{l\in L} M_{e,l}(e) H_{ln}(l,n) = M_{vn}(dst(e),n)$	If a vertex is mapped to a node, then the input edges are mapped to incoming links
$\forall l \in L, e_1 \in E, M_{el}(e_1,l) + max_{e_2\in E \mid src(e_1)\neq src(e_2)} M_{el}(e_2,l) \leq 1$	Edges that do not share the same source are not mapped to the same links

$src(e) := v \in V$ and v is the source of e $\mid$ $dst(e) := v \in V$ and v is the destination of e

Table A.2: SNAFU's ILP formulation.

Table A.2 describes SNAFU's (binary) ILP formulation for mapping. The formulation minimizes average routing distance given the constraints in the table.

A.2. RIPTIDE'S MAPPER

Input	Explanation
V	Set of DFG vertices
E	Set of DFG edges
N	Set of hardware nodes (PEs & CF-modules)
F	Set of CF-modules $F \subset N$
R	Set of hardware routers
L	Set of hardware links
$C_{el}(E, L) = 1$ if e can map to l	Edge-link compatibility matrix
$C_{vn}(V, N) = 1$ if v can map to n	Vertex-node compatibility matrix
$H_{ln}(L, N) = 1$ if l originates from n	Link-to-node matrix
$H_{nl}(N, L) = 1$ if l comes from n	Node-to-link matrix
$H_{lr}(L, R) = 1$ if l originates from r	Link-to-router matrix
$H_{rl}(R, L) = 1$ if l comes from r	Router-to-link matrix
Variable	**Explanation**
$M_{vn}(V, N) = 1$ if v is mapped to n	Vertex-to-node matrix
$M_{el}(E, L) = 1$ if e is mapped to l	Edge-to-link matrix

Table A.3: Inputs & variables of RipTide's ILP & SAT formulations.

Table A.3 lists the inputs and variables of RipTide's SAT and ILP formulations for mapping. The inputs and variables are quite similar to those for Snafu's formulation. The goal of the mappers is to solve for M_{vn} and M_{el}, which map a DFG's vertices to hardware nodes and a DFG's edges to hardware links, respectively. However, there are two primary additions v. Snafu's formulation. First, RipTide adds matrix C_{el}, which captures the compatibility of a DFG's edge-to-hardware link, and is used to ensure that incoming and outgoing ports match. This is unnecessary in Snafu because all operations in Snafu have a single outgoing value and the Snafu μcore includes a router that makes an all-to-all connection between incoming network ports and internal FU ports. Second, RipTide makes a distinction between PEs and CF-modules (both hardware nodes) because CF-modules, when unused, can pass through a signal.

A.2.1 ILP formulation

Objective: *minimize* $\sum_{e \in E, l \in L} M_{el}(e, l)$ *subject to*

Constraint	Explanation
$\forall e \in E, l \in L, M_{el}(e, l) \leq C_{el}(e, l)$	Edges are mapped to compatible links
$\forall v \in V, n \in N, M_{vn}(v, n) \leq C_{vn}(v, n)$	Vertices are mapped to compatible nodes
$\forall v \in V, \sum_{n \in N} M_{vn}(v, n) = 1$	Every vertex must be mapped to a node
$\forall n \in N, \sum_{v \in V} M_{vn}(v, n) \leq 1$	No node can be used by more than one vertex
$\forall e \in E, r \in R, \sum_{l \in L} M_{el}(e, l) H_{lr}(l, r) = \sum_{l \in L} M_{el}(e, l) H_{rl}(r, l)$	Flow into a router must equal the flow out
$\forall e \in E, n \in N \vert n \notin F, \sum_{l \in L} M_{e,l}(e) H_{nl}(n, l) = M_{vn}(src(e), n)$	If a vertex is mapped to a non-CF node, then the output edges are mapped to outgoing links
$\forall e \in E, n \in N \vert n \notin F, \sum_{l \in L} M_{e,l}(e) H_{ln}(l, n) = M_{vn}(dst(e), n)$	If a vertex is mapped to a non-CF node, then the input edges are mapped to incoming links
$\forall l \in L, e_1 \in E, M_{el}(e_1, l) + max_{e_2 \in E \vert src(e_1) \neq src(e_2)} M_{el}(e_2, l) \leq 1$	Edges that do not share the same source are not mapped to the same links
$\forall e \in E, n \in F, \sum_{l \in L} M_{el}(e, l) H_{ln}(l, n) + \sum_{v \in V} M_{vn}(v, n) \geq \sum_{l \in L} M_{el}(e, l) H_{nl}(n, l)$	Unused CF-modules can pass through edges
$\forall e \in E, n \in F, \sum_{l \in L} M_{el}(e, l) H_{ln}(l, n) \leq \sum_{v \in V} M_{vn}(v, n) + \sum_{l \in L} M_{el}(e, l) H_{nl}(n, l)$	Unused CF-modules can pass through edges
$\forall e \in E, n \in F, \sum_{l \in L} M_{el}(e, l) H_{nl}(n, l) \geq M_{vn}(src(e), n)$	If a vertex is mapped to a CF node, then the output edges are mapped to outgoing links
$\forall e \in E, n \in F, \sum_{l \in L} M_{el}(e, l) H_{ln}(l, n) \geq M_{vn}(dst(e), n)$	If a vertex is mapped to a CF node, then the input edges are mapped to incoming links

$src(e) := v \in V$ and v is the source of e $\mid$ $dst(e) := v \in V$ and v is the destination of e

Table A.4: RipTide's ILP formulation.

Table A.4 describes RipTide's (binary) ILP formulation. It is similar to Snafu's formulation, sharing several constraints and the same objective of minimizing average routing distance. However, RipTide's formulation draws a distinction between CF-modules and PEs that requires additional constraints to allow unused CF-modules to pass through values and used CF-modules to effectively act in the formulation as if they were PEs. It also ensures that ports match by constraining edges to certain hardware links according to matrix, C_{el}.

A.2.2 SAT formulation

Clause	Explanation
$\forall e \in E, l \in L \vert C_{el}(e,l) = 0, \neg M_{el}(e,l)$	Edges are mapped to compatible links
$\forall v \in V, n \in N \vert C_{vn}(v,n) = 0, \neg M_{vn}(v,n)$	Vertices are mapped to compatible nodes
$\forall v \in V, ExactlyOne(\{M_{vn}(v,n) \vert n \in N\})$	Every vertex must be mapped to a node
$\forall n \in N, AtMostOne(\{M_{vn}(v,n) \vert v \in V\})$	No node can be used by more than one vertex
$\forall r \in R, e \in E, \vee_{l \vert H_{rl}(r,l)} M_{el}(e,l) \iff \vee_{H_{lr}(l,r)} M_{el}(e,l)$	An edge mapped to incoming link to a router must also be mapped to an outgoing link
$\forall r \in R, e \in E, AtMostOne(\{M_{el}(e,l) \vert l \in L \text{ and } H_{rl}(r,l)\})$	An edge can only be mapped to a single outgoing link of a router
$\forall e \in E, n \in N \vert n \notin F, \vee_{l \vert H_{nl}(n,l)} M_{el}(e,l) \iff M_{vn}(src(e),n)$	If a vertex is mapped to a non-CF node, then the input edges are mapped to incoming links
$\forall e \in E, n \in N \vert n \notin F, \vee_{l \vert H_{ln}(l,n)} M_{el}(e,l) \iff M_{vn}(dst(e),n)$	If a vertex is mapped to a non-CF node, then the output edges are mapped to outgoing links
$\forall l \in L, e_1 \in E, e_2 \in E \vert src(e_1) \neq src(e_2), \neg M_{el}(e_1,l) \vee \neg M_{el}(e_2,l)$	Edges that do not share the same source are not mapped to the same links
$\forall e \in E, n \in F, K_{nl}(e,n) \vee \neg M_{vn}(src(e),n)$	If a vertex is mapped to a CF node, then the output edges are mapped to outgoing links
$\forall e \in E, n \in F, K_{ln}(e,n) \vee \neg M_{vn}(dst(e),n)$	If a vertex is mapped to a CF node, then the input edges are mapped to incoming links
$(\forall e \in E, n \in F, (K_{ln}(e,n) \vee K_n \vee \neg K_{nl}(e,n)) \wedge (\neg K_{ln}(e,n) \vee K_n(n) \vee K_{nl}(e,n)) \wedge (\neg K_{ln}(e,n) \vee \neg M_{vn}(src(e),n))$	Unused CF-modules can pass through edges
$\forall e \in E, n \in F, l \in L \vert H_{ln}(l,n), \neg M_{el}(e,l) \vee K_{ln}(e)$	An edge mapped to an output link of CF-module cannot be mapped to an input of the CF-module

$src(e) := v \in V$ and v is the source of e $\mid$ $dst(e) := v \in V$ and v is the destination of e $\mid$ $K_{nl}(e,n) := \vee_{l \vert H_{nl}(n,l)} M_{el}(e,l)$

$K_{ln}(e,n) := \vee_{l \vert H_{ln}(l,n)} M_{el}(e,l)$ $\mid$ $K_n(n) := \vee_{v \in V} M_{vn}(v,n)$

Table A.5: RipTide's SAT formulation.

Table A.5 describes RipTide's SAT formulation. Since there is no objective, the formulation may yield longer routes, duplicate routes or routes with cycles. We post-process the routes to find the shortest between two nodes.

Bibliography

[1] "Cortex-m0." [Online]. Available: https://developer.arm.com/ip-products/processors/cortex-m/cortex-m0

[2] "Genus synthesis solution." [Online]. Available: https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/genus-synthesis-solution.html

[3] "Hm01b0 ultralow power cis." [Online]. Available: https://www.himax.com.tw/products/cmos-image-sensor/always-on-vision-sensors/hm01b0/

[4] "Innovus implementation system." [Online]. Available: https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/soc-implementation-and-floorplanning/innovus-implementation-system.html

[5] "Joules rtl power simulation." [Online]. Available: https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/power-analysis/joules-rtl-power-solution.html

[6] "Parallel thread execution isa version 7.7." [Online]. Available: https://docs.nvidia.com/cuda/parallel-thread-execution/index.html

[7] "Powercast p2110b." [Online]. Available: http://www.powercastco.com/wp-content/uploads/2016/12/P2110B-Datasheet-Rev-3.pdf

[8] "Powercaster transmitter." [Online]. Available: http://www.powercastco.com/wp-content/uploads/2016/11/User-Manual-TX-915-01-Rev-A-4.pdf

[9] "Stm32l152re." [Online]. Available: https://www.st.com/en/microcontrollers-microprocessors/stm32l152re.html

[10] "Xcelium logic simulation." [Online]. Available: https://www.cadence.com/en_US/home/tools/system-design-and-verification/simulation-and-testbench-verification/xcelium-simulator.html

[11] "What is lorawan specification?" Oct 2021. [Online]. Available: https://lora-alliance.org/about-lorawan/

[12] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[13] A. V. Aho, M. R. Garey, and J. D. Ullman, "The transitive reduction of a directed graph," *SIAM Journal on Computing*, vol. 1, no. 2, pp. 131–137, 1972. [Online]. Available: https://doi.org/10.1137/0201008

[14] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, 2016.

[15] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer cnn accelerators," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, 2016.

[16] Angus Galloway, "Tensorflow XNOR-BNN," https://github.com/AngusG/tensorflow-xnor-bnn, 2018.

[17] ARM, "Arm neon," 2019. [Online]. Available: https://developer.arm.com/architectures/instruction-sets/simd-isas/neon

[18] K. Asanovic, J. Beck, B. Irissou, B. E. Kingsbury, and J. Wawrzynek, "T0: A single-chip vector microprocessor with reconfigurable pipelines," in *ESSCIRC 22*.

[19] H. Asghari Esfeden, F. Khorasani, H. Jeon, D. Wong, and N. Abu-Ghazaleh, "Corf: Coalescing operand register file for gpus," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 701–714.

[20] G. Aşılıoğlu, Z. Jin, M. Köksal, O. Javeri, and S. Önder, "Lazy superscalar," in *ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015.

[21] O. Bachmann, P. S. Wang, and E. V. Zima, "Chains of recurrences—a method to expedite the evaluation of closed-form functions," in *Proceedings of the International Symposium on Symbolic and Algebraic Computation*, ser. ISSAC '94. New York, NY, USA: Association for Computing Machinery, 1994, p. 242–249. [Online]. Available: https://doi.org/10.1145/190347.190423

[22] M. Balasubramanian and A. Shrivastava, "Pathseeker: a fast mapping algorithm for cgras," in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2022, pp. 268–273.

[23] J. Balfour, W. Dally, D. Black-Schaffer, V. Parikh, and J. Park, "An energy-efficient processor architecture for embedded systems," *IEEE Computer Architecture Letters*, vol. 7, no. 1, 2008.

[24] J. D. Balfour, W. J. Dally, M. Horowitz, and C. Kozyrakis, "Efficient embedded computing," Ph.D. dissertation, 2010.

[25] D. Balsamo, A. Weddell, A. Das, A. Arreola, D. Brunelli, B. Al-Hashimi, G. Merrett, and L. Benini, "Hibernus++: A self-calibrating and adaptive system for transiently-powered embedded devices," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. PP, no. 99, pp. 1–1, 2016.

[26] T. K. Bandara, D. Wijerathne, T. Mitra, and L.-S. Peh, "Revamp: A systematic framework for heterogeneous cgra realization," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 918–932. [Online]. Available: https://doi.org/10.1145/3503222.3507772

[27] D. Bankman, L. Yang, B. Moons, M. Verhelst, and B. Murmann, "An always-on 3.8μ j / 86% cifar-10 mixed-signal binary cnn processor with all memory on chip in 28-nm cmos," *IEEE Journal of Solid-State Circuits*, vol. 54, no. 1, pp. 158–172, 2018.

[28] L. Bettini, P. Crescenzi, G. Innocenti, and M. Loreti, "An environment for self-assessing java programming skills in first programming courses," in *IEEE International Conference on Advanced Learning Technologies, 2004. Proc.*., 2004.

[29] S. Bhattacharya and N. D. Lane, "Sparsification and separation of deep learning layers for constrained resource inference on wearables," in *Proc. of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM*, 2016.

[30] A. Biere, "Yet another local search solver and Lingeling and friends entering the SAT Competition 2014," in *Proc. of SAT Competition 2014 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Series of Publications B, A. Balint, A. Belov, M. Heule, and M. Järvisalo, Eds., vol. B-2014-2. University of Helsinki, 2014, pp. 39–40.

[31] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger, "CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020," in *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Report Series B, T. Balyo, N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda, Eds., vol. B-2020-1. University of Helsinki, 2020, pp. 51–53.

[32] D. Bol, M. Schramme, L. Moreau, T. Haine, P. Xu, C. Frenkel, R. Dekimpe, F. Stas, and D. Flandre, "19.6 a 40-to-80mhz sub-4μw/mhz ulv cortex-m0 mcu soc in 28nm fdsoi with dual-loop adaptive back-bias generator for 20μs wake-up from deep fully retentive sleep mode," in *ISSCC*, 2019.

[33] A. Bracy, P. Prahlad, and A. Roth, "Dataflow mini-graphs: Amplifying superscalar capacity and bandwidth," in *37th International Symposium on Microarchitecture (MICRO-37'04)*, 2004.

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

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

[36] M. Buettner, B. Greenstein, and D. Wetherall, "Dewdrop: An energy-aware task scheduler for computational RFID," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Mar. 2011.

[37] D. Chen, J. Cong, P. Pan *et al.*, "Fpga design automation: A survey," *Foundations and Trends® in Electronic Design Automation*, vol. 1, no. 3, pp. 195–330, 2006.

[38] D.-K. Chen and P.-C. Yew, "Redundant synchronization elimination for doacross loops," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 5, pp. 459–470, 1999.

[39] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proc. of the 19th intl. conf. on Architectural Support for Programming Languages and Operating Systems*, 2014.

[40] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Proc. of the 43rd annual Intl. Symp. on Computer Architecture (Proc. ISCA-43)*, 2016.

[41] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun *et al.*, "Dadiannao: A machine-learning supercomputer," in *Proc. of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

[42] H. Cherupalli, H. Duwe, W. Ye, R. Kumar, and J. Sartori, "Bespoke processors for applications with ultra-low area and power constraints," in *ISCA 44*, 2017.

[43] S. A. Chin and J. H. Anderson, "An architecture-agnostic integer linear programming approach to cgra mapping," in *Proceedings of the 55th Annual Design Automation Conference*, 2018, pp. 1–6.

[44] S. A. Chin, N. Sakamoto, A. Rui, J. Zhao, J. H. Kim, Y. Hara-Azumi, and J. Anderson, "Cgra-me: A unified framework for cgra modelling and exploration," in *2017 IEEE 28th international conference on application-specific systems, architectures and processors (ASAP)*. IEEE, 2017, pp. 184–189.

[45] F. Chollet, "Xception: Deep learning with depthwise separable convolutions."

[46] A. Colin, G. Harvey, B. Lucia, and A. P. Sample, "An energy-interference-free hardware-software debugger for intermittent energy-harvesting systems," *SIGOPS Oper. Syst. Rev.*, vol. 50, no. 2, Mar. 2016. [Online]. Available: http://doi.acm.org/10.1145/2954680.2872409

[47] A. Colin and B. Lucia, "Chain: Tasks and channels for reliable intermittent programs," in *Proc. of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2016.

[48] A. Colin, E. Ruppel, and B. Lucia, "A reconfigurable energy storage architecture for energy-harvesting devices," in *ASPLOS 23*, 2018.

[49] A. Colin, E. Ruppel, and B. Lucia, "A reconfigurable energy storage architecture for energy-harvesting devices," in *ASPLOS*, 2018.

[50] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, H. Huang, and G. Reinman, "Composable accelerator-rich microprocessor enhanced for adaptivity and longevity," in *International Symposium on Low Power Electronics and Design (ISLPED)*, 2013, pp. 305–310.

[51] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, and G. Reinman, "Charm: A composable heterogeneous accelerator-rich microprocessor," in *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design*, ser. ISLPED '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 379–384. [Online]. Available: https://doi.org/10.1145/2333660.2333747

[52] J. Cong, H. Huang, C. Ma, B. Xiao, and P. Zhou, "A fully pipelined and dynamically composable architecture of cgra," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, 2014, pp. 9–16.

[53] N. Corporation, "Nvidia's next generation cuda compute architecture: Fermi," 2009.

[54] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1," *arXiv preprint arXiv:1602.02830*, 2016.

[55] Cray Computer, "U.S. Patent 6,665,774," December 2003.

[56] D. Culler, J. Hill, M. Horton, K. Pister, R. Szewczyk, and A. Wood, "Mica: The commercialization of microsensor motes," *Sensor Technology and Design, April*, 2002.

[57] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, p. 451–490, oct 1991. [Online]. Available: https://doi.org/10.1145/115372.115320

[58] V. Dadu, S. Liu, and T. Nowatzki, *PolyGraph: Exposing the Value of Flexibility for Graph Processing Accelerators*. IEEE Press, 2021, p. 595–608. [Online]. Available: https://doi.org/10.1109/ISCA52012.2021.00053

[59] V. Dadu and T. Nowatzki, *TaskStream: Accelerating Task-Parallel Workloads by Recovering Program Structure*. New York, NY, USA: Association for Computing Machinery, 2022, p. 1–13. [Online]. Available: https://doi.org/10.1145/3503222.3507706

[60] V. Dadu, J. Weng, S. Liu, and T. Nowatzki, "Towards general purpose acceleration by exploiting common data-dependence forms," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 924–939.

[61] W. J. Dally, J. Balfour, D. Black-Shaffer, J. Chen, R. C. Harting, V. Parikh, J. Park, and D. Sheffield, "Efficient embedded computing," *Computer*, vol. 41, no. 7, 2008.

[62] S. Das, D. Rossi, K. J. Martin, P. Coussy, and L. Benini, "A 142mops/mw integrated programmable array accelerator for smart visual processing," in *ISCAS*, 2017.

[63] S. Dave, M. Balasubramanian, and A. Shrivastava, "Ramp: Resource-aware mapping for cgras," in *DAC 55*, 2018.

[64] L. De Lathauwer, B. De Moor, and J. Vandewalle, "A multilinear singular value decomposition," *SIAM journal on Matrix Analysis and Applications*, vol. 21, no. 4, 2000.

[65] L. De Lathauwer, B. De Moor, and J. Vandewalle, "On the best rank-1 and rank-(r 1, r 2,..., rn) approximation of higher-order tensors," *SIAM journal on Matrix Analysis and Applications*, vol. 21, no. 4, 2000.

[66] C. De Sa, M. Feldman, C. Ré, and K. Olukotun, "Understanding and optimizing asynchronous low-precision stochastic gradient descent," in *Proc. of the 44th annual Intl. Symp. on Computer Architecture (Proc. ISCA-44)*, 2017.

[67] C. Deng, S. Liao, Y. Xie, K. K. Parhi, X. Qian, and B. Yuan, "Permdnn: Efficient compressed dnn architecture with permuted diagonal matrices," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 189–202.

[68] J. B. Dennis, "Data flow supercomputers," *Computer*, no. 11, 1980.

[69] J. B. Dennis and G. R. Gao, "An efficient pipelined dataflow processor architecture," in *Proc. of the 1988 ACM/IEEE conference on Supercomputing*, 1988.

[70] J. B. Dennis and D. P. Misunas, "A preliminary architecture for a basic data-flow processor," in *ACM SIGARCH Computer Architecture News*, vol. 3, no. 4, 1975.

[71] H. Desai and B. Lucia, "A power-aware heterogeneous architecture scaling model for energy-harvesting computers," *IEEE Computer Architecture Letters*, vol. 19, no. 1, 2020.

[72] S. Diamond and S. Boyd, "CVXPY: A Python-embedded modeling language for convex optimization," *Journal of Machine Learning Research*, vol. 17, no. 83, pp. 1–5, 2016.

[73] C. Ding, S. Liao, Y. Wang, Z. Li, N. Liu, Y. Zhuo, C. Wang, X. Qian, Y. Bai, G. Yuan *et al.*, "Circnn: accelerating and compressing deep neural networks using block-circulant weight matrices," in *Proc. of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017.

[74] A. Dongare, C. Hesling, K. Bhatia, A. Balanuta, R. L. Pereira, B. Iannucci, and A. Rowe, "Openchirp: A low-power wide-area networking architecture," in *Pervasive Computing and Communications Workshops (PerCom Workshops), 2017 IEEE International Conference on*, 2017.

[75] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in *Proc. of the 42nd annual Intl. Symp. on Computer Architecture (Proc. ISCA-42)*, 2015.

[76] M. Duric, O. Palomar, A. Smith, O. Unsal, A. Cristal, M. Valero, and D. Burger, "Evx: Vector execution on low power edge cores," in *DATE*, 2014.

[77] H. Elnawawy, M. Alshboul, J. Tuck, and Y. Solihin, "Efficient checkpointing of loop-based codes for non-volatile main memory," in *Parallel Architectures and Compilation Techniques (PACT), 2017 26th International Conference on*, 2017.

[78] L. Fick, D. Blaauw, D. Sylvester, S. Skrzyniarz, M. Parikh, and D. Fick, "Analog in-memory subthreshold deep neural network accelerator," in *IEEE Custom Integrated Circuits Conference (CICC)*, April 2017, pp. 1–4.

[79] N. Firasta, M. Buxton, P. Jinbo, K. Nasri, and S. Kuo, "Intel avx: New frontiers in performance improvements and energy efficiency," *Intel white paper*, vol. 19, no. 20, 2008.

[80] H. Foundation, "Hsa platform system architecture specification," 2018. [Online]. Available: http://www.hsafoundation.com/standards/

[81] G. Gobieski, A. O. Atli, K. Mai, B. Lucia, and N. Beckmann, "Snafu: an ultra-low-power, energy-minimal cgra-generation framework and architecture," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 1027–1040.

[82] G. Gobieski, N. Beckmann, and B. Lucia, "Intermittent deep neural network inference," in *SysML*, 2018.

[83] G. Gobieski, S. Ghosh, M. Heule, T. Mowry, N. Beckmann, and B. Lucia, "Riptide: A programmable, energy-minimal dataflow compiler and architecture," in *MICRO*, 2022.

[84] G. Gobieski, B. Lucia, and N. Beckmann, "Intelligence beyond the edge: Inference on intermittent embedded systems," in *ASPLOS*, 2019.

[85] G. Gobieski, A. Nagi, N. Serafin, M. M. Isgenc, N. Beckmann, and B. Lucia, "Manic: A vector-dataflow architecture for ultra-low-power embedded systems," in *MICRO*, 2019.

[86] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. R. Taylor, "Piperench: A reconfigurable architecture and compiler," *Computer*, vol. 33, no. 4, 2000.

[87] D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J. Karro, and D. Sculley, "Google vizier: A service for black-box optimization," in *Proc. of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2017.

[88] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim, "Dyser: Unifying functionality and parallelism specialization for energy-efficient computing," *IEEE Micro*, vol. 32, no. 5, 2012.

[89] C. Gupta, A. S. Suggala, A. Goyal, H. V. Simhadri, B. Paranjape, A. Kumar, S. Goyal, R. Udupa, M. Varma, and P. Jain, "Protonn: Compressed and accurate knn for resource-scarce devices," in *International Conference on Machine Learning*, 2017.

[90] S. Gupta, S. Feng, A. Ansari, S. Mahlke, and D. August, "Bundled execution of recurring traces for energy-efficient general purpose processing," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011, pp. 12–23.

[91] Gurobi Optimization, LLC, "Gurobi Optimizer Reference Manual," 2022. [Online]. Available: https://www.gurobi.com

[92] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, 2010.

[93] M. Hamzeh, A. Shrivastava, and S. Vrudhula, "Epimap: Using epimorphism to map applications on cgras," in *Proceedings of the 49th Annual Design Automation Conference*, 2012, pp. 1284–1291.

[94] M. Hamzeh, A. Shrivastava, and S. Vrudhula, "Branch-aware loop mapping on cgras," in *Proceedings of the 51st Annual Design Automation Conference*, 2014, pp. 1–6.

[95] S. Han, X. Liu, H. Mao, J. Pu, A. Pdream, M. A. Horowitz, and W. J. Dally, "Eie: Efficient inference engine on compressed deep neural network," in *Proc. of the 43rd annual Intl. Symp. on Computer Architecture (Proc. ISCA-43)*, 2016.

[96] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization, and huffman coding," in *Proc. of the 5th Intl. Conf. on Learning Representationas (Proc. ICLR'16)*, 2016.

[97] J. R. Hauser and J. Wawrzynek, "Garp: A mips processor with a reconfigurable coprocessor," in *Proceedings. The 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines Cat. No. 97TB100186)*. IEEE, 1997, pp. 12–21.

[98] M. Hempstead, N. Tripathi, P. Mauro, G.-Y. Wei, and D. Brooks, "An ultra low power system architecture for sensor network applications," in *ISCA 32*, 2005.

[99] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[100] J. Hester, T. Peters, T. Yun, R. Peterson, J. Skinner, B. Golla, K. Storer, S. Hearndon, K. Freeman, S. Lord, R. Halter, D. Kotz, and J. Sorber, "Amulet: An energy-efficient, multi-application wearable platform," in *Proc. of the 14th ACM Conference on Embedded Network Sensor Systems*, ser. SenSys '16. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2994551.2994554

[101] J. Hester, L. Sitanayah, and J. Sorber, "Tragedy of the coulombs: Federating energy storage for tiny, intermittently-powered sensors," in *Proc. of the 13th ACM Conference on Embedded Networked Sensor Systems*, 2015.

[102] J. Hester, L. Sitanayah, and J. Sorber, "Tragedy of the coulombs: Federating energy storage for tiny, intermittently-powered sensors," in *SenSys 13*. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2809695.2809707

[103] J. Hester and J. Sorber, "Flicker: Rapid prototyping for the batteryless internet of things," in *SenSys 15*, 2017.

[104] J. Hester, K. Storer, and J. Sorber, "Timely execution on intermi!ently powered ba!eryless sensors," in *Proc. of the 15th ACM Conference on Embedded Network Sensor Systems*, ser. SenSys '17.

[105] M. Hicks, "Clank: Architectural support for intermittent computation," in *Proc. of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080238

[106] M. Hind, M. Burke, P. Carini, and J.-D. Choi, "Interprocedural pointer alias analysis," *ACM Trans. Program. Lang. Syst.*, vol. 21, no. 4, p. 848–894, jul 1999. [Online]. Available: https://doi.org/10.1145/325478.325519

[107] M. Horowitz, "Computing's energy problem (and what we can do about it)," in *ISSCC*, 2014.

[108] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016.

[109] A. Ignatov, "Har." [Online]. Available: https://github.com/aiff22/HAR

[110] T. Instruments, "Msp430fr5994 sla," 2017. [Online]. Available: http://www.ti.com/lit/ds/symlink/msp430fr5994.pdf

[111] T. Instruments, "Low energy accelerator faq," 2018. [Online]. Available: http://www.ti.com/lit/an/slaa720/slaa720.pdf

[112] ISOCPP, "Cpp spec," 2019. [Online]. Available: https://isocpp.org/std/the-standard

[113] N. Jackson, "lab11/permamote," Apr 2019. [Online]. Available: https://github.com/lab11/permamote

[114] H. Jayakumar, A. Raha, and V. Raghunathan, "QuickRecall: A low overhead HW/SW approach for enabling computations across power cycles in transiently powered computers," in *Int'l Conf. on VLSI Design and Int'l Conf. on Embedded Systems*, Jan. 2014.

[115] H. Jeon, G. S. Ravi, N. S. Kim, and M. Annavaram, "Gpu register file virtualization," in *Proc. of the 48th International Symposium on Microarchitecture*, 2015.

[116] N. P. Jouppi, D. H. Yoon, M. Ashcraft, M. Gottscho, T. B. Jablin, G. Kurian, J. Laudon, S. Li, P. Ma, X. Ma *et al.*, "Ten lessons from three generations shaped google's tpuv4i: Industrial product," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 1–14.

[117] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," *arXiv preprint arXiv:1704.04760*, 2017.

[118] T. Karnik, D. Kurian, P. Aseron, R. Dorrance, E. Alpman, A. Nicoara, R. Popov, L. Azarenkov, M. Moiseev, L. Zhao *et al.*, "A cm-scale self-powered intelligent and secure iot edge mote featuring an ultra-low-power soc in 14nm tri-gate cmos," in *ISSCC*, 2018.

[119] M. Karunaratne, A. K. Mohite, T. Mitra, and L.-S. Peh, "Hycube: A cgra with reconfigurable single-cycle multi-hop interconnect," in *DAC*, 2017.

[120] M. Karunaratne, C. Tan, A. Kulkarni, T. Mitra, and L.-S. Peh, "Dnestmap: mapping deeply-nested loops on ultra-low power cgras," in *Proceedings of the 55th Annual Design Automation Conference*, 2018, pp. 1–6.

[121] M. Karunaratne, D. Wijerathne, T. Mitra, and L.-S. Peh, "4d-cgra: Introducing branch dimension to spatio-temporal application mapping on cgras," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2019, pp. 1–8.

[122] B. Keller, M. Cochet, B. Zimmer, J. Kwak, A. Puggelli, Y. Lee, M. Blagojević, S. Bailey, P.-F. Chiu, P. Dabbelt *et al.*, "A risc-v processor soc with integrated power management at submicrosecond timescales in 28 nm fd-soi," *JSSC*, 2017.

[123] C. Kim, M. Chung, Y. Cho, M. Konijnenburg, S. Ryu, and J. Kim, "Ulp-srp: Ultra low power samsung reconfigurable processor for biomedical applications," in *ICFPT*, 2012.

[124] H.-S. Kim and J. E. Smith, "An instruction set and microarchitecture for instruction level distributed processing," in *Proc. 29th Annual International Symposium on Computer Architecture*, 2002.

[125] Y. Kim and R. N. Mahapatra, "Hierarchical reconfigurable computing arrays for efficient cgra-based embedded systems," in *Proceedings of the 46th Annual Design Automation Conference*, 2009, pp. 826–831.

[126] D. Koeplinger, R. Prabhakar, Y. Zhang, C. Delimitrou, C. Kozyrakis, and K. Olukotun, "Automatic generation of efficient accelerators for reconfigurable hardware," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 115–127.

[127] M. Kou, J. Gu, S. Wei, H. Yao, and S. Yin, "Taem: fast transfer-aware effective loop mapping for heterogeneous resources on cgra," in *DAC 57*, 2020.

[128] C. Kozyrakis and D. Patterson, "Overcoming the limitations of conventional vector processors," *ACM SIGARCH Computer Architecture News*, vol. 31, no. 2, 2003.

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

[130] H. Kwon, A. Samajdar, and T. Krishna, "Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects," in *Proc. of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: ACM, 2018. [Online]. Available: http://doi.acm.org/10.1145/3173162.3173176

[131] G. Laput, Y. Zhang, and C. Harrison, "Synthetic sensors: Towards general-purpose sensing," in *Proc. of the 2017 CHI Conference on Human Factors in Computing Systems*, 2017.

[132] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO*, Mar. 2004.

[133] C. R. Lazo, E. Reggiani, C. R. Morales, R. F. Bagué, L. A. V. Vargas, M. A. R. Salinas, M. V. Cortés, O. S. Unsal, and A. Cristal, "Adaptable register file organization for vector processors," *arXiv preprint arXiv:2111.05301*, 2021.

[134] Y. Le Cun, L. Jackel, B. Boser, J. Denker, H. Graf, I. Guyon, D. Henderson, R. Howard, and W. Hubbard, "Handwritten digit recognition: Applications of neural network chips and automatic learning," *IEEE Communications Magazine*, vol. 27, no. 11, 1989.

[135] Y. LeCun, "The mnist database of handwritten digits," *http://yann. lecun. com/exdb/mnist/*, 1998.

[136] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. of the IEEE*, vol. 86, no. 11, 1998.

[137] D. Lee, P. M. Chen, J. Flinn, and S. Narayanasamy, "Chimera: hybrid program analysis for determinism," 2012.

[138] J. Lee and T. E. Carlson, "Ultra-fast cgra scheduling to enable run time, programmable cgras," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 1207–1212.

[139] S. Lee, K. Kim, G. Koo, H. Jeon, W. W. Ro, and M. Annavaram, "Warped-compression: Enabling power efficient gpus through register compression," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3S, pp. 502–514, 2015.

[140] A. Li, T.-J. Chang, and D. Wentzlaff, "Automated design of fpgas facilitated by cycle-free routing," in *FPL 30*, 2020.

[141] Z. Li, D. Wijerathne, X. Chen, A. Pathania, and T. Mitra, "Chordmap: Automated mapping of streaming applications onto cgra," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 2, pp. 306–319, 2021.

[142] J. Lin, Y. Yang, R. K. Gupta, and Z. Tu, "Local binary pattern networks," *CoRR*, vol. abs/1803.07125, 2018. [Online]. Available: http://arxiv.org/abs/1803.07125

[143] D. Liu, T. Chen, S. Liu, J. Zhou, S. Zhou, O. Teman, X. Feng, X. Zhou, and Y. Chen, "Pudiannao: A polyvalent machine learning accelerator," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 1, 2015.

[144] F. Liu, H. Ahn, S. R. Beard, T. Oh, and D. I. August, "Dynaspam: Dynamic spatial architecture mapping using out of order instruction schedules," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 541–553. [Online]. Available: https://doi.org/10.1145/2749469.2750414

[145] T. Liu, C. M. Sadler, P. Zhang, and M. Martonosi, "Implementing software on resource-constrained mobile sensors: Experiences with impala and zebranet," in *MobiSys 2*. New York, NY, USA: ACM, 2004.

[146] E. Lockerman, A. Feldmann, M. Bakhshalipour, A. Stanescu, S. Gupta, D. Sanchez, and N. Beckmann, "Livia: Data-centric computing throughout the memory hierarchy," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 417–433.

[147] B. Lucia, V. Balaji, A. Colin, K. Maeng, and E. Ruppel, "Intermittent computing: Challenges and opportunities," in *SNAPL 2*, 2017.

[148] B. Lucia and B. Ransford, "A simpler, safer programming and execution model for intermittent systems," in *Proc. of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2015. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2737924.2737978

[149] K. Ma, X. Li, J. Li, Y. Liu, Y. Xie, J. Sampson, M. T. Kandemir, and V. Narayanan, "Incidental computing on iot nonvolatile processors," in *Proc. of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017.

[150] K. Ma, Y. Zheng, S. Li, K. Swaminathan, X. Li, Y. Liu, J. Sampson, Y. Xie, and V. Narayanan, "Architecture exploration for ambient energy harvesting nonvolatile processors," in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, 2015.

[151] K. Maeng, A. Colin, and B. Lucia, "Alpaca: Intermittent execution without checkpoints," in *Proc. of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. Vancouver, BC, Canada: ACM, Oct. 22–27, 2017.

[152] K. Maeng and B. Lucia, "Supporting peripherals in intermittent systems with just-in-time checkpoints," in *PLDI*, 2019.

[153] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "Dresc: A retargetable compiler for coarse-grained reconfigurable architectures," in *2002 IEEE International Conference on Field-Programmable Technology, 2002.(FPT). Proceedings.* IEEE, 2002, pp. 166–173.

[154] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "Adres: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix," in *International Conference on Field Programmable Logic and Applications*. Springer, 2003, pp. 61–70.

[155] J. Melchert, K. Feng, C. Donovick, R. Daly, C. Barrett, M. Horowitz, P. Hanrahan, and P. Raina, "Automated design space exploration of cgra processing element architectures using frequent subgraph analysis," *arXiv preprint arXiv:2104.14155*, 2021.

[156] S. Midkiff and D. Padua, "A comparison of four synchronization optimization techniques," in *Intl. Conf. on Parallel Processing*, vol. 2, 1991, pp. 9–16.

[157] S. P. Midkiff and D. A. Padua, "Compiler algorithms for synchronization," *IEEE Transactions on Computers*, vol. C-36, no. 12, pp. 1485–1495, 1987.

[158] J. S. Miguel, K. Ganesan, M. Badr, and N. E. Jerger, "The eh model: Analytical exploration of energy-harvesting architectures," *IEEE Computer Architecture Letters*, vol. 17, no. 1, pp. 76–79, Jan 2018.

[159] A. Mirhoseini, E. M. Songhori, and F. Koushanfar, "Idetic: A high-level synthesis approach for enabling long computations on transiently-powered ASICs," in *IEEE Pervasive Computing and Communication Conference (PerCom)*, Mar. 2013. [Online]. Available: http://aceslab.org/sites/default/files/Idetic.pdf

[160] A. Mirhoseini, A. Goldie, M. Yazgan, J. Jiang, E. Songhori, S. Wang, Y.-J. Lee, E. Johnson, O. Pathak, S. Bae *et al.*, "Chip placement with deep reinforcement learning," *arXiv preprint arXiv:2004.10746*, 2020.

[161] E. Mirsky, A. DeHon *et al.*, "Matrix: a reconfigurable computing architecture with configurable instruction distribution and deployable resources." in *FCCM*, vol. 96, 1996, pp. 17–19.

[162] M. Mishra, T. J. Callahan, T. Chelcea, G. Venkataramani, S. C. Goldstein, and M. Budiu, "Tartan: evaluating spatial computation for whole program execution," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 5, 2006.

[163] T. M. Mitchell, *Machine Learning*, 1st ed. New York, NY, USA: McGraw-Hill, Inc., 1997.

[164] T. Miyamori and K. Olukotun, "Remarc: Reconfigurable multimedia array coprocessor," *IEICE Transactions on information and systems*, vol. 82, no. 2, pp. 389–397, 1999.

[165] P. Mohan, O. Atli, O. Kibar, M. Z. Vanaikar, L. Pileggi, and K. Mai, "Top-down physical design of soft embedded fpga fabrics," in *FPGA*. ACM, 2021.

[166] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, W. Paul, M. I. Jordan, and I. Stoica, "Ray: A distributed framework for emerging ai applications," *arXiv preprint arXiv:1712.05889*, 2017.

[167] T. Moscibroda and O. Mutlu, "A case for bufferless routing in on-chip networks," in *ISCA 36*, 2009.

[168] T. M. Nabhan and A. Y. Zomaya, "Toward generating neural network structures for function approximation," *Neural Networks*, vol. 7, no. 1, 1994.

[169] S. Naderiparizi, M. Hessar, V. Talla, S. Gollakota, and J. R. Smith, "Towards battery-free {HD} video streaming," in *NSDI 15*, 2018.

[170] S. Naderiparizi, Z. Kapetanovic, and J. R. Smith, "Wispcam: An rf-powered smart camera for machine vision applications," in *Proc. of the 4th International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems*, ser. ENSsys'16. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2996884.2996888

[171] P. Nakkiran, R. Alvarez, R. Prabhavalkar, and C. Parada, "Compressing deep neural networks using a rank-constrained topology." in *Sixteenth Annual Conference of the International Speech Communication Association*, 2015.

[172] M. Nardello, H. Desai, D. Brunelli, and B. Lucia, "Camaroptera: A batteryless long-range remote visual sensing system," in *ENSSys 7*, 2019.

[173] L. Nazhandali, B. Zhai, A. Olson, A. Reeves, M. Minuth, R. Helfand, S. Pant, T. Austin, and D. Blaauw, "Energy optimization of subthreshold-voltage sensor network processors," in *ISCA 32*, 2005.

[174] Q. M. Nguyen and D. Sanchez, "Fifer: Practical acceleration of irregular applications on reconfigurable architectures," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 1064–1077.

[175] C. Nicol, "A coarse grain reconfigurable array (CGRA) for statically scheduled data flow computing," *WaveComputing WhitePaper*, 2017.

[176] R. S. Nikhil, "The parallel programming language id and its compilation for parallel machines," *International Journal of High Speed Computing*, vol. 5, no. 02, pp. 171–223, 1993.

[177] R. S. Nikhil *et al.*, "Executing a program on the mit tagged-token dataflow architecture," *IEEE Transactions on computers*, vol. 39, no. 3, 1990.

[178] R. S. Nikhil *et al.*, "Executing a program on the mit tagged-token dataflow architecture," *IEEE Transactions on computers*, vol. 39, no. 3, 1990.

[179] T. Nowatzki, V. Gangadhan, K. Sankaralingam, and G. Wright, "Pushing the limits of accelerator efficiency while retaining programmability," in *HPCA*, March 2016, pp. 27–39.

[180] T. Nowatzki, N. Ardalani, K. Sankaralingam, and J. Weng, "Hybrid optimization/heuristic instruction scheduling for programmable accelerator codesign," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '18. New York, NY, USA: ACM, 2018, pp. 36:1–36:15. [Online]. Available: http://doi.acm.org/10.1145/3243176.3243212

[181] T. Nowatzki, N. Ardalani, K. Sankaralingam, and J. Weng, "Hybrid optimization/heuristic instruction scheduling for programmable accelerator codesign," in *PACT 27*, 2018.

[182] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, "Stream-dataflow acceleration," in *ISCA 44*, 2017.

[183] T. Nowatzki, V. Gangadhar, and K. Sankaralingam, "Exploring the potential of heterogeneous von neumann/dataflow execution models," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 298–310.

[184] T. Nowatzki, V. Gangadhar, K. Sankaralingam, and G. Wright, "Domain specialization is generally unnecessary for accelerators," *IEEE Micro*, vol. 37, no. 3, 2017.

[185] T. Nowatzki, M. Sartin-Tarm, L. De Carli, K. Sankaralingam, C. Estan, and B. Robatmili, "A general constraint-centric scheduling framework for spatial architectures," *ACM SIGPLAN Notices*, vol. 48, no. 6, 2013.

[186] Nvidia, "Nividia jetson tx2," 2019. [Online]. Available: https://developer.nvidia.com/embedded/develop/hardware

[187] N. Ozaki, Y. Yasuda, M. Izawa, Y. Saito, D. Ikebuchi, H. Amano, H. Nakamura, K. Usami, M. Namiki, and M. Kondo, "Cool mega-arrays: Ultralow-power reconfigurable accelerator chips," *IEEE Micro*, vol. 31, no. 6, 2011.

[188] G. M. Papadopoulos and D. E. Culler, "Monsoon: An explicit token-store architecture," *SIGARCH Comput. Archit. News*, vol. 18, no. 2SI, p. 82–91, may 1990. [Online]. Available: https://doi.org/10.1145/325096.325117

[189] A. Parashar, M. Pellauer, M. Adler, B. Ahsan, N. Crago, D. Lustig, V. Pavlov, A. Zhai, M. Gambhir, A. Jaleel *et al.*, "Triggered instructions: a control paradigm for spatially-programmed architectures," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3, 2013.

[190] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," in *Proc. of the 44th annual Intl. Symp. on Computer Architecture (Proc. ISCA-44)*, 2017.

[191] H. Park, K. Fan, S. A. Mahlke, T. Oh, H. Kim, and H.-s. Kim, "Edge-centric modulo scheduling for coarse-grained reconfigurable architectures," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008, pp. 166–176.

[192] H. Park, Y. Park, and S. Mahlke, "Polymorphic pipeline array: A flexible multicore accelerator with virtualized execution for mobile multimedia applications," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. New York, NY, USA: Association for Computing Machinery, 2009, p. 370–380. [Online]. Available: https://doi.org/10.1145/1669112.1669160

[193] A. Parks, A. Sample, Y. Zhao, and J. R. Smith, "A wireless sensing platform utilizing ambient rf energy," in *Proc. of the IEEE Topical Meeting on Wireless Sensors and Sensor Networks (WiSNET)*. IEEE, 2013.

[194] D. Patterson, T. Anderson, and K. Yelick, "A Case for Intelligent DRAM: IRAM," in *Hot Chips VIII Symposium Record*, August 1996.

[195] P. M. Phothilimthana, T. Jelvis, R. Shah, N. Totla, S. Chasins, and R. Bodik, "Chlorophyll: Synthesis-aided compiler for low-power spatial architectures," *SIGPLAN Not.*, vol. 49, no. 6, p. 396–407, jun 2014. [Online]. Available: https://doi.org/10.1145/2666356.2594339

[196] M. Poremba, S. Mittal, D. Li, J. S. Vetter, and Y. Xie, "Destiny: A tool for modeling emerging 3d nvm and edram caches," in *DATE*, 2015.

[197] Powercast Co., "Development Kits - Wireless Power Solutions," http://www.powercastco.com/products/development-kits/, visited July 30, 2014.

[198] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, "Plasticine: A reconfigurable architecture for parallel patterns," in *ISCA 44*, 2017.

[199] P. Prabhat, B. Labbe, G. Knight, A. Savanth, J. Svedas, M. J. Walker, S. Jeloka, P. M.-Y. Fan, F. Garcia-Redondo, T. Achuthan *et al.*, "27.2 m0n0: A performance-regulated 0.8-to-38mhz dvfs arm cortex-m33 simd mcu with 10nw sleep power," in *ISSCC*, 2020.

[200] B. Ransford, J. Sorber, and K. Fu, "Mementos: System support for long-running computation on RFID-scale devices," in *ASPLOS*, Mar. 2011.

[201] A. Ren, Z. Li, C. Ding, Q. Qiu, Y. Wang, J. Li, X. Qian, and B. Yuan, "Sc-dcnn: highly-scalable deep convolutional neural network using stochastic computing," in *Proc. of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[202] Riscv, "riscv-v-spec," Apr 2019. [Online]. Available: https://github.com/riscv/riscv-v-spec

[203] A. Rowe, M. E. Berges, G. Bhatia, E. Goldman, R. Rajkumar, J. H. Garrett, J. M. Moura, and L. Soibelman, "Sensor andrew: Large-scale campus-wide sensing and actuation," *IBM Journal of Research and Development*, vol. 55, no. 1.2, 2011.

[204] A. Rucker, M. Vilim, T. Zhao, Y. Zhang, R. Prabhakar, and K. Olukotun, "Capstan: A vector rda for sparsity," 2021.

[205] T. N. Sainath and C. Parada, "Convolutional neural networks for small-footprint keyword spotting," in *16th Annual Conference of the International Speech Communication Association*, 2015.

[206] A. P. Sample, D. J. Yeager, P. S. Powledge, A. V. Mamishev, and J. R. Smith, "Design of an RFID-based battery-free programmable sensing platform," *IEEE Transactions on Instrumentation and Measurement*, vol. 57, no. 11, pp. 2608–2615, Nov. 2008.

[207] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 4510–4520.

[208] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore, "Exploiting ilp, tlp, and dlp with the polymorphous trips architecture," in *ISCA 30*, 2003.

[209] K. Sankaralingam, T. Nowatzki, G. Wright, P. Palamuttam, J. Khare, V. Gangadhar, and P. Shah, "Mozart: Designing for software maturity and the next paradigm for chip architectures," in *IEEE Hot Chips 33 Symposium, HCS 2021, Palo Alto, CA, USA, August 22-24, 2021*. IEEE, 2021, pp. 1–20. [Online]. Available: https://doi.org/10.1109/HCS52781.2021.9567306

[210] P. G. Sassone and D. S. Wills, "Dynamic strands: Collapsing speculative dependence chains for reducing pipeline communication," in *37th International Symposium on Microarchitecture (MICRO-37'04)*, 2004.

[211] M. Satyanarayanan, N. Beckmann, G. A. Lewis, and B. Lucia, "The role of edge offload for hardware-accelerated mobile devices," in *HotMobile*, 2021.

[212] A. Sembrant, T. Carlson, E. Hagersten, D. Black-Shaffer, A. Perais, A. Seznec, and P. Michaud, "Long term parking (ltp): criticality-aware resource allocation in ooo processors," in *48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015.

[213] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks, "Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures," in *ISCA 41*, 2014.

[214] P. Shivakumar and N. P. Jouppi, "CACTI 3.0: An Integrated Cache, Timing, Power, and Area Model," Compaq Western Research Laboratory, Tech. Rep., February 2001.

[215] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[216] H. Singh, M.-H. Lee, G. Lu, F. Kurdahi, N. Bagherzadeh, and E. Chaves Filho, "Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications," *IEEE Transactions on Computers*, vol. 49, no. 5, pp. 465–481, 2000.

[217] W. Snyder, "Verilator and systemperl," in *North American SystemC Users' Group, DAC*, 2004.

[218] M. Song, K. Zhong, J. Zhang, Y. Hu, D. Liu, W. Zhang, J. Wang, and T. Li, "In-situ ai: Towards autonomous and incremental deep learning for iot systems," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2018, pp. 92–103.

[219] P. Sparks, "A route to a trillion devices," *Arm WhitePaper*, 2017.

[220] M. Surbatovich, L. Jia, and B. Lucia, "Automatically enforcing fresh and consistent inputs in intermittent systems," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 851–866.

[221] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin, "Wavescalar," in *MICRO 36*, 2003.

[222] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning." in *AAAI*, vol. 4, 2017.

[223] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proc. of the IEEE conference on computer vision and pattern recognition*, 2015.

[224] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[225] C. Tan, M. Karunaratne, T. Mitra, and L.-S. Peh, "Stitch: Fusible heterogeneous accelerators enmeshed with many-core architecture for wearables," in *ISCA 45*, 2018.

[226] C. Tan, C. Xie, A. Li, K. J. Barker, and A. Tumeo, "Opencgra: An open-source unified framework for modeling, testing, and evaluating cgras," in *2020 IEEE 38th International Conference on Computer Design (ICCD)*. IEEE, 2020, pp. 381–388.

[227] M. Tan and Q. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *International conference on machine learning*. PMLR, 2019, pp. 6105–6114.

[228] M. B. Taylor, "Is dark silicon useful? harnessing the four horsemen of the coming dark silicon apocalypse," in *DAC*, 2012.

[229] Texas Instruments, "CC2650MODA SimpleLink™ Bluetooth® low energy Wireless MCU Module," http://www.ti.com/product/cc2650moda/datasheet, 2017.

[230] TI Inc., "Overview for MSP430FRxx FRAM," http://ti.com/wolverine, 2014, visited July 28, 2014.

[231] C. Torng and P. Pan, "Ue-cgra hpca 2021 artifact," Mar 2021. [Online]. Available: https://github.com/cornell-brg/torng-uecgra-scripts-hpca2021

[232] C. Torng, P. Pan, Y. Ou, C. Tan, and C. Batten, "Ultra-elastic cgras for irregular loop specialization," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 412–425.

[233] A. Traber, "Pulpino: A small single-core risc-v soc."

[234] L. R. Tucker, "Some mathematical notes on three-mode factor analysis," *Psychometrika*, vol. 31, no. 3, 1966.

[235] J. Van Der Woude and M. Hicks, "Intermittent computation without hardware support or programmer intervention," in *Proc. of OSDI'16: 12th USENIX Symposium on Operating Systems Design and Implementation*, 2016.

[236] N. Vedula, A. Shriraman, S. Kumar, and W. N. Sumner, "Nachos: Software-driven hardware-assisted memory disambiguation for accelerators," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 710–723.

[237] 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," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 1, 2010.

[238] N. Vijaykumar, K. Hsieh, G. Pekhimenko, S. Khan, A. Shrestha, S. Ghose, A. Jog, P. B. Gibbons, and O. Mutlu, "Zorua: A holistic approach to resource virtualization in gpus," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, 2016.

[239] M. Vilim, A. Rucker, Y. Zhang, S. Liu, and K. Olukotun, "Gorgon: Accelerating machine learning from relational data," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 309–321.

[240] D. Voitsechov and Y. Etsion, "Single-graph multiple flows: Energy efficient design alternative for gpgpus," *ACM SIGARCH computer architecture news*, vol. 42, no. 3, 2014.

[241] D. Voitsechov, O. Port, and Y. Etsion, "Inter-thread communication in multithreaded, reconfigurable coarse-grain arrays," in *MICRO 51*, 2018.

[242] E. Waingold *et al.*, "Baring It All to Software: Raw Machines," in *IEEE Computer*, September 1997.

[243] B. Wang, M. Karunarathne, A. Kulkarni, T. Mitra, and L.-S. Peh, "Hycube: A 0.9 v 26.4 mops/mw, 290 pj/op, power efficient accelerator for iot applications," in *2019 IEEE Asian Solid-State Circuits Conference (A-SSCC)*. IEEE, 2019, pp. 133–136.

[244] B. A. Warneke and K. S. Pister, "17.4 an ultra-low energy microcontroller for smart dust wireless sensor networks," 2004.

[245] M. A. Watkins, T. Nowatzki, and A. Carno, "Software transparent dynamic binary translation for coarse-grain reconfigurable architectures," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 138–150.

[246] J. Wawrzynek, K. Asanovic, B. Kingsbury, D. Johnson, J. Beck, and N. Morgan, "Spert-ii: A vector microprocessor system," *Computer*, vol. 29, no. 3, 1996.

[247] J. Weng, S. Liu, V. Dadu, Z. Wang, P. Shah, and T. Nowatzki, "Dsagen: synthesizing programmable spatial accelerators," in *ISCA 47*, 2020.

[248] J. Weng, S. Liu, Z. Wang, V. Dadu, and T. Nowatzki, "A hybrid systolic-dataflow architecture for inductive matrix algorithms," in *HPCA*, 2020.

[249] A. Wickramasinghe, D. Ranasinghe, and A. Sample, "Windware: Supporting ubiquitous computing with passive sensor enabled rfid," in *RFID*, April 2014.

[250] D. Wijerathne, Z. Li, A. Pathania, T. Mitra, and L. Thiele, "Himap: Fast and scalable high-quality mapping on cgra via hierarchical abstraction," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.

[251] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross, "Q100: The architecture and design of a database processing unit," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: ACM, 2014, pp. 255–268. [Online]. Available: http://doi.acm.org/10.1145/2541940.2541961

[252] C.-X. Xue, W.-H. Chen, J.-S. Liu, J.-F. Li, W.-Y. Lin, W.-E. Lin, J.-H. Wang, W.-C. Wei, T.-W. Chang, T.-C. Chang *et al.*, "24.1 a 1mb multibit reram computing-in-memory macro with 14.6 ns parallel mac computing time for cnn based ai edge processors," in *2019 IEEE International Solid-State Circuits Conference-(ISSCC)*. IEEE, 2019, pp. 388–390.

[253] C.-X. Xue, T.-Y. Huang, J.-S. Liu, T.-W. Chang, H.-Y. Kao, J.-H. Wang, T.-W. Liu, S.-Y. Wei, S.-P. Huang, W.-C. Wei *et al.*, "15.4 a 22nm 2mb reram compute-in-memory macro with 121-28tops/w for multibit mac computing for tiny ai edge devices," in *2020 IEEE International Solid-State Circuits Conference-(ISSCC)*. IEEE, 2020, pp. 244–246.

[254] Y. Yang, J. S. Emer, and D. Sanchez, "Spzip: architectural support for effective data compression in irregular applications," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 1069–1082.

[255] Zac Manchester, "KickSat," http://zacinaction.github.io/kicksat/, 2015.

[256] H. Zhang, J. Gummeson, B. Ransford, and K. Fu, "Moo: A batteryless computational rfid and sensing platform," *Department of Computer Science, University of Massachusetts Amherst., Tech. Rep*, 2011.

[257] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-x: An accelerator for sparse neural networks," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, 2016.

[258] Z. Zhao, W. Sheng, Q. Wang, W. Yin, P. Ye, J. Li, and Z. Mao, "Towards higher performance and robust compilation for cgra modulo scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 9, pp. 2201–2219, 2020.