Title
Programming Abstractions and Synthesis-Aided Compilation for Emerging Computing Platforms

Author
Phothilimthana, Phitchaya

Publication Date
2018

Peer reviewed|Thesis/dissertation

Programming Abstractions and Synthesis-Aided Compilation for Emerging Computing Platforms

by

Phitchaya Phothilimthana

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Rastislav Bodik, Co-chair
Professor Katherine Yelick, Co-chair
Professor David Culler
Assistant Professor Zachary Pardos

Fall 2018

Programming Abstractions and Synthesis-Aided Compilation for Emerging Computing Platforms

Abstract

Programming Abstractions and Synthesis-Aided Compilation for Emerging Computing Platforms

by

Phitchaya Phothilimthana

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Rastislav Bodik, Co-chair

Professor Katherine Yelick, Co-chair

Today's cutting-edge applications, ranging from wearable devices and embedded medical sensors to high-performance data centers, put new demands on computer architectures. Those demands include more computation capability, a tight power budget, low latency, high throughput, and many more. To meet these requirements, specialized architectures with low energy consumption are becoming more prevalent. Many of these architectures trade off programmability features for gains in energy efficiency and performance. Hence, programmability challenges are inevitable as applications continue to evolve and make new demands on computing architectures.

I propose key principles for improving programmability intended for application writers as well as compiler developers and language designers. First, I address programmability issues by providing a programming model that hides low-level details but sufficiently exposes essential details for application writers to control. Second, to compile and optimize programs, I apply a new compilation methodology based on synthesis. Unlike a classical compiler's transformation, synthesis obtains a correct and optimal solution by searching for an optimal candidate that is semantically equivalent to a specification program. This search helps compilers generate efficient code without deriving a program via a sequence of transformations, which are challenging for compiler developers to design for new unconventional architectures.

In this thesis, I demonstrate the key principles in three projects: Chlorophyll, a language and compiler for low-power spatial architectures; Floem, a programming system for NIC-accelerated data center applications; and GreenThumb, a framework for building a superoptimizer (an assembly program optimizer based on synthesis).

To my family.

Contents

List of Figures

List of Tables

Acknowledgments

I would like to give acknowledgments in chronological order through the journey of my life and this thesis.

I was born to the Phothilimthana family. I am eternally grateful for unconditional love from my parents and sister. I thank them for teaching me to always have faith in myself and believe that I can do whatever I wish and work toward. I thank them for raising me to be me today.

In 2004, I was introduced to programming in the required programming course during my first year of high school at Mahidol Wittayanusorn School. Mahidol Wittayanusorn School is the best high school in Thailand, in my opinion of course. I thank all my teachers and the school for taking me to the beginning of my exciting career path in computer science.

In 2006, I represented Thailand at International Olympiad in Informatics (IOI) and thus received a scholarship from the Thai government to study abroad. Thanks to all the organizers of the programming camps, especially to Jittat Fakcharoenphol, for making me discover the joy of solving programming problems through my path to IOI. I am grateful to the Thai government for giving me an amazing opportunity to study at the top university in the world.

In 2010, friends convinced me to take the 6.035 compiler class at MIT taught by Saman Amarasinghe. I later joined his research group. I never thought that I would ever take a compiler class, but I did only because of my friends, Yod Watanaprakornkul and Melissa Gymrek. I thank them for convincing me to take this class. Then, Saman asked me to join his research group after I took his class. I accepted this great offer, and I have been doing compiler research ever since. I would like to thank Saman for introducing me to compilers and research and for giving career advice. I also thank Jason Ansel, my graduate student mentor, who advised me through my first published project at MIT, and for all his support as my friend and husband.

In 2012, I met my Ph.D. advisor, Ras Bodik. I first met Ras during the visit days, and I instantly felt his enthusiasm and love in research. He introduced me to GreenArrays, my then future-to-be research project, even before I decided to join Berkeley. After working with Ras for six years, I can say that I could not have asked for a better advisor. Ras pushed me for the better, but at the same time cared for my well-being. Ras liked to ask for many revisions for that perfect presentation and paper's introduction and overview. He sometimes "twisted" my arm to make cool demos, like running applications using a lemon or potato battery! Although they were a lot more work than I signed up for, I am grateful for such

advice, which certainly made my work ten-times cooler and more exciting. I am genuinely thankful for all the guidance and lengthy discussions during our weekly meetings that always ran late.

Shortly, I joined the Berkeley Computer Science Ph.D. program. Apart from having a great advisor, I also had an awesome, supportive research group. Thanks to Leo Meyerovich, Joel Galenson, Thibaud Hottelier, Shaon Barman, Ali Koksal, Sarah Chasins, Julie Newcomb, Sam Elliott, Sam Kaufman, Chenglong Wang, and Rohin Shah. They showed me how to have fun and be happy in grad school. I enjoyed all our conversations over daily lunch from the co-op living situation to a serious grad school business. Special thanks to Sarah Chasins, my academic twin, who is the most caring and supportive grad student friend I have. As a token of appreciation, I dedicate the illustration in Figure 0.1 to the group. Also, many thanks to my sweet grad student fellows, Penporn Koanantakool and Neeraja Yadwadkar, for listening, sharing, and fun hangouts. Thanks to everyone in Chaperone, the Berkeley PL research lab.

In 2015, here came the surprise. Ras moved to the University of Washington, so I asked Kathy Yelick to be my co-advisor. I am grateful to Kathy for making sure I was doing okay when I worked remotely with Ras. I am also thankful for her feedback on writing and presentation for my qualifying examination and this thesis.

In 2016, I moved to the University of Washington as a visiting student. Shoutouts to PLSE and SAMPA labs for the fun and lively research environment at UW.

In 2018, my ultimate frisbee team got the third place from Masters Women's Nationals Championship. A big part of my grad student's life apart from research is playing ultimate frisbee. I am grateful to the ultimate frisbee community for the welcoming, inclusive environment and the fun breaks from research.

Finally, I filed this thesis. To all my collaborators, Emina Torlak, Tikhon Jelvis, Rohin Shah, Nishant Totla, Sarah Chasins, Michael Schuldt, Aditya Thakur, Dinakar Dhurjati, Ming Liu, Antoine Kaufmann, Simon Peter, Tom Anderson, Greg Bailey, Charley Shattuck, Per Ljung, Paulo Matos, and Vinod Grover, it has been a pleasure collaborating with you all. Thank you for making these projects in my thesis possible. Thanks to the rest of my dissertation committee members, David Culler and Zachary Pardos, for their insightful feedback and suggestions. Last but not least, I appreciate the help from the Berkeley ParLab and UW PLSE staff: Roxana Infante, Tamille Chouteau, Kostadin Ilov, Lydia Raya, and Amanda Robles.

Figure 0.1: Bodik's research group in 2012–2018

Chapter 1

Introduction

Leading-edge applications, ranging from smart devices to high-performance data centers, put new demands on computing hardware. These demands include more computation power, low energy consumption, high performance, etc. To meet these requirements after the end of Moore's law, specialized architectures with low power are becoming more popular. These architectures often sacrifice programmability features for gains in performance and energy efficiency. Hence, programmability challenges are unavoidable as applications evolve and make new demands on computing hardware.

1.1 Programmability Challenges

In this thesis, programmability challenges posed by recent emerging computer architectures are categorized into three kinds, as follows.

Challenges due to specialization. Specialized architectures force programmers to reason about their unconventional characteristics or control decisions that are not typically made by programmers, resulting in new programming challenges.

Specialized processors for low-power, energy-efficient applications often have minimalistic resources, such as narrow bitwidth, small memory, and simple interconnects. For example, TI MSP430 ultra-low-power MCUs, widely used microcontrollers, have 16-bit words and contain only up to 256 kilobytes of memory [75]. Atmel AVR, used in most Arduino boards, is even more extreme, using only 8 bits per word [15, 109]. GreenArrays GA144, a highly energy-efficient chip, has 300 bytes of memory per core and 18-bit words [65]. Narrow bitwidth introduces more overflow and precision problems, forcing programmers to carefully manipulate numbers in certain ways or use multiple words to represent a number that requires high precision. At the same time, programmers have to make their programs and data fit in tiny memory. Besides narrow bitwidth and small memory, some processors employ spatial designs that reduce energy consumption by requiring software-managed communication between computing elements with limited interconnects. GA144 has a simple 2D mesh inter-

connect of 144 asynchronous cores with no shared memory. This design allows a few small cores to be active at a time, thus, consuming very little power overall. Furthermore, the nonexistence of synchronized clock in GA144 is crucial to achieve ultra-low energy consumption [30]. However, the spatial design forces programmer to partition programs and data onto different locations (e.g., cores). This process is extremely challenging when each core is minuscule.

Specialized hardware developed for accelerating applications in a specific domain also exposes its own unique features. GPUs, originally designed for graphic computations, are now used for accelerating applications from various domains — such as scientific computing, data processing, and machine learning — thanks to its massive parallelism [110]. However, GPU programmers must coordinate a massive number of threads to work collaboratively in a SIMT style and manage scratchpad memory manually. Barefoot Tofino, a programmable Ethernet switch, exposes its match-action computing stages to users and forces them to program in a match-action programming paradigm [20]. The spatial design that is employed in processor design for energy efficiency can also be used for performance. Spatial architectures such as FPGAs and WaveFlow [45, 39] exploit data locality for low latency, and massive pipeline parallelism in data-flow computation for high throughput.

Challenges due to heterogeneity. Heterogeneous computing refers to systems that use more than one kind of processor [53, 110]. Heterogeneity in a system poses multiple new challenges to programmers. The first challenge is dividing the workload among different computing devices. This process is extremely difficult because programmers must understand the tradeoffs between running different kinds of computation on different devices as well as inter-device communication, which are expensive across devices. Second, after the programmers decide how to partition an application across multiple devices, they have to write multiple programs to run on those devices. Different hardware architectures often accept code written in different languages with different programming paradigms and require different optimization tricks. If the programmers change how they partition the application, they will need to modify programs running on all devices. Therefore, heterogeneous computing demands a huge effort from programmers in order to explore a variation of workload assignment.

Challenges due to performance understanding. New programming challenges also arise from complicated performance characteristics of interacting components in the hardware architectures. For example, GPU programmers must understand thread divergence, memory coalescing, shared memory bank conflicts, and register spill in order to achieve good performance on GPUs. Spatial architectures such as GA144, WaveFlow, and FPGAs expose highly-variable latency for instruction operands, depending on the spatial distance between the operands and the instructions. For heterogeneous computing, it is especially hard to estimate the performance of an implementation accurately. A static performance model is often imprecise because it abstracts away some complexity of the hardware model.

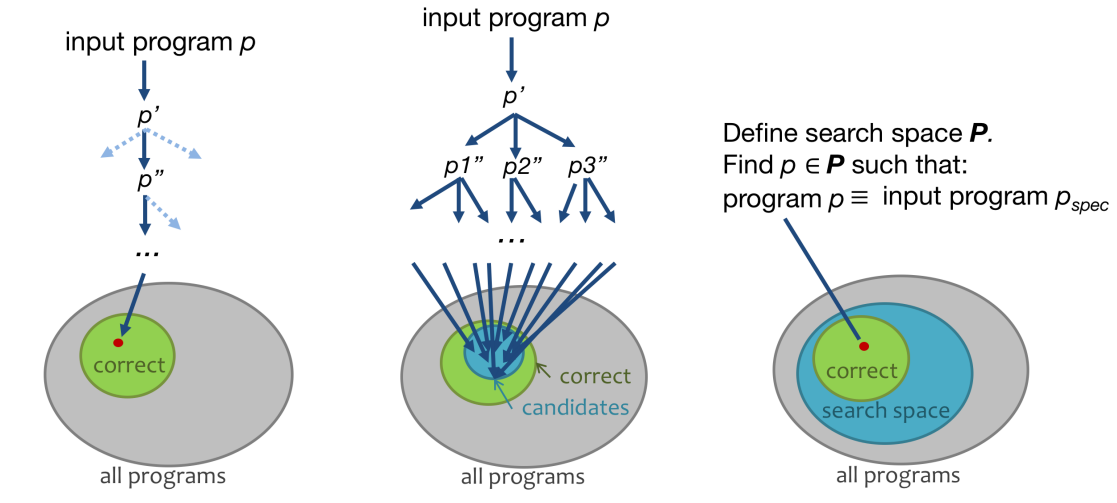
We can avoid a static approach by evaluating the performance of a task empirically, but we still cannot simply assign a placement of a task in isolation because it interacts with other parts of the application. Unfortunately, we often cannot afford to test all possible choices considering the entire application because the space is too big. Therefore, there has been a large body of work on automatically mapping a program to different hardware nodes [110], but there is still no solution for all.

This thesis primarily addresses the challenges due to specialization and heterogeneity. Some of the proposed methods, however, indirectly solve the last challenge by making it easier to explore many implementations of programs.

1.2 Compilation Approaches

Programmability challenges can be solved by a programming model that has the right level of abstraction for an application domain, together with a compiler that can lower a higher-level program written by a programmer to executable code. To achieve good performance, the programmer relies on the compiler to generate efficient code. In this section, I compare and contrast different compilation techniques, and discuss their potential to solve the programmability challenges posed by emerging specialized architectures and their impacts on programming abstractions. Figure 1.1 summarizes how each approach produces an output code.

Classical compiler transformations. Researchers have studied compiler technology for over sixty years [9]. A set of program transformations allow programmers to write pro-



(a) Classical transformation (b) Autotuning (c) Synthesis

Figure 1.1: Comparison of different compilation techniques

grams in higher-level languages like Fortran instead of assembly. Many transformations have forever changed the way we program. Register allocation allows programmers to reason about virtual variables instead of physical registers. Instruction scheduling hides some low-level performance understanding from programmers. Compilers have become easier to build thanks to parser generators, analysis frameworks, and compiler infrastructures such as LLVM [89]. Nevertheless, building compilers in general, and especially those for new, unconventional processors, remains daunting. The classical approach to building compilers requires the laborious development of program analyses and transformations along with heuristics that select the best set of transformations and order them. As a result, building a mature new compiler may take a decade or more and require extremely high levels of professional expertise [131]. Furthermore, existing well-developed analyses and transformations may no longer be applicable to unconventional architectures, and hard-coded heuristic decisions that are tuned on a specific architecture often lead to suboptimal performance on other architectures.

Autotuning. One promising technology that addresses some programmability challenges due to performance understanding and heterogeneity is autotuning [12, 13, 55, 56, 119, 126, 129, 164]. Unlike classical compilers, autotuning compilers do not use fixed heuristics regarding how to apply different transformations. Instead, they try applying various semantics-preserved transformations with different parameters to obtain many correct output programs, run these programs, and select the one that performs best. Autotuning has been used to determine best configurations for long-running or frequently-executed applications to justify the time spent on tuning. While developers do not need to implement heuristics to select and order program transformations, they must still develop the transformations, a major challenge due to specialization in unconventional hardware.

Synthesis. Synthesis is an automatic code generation technique that has been rarely utilized in compilers. Synthesis obtains a correct and optimal solution by searching through a candidate space for an optimal candidate that is semantically equivalent to a specification program or satisfies some constraints [103, 147]. A candidate space typically includes both correct and incorrect candidates, for example, all instruction sequences of length up to 10. In contrast, a candidate space for autotuning typically includes only correct candidates.[1] Synthesis typically employs a constraint solver, such as an SMT solver, to prove the correctness (e.g., being equivalent to a specification program) of an output solution and discard incorrect candidates. By searching the space by increasing length, one is guaranteed to eventually find a correct program (and probably one that is optimal).

The search among both correct and incorrect programs allows us to produce efficient code without deriving an input program through a sequence of semantics-preserved trans-

[1]It is possible that autotuning search space may contain illegal candidates such as candidates that use more memory than available, but a percentage of illegal candidates is much smaller than that in synthesis search space.

formations, which are challenging for compiler developers to design and implement. More importantly, fast and correct implementations will be missed by a transformation-based compiler if the compiler developers do not include transformations required to produce the desired output. Thus, synthesis is more promising than the other approaches for solving programmability challenges due to specialization of unconventional architectures.

However, a major limitation of this approach is its speed and scalability to solve larger programs, similar to autotuning. This limitation is more pronounced in the synthesis setting because a search space of synthesis typically contains many incorrect candidates, much more than correct ones. If a search space is too big, we may not even discover any correct candidate. Therefore, synthesis has not been used as part of the main workflow of a compiler.

1.3 Thesis and Goals

This thesis primarily addresses the programmability challenges due to specialization and heterogeneity of unconventional emerging computing platforms. The main goal of this thesis is to enable programmability on new hardware architectures for gains in energy efficiency and/or performance of applications. My approach toward this goal advocates the following key concepts for designing and developing programming systems for emerging computing platforms.

- Concept I: design a programming model with a division of responsbilities between programmers and different components in a compiler.
- Concept II: apply synthesis in a compiler to sidestep hard-to-develop program transformations.

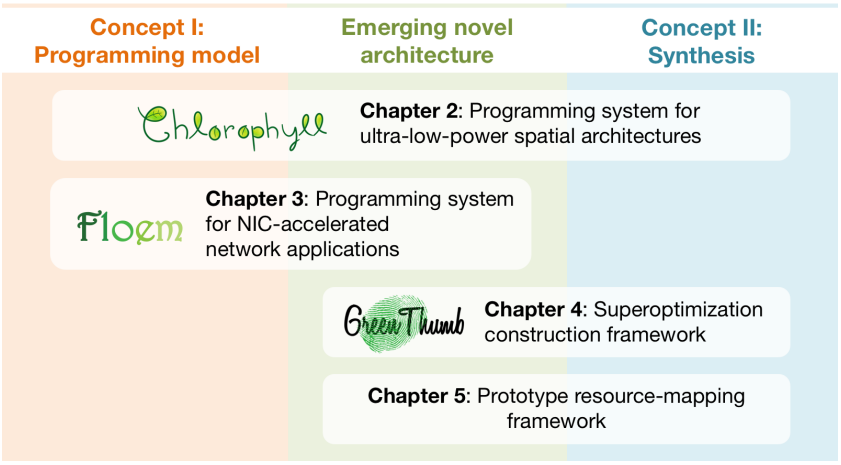


Figure 1.2: Instantiation of the key concepts in the thesis, centering around exploiting an emerging computer architecture for performance and energy efficiency.

Figure 1.2 summarizes the instantiation of these concepts in the chapters throughout this thesis, centering around exploiting an emerging novel computer architecture for performance and energy efficiency.

1.4 Concept I: Programming Model

We must design a programming model in a way that the responsibilities are divided judiciously among compiler's components (e.g., classical transformations, autotuners, and synthesizers), as well as programmers to achieve the most efficiency and/or performance. The philosophy behind this concept is that different components of the system, including humans, are good at different tasks. For example, human experts are often still better than machines on designing algorithms and data structures. Programmers may also relax semantics of a program to explore a new design without compromising the correctness of the program, while a compiler is unable to do this.

Hence, a programming model should strike a balance between hiding low-level details that can be solved efficiently by a compiler from programmers and exposing important details that programmers can control when the details are more suitable for humans to solve. This balance will ultimately let programmers productively explore various implementation choices to choose the best one. I demonstrate in this thesis how to apply this concept when designing the programming models for (1) Chlorophyll, a programming system for low-power embedded applications and (2) Floem, a programming system for NIC-accelerated network applications.

Chlorophyll (Chapter 2). I introduce a programming system for ultra-low-power spatial architectures. The system can generate code for GreenArrays GA144 [65], an energy-efficient, 144-core tile processor with distributed memory and simple interconnect. GA144 has very tiny per-core memory and uses stacks instead of registers.

The Chlorophyll programming model lets programmers provide their insights into how to partition data structures and code for the parts they choose since humans are more skilled than compilers in designing high-level program layouts. Chlorophyll mitigates programability challenges due to idiosyncratic characteristics of GA144 by letting the compiler handle the rest of programming partitioning, layout, data routing, and code generation for stack-based instructions. We use Chlorophyll to implement an accelerometer-based, hand-gesture recognition application for GA144. Compared to MSP430, GA144 is 19 times more energy efficient and 23 times faster when running this application. As a result, this application can run on a GA144 powered by a single potato!

Floem (Chapter 3). I introduce a programming system for NIC-accelerated network applications, balancing the ease of developing a correct application and the ability to refactor it to explore different design choices in a combined CPU-NIC environment.

Floem mitigates programmability challenges due to system heterogeneity and specialization of emerging programmable NICs through convenient program constructs. Floem enables offload design exploration by providing programming abstractions to assign computation to hardware resources; control mapping of logical queues to physical queues; access fields of a packet and its metadata without manually marshaling a packet; use a NIC to memoize expensive computation; and interface with an external application. The compiler infers which data must be transferred between the CPU and NIC and generates a complete cache implementation, while the runtime transparently optimizes DMA throughput. I use Floem to explore NIC-offloading designs of real-world applications, including a key-value store and a distributed real-time data analytics system; improve their throughput by 1.3–3.6× and by 75–96%, respectively, over a CPU-only implementation.

1.5 Concept II: Synthesis-Aided Compilation

Synthesis searches through a candidate space for an optimal candidate that is semantically equivalent to a specification program or satisfies some constraints. I envision synthesis as a means to enable a compiler to solve more sophisticated problems and discover better solutions, bridging increasingly larger gaps between high-level programming abstractions and actual hardware instructions. Additionally, synthesis is also a low-effort approach to compiler construction, lessening a compiler's development time while still generating code that is comparable in quality to code written by an expert or generated by a classical optimizing compiler. This is because synthesis sidesteps the need to develop program transformations, which are difficult to design and realize [101, 131], mitigating programmability challenges due to specialization from compiler developers. Furthermore, synthesis-aided compilation is more robust than transformation-based compilation because synthesis does not rely on pattern-matching using pre-defined rules.

In **Chlorophyll (Chapter 2)**, I phrase the partitioning, layout, and routing as synthesis problems because heuristic algorithms often fail to discover optimal solutions [14, 59, 114, 168]. Code generation uses a classical transformation that translates intermediate representations (IRs) to assembly. This transformation is straightforward to develop because it contains no optimization. After the IR-to-assembly transformation, I apply superoptimization (instruction-level synthesis) to perform local, machine-specific optimizations, without developing any semantics-preserved transformation. The compilation steps are summarized in Figure 1.3. Programs generated by Chlorophyll are faster than programs produced by a heuristic, classical version of our compiler and comparable to highly optimized, expert-written programs.

Despite its potential, synthesis has not been used in main-stream compilers because of its scability issues and lack of a development tool for synthesis-aided compilers. In the domain of classical compilers, researchers have developed design patterns and frameworks that accelerate the development of classical compilers [54, 89, 163]. These frameworks provide well-developed analyses, transformations, and heuristics that are adaptable to new, but

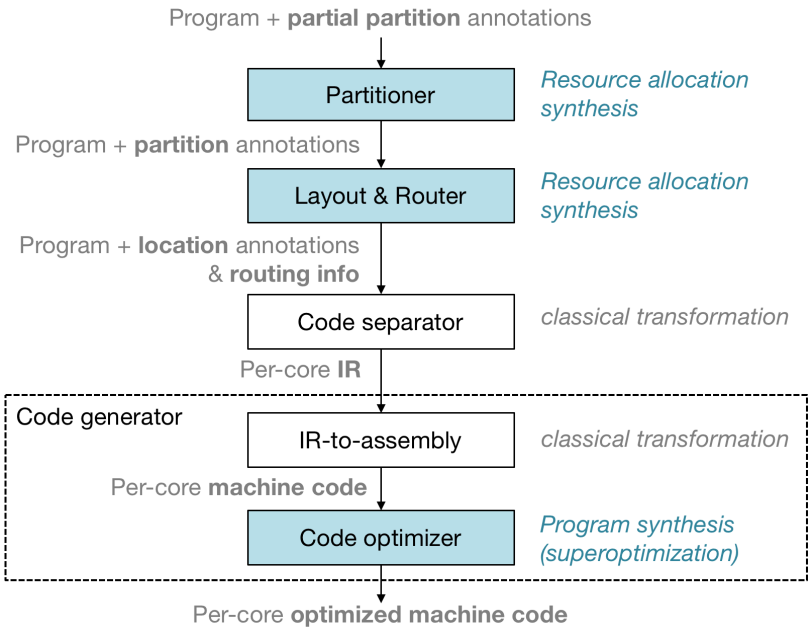


Figure 1.3: Decomposition of the CHLOROPHYLL compiler. Synthesis components are highlighted in blue.

still conventional, architectures. In this thesis, I introduce the analogue of these reusable components for synthesis-aided compilers and offer similar frameworks to developers. From my experience with CHLOROPHYLL and FLOEM, *program synthesis* and *resource allocation synthesis* are important building blocks for synthesis-aided compilers. As part of these frameworks, I also tackle the scability challenge and present techniques that make synthesis more scalable.

Program Synthesis

I believe that program synthesis — which searches for an optimal program equivalent to a specification program — can be used to sidestep the development of difficult transformations. This technique has been used to optimize both high-level programs [21, 42, 147] and low-level machine code [19, 103, 136], but it is often used as a standalone tool separate from a compiler's workflow. In CHLOROPHYLL, I demonstrate that it is feasible to use a superoptimizer (the last component in Figure 1.3) as a part of the compiler.

As an effort toward building a development tool for synthesis-aided compilers, I develop **GreenThumb (Chapter 4)**, a framework for building a superoptimizer for any Instruction Set Architecture (ISA). To address the scalability problem, I implement LENS, an enumerative search algorithm that can optimize small- to medium-size program fragments faster than existing techniques. To optimize larger program fragments, I introduce a context-aware window decomposition, optimizing a fragment of the entire code with the precise precondition

and postcondition from the surrounding context. Lastly, I compensate for the limitation of an enumerative search by combining symbolic and stochastic search into the system. To make superoptimization even more practical, we can cache superoptimized code to avoid an expensive search when optimizing programs we have seen before.

I use GreenThumb to build superoptimizers for GA144, ARM, and a subset of LLVM. The ARM superoptimizer synthesizes code fragments that are up to 82% faster than code generated by `gcc -O3` on realistic benchmarks. Linki Tools — a startup that helps customers develop toolchains for programming various embedded systems — create S10 superoptimization framework based on GreenThumb. The initial prototype of S10 is built upon GreenThumb and has been refactored into a commercial product that supports multiple variations of RISC-V ISAs.

Resource Allocation Synthesis

Second, I envision that resource allocation synthesis — which searches for an optimal solution that satisfies hard resource constraints — can be used to sidestep the development of complicated algorithms to solve complex resource allocation problems. Prior work has used synthesis (as known as *constraint solving* in the compiler literature) to perform optimal register allocation [14, 59], instruction scheduling [93, 168], instruction selection [22, 169], high-level synthesis [87], partitioning [114], and layout and routing [170]. However, it remains difficult to encode a resource allocation problem into constraints that can be solved efficiently. In Chlorophyll, I demonstrate that program partitioning (the first component in Figure 1.3) can be formulated as a type inference problem, which can be easily encoded into SMT formulas by implementing a type checker in Rosette [159, 160]. This approach eases the development of the partitioning synthesizer by utilizing Rosette to automatically generating SMT formulas through symbolic evaluation.

Later, I develop a **resource-mapping library (Chapter 5)**, a prototype toward a retargettable resource allocation mapping framework. Using this library, users can simply implement a function that calculates the cost of a given candidate and use assertion constructs to specify hard constraints such as memory capacity constraints. The framework then automatically generates integer linear programming (ILP) constraints, which can be solved more efficiently than does SMT, for resource allocation problems [80, 114, 116, 168]. I modify the partition type checker in Chlorophyll to use this library, which reduces its search time by more than a few orders of magnitude.

1.6 Summary of Contributions

In summary, this thesis makes the following contributions:

- Chlorophyll (Chapter 2), the first synthesis-aided compiler that applies both program synthesis and resource allocation synthesis as parts of the main workflow of the compilation process. It provides a programming model that allows programmers to design parts of the high-level program partitioning and layout, and leave low-level details for the compiler to fill in.
- Floem (Chapter 3), a programming system for developing network applications in a combined CPU-NIC environment. Its programming abstractions allow programmers to precisely specify offloading strategies and let the system infer what data need to be transferred and handle low-level communication details between CPU and NIC.
- GreenThumb (Chapter 4), a superoptimizer construction framework that tackles the scalability and retargettability challenges of superoptimization. It is equipped with a new scalable superoptimization algorithm and designed to be extensible to new ISAs. The Linki Tools company has developed commercial superoptimization framework, S10 [99], from the resurgence of GreenThumb. The company used S10 to build superoptimizers for many variants of RISC-V and is in the process of developing superoptimizers for x86 and ARM.
- The resource-mapping library (Chapter 5), a prototype toward a scalable, retargettable resource allocation synthesis that tackles the challenge of formulating resource mapping problems into efficient constraints.

Chapter 2

Chlorophyll: Programming Spatial Architectures

Energy requirements have been dictating simpler processor implementations with more energy dedicated to computation and less to processor control. Simplicity is already the norm in low-power systems, where 32-bit ARM dominates the phone computer class [155]; the 16-bit TI 430MSP is a typical example of a low-power embedded controller; the even simpler 8-bit Atmel AVR controller powers Arduino [15, 109].

The GreenArrays GA144 is a recent example of a low-power minimalistic spatial processor[1], composed of many small, simple, identical cores [64]. Likely the most energy-efficient commercially available processor, it consumes 9-times less energy and runs 11-times faster than the TI MSP430 low-power microcontroller on a finite impulse response benchmark [17]. Naturally, energy efficiency comes at the cost of low programmability; among the many challenges of programming the GA144, programs must be meticulously partitioned and laid out onto the physical cores.

We imagine that future low-power processors will likely be similar to the GA144. First, they will likely be spatial with simple interconnects between resources or cores. Second, they will likely have radically different ISAs from what we commonly use today. Third, they will likely be minimalistic, providing little programmability support and therefore placing a greater burden on programmers and compilers.

Materials in this chapter are based on work published as (1) Phothilimthana et al., "Chlorophyll: Synthesis-Aided Compiler for Low-Power Spatial Architectures," in proceedings of PLDI 2014 [121] and (2) Phothilimthana et al., "Compiling a Gesture Recognition Application for a Low-Power Spatial Architecture," in proceedings of LCTES 2016 [123].

[1] A spatial architecture is an architecture for which the user or the compiler must assign data, computations, and communication primitives explicitly to its specific hardware resources such as computing units, storage, and an interconnect network.

2.1 Contributions and Rationale

In this chapter, we introduce a new programming model and a synthesis-based compiler for such spatial processors. Our primary hardware target is the GA144 which takes these design features — spatiality, idiosyncrasy of ISA, and minimalism — to extremes, maximizing the demands on our programming tool chain; if we can build a synthesizer for this processor, we should be able to build ones for other low-power sptial processors as well.

Our programming model allows programmers to selectively partition key data structures and code, leaving the remaining partitioning and communication code generation to synthesizers. In particular, we design the programming model with a specific division of responsibilities as follows.

A programmer is responsible for:

- Partial program partitioning and layout. Programmers sometimes prefer to control how data structures and code should be layouted but not to deal with the low-level details of the resulting communication code.
- Control-flow statement replication. Although this decision may not need human's insight, without a programmer's guidance, the compiler will have to deal with a very complicated cost model. Therefore, we delegate this task to the programmer.
- Parallelism. We do not support automatic parallelization because of conflicting goals. The compiler has to choose between minimizing for latency (consequently, obtaining parallelism) and resource usage (consequently, minimizing power consumption). These are conflicting goals because parallelism on GA144 requires more cores and memory. We believe that this kind of decision should be made by the programmer and not by the compiler.

Synthesizers are responsible for:

- The rest of program partitioning, layout, and routing. This is because heuristics often fail to find optional solutions, and developing a good heuristic algorithm to solve this task is difficult.
- Instruction-level optimization. GA is a stack-based ISA, and there is no well-known optimization passes developed for stack-based instructions, so we delegate this task to a synthesizer to automatically discover optimizations.

Classical transformations are responsible for tasks that can be solved by simple transformations:

- Code separation from a single program into 144 per-core programs.
- IR-to-assembly transformation (with no optimizaiton).

2.2 Overview

CHLOROPHYLL compiles a high-level program to spatial machine code via synthesis. However, solving large and complex problems with synthesis is infeasible. We demonstrate that the compilation problem can be decomposed into partitioning, layout and routing, code separation, and code generation, as depicted in Figure 1.3. Most of these subproblems are difficult for classical compilers but can be solved naturally using synthesis techniques.

Step 1 (partition) The input to this step is a source program with *partition annotations* which specify the logical core (partition) where code and data reside. The annotations allow the programmer to provide insight about the partitioning or experiment with different partitioning just by changing the annotations. An input program does not have to be fully annotated. For example, in this program

```
int@0 mult(int x, int y) { return x * y; }
```

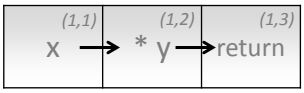
we specify that the result will be delivered at partition 0 but do not specify the partitions of variable x, y, and operation $+$.

The compiler then infers (i.e., synthesizes) the rest of the partition annotations such that each program fragment (per-core program) fits into a core, minimizing a static over-approximation of the amount of messages between partitions. Here is one possible mapping (for a very tiny core):

```
int@0 mult(int@2 x, int@1 y) { return (x!1 *@1 y)!0; }
```

The inferred annotations indicate that when function `mult` is called, x is passed as an argument at partition 2 and y is passed as another argument at partition 1. ! is a send operation. The program body's annotations specify that the value of x at partition 2 is sent to partition 1, and is multiplied with the value of y. Finally, the result of the addition is sent to partition 0 as the function's return value.

Step 2 (layout) The layout synthesizer maps program fragments onto physical cores, minimizing a refined approximation of communication costs. It also determines a communication path (routing) between each pair of cores. We map this synthesis problem to an instance of the well-known Quadratic Assignment Problem (QAP) which can be solved exactly or approximately [90, 46, 51, 145]. We chose to use the Simulated Annealing algorithm as it is one of the fastest techniques and produces a nearly optimal solution [46]. Given the partitioned *mult* function from the previous step, the figure below shows the result of this step.



Step 3 (code separation) The separator splits the fully partitioned program into per-core program fragments, inserting sends and receives for communication. This step uses a classical program transformation. We guarantee that the resulting separated programs are deadlock-free by disallowing instruction reordering within each core. Our running example results in these program fragments:

```
// core(1,1) core ID is (x,y) position on the chip
void mult(int x) { send(EAST, x); }
// core(1,2)
void mult(int y) { send(EAST, read(WEST) * y); }
// core(1,3)
int  mult()      { return read(WEST); }
```

Step 4 (code generation) The code generator first naively compiles each program fragment into machine code (arrayForth, a stack-based assembly for GA144). The code is then optimized with a superoptimizing synthesizer, which searches the space of possible instruction sequences to find ones that are correct and fastest [103]. Although the superoptimizer is allowed to reorder evaluations, it preserves the order of sends and receives which is sufficient to prevent deadlock. We apply a sliding window technique to the synthesizer to adaptively merge small code sequences into bigger ones and input it back into the synthesizer. The synthesizer persistently caches synthesized code to avoided unnecessary recomputation.

Figure 2.1 illustrates these four steps with a larger example, `LeftRotate` program. Figure 2.1(b) shows the result after partitioning the program in Figure 2.1(a) with 64 words of memory per core. The layout and routing of the program is shown in Figure 2.1(c) after the layout step, and Figure 2.1(d) displays the program at core (2,6) after the code separation step.

2.3 Programming Model for Partitioning

In a very limited-resource environment such as a very small many-core, distributed-memory processor, a program partitioning strategy is very critical because it not only affects the performance of the application but, more importantly, determines whether the application can fit and run on the processor. In this section, we present programming abstractions to control mapping of data and code to different partitions, as well as replication of program control flow constructs (i.e., program structures). These abstractions simplify reasoning about partitioning and obviate the need for explicit communication code. We achieve these goals by:

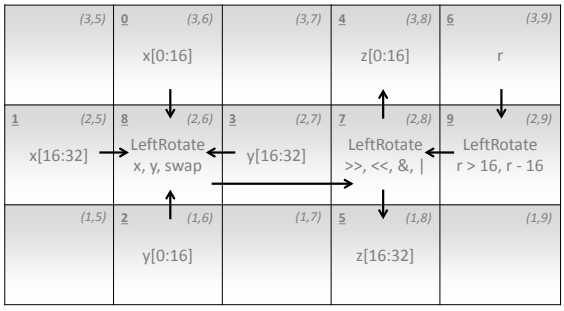
- extending a simple type system with a *partition type* and optimally inferring unspecified partitions, and
- introducing a language construct to control the replication of control statements, trading communication efficiency for code size reduction.

```
1  int leftrotate(int x, int y, int r) {
2    if(r > 16) {
3      int swap = x;
4      x = y;
5      y = swap;
6      r = r - 16;
7    }
8    return ((y >> (16 - r)) | (x << r)) & 65535;
9  }
10
11 void main() {
12   int@{[0:32]=0,[32:64]=1} x[64];
13   int@{[0:32]=2,[32:64]=3} y[64];
14   int@{[0:32]=4,[32:64]=5} z[64];
15   // x[0] to x[31] live at partition 0,
16   // x[32] to x[63] live at partition 1, and so on.
17
18   int@6 r = 0;
19   for (i from 0 to 64) {
20     z[i] = leftrotate(x[i],y[i],r) -@place(z[i]) 1;
21     r = r +@place(r) 1; // + happens at where r is
22     if (r > 32) r = 0;
23   }
24 }
```

(a) Input source code written in CHLOROPHYLL

```
int @7 leftrotate(int@8 x, int@8 y, int@9 r) {
  if(r >@9 16) {
    int@8 swap = x;  x = y;  y = swap;
    r = r -@9 16;
  }
  return ((y >>@7 (16 -@7 r!7)) |@7
         (x!7 <<@7 r!7)) &@7 65535;
}

void main() {
  int@{[0:32]=0,[32:64]=1} x[64];
  int@{[0:32]=2,[32:64]=3} y[64];
  int@{[0:32]=4,[32:64]=5} z[64];
  int@6 r = 0;
  for (i from 0 to 32) {
    z[i] = leftrotate(x[i]!8,y[i]!8,r!9)!4 -@4 1;
    r = r +@6 1;
    if (r >@6 32) r = 0;
  }
  for (i from 32 to 64) {
    z[i] = leftrotate(x[i]!8,y[i]!8,r!9)!5 -@5 1;
    r = r +@6 1;
    if (r >@6 32) r = 0;
  }
}
```

(b) Output from partitioner when memory is 64 words



(c) Output from layout synthesizer. Numbers at the top-left corners represent partition IDs corresponding to the partition annotations in the source code. Numbers at the top-right corners are physical core coordinates.

```
void leftrotate(int x,int y) {
  if(read(E)) {
    int swap = x;  x = y;  y = swap;
  }
  write(E,y);
  write(E,x);
}

void main() {
  for(i from 0 to 32) {
    leftrotate(read(N), read(S));
  }

  for(i from 32 to 64) {
    leftrotate(read(W), read(E));
  }
}
```

(d) Program at core (2,6) after code separation

Figure 2.1: Example program written in CHLOROPHYLL, and intermediate results from partitioning, layout, and code separation steps

Partition Annotation

CHLOROPHYLL syntax is a subset of C with some modifications and *partition annotation*, specifying the partitions of data and operations. Programmers must be able to use partition annotations to:

(1) place data and operations to specific cores

(2) constrain some data and operations to be on the same core

(3) distribute an array onto multiple cores, and

(4) replicate some operations on multiple cores to process data in parallel.

Following these requirements, we design partition annotations (A) to be expressed as follows:

$$
\begin{array}{rcl}
A & := & N \mid \{range = N, ...\} \texttt{ | place}(var) \texttt{ | place}(array) \\
range & := & [N : N] \\
N & := & \texttt{natural number} \\
var & := & \texttt{variable} \\
array & := & \texttt{array access}
\end{array}
$$

For (1), to place data or an operation to a specific partition (logical core), programmers annotate a variable or an operation with $place(l)$. Programmers can then map a logical core l to a physical core p as explained later in this section. We do not want l in $place(l)$ to control the mapping to a physical core directly because we do not want the programmers to reason about the chip's layout and data routing when they want to partition programs only at the logical cores level.

For (2), to constrain a variable or an operation y to be at the same core as another variable x, programmers annotate y with $place(x)$.

For (3), to distribute an array x onto multiple cores, programmers annotate x with $\{[n_0 : n_1] = l_0, [n_1 : n_2] = l_1, ...\}$. The annotation will put $x[i]$ on logical core l_k, when $n_k \leq i < n_{k+1}$.

For (4), to replicate an operation on multiple cores to process a distributed array x in parallel, programmers annotate the operation with $place(x[i])$, where i is a loop variable. $place(x[i])$ refers to the partition where the ith entry of array x lives. Note that $place(x[i])$ can only be used inside the body of a `for` loop with an iterator i.

Example. Figure 2.1(a) shows `LeftRotate` program implemented in CHLOROPHYLL.

- On line 18, we set the partition of variable r to be 6 by annotating its declaration.
- On line 21, we annotate + with $place(r)$ to execute + on the same partition as r.
- On line 12, we assign the partitions of distributed array x such that for $0 \leq i < 32$, $x[i]$ lives in partition 0, and the rest in partition 1.

- On line 21, operation + is assigned to partition 6.
- On line 20, operation − is assigned to $place(z[i])$; when $0 \leq i < 32$, operation − at partition 4 is executed, and when $32 \leq i < 64$, operation − at partition 5 is executed. The loop on line 19 is a loop with a statically-known bound of the form `for (i from e1 to e2) {...}`, where $e1$ and $e2$ are constants. The iterator i starts from $e1$ and is incremented by 1, and the loop condition is $i < e2$.

Note that most of the data and operations in the program are left unannotated. Their partitions will be automatically inferred by the partitioning synthesizer to minimize communication between cores. Hence, CHLOROPHYLL allows the programmers to specify data and operation when they have a rough partitioning plan in mind, and to leave some unannotated when they want the compiler to decide.

Our partition annotations resemble data distribution annotations from many high performance computing languages [28, 37, 107, 112, 173]. While a scalar variable is replicated on all processors in these languages, a scalar variable lives in only one place in CHLOROPHYLL because of our highly-restricted space requirement; same for an operator. Our space requirement is a major factor that distinguishes our language design from prior high-performance parallel languages.

Language Constructs

Constants, variables, arrays, operators, and statements all occupy space in memory. Most programming constructs — such as variable declarations, variable accesses, variable assignments and binary operations — occupy a constant amount of memory in one partition, so we can estimate the space occupied by the program with a simple lookup table. However, we have to handle control flow constructs and arrays with more care as they may require more complex communication between the involved partitions.

Arrays

There are two kinds of arrays:

- ***Non-distributed arrays*** only live in one partition. An index into this type of array has to live at the same partition as the array itself.
- ***Distributed arrays*** live in multiple partitions. Arrays x, y and z from `LeftRotate` are examples. This type of array can only be indexed by affine expressions of surrounding loop variables and constants. Accessing this type of array requires no communication because the indexes are comprised of loop variables, which live in every body partition. CHLOROPHYLL currently does not support other kinds of indexing into distributed arrays.

A distributed array can be declared in two ways. First, the distributed array x in `LeftRotate` in Figure 2.1 is declared as:

```
int@{[0:32]=0,[32:64]=1} x[64];
```

Alternatively, it can be declared using the following syntax:

```
int[blocksize]@{p_1,p_2,...,p_n} array[N];
```

where $n = ceil(N/blocksize)$. With the second syntax, the same distributed array x can be declared as:

```
int[32]@{0,1} x[64];
```

Control Flow Constructs

Chlorophyll provides a construct for programmers to control the replication of control statements, trading communication efficiency for code size reduction. We first describe the two extreme strategies: the *actor model*, which replicates no code, and the *SPMD model*, which replicates all control statements. We then describe the construct to control the replication.

SPMD vs. Actor Partitioning Strategy

An invariant of our partitioning is that data and non-control computations are not replicated; they are always assigned to exactly one logical partition, which is mapped to one physical core. The programmer controls only replication of control statements (`if`s and loops), which determine the control condition, i.e., when a statement is executed.

The Actor Strategy. In the actor strategy, when a partition p_1 needs a value to be computed on another partition p_2, it sends a request message to p_2 and waits until the value is returned. On partition p_2, the computation is executed by an actor that is active only when responding to a request. This strategy is similar to how X10 handles place changes [134, 153]. While the actor strategy minimizes code duplication, it may incur more communication. Consider the program in which the function `f` executes on partition 2.

```
int@2 f(int@2 i) { ... }
int@1 x;
for (i from 0 to 100) x += f(i);
```

When `f` is an actor, each loop iteration requires three messages (the request to execute `f`, the argument, and the return value):

```
// Partition 1
for (i from 0 to 100)
  x += actor_call(f, i); // call f in the other partition

// Partition 2
port_execution_mode(); // execute f when requested
int f(int i) { ... }
```

Many messages from 1 to 2 can be eliminated by replicating the loop on partition 2, which is done by the the SPMD model.

SPMD Strategy. The SPMD model, introduced by Callahan and Kennedy [34], replicates all control flow constructs onto every partition, in the spirit of the Single-Program Multiple-Data model [84]. When using this strategy, each partition becomes independent in that it decides when to execute each statement, but naturally the replicated control flow constructs need to obtain their predicate values, and these may need to be communicated from the partition that computes the predicate.

More specifically, when the body of a control flow construct (`if` or `while`) is spread across many partitions, called *body partitions*, the control flow construct needs to be replicated and placed in each of these partitions. The condition expression is not replicated, but its result is sent to all the body partitions, each of which in turn uses the received value as its condition. For a loop with a statically-known bound, the condition expression is also replicated and computed locally. For the example program above, the SPMD strategy will split the program into:

```
// Partition 1
for (i from 0 to 100) x += recv();

// Partition 2
int f(int i) { ... }
for (i from 0 to 100) send(f(i));
```

Each of the body partitions uses its own copy of i. While this strategy duplicates control flow constructs, it can reduce communication significantly. In this program, only one message is sent per iteration. However, if the control flow of a program is complex, and the predicates of the control flow statements are not known at compile time, this strategy may require a lot of messages for sending the predicate values.

Tradeoffs. In summary, both strategies produce more efficient code in different scenarios. The actor strategy can significantly help reduce code size and also amount of communication when the control flow of the program is complex, while the SPMD strategy may be better if the control flow is simple and statically determined, such as a loop with constant bounds.

Language Construct for Controlling Replication

Rather than controlling replication of control statements in each individual partition, we control replication at the granularity of a function (a set of partitions or cores). The programmer specifies which functions should be compiled into actors, and the remaining function calls are invoked under replicated control flow. Programmers *actorize* a function by defining:

```
// Do not specify the requester and master actor
actor FUNC;
// Specify the requester and master actor
```

```
actor FUNC(REQUESTER => MASTER);
```

Without any actor directive, the compiler defaults to the SPMD partitioning strategy. When all functions are annotated as actors, the actor strategy is used exclusively.

One of the actor partitions in the function is a dedicated *master actor* partition, and the rest are *subordinate actor* partitions. A *requester* partition is responsible for sending a remote execution request to the master actor to invoke the function. The master actor in turn triggers subordinate actors to invoke their functions through data dependencies. More specifically, a subordinate actor waits for data to be used in its computations inside the function; the data essentially triggers the rest of the computations inside the function. Relying on these triggers, program fragments of actor partitions of a function `func` do not need to contain the control statements between the function `main` to the calls to `func`. The program fragments, however, should contain the control flow slice of the program starting from the function `func`. If a partition does not belong to any actor function, it contains the control statements starting from `main`.

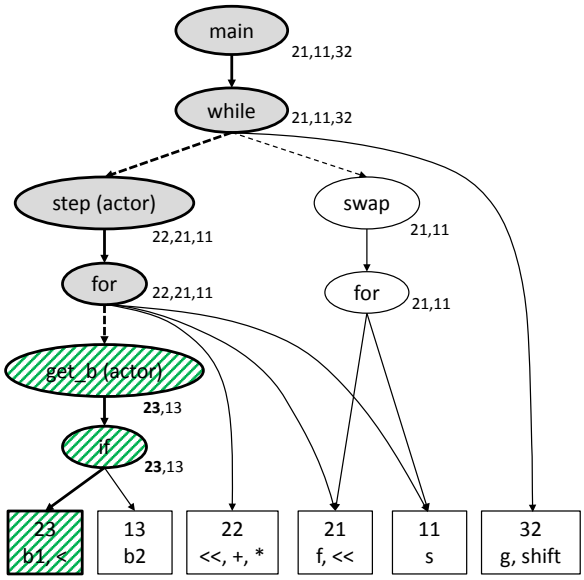
Consider the fully-annotated simplified snippet of code from a hand-gesture recognition application in Figure 2.2(a). We actorize the function `get_b` by annotating `actor get_b(22=>23)`, specifying that partition 22 is a requester and 23 is a master actor. Since `get_b` also contains partition 13 in addition to partition 23, partition 13 becomes a subordinate actor. Figure 2.3 displays illustrative per-core program fragments generated by our compiler before being converted into machine code. Figures 2.3(b) and 2.3(c) display the program fragments of partition 23 and 13 respectively. Notice that both partitions do not have the control statements between `main` and the call to `get_b` (i.e., `while`, `step`, and `for`). Partition 23, the master actor, is in the port execution mode (line 1), waiting for 22 to send a remote execution request to invoke `get_b`. When 23 finishes executing `get_b`, it goes back to the port execution mode waiting for the next request. Partition 23 triggers 13, the subordinate actor, to start computations in `get_b` by sending the predicate of `if` (data dependency). Partition 13 waits for this data (line 5) to start its task. When it finishes the task, it loops back to the beginning of the program to handle the next request. Figure 2.3(a) displays the partitioned program fragment at 22. Since it is the requester for `get_b`, it sends a remote execution request (line 7) to 23, the master actor. Partition 22 itself is an actor for another function `step`, so it is also in the port execution mode.

Design Decisions

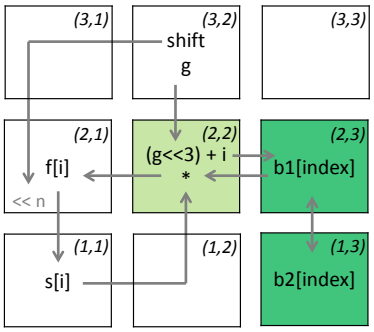
To support both partitioning strategies, we make the following design decisions. First, we let programmers actorize at a function-granularity level. Among the constructs provided by Chlorophyll, a partition and a function are referenceable entities that can be considered as an actor, an entity that acts upon receiving a request. However, we believe that programmers prefer to reason about functionality of programs rather than reason about implementation details. Furthermore, if programmers let the compiler infer partition types for most parts of their programs, they will not know which partitions are responsible for which parts of the programs, so they cannot actorize their programs appropriately. Second, we let programmers

```
1  fix1_t@21 f[8];
2  fix1_t@11 s[8];
3  fix1_t@23 b1[32];
4  fix1_t@13 b2[32];
5
6  actor get_b(22=>23);
7  fix1_t@23 get_b(int@23 index) {
8    if (index <@23 32)
9      return b1[index];
10   else
11     return b2[index -@13 32];
12 }
13
14 actor step(32=>22);
15 void step(int@22 g) {
16   for (i from 0 to 8)
17     f[i] = s[i] *@22 get_b((g <<@22 3) +@22 i);
18 }
19
20 void swap(int@21 n) {
21   for (i from 0 to 8) s[i] = f[i] <<@21 n;
22 }
23
24 void main() {
25   while(1) {
26     int@32 g = ...; int@32 shift = ...;
27     step(g);
28     swap(shift);
29   }}
```

(a) CHLOROPHYLLsource. Blue, pink, and purple highlight data and computations assigned to partition 22, 23, and 13 respectively. `fix1_t` is a fixed point data type with one bit for the integer part.



(b) CDG. An oval represents a control flow construct node. A rectangle represents a partition node, grouping operations and data that belong to the partition. The entire highlighted path is the *control flow slice* of partition 23. The green striped-highlighted path is its *relevant control flow slice*. Numbers attached to each node are the *relevant partitions* of the node.



(c) Layout and routing. Partition xy is mapped to physical core (x, y). Dark and light green denote actor cores of `get_b` and `step` respectively.

Figure 2.2: A simplified example program taken from the gesture recognition application

```
1  port_execution_mode(N);
2
3  void step(int g) {
4    for (i from 0 to 8) {
5      int s = recv(S);
6      // call actor get_b in p.23 (east)
7      int b = actor_call(E, get_b, (g << 3) + i);
8      send(W, s * b);
9    }
10 }
```

(a) A requester of the function `get_b`: partition 22 or core (2,2)

```
1  port_execution_mode(W); // wait for request from p.22
2
3  fix1_t b1[32];
4  fix1_t get_b(int index) {
5    int cond = index < 32;
6    send(S, cond);    // send condition to p.13
7    if (cond) {
8      return b1[index];
9    } else {
10     send(S, index);  // send index to p.13
11     return recv(S); // return value from p.13
12   }}
```

(b) A master actor of the function `get_b`: partition 23 or core (2,3)

```
1  fix1_t b2[32];
2
3  void get_b() {
4    // wait for condition from p.23 to start
5    if (recv(N)) { }
6    else {
7      // get index from p.23 and return value to p.23
8      send(N, b2[recv(N) - 32]);
9    }
10   get_b(); // loop back to the beginning
11 }
12
13 void main() { get_b(); }
```

(c) A subordinate actor of the function `get_b`: partition 13 or core (1,3)

Figure 2.3: Program fragments of partitions 22, 23, and 13 generated by the compiler when compiling the program in Figure 2.2. The compiler places `blue`, `pink`, and `purple` highlighted data and computations from Figure 2.2(a) in partitions 22, 23, and 13 respectively. `N`, `S`, `E`, and `W` stand for north, south, east, and west ports.

make a decision if each function should be an actor or not. While it is possible to automate this decision, this automation is not the focus of this thesis. Third, we associate an actor function with a requester and a master actor. To invoke an actor function, the requester sends a remote execution request to the master actor, which in turn triggers the subordinate actors through data dependencies. We could have made the master actor trigger the subordinate actors using explicit remote execution requests, the same way the requester triggers the master actor; however, using remote execution requests may incur a lot of communication, so we decide to utilize data dependencies instead. Nevertheless, we do not rely on the data dependency to invoke the master actor because we would like to support actorizing functions with no argument that perform I/O activities.

Restrictions Imposed by Our Decisions. A subordinate actor partition is invoked upon receiving any data from an expected neighbor port. Since one message sent between two GA144 cores is only 18-bit word, we choose to put only data (no metadata) in a message sent to a subordinate actor partition. As a result, subordinate actor partitions do not have a mechanism to distinguish between different requests for different tasks. Therefore, we restrict a partition to be an actor for no more than one actor function. If a partition is in more than one actor functions, that partition will not be an actor for any function, and the partition will have the entire control flow from `main`. If a partition is an actor for a function, that partition cannot be used anywhere else outside the function, including for routing messages for any computation outside the function. Thus, too much actorization may cause the compilation to fail due to its implication on the routing restriction.

Advantages Over Low-Level Partitioning Control. The GreenArrays vendor provides a programming environment for GA144, called arrayForth. It allows programmers to explicitly write separate programs for individual cores with the flexibility to manually duplicate data, operations, and control statements, however, in a stack-based assembly language. Recall that Chlorophyll compiles to arrayForth. We can think of arrayForth as an MPI-style programming model. Initially, partitioning control flow statements in arrayForth may seem more intuitive than controlling the partitioning strategy in Chlorophyll because programmers maybe more familiar with MPI than SPMD and actor concepts. However, there are several advantages to program GA144 using Chlorophyll with the programmer-controlled hybrid partitioning strategy. First, both SPMD and actor partitioning strategies guarantee that the generated code is deadlock-free, unlike MPI. Although actorization may cause the compilation to fail because of too many constraints on communication routing, the failure happens at compile time, so programmers can fix their programs accordingly. Second, it is easier for a programmer to write a whole program in a sequential style than to write separate program fragments that run in parallel. Furthermore, using annotations to control the partitioning strategy enables programmers to easily explore different ways to partition program structures just by inserting or deleting actor annotations.

Type system

We present a simplified version of our complete type system to convey the core idea. Types in CHLOROPHYLL can be expressed as follows:

$$\begin{array}{rlrl} \tau := & \tau@\rho & \tau := & \texttt{val} \mid \texttt{int} \mid \texttt{void} \\ \rho := & N \mid \texttt{any} \mid \rho_{dist} & \rho_{dist} := & \{(N,)^+\} \\ N := & \texttt{natural number} & & \end{array}$$

Our types consist of data types τ and partition types ρ. For simplicity, the data types only include `int`, `val`, and `void`. ρ_{dist} is a type of distributed array.

The typing rules shown in Figure 2.4 (omitting some trivial rules) enforce that operands and operators are in the same partition. Constants and loop variables have partition type `any` indicating that they can be at any partition. The *partition subtype* rule allows an expression

$$\frac{}{\text{val} < \text{int}}\ [\text{basic subtype}] \qquad \frac{}{\text{any} < \text{N}}\ [\text{partition subtype}]$$

$$\frac{\tau_1 < \tau_2 \qquad \rho_1 < \rho_2}{\tau_1@\rho_1 < \tau_2@\rho_2}\ [\text{subtype}] \qquad \frac{}{\Gamma \vdash n : \text{val@any}}\ [\text{const}]$$

$$\frac{x : \tau@\rho \in \Gamma}{\Gamma \vdash x : \tau@\rho}\ [\text{variable}] \qquad \frac{i : \text{val@any} \in \Gamma}{\Gamma \vdash i : \text{val@any}}\ [\text{iterator}]$$

$$\frac{\Gamma x : \tau@\{\rho\} \in \Gamma}{\Gamma \vdash x : \tau@\{\rho\}}\ [\text{array}] \qquad \frac{\Gamma x : \tau@\{\rho_1, \rho_2, ..., \rho_n\} \in \Gamma}{\Gamma \vdash x : \tau@\{\rho_1, \rho_2, ..., \rho_n\}}\ [\text{dist array}]$$

$$\frac{\Gamma \vdash e_1 : \tau_1@\rho_1 \quad \Gamma \vdash e_2 : \tau_2@\rho_2 \quad \tau_1@\rho_1 < \tau@\rho \quad \tau_2@\rho_2 < \tau@\rho}{\Gamma \vdash e_1 \ \ \text{op}@\rho\ e_2 : \tau@\rho}\ [\text{op}]$$

$$\frac{\Gamma \vdash f : \tau_1@\rho_1 \to \tau_2@\rho_2 \quad \Gamma \vdash e : \tau_3@\rho_3 \quad \tau_3@\rho_3 < \tau_1@\rho_1}{\Gamma \vdash f\ e : \tau_2@\rho_2}\ [\text{function call}]$$

$$\frac{\Gamma \vdash x : \tau@\{\rho\} \qquad \Gamma \vdash e : \tau_e@\rho_e \qquad \tau_e@\rho_e < \text{int}@\rho}{\Gamma \vdash x[e] : \tau@\rho}\ [\text{access array}]$$

$$\frac{\Gamma \vdash x : \tau@\{\rho_1, ..., \rho_n\} \qquad \Gamma \vdash e : \text{val@any} \qquad e \downarrow v}{\Gamma \vdash x[e] : \tau@\rho_v}\ [\text{access dist-array}]$$

$$\frac{\Gamma \vdash e : \tau@\rho_1}{\Gamma \vdash e!\rho_2 : \tau@\rho_2}\ [\text{send}]$$

Figure 2.4: Typing rules

with partition type any to be used everywhere. The *access dist-array* rule ensures that the index to a distributed array is only comprised of loop variables and constants, and the index is not out-of-bound, in order to generate valid code for accessing the array.

! is an operation for sending data from one partition to another. It will be translated to both a write operation at the sending partition and a read operation at the receiving partition. It is the only operation that accepts an operand whose partition type may not be a subtype of the output's partition type. The compiler automatically generates this operator during type checking and inferring, so programmers are not required to insert any ! in the source code.

Parallelism

Implicit Parallelism

Besides pinning data and operations to partitions, partition type can be used for expressing parallelism. If two operations have different partition types, they will be executed at two different cores and maybe executed at the same time in parallel, depending on data and control dependency. Consider the following program:

```
int@1 x;
int@2 y;
x = x +@1 1;
y = y -@2 1;
```

The increment of x and and decrement of y run in parallel in partition 1 and 2 respectively.

Data Parallelism from Distributed Array

The language also allows programmers to declare distributed arrays, which can be used to express data parallelism. For example,

```
// The first 16 elements are in partition 0.
// The last 16 elements are in partition 1.
int@{[0:16]=0, [16:32]=1} x[32];
for (i from 0 to 32)
  x[i] = x[i] +@place(x[i]) 1;
```

will be separated to

```
// Partition 0
int x[16];
for (i from 0 to 16)
  x[i] = x[i] + 1;


// Partition 1
int x[16];
for (i from 16 to 32)
  x[i-16] = x[i-16] + 1;
```

Consequently, the two parts of the array are incremented in parallel.

Parallel Module

Assume that we would like to execute the same function on different data in parallel. For example, we want to update the belief states of HMM classifiers, running these two `hmm_step`, in parallel in a hand-gesture recognition program:

```
hmm_step(acc,model1);
hmm_step(acc,model2);
```

`model1` and `model2` contain parameters for classifiers 1 and 2 respectively, and `acc` are 3-axis accelerometer data. These two `hmm_step`s do not run in parallel because only one set of partitions (hence, one set of cores) is responsible for executing the function. In order to make them run in parallel, we need to make two copies of `hmm_step` — e.g., `hmm_step1` and `hmm_step2` — and ensure that `hmm_step1` and `hmm_step2` do not use the same partitions. Additionally, we need to explicitly assign every data and operation in the function to a partition to make sure that the two copies of `hmm_step` do not share any common partition. Doing this manually is highly error-prone and unproductive. Even then, the two functions may not run in parallel because communication routing may introduce dependency between the two functions.

We introduce an explicit parallel construct called *parallel module*, which can be used to express this kind of parallelism. Module and module instance are syntactically similar to class and object. For example, the program in Figure 2.5(a) uses the module construct to update the belief states of the HMM classifiers in parallel. Despite its syntactic similarity to class, module behaves like a macro. Our module expansion pass expands the program in Figure 2.5(a) to the program in Figure 2.5(b), similar to a macro expansion. After the expansion, we obtain the program in the original Chlorophyll language. To ensure parallelism, Chlorophyll enforces that two module instances of the same module occupy two disjoint sets of partitions. A module instance is essentially a privatization [68, 115, 161] of variables and operations in a module. Note that our privatization problem is easier than the privatization problem presented in the literature, since we let the programmers guide the compiler what to privatize.

```
// Define module.
module Hmm(model_init) {
  fix1_t model[N] = model_init;
  fix1_t step(fit1_t[] acc) { ... }
}

// Create module instances.
hmm1 = new Hmm(model1);
hmm2 = new Hmm(model2);

// Call two different functions.
hmm1.step(acc);
hmm2.step(acc);
```

(a) Source code in Chlorophyll

```
// Expanded from module instance 1.
fix1_t hmm1_model[N] = model1;
fix1_t hmm1_step(fit1_t[] acc) { ... }

// Expanded from module instance 2.
fix1_t hmm2_model[N] = model2;
fix1_t hmm2_step(fit1_t[] acc) { ... }

hmm1_step(acc);
hmm2_step(acc);
```

(b) De-sugared code after module expansion

Figure 2.5: Example of a parallel HMM classification program using the module construct

Parallel Map and Reduce

We also support parallel map and reduce on distributed arrays. CHLOROPHYLL handles parallel map and reduce the same way it handles parallel module: using privatization by desugaring into the original language and adding constraints to the partition type inference and the routing algorithm. We use parallel `map` and `reduce` to inform the compiler to perform function privatization. For example, the compiler will expand the program with `map` on the left into the program on the right, and ensures that functions `ssd0` and `sdd1` occupy disjoint sets of partitions.

```
// Input source program
int ssd(int a, int b) {
  return (a - b) * (a - b);
}

void main() {
  int[5]@{0,1} x[10];
  int[5]@{0,1} y[10];
  int[5]@{0,1}  z[10];
  z = map(ssd, x, y);
}
```

```
// Intermediate program
int ssd0(int a, int b) { ... }
int ssd1(int a, int b) { ... }

void main() {
  int[5]@{0,1} x[10];
  int[5]@{0,1} y[10];
  int[5]@{0,1}  z[10];
  for (i from 0 to 5)
    z[i] = ssd0(x[i], y[i])
  for (i from 5 to 10)
    z[i] = ssd1(x[i], y[i])
}
```

Similarly, it will expend the program with `reduce` on the left into the program on the right, and ensures that functions `add0` and `add1` occupy disjoint sets of partitions.

```
// Input source program
int add(int a, int b) {
  return a + b;
}

void main() {
  int[5]@{0,1} x[10];
  int ans  = reduce(add, 0, x);
}
```

```
// Intermediate program
int add0(int a, int b) { ... }
int add1(int a, int b) { ... }

void main() {
  int[5]@{0,1} x[10];
  int ans;
  int@0 ans0 = 0;
  int@1 ans1 = 0;
  for (i  from 0 to 5)
    ans0 = add0(ans0, x[i]);
  for (i from 5 to 10)
    ans1 = add1(ans0, x[i]);
  ans = ans0 + ans1;
}
```

The compiler requires the function given as the first argument to `reduce` to be associative.

Pinning Partition to Core

We allow programmers to design their own program layouts by mapping partitions to physical cores as desired by annotating:

```
# PARTITION --> CORE
```

We also support pinning a set of partitions to a set of cores by pinning a module instance. In the program in Figure 2.5(a), we can pin partitions in `hmm1` to be in cores (1,1), (1,2),

(2,1), and (2,2) by:

```
hmm1 = new Hmm(model1)@{(1,1),(1,2),(2,1),(2,2)};
// or
hmm1 = new Hmm(model1)@REG((1,1),(2,2));
```

where `REG(BL,TR)` is an abbreviation for a set of cores covered by a rectangle whose bottom left is at the first argument to `REG`, and top right is at the second argument to `REG`.

Inside a pinned module, we can pin individual partitions as well. For example, in this program:

```
module Hmm(model_init) {
  # 0 --> (0,1)
  fix1_t@0 model[N] = model_init;
  fix1_t@0 process(x,y,z) { ... }
}

hmm1 = new Hmm(model1)@REG((1,1),(2,2));
```

we specify that partitions inside `hmm1` should be placed at core (1,1), (1,2), (2,1), and (2,2), and specifically we want partition 0 in `hmm1` to be at (0,1) relative to the most bottom-left core of `hmm1`, so partition 0 is placed at core $(1+0, 1+1) = (1, 2)$.

Limitations

CHLOROPHYLL does not handle recursive calls and multidimensional arrays. Unbounded loops can be implemented using `while`; however, the `for` loop is currently restricted to have a statically known bound.

2.4 Synthesis-Aided Compiler

Step 1: Partitioning Process

Partitioning a program can be thought of as a type inference on partition types. The partitioning synthesizer is constructed from 1) the *communication interpreter*, which counts the number of communications needed and 2) the *partition space check*, which ensures code and data fit in the memory of the appropriate core.

Communications Interpreter

Let $Comm(P, \sigma, x)$ be a function that counts the number of communications in a given program P with complete annotated partitions σ and a concrete input x. The communication count is calculated with $MaxComm(P, \sigma) = \max_{x \in Input} Comm(P, \sigma, x)$, where $Input$ is a set of all valid inputs to the program, assuming *while* loops are executed a certain number of times (currently 100). $MaxComm$ computes the maximum number of communications by considering all possible program paths.

Figure 2.6 shows the evaluation rules for the interpretation of a program under $MaxComm$. In the figure, $< P, \sigma > \Downarrow n$ is used as $MaxComm(P, \sigma) = n$. For most constructs, the communication count is equal to sum of its components' counts. ! increments the communication count by 1. Loops multiply the count. Conditional statements add the communication count by the number of the body partitions (subtracted by 1 if one of the body partitions is the same as the partition of the result of the condition expression).

Partition Space Check

Given a program with complete partition annotations, the *partition space checker* computes how much space is used in each partition. The compiler only accepts the program if the occupied space in every partition is not more than the amount of memory available in a core. Operations, statements, and communication operations (i.e. read and write) occupy the memory of the partitions they belong to. Constants occupy memory of the partitions in which they are inferred to be (usually the partitions of their operators or the left-hand-side variables they are assigned to).

Partitioning Synthesizer

We implement the communication count interpreter and the partition space checker using Rosette, a language for building light-weight synthesizers [159, 160]. We represent a specified partition annotation as a concrete value and an unspecified partition annotation as a symbolic variable. Given a fully annotated program (one with all concrete partitions), the communication count interpreter compute a concrete number of communications, and the partition space checker simply verifies that the memory constraint holds. Given a partially annotated or unannotated program (a program with some or all symbolic partitions), the result from the communication count interpretation is a formula in terms of the symbolic variables, and the partition space check becomes a constraint on the symbolic variables. To ensure module parallelism, we add an extra constraint to enforce that two module instances of the same module do not share any common partition types; this guarantees that the two module instances can run in parallel because they occupy two disjoint sets of partitions.

Once we obtain a formula from the communication count and the partition space constraint, we query Rosette's back-end solver to find an assignment to the symbolic partitions such that the space constraint holds. If the solver returns a solution, we attempt to reduce the communication count further by asking the solver the same query with an additional constraint setting an upper bound on the count. We lower the upper bound until no solution can be found.

Chapter 5 presents an improved version of the partitioning synthesizer, as a case study for the resource-mapping library, developed as a general framework for solving resource mapping problems.

$$\frac{< e, \sigma >\Downarrow n \qquad \Gamma \vdash e : \mu@\rho_1}{< e!\rho_2, \sigma >\Downarrow n + commBetween(\rho_1, \rho_2)} \text{ [send]}$$

$$\frac{< p, \sigma >\Downarrow n}{<\text{def } f(x_1, x_2, ...) \{ p \}, \sigma >\Downarrow lut[f] \leftarrow n} \text{ [function declaration]}$$

$$\frac{}{< n, \sigma >\Downarrow 0} \text{ [const]}$$

$$\frac{}{< x, \sigma >\Downarrow 0} \text{ [variable]}$$

$$\frac{< e_1, \sigma >\Downarrow n_1 \qquad < e_2, \sigma >\Downarrow n_2}{< e_1 \; bop@\rho \; e_2, \sigma >\Downarrow n_1 + n_2} \text{ [binary-op]}$$

$$\frac{< e, \sigma >\Downarrow n}{< x[e], \sigma >\Downarrow n} \text{ [array-access]}$$

$$\frac{< e_1, \sigma >\Downarrow n_1 \qquad < e_2, \sigma >\Downarrow n_2 \qquad ...}{< f \; e_1 \; e_2 \; ..., \sigma >\Downarrow lut[f] + n_1 + n_2 + ...} \text{ [function-call]}$$

$$\frac{< e_1, \sigma >\Downarrow n_1 \qquad < e_2, \sigma >\Downarrow n_2}{< e_1 := e_2, \sigma >\Downarrow n_1 + n_2} \text{ [assignment]}$$

$$\frac{< c_1, \sigma >\Downarrow n_1 \qquad < c_2, \sigma >\Downarrow n_2 \qquad ...}{< c_1; c_2; ..., \sigma >\Downarrow n_1 + n_2 + ...} \text{ [sequence]}$$

$$\frac{\Gamma \vdash b : \tau@\rho \quad < c, \sigma >\Downarrow n_c \quad < t, \sigma >\Downarrow n_t \quad < f, \sigma >\Downarrow n_f}{< \text{if } c \text{ then } t \text{ else } f, \sigma >\Downarrow n_c + max(n_t, n_f) + commBody(\sigma, \rho, t, f)} \text{ [if-else]}$$

$$\frac{\Gamma \vdash b : \tau@\rho \qquad < c, \sigma >\Downarrow n_c \qquad < b, \sigma >\Downarrow n_b}{< \text{while } c \text{ do } b, \sigma >\Downarrow (n_c + n_b + commBody(\sigma, \rho, b)) \times loopBound} \text{ [while]}$$

$$\frac{< c, \sigma >\Downarrow n}{< \text{for}(i \text{ from } a \text{ to } b) \{ c \}, \sigma >\Downarrow n \times (b - a)} \text{ [for]}$$

```
def commBetween(p1,p2):
  return if p1 == p2 then 0 else 1;

def commBody(sigma,p,body1, body2, ...):
  return |(set of  partitions occupied by
           body1, body2, ... according to sigma) - {p}|
```

Figure 2.6: Evaluation rules for an upper bound on the number of communications. *lut* is the function table. $lut[f] \leftarrow n$ is storing n at index f of the lookup table.

Pre-Partitioning Process: Loop Splitting

Our communication counter and partition space checker cannot reason about distributed partition type such as partition type of $x[i]$ if x is a distributed array because a distributed partition type complicates communication count and space check. To handle this, we introduce a pre-partition process to ensure that for every reference to a distributed array, the reference corresponds to a part of the array in one partition. With this, $place(x[i])$ is just a concrete partition. This process is done by loop splitting.

However, computing loop bounds for the resulting loops is quite complex, so we use Rosette to implement a loop bound synthesizer similar to the way we implemented the partitioning synthesizer. Consider this prefixsum program:

```
int@{[0:5]=0,[5:10]=1} x[10];
for (i from 1 to 10)
  x[i] = x[i] + x[i-1];
```

We first duplicate the loop into k loops and replace the loop bounds with symbolic values. Let k be 3 in this particular example. The main prefixsum loop is expanded to:

```
for (i from a0 to b0) x[i] = x[i] + x[i-1];
for (i from a1 to b1) x[i] = x[i] + x[i-1];
for (i from a2 to b2) x[i] = x[i] + x[i-1];
```

The first loop iterates over i from a_0 to b_0, the second loop from a_1 to b_1, and so on. We then check that $a_0 = 1$ and $b_2 = 10$. For every iteration l such that $0 \leq l < k - 1$, we check that $a_{l+1} = b_l$. For every loop l, we check that each array reference (i.e., $x[i]$ and $x[i-1]$) belongs to only one partition for all i's when $a_l \leq i < b_l$; for the $x[i]$ reference, $x[a_l], x[a_l + 1], ..., x[b_l - 1]$ must be in the same partition. We implemente the checker as if the bounds are concrete. When the bounds are unknown, they become symbolic values, and the checking conditions are used as constraints. Finally, the solver outputs one feasible solution for loop bounds. In this particular example, the output is:

```
for (i from 1 to 5) x[i] = x[i] + x[i-1]; // x[i] at 0, x[i-1] at 0
for (i from 5 to 6) x[i] = x[i] + x[i-1]; // x[i] at 0, x[i-1] at 1
for (i from 6 to 10) x[i] = x[i] + x[i-1]; // x[i] at 1, x[i-1] at 1
```

The final output is the solution with the smallest possible k.

Example

Figure 2.1(b) shows the result after partitioning the program in Figure 2.1(a) with 64 words of memory per core. Notice that ! operations are automatically inserted into the program. If the programmer writes partition annotations such that it is impossible to partition the program into program fragments that fit on cores, this will result in a compile-time error. In bigger processors where each core has 128 words of memory, the function `LeftRotate` can fit entirely on one core, so this step would annotate everything within the function with the same partition.

Post-Partitioning Process: Identifying Relevant Control Flow

We decide which partitions need copies of each control flow construct based on the *relevant control flow slice* of each program partition. After the partitioning process, we define the control flow slice and the relevant control flow slice. The compiler will use this information to route data and separate a program into per-core program fragments.

Control flow slice. We define a *control flow slice* of a partition, based on the program slicing terminology [158], to consist of the control flow constructs in the slice of the source program with respect to the partition's data and computations as a *slicing criterion*. A slice of a program with respect to a slicing criterion can be determined from a Program Dependence Graph (PDG). We are interested in computing a control flow slice of a program with respect to a program partition, so we can use a Control Dependence Graph (CDG), a subgraph of PDG. Typically, a node in a CDG is either a control flow construct or a statement in the program. However, since our slicing criterion is in the level of a logical partition instead of a statement, we group statements, operations, and data that belong to the same partition together in one node, called a partition node. Thus, our CDG contains partition nodes instead of statement nodes. An edge represents a control dependency between nodes in a CDG.

Figure 2.2(b) depicts the CDG of the fully-annotated program in Figure 2.2(a). Ovals represent control flow constructs. Rectangles represent logical partitions. Dashed edges indicate interprocedural control dependency (function calls). A control flow slice of a partition can be directly derived from a CDG. A control flow slice of a partition consists of all paths from the `main` node to the partition node. Figure 2.2(b) highlights the control flow slice of partition 23.

Actor and its relevant control flow slice. With the pure SPMD strategy, a program fragment of a partition generated by the compiler will contain the data and computations assigned to that particular partition and the entire control flow slice of the partition. A program fragment of a non-actor partition needs to include all control flow constructs in its control flow slice. In contrast, a partition of an actor function does not need to own the control flow constructs between `main` and the calls to that particular function because a requester partition is responsible for remotely invoking that function. We call the control flow constructs that an actor partition actually needs (excluding the constructs between `main` and the actor function) the *relevant control flow slice* of the partition.

Therefore, our task is to determine what the relevant control flow slice of each partition in the program is. First, we need to identify which partition is an actor (called *actor partition*) for which function. Identifying an actor partition is not as straightforward as it seems because not all partitions inside an actor function are actors. To identify actor partitions, we perform a reachability analysis on a CDG. Partition p is an actor and belongs to actor function f, if f is the most *immediate* node in the CDG that *covers* p. Function f *covers* p if and only if there is no path from `main` to p when f is removed from the CDG. p can be

covered by multiple actor functions. In such a case, p belongs to the most *immediate* actor function; there is no other actor function between the most immediate actor function and p.

For example, in Figure 2.2(b), partition 23 is covered by both actor functions `get_b` and `step`, but `get_b` is the most immediate function to 23, so 23 belongs to `get_b`. Now consider 21, which is a partition inside the actor function `step`, but it is being used in the function `swap` outside the scope of `step`. According to the CDG, `step` does not cover 21. Thus, 21 is not an actor partition for step.

Once we identify actor partitions for all actor functions, we can compute the *relevant control flow slice* for every partition. Figure 2.2(b) highlights nodes in the relevant control flow slice of partition 23 with green stripes, which is its original control flow slice excluding the control flow constructs from `main` to `get_b`. We can also see that each program fragment of a partition in Figure 2.3 only contains the relevant control flow slice of that particular partition. Partition 22 only contains `step` and `for`. Partitions 23 and 13 only contain `get_b` and `if`. Inversely, at each control flow construct, we can also compute a set of *relevant partitions*, partitions whose relevant control flow slices contain that particular control flow construct. Figure 2.2(b) labels each node in the CDG with its relevant partitions.

Step 2: Layout and Routing

In this step, we assign program fragments to physical cores by solving an instance of quadratic assignment problem (QAP), stated as follows:

Given a set of facilities F, a set of locations L, a flow function $t : F \times F \to \mathbb{R}$, and a distance function $d : L \times L \to \mathbb{R}$, find the assignment $a : F \to L$ that minimizes the following cost function:

$$\sum_{f_1 \in F, f_2 \in F} t(f_1, f_2) \cdot d(a(f_1), a(f_2))$$

The facilities represent code partitions, the flow is the number of messages between any two partitions, and the distance matrix stores the Manhattan distances between each pair. The solution is a layout that minimizes communication.

This QAP instance can be solved with techniques ranging from Branch and Bound search with pruning [90], to Simulated Annealing (SA) [46], to Ant System [51], to Tabu Search [145]. According to our preliminary experiments, SA takes the least amount of time and generates the best (often optimal) solutions.[2]

We used an existing SA implementation for the layout synthesizer in our compiler. The compiler generates a flow graph f by adding flow units for every `!` operator and conditional statement, and the graph is given to the SA program. The result maps program fragments

[2]On 8×18 grid locations and a random flow graph of 144 facilities, SA took 52 seconds, Ant took 157 seconds, Tabu took 1163 seconds, and Branch and Bound timeout. SA returned the best solution compared to Ant and Tabu.

to physical cores. We use this result to generate a communication path, which is the shortest path between every two program fragments. The layout and routing of the program in Figure 2.1(b) is shown in Figure 2.1(c).

Additional Constraints

Actor partitions. After the compiler maps logical partitions to physical cores, actor partitions become actor cores. This section describes how the routing algorithm works in the presence of actor cores.

Recall our restriction that if an actor partition is associated with an actor function f, the actor partition cannot contain code or data used anywhere outside f. In a later step, the compiler will insert communicate code between communicating cores; therefore, we have to make sure that for every actor core u in an actor function f, the compiler does not insert communication code associated with data sending outside f in u.

To ensure this property, we have to modify the routing algorithm. The basic routing algorithm simply returns any shortest path between the two cores. Since there is no obstacle, finding a shortest path is as simple as moving along the x-axis to the target y-coordinate and then moving along the y-axis to the target x-coordinate. With the new restriction, the routing needs to be able to avoid obstacles. Algorithm 1 displays our routing algorithm. Generally, when routing from core a to b, we need to avoid routing through any actor core. However, if a and b are actors of functions f_a and f_b respectively, then the communication path between a and b can go through other actor cores of both functions (lines 3–8). We use A* search algorithm to find a shortest path between a and b that avoids obstacles (actor cores from the other actor functions) (line 11). After we obtain the path, if a and b are both actors of the same function, we promote all cores along the path as actors of that function if they have never been used before (line 12–13).

Figure 2.2(c) displays the layout and routing of the hand-gesture regconition example in Figure 2.2(a). Assume that the layout synthesizer maps partition xy to physical core (x, y), which represents a coordinate on a 2D grid. When the compiler finds a path for sending the value of the variable `shift` from node (3,2) to (2,1), it avoids routing through actor cores (2,2), (2,3), and (1,3).

Parallel modules, map, and reduce. To ensure parallelism, the routing algorithm enforces that when cores a and b are in the same module instance, a communication path between a and b cannot pass any core in the other module instances of the same module (line 9 and 18–21). Once the algorithm finds a path between a and b, it adds cores along the path as members of a's and b's module instance as well (line 15 and 23–26). We also handle a parallel map/reduce the same way as a parallel module; we treat a function as a module, and a function invocation as a module instance.

Algorithm 1 Communication routing

Global variables: *ActorsMap*, *AllActors*, *UsedCores*

function ROUTE($a, b, modulesInfo$)
  $Allow \leftarrow \{\}$
  $actorFuncA \leftarrow getActorFunc(a)$
  $actorFuncB \leftarrow getActorFunc(b)$
  if $actorFuncA$ **then**
    $Allow \leftarrow Allow \cup ActorsMap[actorFuncA]$
  if $actorFuncB$ **then**
    $Allow \leftarrow Allow \cup ActorsMap[actorFuncB]$
  $Extras \leftarrow otherModuleInstanceCores(modulesInfo, a, b)$
  $Obstacles \leftarrow (AllActors - Allow) \cup Extras$
  $Path \leftarrow A^*search(a, b, Obstacles)$
  if $actorFuncA$ and $actorFuncA = actorFuncB$ **then**
    $ActorsMap[actorFuncA] \leftarrow ActorsMap[actorFuncA] \cup (Path - UsedCores)$
  $UsedCores \leftarrow UsedCores \cup Path$
  $updateModulesInfo(modulesInfo, a, b, Path)$
  return $Path$

function OTHERMODULEINSTANCECORES($modulesInfo, a, b$)
  if a and b in the same module instance I of module M **then**
    return Set of cores of module $M-$ set of cores of module instance I
  return $\emptyset$

function UPDATEMODULESINFO($modulesInfo, a, b, S$)
  if a and b in the same module instance I of module M **then**
    Add S to set of cores of module instance I.
    Add S to set of cores of module M.

Step 3: Code Separation

The program is separated into multiple program fragments communicating through read and write operations. We chose this particular scheme because GA does not support shared memory; cores can only communicate with neighbors using synchronous channels. We preserve the order of operations within each program fragment with respect to their order in the original program to prevent deadlock. The rest of this section describes this process for each language construct.

Pre-Separation Process

The compiler recomputes the CDG of the program. At this step, a partition node in the old CDG becomes a physical core node in the new CDG. We insert additional cores that are only responsible for routing data into the new CDG, for example, cores (3,1) and (1,2) in Figure 2.2(c), which are only used for routing data. We also add an edge connecting `main` node to core (3,1) node in the new CDG because the value of the variable `shift` is sent from core (3,2) to (2,1) through (3,1) in `main`. Similarly, we add an edge connecting `step` to

core (1,2) node because the value of `s[i]` is sent from core (1,1) to (2,2) through (1,2) in the function `step`. With the new CDG, we recompute *relevant cores* (analogous to relevant partitions) for each control flow construct.

Separation Process

Afterwards, we separate the program AST into per-core program fragments. While traversing the program AST in the post-order fashion, we put each piece of data and computation into the assigned core, and add communication code preserving the original order. For example, consider

```
int@3 x = (1 +@2 2)3 *@3 (3 +@1 4)3;
```

Assume partitions 1, 2, and 3 map to cores (0,1), (0,2), and (0,3) arranged from west to east. The result after separation is

```
partition 1:  write(E, 3 + 4);
partition 2:  write(E, 1 + 2); write(E, read(W));
partition 3:  int x = read(W) * read(W);
```

`E` and `W` are the east and west ports. Note the implicit parallelism in this program: $1 + 2$ and $3 + 4$ are executed in parallel.

Arrays. Distributed arrays are stored in multiple cores. For example,

```
int @{[0:16]=0, [16:32]=1} x[32];
for (i from 0 to 32)
  x[i] = x[i] +@place(x[i]) 1;
```

is separated to

```
partition 0:
  int x[16];
  for (i from 0 to 16)
    x[i] = x[i] + 1;
partition 1:
  int x[16];
  for (i from 16 to 32)
    x[i-16] = x[i-16] + 1;
```

Consequently, the program updates the distinct parts of the array in parallel.

Control flow constructs. When we encounter a control flow construct during the AST traversal, we place it in all of its relevant cores. When we encounter an actor function call, we insert a command to send a remote execution request in the requester core to be sent to the master actor core. We set all master actor cores to port execution mode, waiting for requests from the appropriate ports. Figure 2.3 displays some program fragments of the hand-gesture recognition example after the code separation step.

Step 4: Code Generation

This section explains our machine code generation process given single-core programs as inputs, and describes our optimization via a *modular superoptimization algorithm*.

Typically, generation of optimized machine code is carried out using an algorithm that selects instruction sequences and performs local optimization along the way [54]. This type of algorithm is well-suited for applications in which the optimizations are known, and we can determine all of the valid ways to generate code. However, this rewrite-based approach is not easily adapted to our target machine. For example, it is unclear how to design rules sufficient to take advantage of common non-local optimizations using hardware features like the bounded, circular stacks.

We sidestep the problem of rule creation by searching for an optimized program in the space of candidate programs. One such approach is called superoptimization [103, 81, 67, 136]. A superoptimizer searches the space of all instruction sequences and verifies these candidate programs behaviorally against a reference implementation. If an optimized program exists in the candidate space, this approach will find it.

Thus, superoptimization leads to an attractive procedure for generating optimal code for unusual hardware: (1) generate naive code to use as a specification and then (2) synthesize optimal code that matches the specification. Unfortunately, superoptimizers scale to small sequences of instructions [81, 67, 136], which can be smaller than basic blocks in programs, which may contain up to 100 instructions.

We find that it is non-trivial to apply superoptimization in our problem domain for two reasons:

- An obvious way to scale superoptimization is to break down large code sequences (specifications) into smaller ones, superoptimize the small fragments, and then compose the optimal fragments. However, choosing fragment boundaries arbitrarily can cause this approach to miss possible optimizations.

- A straightforward method for specifying the input-output behavior of a program fragment prevents some hardware-specific optimizations. For example, the method may reject a fragment that leaves garbage values on the stack even when it is acceptable to do so.

Therefore, we propose our code generation strategy, as summarized in Figure 2.7. The compiler first produces code without optimizations using a naive code generator, and employs a superoptimizer to generate optimized code. In the next subsection, we explain the naive code generator and the terminology used in the rest of this section. We then explain solutions to the above two problems in the subsequent subsections. Finally, we describe our superoptimizer for program fragments and our approach to encoding the space of candidates as a set of constraints.

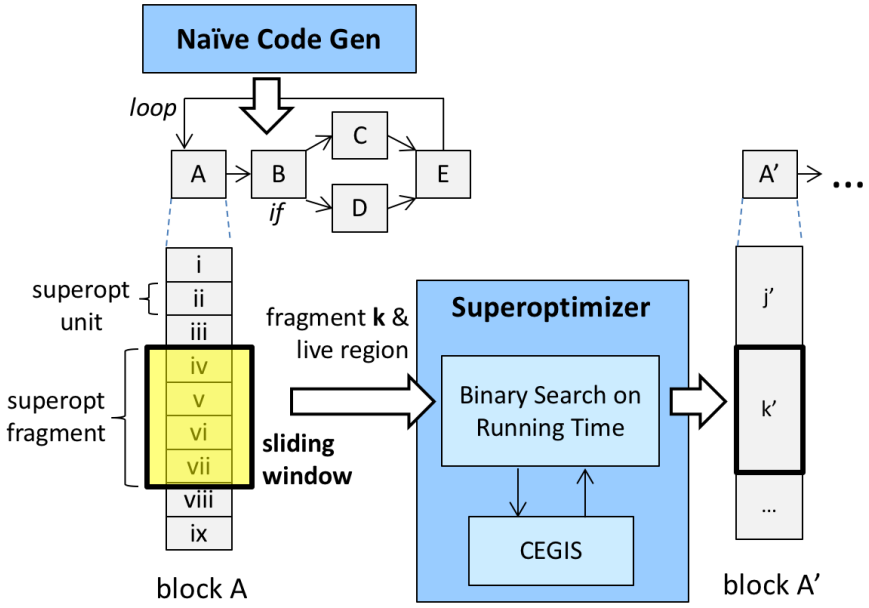


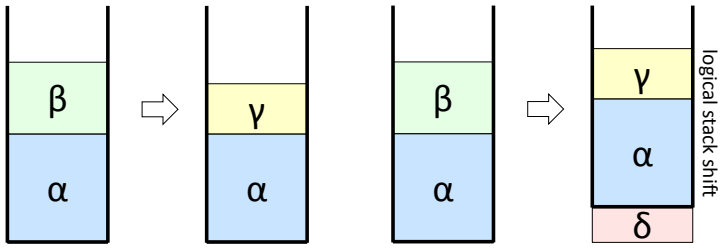
Figure 2.7: Overview of the modular superoptimizer

Naive Code Generation and Terminology

The naive code generator translates each per-core high-level program into machine code that preserves the program's control flow. The straight-line portions of machine code are stored in many small units called *superoptimizable unit*s. A superoptimizable unit corresponds to one operation in the high-level program and thus contains a few instructions. Contiguous superoptimizable units can be merged into a longer sequence called a *superoptimizable fragment*.

We define a state of the machine as a collection of data stack, return stack, memory, and special registers. Each superoptimizable unit contains not only a sequence of instructions but also a *live region* that indicates which parts of the machine's state store live variables at the end of executing the sequence of instructions. The live region of a superoptimizable fragment is simply the live region of the last superoptimizable unit. Currently, a live region always contains the entire memory and usually contains some parts of the return stack and data stack, and some of the registers.

Sequences of instructions P and P' change the state of the machine from S to T and T' respectively. Given a live region L, we define $P \overset{L}{\equiv} P'$ if $Extract(T, L) \equiv Extract(T', L)$, where *Extract* extracts values that reside in the given live region. Since we do not support recursion, it is possible to statically determine the depth of the stack at any point of the program. Since the physical stacks are bounded, our compiler rejects programs that overflow the data stack or return stack at any point.



(a) Basic (b) Improved

Figure 2.8: Specification on data stack

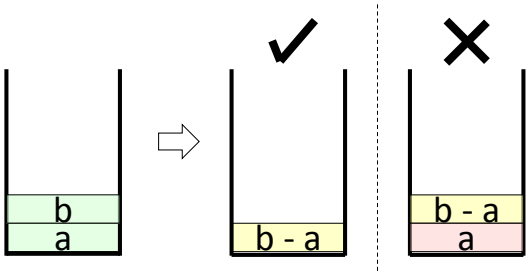


Figure 2.9: Basic specification rejects an instruction sequence that leaves a at the bottom of the stack.

Specifications for Modular Superoptimization

We specify the behavior of a fragment using a sequence of instructions P and its live region L. In this section, we will focus on the constraints on the data stack since it is used for performing every kind of computation and may be used for storing data.

Assume an instruction sequence P changes the data stack from $\alpha|\beta$ to $\alpha|\gamma$ as shown in Figure 2.8(a), and $\alpha|\gamma$ is in the live region. α is a part of the stack that contains intermediate values that will be used later. β is the part of the stack that needs to be removed, and γ is the part of the stack that needs to be added. P' is equivalent to P if P' produces $\alpha|\gamma$, and the stack pointers after executing P and P' are pointing to the same location.

However, this specification is too strict, preventing some optimizations. For instance, consider the example in Figure 2.9 when α is empty, and we want $b - a$ on top of the stack. The shortest sequence of instruction that has this behavior is eight instructions long, with the three final instructions dedicated to removing a remaining garbage value (a in this case) from the stack. It is, in fact, legal to leave a at the bottom of the stack, saving space by eliminating the three instructions. However, this basic specification rejects the shorter sequence because its output data stack is $a|\alpha|b - a$, not $\alpha|b - a$.

We modify the specification, as shown in Figure 2.8(b), such that P' is equivalent to P if it produces $\delta|\alpha|\gamma$ without any constraint on the stack pointer, where δ can be empty. Since GA stacks are circular, leaving garbage items at the bottom of the stack is essentially

shifting the logical stack upward. Note that this specification allows not only upward but also downward logical stack shifts. Thus, this modified specification allows the superoptimizer to discover hardware-specific optimizations that otherwise cannot be discovered when using the straightforward specification.

Sliding Window

Our current tool can superoptimize approximately 16 instructions in a reasonable amount of time. Often, straight-line portions of programs contain more than 16 instructions. Therefore, we have to decompose a long sequence of instructions into smaller ones and run the superoptimizer on the smaller sequences in order to make superoptimization technique scalable. Instead of breaking the long sequence into multiple fixed-length sequences, our *sliding-window* technique adaptively merges superoptimizable units, which usually contain a few instructions, into a *superoptimizable fragment*, which will be given as an input to the superoptimizer. The sliding-window technique makes our modular superoptimization more effective when superoptimization instances timeout or do not find better (more optimal) solutions. More specifically, in these scenarios, instead of entirely skipping the current fixed sequence and working on the next one, the superoptimizer will adjust its window size and attempt to optimize a part of the same sequence (with or without additional instructions) again.

Given a sequence of superoptimizable units called a *unit sequence*, the sliding window technique proceeds as follows.

1. Start with an empty superoptimizable fragment.
2. Append the superoptimizable unit at the head of the unit sequence to the superoptimizable fragment, until the number of instructions is greater than the upper bound.
3. Superoptimize the fragment.
4. If a valid superoptimized fragment is found, append the fragment to the global output, and repeat from 1. If no valid superoptimized fragment is found, append only the first unit to the global output, remove the first unit from the superoptimizable fragment, and repeat from 2. If superoptimization times out, add the last unit from the fragment back to the head of the sequence, and repeat from 3.
5. The process is done when the unit sequence is empty.

Superoptimization and Program Encoding

Given a program fragment and its specification as described in the previous section, our superoptimizer uses counterexample-guided inductive synthesis (CEGIS) to search for an equivalent program fragment [148]. Within the CEGIS loop, we use the Z3 [48] SMT solver to perform the search.

We model the program fragment's approximate execution time based on the cost of each instruction as provided by GreenArrays. We use this cost model to perform a binary search over generated programs looking for optimal performance. Each step involves looking for a program that finishes under a certain time limit by adding that time as a constraint to our SMT formula and synthesizing a program that meets both our performance and our correctness criteria. We can similarly optimize for the length of the program fragment instead of its execution time.

Encoding to SMT formulas. At a particular point of a program, the state of the machine consists of two registers, the data stack, the return stack, memory, and stack pointers. Since each core can communicate with its four neighbors, we represent the data that the core receives and sends using a communication channel, which is an ordered list of (data, neighbor port, read/write) tuples. Hence, the machine's state also includes a communication channel representing the data the core expects to receive or send along with the relevant ports. We use this communication channel to preserve the order of receives and sends to prevent deadlock.

The stacks, the memory, and the communication channel are represented by large bitvectors because Z3 can handle large bitvectors much faster than arrays of integers or arrays of bitvectors. Each instruction in a program converts a machine's state into a new machine's state. We encode each instruction in our SMT formula as a switch statement that alters a machine's state according to which instruction value is chosen.

Address space compression. Address space compression is necessary to scale superoptimization to large problems. Each core in GA144 can store up to 64 18-bit words of data and instructions in memory. The generated code assigns each variable a unique location in memory. An array with 32 entries occupies 32 words of memory. When the formula generator translates programs to formulas, it discards the free memory space and includes just enough memory to contain all variables and arrays; the smaller the memory, the smaller the search space.

Arrays occupy substantial memory space but are usually accessed with a symbolic index during superoptimization. The index is symbolic if it is an expression of one or more variables as it depends on the values of those variables. In light of this observation, we compress the memory of the input program by truncating each array to contain only two entries, and modifying the variable and array addresses throughout the program accordingly. After we get a valid optimal output program, we decompress the output program, and ask the verifier if the decompressed output program is indeed the same as the original input program. Verification is much faster than synthesis, so we can verify programs with a full address space in a reasonable amount of time.

Improvement upon Superoptimizer and Sliding Window

In Chapter 4, we present more advanced algorithms for superoptimization and decomposition in the context of a retargetable superoptimization framework, based on our experience developing a superoptimizer for GA144.

Interactions Between Steps

Since our compilation problem is decomposed into four subproblems, we lose some optimization opportunities, and in some circumstances the compiler produces program fragments that do not fit on cores. We will discuss these issues in this section.

Program Size and Iterative Refinement Method

One goal of our compiler is to partition a high-level program into program fragments such that each fragment can fit in a core. Although the partitioning synthesizer overapproximates the size of each fragment, it still does not consider all communication code. For example, assume that partition A sends some data to partition B. The partitioner increases the sizes of both partitions A and B to reflect the effects of the necessary communication code. However, after the layout step, it is possible that partition A and B are not next to each other. In this case, partition A communicates to partition B via one or more intermediate partitions. Since the partitioner does not have any knowledge about the intermediate nodes, the partitioner does not take into account the space occupied by the communication code associated with the intermediate nodes. As a result, it is possible that the generated program partitions will be too large.

For most programs, our compiler generates final programs that fit in cores. Occasionally, the machine code generated from the compiler does not fit in some cores. When this happens, programmers have to manually pin data and operations to partitions and/or pin partitions to cores to make the code fits. Ideally, the compiler should fix the failed program automatically. One solution is to apply an iterative refinement. The compiler learns from the previously generated code how much communication code may be inserted if it partitions the programs, mapping partitions to cores, and generate communication paths the way it has done before, and then it adjusts its space estimations to be more precise. The iterative refinement reruns the compilation process until all final fragments fit in physical cores.

Optimization Opportunity Loss

There are some lost optimization opportunities that result from decomposing the compilation problem into smaller subproblems. We discuss a few examples of optimization losses in this section.

First, partitioning before optimizing may lead to missed opportunities. For example, let A, B, and C be program fragments that do not fit in one core. Assume the partitioner groups A and B together because that yields the lowest communication count. However, if B and

C are grouped together, the superoptimizer may find a very large execution time reduction such that grouping B and C together yields faster code than grouping A and B does.

Second, our schedule-oblivious routing strategy introduces another potential loss. Assume core A can communicate with core B via either core X or Y, and X is very busy before A sends data to B, while Y is not. The current routing strategy will route data from A to B via either X or Y arbitrarily. However, in this particular case, we should route through Y so that B will receive the data from A more quickly, without having to wait for X to finish its work.

Finally, the scope of superoptimization may prevent some optimizations. We do not optimize across superoptimizable fragments, because we want the compiler to finish in a reasonable amount of time. However, knowing the semantics of the fragments that come before the current fragment could definitely allow the superoptimizer to discover additional optimizations. Increasing the scope to include loops and branches will help even more.

2.5 Toolchain and Debugger

We have seen in practice that having multiple stages of testing is important and productive when programming for a complex hardware. For instance, a FPGA toolchain provides multiple simulations: behavioral simulation, functional simulation, timing simulation, and circuit verification [172]. Therefore, we develop a functional simulator, multicore simulator, and machine simulator for testing at different stages of the compilation.

First, the functional simulator allows programmers to test their algorithms without worrying about any implementation detail such as partitioning and layout. Since the core Chlorophyll language is a subset of C, we can easily generate a C program to be used for the functional simulation.

Second, the multicore simulator allows us to doubly verify that the compiler indeed generates deadlock-free code. After the code separation step, we obtain per-core program fragments. At this stage, we utilize C++ pthread and mutex lock to produce the multicore simulator. Specifically, we create a thread to simulate a core running each program fragment. Each communication channel between two cores is represented by a variable with a lock to simulate blocking reads and writes.

Third, the machine simulator interprets programs at the bit level, as if running on real hardware. It gives programmers many standard debugging methods to help eliminate bugs before running on real hardware, including breakpoints, state examination, and code execution. This has been especially useful when debugging problems with compiler's generated machine code. Other useful features include a support for multiple GA144 chips and a support for building testbeds to simulate I/O devices. When debugging multiple chips, their pins may be virtually wired together, and programs on different chips can be debugged in a single system.

This choice of having multiple simulators allows for simulating program execution at the abstraction level of the problem a programmer is trying to resolve.

2.6 Evaluation

In this section, we present the results of running programs on the GA144 chip to test our hypothesis that using synthesis provides advantages over classical compilation. The CHLOROPHYLL compiler is open-source and available at `github.com/mangpo/chlorophyll`. The toolchain and debugger can be found at `github.com/mschuldt/ga144`.

Hypothesis 1 *The partitioning synthesizer, layout synthesizer, superoptimizer, and sliding windows technique help generate faster programs than alternative techniques.*

We conduct experiments to measure the effectiveness of each component. First, to assess the performance of the partitioning synthesizer, we implement a heuristic partitioner that greedily merges an unknown partition into another known or unknown partition of a sufficiently small size when there is communication between the two. This heuristic partitioning strategy is similar to the merging algorithm used in the instruction partitioner in the space-time scheduler for Raw [92]. Second, to assess the performance of the layout synthesizer, we compare the default layout synthesizer that takes communication counts between partitions into account with the modified version that assumes the communication count between every pair of partitions that communicate is equal to one. Third, we compare the performance of programs generated with and without superoptimization. Last, we compare sliding-window algorithm against fixed-window algorithms, in which the superoptimization windows are fixed.

For each benchmark, five different versions of the program are generated.

1. ***sliding s+p+l***: sliding-window superoptimization, partitioning synthesizer, and layout synthesizer
2. ***fixed s+p+l***: with fixed-window superoptimization, partitioning synthesizer, and layout synthesizer
3. ***ns+p+l***: with no superoptimization, partitioning synthesizer, and layout synthesizer
4. ***ns+hp+l***: with no superoptimization, heuristic partitioner, and layout synthesizer
5. ***ns+hp+il***: with no superoptimization, heuristic partitioner, and imprecise layout synthesizer

We run five benchmarks in this experiment.

- *Prefixsum* sequentially computes the prefixsum of a distributed array spanning 10 cores.
- *SSD* computes the 36-bit sum of squared distance between two distributed 18-bit arrays of size 160, each of which spans four cores. SSDs of different chunks of an array can be computed in parallel since there is no dependency between them.

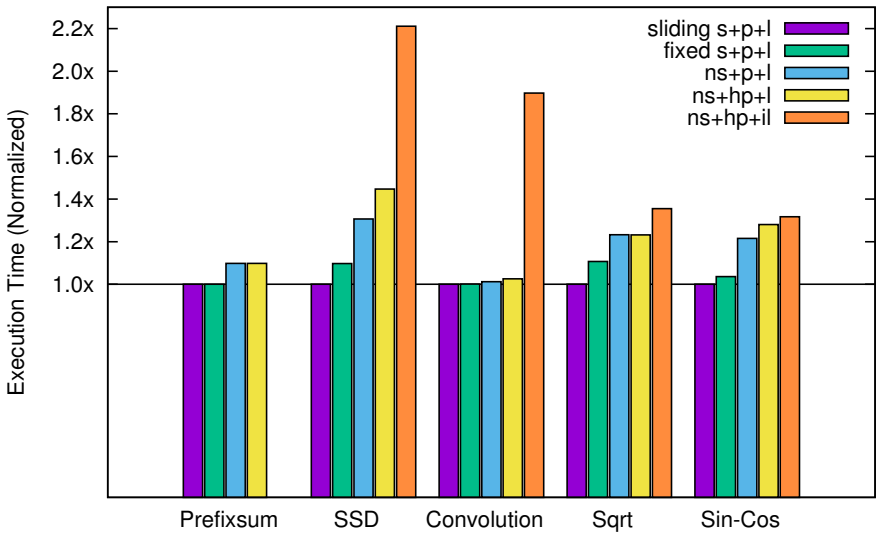


Figure 2.10: Execution time of multicore benchmarks normalized to program generated by the complete synthesizing compiler

- *Convolution* performs 1D convolution on a 4-core distributed array with kernel's width equal to five in parallel. The program first fills in the ghost regions to eliminate loop dependency before the main convolution computation starts.
- *Sqrt* computes the 16-bit square roots of 32-bit inputs.
- *Sin-Cos* computes $cos(x)$ and $sin(x)$.

The execution time result shown in Figure 2.10 confirms our hypothesis. First, comparing *ns+p+l* (third bars) vs. *ns+hp+l* (fourth bars) shows that the partitioning synthesizer offers 5% on average and up to 11% speedup over the heuristic partitioner. Second, comparing *ns+hp+l* (fourth bar) vs. *ns+hp+il* (fifth bar) shows that more precise layout is crucial, providing 1.8x speed up on Convolution. When the layout synthesizer does not take communication count into account, it fails to group the heavily communicating cores next to each other; as a result, the communication paths of different parallel groups share some common cores, preventing those groups from running in parallel. In Prefixsum, the imprecise layout generates program fragments that are too large. Third, comparing *sliding s+p+l* (first bar) vs. *ns+p+l* (third bar) shows that superoptimization gives 15% on average and up to 30% speedup over programs generated without superoptimization. Finally, comparing *sliding s+p+l* (first bar) vs. *fixed s+p+l* (second bar) shows that programs generated with sliding-window superoptimization are 4% on average and up to 11% faster than programs generated with fixed-window strategy.

Hypothesis 2 *The partitioning synthesizer produces smaller programs and is more robust than the heuristic one.*

The previous experiment shows that the partitioning synthesizer does not generate a slower program for any of the five benchmarks. In this experiment, we look at the number of cores the programs occupy, on the same set of benchmarks. In three out of five benchmarks, the synthesizer generates programs that require significantly fewer cores (using 50-72% of the number of cores used by the heuristic).

Another experiment also shows that the heuristic algorithm requires parameter tuning specific to each program, while synthesis does not. The heuristic partitioner does not account for the space occupied by communication code because calculating the size of communication code precisely is complicated in the heuristic one. Therefore, we set the space limit per core by scaling the available space by a factor k, ranging between 0 and 1, in the heuristic partitioner. The higher the scaling factor, the smaller the number of cores it uses. However, the maximum feasible k—while generating code that still fits in cores—for different programs varies ($k = 0.8$ on SSD and $k = 0.4$ on Sqrt). Hence, the synthesizer is more robust than the heuristic.

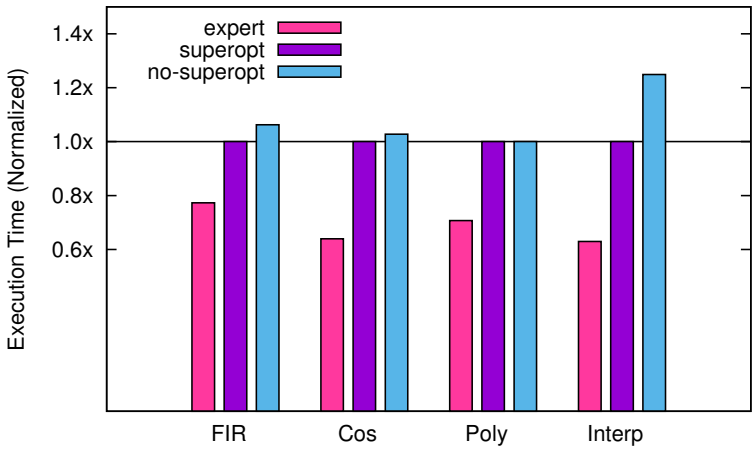
Hypothesis 3 *Programs generated with synthesis are comparable to highly-optimized expert-written programs.*

We compare the execution time and program size of highly-optimized programs written by GA144 developers, programs generated with superoptimization, and programs generated without superoptimization. We have access to the following single-core expert-written programs.

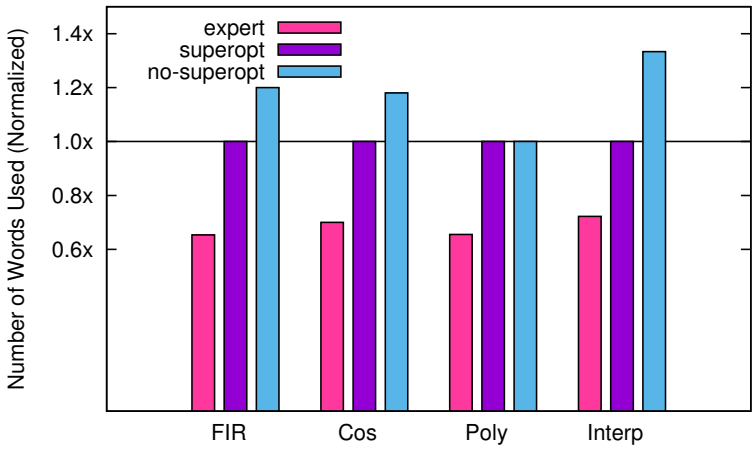
- *FIR* applies 16th-order discrete-time finite impulse response filter on a sequence of samples.
- *Cos* computes cosine.
- *Polynomial* evaluates a polynomial using Horner's method given the coefficients and an input.
- *Interp* performs linear interpolation on input data given a sequence of reference points.

Figure 2.11 shows that our generated programs are 46% slower, 44% less energy-efficient, and 47% longer than the experts' on average, and the superoptimizer improves the running time by 7%, reduces the energy used by 8%, and shortens the program length by 14% compared to no superoptimization on average.

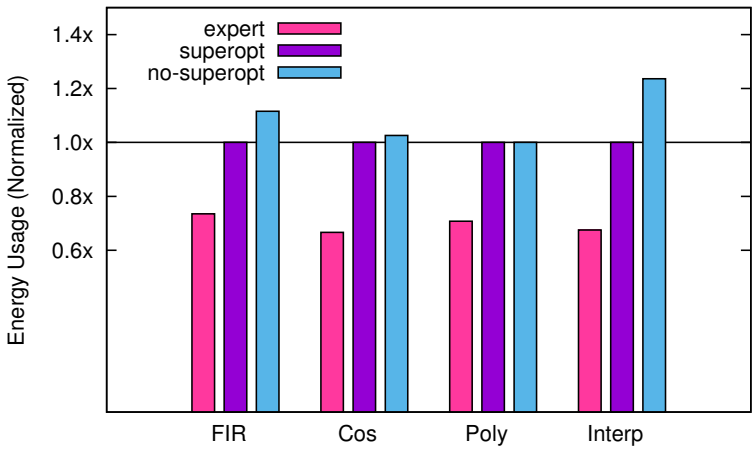
The only multicore application written by experts against which we can compare is MD5 hash function. The other applications published on the GreenArrays website, including SRAM control cluster, programmable DMA channel, and dynamic message routing, require interaction with a GA virtual machine and specific I/O instructions for accessing external memory that Chlorophyll does not support. The MD5 benchmark computes the



(a) Execution time



(b) Space



(c) Energy Usage

Figure 2.11: Single-core benchmarks

hash value of a random message with one million characters. The sequence of characters is streamed into the computing cores while the hash value is being computed.

For the MD5 benchmark, given no partition annotations to the operators, the partitioning synthesizer times out, while the heuristic partitioner fails to produce a program that fits in memory. We have to manually obtain partition annotations with the assistance of the partitioning synthesizer. We first ignore all functions except main. After we solve main, we reintroduce other functions one by one. Finally, we refine the partitioning by examining the machine code and further breaking or combining partitions just by changing the partition annotations. Thus, we can generate code for different partitioning (without superoptimization) in a very short amount of time. Ideally, this iterative, incremental partitioning process should be automated as a part of the compiler. We leave this for future work.

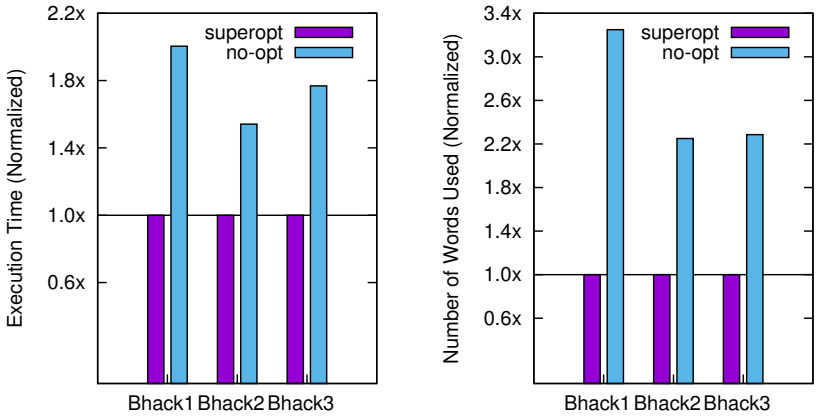
We generate two versions of MD5 program. First, we partition the program such that the generated non-superoptimized code is slightly bigger than memory, but the excess is small enough that the final superoptimized code still fits. We also generate a second version that fits on cores without superoptimization. The generated program with superoptimization is 7% faster and 19% more energy-efficient than the one without superoptimization, and uses 10 fewer cores. Compared to the experts' implementation, it is only 19% slower and 31% less energy-efficient, and it uses two-times more cores. This result confirms that our generated programs are comparable with experts' not only on small programs but also on a real application.

Hypothesis 4 *The superoptimizer can discover optimizations that traditional compilers may not.*

We implement a few small programs taken from the book *Hacker's Delight* [165]: *Bithack 1*, $x - (x\&y)$, *Bithack 2*, $\sim (x - y)$, and *Bithack 3*, $(x \oplus y) \oplus (x\&y)$. Figure 2.12 shows that superoptimization provides 1.8x speedup and 2.6x code length reduction on average. The superoptimizer successfully discovers bit tricks $x\& \sim y$, $\sim x + y$ and $(x\& \sim y) + y$ as the faster implementations for the three benchmarks respectively. Investigating generated programs in many benchmarks, we find that the superoptimizer can discover various strength reductions and clever ways to manipulate data and return stacks. It also automatically performs CSE within program fragments, and exploits special instructions that do not exist in common ISAs. Hence, the superoptimizer can discover an unlimited number of optimizations specific to the machine, while the optimizing compiler can only perform a limited number of optimizations implemented by the compiler developers.

Hypothesis 5 CHLOROPHYLL *increases programmers' productivity and offers the ability to explore different implementations quickly to obtain one with satisfying performance.*

A graduate student spent one summer testing the performance of the GA144 and TI MSP430 micro-controller. He managed to learn arrayForth to program the GA144. However, he was able to implement only two benchmarks: FIR and a simple pedometer application [17]. In contrast, with our compiler, we can implement five different FIR implementations within



(a) Execution time (b) Space

Figure 2.12: Bithack benchmarks

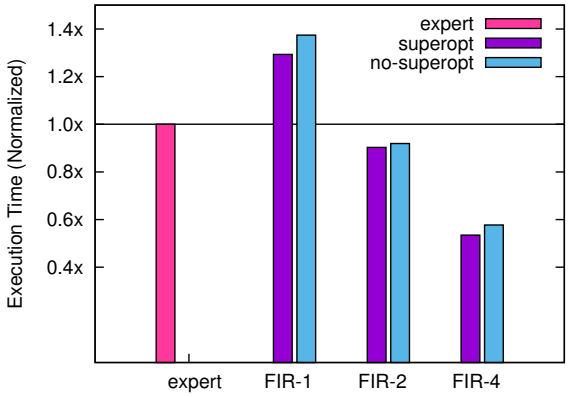


Figure 2.13: FIR benchmark

an afternoon. Figure 2.13 shows the running time of three different implementations of FIR—sequential FIR-1, parallel FIR-2 on two cores, and parallel FIR-4 on four cores—as well as the experts' implementation. Parallel FIR-4 is 1.8x faster than the experts', with the cost of more cores. Hence, programmers can use our tool to productively test different implementations and to exploit parallelism to get the fastest implementation. Although superoptimization makes compilation slower, we can still test implementations quickly by running the non-superoptimized program for a rough estimate of the performance.

Hypothesis 6 *The compiler can be improved by providing more human insights to the synthesizers.*

The GA instruction set does not include division, but expert-written integer division code is provided in ROM, so programmers can conveniently call that function. Hoewver, an even faster division can be implemented when a divisor is known; $x/k = (k_1 * x) >> k_2$ where k_1 and k_2 are magic numbers depending on k. We modify the superoptimizer so that it understands division and accepts the division instruction in an input specification. Then,

Benchmarks	Program Length	Superopt Time (hr)
FIR	90	3.23
Cos	59	2.35
Polynomial	29	1.42
Interp	48	10.01
Bithack 1	13	0.37
Bithack 2	9	4.92
Bithack 3	16	25.08*

Table 2.1: Superoptimization time (in hours) and program length (in words) for single-core benchmarks. A word in program sequences contains either four instructions or a constant literal. *Bithack-3 takes 25.08 hours when the program fragment length is capped at 30 instructions. With the default length (16 instructions), it takes 2.5 hours.

Benchmarks	# of Given Core	Loop Split (s)	Part (s)	Layout (s)	Superopt (hr)
Prefixsum	64	3	36	24	10.78
SSD	64	12	225	24	4.46
Convolution	64	23	122	24	8.39
Sqrt	16	0	566	7	3.60
Sin-Cos	16	2	527	7	6.31
MD5	64	7	N/A	24	16.07

Table 2.2: Compile time of multicore benchmarks. Time is in seconds except for superoptimization time, which is in hours. The compiler runs on an 8-core machine, so it superoptimizes up to eight independent instances in parallel. Layout time only depends on the number of given GA cores. Heuristic partitioning takes less than one second to generate a solution.

we provide this template to the superoptimizer to fill in the numbers for a specific divisor, similar to Sketch [148]. Given the template, the compiler can produce a program that is 6-time faster and 3-time shorter than the experts' general integer division program within three seconds. In theory, the superoptimizer can discover the entire program without the sketch, but it could take much longer time to synthesize since this program is 33-instruction long.

Thus, adding more templates improves performance of generated programs and scalability of the synthesizer. Regarding the performance improvement, this is similar to implementing optimizations for traditional compilers. However, synthesis is in general more powerful because it does not rely on a lookup table and simply discovers faster code by searching.

Tables 2.1 and 2.2 show the compile times for the single-core benchmarks and multicore benchmarks used in our experiments respectively. Partitioning is also slow, but such algorithms are generally slow; consider, for example, partitioning for FPGA [171]. We address the issue by allowing programmers to accelerate the partitioning process by pinning data or code to specific cores when they have relevant insights. The supertopimizer and the parti-

tioning synthesizer in CHLOROPHYLL are the motivation and the basis of a more advanced supertopimizer and partitioning synthesizer described in Chapters 4 and 5 respectively.

2.7 Extensive Case Study

We select an accelerometer-based hand-gesture recognition application as our case study, because this application becomes more and more popular as it provides an intuitive interaction from human to computer. Typically, the classification is not done locally on an embedded device that collects the data, because it is computationally expensive. However, we believe that performing the classification locally is more energy-efficient. We use the gesture recognition algorithm from Wiigee [138]. The main components of the algorithm, displayed in Figure 2.14, are a filter and a gesture classifier, which is composed of a quantizer and a Hidden Markov Model (HMM) classifier. The filter removes acceleration vectors, i.e., (x,y,z) values, that do not significantly deviate from the gravitational acceleration or the previously accepted vector from the incoming stream of acceleration vectors. Vectors that are passed continue to the quantizer, which maps each input vector to a group number. The group number is found by searching for the closest vector to the input vector from the 14 centroid vectors. The set of centroid vectors are created during training using k-mean clustering. Given a group number, the HMM classifier of each gesture updates its belief state. The HMM model is created during training using the Baum-Welch algorithm. After a gesture completes, we obtain the probabilities from the different gesture classifiers, the gesture with the highest probability is the most likely gesture.

Figure 2.15 displays the program layout of our gesture recognition application on GA144. The accelerometer reading, filtering, and the connector of all components are located in the top part of the chip, as shown in Figure 2.15(a). This leaves the entire bottom portion of the chip for the gesture classifiers, but there is only room for two gesture classifiers, given their size. The two gesture classifiers have an identical layout shown in Figure 2.15(b).

Functional Implementation Details

Accelerometer reading. The I2C implementation used for communicating with the accelerometer is based on GreenArrays' SensorTag application documentation [66]. A 32KHz software controlled crystal is used to clock the application to read the accelerometer at 200

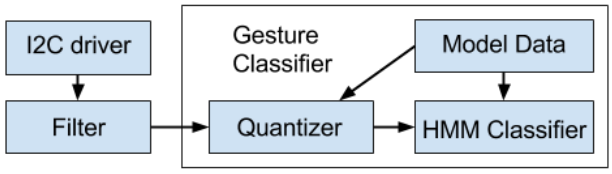
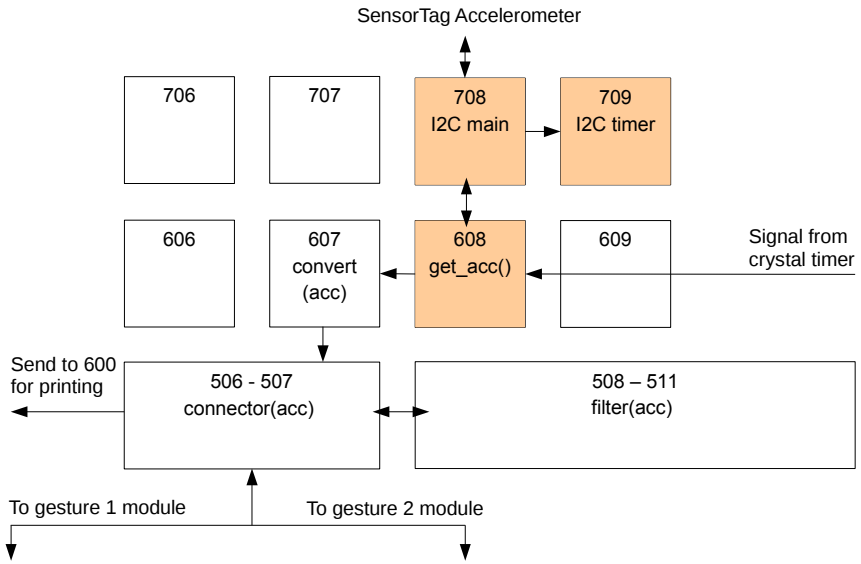
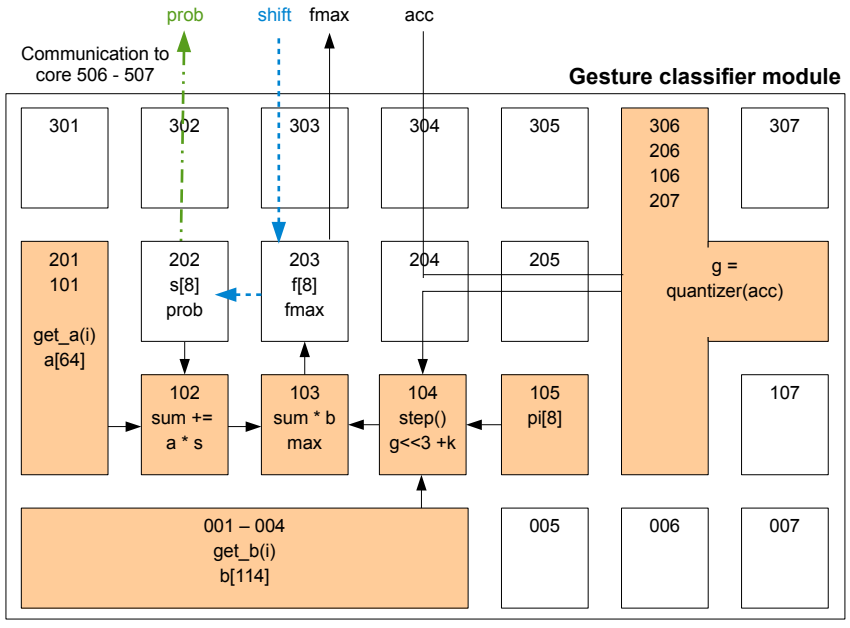


Figure 2.14: Accelerometer-based gesture recognition algorithm



(a) Program layout of accelerometer reading and data filtering



(b) Program layout of one gesture classifier. Black solid and blue dashed arrows denote data flow for updating the belief states `f` and `s` respectively in every classification round. A dotted-dashed green arrow denotes data flow for obtaining a final probability after 1000 rounds.

Figure 2.15: Program layout for the gesture recognition application. Each core is labeled with three digits. The first digit indicates the x-coordinate. The last two digits indicate the y-coordinate. Orange highlights cores that are actors.

Hz. The main I2C node 708 communicates with the accelerometer using node 709 to wait for clock edges. Node 608 passes the raw accelerometer register values to node 607, which converts the raw values into a proper fixed-point format and sends the converted values to the connector nodes 506 and 507.

Gesture classifier. The connector nodes pass the accelerometer data to the filter function and the gesture classifiers. They also gather the final probabilities from the gesture classifiers and send them to node 600 to be printed via a serial port.

There are three types of communication between the connector and a gesture classifier as illustrated with solid, dashed, and dotted-dashed arrows in Figure 2.15(b). The solid arrows depict the data flow for the main computation for each round of accelerometer reading. The main computation derives the group number of an acceleration vector and uses the group number along with the model data `a`, `b`, and `pi` to update the belief states `f` and `s`. Because of the accuracy limitation of an 18-bit fixed point arithmetic, we have to prevent the belief state values from getting too small by shifting the values left by some amount. To maintain correctness, all belief state values in all models must be shifted by the same amount. Thus, the main computation returns the largest value in its belief state `fmax` to the connector. Once the connector collects `fmax` values from all gesture classifiers, it determines the shifting value `shift`. The dashed arrows depicts the data flow for updating the belief state `s` to be equal to `f` shifted left by `shift` bits. After 1,000 rounds, we print out the final probabilities of all gestures and reset the classifiers. The dotted-dashed arrows depict the data flow for obtaining the final probabilities.

Utilizing Language Features

Actorization. Actor functions are used throughout the program, as evidenced in Figure 2.15, which highlights actor cores. We use actors to both reduce communication and code size for I2C cores. The crystal timing node 713 sends a request to node 608 to read accelerometer data. Node 608, in turn, triggers node 708 to start the communication with the accelerometer. Node 708 also sends a request to node 709 to wait for the next clock edge and then sends a signal back. Within a gesture classifier module, we also use actors to reduce code size for cores that store large model data arrays or perform many computations. The use of actors inside the gesture classifier incurs more communication between cores, but it is crucial to make the application fit in a small distributed memory.

Parallel module. To make two gesture classifiers run in parallel, we use the parallel module construct to create two gesture classifiers and place them on different regions of the chip: core 001–307 and core 008–314. Each module instance contains gesture-specific model data for a quantizer and an HMM classifier, which is parameterized during the creation of the module instance.

Programmer-specified program layout. Last, we use partition annotations and partition-core pinning annotations to specify the program layout displayed in Figure 2.15 in order to make the code fits in GA144.

- We pin some data and operations to specific partitions.
- We pin those partitions inside the HMM classifier module to make the layout in each module instance identical.
- We pin each module instance to a specific set of cores.

Experimental Results

In this section, we first evaluate the accuracy of the application running on GA144. Second, we evaluate the impact of being able to compile the application for GA144. Third, we evaluate the impact of some language features.

Classification Accuracy

In this experiment, we verified that the compiled application on GA144 is able to predict hand gestures accurately. We asked two participants to perform circle and flip-roll gestures, 11 times for each gesture. The prediction accuracies for the two participants were 90.91% and 80.82%. We obtained a similar prediction accuracy to Wiigee's (the original implementation that our application was based on), which ranges from 84% to 94% [138]. The demonstration of the application running on GA144 can be viewed at: `https://youtu.be/GD9lVm1ZyNQ`

GA144 vs. MSP430

Next, we evaluated the impact of being able to run the application on GA144. If running the application on other processors that do not require difficult programming partitioning and a careful program layout design were as good as running the application on GA144, we would not have to bother developing the compiler extensions we have introduced. For this purpose, we selected MSP430, a widely-used ultra low-power micro-controller to compare GA144 against.

Implementation for MSP430. We implemented the same application for MSP430F5529 with 128-kB flash, 8-kB RAM, and up to 25 MHz CPU speed. We used 16 bits to represent a fixed-point number instead of 18 bits as implemented on GA144. We interfaced the ADXL345 accelerometer to MSP430 via I2C protocol. Note that we used the SensorTag accelerometer for GA144. Although the accelerometers were different, we implemented the same I2C protocol on both GA144 and MSP430. Therefore, the two processors performed exactly the same activities to communicate with the accelerometers. The accelerometers were powered by a different energy source from the one that powered the processors, and we excluded the

Processor	Execution time per round		Energy consumption per round	
	absolute (ms)	relative to GA	absolute (μJ)	relative to GA
GA144	2.639	-	2.231	-
MSP430	61.346	23.2x	41.920	18.8x

Table 2.3: Total execution time and energy consumption per one round of classification

Processor	Accelerometer Reading				Filter & Classification			
	power (mW)	time (ms)	energy (μJ)	energy relative to GA	power (mW)	time (ms)	energy (μJ)	energy relative to GA
GA144	0.633	2.610	1.652	-	19.957	0.029	0.579	-
MSP430	0.565	1.346	0.760	0.46x	0.686	60.00	41.160	71.1x

Table 2.4: Energy consumption per each task per one round of classification

energy consumed by the accelerometers when comparing GA144 and MSP430. Therefore, we believe that our comparison was fair.

Results. We measured the energy consumption for one round of classification. In each round, the application read an (x,y,z) acceleration vector and updated the belief states of the two gesture classifiers if the input vector passed the filter. Reading the accelerometer required many I/O activities; whereas, filtering accelerometer values and updating the belief states required a lot of computations. Thus, we measured the energy consumptions of the two tasks separately to compare the performance of GA144 and MSP430 for the different types of usages. We powered GA144 with 1.8 V and MSP430 with 2.2 V, as these are the typical voltages. We powered the accelerometer from a different power source because we are only interested in energy consumption by the processors. We ran each task in a loop hundred to hundreds-of-thousand times to measure the average current drawn (using a multimeter with a microamp precision) and completion time by each processor.

Table 2.3 reports completion time and energy consumption of running one round of classification on GA144 and MSP430. Overall, GA144 was 18.8x more energy-efficient and 23.2x faster than MSP430. If we look at each task separately in Table 2.4, GA144 was excellent at performing a computationally heavy task: filtering and classification. It was three orders of magnitude faster and 71.1x more energy-efficient than MSP430. Recall that the application should run 200 rounds of classification in one second; each round takes 5 milliseconds. However, MSP430 took 60 milliseconds to update the belief states, if the accelerometer values passed the filter. Therefore, MSP430 was not able to run 200 rounds in one second consistently like GA.

On the other hand, MSP430 was better at the accelerometer reading task, 2.2x more energy-efficient than GA144. However, we believe that we can further optimize GA144 for this task. In our implementation, the main I2C core, which interacts with the accelerometer, waits for its I/O pin to become high by spinning in a loop. Therefore, the program can be optimized by avoiding this loop. Unfortunately, the main I2C core is completely full, and

Features used	Number of cores	Overflowed cores	Biggest core (words)	Total words used
Actor + layout	82	0	64	2,609
No actor + layout	90	12	87	3,152
No actor + no layout	82	20	89	3,071

Table 2.5: Size of generated code. Each core can store up to 64 words of data and program.

we need more space for this optimization. To make the optimized code fit in this core, the compiler will need advanced transformations that exploit transferring code (not just data) between cores.

Impacts of Actorization, Layout Pinning, and Parallel Module

Without actorization and layout pinning features, we would not be able to run the application on GA144 because the code could not fit in its memory. Table 2.5 shows the impact of the new extensions on the sizes of generated programs. 'Actor' indicates actorizing some functions (yielding hybrid partitioning strategy), and 'layout' indicates specifying program layout. When we actorized program appropriately and specified the program layout design ('actor + layout'), the application fit on every core. However, when all functions were actors, the compiler failed to generate code because it could not find a feasible routing between every communicating pair of cores. Recall that actors impose additional constraints to the routing algorithm; the more actors, the more obstacles the routing algorithm has to avoid. When we did not actorize any function but specified the program layout ('no actor + layout'), the compiler successfully generated code, but 12 cores overflowed, and the total number of cores used was 90 instead 82. When we did not actorize any function and did not specify the program layout ('no actor + no layout'), 20 cores overflowed, and the biggest core overflowed by 25 words. When we actorized some functions but did not specify the program layout, the compiler failed to find feasible routing between some cores. We excluded the failed versions from the table. In summary, these results reveal that both hybrid partitioning strategy and programmer-specified layout are crucial for compiling code for a very limited-resource environment.

Furthermore, without the parallel module construct, we would have to duplicate the classification code and explicitly assign every data and computation to a partition to achieve parallelism. Hence, the parallel module construct tremendously increased our productivity.

2.8 Related Work

Programming Models

A number of programming models have been developed for spatial architectures for different application domains. StreamIt, a programming model for streaming applications, decom-

poses the compilation problem much as we do [61]. Partitions are defined by programmers using *filters* and can be merged by the compiler. GA144 also shares many characteristics with systolic arrays. Systolic arrays are designed for massively parallel applications such as applications with rhythmic communications [79]. Thus, the programming model for systolic arrays is domain-specific, tailored to such applications [88, 74]. Unlike StreamIt or Systolic, CHLOROPHYLL targets more general-purpose programming.

The high-performance computing community has developed programming models to support programming on distributed memory. The SPMD program partitioning strategy was proposed by Callahan and Kennedy [34]. They pointed out that the partitioned program can be described as SPMD because in the most naively compiled code, every node executes the same program but performs computation on distinct data items. Many Distributed Fortran compilers apply this partitioning strategy with an *owner computes rule* to partition programs such that computations happen at the same place where the left-hand-side data element lives [28, 107].

The actor program partition strategy is similar to the strategy the X10 compiler uses for handling place changes when programmers use the construct `at` to specify where the data and the computations inside the scope of `at` live and happen [153]. However, our language construct for actorization is very different from the X10 construct. Since GA144 cores are very small, multiple cores may be required to perform one functional task. Therefore, we provide the construct that is suitable for actorizing a task performed by multiple cores. In contrast, X10 targets much bigger nodes, so a task can normally fit in one node. Thus, the `at` construct seems appropriate for X10's use cases. We borrow the name *actor* and its concept of reacting upon a request to perform a task from the actor model for a concurrent computation [71]. However, we use the actor concept to avoid duplicating control flow constructs instead of obtaining concurrency.

Our memory model is Partitioned Global Address Space (PGAS), a model used many languages [35, 173, 112, 134]. Although these languages offer programmers control over mapping operators to computing resources, they do not provide the programmers an easy way of exploring different mappings.

Type Systems

Many distributed programming languages have exploited type systems to ensure properties of interest. Delaval et al. presented a type system for the automatic distribution of high-order synchronous dataflow programs, allowing programmers to localize some expressions onto processors [49]. The type system can infer the localization of non-annotated values to ensure the consistency of the distribution. Like our compiler, the framework generates local programs to be executed by each computing resource from a centralized typed program. X10 introduces *place type* and exploits type inference to eliminate dynamic references of global pointers [38]. Titanium, similarly, uses type inference to minimize the number of global pointers in the program [98].

Heuristic-based Compilers

There is substantial work on heuristic-based compilers for spatial architectures. The partitioning and placement algorithms used in TRIPS compiler, Raw space-time scheduler, and Occam to transputer system, may be applied with some modifications to our problem. However, these architectures are substantially different from GA144.

TRIPS compiler distributes a computation DAG of up to 128 instructions in each hyperblock onto 16 cores [33, 146]. Chlorophyll partitions much larger programs — MD5, for example has, 4,600 instructions — with loops and branches onto 144 cores. TRIPS also has hardware-supported routing, while GA144 does not. In Raw compiler, the space-time scheduler decomposes the partitioning problem into three subproblems: clustering, merging, and global data partitioning [92], while Chlorophyll solves the partitioning problem as one problem. The merging algorithm is essentially the same as the heuristic partitioner with which we compare Chlorophyll in our evaluation. The transputer compiler and StreamIt's Raw compiler also use SA for solving the layout problem [140, 61].

Constraint-based Compilers

Though not as common as heuristic-based compilers, constraint-based compilers have been studied and used in practice.

Vivado Design Suite performs High-Level Synthesis that transforms a C, C++ or SystemC design specification into a RTL implementation, which in turn can be synthesized onto a FPGA [171]. The programmer can specify additional constraints using directives, such as controlling the binding process of operations to cores, albeit in ways that are much more limited than our programming model facilitates. For example, multiplication is implemented by a specific hardware multiplier in the RTL design using a specific core.

Yuan et al. solve hardware/software partitioning and pipelined scheduling on runtime reconfigurable FPGAs using an SMT solver [174]. Although the problem domains of our compiler and of their partitioner and scheduler are different, Yuan et al. also shows that solutions obtained from the SMT solver are superior to the solutions obtained from a heuristic algorithm, but that constraint solving techniques face scalability challenges.

Another constraint-based approach to solve the placement and routing problems uses Integer Linear Programming (ILP) to map the computation DAG to the graph representing the hardware's structure [114]. Its constraints capture placement of computation, data routing, and resource utilization. However, this technique cannot be applied to our partitioning and layout problems because it assumes a simple program control flow with no loops, as it targets scheduling problems at a finer granularity. Consequently, it does not address the problem of partitioning control statements.

Superoptimization and Program Synthesis

I will discuss more about superoptimization and its related work in Chapter 4.

2.9 Conclusion

Building efficient optimizing compilers is difficult, even for traditional architectures that are designed for programmability. With radically stripped down and evolving target architectures such as GA144, the classical compilation approach becomes even more difficult and less practical to implement.

We have built the first synthesis-aided compiler for extremely minimalist architectures and introduced a new spatial programming model to provide programmability for programmer-unfriendly hardware. We introduced language constructs to make complex applications run on a very small distributed-memory multicore processor and to allow programmers to express parallelism. First, we let programmers express their insights on how to partition their programs using partition annotations. Second, we compared actor and SPMD partitioning strategies and designed the Chlorophyll language to allow programmers to control when to use which partitioning strategies. Third, we allowed programmers to express parallelism and program layout. Our compiler decomposed the compilation problem into smaller subproblems that can be solved by various synthesizers or easy-to-implement transformations. Although program synthesis may not scale to large problems on its own, our work showed that we can overcome these issues by decomposing problems into smaller ones and relying on more human insight.

The contribution of this chapter is not that our algorithms for partitioning, layout, routing, and code generation are individually superior to the existing ones, but we show that our compiler is simpler than a classical compiler while producing comparable code. Program synthesis techniques enable compiler developers to quickly develop a new high-performance compiler for a radical architecture without knowing how to implement optimizations specific to the architecture.

Chapter 3

Floem: Programming NIC-Accelerated Applications

Network bandwidth is growing much faster than CPU performance [5], forcing many data-center applications to sacrifice application cycles for packet processing [24, 85, 118]. As a result, system developers have started to offload computation to programmable network interface controllers (NICs), dramatically improving the performance and energy efficiency of many data-center applications, such as search engines, key-value stores, real-time data analytics, and intrusion detection [36, 85, 94, 127]. These NICs have a variety of hardware architectures including FPGAs [36, 105, 177], specialized flow engines [6], and more general-purpose network processors [3, 104].

However, implementing data-center network applications in a combined CPU-NIC environment is difficult. It often requires many design-implement-test iterations before the accelerated application can outperform its CPU-only version. These iterations involve non-trivial changes: programmers may have to move portions of application code across the CPU-NIC boundary and manually refactor the program.

3.1 Contributions

We propose Floem, a programming system for NIC-accelerated applications. Our current prototype targets a platform with the Cavium LiquidIO [3], a general-purpose programmable NIC that executes C code. Floem is based on a data-flow language that is natural for expressing packet processing logic and mapping *elements* (modular program components) onto hardware devices. The language lets developers easily move an element onto a CPU or a NIC to explore alternative offloading designs, as well as parallelize program components. Application developers can define a Floem element as a Python class that contains a C

Materials in this chapter are based on work published asPhothilimthana et al., "Floem: A Programming System for NIC-Accelerated Network Applications," in proceedings of OSDI 2018 [122].

implementation of the element. To aid programming productivity, we provide a library of common elements.

Further examining how developers offload data-center applications to NICs, we have identified the following commonly encountered problems, which led us to propose abstractions and mechanisms amenable to a data-flow programming model that can solve these problems.

- Different offloading choices require different communication strategies. We observe that these strategies can be expressed by a **mapping of logical communication queues to physical queues**, so we propose this mapping as a part of our language.
- Moving computation across the CPU-NIC boundary may change which parts of a packet must be sent across the boundary. Marshaling the necessary packet fields is tedious and error-prone. Thus, we propose **per-packet state** — an abstraction that allows a packet and its metadata to be accessed anywhere in the program — while FLOEM automatically transfers only required packet parts between a NIC and CPU.
- Using an in-network processor to cache application state or computation is a common pattern for accelerating data-center applications. However, it is non-trivial to implement a cache that guarantees the consistency of data between a CPU and NIC. We propose a **caching construct** for memoizing a program region, relieving programmers from having to implement a complete cache protocol.
- Developers often want to **offload an existing application** without rewriting the code into a new language. We let programmers embed C code in elements and allow a legacy application to interact with FLOEM elements via a simple function call, executing those elements in the host process of the legacy application.

We demonstrate that without significant programming effort, FLOEM can help offload parts of real-world applications — a key-value store and a real-time analytics system — improving their throughput by 1.3–3.6× and 75–96%, respectively, over a CPU-only configuration.

In summary, this chaper makes the following contributions:

- Identifying *challenges* in designing of NIC-accelerated data-center applications (Section 3.2)
- Introducing *programming abstractions* to address these challenges (Sections 3.3 and 3.4)
- Developing a programming system that enables exploration of alternative offloading designs, including a *compiler* (Section 3.5) and a *runtime* (Section 3.6) for efficient data transfer between a CPU and NIC

3.2 Design Goals and Rationale

We design FLOEM to help programmers explore how to offload their server network applications to a NIC. The applications that benefit from FLOEM have computations that *may* be more efficient to run on the NIC than on the CPU because of the NIC's hardware-accelerated functions, parallelism, or reduced latency when eliminating the CPU from fast-path processing. These computations include packet filtering (e.g., format validation and classification), packet transformation (e.g., serialization, compression, and encryption), packet steering (e.g., load balancing to CPU cores), packet generation, and caching of application state. This list is not exhaustive. Ultimately, we would like FLOEM to help developers discover new ways to accelerate their applications.

The main challenge when designing programming abstractions is to realize a small number of constructs that let programmers express a large variety of implementation choices. This requires an understanding of common challenges within the application domain. We build FLOEM to meet the following design goals.

Goal 1: Expressing Packet Processing

As described above, computations suitable for NIC offloading are largely packet processing. Programming abstractions and systems for packet processing have long been studied, and the Click modular router [111] is widely used for this task. We adopt its data-flow model to ease the development of packet processing logic (Section 3.3).

Goal 2: Exploring Offload Designs

A data-flow model is also suitable for mapping computations to desired hardware devices, as we have seen with many Click extensions that support offloading [86, 95, 152]. Similarly, FLOEM programmers implement functionality once, as a data-flow program, after which they can use code annotations to assign elements to desired devices and to parallelize the program. However, trivially adopting a data-flow model is insufficient to meet this design goal. By inspecting the design of a key-value store and a TCP stack offloaded with FlexNIC [85], we discover several challenges that shape the design of our language.

Logical-to-physical queue mapping. One major part of designing an offloading strategy is managing the transfer of data between the host and accelerator. Various offloading strategies require different communication strategies, such as how to steer packets, how to share communication resources among different types of messages, and whether to impose an order of messages over a communication channel. We want to implement these strategies with only a small number of high-level constructs.

By examining hand-optimized offloads, we find that developers typically express communication in terms of logical queues and then manually implement them using the provided

hardware communication mechanisms. A logical queue handles messages sent from one element to another, while a hardware communication channel implements one physical queue. As part of an offload implementation, developers have to make various mapping choices among logical and physical queues. The right mapping depends on the workload and hardware configuration and is typically realized via trial-and-error.

To aid this task, we design a queue construct with an explicit logical-to-physical queue mapping that can be controlled via parameters and by changing element connections. Existing frameworks [86, 95, 152] do not support mapping logical to physical queues. To control the number of physical queues in these frameworks, programmers have to explicitly: (1) create more logical queues by demultiplexing the flow into multiple branches and making more elements and connections, or (2) merge logical queues by multiplexing multiple branches into one.

Per-packet state. In a well-optimized program, developers meticulously construct a message by copying only the necessary parts of a packet to send between a CPU and NIC; this minimizes the amount of data transferred over PCIe. When developers move computation between the CPU and NIC, they may need to rethink which fields must be sent, slowing the exploration of alternative offloading designs.

Nevertheless, no existing system performs this optimization automatically. ClickNP [95] sends an entire packet, while NBA [86] and Snap [152] rely on developers to annotate each element with a packet's *region of interest*, specified as numeric offsets in a packet buffer. We design Floem to automatically infer what data to send across the CPU-accelerator boundary and offer the *per-packet state* abstraction as if an entire packet could be accessed anywhere in the program. This abstraction resembles P4's per-packet metadata [26] and remote procedure call interface description languages (RPC IDLs, e.g., XDR [44] and Google's protobuf [60]). However, P4 allows per-packet metadata to be carried across multiple processing pipelines only within a single device, while RPC IDLs generate marshaling code based on interface descriptions, rather than automatically inferring what to send.

Caching construct. Caching application state or memoizing computation in an in-network processor is a common strategy to accelerate server applications [47, 78, 94, 100]. While the abstractions we have so far are sufficient to express this offloading strategy, implementing a cache protocol still requires a significant effort to guarantee both data consistency and high performance when messages between a CPU and NIC may arrive out-of-order. Thus, we introduce a *caching construct*, a general abstraction for caching that integrates well with the data-flow model. This construct provides a full cache protocol that maintains data consistency between the CPU and NIC. Unlike Floem, existing systems support caching only of flow state [6, 95] — which typically does not require maintaining consistency between the CPU and NIC — but not caching of application state.

Goal 3: Integrating with Existing Applications

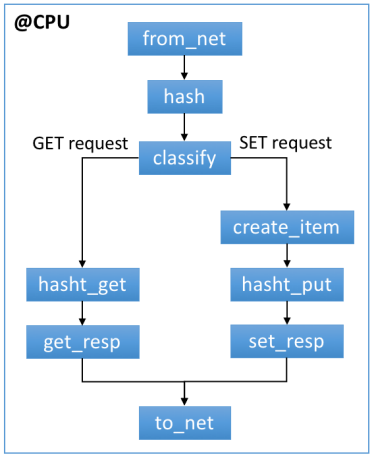
Prior frameworks were designed exclusively to implement network functions and packet processing [40, 50, 86, 95, 111, 117, 152], where computation is mostly stateless and simpler than in our target domain of server applications. While parts of typical server applications can be built by composing pre-defined elements, many parts cannot. In our target domain, developers often want to offload an application by reusing existing application code instead of writing code from scratch. Besides porting existing applications, some developers may prefer to implement most of their applications in C because a data-flow programming model may not be ideal for the full implementation of complex applications.

Therefore, FLOEM lets developers combine custom and stock elements, embed C code in data-flow elements, and integrate a FLOEM program with an external program. As a result, developers can port only program parts that may benefit from offloading into the data-flow model. The impedance mismatch between the data-flow model and the external program's model (e.g., event-driven or imperative) raises the issue of interoperability. Our solution builds on the queue construct to decouple the internal part from the interface part, which appears to the external program as a function. The external program can execute the function using its own thread to (1) retrieve a message from the queue and process it through elements in the interface part, or (2) process a message through the interface part and push it to the queue.

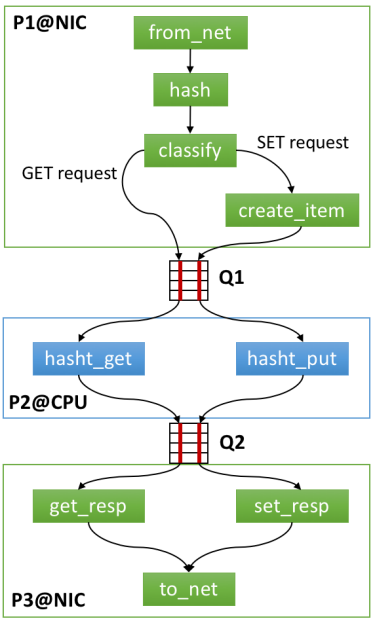
3.3 Core Abstractions

This section explains the core FLOEM programming abstractions. A more detailed language definition can be found at `github.com/mangpo/floem`. We use a key-value store application as our running example. Figure 3.1 displays several offloading designs for the applicaton: CPU-only (Figure 3.1(a)), split CPU-NIC (Figure 3.1(b)), and NIC as cache (Figure 3.1(c)). Figure 3.1(d) illustrates how to create an interface that an external program can use to interact with FLOEM. We show how to implement these offloads using our programming abstractions in this and the next sections.

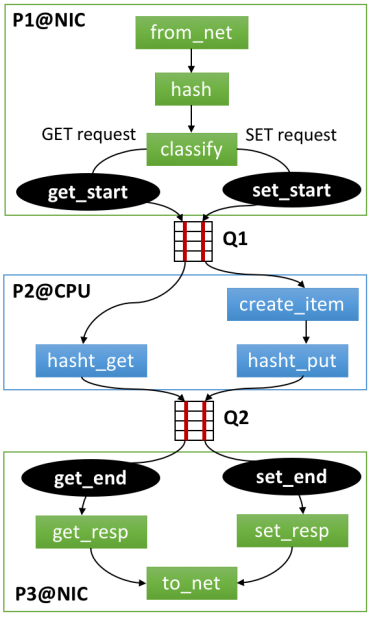
Elements. FLOEM is an asynchronous data-flow programming model. An *element* is a function that is invoked when tokens from all input ports are ready (have arrived). When an element is invoked, it will consume input tokens from all input ports and produce output tokens to output ports. An element e have O^e output ports where $O^e \geq 0$, and an output port O^e_i can deliver OT^e_i tokens where $OT^e_i \geq 0$. An element fires an output port O^e_i with all OT^e_i tokens at the same time. An output port of element e becomes an input port of another element e', to which e connects. One invocation of a basic element will fire all its output ports, while one invocation of a *switch* element will fire zero or one of its output ports. A switch element can be used to dynamically drop packets, so it is one source of asyncrony in the programming model.



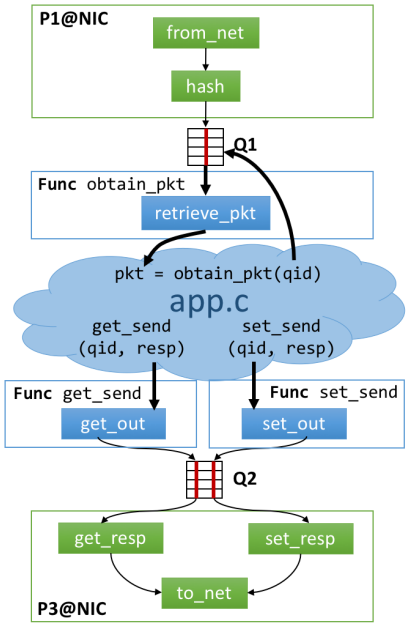
(a) No offloading



(b) Splitting work



(c) NIC as cache



(d) Interface to external program

Figure 3.1: Several offloading strategies of a key-value store implemented in FLOEM

Programmers define a port of an element with a list of tokens' types the port accepts. A token can be a primitive value, struct, or a pointer. An element can pass a pointer to another element if they are on the same process and device. Otherwise, the compiler returns an error.

The listing below illustrates how to create the `classify` element in our key-value store example, which classifies incoming requests by type (GET or SET).

```
class Classify(Element): # Define an element class
  def configure(self):
    self.inp = Input(pointer(kvs_message))
    self.get = Output(pointer(kvs_message))
    self.set = Output(pointer(kvs_message))

  def impl(self):
    self.run_c(r'''   // C code
      kvs_message *p = inp();
      uint8_t cmd = p->mcr.request.opcode;

      output switch { // switch --> emit one output port
        case (cmd == PROTOCOL_BINARY_CMD_GET): get(p);
        case (cmd == PROTOCOL_BINARY_CMD_SET): set(p);
      }
    ''')
classify = Classify()  # Instantiate an element
```

We specify input and output ports in the `configure` method. We express the logic for processing a single packet in the `impl` method by calling `run_c`, which accepts C code with special syntax to retrieve value(s) from an input port and emit value(s) to an output port.

To create the program shown in Figure 3.1(a), we connect elements as follows:

```
from_net >> hash >> classify
classify.get >> hasht_get >> get_resp >> to_net
classify.set >> item >> hasht_put >> set_resp >> to_net
```

Note that `.get` and `.set` refer to the output ports of `classify`.

FLOEM accepts only a data-flow program that guarantees that when a processing of one packet is complete, there is no unconsumed token remaining at any element's input port; a token will remain at an input port of an element, if the element has multiple input ports, and not all input ports are ready.

Queues. Instead of pushing data to the next element instantaneously, a queue can store data until the next element dequeues it. A queue can connect and send data between elements on both different devices (e.g., CPU and NIC) and on the same device. For example, in Figure 3.1(b), queue `Q1` sends packets from elements on the NIC to elements on the CPU as packets are received.

Another source of asyncrony comes from a batching queue, which can be thought of as a queue into which we insert a single token, then another, and so on, but the other side of the queue outputs the tokens only when a batch is filled; a number of tokens in a batch cannot be determined statically.

Shared states. FLOEM provides a shared state abstraction that lets multiple elements share a set of variables that are persistent across packets. For example, elements `hasht_get` and `hasht_put` share the same state containing a hash table. FLOEM normally prohibits elements on different devices from sharing the same state. Instead, programmers must use message passing across queues to share information between those elements. Shared state lets programmers express complex stateful applications. An element that does not access a shared state is functional.

Segment and execution model. A *segment* is a set of connected elements that begins with from a *source* element, which is either a `from_net` element or a queue, and ends with *leaf* elements (elements with no output ports) or queues. A queue sends packets between segments. Our execution model is run-to-completion within a segment. A source element processes a packet and pushes it to subsequent elements until the packet reaches the end of the segment. When the entire segment finishes processing a packet, it starts on the next one. A segment can have one or more instances. By default, each segment has only one instance on a CPU, so elements within a segment run sequentially with respect to their data-flow dependencies. If a segment has more then one instances, then these instances execute concurrently without being synchronized in anyway. Users are responsible to insert synchronization (e.g., using queues or locks) to ensure the correctness of the program.

FLOEM assigns a thread to execute one instance of a segment, and each thread processes a packet to completion within a segment; in the other word, a packet is processed solely by one thread in a segment. However, an execution in a segment may be blocked by a lock or waiting for an empty slot in a queue.

The program in Figure 3.1(a) has a single segment, while the program in Figure 3.1(b) has three. Note that not all elements in a segment must be executed for each packet. In our example, either `hasht_get` or `hasht_put` (not both) will be executed depending on the port where `classify` pushes a packet to.

Offloading and parallelizing. A segment is a unit of code migration and parallelization. Programmers map each segment to a specific device by supplying the `device` parameter. They can also assign multiple threads to run the same segment to process different packets in parallel using the `cores` parameter. Programmers cannot assign a segment to run on both the NIC and CPU in parallel; the current workaround is to create two identical segments, one for NIC and another for CPU. Figure 3.2 displays a FLOEM program that implements a sharded key-value store with the offloading strategy in Figure 3.1(b).

3.4 Advanced Abstractions

This section presents programming abstractions that we propose to mitigate recurring programming challenges encounters when exploring different ways to offload applications to a NIC.

```
1  Q1 = Queue(channel=2, inst=3)
2  Q2 = Queue(channel=2, inst=3)
3
4  class P1(Segment):
5    def impl(self):
6      from_net >> hash >> queue_id >> classify
7      classify.get >> Q1.enq[0]  # virtual channel 0
8      classify.set >> create_item >> Q1.enq[1] # virtual channel 1
9
10 class P2(Segment):
11   def impl(self):
12     self.core_id >> Q1.qid     # use core id as queue id
13     Q1.deq[0] >> hasht_get >> Q2.enq[0]
14     Q1.deq[1] >> hasht_put >> Q2.enq[1]
15
16 class P3(Segment):
17   def impl(self):
18     scheduler >> Q2.qid        # scheduler produces queue id
19     Q2.deq[0] >> get_resp >> to_net
20     Q2.deq[1] >> set_resp >> to_net
21
22 P1(device=NIC, cores=[0,1])    # run on core id 0,1
23 P2(device=CPU, cores=[0,1,2])
24 P3(device=NIC, cores=[2,3])
```

Figure 3.2: FLOEM program implementing a sharded key-value store with the CPU-NIC split strategy of Figure 3.1(b)

Logical-to-Physical Queue Mapping

To achieve correctness and maximize performance, FLOEM gives programmers control over how the compiler instantiates logical queues for a particular offloading strategy. The queue construct `Queue(channel=n, inst=m)` represents n logical queues (n channels) using m physical queues (m instances). For example, `Q1` on line 1 of Figure 3.2 represents two logical queues — displayed as red channels in Figure 3.1(b) — using three physical queues. Different mappings of logical to physical queues lead to different communication strategies, as elaborated below.

Packet steering. Developers can easily implement packet steering by creating a queue with multiple physical instances. For example, in the split CPU-NIC version of the key-value store (Figure 3.1(b)), we want to shard the key-value store so that different CPU threads can handle different subsets of keys to avoid lock contention and CPU cache misses. As a result, we want to represent queue `Q1` by multiple physical queues, with each CPU thread having a dedicated physical queue to handle requests for its shard. The NIC then steers a packet to the correct physical queue based on its key. FlexNIC [85] shows that such key-based steering improves throughput of the key-value store application by 30–45%.

To implement this strategy, we create `Q1` with multiple physical queues (line 1 in Figure 3.2). Steering a packet is controlled by assigning the target queue instance ID to the `qid` field of *per-packet state* in the C code of any element that precedes the queue. In this example, we set `state.qid = hash(pkt.key) % 3`, where `state` refers to *per-packet state*.

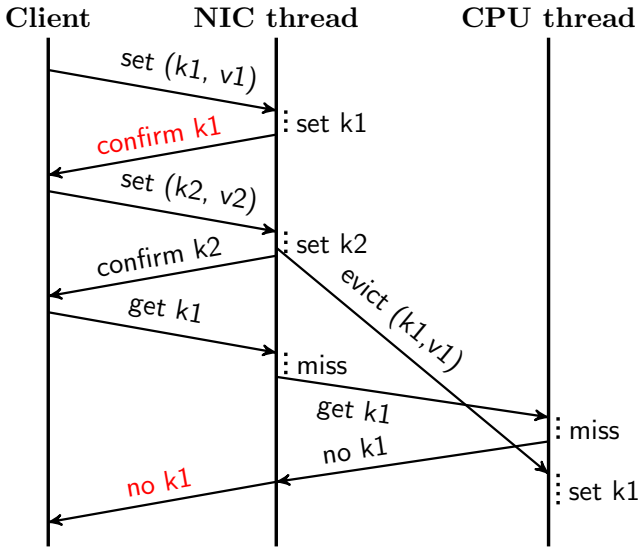


Figure 3.3: Inconsistency of a write-back cache if messages from NIC to CPU are reordered

Resource sharing. Developers may want to map multiple logical queues to the same physical queue for resource sharing, or vice versa for resource isolation. For example, they may want to consolidate infrequently used logical queues into one physical queue to obtain a larger batch of messages per PCIe transfer. In the sharded key-value store, we want to use the same physical queue to transport both the GET and SET requests of one shard so that the receiver's side processes these requests at the same rate as the sender's side. To implement this, we use `Q1` to represent two logical queues (line 1 in Figure 3.2): one for GET and another for SET. Different degrees of sharing can vary application performance by up to 16% (see Section 3.7).

Packet ordering. For correctness, developers may want to preserve the order of packets being processed from one device to another. For example, an alternative way to offload the key-value store is to use the NIC as a key-value cache, only forwarding misses to the CPU. To ensure consistency of the write-back cache, we must enforce that the CPU handles evictions and misses of the same key in the same order as the cache. Figure 3.3 shows an inconsistent outcome when an eviction and a miss are reordered. To avoid this problem, developers can map logical queues for evictions and misses to the same physical queue, ensuring in-order delivery.

The ability to freely map logical to physical queues lets programmers express different communication strategies with minimal effort in a declarative fashion. A queue can also be parameterized by whether its enqueuing process is lossless or lossy, where a lossless queue is blocking. Note that programmers are responsible for correctly handling multiple blocking queues.

Per-Packet State

FLOEM provides per-packet state, an abstraction that allows access to a packet and its metadata from any element without explicitly passing the state. To use this abstraction, programmers define its format and refer to it using the keyword `state`. For our key-value store, we define the format of the per-packet state as follows:

```
class MyState(State): # define fields in a state
    hash = Field(uint32_t)
    pkt  = Field(pointer(kvs_message))
    key  = Field(pointer(void),  size='state.pkt->keylen')
```

The provided element `from_net` creates a per-packet state and stores a packet pointer to `state.pkt` so that subsequent elements can access the packet fields, such as `state.pkt->keylen`. The element `hash` computes the hash value of a packet's key and stores it in `state.hash`, which is used later by element `hasht_get`. To handle a variable-size field, FLOEM requires programmers to specify its size, as with the key field above.

Caching Construct

With only minimal changes to a program, FLOEM offers developers a high-level caching construct for exploring caching on the NIC and storing outputs of expensive computation to be used in the future. First, programmers instantiate the caching construct `Cache` to create an instance of a cache storage and elements `get_start`, `get_end`, `set_start`, and `set_end`. Programmers then insert `get_start` right before the get query begins, and `get_end` right after the get query ends; a get query is computation we want to memoize. Programmers must also specify what to store as a key (input) and a value (output) in the cache; this can be done by assigning `state.key` and `state.keylen` (key and keylen fields of per-packet state) before the element `get_start`, and assigning `state.val` and `state.vallen` before `get_end`. If the application has a corresponding set query, elements `set_start` and `set_end` must be inserted, and those fields of the per-packet state must be assigned accordingly for the set query; a set query mutates application state and must be executed when a cache eviction occurs. Finally, programmers can use parameters to configure the cache with the desired table size, cache policy (either write-through or write-back), and a write-miss policy (either write-allocate or no-write-allocate).

For our key-value store example, we can use the NIC to cache outputs from hash table get operations by just inserting the caching elements, as shown in Figure 3.1(c). Notice that queues `Q1` and `Q2` are parts of the expensive queries (between `get_start` and `get_end` and between `set_start` and `set_end`) that can be avoided if outputs are in the cache.

Requirements. The get and set query regions cannot contain any *callable segment* (see the next subsection). Elements `get_start`, `get_end`, `set_start`, and `set_end` must be on the same device. Paths between `get_start` and `get_end`, and between `set_start` and `set_end`, must pass through the same set of queues (e.g., Figure 3.1(c)) to ensure the in-order delivery

of misses and evictions of the same key. Multiple caches can be used as long as cached regions are not overlapped. The compiler returns an error if a program violates these requirements.

Interfacing with External Code

To help developers offload parts of their applications to run on a NIC, we let them: (1) embed C code in elements, (2) implement elements that call external C functions available in linkable object files, and (3) expose segments of FLOEM elements as functions callable from any C program. The first mechanism is the standard way to implement an element. The second simply links FLOEM-generated C code with object files. For the last mechanism, we introduce a *callable segment*, which contains elements between a queue and an endpoint, or vice versa. An endpoint element may send/receive a value to/from an external program through its output/input port. A callable segment is exposed as a function that can be called by an external program to execute the elements in a segment.

In Figure 3.1(d), we implement simple computation, such as hashing and response packet construction, in FLOEM, but we leave complex functionality, including the hash table and item allocation, in an external C program. The external program interacts with the FLOEM program to retrieve a packet, send a get response, and send a set response via function `obtain_pkt`, `get_send`, and `set_send`, respectively. The following listing defines the function `obtain_pkt` using a callable segment. This function takes a physical queue ID as input, pulls the next entry from the queue with the given ID, executes element `retrieve_pkt` on the entry, and returns the output from `retrieve_pkt` as the function's return value.

```
class ObtainPkt(CallableSegment):
  def configure(self):
    self.inp = Input(int)      # argument is int
    self.out = Output(q_entry) # return value is q_entry

  def impl(self):
    self.inp >> Q1.qid
    Q1.deq >> retrieve_pkt >> self.out

ObtainPkt(name='obtain_pkt')
```

The external program running on the CPU calls `obtain_pkt` to retrieve a packet that has been processed by element `hash` on the NIC and pushed into queue `Q1`.

3.5 Compiler

The FLOEM compiler contains three primary components that: (1) translate a data-flow program with elements into C programs, (2) infer minimal data transfers across queues, and (3) expand the high-level caching construct into primitive elements, as depicted in Figure 3.4.

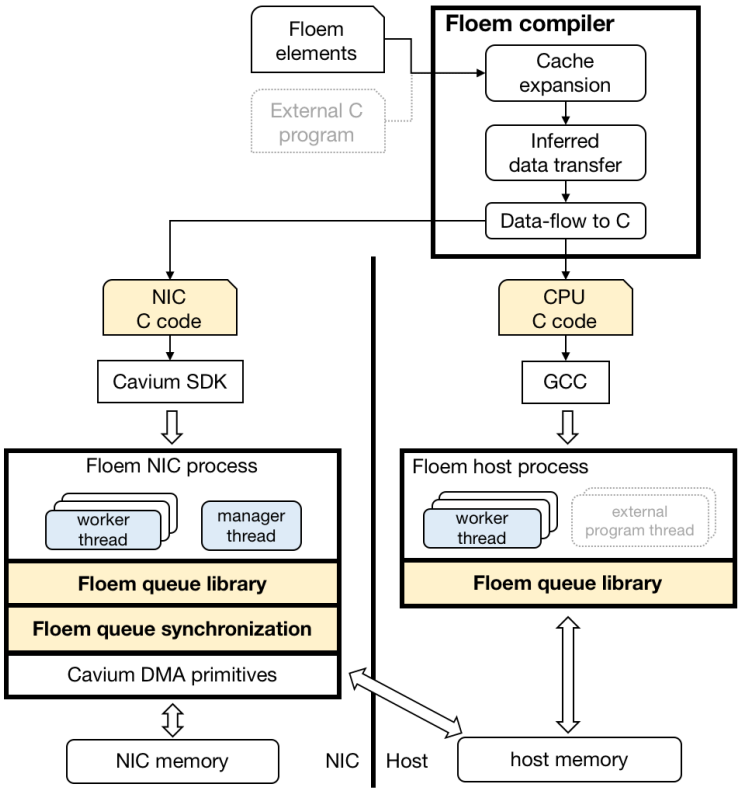


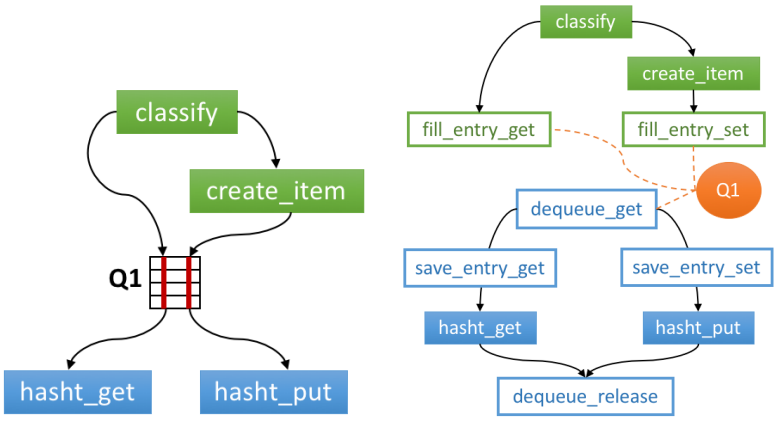
Figure 3.4: FLOEM system architecture

Data-flow To C

FLOEM compiles a data-flow program into two executable C programs: one running on the CPU and the other on the NIC. Our code generator compiles a segment of primitive elements into a chain of function calls, where one element corresponds to a function. The compiler replaces an output port invocation with a function call to the next element connected to that output port. The calling element passes an output value to the next element as an argument to the function call. If an output port is connected to multiple elements, we replace the output port invocation with multiple corresponding function calls. An `output switch {...}` block is transformed into conditional branches, while an `output {...}` block is removed. Earlier compiler passes transform queues (Section 3.5) and caching constructs (Section 3.5) into primitive elements.

Inferred Data Transfer

In this section, we explain how the FLOEM compiler infers which fields of a packet and its metadata must be sent across each queue, and how it transforms queues into a set of primitive elements.



(a) Before transformation (b) After transformation

Figure 3.5: The key-value store's data-flow subgraph in the proximity of queue `Q1` from the split CPU-NIC version

Liveness analysis. The compiler infers fields of per-packet state to send across a logical queue using a classical liveness analysis [8]. The analysis collects used and defined fields at each element and propagates information backward to compute a *live* set at each element (i.e., a set of fields that will be used by the element's successors). For each segment, the compiler also collects a *use* set of all fields that are accessed in the segment. We separately compute *live* and *use* sets for each queue's channel.

Transformation. After completing the liveness analysis, the compiler transforms each queue construct into multiple primitive elements that implement enqueue and dequeue operations. In the split CPU-NIC version of the key-value store example, the compiler transforms queue `Q1` in Figure 3.5(a) into the elements in Figure 3.5(b).

To enqueue an entry to a logical queue at a channel `X`, we first create element `fill_entry_X` to reserve a space in a physical queue specified by `state.qid`. We then copy the *live* per-packet state's fields at channel `X` into the queue. To dequeue an entry, element `dequeue_get` locates the next entry in a specified physical queue, classifies which channel the entry belongs to, and passes the entry to the corresponding output port (i.e., demultiplexing). Element `save_entry_X` allocates memory for the per-packet state on the receiver's side to store the *use* fields and a pointer to the queue entry so that the fields in the entry can be accessed later. Each `save_entry_X` is connected to the element that was originally connected to that particular queue channel. Finally, the compiler inserts a `dequeue_release` element to release the queue entry after its last use in the segment. The compiler generates different variations of `fill_entry_X` and `dequeue_get` depending on the parameters programmers select when creating queues in their applications. These generated elements utilize the built-in queue implementations described in Section 3.6.

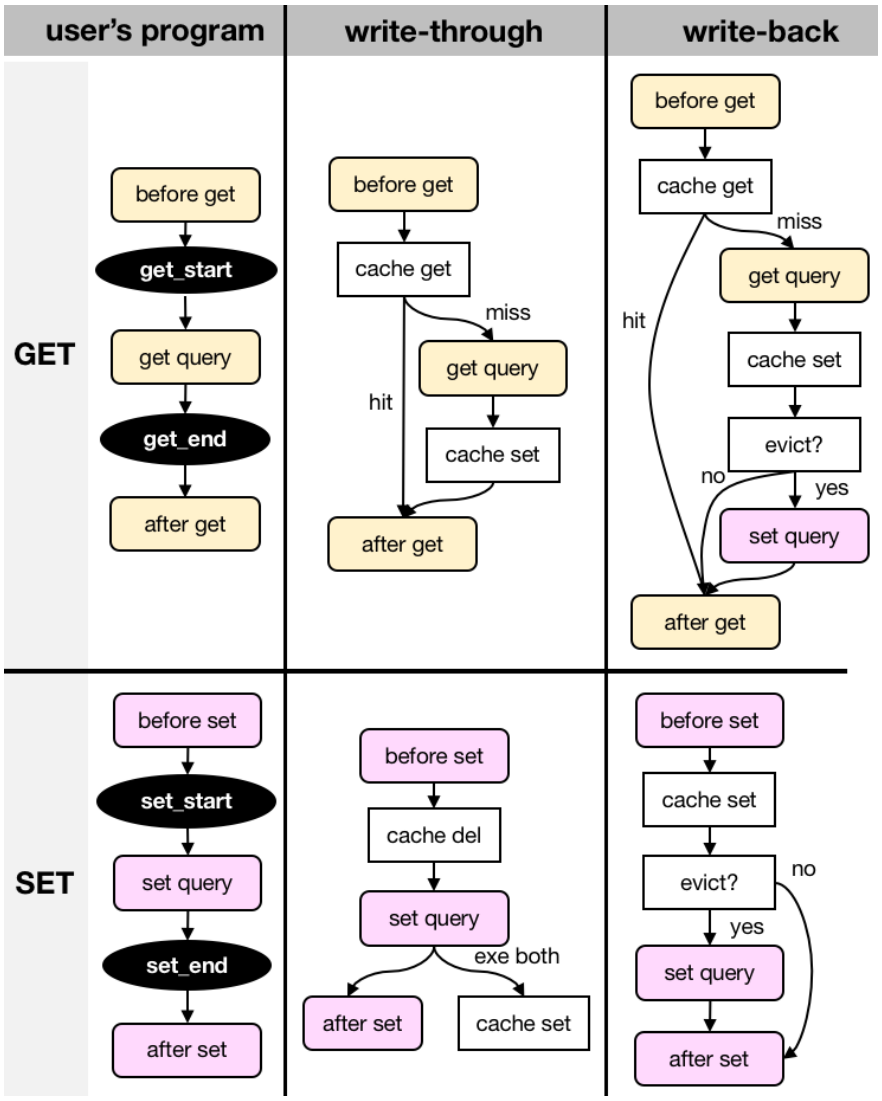


Figure 3.6: Cache expansion rules

Cache Expansion

The compiler expands each high-level caching construct into primitive elements that implement a cache policy using the expansion rules shown in Figure 3.6. Each node in the figure corresponds to a subgraph of one or more elements. For a write-through cache without allocation on write misses, the compiler expands the program graphs that handle get and set queries in the left column into the graphs in the middle column. For a write-back policy with allocation on write misses, the resulting graphs are shown in the right column. For get-only applications, we skip the set expansion rule.

We apply various optimizations to reduce response time. For example, when a new allocation causes an eviction in a write-back cache, we write back the evicted key asynchronously.

Instead of waiting for the entire `set query` to finish before executing `after get` (e.g., sending the response), we wait only until the local part of `set query` (on a NIC) reaches a queue to the remote part of `set query` (on a CPU). Once we successfully enqueue the eviction, we immediately execute `after get`.

Supported Targets

We prototype FLOEM on a platform with a Cavium LiquidIO NIC [3]. We use GCC and Cavium SDK [2] to compile C programs generated by FLOEM to run on a CPU in user mode and on a NIC, respectively. If a FLOEM program contains an interface to an external C program, the compiler generates a C object file that the external application can link to in order to call the interface functions.

Intrinsics, libraries, and system APIs of the two hardware targets differ. To handle these differences, FLOEM lets programmers supply different implementations of a single element class to target x86 and Cavium via `impl` and `impl_cavium` methods, respectively. If `impl_cavium` is not implemented, the compiler refers to `impl` to generate code for both targets. To generate programs with parallelism, FLOEM uses *pthread* on the CPU for multiple segments and relies on the OS thread scheduler. On the NIC, we directly use hardware threads and assign each segment to a dedicated NIC core. Consequently, the compiler prohibits creating more segments on the NIC than the maximum number of cores (12 for LiquidIO).

3.6 PCIe I/O Communication

To efficiently communicate between the NIC and CPU over PCIe, FLOEM provides high-performance, built-in FIFO queue implementations. Because DMA engines on the NIC are underpowered, they must be managed carefully. If we implemented queue logic together with data synchronization, the queue implementation would be extremely complicated and difficult to troubleshoot. Hence, we decouple queue logic (handled by a queue implementation) from data synchronization (handled by a queue synchronization layer). The queue synchronization layer (sync layer) can also be used for other queue implementations not yet provided, such as a queue with variable-size entries.

Our sync layer provides the illusion that the NIC writes directly to a circular buffer in host memory, where one circular buffer represents one physical queue. Under the hood, the sync layer performs the following optimizations:

- keep shadow a copy of a queue in the local NIC memory
- asynchronously synchronize the local copy with the master copy in host memory
- batch multiple DMA requests
- overlap DMA operations with other computation

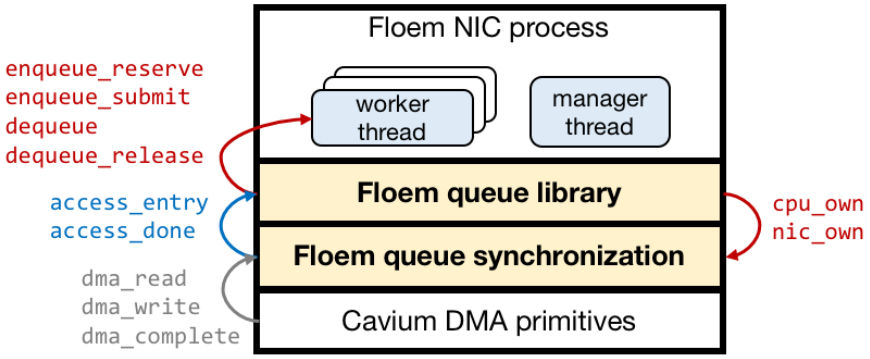


Figure 3.7: Provided API functions by different components. $A \xrightarrow{f} B$ represents: A provides function f for B to use.

Currently, we support only a one-way queue that transfers data either from CPU to NIC or from NIC to CPU.

Components, Agents, and Interaction

Code components. Sending data across PCIe using a queue in FLOEM requires five code components: NIC application code, CPU application code, a NIC queue, a CPU queue, and the queue synchronization layer. These are components highlighted in yellow in Figure 3.4. The NIC queue, CPU queue, and sync layer are libraries provided by FLOEM. The queue library and sync layer provide API functions for the other components to use, as depicted in Figure 3.7 and explained below.

- The sync layer provides functions `access_entry` and `access_done` for a NIC queue implementation to use.
- The NIC/CPU FIFO queue library provides functions `enqueue_reserve`, `enqueue_submit`, `dequeue`, and `dequeue_release` for NIC/CPU application code to use, and functions `cpu_own` and `nic_own` for the sync layer to use. The queue is FIFO with respect to the order of `enqueue_reserve` and `dequeue`.

NIC and CPU application code are generated by FLOEM compiler. However, programmers can also use our queue library for their own NIC and CPU application code written in C (instead of application code written in FLOEM data-flow language).

Agents. The process of transferring data requires at least three threads, highlighted in blue in Figure 3.4.

- A NIC runtime manager thread maintains coherence between queue storages on the NIC and CPU.

- A NIC worker thread runs NIC application code, which calls NIC queue functions, which in turn call functions provided by the sync layer.
- A CPU worker thread runs CPU application code, which calls CPU queue functions.

Typical interaction. To simplify the explanation, consider sending data from CPU to NIC using a queue with a single entry. For a queue that sends data from a CPU to NIC, a queue entry is CPU owned, if the entry is empty or being enqueued; it is NIC owned, if it contains data. A typical interaction between the NIC runtime manager thread, NIC worker thread, and CPU worker thread is depicted in Figure 3.8. The sync layer provides an illusion to the NIC and CPU application threads as if there is one copy of a queue storage. Thus, from the application threads' perspective, Figure 3.8(b) is seen as Figure 3.8(a). The pseudocode of our example CPU and NIC application code in C syntax is shown in Figure 3.9. The pseudocode of our queue library is shown in Figure 3.10, while the pseudocode of our sync layer is shown in Figure 3.11.

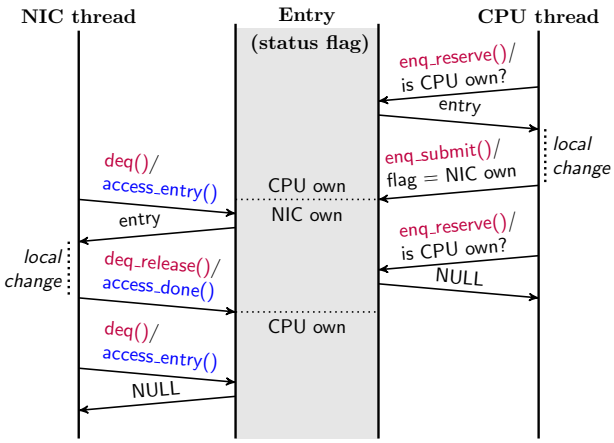
We do not explicitly track a queue's head and tail; instead, we use a status flag in each entry to determine if an entry is filled or empty. We choose this design to synchronize both the queue entry's content and status using one DMA operation instead of two. Thus, our runtime continuously checks the state of every queue entry and performs actions accordingly. Typically, a queue entry on the NIC cycles through *invalid*, *reading*, *valid*, *modified*, and *writing* states, as shown in Figure 3.8(b).

To send a queue entry from CPU to NIC, the CPU application obtains an access to a queue entry by calling `enqueue_reserve` (label A in Figure 3.9), which returns an entry if it is CPU owned (label E in Figure 3.10). The CPU application writes content into the queue entry and calls `enqueue_submit` (label B), which changes the status field of the entry to be NIC owned (label F).

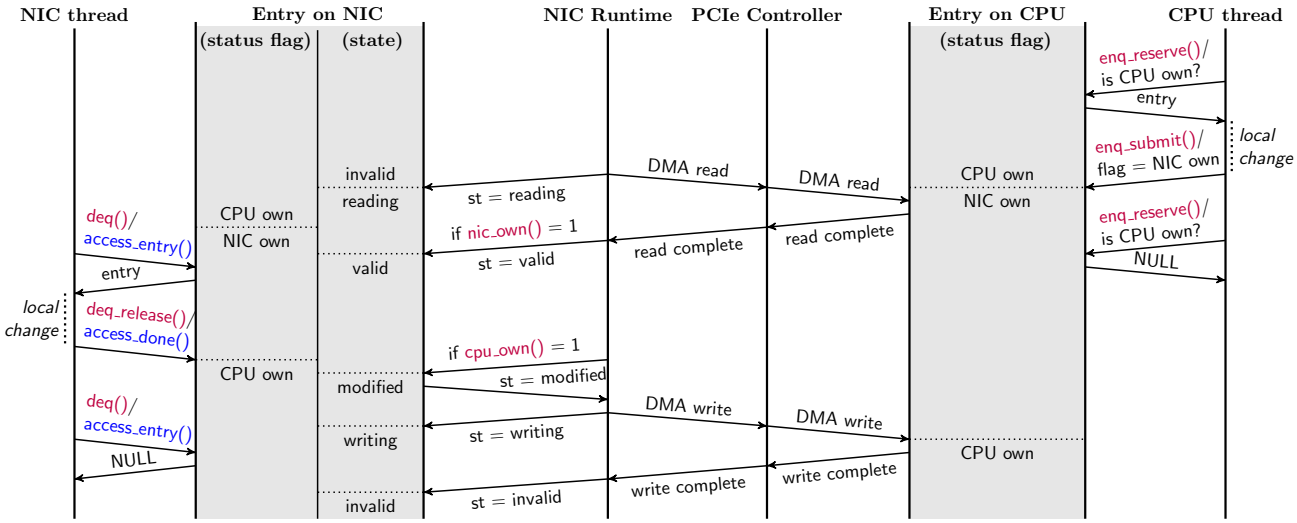
The NIC runtime continuously checks the state of every queue entry. If an entry's state is *invalid* (label J in Figure 3.11), it issues DMA read and sets the entry's state to *reading* to mark that the entry is being read from the host memory. Once the DMA read request is complete (label K), the runtime checks if the entry is NIC owned using `nic_own` function. If the entry is NIC owned, the runtime changes the entry's state to *valid*.

The NIC application dequeues an entry by calling `dequeue` (label C), and `dequeue` in turn calls `access_entry` function (label G). If the entry is *valid*, then `access_entry` returns an entry pointer that the NIC application can access (label I). Once the NIC application is done accessing the entry, it releases the entry by calling `dequeue_release` (label D). `dequeue_release` modifies the status field of the entry to be CPU owned and calls `access_done` (lable H). `dequeue_release` is a way for the data receiver (NIC) to notify the data sender (CPU) that it is done processing the entry.

When the runtime sees that the entry is CPU owned (labels L and M), via `cpu_own` function, it issues a DMA write request to write the modified status field to CPU memory.



(a) Assuming there were shared memory between CPU and NIC



(b) Actual interaction

Figure 3.8: Interaction between NIC runtime manager thread, NIC worker thread, CPU workder thread, and status of a queue entry. Red highlights functions provided by the queue library. Blue highlights functions provided by the queue synchronization layer. enq and deq are abbreviations for enqueue and dequeue respectively.

```
// CPU application
void cpu_app() {
  int qid = queue_init();

  while(true) {
    struct my_entry *entry = enqueue_reserve(qid); (A)
    if(entry) {
      fill_content(entry->content);
      enqueue_submit(entry); (B)
    }
  }
}

// NIC application
void nic_app() {
  int qid = queue_init(host_storage_address);

  while(true) {
    struct my_entry *entry = dequeue(qid); (C)
    if(entry) {
      do_something(entry->content);
      dequeue_release(entry); (D)
    }
  }
}

// Running threads
NIC runtime manager thread: runtime();
NIC worker thread: nic_app();
CPU worker thread: cpu_app();
```

Figure 3.9: Example application pseudocode

Once the DMA write request is complete (label N), the runtime changes the entry's state from *writing* to *invalid*. Then, this process repeats forever.

If the NIC application tries to dequeue an entry that is not in a *valid* state, `access_entry` returns NULL, and `dequeue` in turn returns NULL. In this scenario, the NIC application will have to retry again later. Similarly, if the CPU application tries to fill in an entry that is NIC owned, the `enqueue_reserve` will return NULL, and the CPU application will have to retry later.

The NIC-to-CPU transfer process is very similar to CPU-to-NIC except that the NIC is the one writing the entry's content, and the CPU modifies a queue entry by only changing the status field. For this, the NIC queue library also provides `enqueue_reserve` and `enqueue` functions, whereas the CPU queue library provides `dequeue` and `dequeue_release` as well.

```
// CPU queue implementation (queue with one entry)
struct my_entry {  // custom queue entry
  int status;
  int content;
}

void *storage[MAX_QUEUES];
int qid = 0;

int queue_init() {
  storage[qid] = malloc(sizeof(struct my_entry));
  qid++;
  return qid - 1;
}

void *enqueue_reserve(int qid) {
  struct my_entry *entry = storage[qid];
  if(entry->status  == 0) (E)
    return entry;
}

void enqueue_submit(struct my_entry *entry) {
  entry->status = 1; (F)
}

// NIC queue implementation (queue with one entry)
struct my_entry { // custom queue entry
  int status;
  int content;
}

int nic_own(void *p) {
  uint8_t *entry = p;
  return entry->status == 1;
}

int cpu_own(void* p) {
  uint8_t *entry = p;
  return entry->status == 0;
}

int queue_init(uint64_t address) {
  return create_dma_circular_queue(
           address, sizeof(struct my_entry), sizeof(struct my_entry),
           nic_own, cpu_own);
}

void *dequeue(int qid) {
  return access_entry(qid, 0); (G)
}

void dequeue_release(struct my_entry *entry) {
  entry->status = 0;
  access_done(entry); (H)
}
```

Figure 3.10: Pseudocode of CPU queue implementation, and NIC queue implementation

```
// Queue synchronization layer
int create_dma_circular_queue(address,
           n * sizeof(struct my_entry),
           sizeof(struct my_entry),
           nic_own, cpu_own);

void *access_entry(int qid, int index) {
  entry = pointer to queue qid at a given index;
  if(state == VALID) return entry; (I)
  else return NULL;
}

void access_done(void *entry) {
  // This function does some work when the batching optimization is enabled.
}

void manage_queue_entry(void* entry) {
    if(state == INVALID) { (J)
      DMA_read(entry_host_address, entry, entry_size);
      state = READING;
    }
    else if(state == READING && DMA read is complete) { (K)
      if(nic_own(entry)) state = VALID;
      else state = INVALID;
    }
    else if(state == VALID && cpu_own(entry)) { (L)
      state = MODIFIED;
    }
    else if(state == MODIFIED) { (M)
      DMA_write(entry_host_address, entry, entry_size);
      state = WRITING;
    }
    else if(status == WRITING && DMA write is complete) { (N)
      state = INVALID;
    }
}

void runtime() {
  while(true) {
    for(e in entries from all queues)
      manage_queue_entry(e);
  }
}
```

Figure 3.11: Pseudocode of the queue synchronization layer

Queue Implementation

A NIC queue implementation must use an API provided by the queue synchronization layer to access queue entries. A CPU queue implementation is the same as the NIC queue implementation except that the CPU implementation checks and modifies the ownership of a queue entry directly without using the API provided by the sync layer. The queue implementation explained in the previous section is an example. Queue developers may implement a queue with a different queue entry structure and provide different functions other than `enqueue_reserve`, `enqueue_submit`, `dequeue`, and `dequeue_release` for application code to use as long as it satisfies the properties explained later in this section.

API provided by the queue synchronization layer

```
create_dma_circular_queue(
  uint64_t host_storage_address,
  int queue_size, // in bytes
  int entry_size, // in bytes
  int (*nic_own)(void*),
  int (*cpu_own)(void*))
--> int qid
```

During initialization, a queue implementation on the NIC must register a queue to the optimization layer via this function. Queue developers must supply `int nic_own(void *entry)`, a status checking function that takes a pointer to a queue entry and returns whether the entry is ready to be processed on the NIC or not; if ready, it returns 1, otherwise returns 0. Similarly, `int cpu_own(void *entry)` returns whether the entry is done being processed on the NIC or not; if done, it returns 1, otherwise returns 0. If the queue is for sending data from NIC to CPU, from the NIC's enqueueing perspective, an entry is NIC owned, if it is empty and available to be filled. If the queue is for receiving data from the CPU, an entry is NIC owned, if it is finished being filled by the CPU. `nic_own` and `cpu_own` checks an entry's status by looking at the entry's status field.

```
access_entry(int qid, int index) --> void *entry

access_done(void* entry) --> void
```

In a NIC queue implementation, enqueue and dequeue routines must call `access_entry (qid,index)` to obtain a pointer to a queue entry at a given index. `access_entry` returns NULL if the entry is not ready to be accessed on the NIC. When the queue finishes accessing the entry, the queue must call `access_done`. For enqueuing process, the queue calls `access_done` when it finishes enqueuing the entry. For dequeue process, the queue calls the function when it finishes using the entry.

When `access_entry(qid,index)` returns a pointer to an entry, the return pointer always contains the same address for the same `index`. For example, if two threads call `access_entry(qid,index)` at the same time, and both get pointers to to the same entry `index`

(the same local memory address that stores entry `index`). A thread will see the change made to the entry by another thread.

Queue properties

1. FIFO.
2. Circular pattern: its entries are stored in a circular pattern according to its enqueuing order.
3. One-way queue: a queue cannot be used for sending data both from NIC to CPU and from CPU to NIC.
4. Fixed size: all entries in the queue are of the same size. This property can be relaxed with a slight modification to the sync layer.
5. Must not access an entry beyond its size.
6. NIC own vs. CPU own status of an entry can be determined by solely examining the entry.
7. Must handle concurrency between multiple threads accessing a queue on the same device.
8. The structure of a queue entry and `nic_own`/`cpu_own` functions must be carefully designed to guarantee correctness.

Regarding Property 6, an entry usually contains a status field. This field must be accessed and modified between `access_entry` and `access_done` because it is a part of an entry. However, an explicit status field is not required as long as the status can be determined by solely examining the entry.

Regarding Property 7, the queue implementation must guarantee race-free enqueue/dequeue. For example, if two threads try to enqueue into a queue at the same time by calling `access_entry(qid,i)` at the same index `i`, both of them may get entry pointers for accessing the same entry. Therefore, the queue implementation is responsible for preventing double enqueuing/dequeuing using locks or atomic instructions. This is something queue developers have to handle already when implementing a queue for a multi-threaded program running on one device.

Property 8 is most difficult to reason about as the queue developers have to understand how the DMA controller transfers data between the NIC and CPU. For the Cavium LiquidIO NIC, the atomic unit of data transfer across PCIe is 64 bytes. When we issue a DMA operation for data larger than 64 bytes, the DMA controller will perform multiple in-order 64-byte transfers. Problems may arise when an entry is larger than 64 bytes. First, we cannot put a status field at the beginning of an entry because, during data transfer, the other side may see a ready status before receiving the content. However, if we put the status field at

the end of an entry, the NIC may still see a ready status but stale content because the DMA controller transfers the old content just before the CPU updates the content and status, and then transfers the updated status. Our solution is to include checksum in an entry, and make `nic_own` checks the checksum. We use this checksum for CPU-to-NIC transfers. However, we do not need the checksum for NIC-to-CPU transfers because the NIC is the one issuing DMA operations, so it will never issue a transfer on an entry that is still being modified locally.

If the queue implementation meets these properties, the sync layer guarantees delivery of every single queue entry for blocking (non-dropping) queues.

Queue synchronization layer

The sync layer relies on FLOEM NIC runtime to maintain coherence between buffers on the NIC and CPU by taking advantage of the circular access pattern of reads followed by writes.

Typically, a queue entry on the NIC cycles through *invalid*, *reading*, *valid*, *modified*, and *writing* states, as shown in Figures 3.8(b) and 3.12. An *invalid* entry contains stale content and must be fetched from host memory. An asynchronous DMA read transitions an entry from *invalid* to *reading* state. Once the read completes, and the entry is NIC owned (indicated by the status flag), the entry transitions to *valid* state. It may transition back to *invalid* if it is still CPU owned, for example, when the NIC attempts to dequeue an entry that the CPU has not finished enqueuing. The runtime uses the status checking functions provided by the queue implementation to check an entry's status flag. The program running on the NIC can access only *valid* entries; function `access_entry` returns the pointer to an entry if it is in *valid* state; otherwise, it returns NULL.

An entry transitions from *valid* to *modified* once the queue implementation calls function `access_done` to indicate that it is finished accessing that entry. An asynchronous DMA write then transitions the entry to *invalid* state, based on the assumption that the CPU side will eventually modify it, and the NIC must read it from the CPU. This completes a typical cycle of states through which an entry passes. Under DMA operation failures (which are rare), the runtime will reissue the operation until successful.

The complete state machine of a queue entry is displayed in Figure 3.12, and the pseudocode of the state transition is shown at labels J–N in Figure 3.11.

In contrast, it is sufficient to keep track of only CPU own and NIC own states for a queue entry on CPU. This is because CPU is not the one issuing DMA operations. A queue entry's flag sufficiently captures these two states, so we do not need an extra layer to keep track of the queue entry's state. Unlike on CPU, NIC has to track more states. A queue entry itself only captures the valid state and non-valid states, and the non-valid states cannot be distinguished further by just looking at the entry. Therefore, the runtime has to track the other states explicitly.

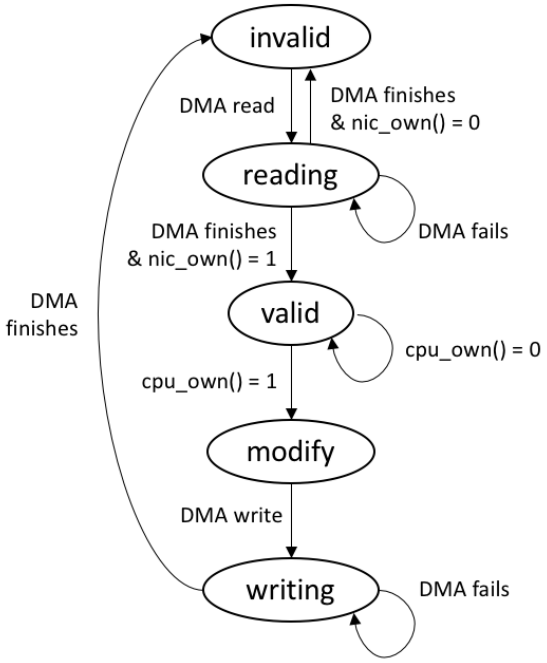


Figure 3.12: Queue entry's state machine

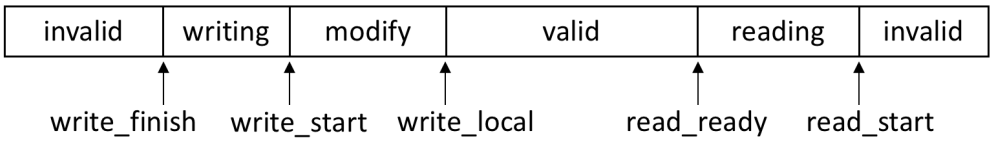


Figure 3.13: States of different portions in a queue. Pointers always advance to the right and wrap around. Each portion contains zero or more queue entries. A timeline of a queue entry is depicted in Figure 3.8(b).

Batching and Runtime

In the actual implementation, we do not track the state of each queue entry one-by-one. Instead, we use pointers to divide a queue into different portions with different states. When a pointer advances to the right, we effectively change states of the entries that the pointer has moved past. These pointers wrap around a circular queue buffer. Figure 3.13 depicts how we keep track of the states of different portions of the queue using pointers; the entries between *read_ready* and *read_start* are in the *reading* state; the entries between *write_local* and *read_ready* are in the *valid* state; and so on.

The runtime has a dedicated routine to advance each of the five pointers and executes these routines in a round-robin fashion.

1. The DMA read routine issues a DMA read on the next batch of entries, and advances the *read_start* pointer to the end of the batch.

2. The read completion handling routine checks if a DMA read is complete. If so, it scans the batch of entries using `nic_own` function. If the next entry is NIC owned, it advances *read_ready*; otherwise, it updates the *read_start* pointer to point at this entry; consequently, the DMA read routine will reissue a DMA read from this entry.

3. The write scanning routine checks the entry at *write_local* using `cpu_own` function, and advances *write_local* when the entry is CPU owned.

4. The DMA write routine issues a DMA write on a batch of entries that are CPU owned, and advances the *write_start* pointer to the end of the batch.

5. The write completion handling routine advances the *write_finish* pointer to the end of the batch when a DMA write is complete.

The read and write completion handling routines are also responsible for handling failures (e.g., a DMA command is dropped because the command queue is full, or an operation takes much longer than the threshold). We use a configurable number of dedicated NIC cores (manager threads) to execute the runtime. Each core manages a non-overlapping subset of queues.

Other Optimizations

Instead of making the runtime thread changes states of entries from *valid* to *modified* one-by-one by calling `cpu_own`, we keep track of the count of CPU own entries in each batch. When NIC worker threads call `access_done`, they atomically increment the count. With this, the runtime thread can check if the count is equal the number of total entries in the batch. If equal, the runtime issues a DMA write.

Another optimization is to reduce the CPU-to-NIC communication for NIC-to-CPU queues. Recall that for a NIC-to-CPU queue, we still need to perform DMA reads from CPU memory to obtain the status fields of queue entries modified by the CPU. This increases the traffic of DMA operations significantly. To reduce the number of DMA reads, we create a special notification queue that sends a notification message from CPU to NIC when CPU dequeues half entries from any queue. When the NIC runtime receives a notification message, it changes the states of the half entries of the notified queue from *invalid* to *valid*, skipping DMA reads entirely.

3.7 Evaluation

FLOEM is open-source and availabe at `github.com/mangpo/floem`. We ran experiments on two small-scale clusters to evaluate the benefit of offloading on servers with different generations of CPUs: 6-core Intel X5650 in our *Westmere* cluster, and 12-core Intel E5-2680 v3 in our *Sandy Bridge* cluster (more powerful). Each cluster had four servers; two were equipped with Cavium LiquidIO NICs, and the others had Intel X710 NICs. All NICs had two 10Gbps ports.

We evaluated CPU-only implementations on the servers with the Intel X710 NICs, using DPDK [4] to send and receive packets bypassing the OS networking stack to minimize overheads. We used the servers with the Cavium LiquidIO NICs to evaluate implementations with NIC offloading. The Cavium LiquidIO has a 12-core 1.20GHz cnMIPS64 processor, a set of on-chip/off-chip accelerators (e.g., encryption/decryption engines), and 4GB of on-board memory.

Programming Abstraction

We implemented in FLOEM two complex applications (key-value store and real-time data analytics) and three less complex network functions (encryption, flow classification, and network sequencer).

Hypothesis 1 FLOEM *lets programmers easily explore various offloading strategies to improve application performance.*

The main purpose of this experiment is to demonstrate that FLOEM makes it easier to explore alternative offloading designs, *not* to show when or how one should or should not offload an application to a NIC.

For the complex applications, we started with a CPU-only solution as a baseline by porting parts of an existing C implementation into FLOEM. Then, we used FLOEM to obtain a simple partition of the application between the CPU and NIC for the first offload design. In both case studies, we found that the first offloading attempt was unsuccessful because an application's actual performance can greatly differ from a conceptual estimate. However, we used FLOEM to redesign the offload strategy to obtain a more intelligent and higher performing solution, with minimal code changes, and achieved 1.3–3.6× higher throughput than the CPU-only version.

For the less complex workloads, FLOEM let us quickly determine whether we should dedicate a CPU core to handle the workload or just use the NIC and save CPU cycles for other applications. By merely changing FLOEM's device mapping parameter, we found that it was reasonable to offload encryption and flow classification to the NIC, but that the network sequencer should be run on the CPU. The rest of this section describes the applications in our experiment in greater detail.

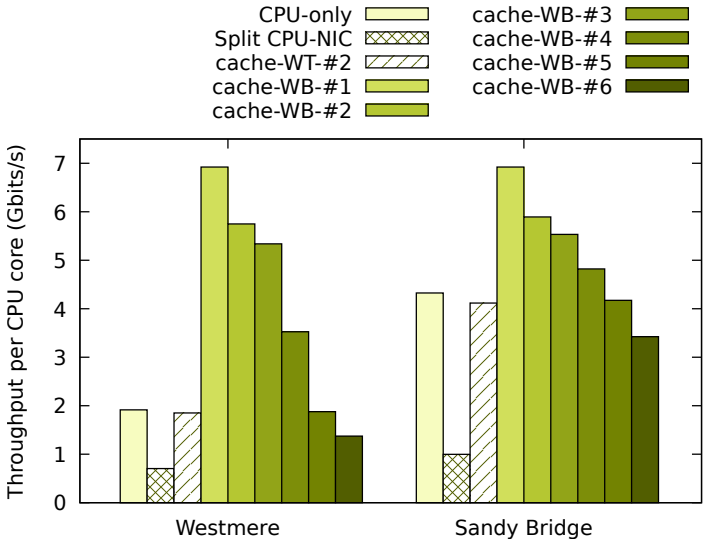


Figure 3.14: Throughput per CPU core of different implementations of the key-value store. WB = write-back, WT = write-through. #N in "cache-WB-#N" is the configuration number. Table 3.2 shows the cache sizes of the different configurations and their resulting hit rates.

Version	**Obtained From**	**Effort (loc)**	**Details**
Existing	N/A	1708	Expert-written C program
CPU-only	Existing	replace 538 with 334	Refactor C program into FLOEM elements.
Split CPU-NIC	CPU-only	add 296	Create queues. NIC remotely allocates items on CPU memory.
Caching	CPU-only	add 43	Create a cache. Assign key, keylen, val, vallen.
NIC caching	Caching	add 62	Create queues and segments.

Table 3.1: Effort to implement key-value store. The last column describes specific modification details other than creating, modifying, and rewiring elements. As a baseline, code relevant to communication on the CPU side alone was 240 lines in a manual C implementation.

Case Study I: Key-Value Store

In this case study, we used one server to run the key-value store and another to run a client generating workload, communicating via UDP. The workload consisted of 100,000 key-value pairs of 32-byte keys and 64-byte values, with the Zipf distribution ($s = 0.9$) of 90% GET requests and 10% SET requests, the same workload used in FlexNIC [85]. We used a single CPU core with a NIC offload (potentially with multiple NIC cores); this setup was reasonable since other CPU cores may be used to execute other applications simultaneously. Figure 3.14 shows the measured throughput of different offloading strategies, and Table 3.1 summarizes the implementation effort.

Config.	#1	#2	#3	#4	#5	#6	#2 (WT)
# of buckets	2^{15}	2^{15}	2^{15}	2^{15}	2^{14}	2^{14}	2^{15}
# of entries	∞	5	2	1	1	1	5
hit rate (%)	100	97.2	88.4	75.3	65.0	55.2	90.3

Table 3.2: The sizes of the cache (# of buckets and # of entries per bucket) on the NIC and the resulting cache hit rates when using the cache for the key-value store. All columns report the hit rates when using write-back policy except the last column for write-through. ∞ entries mean a linked list.

CPU-only (Figure 3.1(a)): We ported an existing C implementation, which runs on a CPU using DPDK, into FLOEM except for the garbage collector of freed key-value items. This effort involved converting the original control-flow logic into the data-flow logic, replacing 538 lines of code with 334 lines. The code reduction came from using reusable elements (e.g., `from_net` and `to_net`), so we did not have to set up DPDK manually.

***CPU-NIC split (Figure 3.1(b))*:** We tried a simple CPU-NIC partition, following the offloading design of FlexKVS [85], by modifying 296 lines of the CPU-only version; this offload strategy was carefully designed to minimize computational cycles on a CPU. It required many changes because the NIC (`create_item` element) creates key-value items that reside in CPU memory. Unexpectedly, this offload strategy lowered performance (the second bar). Profiling the application revealed a major bottleneck in the element that prepares a GET response on the NIC. The element issued a blocking DMA read to retrieve the item's content from host memory. This DMA read was not part of queue `Q2` because that queue sent only the pointer to the item, not the item itself. Therefore, the runtime could not manage this DMA read; as a result, this strategy suffered from this additional DMA cost.

***NIC caching (Figure 3.1(c))*:** We then used FLOEM to explore a completely different offload design. Since the Cavium NIC has a large amount of local memory, we could cache a signification portion of the key-value store on the NIC. This offload design, previously explored, was shown to have high performance [94]. Therefore, we modified the CPU-only version by inserting the caching construct (43 lines of code) as well as creating segments and inserting queues (62 lines of code). For a baseline comparison, code relevant to communication on the CPU side alone was already at 240 lines in a manually-written C implementation of FlexKVS with a software NIC emulation. This translated to fewer than 15 lines of code in FLOEM. These numbers show that implementing a NIC-offload application without FLOEM requires significantly more effort than with FLOEM.

Regarding performance, the third bar in Figure 3.14 reports the throughput when using a write-through cache with 2^{15} buckets and five entries per bucket, resulting in a 90.3% hit rate. According to the result, the write-through cache did not provide any benefit over the CPU-only design, even when the cache hit rate was quite high. Therefore, we configured the caching construct to use a write-back policy (by changing the cache policy parameter) because write-back generally yields higher throughput than write-through. The remaining bars show the performance when using a write-back cache with different cache sizes, resulting in the different hit rates shown in Table 3.2. This offloading strategy improved throughput

over the CPU-only design by 2.8–3.6× on Westmere and 28–60% on Sandy Bridge when the hit rate exceeded 88% (configuration #1–3).

Notice that at high cache hit rates, the throughput for this offload strategy was almost identical on Westmere and Sandy Bridge regardless of the CPU technology. The NIC essentially boosted performance on the Westmere server to be on par with the Sandy Bridge one. In other words, an effective NIC offload reduced the workload's dependency on CPU processing speed.

Case Study: Distributed Real-Time Data Analytics

Distributed real-time analytics is a widely-used application for analyzing frequently changing datasets. Apache Storm [1], a popular framework built for this task, employs multiple types of workers. A worker thread executes one worker. Spout workers emit tuples from a data source; other workers consume tuples and may emit output tuples. De-multiplexing threads route incoming tuples from the network to local workers. Multiplexing threads route tuples from local workers to other servers and perform simple flow control. Our specific workload ranked the top n users from a stream of Twitter tweets. In this case study, we optimized for throughput per CPU core. Figure 3.15 and Table 3.3 summarize the throughput and implementation effort of different strategies, respectively.

CPU-only: We ported demultiplexing, multiplexing, and DCCP flow-control from FlexStorm [85] into Floem but kept the original implementation of the workers as an external program. We used *callable segments* to define functions `inqueue_get` and `outqueue_put` for workers (in the external program) to obtain a task from the demultiplexer and send a task to the multiplexer (in Floem). This porting effort involved replacing 1,192 lines of code with only 350 lines. The code reduction here was much higher than in the key-value store application because FlexStorm's original implementation required many communication queues, which were replaced by Floem queues. The best CPU-only configuration that achieved the highest throughput per core used three cores for three workers (one spout, one counter, and one ranker), one core for demultiplexing, and two cores for multiplexing.

Split CPU-NIC: As suggested in FlexNIC, we offloaded (de-)multiplexing and flow control to the NIC, by annotating parameter `device=NIC` of the segments that run this computation (one line of code change). This version, however, lowered throughput slightly compared to the CPU-only version.

Redesigned CPU-NIC: The split CPU-NIC version can be optimized further. A worker can send its output tuple to another local worker or a remote worker over the network. For the former case, a worker sends a tuple to the multiplexer on the NIC, which in turn forwards it to the target worker on the CPU. Notice that this CPU-NIC-CPU round-trip is unnecessary. To eliminate this communication, we created bypass queues for workers to send tuples to other local workers without involving the multiplexer.

To do this, we had to modify only a few lines of code in the callable segment for `outqueue_put` function as follows.

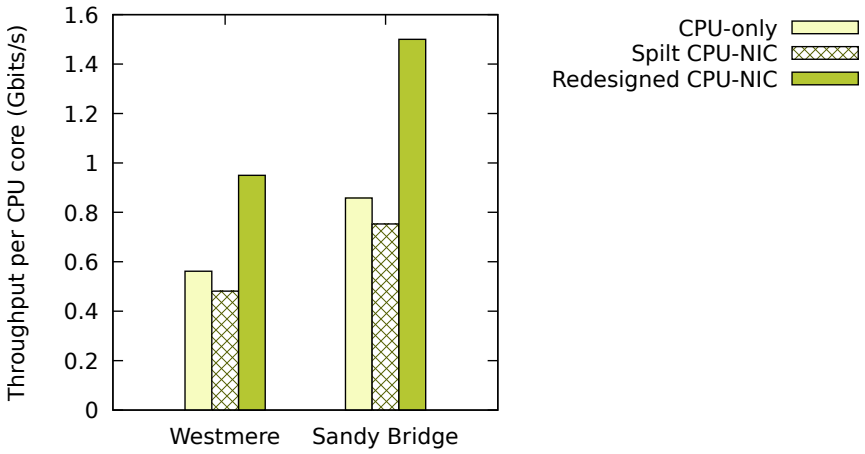


Figure 3.15: Throughput per CPU core of different Storm implementations

Version	Obtained From	Effort (loc)	Details
Existing	N/A	2935	Expert-written C program
CPU-only	Existing	replace 1192 with 350	Refactor C program into FLOEMelements.
Split CPU-NIC	CPU-only	modify 1	Change `device` parameter.
Redesigned	Split CPU-NIC	add 23	Create bypass queues.

Table 3.3: Effort to implement Storm. The last column describes specific modification details other than creating, modifying, and rewiring elements.

```
class outqueue_put(CallableFunction):
  ...
  def impl(self):
-     self.inp >> outqueue_enq
+     self.inp >> local_or_remote
+     local_or_remote.local >> get_worker_core >> bypass_enq
+     local_or_remote.send >> outqueue_enq
```

With this slight modification, we achieved 96% and 75% higher throughput than the CPU-only design on the Westmere and Sandy Bridge cluster, respectively.

Other Applications

The following three applications are common network function tasks. Because of their simplicity, we did not attempt to partition them across the CPU and NIC. Figure 3.16 reports throughput when using one CPU core on a Sandy Bridge server or offloading everything to the Cavium NIC. In our experiment, we used a packet size of 1024 bytes for encryption and network sequencer, and 80 bytes for flow classification.

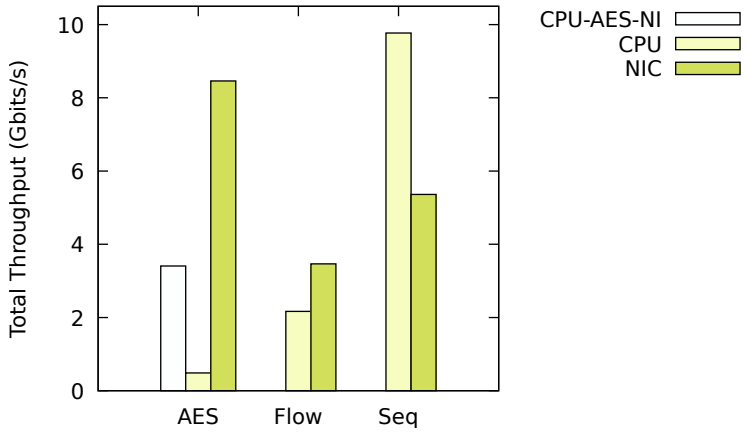


Figure 3.16: Throughput of AES encryption, 3DES encryption, flow classification, and network sequencer running on one CPU core and the LiquidIO NIC. 'CPU-AES-NI' is running on one CPU core with AES-NI.

Encryption is a compute-intensive stateless task, used for Internet Protocol Security. In particular, we implemented AES-CBC-128. We wrote two CPU versions: (1) using Intel Advanced Encryption Standard New Instructions (AES-NI), and (2) without AES-NI, which is available in only some processors. NIC Offloading improved throughput by 2.5× and 17.5× with and without AES-NI on CPU, respectively. Using AES-NI improved performance on the CPU but to a lesser degree than utilizing all encryption co-processors on the NIC. This result would be difficult to predict without an empirical test.

Flow classification is a stateful task that tracks flow statistics. We categorized flows using the header 5-tuple and used a probabilistic data structure (a count-min sketch) to track the number of bytes per flow. This application ran slightly faster on the NIC. Therefore, it seems reasonable to offload this task to the NIC if we want to spare CPU cycles for other applications.

Network sequencer orders packets based on predefined rules. It performs simple computation and maintains limited in-network state. This function has been used to accelerate distributed system consensus [97] and concurrency control [96]. Our network sequencer was 82% faster on the CPU core than on the NIC. Application throughput did not scale with the number of cores because of the group lock's contention; the number of locks acquired by each packet was 5 out of 10 on average in our synthetic workload, making this task inherently sequential. Therefore, using one fast CPU core yielded the best performance. We also tried running this program using multiple CPU cores, but throughput stayed the same as we increased the number of cores. On the NIC, using three cores offered the highest performance.

In summary, even for simple applications, it is not obvious whether offloading to the NIC improves or degrades performance. Using FLOEM lets us answer these questions quickly and precisely by simply changing the device parameter of the computation segment to either CPU or NIC. Comparing cost-performance or power-performance is beyond the scope of this paper. Nevertheless, one can use FLOEM to experiment with different configurations for a specific workload to optimize for a particular performance objective.

Logical-to-Physical Queue Mapping

Hypothesis 2 *Logical-to-physical queue mapping lets programmers implement packet steering, packet ordering, and different degrees of resource sharing.*

Packet steering. Storm, the second case study, required packet steering to the correct input queues, each dedicated to one worker. This was done by creating a queue with multiple physical instances and by setting `state.qid` according to an incoming tuple's type.

Packet ordering. The write-back cache implementation required in-order delivery between CPU and NIC to guarantee consistency (see Section 3.4).

Resource sharing. For the split NIC-CPU version of the key-value store, sending both GET and SET requests on separate physical queues offered 7% higher throughput than sharing the same queue. This is because we can use a smaller queue entry's size to transfer data for GET requests. In contrast, for our Storm application, sharing the same physical output queue between multiple workers yielded 16% higher throughput over separate dedicated physical queues. Since some workers infrequently produce output tuples, it was more efficient to combine tuples from all workers to send over one queue. Hence, it is difficult to predict whether sharing or no sharing is more efficient, so queue resource sharing must be tunable.

Inferred Data Transfer

Hypothesis 3 *Inferred data transfer improves performance relative to sending an entire packet.*

In this experiment, we evaluated the benefit of sending only a packet's live fields versus sending an entire packet over a queue. We measured the throughput of transmitting data over queues from the NIC to CPU when varying the ratio of the live portion to the entire packet's size (*live ratio*), detailed in Table 3.4. The sizes of live portions and packets were multiples of 64 bytes because performance was degraded when a queue entry's size was not a multiple of 64 bytes, the size of a CPU cache line. We used numbers of queues and cores that maximized throughput. As shown on the table, sending only live fields improved throughput by 1.2–3.1×. Additionally, we evaluated the effect of this optimization on the split CPU-NIC version of the end-to-end key-value store, whose queues from NIC to CPU

Live ratio	1/5	1/4	1/3	1/2	2/3	3/4	4/5
Live size (B)	64	64	64	64	128	192	256
Total size (B)	320	256	192	128	192	256	320
Speedup	3.1x	2.5x	2x	1.5x	1.3x	1.2x	1.2x

Table 3.4: Speedup when sending only the live portions when varying live ratios from a micro-benchmark. Sizes are in bytes (B).

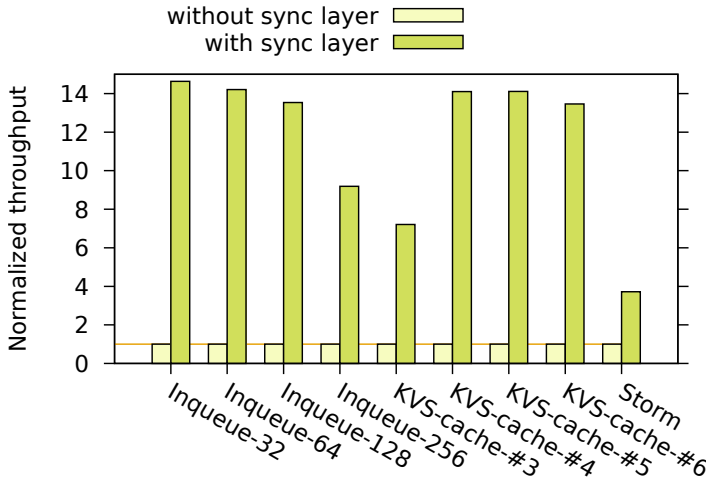


Figure 3.17: Effect of the queue synchronization layer. Throughput is normalized to that without the sync layer.

transfer packets with a live ratio of 1/2. The optimization improved the throughput of this end-to-end application by 6.5%.

Queue synchronization layer

Hypothesis 4 *The queue synchronization layer enables high-throughput communication queues.*

We measured the throughput of three benchmarks. The first benchmark performed a simple packet forwarding from the NIC to CPU with no network activity, so its performance purely reflects the rate of data transfer over the PCIe bus rather than the rate of sending and receiving packets over the network. We used packet sizes of 32, 64, 128, and 256 bytes. The other two benchmarks were the write-back caching version of the key-value store and the redesigned CPU-NIC version of Storm.

Figure 3.17 displays the speedup when using the sync layer versus using primitive blocking DMA operations without batching (labeled "without sync layer"). The sync layer offered 9–15× speedup for pure data transfers in the first benchmark. Smaller packet sizes showed a higher speedup; this is because batching effectiveness increases with the number of packets in a batch. For end-to-end applications, we observed a 7.2–14.1× speedup for the key-value

store and a 3.7× speedup for Storm. Note that the sync layer is always enabled in the other experiments. Hence, it is crucial for performance of our system.

Compiler Overhead

Hypothesis 5 *The* FLOEM *compiler has negligible overhead compared to hand-written code.*

We compared the throughput of code generated from our compiler to hand-optimized programs in C. To measure the compiler's overhead on the CPU, we ran a simple echo program, Storm, and key-value store. The C implementations of Storm and key-value store were taken from FlexStorm and one of FlexKVS's baselines [85]; these implementations are highly-optimized and perform better than the standard public implementations of Storm and memcached. On the NIC, we compared a simple echo program, encryption, flow classification, and network sequencer. On average, the overhead was 9% and 1% on CPU and NIC, respectively. We hypothesize that the higher overhead on the CPU was primarily because we did not implement computation batching [86, 152], which was used for hand-optimized programs.

3.8 Discussion and Future Work

Multi-message packets. FLOEM can support a packet whose payload contains multiple requests via `Batcher` and `Debatcher` elements. Given one input packet, `Debatcher` invokes its one output port n times sequentially, where n is the number of requests in the payload. `Batcher` stores the first $n-1$ packets in its state. Upon receiving the last token, it sends out n packets as one value. The `Debatcher` element can inform the value of n to the `Batcher` element via the per-packet state. One can also take advantage of this feature to support computation batching, similar to Snap [152].

Multi-packet messages and TCP. Exploring the TCP offload with FLOEM is future work. FLOEM supports multi-packet messages via `Batcher` and `Debatcher` elements and could be used together with a TCP offload on the NIC, but our applications do not use TCP.

Shared data structures. In FLOEM, queues and caches are the only high-level abstractions for shared data structures between the NIC and CPU. However, advanced developers can use FLOEM to allocate a memory region on the CPU that the NIC can access via DMA operations, but they are responsible for synchronizing data and managing the memory by themselves.

Automation. Automatic program partitioning was among our initial goals, but we learned that it cannot be done entirely automatically. Different offloading strategies often require program refactoring by rewriting the graph and even graph elements. These program-specific

changes cannot be done automatically by semantics-preserving transformation rules. Therefore, we let programmers control the placement of elements while refactoring the program for a particular offload design. However, FLOEM would benefit from and integrate well with another layer of automation, like an autotuner or a runtime scheduler, that could select parameters for low-level choices (e.g., the number of physical queues, the number of cores, and the placement of each element) after an application has been refactored.

Other SmartNICs. The current FLOEM prototype targets Cavium LiquidIO but can be extensible to other SmartNICs that support C-like programming, such as Mellanox BlueField [104] and Netronome Agilio [6]. However, FPGAs [36, 105, 177] require compilation to a different execution model and the implementation of bodies of elements in a language compatible with the hardware.

3.9 Related Work

Packet processing frameworks. The FLOEM data-flow programming model is inspired by the Click modular router [111], a successful framework for programmable routers, where a network function is composed from reusable elements [111]. SMP Click [40] and RouteBricks [50] extend Click to exploit parallelism on a multi-processor system. Snap [152] and NBA [86] add GPU offloading abstractions to Click, while ClickNP [95] extends Click to support joint CPU-FPGA processing. Dragonet, a system for a network stack design, automatically offloads computations (described in data-flow graphs) to a NIC with fixed hardware functions rather than programmable cores [141, 142].

Other packet processing systems adopt different programming models. PacketShader [70] is among the first to leverage GPUs to accelerate packet processing in software routers. APUNet [58] identifies the PCIe bottleneck between the CPU and GPU and employs an integrated GPU in an APU platform as a packet processing accelerator. Domain-specific languages for data-plane algorithms, including P4 [26] and Domino [144], provide even more limited operations.

Overall, programming abstractions provided by existing packing processing frameworks are insufficient for our target domain, as discussed in Section 3.2.

Synchronous data-flow languages. Synchronous data-flow (SDF) is a data-flow programming model in which computing nodes have statically known input and output rates [91]. StreamIt [157] adopts SDF for programming efficient streaming applications on multicore architectures. Flextream [72] extends StreamIt with dynamic runtime adaptation for better resource utilization. More recently, Lime [73] provides a unified programming language based on SDF for programming heterogeneous computers that feature GPUs and FPGAs. Although some variations of these languages support dynamic input/output rates, they are designed primarily for static flows. As a result, they are not suitable for network applications, where the flow of a packet through a computing graph is highly dynamic.

Systems for heterogeneous computing. Researchers have extensively explored programming abstractions and systems for various application domains on various heterogeneous platforms [23, 29, 102, 113, 119, 132, 133]. FLOEM is unique among these systems because it is designed specifically for data-center network applications in a CPU-NIC environment. In particular, earlier systems were intended for non-streaming or large-grained streaming applications, whose unit of data in a stream (e.g., a matrix or submatrix) is much larger than a packet. Furthermore, most of these systems do not support a processing task that maintains state throughout a stream of data, which is necessary for our domain.

3.10 Conclusions

Developing NIC-accelerated network applications is exceptionally challenging. FLOEM aims to simplify the development of these applications by providing a unified framework to implement an application that is split across the CPU and NIC. It allows developers to quickly explore alternative offload designs by providing programming abstractions to place computation to devices; control mapping of logical queues to physical queues; access fields of a packet without manually marshaling it; cache application state on a NIC; and interface with an external program. Our case studies show that FLOEM simplifies the development of applications that take advantage of a programmable NIC, improving the key-value store's throughput by up to 3.6×.

Chapter 4

GreenThumb: Superoptimization Framework

4.1 Motivation

Code optimization is more important today than ever before. For example, CERN's internal study demonstrated that using a highly optimizing compiler with profile-guided optimizations increased the power efficiency of its data center by 65% [76]. Another study shows that loop optimizations alone improved energy consumption of applications running on battery-operated portable devices by up to 10 times [83]. Code optimizers may also reduce costs of devices by enabling developers to select lower-power computing resources and smaller memory [32].

Developing a code optimizer still remains a challenging problem. The task of implementing a code optimizer is further exacerbated by the development of different instruction set architectures (ISAs) for different types of processors. For example, ARM alone has over 30 variants of ISAs [167], and new architectures are constantly being developed [156, 65, 52, 62, 128, 175, 108]. Even when compiling for widely-used architectures, like x86 or ARM, compilers may miss some optimizations that human experts can recognize. Many of these optimizations are local and very specific to the architectures. Although the expert developers can specify peephole optimizations in the compilers to perform these local rewrites, they may still miss some optimizations, and their rewrite rules may be buggy [101].

Superoptimization, introduced by Massalin [103], is a program optimization technique that *searches* for a correct and optimal program given an optimality criterion, instead of relying on rewrite rules. Thus, a superoptimizer can be used for automatically generating

Materials in this chapter are based on work published as (1) Phothilimthana et al., "Scaling up Superoptimization," in proceedings of ASPLOS 2016 [125], and (2) Phothilimthana et al, "GreenThumb: Superoptimizer Construction Framework," in proceedings of CC 2016 [124].

peephole optimization rules for compilers [19, 63] or optimizing small sequences of instructions produced by compilers on the fly [136, 137, 7, 135]. With this technique, we can avoid buggy human-written rewrite rules and potentially discover even more optimizations. Note that superoptimization subsumes instruction selection, instruction scheduling, and local register allocation. A superoptimizer is shown to optimize a complex multiplication kernel and offer 60% speedup over an optimizing compiler [136].

4.2 Contributions

Scalable Superoptimization

Our first goal is to develop a search technique that can synthesize optimal programs more consistently and faster than existing techniques. We experimented with the most common superoptimization search techniques: symbolic (SAT-solver-based) [148, 160], enumerative [103, 63, 166, 19, 21, 57, 162], and stochastic [136, 137] search. A symbolic search could synthesize arbitrary constants, but it was the slowest. An enumerative search synthesized relatively small programs the fastest, but it could only synthesize up to three ARM instructions within an hour. A sliding window decomposition [121] could scale symbolic and enumerative search to larger programs, but it does not guarantee the optimality of the final output programs. A stochastic search could synthesize larger programs compared to symbolic and enumerative search, but it sometimes could not find the optimal program. This is because a stochastic search can get stuck at local minima.

We develop Lens, an enumerative search algorithm that rapidly prunes away invalid candidate programs. It employs a bidirectional search to prune the search space from both forward and backward directions. It also refines the abstraction under which candidates are considered equivalent selectively via an incremental use of test cases. In our experiment, these pruning techniques increase the number of benchmarks the enumerative search can solve from 11 to 20 (out of 22) and offer 11x reduction on the search time on average.

Although Lens performs better than the existing enumerative algorithms, it still cannot synthesize ARM code with more than five instructions or GreenArrays (GA) [65] code with more than 12 instructions. To scale this search algorithm to synthesize larger code, we introduce a context-aware window decomposition. With this decomposition, our enumerative search can synthesize an optimal (or nearly optimal) ARM program of 16 instructions within 10 minutes.

Optimizing code may require creating new constants or transforming the code fragment globally, which cannot be achieved by the enumerative search with the window decomposition. Thus, we compensate these limitations by combining stochastic and symbolic search into our superoptimizer, yielding a cooperative superoptimizer.

Retargettable Superoptimization

Superoptimization is not commonly used today because implementing a superoptimizer for a new ISA is laborious, and the optimizing process can be slow. First, one must implement a search strategy for finding a candidate program that is optimal and correct on all test inputs, as well as a checker that verifies the equivalence of a candidate program and a reference program when the candidate program passes on all test inputs. The equivalence checker is usually constructed using bounded verification, which requires translating programs into logical formulas. This effort requires debugging potentially complex logical formulas. Second, it is equally, if not more difficult to develop a search technique that scales to program fragments larger than ten or more instructions.

Hence, we develop GREENTHUMB, an extensible framework for constructing superoptimizers. Unlike existing superoptimizers, GREENTHUMB is designed to be easily extended to a new target ISA. Specifically, extending GREENTHUMB to a new ISA involves merely describing the ISA—a program state representation, a functional ISA simulator, and a performance model—and some ISA-specific search utility functions. The framework provides our scalable cooperativesearch strategy that can be reused for any ISA. GREENTHUMB is available at `github.com/mangpo/greenthumb`.

4.3 Overview of Search Strategy and Insights

Figure 4.1 displays the interaction between the LENS algorithm (Section 4.4), the context-aware window decomposition (Section 4.5), and the cooperation of multiple search instances (Section 4.6), which can either employ LENS or different search techniques. The terminology used in this chapter is defined as follows.

- A *program* is a sequence of instructions without loops and branches.
- A *reference program* is a program to be optimized.
- A *program state* contains values in the locations of interest such as registers, stacks, and memory.
- A *test input* is a program state that is used for checking correctness (being equivalent to a reference program).
- A *test output* is an expected program state after executing a candidate program on a given test input. A pair of a test input and a test input constitutes a *test case*.
- An *equivalence verification* is a process to verify if a candidate program is equivalent to a reference program on all inputs using a constraint solver.

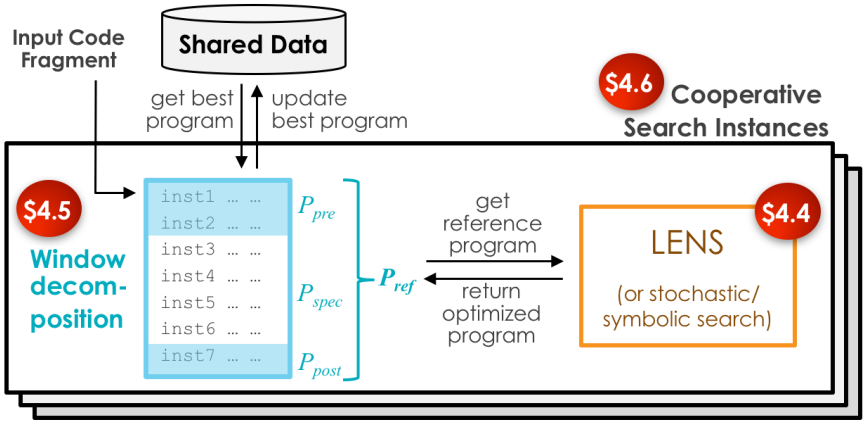


Figure 4.1: Interaction between the main components in our superoptimizer

Search Technique

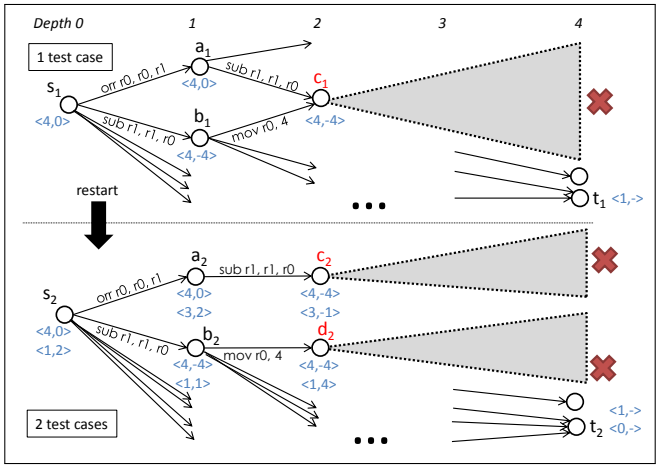
A search technique searches for a program that is semantically equivalent to a reference program but faster according to a given performance model. This section provides our insights on how we design our search technique.

Problem Formulation

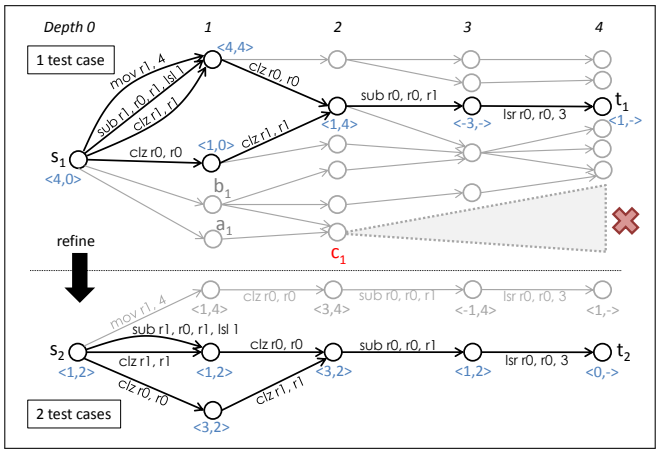
Let p_{spec} be a program we want to optimize. The set of test inputs $I = (i_1, ..., i_n)$ and test outputs $O = (o_1, ..., o_n)$ can be generated. Each test case (i_k, o_k) is an input-output pair such that $p_{spec}(i_k) = o_k$. We formalize the superoptimization problem as a graph search problem. A node u in the graph represents a vector of n program states. The initial node s represents I, and the goal node t represents O. There is an edge from node u — representing program states $(x_1, ..., x_n)$—to node v — representing program states $(y_1, ..., y_n)$ — labeled with an instruction $inst$, if $inst(u) = v$, an abbreviation for $\bigwedge_{i=1}^{n} inst(x_i) = y_i$. We use $u \rightsquigarrow v$ to denote a set of all paths from u to v, which represents a set of instruction sequences. A program that passes all n test cases corresponds to a path from s to t. Therefore, the superoptimization problem reduces to searching for a path p from s to t such that $cost(p) < cost(p_{spec})$. We use $q \oplus r$ to denote concatenation of programs q and r.

Enumerative Search Algorithms

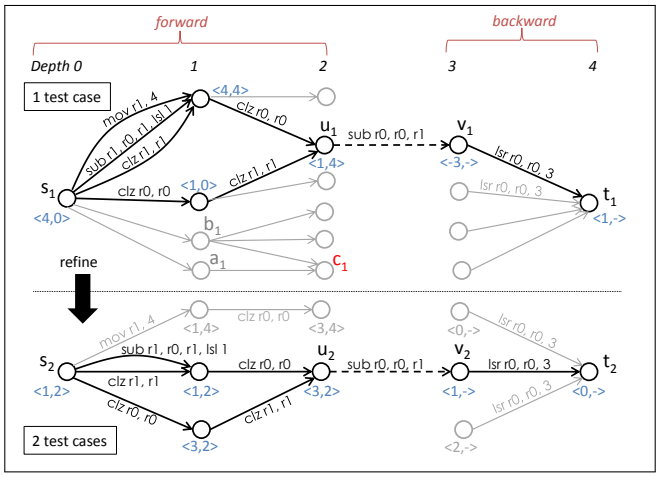
In this section, we illustrate the differences between existing enumerative algorithms and the LENS algorithm. Assume we want to synthesize an ARM program of four instructions using only two registers. A program state is represented by `<r0,r1>`. Figure 4.2 shows the search graphs constructed by different algorithms, which will be explained in detail.



(a) Existing strategy

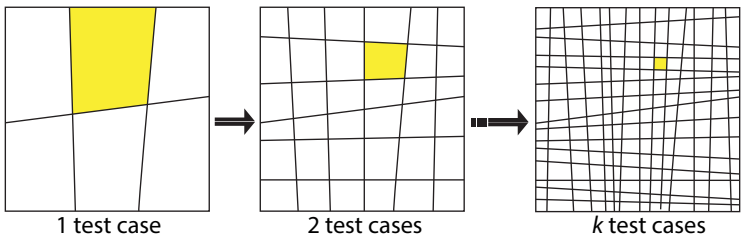


(b) Selective refinement via incremental use of test cases

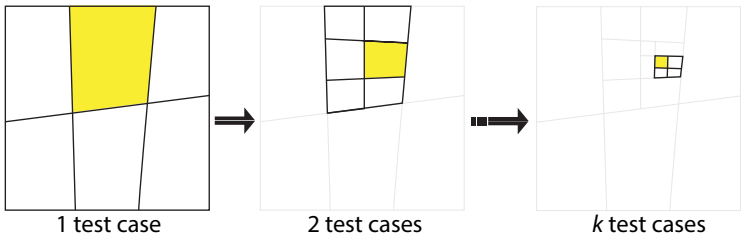


(c) Bidirectional strategy

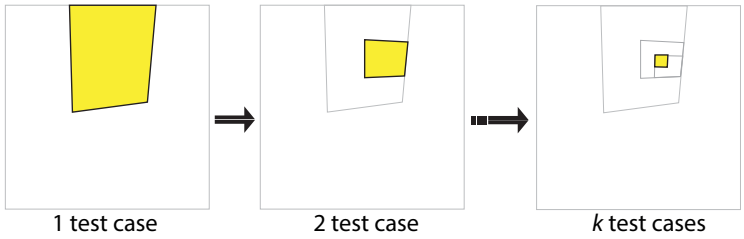
Figure 4.2: Search graphs of ARM programs of length 4. In (b) and (c), the highlighted paths are programs that pass the test cases. Assume programs are executed on 4-bit machine.



(a) Existing strategy



(b) Selective refinement via incremental use of test cases



(c) Bidirectional strategy

Figure 4.3: Division of search space of length d programs. Yellow boxes represent feasible equivalence classes.

Existing Algorithms. Enumerative algorithms enumerate all possible programs whose cost are less than $cost(p_{spec})$ and search for a program that is equivalent to p_{spec}. The existing successful enumerative program synthesizers [21, 11, 162, 57, 10] apply an *equivalence class* concept, grouping programs into equivalence classes based on their behaviors on a set of test inputs. The search enumerates all possible behaviors, which can be many orders of magnitude fewer than all possible programs. Grouping programs based on a set of test cases is effectively abstracting the search space. The fewer the test cases, the more abstract the equivalence classes are; each equivalence class may contain more programs that are, in fact, not equivalent. Node u in the search graph essentially corresponds to the equivalence class containing programs $s \rightsquigarrow u$, which have the same behavior according to the set of inputs I.

The SIMD synthesizer [21] and the SyGus enumerative solver [11] are enumerative synthesizers that solve similar problems to ours. Both synthesizers use equivalence classes in a similar way to prune the search space. Here, we will explain their pruning strategy using our new formulation. Let p be a program prefix from s to u. The algorithm searches for a

program postfix q such that $q(u) = t$. If there is no such q, the search can prune all program prefixes in the same equivalence class as p away. The top part of Figure 4.2(a) illustrates this idea. $s_1 \rightsquigarrow c_1$ corresponds to programs in the same equivalence class. The algorithm only needs to explore the subgraph rooted at c_1 once to prune away all paths from s_1 to c_1.

We observed two main sources of inefficiency in the existing algorithms. The first source of inefficiency comes from restarts. A restart happens when the search finds a *feasible* program, a program that passes the current set of test cases but is not equivalent to p_{spec}; the abstraction is too coarse. The counterexample generated by a constraint solver is added to the test cases to refine the abstraction, and the search restarts building a new graph from scratch with respect to the updated I and O. Upon restarting, the search *forgets* which programs it has already pruned away, so it revisits them again. Figure 4.2(a) illustrates that the search revisits programs from s_1 to c_1 in the new graph. Conceptually, when a new counterexample is found, the algorithm redivides the search space entirely as shown in Figure 4.3(a). The figure visualizes the space of all programs of size d (four in the example in Figure 4.2) divided into equivalence classes.

The second source of inefficiency comes from using more test cases than necessary. Consider programs p_1 and p_2 whose behaviors are the same on the first test case but different on the second one. If there is no q such that $(p_1 \oplus q)(I[1]) = O[1]$ with respect to the first test case, the search can also prune away p_2. However, since p_1 and p_2 are not in the same equivalence class because of the second test case, the search does not prune away p_2. Figure 4.2(a) illustrates that the additional test case splits programs $s_1 \rightsquigarrow c_1$ into two equivalence classes $s_2 \rightsquigarrow c_2$ and $s_2 \rightsquigarrow d_2$, so the search has to traverse the same subgraphs at c_2 and d_2 separately, to find out that both of them cannot reach t_2.

Lens Algorithm. Our enumerative search does not have the aforementioned inefficiencies. It does not restart the search and uses just enough test cases to prune the search space. More specifically, when a counterexample is found, we build a new search graph according to the next test case only on the programs that pass all previous test cases, as shown in Figure 4.2(b). The search graph of test case 2 only includes programs that pass test case 1 (the highlighted paths in the search graph of test case 1). Therefore, we never revisit programs from s_1 to c_1. Conceptually, when we find a counterexample, we refine the search by only subdividing the *feasible* equivalence class, as shown in Figure 4.3(b).

Additionally, we discover that when we search for a program of length d, we can in fact direct the search to a feasible equivalence class without constructing the other equivalence classes of programs of size d, as shown in Figure 4.3(c). This can be achieved through bidirectional search, which builds the search graph from both s and t, as shown in Figure 4.2(c).

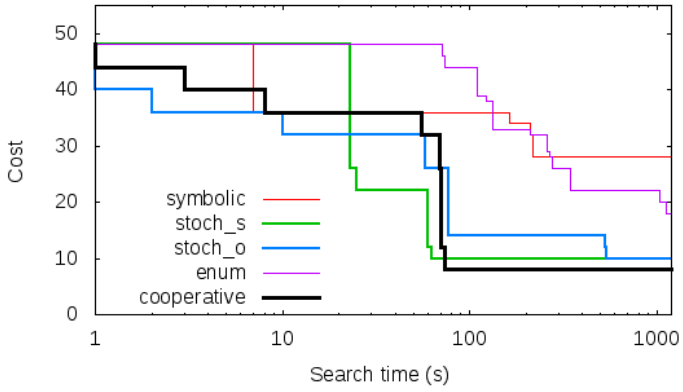
Context-aware window decomposition

A context-aware window decomposition scales search techniques that can solve relatively small problems to larger problems without losing much optimality of the final solutions.

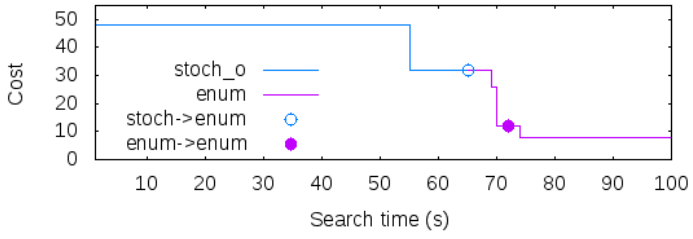
The key idea is to inform the superoptimizer about the precise precondition and postcondition under which the optimized fragment will be executed. We harvest a precondition and postcondition from a context—code surrounding the code to be optimized—and used them to relax the correctness condition. The decomposition selects a random code fragment p_{spec} in a reference program and optimizes the fragment in the context of the prefix p_{pre} and the postfix p_{post} (as depicted in Figure 4.1). This process repeats until none of the fragments in the program can be optimized further. Consequently, this decomposition increases the *effective size* of programs that the superoptimizer can synthesize.

Cooperative search

A cooperative superoptimizer runs multiple search instances of enumerative, stochastic, and symbolic search. The superoptimizer exploits the strengths of all search techniques through communication between search instances, exchanging the best programs they have discovered so far.



(a) Costs of best programs found over time



(b) Trace to the best program found by cooperative search. Circles indicate communications between search instances.

Figure 4.4: Optimizing a sequence of GA instructions from a SHA-256 program. 'stoch_s' is stochastic search that starts from random programs. 'stoch_o' is stochastic search that starts from the correct reference program.

To demonstrate the effectiveness of the cooperative superoptimizer, we show how it optimized a GA code fragment from a SHA-256 program. According to Figure 4.4(a), the cooperative superoptimizer was the only superoptimizer that found the best known code, while being as quick as the stochastic superoptimizer. Although some of the other techniques might seem better at the beginning, the cooperative superoptimizer eventually found the best solution that the other techniques could not; the cooperation costs some overhead but eventually pays off. Note that all supertoptimizers execute the same number of search instances. The detailed descriptions of these five superoptimizers are in Section 4.8.

Figure 4.4(b) depicts how the cooperative superoptimizer arrived at the best solution. A stochastic instance that started mutating from the correct reference program first found a better solution, so it updated the best program shared between the search instances. An enumerative instance took that newly updated program, applied the context-aware window decomposition, and found two better solutions before another enumerative search instance took the latest best program, applied window decomposition, and found the final best program. Our experiment shows that the cooperative superoptimizer increased the number of benchmarks in which the superoptimizer found best known solutions *consistently* from 23 to 29 (out of 32) over using the enumerative search alone. We define a superoptimizer as *consistent* at solving a benchmark if it found best known solutions in all runs.

4.4 The Lens Algorithm

In Section 4.3, we outlined the LENS algorithm's pruning strategies. For the sake of simplification, we assumed that the size of the synthesized program was fixed *a priori*. The complete description provided in this section explains how the algorithm simultaneously grows the program size and refines the search.

Representation of Search Graphs

Each test case (i_k, o_k) is associated with a forward search graph F_k of program prefixes of length ℓ_F, and a backward search graph B_k of program postfixes of length ℓ_B. The root s_k of F_k is labeled with the input i_k, and the root t_k of B_k is labeled with the output o_k. We store $F_1, ..., F_n$ in the nested map M_F such that $M_F[u_1][u_2]...[u_n]$ returns the set of programs p of length ℓ_F such that $p(i_1) = u_1, p(i_2) = u_2, ..., p(i_n) = u_n$. For example, in the search graphs in Figure 4.2(c), $M_F[\langle 1, 4\rangle][\langle 3, 2\rangle]$ maps to three programs:

1. `sub r1, r0, rl, lsl 1`
2. `clz r1, r1; clz r0, r0`
3. `clz r0, r0; clz r1, r1`

We use $Progs(M_F)$ to refer to all programs stored inside M_F.

Algorithm 2 Main search

```
1: n ← 1                                                    ▷ Number of test cases
2: ℓ_F ← 0, ℓ_B ← 0
3: p_spec^ ← ReduceBitwidth(p_spec)
4: cost ← cost(p_spec)
5: (I, O) ← GenTest(p_spec^)
6: M_F ← Init(I), M_B ← Goal(O)
7: while true do
8:     for all inst ∈ Insts do                              ▷ Searching Phase
9:         (M_F, M_B) ← Connect(M_F, M_B, inst, 1)
10:    if Forward?(ℓ_F, ℓ_B) then                           ▷ Expanding Phase
11:        M_F ← ExpandForward(M_F), ℓ_F ← ℓ_F + 1
12:    else
13:        M_B ← ExpandBackward(M_B), ℓ_B ← ℓ_B + 1
```

The backward search graphs are stored differently, but to simplify the explanation of our algorithm, let us assume that the backward search graph offers the same interface; there is a map M_B such that $M_B[u_1][u_2]\dots[u_n]$ returns the set of programs p of length ℓ_B such that $p(u_1) = o_1, p(u_2) = o_2, ..., p(u_n) = o_n$. Our efficient implementation of the backward search graphs is described later in this section.

The Algorithm

Algorithm 2 displays our main algorithm. We first create one test case. Therefore, at the beginning, we start the search from F_1 containing only s_1, and B_1 containing only t_1. Then, the main loop performs two actions—search and expand—in each iteration. The search phase searches for programs of size $\ell_F + \ell_B + 1$ that pass all test cases. When the search phase is complete, the expand phase increases the size of programs we will be searching in the next iteration by one. This process repeats until timeout.

The expanding phase (on line 10–13) increases the size of programs by expanding all leaf nodes of either F_1 or B_1. *Forward?* is a heuristic function that decides whether to expand forward or backward. In particular, we expand each leaf node u in F_1 by adding $u \xrightarrow{inst} v$ for all $inst \in Insts$, where *Insts* is a set of all possible instructions. Similarly, we expand each leaf node v in B_1 backward by adding $u \xrightarrow{inst} v$ for all $inst \in Insts$.

The searching phase (on line 7–9) find programs that pass all n test cases by finding an instruction that can connect leaf nodes in $F_1, ..., F_n$ to leaf nodes in $B_1, ..., B_n$ respectively. The main algorithm calls *Connect* to find such programs. $Connect(M_F, M_B, inst, k)$, shown in Algorithm 3, searches for programs in $Progs(M_F) \oplus inst \oplus Progs(M_B)$ that pass test cases k to n. It maintains the invariant that all programs in $Progs(M_F) \oplus inst \oplus Progs(M_B)$ pass all test cases 1 to $k-1$. This invariant is the key to refining the search only on a feasible equivalence class.

After F_k and B_k are built, the loop on lines 12–15 searches for a leaf node u in F_k and v in B_k that can be connected by *inst*. $keys(M_F)$ and $keys(M_B)$ on line 12 and 14 are sets

Algorithm 3 Connect and refine

Global variables: $I, O, cost, n, p_{spec}, \hat{p_{spec}}$

```
 1: function CONNECT(M_F, M_B, inst, k)
 2:     if k > n then                                          ▷ Pass all test cases
 3:         for all p ∈ M_F, p' ∈ M_B do                       ▷ M_F, M_B are sets of programs
 4:             if cost(p ⊕ inst ⊕ p') < cost then
 5:                 Verify(p ⊕ inst ⊕ p')
 6:                                                            ▷ Build search graph on test case k
 7:     if M_F is not a map then                               ▷ M_F is a set of programs
 8:         M_F ← BuildForward(M_F, I[k])
 9:     if M_B is not a map then                               ▷ M_B is a set of programs
10:         M_B ← BuildBackward(M_B, O[k])
11:
12:     for all u ∈ keys(M_F) do                               ▷ Search for a connection
13:         v ← inst(u)
14:         if v ∈ keys(M_B) then                              ▷ Find a connection, so refine the search
15:             (M_F[u], M_B[v]) ← Connect(M_F[u], M_B[v], inst, k + 1)
16:     return (M_F, M_B)

17: function VERIFY(p̂)
18:     if p̂ ≡ p_spec then                                     ▷ Check via a constraint solver
19:         for all p ∈ IncreaseBitwidth(p̂) do
20:             if p ≡ p_spec then                             ▷ Found a better program!
21:                 cost ← cost(p)
22:                 yield p
23:     else
24:         n ← n + 1
25:         (I[n], O[n]) ← CounterExample(p_spec, p̂)
```

of leaf nodes in F_k and B_k. If $inst$ can connect u to v, programs in $Progs(M_F[u]) \oplus inst \oplus Progs(M_B[v])$ pass test case k, so the algorithm refines the search on $Progs(M_F[u]) \oplus inst \oplus Progs(M_B[v])$ with the next test case $k+1$. For our running example in Figure 4.2(c), we find an instruction `sub r0, r0, r1` connecting u_1 and v_1 of test case 1, so we refine the search on the highlighted programs $s_1 \rightsquigarrow u_1 \rightarrow v_1 \rightsquigarrow t_1$.

When we recursively call *Connect*, M_F will eventually become a set of programs instead of a nested map, as well as M_B. Lines 7–10 take care of building F_k for programs in M_F and B_k for programs in M_B. F_k and B_k for each k are built once and saved on line 15 to be used later when *Connect* is called with different *inst*s. If there are no more test case left, lines 2–5 verify all programs in $M_F \oplus inst \oplus M_B$ against the reference program. *Verify* function performs equivalence verification. If the two programs are not equivalent, an counterexample is added to I and O on line 25. If they are equivalent, the algorithm yields the candidate program and continues searching for solutions with lower costs until timeout.

Implementation Details

Challenges of Backward Search

We have identified two main challenges in implementing backward search in a program synthesizer. First, the synthesizer needs to evaluate an instruction backward; it needs an inverse function for every instruction. Second, in the forward direction, an instruction *inst* is a one-to-one function that map a state u to v. In contrast, in the backward direction, *inst* is a one-to-many function that map the state v to a set of states, one of which is u.

Fortunately, we can mitigate these challenges by reducing bitwidth, using only four bits to represent a value. First, we can avoid implementing an inverse emulator by constructing an inverse function table for every instruction. We execute every instruction on all possible combinations of 4-bit input arguments' values and memorize them in the inverse table. Second, the small bitwidth also reduces the number of states an instruction can transition to in the backward direction. For example, in 32-bit domain, an inverse instruction *add* transitions from one state to 2^{32} states; in contrast, in 4-bit domain, the same instruction only transitions to 2^4 states.

Reduced Bitwidth

Let *bit* be the actual bitwidth and $\hat{bit}$ be the reduced bitwidth, which is four in our case. The reduced bitwidth not only enables the backward search but also allows us to initially divide the search space more coarsely, which is desirable because the search graph even for a single test can be very large. For example, an ISA with four 32-bit registers can have $2^{32\times4}$ states and, hence, up to $2^{32\times4}$ nodes in the graph.

Apart from the second-step equivalence verification (line 20 of Algorithm 3), the search algorithm operates in the reduce-bitwidth domain. Therefore, we need both reduced-bitwitdh and precise versions of a program state and an ISA emulator. We implement an emulator that can be parameterized by bitwidth to instantiate both versions. For example, the precise ARM emulator interprets instruction `movt r0, 1` as writing 1 to the top 16 bits of a 32-bit register. The 4-bit ARM emulator should interpret the same instruction as writing 1 to the top 2 bits of a 4-bit register. Implementing a parameterizable program state is simple. We just need to use a specified number of bits to represent each entry in a program state.

Additionally, we must have a way to convert programs between the two domains. In particular, at the beginning, we convert the reference program p_{ref} from the precise domain to the reduced-bitwidth domain (line 3 in Algorithm 2) by replacing constants appearing in the program with their reduced-bitwidth counterparts. We replace a constant c using the following function α:

$$\hat{c} = \alpha(c) = \begin{cases} \hat{bit} & \text{if } shift?(c) \wedge (c = bit) \\ \hat{bit} - 1 & \text{if } shift?(c) \wedge (c = bit - 1) \\ \hat{bit}/2 & \text{if } shift?(c) \wedge (bit/2 \leq c < bit - 1) \\ 1 & \text{if } shift?(c) \wedge (1 < c < bit/2) \\ c \bmod 2^{\hat{bit}} & \text{otherwise} \end{cases}$$

where $shift?(c)$ checks if c is a shift operand. α is designed to preserve semantics of shift operations in a meaningful way. For example, it translates shift by 31 in 32-bit domain to shift by 3 in 4-bit domain. Apart from shift constants, α simply masks in the lowest $\hat{bit}$ bits.

During this conversion, we memorize every replacement of c with $\hat{c}$, so that we can map each reduced-bitwidth constants back to the set of original constants to obtain candidate programs in the precise domain. We construct the replacement map γ by storing $\gamma[\hat{c}] \leftarrow \gamma[\hat{c}] \cup \{c\}$ for every constant c in p_{ref}. Before the precise equivalence verification, the reduced-bitwidth constant $\hat{c}$ is replaced with every constant in the set $\gamma[\hat{c}]$ (line 19 in Algorithm 3) with the expectation that one of them will lead to a correct solution.

We are able to optimize many bitwidth-sensitive programs (e.g. population count and computing higher-order half of multiplication) using this reduced-bitwidth trick.

Data Structure for Backward Search Graph

We could store backward search graphs the same way we store forward search graphs. However, it would require a large amount of memory because in the backward direction, an instruction is a one-to-many function; one program postfix can appear in a large number of backward equivalence classes. Instead of using a nested map to store all backward search graphs, we construct n separate maps to store n backward search graphs $B_1, ..., B_n$. We can find a program postfix p such that $p(u_1) = o_1, ..., p(u_n) = o_n$, by looking up $Y[u_1] \cap ... \cap Y[u_n]$. The pseudocode in Algorithm 2 and Algorithm 3 has to be modified slightly to support this data structure.

Existing Backward Search in Superoptimization

We have known of one superoptimization work by Bansel et al. that also applies backward search strategy [18]. Their key concept of *meet-in-the-middle* strategy is similar to that of our algorithm. However, the implementation details differ significantly. One major difference is that in LENS, an inverse function produces a set of all possible output program states, and we look for an exact match between forward and backward program states in the reduced bitwidth domain. In contrast, their inverse function outputs one program state with *don't-know* bits in the original domain, and they look for a match such that a concrete bit in a forward program state can match to either the same concrete value or a don't-know value of the same location in a backward program state. An advantage of their strategy is that the number of backward program states can be much smaller than ours. However, if a program

state contains many don't-know bits, they may have many more program candidates that cannot be pruned away because of the imprecision of don't-know bits.

In our experiment, we observe that for all programs for which LENS without the backward search can synthesize an optimal solution under 20 minutes, LENS with the backward search always finds an optimal solution with less time, 5.2x faster on average. However, Bansal et al. observe a slowdown in synthesis time when using the backward search for 37% of their benchmarks. They comment that the slowdown usually happens when there are many don't-know bits.

4.5 Context-Aware Window Decomposition

We can scale a search technique that can synthesize relatively small programs to optimize larger programs using a decomposition. Let p_{ref} be a large program to be optimized, and L be a window size. We can decompose p_{ref} into $p_{pre} \oplus p_{spec} \oplus p_{post}$ such that $length(p_{spec}) < L$, and optimize p_{spec}, the code inside the window. Peephole optimizations will try to optimized p_{spec} alone, or in the best scenario, with a precondition that is often not precise. The precondition and postcondition relax the correctness condition and provide invariants that may be exploited by the search. Therefore, we believe that optimizing p_{spec} with the most precise precondition and postcondition, essentially in the context of its prefix p_{pre} and postfix p_{post}, can lead to finding a better program. We call this decomposition a *context-aware window decomposition*. In our implementation, we pick a random position of the window and optimize the program. This process repeats until we cannot optimize the program at any window's position anymore.

To support the context-aware decomposition, we need to modify search algorithms slightly. Note that any search technique can be modified to be context-aware. Recall that a search technique looks for a program p such that for each $i \in I, o \in O \,.\, p(i) = o$. To make the search context-aware, we do not need to change this search routine, but only need to adjust the equivalence condition used during equivalence verification and the way test cases are updated as shown in Table 4.1. Normally, when we find p that passes all test cases, we uses a constraint solver to verify if $p_{spec} \equiv p$. If they are not equivalent, the constraint solver will return an input counterexample i_{ce}, which we use to update the test inputs I and test outputs O as shown in Column 'non-context-aware'. Then, the search continues to find a new candidate program, and so on. To make the search context-aware, we ask the constraint solver if p is equivalent to p_{spec} in the context of p_{pre} and p_{post}, in particular if $p_{pre} \oplus p_{spec} \oplus p_{post} \equiv p_{pre} \oplus p \oplus p_{post}$. If they are not equivalent, the constraint solver will return i_{ce}, which is an input to p_{pre} (not directly to p), so we have to update the test cases differently as shown in Column 'context-aware'.

Routine	Non-context-aware	Context-aware
Equivalence verification	$p_{spec} \equiv p$	$p_{pre} \oplus p_{spec} \oplus p_{post} \equiv p_{pre} \oplus p \oplus p_{post}$
Input text-cases update	$I = I \cup \{i_{ce}\}$	$I = I \cup \{p_{pre}(i_{ce})\}$
Output text-cases update	$O = O \cup \{p_{spec}(i_{ce})\}$	$O = O \cup \{p_{pre} \oplus p_{spec}(i_{ce})\}$

Table 4.1: The differences between non-context-aware and context-aware decomposition. p is a candidate program. i_{ce} is the input counterexample returned by the constraint solver if the candidate program is not equivalent to the reference program.

Concrete Example

Assume we want to optimize the following ARM program:

```
P_pre:  cmp r3, r4
        moveq r1, #0       // mov if r3 = r4
        movne r1, #1       // mov if r3 != r4
P_spec: cmp r2, #31
        movhi r1, #0       // mov if r2 > 31
        andls r1, r1, #1   // and if r2 <= 31
```

The decomposition selects the window as labeled; p_{post} is empty. Without p_{pre}, p_{spec} cannot be improved because no faster code modifies `r1` as p_{spec} does. With p_{pre}, however, the superoptimizer learns that the value of `r1` is either 0 or 1 at the beginning of p_{spec}, so the last instruction `r1 = r1 & 1` does not have any effect. Thus, the superoptimizer can simply remove it. Note that we do not have to explicitly infer this precondition of p_{spec}. It is implicit, captured by running p_{pre} along with p_{spec} during test case evaluations and equivalence verification. We also find that p_{post} helps the superoptimizer discover faster code.

4.6 Cooperative Superoptimizer

To utilize strengths of different search techniques, we introduce a cooperative superoptimizer that combines all search techniques in a simple fashion. The cooperative superoptimizer launches all search techniques in parallel. Each search instance executes one of the three following state-of-the-art search techniques.

Symbolic search (SM). Our symbolic search exploits an SMT solver to perform the search. The search problem is written as a logical formula whose symbolic variables encode the choices of the program p. The formula embeds the ISA semantics and ensures that the program p computes an output o given an input i. Using Rosette [160], we obtain the symbolic search for free without having to implement a translator for converting programs to SMT formulas and vice versa. Compared to the other two algorithms, the symbolic search is slow, but it is able to synthesize arbitrary constants, which are sometimes needed in optimal code.

Enumerative search (E). Our enumerative search implements the LENS algorithm (Section 4.4). The enumerative search synthesizes relatively small programs the fastest, but it does not scale to large programs by itself.

Stochastic search (ST). Our stochastic search explores the search space through a random walk using Metropolis Hastings acceptance probability with a cost function that reflects the correctness and performance of candidate programs [136, 137]. The stochastic search can synthesize larger programs compared to the symbolic and enumerative search because of the guidance of the cost function. However, it sometimes misses optimal programs because it can get stuck in local minima.

There are two modes of search. Synthesize mode (s) is when a search does not use a starting correct program except for equivalence verification. Optimize mode (o) is when a search uses a starting correct program beyond equivalence verification. The table below summarizes the types of search instances we use.

Search Instance	Description
E^s	enumerative on entire code fragment
E^o	enumeratvie with decomposition
SM^s	symbolic on entire code fragment
SM^o	symbolic with decomposition
ST^s	stochastic that starts from a random program
ST^o	stochastic that starts from the input correct program

Communication between Search Instances

The search instances aid each other by exchanging information about the current best solution equivalent to p_{ref}. When a search instance finds a new best program, it updates the shared best solution p_{best}. The other search instances may obtain p_{best} to aid in their search processes. In particular, different types of search techniques utilize p_{best} as follows:

- SM^s does not use p_{best}.
- E^o and SM^o apply the context-aware window decomposition on p_{best}.
- ST^s reduce its search space by only exploring programs with up to $length(p_{best})$ instructions.
- ST^o restarts the search from p_{best}. In practice, it is better to allow some divergence among stochastic instances. Therefore, our stochastic instances check p_{best} every 10,000 mutations and restart the search from p_{best} only if $cost(p_{best})$ is much less than the cost of the local best solution; in our implementation, we restart when the difference is more than 5.
- E^s, E^o, ST^s, and ST^o harvest new constants from p_{best} and include them into its list of constants to try. These new constants come from SM.

Practical Configuration of Search Instances

We present a configuration for allocating search instances that worked well in our experiment; however, it might not be optimal. Our cooperative superoptimizer executes N search instances with the following distributions: $N/2 - 1$ E^o, one E^s, two ST^s, three ST^o, and the rest for SM^o. We dedicate almost half of the threads for enumerative search because it performs the best in most benchmarks (see Section 4.8). Multiple enumerative instances attempt to optimize different parts of the program at the same time, reducing overall search time to find a final solution. We allocate one thread for E^s because if the size of the final solution is small, E^s will find an optimal solution quickly. A few ST^s instances are allocated because they can perform very well on some benchmarks on which E performs poorly. We also dedicate a few threads for ST^o instances because they often help E^o instances reach the final solution faster. Finally, we allocate the rest of the resources to SM^o, which helps discover optimizations that involve synthesizing arbitrary constants. For search instances that use the window decomposition, we use four sizes of window L, $2L$, $3L$, and $4L$, where L is a constant specific to the ISA.

4.7 Superoptimization Construction Framework

We develop GREENTHUMB, a framework for building superoptimizers for different ISAs and testing search techniques. Figure 4.5 depicts the major components of GREENTHUMB and their interactions. At the core is the cooperative search algorithm (Section 4.6), which launches parallel search instances running different search techniques. Each instance consists of multiple components. First, the encoder-decoder parses a program into an IR. It is also used to print output optimized programs to files. On large programs, performs a context-

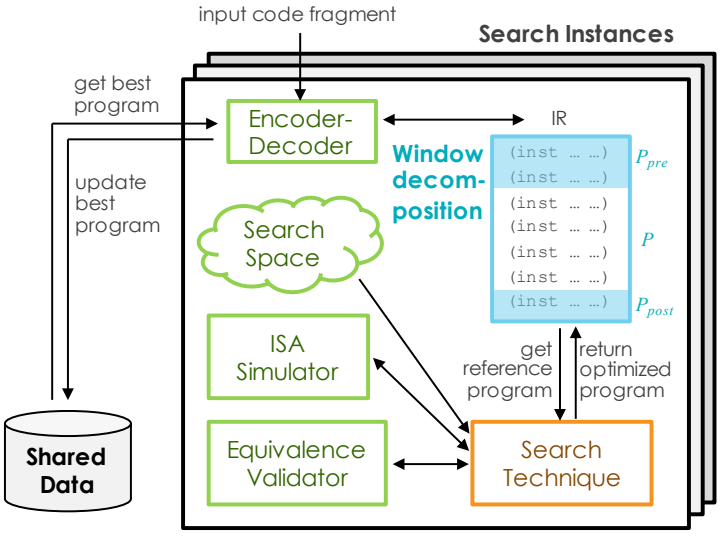


Figure 4.5: Major components in GREENTHUMB

aware window decomposition (Section 4.5) and uses a search technique to optimize a fragment p in the context of prefix p_{pre} and postfix p_{post}. An ISA simulator evaluates the correctness of a candidate program on concrete test cases. If a candidate passes all test cases, the equivalence validator verifies the candidate program against the reference program on all inputs using a constraint solver. If they are equivalent, and the candidate program is better than the current best program, the search instance updates the shared data. If they are not equivalent, the counterexample input is added to the set of concrete test cases.

The detailed documentation on how to extend GREENTHUMB to a new ISA can be found on `https://github.com/mangpo/greenthumb`. We have used GREENTHUMB to build superoptimizers for ARM, GreenArrays (GA), and a subset of LLVM. Although ARM and GA are drastically different, our framework is able to support both ISAs. This demonstrates the retargetability of our framework.

ARM is a widely-used RISC architecture. An ARM program state includes 32-bit registers, memory, and condition flags. We extended GREENTHUMB for ARMv7-A and modeled the performance based on ARM Cortex-A9 [16]. An ARM program state includes 32-bit registers, memory, and condition flags. The smallest window size L is set to 2. Recall that there are four sizes of window L, $2L$, $3L$, and $4L$.

GreenArrays GA144 is a low-power processor, composed of many small cores [65]. The program state for GA includes registers, stacks, memory, and a communication channel, similar to the one used in the superoptimizer in CHLOROPHYLL. A communication channel is an ordered list of (data, neighbor port, read/write) tuples representing the data that the core receives and sends. For two programs to be equivalent, their communication channels have to be identical. We set the smallest window size L to 7.

Limitations

The performance cost is the sum of average latency of instructions in a program for both ISAs. We do not model memory access latency variations caused by misses at different levels of caches. We assign the same cost to all loads and stores. Therefore, our performance model is imprecise; as a result the superoptimizer may output a program that is actually slower than other candidates it has explored. To work around this problem, the superoptimizer can output the best K programs instead of only the best one. This way, we can try running all of them on the real machine and select the fastest one empirically.

The second limitation is that the superoptimizer can only optimize code without loops and branches. In order to optimize across multiple basic blocks with loops and branches, we will need to modify the superoptimizer.

Framework Development

GREENTHUMB 1.0 was released in March 2016. However, it still requires a considerable amount of efforts to (1) define the description of a new ISA — for example, the program state structure, types of instructions, types of instruction operands, the parser, the printer, etc — and (2) implement an inverse interpreter used in the LENS algorithm. Thus, we improved the framework and released GREENTHUMB 2.0 in October 2016. Shortly after the release, Linki Tools — a compary specialized in building development tools and compilers for programming embeded system devices — expressed their interest in using GREENTHUMB to build superoptimizers. The company had experimented using GREENTHUMB to build a superoptimizer for RISC-V. Finally, they decided to build their own commercial superoptimization framework, S10 [99], from the resurgence of GREENTHUMB. The system implementation of S10 is more modular, more maintainable, and more performance. On top of the features GREENTHUMB provides, S10 supports:

- a DSL to describe an ISA
- running search on multiple machines
- big-endian/little-endian accessing and byte addressing
- unit tests, integration tests, randomized tests, and fuzzing tests

The company used S10 to build superoptimizers for many variants of RISC-V and is in the process of developing superoptimizers for x86 and ARM.

4.8 Evaluation

The key result in this chapter is that we improve on the state of the art in superoptimization, represented by STOKE [136, 137], the stochastic superoptimizer. On large benchmarks, our implementation of STOKE produced ARM programs of length 10–27 and GA programs of length 18–32. Our cooperative superoptimizer optimized the benchmarks faster (12x faster on average) and obtained better solutions (the performance cost of our code is, on average, 18% lower than that of stochastic search).

We implemented all search techniques as well as ARM and GA emulators in GREENTHUMB using Racket. Since all search techniques are implemented in the same language and using the same emulator, we can compare them fairly.

This section presents detailed evaluation of our algorithms, starting from the new enumerative algorithm, using the following benchmark suites.

ARM Hacker's Delight Benchmarks consist of 16 of the 25 programs identified by [67] drawn from Hacker's Delight [165]. We excluded the first nine programs from our set of benchmarks because they are very small. We used code produced by `gcc -march=armv7-a`

`-O0` as the input programs to the superoptimizers. Their sizes ranged from 16 to 60 instructions. The timeout was set to one hour.

GA Benchmarks consist of frequently-executed basic blocks from MD5, SHA-256, FIR, sine, and cosine functions from the CHLOROPHYLL compiler's benchmarks. We used CHLOROPHYLL without superoptimization to generate these basic blocks. The sizes of the input programs in this benchmark suite ranged from 10 to 56 instructions. The timeout was set to 20 minutes.

Experiment I: Evaluating the Lens algorithm

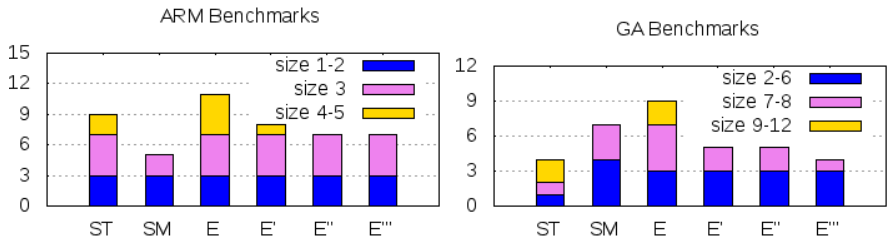
Experiment I is designed to evaluate the base search techniques: SM^s, E^s, and ST^s. Recall that superscript ^{s} indicates synthesize mode (no window decomposition). This experiment will help us answer which search technique is a suitable building block for a superoptimizer with window decomposition. For each benchmark, we ran each search technique on a single thread 16 times.

Hypothesis A *Enumerative search is faster and can solve larger benchmarks than the other base search techniques.*

E^s is superior in terms of speed and scalability: it was the fastest search; it solved all except two benchmarks; and it could solve larger benchmarks than other synthesizers. Regarding consistency—which is desirable because it obviates the need for redundant instances, improving the chances of finding optimal solutions—almost all of E^s 16 search instances found optimal solutions in each of its solved benchmarks. Note that there is small amount of randomness in the enumerative search because the initial test cases are generated randomly.

There were a total of 22 benchmarks in which one of the search techniques found optimal solutions in at least one of the 16 runs. Columns SM^s, E^s, and ST^s of Figure 4.6 summarize the results. Figure 4.6(a) displays the number of benchmarks *solved* by each search technique, categorized by size. A search technique *solved* a benchmark if it an optimal solution in one of its run. Row 'benchmarks' in Figure 4.6(b) summarizes the numbers of solved benchmarks. Row 'instances' displays the average numbers of search instances that found optimal solutions per solved benchmark. In terms of search time, we evaluated each search technique against E^s by comparing the best runs of the benchmarks they both solved. Row 'E^s speedup' shows how much faster E^s was on average compared to a particular search technique.

According to Figure 4.6(a), E^s could synthesize larger ARM programs than ST^s and SM^s could. For GA, E^s could synthesize larger programs than SM^s could. While E^s and ST^s were comparable at synthesizing large GA programs, ST^s was much worse at synthesizing smaller GA programs. This might be because the cost function of ST^s does not fit well with these GA benchmarks, or the mutations we have are not the best for GA. Interestingly, the largest GA benchmark, which E^s failed to solve, was solved by ST^s. This result suggests that sometimes the cost function can be very effective in guiding the search in some particular



(a) Number of solved benchmarks, categorized by size

Solved	ST^s	SM^s	E^s	$E^{s\prime}$	$E^{s\prime\prime}$	$E^{s\prime\prime\prime}$
Benchmarks	13	12	20	13	12	11
Instances	7.2	13.5	14.9	15.8	15.5	15.9
E^s speedup	14x	52x	1x	2.7x	5.2x	11x

(b) Total number of solved benchmarks, average number of instances per solved benchmark, and search time speedup by E^s

Figure 4.6: Comparing base search techniques

problems. Another benchmark that E^s failed to solve can be solved by SM^s. This is because the optimal program contains a constant not included in the pre-defined constant list of E^s and ST^s.

Hypothesis B *The* LENS *algorithm improves on the existing enumerative algorithms.*

With the same experimental setting, we compared multiple versions of enumerative search:

- E^s: LENS with all pruning strategies
- $E^{s\prime}$: E^s without backward search (unidirectional search)
- $E^{s\prime\prime}$: $E^{s\prime}$ without reduced-bitwidth trick
- $E^{s\prime\prime\prime}$: $E^{s\prime\prime}$ without refinement through incremental test cases. $E^{s\prime\prime\prime}$ represents the existing enumerative search but without the stack representation [21].

Columns E^s–$E^{s\prime\prime\prime}$ of Figure 4.6 summarizes the results. The pruning strategies we introduce not only increase the size of code an enumerative search can solve but also speed up the search; E^s was, on average, 11x faster than $E^{s\prime\prime\prime}$.

Experiment II: Evaluating window decomposition

Experiment II is designed to test the effectiveness of context-aware window decomposition. We test E^o, which is context-aware, against a modified version of E^o, which is not context-aware, on the 12 benchmarks that E^s cannot synthesize optimal solutions from the previous experiment. Recall that the superscript ^{o} indicates optimize mode (see Section 4.6). On

ARM benchmarks, we ran a superoptimizer using 32 E^o search instances on a 16-core hyper-threaded machine. On GA benchmarks, we ran 16 search instances on a 16-core Amazon EC2 machine. For each benchmark, we repeated the experiment three times.

Hypothesis C *The context-aware window decomposition technique enables the enumerative search to find better code than does the non-context-aware window decomposition.*

Considering the best out of the three runs, in six benchmarks, the context-aware decomposition found solutions with 1.3x–3x lower cost than did the non-context-aware decomposition. In the rest, both of them found solutions with the same costs.

Experiment III: Evaluating cooperative search

Experiment III is designed to evaluate superoptimizers based on different search techniques with context-aware window decomposition. We use the same experimental set up as in Experiment II. We evaluate the following five versions of superoptimizers, each of which runs N search instances ($N = 32$ for ARM, and $N = 16$ for GA).

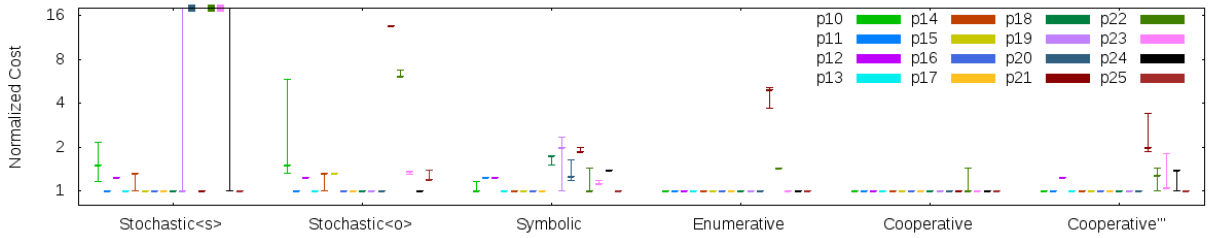
Superoptimizer	Search instances use:
$\widetilde{ST}^s$	all ST^s instances with no communication
$\widetilde{ST}^o$	all ST^o instances with no communication
$\widetilde{SM}$	one SM^s instance, $N-1$ SM^o instances
$\widetilde{E}$	one E^s instance, $N-1$ E^o instances
$\widetilde{C}$	one E^s, $N/2-1$ E^o, two ST^s, three ST^o, and the rest for SM^o instances

Search instances of each superoptimizer communicate with each other except in $\widetilde{ST}^s$ and $\widetilde{ST}^o$, which represent STOKE implemented in our framework. In $\widetilde{E}$, $\widetilde{SM}$, and $\widetilde{C}$, we add one instance of an enumerative or symbolic search in synthesize mode (E^s or SM^s) because these instances can find an optimal solution should the optimal solution be small.

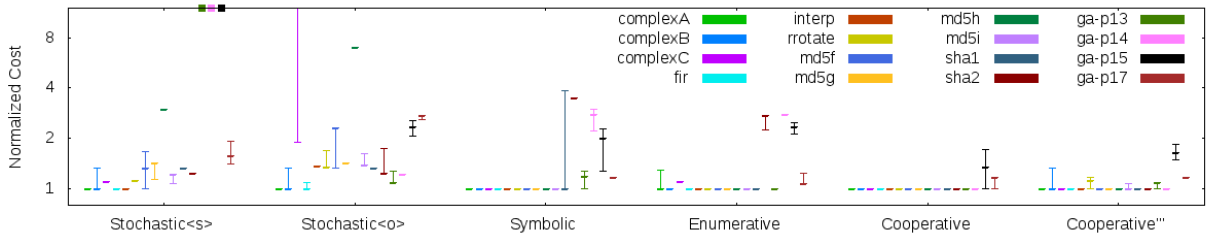
Hypothesis D *The enumerative superoptimizer can often synthesize best known programs more consistently and faster than the stochastic and symbolic superoptimizers.*

$\widetilde{E}$ was *consistent* on 2.1x, 2.6x, and 1.4x more benchmarks than $\widetilde{ST}^s$, $\widetilde{ST}^o$, and $\widetilde{SM}$. We define a superoptimizer as *consistent* at solving a benchmark if it found programs as optimal as the best known solution in all runs. Consistency is desirable because in practice we want to find the best program in one run not multiple runs. Then, we did a pair-wise comparison of the median search time between $\widetilde{E}$ and each of the other superoptimizers on the benchmarks they both solved consistently. We found that $\widetilde{E}$ was also on average 9x, 4.6x, and 14x faster than $\widetilde{ST}^s$, $\widetilde{ST}^o$, and $\widetilde{SM}$.

Figure 4.7 shows the performance costs of the best correct programs found in each of three runs; the lower the better. The reported costs of each benchmarks are normalized by the cost of the best known program of that particular benchmark. Table 4.2 reports the median time to finding the best known solutions for the various superoptimizers. If a



(a) ARM Hacker's Delight Benchmarks



(b) GA Benchmarks

Figure 4.7: Costs of best programs found by the different superoptimizers (normalized by the cost of the best known program). A **dash** represents the cost of the best program found in one run. A dash may represent more than one run if the best programs found in different runs have the same cost. If one or two runs did not find any correct program that is better than the input program, the vertical line is extended past the chart. If none of the runs found a correct program that is better than the input program, a **rectangle** is placed at the top of of the chart.

(a) ARM Hacker's Delight Benchmarks

Prog	$\widetilde{ST}^s$	$\widetilde{ST}^o$	$\widetilde{SM}$	$\widetilde{E}$	$\widetilde{C}$	$\widetilde{C'''}$
p10	-	-	-	145	88	188
p11	244	188	-	49	92	1171
p12	-	-	-	566	646	-
p13	13	6	85	3	3	2
p14	-	-	755	19	11	9
p15	837	-	591	26	8	8
p16	5	5	83	-	7	6
p17	15	12	82	11	6	72
p18	21	38	-	7	9	89
p19	-	21	-	76	36	49
p20	-	254	-	129	113	365
p21	1316	-	-	-	1139	-
p22	-	-	-	-	-	-
p23	-	-	-	707	665	-
p24	-	1440	-	73	151	-
p25	72	-	47	2	2	1

(b) GA Benchmarks

Prog	$\widetilde{ST}^s$	$\widetilde{ST}^o$	$\widetilde{SM}$	$\widetilde{E}$	$\widetilde{C}$	$\widetilde{C'''}$
complexA	45	258	136	-	72	63
complexB	-	-	186	43	52	-
complexC	-	-	7	-	21	17
fir	7	-	501	153	23	63
interp	119	-	109	12	7	22
rrotate	-	-	104	108	92	-
md5f	-	-	832	97	71	34
md5g	-	-	1078	206	163	259
md5h	-	-	44	2	1	1
md5i	-	-	690	549	520	-
sha1	-	-	-	20	24	178
sha2	-	-	-	-	179	214
ga-p13	-	-	-	27	127	-
ga-p14	-	-	-	-	187	281
ga-p15	-	-	-	-	-	-
ga-p17	-	-	-	-	-	-

Table 4.2: Median time in seconds to reach best known programs. "-" indicates that the superoptimizer failed to find a best known program in one or more runs. Bold denotes the fastest superoptimizer to find a best known program in each benchmark.

superoptimizer did not find a program as optimal as the best known solution on one or more runs on a benchmark, the table excludes that corresponding entry.

Hypothesis E *The cooperative superoptimizer improves on the enumerative superoptimizer by utilizing the strengths of other search techniques.*

We compare $\widetilde{C}$ and $\widetilde{E}$. While $\widetilde{E}$ uses only enumerative search instances, $\widetilde{C}$ uses enumerative as well as symbolic and stochastic search instances. According to the result, $\widetilde{C}$ was consistent at finding best known solutions on 29 out of 32 benchmarks, while $\widetilde{E}$ was consistent on 23 benchmarks. $\widetilde{C}$ and $\widetilde{E}$ were comparable in term of search time; $\widetilde{C}$ was 33% faster, on average. Columns $\widetilde{C}$ of Figure 4.7 and Table 4.2 display the costs of the best correct programs found by $\widetilde{C}$ and its median time to find the best known solutions for all benchmarks. Compared to the algorithm used in the state-of-the-art superoptimizer (STOKE), $\widetilde{C}$ was, on average, 12x faster than the best of $\widetilde{ST}^s$ and $\widetilde{ST}^o$. The performance cost of code produced by $\widetilde{C}$ is, on average, 18% lower than that of the best from $\widetilde{ST}^s$ and $\widetilde{ST}^o$.

We also tested $\widetilde{C}'''$—the cooperative superoptimizer with the enumerative search without our pruning strategies—to examine how much the performance of the enumerative instances affect the performance of the cooperative superoptimizer. Columns $\widetilde{C}'''$ of Figure 4.7 and Table 4.2 display the costs of the best correct programs found by $\widetilde{C}'''$ and its median time to find the best known solutions. According to the result, $\widetilde{C}'''$ could not consistently solve seven benchmarks that $\widetilde{C}$ could. Hence, we conclude that our pruning strategies in the enumerative search are crucial for obtaining the best performance out of the cooperative superoptimizer.

Experiment IV: Runtime speedup over `gcc -O3`

Experiment IV is designed to test the effectiveness of the cooperative superoptimizer against an optimizing compiler. We measure the execution time of all benchmarks in this experiment on an actual ARM Cortex-A9.

Hypothesis F *Cooperative superoptimizer can optimize code generated from a non-optimizing compiler and obtain code as fast as generated from an optimizing compiler.*

From the previous experiment, $\widetilde{C}$ optimized code generated from `gcc -O0` and produced code as fast as `gcc -O3` code for all ARM benchmarks. In fact, $\widetilde{C}$ found faster code than those generated from `gcc -O3` on five benchmarks. One of them is 17.8x faster. Thus, for the new architectures for which we do not have good optimizing compilers, our superoptimizer can help generating efficient code.

Program	gcc -O3 length	Output length	Search time (s)	Runtime Speedup	Path to best code
p18	7	4	9	2.11	E^s
p21	6	5	1139	1.81	E^o*, SM^o*, ST^o*
p23	18	16	665	1.48	$ST^o* \rightarrow E^o*$
p24	7	4	151	2.75	$ST^o* \rightarrow E^o* \rightarrow ST^o \rightarrow E^o*$
p25	11	1	2	17.8	E^s
wi-txrate5a	9	8	32	1.31	$SM^o \rightarrow ST^o$
wi-txrate5b	8	7	66	1.29	E^o
mi-bitarray	10	6	612	1.82	$SM^o* \rightarrow E^o*$
mi-bitshift	9	8	5	1.11	E^o
mi-bitcnt	27	19	645	1.33	$E^o \rightarrow ST^o \rightarrow E^o \rightarrow ST^o \rightarrow E^o$
mi-susan	30	21	32	1.26	ST^o

Table 4.3: Execution time speedup over `gcc -O3` code and search instances involved in finding the solution. In the last column, $X \rightarrow Y$ indicates that Y uses the best code found by X. $*$ indicates exchanging of the best code among search instances of the same search technique.

Hypothesis G *Cooperative superoptimizer can further optimize real-world code generated by an optimizing compiler.*

We compiled *WiBench* [176] (a kernel suite for benchmarking wireless systems) and *MiBench* [69] (an embedded benchmark suite) using `gcc -O3` for ARM. We extracted basic blocks from the compiled assembly and selected 13 basic blocks that contain more than seven instructions and have more data processing than load/store instructions. For six out of 13 code fragments, $\widetilde{C}$ found faster fragments compared to those generated by `gcc -O3`, offering up to 82% speedup.

Table 4.3 summarizes characteristics of the program fragments found by $\widetilde{C}$ that are faster than those generated by `gcc -O3`. Column 'runtime speedup' reports how much faster the fragments are when running on an actual ARM processor. The last column demonstrates that different base search techniques contribute to finding the best solutions in many benchmarks. For example, in the `wi-txrate5a` benchmark from WiBench's rate matcher kernel, a SM^o instance first optimized the input program, and then a ST^o instance optimized the program found by the SM^o instance and arrived at the best known solution. In `mi-bitshift` from MiBench's bit shift benchmark, an E^o instance immediately found the best program by applying the optimization explained in the concrete example in Section 4.5. In `p21` from Hacker's Delight, the path to the best known solution involves passing the latest best programs between many E^o, SM^o, and ST^o instances repeatedly.

To illustrate how the different types of search instances work together in practice, we explain how the cooperative superoptimizer found the best program in the `mi-bitarray` benchmark, getting a specific bit in an array. The superoptimizer found two optimizations (Optimization I and II) displayed in Figure 4.8. The code in blue is inside a window, and the rest are the context used in the window decomposition. First, a SM^o instance optimized code inside a small window of two instruction to one instruction. The SM^o instance was able to perform this optimization because it can synthesize an arbitrary constant, in this case, 248. After the SM^o instance discovered Optimization I, an E^o instance optimized the

Optimization 1

before

```
cmp   r1, #0
mov   r3, r1, asr #31
add   r2, r1, #7
mov   r3, r3, lsr #29
movge r2, r1
ldrb  r0, [r0, r2, asr #3]
add   r1, r1, r3
and   r1, r1, #7
sub   r3, r1, r3
asr   r1, r0, r3
and   r0, r1, #1
```

(a)

after

```
cmp   r1, #0
mov   r3, r1, asr #31
add   r2, r1, #7
mov   r3, r3, lsr #29
movge r2, r1
ldrb  r0, [r0, r2, asr #3]
bic   r1, r2, #248
sub   r3, r1, r3
asr   r1, r0, r3
and   r0, r1, #1
```

(b)

Optimization 2

before

```
cmp   r1, #0
mov   r3, r1, asr #31
add   r2, r1, #7
mov   r3, r3, lsr #29
movge r2, r1
ldrb  r0, [r0, r2, asr #3]
bic   r1, r2, #248
sub   r3, r1, r3
asr   r1, r0, r3
and   r0, r1, #1
```

(b)

after

```
asr   r3, r1, #2
add   r2, r1, r3, lsr #29
ldrb  r0, [r0, r2, asr #3]
and   r3, r2, #248
sub   r3, r1, r3
asr   r1, r0, r3
and   r0, r1, #1
```

(c)

Figure 4.8: Optimizations that the cooperative superoptimizer discovered when optimizing `mi-bitarray` benchmark. Blue highlights the difference between before and after each optimization. (a) is the original program. (b) is the intermediate program. (c) is the final optimized program.

code further. Optimization II performed by the E^o instance, in fact, consists of two different optimizations. The first optimization transforms:

```
cmp   r1, #0
add   r2, r1, #7
movge r2, r1              // mov when r1 >= 0 (signed)
```

to:

```
asr r3, r1, #2            // r3 = r1 s>> 2
add r2, r1, r3, lsr #29   // r2 = r1 & (r3 u>> 29)
```

eliminating the `cmp` instruction and the conditional suffix. Note that this transformation is valid in any context. In the second optimization, the superoptimizer learned from the postfix—specifically at the instruction `sub r3, r1, r3`—that only the difference between the values of `r1` and `r3` matters, and the exact values of `r1` and `r3` do not. This particular optimization illustrates that not only precondition but postcondition also helps the superoptimizer discover more optimizations. Notice that the E^o instance used the constant 248, found by SM^o, to synthesize the final code, as the optimized fragment contains 248. Hence, in order to obtain the final code in this benchmark, we need the enumerative search, the symbolic search, and the context-aware window decomposition all together.

Existing Superoptimizers' Implementations

The original stochastic superoptimizer (STOKE) [136, 137] is for x86. Consequently, we could not use STOKE in our experiments. STOKE can evaluate approximately 10^6 candidates per second by executing programs natively [137] or running emulators on a cluster of machines [136]. Without an ability to run programs natively or an access to run programs on a cluster of machines, one will not be able to achieve this kind of performance. Nevertheless, our stochastic superoptimizer is able to evaluate approximately 20,000 candidates per second and synthesize up to 10 ARM instructions within an hour using emulators on one machine with 32 cores. However, the optimized program with 10 instructions that our stochastic superoptimizer synthesized is not optimal, so it is not reported in the experiment in Section 4.8. Note that the size of our ARM search space is similar to x86 search space explored in the original STOKE without JIT [136]: they both have 400–1,000 different variations of opcodes [139]. Although ARM has much fewer actual opcodes than x86, many variations are created by the combination of opcodes, optional shift, and conditional suffixes.

Similarly, we created $E^{s'''}$ by modifying E^s to implement the algorithm used in the SIMD synthesizer [21] and the SyGus enumerative solver [11]. We note that $E^{s'''}$ does not use the stack-based program representation, used in [21] to remove search space symmetries due to register renaming. We did not use this representation because we observed that some optimal programs cannot be obtained from this representation, unless we introduce new pseudo-instructions for peaking into the stack and dropping values from the stack. Note that this optimization is orthogonal from the search algorithm and can improve our search technique.

4.9 Related Work

Symbolic search is popular in program synthesis tools such as Sketch [148] and Rosette [160]. Symbolic search has been used in superoptimizers. For example, the first symbolic superoptimizer is Denali [81], which significantly improves upon the pioneer superoptimizer by Massalin [103] because of the goal-directed nature of constraint solvers. Although constraint solvers have many clever pruning strategies (e.g. conflict clauses) and heuristics to make decisions, constraint solvers are not optimized for program synthesis problems. Component-based synthesis [67] introduces an alternative encoding, which significantly improves the performance of a symbolic search; however, even with this encoding, the symbolic solver from SyGus'14 competition still did not perform well [11]. Another pruning strategy using divide-and-conquer to break QFBV formula potentially reduces synthesis time by many orders of magnitude [149], but it is likely synthesizing the same program as given. The refutation-based approach used in the CVC4 solver [130], the winner of SyGus'15 competition, is also not suitable for superoptimization problems because it tends to produce very large solutions with many if-else constructs. Nevertheless, a constraint solver is extremely

powerful for synthesizing arbitrary constants in programs, unlike the other search techniques, and therefore we include this search technique in our cooperative superoptimizer.

Stochastic search, first used in STOKE [136, 137], randomly mutates a program to another using cost function to determine the acceptance of the mutation. STOKE is the first superoptimizer that is able to synthesize large programs (10–15 x86 instructions) in under an hour. The use of cost function to guide the search is one of the keys to its effectiveness. The weakness of this search is a possibility to get stuck at local minima and, as a result, fail to reach an optimal solution.

Enumerative search is used in many superoptimizers and program synthesizers. The pioneer superoptimizer, introduced by Massalin, employs enumerative search [103]. More recent work has shown that enumerative search can be extremely fast if it is done right, as the winning teams of two synthesis competitions (ICFP'13 [10] and SyGus'14 [11]) employed this technique. This is because an enumerative search is highly customized to solve a particular problem it is designed for. The problem domain knowledge can be encoded into the search as systematic pruning strategies or just as ad-hoc heuristics, such as which branch in the search tree should be explored first. In our experience, building an enumerative search is easy, but building a fast enumerative search is difficult because a fast enumerative algorithm requires many clever pruning strategies to make the search tractable.

We have tried existing pruning strategies including using virtual registers [166] and a stack-base program representation [21] to reduce symmetry, using a canonical form [19], and memorization [10]. However, these pruning strategies alone do not work for our problem domain as well as for [21, 11, 10]. This is because our search space is bigger. For example, the SIMD synthesizer usually considers only a small number of instructions that are predicted from the input non-vectorized programs. However, we cannot restrict the search space in the same way because our goal is to find the optimal code fragment, which may require unexpected instructions. The SyGus and ICFP competitions only include programs that take in one argument and produce one return value. Thus, we have introduced new pruning strategies that make program synthesis problems more tractable.

However, new pruning strategies are still needed if we want to solve synthesis problems with even bigger search space. For example, memorization similar to [10] should accelerate the search even more. However, from our experience, the memorization system requires a decent amount of engineering effort to support quick lookup in a very large database that contains more than billion programs in order to speed up our synthesis process. We did not spend enough effort to implement a very efficient memorization system, so our superoptimizer does not currently utilize this technique.

4.10 Conclusion

This chapter introduced the LENS algorithm, which can optimize larger program fragments compared to existing techniques. To optimize even larger program fragments, we applied a

context-aware window decomposition, optimizing a subfragment of the entire code with the precise precondition and postcondition from the surrounding context. Lastly, we improved upon the LENS algorithm by combining symbolic and stochastic search into our system. To make superoptimization even more practical, we can cache superoptimized code to avoid an expensive search when optimizing programs we have seen before.

We also introduced GREENTHUMB, an extensible framework for constructing superoptimizers for diverse ISAs, providing a variety of search techniques. We believe that this framework represents an important step towards making superoptimization more widely used. That is, GREENTHUMB is an enabling technology supporting rapid development of superoptimizers and retargetability of efficient search techniques across different architectures.

Chapter 5

Toward Resource-Mapping Framework

5.1 Overview

Similar to superoptimizer construction framework, I envision a framework for building resource allocation synthesizers. A large body of literature exists on the use of various constraint solvers to solve compiler resource allocation problems. Commonly used constraint solvers for optimal solutions include Integer Linear Programming (ILP), Constraint Programming (CP), and Satisfiability Problem (SAT). For approximate solutions, the common solver is Simulated Annealing (SA). Existing work demonstrates that these techniques produce better solutions than heuristic algorithms. However, it requires special expertise to translate a resource allocation problem into a specific constraint, especially ILP.

Inspired by Rosette [159, 160], I prototype a framework in which the users can simply implement a function that calculates the cost of a given candidate and use assertion constructs to specify hard constraints. The framework then automatically generates ILP constraints, which can be solved more efficiently than does SMT, for resource mapping problems [168, 116, 114, 80].

5.2 Library

The prototype is in form of a Rosette library. The library provides functions — `mapped-to?`, `different?`, and `count-different`, as described in Table 5.1 — which are commonly used for computing communication cost for resource mapping problems but are non-trivial to be

Materials in this chapter are based on a part of work published as Bodik et al., "Domain-Specific Symbolic Compilation," in proceedings of SNAPL 2017 [25].

Function	Type	Description

`(mapped-to? p n)`	`p: sym/conc integer`
	`n: concrete integer`
	`return: sym/conc integer`
		returns 1 if place p is core n (i.e. $p = n$), otherwise returns 0

`(different? p r n)`	`p: sym/conc integer`
	`r: sym/conc integer`
	`n: concrete integer`
	`return: sym/conc integer`
		returns 1 if places p and r are different, and place p is core n (i.e. $(p \neq r) \wedge (p = n)$), otherwise returns 0

`(different? ps r n)`	`ps: list of sym/conc integers`
	`r: sym/conc integer`
	`n: concrete integer`
	`return: sym/conc integer`
		returns 1 if there is at least one place p in ps such that $(p \neq r) \wedge (p = n)$, otherwise returns 0

`(count-different p rs n)`	`p: sym/conc integer`
	`rs: list of sym/conc integers`
	`n: concrete integer`
	`return: sym/conc integer`
		returns a number of unique places in rs that differ from p if place p is core n, otherwise returns 0

Table 5.1: Description of resource language operations. Sym/conc stands for symbolic or concrete.

formulated in ILP. A return value of a function from the library is a symbolic expression. Recall that there are two kinds of vales in Rosette: concrete and symbolic. Concrete values can be evaluated normally in Racket. An operation on at least one symbolic value or symbolic expression produce a symbolic expression. A symbolic variable is an unknown variable that users want to solve for.

If a program uses our library in a way that meets the following requirements, the program can be solved more efficiently using an ILP or Mixed Integer Linear Programming (MIP) solver such as CPLEX.

- A symbolic expression can only be used as an operand to operations $+$, $-$, and $\times$.
- At least one operand of $\times$ must be concrete.
- All program path conditions must be concrete.

Implementation

Table 5.2 details the implementation of the library functions. The function `(mapped-to? p n)` creates symbolic variables $M_{pn}(p, n')$ for all $n' \in N$ — where N is a set of values that p can take — and returns $M_{pn}(p, n)$; $M_{pn}(p, n) = 1$ if $p = n$, and $M_{pn}(p, n) = 0$ otherwise. Since p can be mapped to only one value, the function adds the constraint $\sum_{n' \in N} M_{pn}(p, n') = 1$ into the global list of assertions. `(different? p r n)` creates and returns a variable $Remote_{prn}(p, r, n)$, as well as adds $Remote_{prn(p,r,n)} \geq M_{pn}(p, n) - M_{pn}(r, n)$ and $0 \leq Remote_{prn(p,r,n)} \leq 1$ to the global list of assertions. Note that $Remote_{prn(p,r,n)}$ can be either 0 or 1 when $p = r$, which

Function → return	**Created Variables**	**Additional Assertions**
`(mapped-to? p n)` $\rightarrow M_{pn}(p,n)$	$\forall n' \in N, M_{pn}(p,n')$	$\forall n' \in N, 0 \leq M_{pn}(p,n') \leq 1$ $\sum_{n' \in N} M_{pn}(p,n') = 1$
`(different? p r n)` $\rightarrow Remote_{prn}(p,r,n)$	$Remote_{prn}(p,r,n)$	$0 \leq Remote_{prn(p,r,n)} \leq 1$ $Remote_{prn(p,r,n)} \geq M_{pn}(p,n) - M_{pn}(r,n)$
`(different? p rs n)` $\rightarrow Remote_{prsn}(p,rs,n)$	$Remote_{prsn}(p,rs,n)$	$0 \leq Remote_{prsn(p,rs,n)} \leq 1$ $\forall r \in rs, Remote_{prsn}(p,rs,n) \geq Remote_{prn}(p,r,n)$
`(count-different p rs n)` $\rightarrow Count_{prsn}(p,rs,n)$	$Count_{prsn}(p,rs,n)$ $\forall n' \in N, M^*_{rsn}(n')$	$\forall n \in N, 0 \leq M^*_{rsn}(n) \leq 1$ $\forall n \in N, r \in rs, M^*_{rsn}(n) \geq M_{pn}(r,n)$ $sum^\dagger = \sum_{n' \in \{N-\{n\}\}} M^*_{rsn}(n')$ $offset^\dagger = (M_{pn}(p,n) - 1) \times MAX_{INT}$ $Count_{prsn}(p,rs,n) \geq 0$ $Count_{prsn}(p,rs,n) \geq sum^\dagger + offset^\dagger$

Table 5.2: Implementation of resource language operations. $sum^\dagger$ and $offset^\dagger$ are temporary variables.

is not what we want. However, this equation is valid if $Remote_{prn(p,r,n)}$ is (indirectly) minimized, so it is 0 when $p = r$ as we expect. The validity check happens when Rosette `minimize` function is called. Similar to $Remote_{prn(p,r,n)}$, $Remote_{prsn(p,rs,n)}$ and $Count_{prsn}(p,rs,n)$ — returned from `(different? ps r n)` and `(count-different p rs n)` respectively — must be minimized as well.

Our approach requires no change in Rosette's internals other than adding CPLEX as another Rosette solver. The library functions simply generate Rosette assertions. We implement our custom `minimize` function, which performs the validity check before calling Rosette's `minimize`.

5.3 Case Study: Program Partitioning

Compilers for fine-grain many-core architectures, such as Chlorophyll must partition the program into tiny code fragments mapped onto physical cores. The Chlorophyll type system ensures that no fragment overflows the 64-word capacity of its core. Recall that each variable and operation have a *partition type* whose value is a core ID. The type checker, whose simplified code is shown in Figure 5.1, computes the code size of each fragment. The `tree-map` function traverses a program AST in the post-order fashion and applies the function `count-space` on each node in the AST (line 28). The checker accumulates the space taken by each node (e.g. a `binexpr` node on line 18), and space occupied by communication code, for both one-to-one communication (e.g. sending operand values to an operator on lines 19–20) and broadcast communication (e.g. sending a condition result to all nodes in the body of `if` on lines 24–25). To account for the communication code inserted during code generation, the checker adds two words each time a value flows between cores.

```
(define cores-space (make-vector n-cores 0)) ; space used up on each core
(define (inc-space p size) (vector-set! cores-space p (+ (vector-ref cores-space p) size)))

; Increase code size whenever core p sends a value to core r.
(define (comm p r) (when (not (= p r)) (begin (inc-space p 2) (inc-space r 2))))

; Increase code size for broadcast communication from p to ps. ps may contain duplicates.
(define (broadcast p ps)
  (define remote-ps (length (remove p (unique ps)))) ; number of unique cores in ps excluding p
  (inc-space p (* 2 remote-ps))   ; space used in the sender core
  (for ([r ps]) (inc-space r 2))) ; space used in the receiver cores

(define (count-space node) ; Count space needed by an AST node.
  (cond
   [(var? node) (inc-space (place-type node) 1)]
   [(binexpr? node) ; The inputs to this operation come from binexpr-e1 and binexpr-e2.
    (define p (place-type node))
    (inc-space p (size node)) ; space taken by the operation
    (comm (place-type (binexpr-e1 node)) p)  ; Add space for communication code when
    (comm (place-type (binexpr-e2 node)) p)] ; operands come from other cores.
   [(if? node)
    ; If is replicated on all cores that run any part of if's body. We omit inc-space here.
    ; The condition result is broadcast to all cores used in if's body.
    (broadcast (place-type (if-test node))
               (append (all-cores (if-then node)) (all-cores (if-else node))))]
  ...))

(tree-map count-space ast)
(for ([space cores-space]) (assert (< space core-capacity)))
(minimize (apply + cores-space)) ; used during inference only
```

Figure 5.1: Original type checker, ensuring that code fragments fit into cores

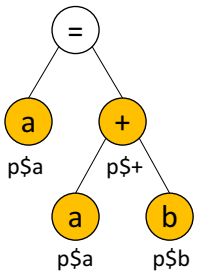
```
(define n (make-parameter #f)) ; a parameter procedure for dynamic binding
(define (comm p r) (inc-space (n) (* 2 (+ (different? p r (n)) (different? r p (n))))))
(define (broadcast p ps)
  (inc-space (n) (* 2 (+ (count-different p ps (n)) ; space used in the sender core
                         (different? ps p (n)) )))) ; space used in the receiver cores

; The function count-space is changed in one place (see text).
; The function inc-space is unchanged.

(for ([i n-cores]) (parameterize ([n i]) (tree-map count-space ast)))
(for ([space cores-space]) (assert (< space core-capacity)))
(minimize (apply + cores-space))
```

Figure 5.2: Type checker in resource language, producing ILP constraints

Automatic Program Partitioning as Type Inference

When a program omits some partition types, the compiler infers them, effectively partitioning the program. We implement the type inference using Rosette, which symbolically evaluates the type checker in Figure 5.1. The type checker needs no changes; we only need to initialize the (unknown) partition types in the program to symbolic values (i.e., `p$0, p$1, ...`).

Figure 5.3(a) shows an example of a program AST with unknown places. Each node in the AST is annotated with its symbolic place type. Figure 5.3(b) shows the conceptual



(a) Example program AST. Each node is annotated with its place type below. The yellow nodes are the ones that have been interpreted.

```
(inc-space p$a 1) ; line 15, node = a [def]
(inc-space p$a 1) ; line 15, node = a [use]
(inc-space p$b 1) ; line 15, node = b
(inc-space p$+ 1) ; line 18, node = +
(comm p$a p$+)    ; line 19, comm a -> +
(comm p$b p$+)    ; line 20, comm b -> +
```

(b) Residual type checking program after traversing the yellow nodes in the AST on the left (post-order). The line numbers in the comments indicate where the expressions come from from Figure 5.1.

Figure 5.3: Running example of program partitioning

```
> cores-space
#(; core 0
  (+ (ite (= p$a 0) 1 0) ; a [def]
     (ite (= p$a 0) 1 0) ; a [use]
     (ite (= p$b 0) 1 0) ; b
     (ite (= p$+ 0) 1 0) ; +
     (ite
       (and (or (= p$+ 0) (= p$a 0))
            (! (= p$+ p$a)))
       2 0)              ; a -> +
     (ite
       (and (or (= p$+ 0) (= p$b 0))
            (! (= p$+ p$b)))
       2 0))             ; b -> +

  ; core 1
  (+ ...)
)
```

(a) Original symbolic expression generated from the original type checker (Figure 5.1)

```
> cores-space
#(; core 0
  (+ (* 1 Mpn(p$a,0)) ; a [def]
     (* 1 Mpn(p$a,0)) ; a [use]
     (* 1 Mpn(p$b,0)) ; b
     (* 1 Mpn(p$+,0)) ; +
     (* 2 Remote_prn(p$+,p$a,0)) ; a -> +
     (* 2 Remote_prn(p$a,p$+,0))
     (* 2 Remote_prn(p$+,p$b,0)) ; b -> +
     (* 2 Remote_prn(p$b,p$+,0)))
  ; core 1
  (+ ...))

> (asserts) ; global assertions
((and (<= 0 Mpn(p$a,0)) (>= 1 Mpn(p$a,0)))
 (and (<= 0 Mpn(p$a,1)) (>= 1 Mpn(p$a,1)))
 (= 1 (+ Mpn(p$a,0) Mpn(p$a,1))) ; a
 ...
 (<= 0 Remote_prn(p$+,p$a,0))    ; a -> +
 (>= 1 Remote_prn(p$+,p$a,0))
 (>= Remote_prn(p$+,p$a,0)
     (- Mpn(p$+,0) Mpn(p$a,0)))
 ...
)
```

(b) ILP symbolic expression generated from the modified type checker (Figure 5.2)

Figure 5.4: Symbolic expression of space occupied in each core after running a type checker on the yellow nodes in the example AST (Figure 5.3(a))

partially-evaluated type checker after checking the yellow nodes in the example AST; concrete expressions are fully evaluated, and the expressions with symbolic variables remain. After we symbolically evaluate the residual type checker in Figure 5.3(b), we obtain `cores-space` shown in Figure 5.4(a). Rosette then uses Z3 to solve the generated SMT constraints on `cores-space` (line 29 of Figure 5.1) and minimize the total code size (line 30 of Figure 5.1).

Hence, we obtain our type inference just by implementing a type checker. The development process requires little effort, but the type inference is slow at inferring place types.

Symbolic Evaluation to ILP Constraints

With our resource-mapping library, we implement the type checker as before but using the provided functions from our library. We make four minimal changes to our original type checker, shown in Figure 5.2. First, we traverse the AST once for every core (line 10). Each iteration i is responsible for accumulating space used in core i. Second, in the function `count-space`, we change the expression to increase the size of core p by the size of the operation of *node* from `(inc-space p (size node))` to `(inc-space (n) (* (size node) (mapped-to? p (n))))`. The previous call produces a non-linear equation because the first argument p, which is symbolic, to `inc-space` is used as a path condition. Third, we avoid symbolic path conditions inside the function `comm` by using `(different? p r (n))` to compute the size of code for sending data at core (n), and similarly for receiving data. Last, in the function `broadcast`, we utilize `count-different` to compute space taken by code for broadcasting a value to a set of cores.

Figure 5.4(b) show the symbolic expression of `cores-space` along with additional assertions after symbolically executing the modified type checker on the yellow nodes of the AST in Figure 5.3(a). Notice that the new expression is linear, while the original one is not.

Evaluation

The ILP encoding produced by our library solves problems inaccessible to the SMT-based partitioner, and it is faster than the SMT encoding optimized for the domain of partitioning problems (namely, flattening deeply nested `ite` expressions). Figure 5.5 shows the

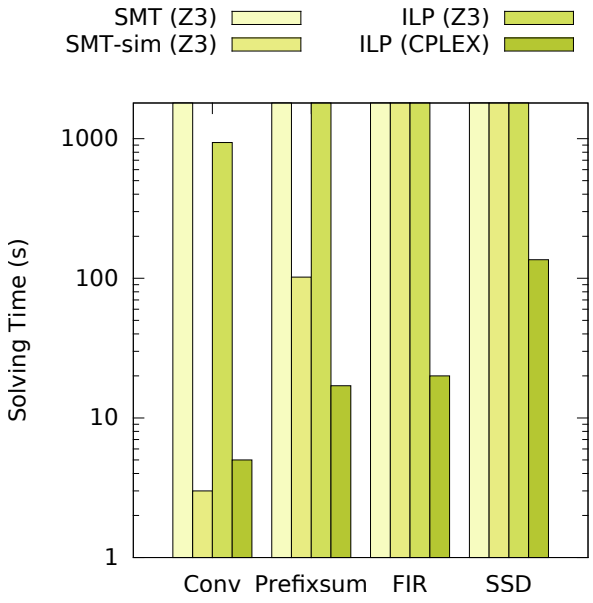


Figure 5.5: Time to solve program partitioning

median time to partition four benchmarks across three runs. We set the timeout to 30 minutes. In summary, SMT always timed out; domain-optimized SMT constraints solved half of the benchmarks; the ILP encoding solved all benchmarks. Note that in the original Chlorophyll experiment, SMT solved these benchmarks within five minutes. This is only because some place types were specified, while all place types were unknown here.

5.4 Future Work

We believe that this library can be extended to solve other resource mapping problems, such as spatial architecture scheduling [114] and mapping programs to reconfigurable switches [80], by supporting more functions. Further, inspired by GreenThumb, we would like to explore how to apply decomposition and make different types of solvers with different encodings collaborate to find the best solution in the fastest way possible.

Chapter 6

Conclusion and Future Work

In this thesis, I have shown that we can enable programmability on new emerging hardware architectures for gains in efficiency, through programming abstractions, synthesis, and problem decomposition.

In Chapter 2, I demonstrated how to build a programming system for low-power spatial architectures using these three key concepts. The programming model allows programmers to express their high-level insights into program partitioning and layout. The compiler applies synthesis to fill in low-level details of program partitioning, layout, and routing, as well as automatically discover instruction-level optimizations. Decomposing the compilation problem into smaller subproblems makes it tractable to generate efficient code for our target hardware. I showed that synthesis-aided compilation is indeed a low effort approach to building a high-quality compiler for an idiosyncratic architecture.

In Chapter 3, I demonstrated how to build another programming system for a drastically different application domain. In this chapter, I primarily explored the power of programming abstractions to enable programmability for a heterogeneous server platform with a CPU and a programmable NIC. Based on the decomposition concept, I proposed a more modular implementation of a communication channel between CPU and NIC by separating the communication handling logic from low-level data synchronization and optimizations. I showed that our programming abstractions allowed developers to discover offloading strategies that outperform programs that run entirely on a CPU on a several representative data-center applications.

In Chapter 4, I moved to a meta level, and asked how we can enable system developers to build new programming systems for new emerging architectures. In this chapter, I tackled the problem of generating efficient code at the instruction level. I built a "superoptimizer generator". Our research prototype has evolved into a commercial framework that is being used to build superoptimizers for various ISAs.

In Chapter 5, similar to the previous chapter, I asked how we can enable building synthesis-aided compilers but focusing on higher-level optimizations. In response to this question, I examined resource mapping problems, which are the common high-level program optimizations in many compilers. I built a prototype toward a scalable, retargettable re-

source mapping framework that enables developers to solve a subset of resource mapping problems without having to formulate the problems into efficient constraints by themselves.

Before concluding, I turn to the present and future trends of emerging computing platforms and compilation, and then discuss future research directions in these domains.

6.1 Trends in Emerging Computing Platforms

Cutting-edge applications continue to put new demands on computer architectures for higher performance and lower energy consumption. Google developed TPU, specifically to run deep learning workload [82]. Tesla also recently announced that the company is developing its own hardware to aid Autopilot [143]. Over forty startups founded in the past few years are developing new AI chips [154].

In networking domain, network virtualization, software-defined network, high-demand data-center services have imposed new requirements to routers, switches, and servers. Many companies responded to this trend with new specialized hardware. Barefoot Networks developed Tofino, a programmable switch, to enable network designers to push updates and deploy new features at the speed of software [20]. Various designs of programmable NICs have been developed to satisfy different purposes such as supporting network function virtualization [3, 6, 104] and accelerating network applications [36, 105].

Certainly in the future, new specialized hardware will continue to be created to serve the demands from new applications. The techniques purposed in this thesis will likely continue to be relevant because of this trend.

6.2 Trends in Compilation

Synthesis-aided compilation has become increasingly more popular. Superoptimization has gained more popularity not only in the research community but also in practice. A large number of papers on superoptimization [31, 43, 77, 106, 135, 150, 151] has been published in the past few years after STOKE [136, 137] and GreenThumb [124, 125]. Souper, an LLVM IR superoptimizer, has been built as an LLVM optimization pass that can run after LLVM's regular peephole optimizations [7, 135]. The Linki Tools company has developed the commercial superoptimization framework [99] from the resurgence of GreenThumb.

Apart from superoptimization, search-based techniques have been used in several well-known open-source and commercial compilers. The Wave Computing company built a SAT-based compiler, similar to Chlorophyll, for its coarse-grain reconfigurable array processor for accelerating deep learning computation [39]. Tensor Comprehensions [164] and TVM [41], widely-known deep learning compilers, both use search to find efficient program schedules. I envision that synthesis-aided compilation will continue to gain more attention as new hardware is emerging.

6.3 Lessons Learned and Thoughts for the Future

Automatic ISA semantics extraction. The most tedious and error-prone part of extending GreenThumb to a new ISA is implementing the semantics (i.e., the interpreter) of each instruction. For some ISAs, there already exists the formal semantics released by the developer of the ISA. For such ISAs, it would be more sensible to automatically translate the official formal semantics into the format required by GreenThumb.

Networking domain. Floem has laid a ground work on programming network applications in a combined CPU-NIC environment. However, programmers are still responsible for making all the decisions on offloading and program partitioning choices. Floem will certainly benefit from another layer of automation like autotuning to help make some of these decisions. Apart from more automation capability, extending Floem to support other kinds of programmable NICs is another interesting direction. FPGA-based NICs [36, 105, 177] have already been widely used in many financial tech companies and data centers. Extending Floem to FPGA-based NICs thus seems to be a natural direction. A reconfigurable match table (RMT) model [27] is a theoretical model, originally designed for switches, that is likely to soon be implemented in a NIC. RMT-based NICs can potentially be more energy efficient than other classes of programmable NICs that we have today. Due to a variety of NICs, another interesting research problem arises: how to make a compiler portable and able to generate efficient code for various classes of accelerators that are substantially different from each other.

Synthesis for high-level optimizations. Our resoure mapping library (Chapter 5) is our attempt toward a generalized framework for solving a resource mapping problem, one of the common high-level optimizations. However, the prototype library only works for a small subset of resource mapping problems. Designing additional library functions to support most resource mapping problems under the condition that the functions must be translated into efficient constraints is a interesting research direction. Another interesting direction is exploring how to apply decomposition and make different types of solvers collaborate to find the best solution in the fastest way possible.

Besides resource mapping problems, there are many other high-level optimization problems that programmers still have to solve manually, which perhaps can be solved automatically by synthesis. For example, in GPGPU programming, programmers are responsible for deciding where and how to store data as well as how to access it in a way that minimizes latency and maximizes throughput. I have been collaborating with NVIDIA to use program synthesis to automatically discover advance data movements required when using high-bandwidth shared memory or local registers for shared temporary storage [120].

Iterative refinement in synthesis-aided compilation. The biggest problem we encountered in Chlorophyll is the cascading effect of an imprecise model used in an earlier

pass of the compiler (e.g., the partitioning synthesizer does not reason about space taken by data routing code). In general, if we build a synthesis-aided compiler with multiple synthesizers operating at different levels of abstractions, the higher-level synthesizers may suffer from an imprecise abstract cost model. One solution to this problem is to introduce an iterative refinement that reruns the compilation with adjusted cost models until the compiler produces satisfied solutions.

Program synthesis advancement. As evidenced in this thesis, program synthesis and constraint solving are powerful techniques that can help generate very efficient code for emerging computing platforms. The more scalable the synthesis, the more complex the problem we can solve. One promising approach that has become increasingly popular is applying statistical models and machine learning to scale synthesis to larger problems.

Synthesis for hardware/software co-design. This thesis applies synthesis to design efficient software for a given hardware. However, to achieve the most efficient program, we will need to design hardware specifically for it too. Ultimately, given a set of target applications, we should synthesize both a hardware design and software implementations that are overall most efficient for the given target domain. To get there, we must continue to improve synthesis techniques to solve more complex problems and handle larger search spaces.

In summary, the hardware trends in both embedded devices and data centers are now driven by both performance and power constraints. These trends will lead to increasingly difficult programming environments. This thesis has laid some of the foundations for programming such hardware, including the superoptimizer generator, a prototype resource mapping framework, and shared my experiences of building languages and compilers for multiple application domains and hardware settings.

Bibliography

[1] Apache Storm. http://storm.apache.org. Accessed: 2017-11-15.

[2] Cavium Development Kits. http://www.cavium.com/octeon_software_develop_kit.html. Accessed: 2017-11-15.

[3] Cavium LiquidIO. http://www.cavium.com/LiquidIO_Adapters.html. Accessed: 2017-11-14.

[4] DPDK: Data Plane Develepment Kit. http://dpdk.org/. Accessed: 2017-11-07.

[5] IEEE P802.3bs 400 GbE Task Force. Adopted Timeline. http://www.ieee802.org/3/bs/timeline_3bs_0915.pdf. Accessed: 2017-11-16.

[6] Netronome Agilio SmartNICs. https://www.netronome.com/products/smartnic/overview/. Accessed: 2017-11-14.

[7] Souper. http://github.com/google/souper.

[8] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[9] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, first edition, 1986.

[10] Takuya Akiba, Kentaro Imajo, Hiroaki Iwami, Yoichi Iwata, Toshiki Kataoka, Naohiro Takahashi, Michal Moskal, and Nikhil Swamy. Calibrating Research in Program Synthesis Using 72,000 Hours of Programmer Time. Technical report, MSR, 2013.

[11] Rajeev Alur, Rastislav Bodik, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shamwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-Guided Synthesis. In *SyGus Competition*, 2014.

[12] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. PetaBricks: A Language and Compiler for Algorithmic Choice.

In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, 2009.

[13] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. OpenTuner: An Extensible Framework for Program Autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, 2014.

[14] Andrew W. Appel and Lal George. Optimal Spilling for CISC Machines with Few Registers. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, 2001.

[15] Arduino. Arduino products. `https://www.arduino.cc/en/Main/Products`. Accessed: 2018-06-18.

[16] ARM. *Cortex-A9: Technical Reference Manual*, 2012.

[17] Rimas Avizienis and Per Ljung. Comparing the Energy Efficiency and Performance of the Texas Instrument MSP430 and the GreenArrays GA144 processors. Technical report, 2012.

[18] Sorav Bansal. *Peephole Optimization*. Stanford, CA, USA, 2008.

[19] Sorav Bansal and Alex Aiken. Automatic Generation of Peephole Superoptimizers. In *ASPLOS*, 2006.

[20] Barefoot. Barefoot Tofino. `https://www.barefootnetworks.com/products/brief-tofino/`. Accessed: 2018-06-18.

[21] Gilles Barthe, Juan Manuel Crespo, Sumit Gulwani, Cesar Kunz, and Mark Marron. From Relational Verification to SIMD Loop Synthesis. In *PPoPP*, 2013.

[22] Steven Bashford and Rainer Leupers. Phase-Coupled Mapping of Data Flow Graphs to Irregular Data Paths. *Design Automation for Embedded Systems*, 4(2):119–165, 1999.

[23] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, 2012.

[24] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014.

[25] Rastislav Bodík, Kartik Chandra, Phitchaya Mangpo Phothilimthana, and Nathaniel Yazdani. Domain-Specific Symbolic Compilation. In *2nd Summit on Advances in Programming Languages*, SNAPL '17, 2017.

[26] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[27] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, 2013.

[28] Zeki Bozkus, Alok Choudhary, Tomasz Haupt, Geoffrey Fox, and Sanjay Ranka. Compiling HPF for Distributed Memory MIMD Computers. In *The Interaction of Compilation Technology and Computer Architecture*. 1994.

[29] Kevin J. Brown, Arvind K. Sujeeth, Hyouk Joong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. A Heterogeneous Parallel Framework for Domain-Specific Languages. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, 2011.

[30] Mihai Budiu, Girish Venkataramani, Tiberiu Chelcea, and Seth Copen Goldstein. Spatial Computation. In *ASPLOS*, 2004.

[31] Rudy Bunel, Alban Desmaison, M. Pawan Kumar, Philip H. S. Torr, and Pushmeet Kohli. Learning to superoptimize programs. *CoRR*, abs/1611.01787, 2016.

[32] Joe Bungo. The Use of Compiler Optimizations for Embedded Systems Software. *Crossroads*, 15(1):8–15, September 2008.

[33] Doug Burger, Stephen W. Keckler, Kathryn S. McKinley, Mike Dahlin, Lizy K. John, Calvin Lin, Charles R. Moore, James Burrill, Robert G. McDonald, William Yoder, and the TRIPS Team. Scaling to the End of Silicon with EDGE Architectures. *Computer*, July 2004.

[34] David Callahan and Ken Kennedy. Compiling programs for distributed-memory multiprocessors. *The Journal of Supercomputing*, 2(2):151–169, Oct 1988.

[35] William W. Carlson, Jesse M. Draper, and David E. Culler. S-246, 187 Introduction to UPC and Language Specification.

[36] Adrian Caulfield et al. A cloud-scale acceleration architecture. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–13. IEEE, 2016.

[37] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel Programmability and the Chapel Language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, August 2007.

[38] Satish Chandra, Vijay Saraswat, Vivek Sarkar, and Rastislav Bodik. Type inference for locality analysis of distributed data structures. In *PPoPP*, 2008.

[39] S. Chaudhuri and A. Hetzel. SAT-based compilation to a non-vonNeumann processor. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2017.

[40] Benjie Chen and Robert Morris. Flexible Control of Parallelism in a Multiprocessor PC Router. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*, 2001.

[41] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '18, 2018.

[42] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Optimizing Database-backed Applications with Query Synthesis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, 2013.

[43] Berkeley Churchill, Rahul Sharma, JF Bastien, and Alex Aiken. Sound Loop Superoptimization for Google Native Client. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, 2017.

[44] Cisco. Introduction To RPC/XDR. `http://www.cisco.com/c/en/us/td/docs/ios/sw_upgrades/interlink/r2_0/rpc_pr/rpintro.html`. Accessed: 2018-09-07.

[45] Wave Computing. Wave Computing's Native Dataflow Technology. `https://wavecomp.ai/technology/`. Accessed: 2018-06-26.

[46] David T. Connolly. An improved annealing scheme for the QAP. *European Journal of Operational Research*, 46(1):93 – 100, 1990.

[47] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, 2017.

[48] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, 2008.

[49] Gwenaël Delaval, Alain Girault, and Marc Pouzet. A Type System for the Automatic Distribution of Higher-order Synchronous Dataflow Programs. In *LCTES*, 2008.

[50] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, 2009.

[51] Marco Dorigo and Thomas Stützle. *Ant Colony Optimization*. Bradford Company, Scituate, MA, USA, 2004.

[52] Andrew Duller, Daniel Towner, Gajinder Panesar, Alan Gray, and Will Robbins. picoArray Technology: The Tool's Story. *CoRR*, abs/0710.4814, 2007.

[53] I. Ekmecic, I. Tartalja, and V. Milutinovic. A survey of heterogeneous computing: concepts and systems. *Proceedings of the IEEE*, 84(8):1127–1144, Aug 1996.

[54] Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. BURG: fast optimal instruction selection and tree parsing. *SIGPLAN Not.*, 27(4):68–76, April 1992.

[55] M. Frigo and S. G. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, Feb 2005.

[56] Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, François Bodin, Phil Barnard, Elton Ashton, Edwin V. Bonilla, John Thomson, Christopher K. I. Williams, and Michael F. P. O'Boyle. Milepost GCC: Machine Learning Enabled Self-tuning Compiler. *International Journal of Parallel Programming*, 2010.

[57] Joel Galenson, Philip Reames, Rastislav Bodik, Björn Hartmann, and Koushik Sen. CodeHint: Dynamic and Interactive Synthesis of Code Snippets. In *ICSE*, 2014.

[58] Younghwan Go, Muhammad Asim Jamshed, YoungGyoun Moon, Changho Hwang, and KyoungSoo Park. APUNet: Revitalizing GPU as Packet Processing Accelerator. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017.

[59] David W. Goodwin and Kent D. Wilken. Optimal and Near-optimal Global Register Allocations Using 0–1 Integer Programming. *Softw. Pract. Exper.*, 26(8):929–965, August 1996.

[60] Google. Protocol Buffers. `http://developers.google.com/protocol-buffers/`. Accessed: 2018-09-07.

[61] Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. A stream compiler for communication-exposed architectures. In *ASPLOS*, 2002.

[62] Venkatraman Govindaraju, Chen-Han Ho, Tony Nowatzki, Jatin Chhugani, Nadathur Satish, Karthikeyan Sankaralingam, and Changkyu Kim. DySER: Unifying Functionality and Parallelism Specialization for Energy-Efficient Computing. *IEEE Micro*, 32(5):38–51, September 2012.

[63] Torbjörn Granlund and Richard Kenner. Eliminating Branches Using a Superoptimizer and the GNU C Compiler. In *PLDI*, 1992.

[64] GreenArrays. *Product Brief: GreenArrays Architecture*, 2010.

[65] GreenArrays. *Product Brief: GreenArrays GA144*, 2010.

[66] GreenArrays. *Appplication Note AB012: Controlling the TI SensorTag with the GA144*, 2013.

[67] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *PLDI*, 2011.

[68] Manish Gupta. On Privatization of Variables for Data-Parallel Execution. In *International Symposium on Parallel Processing (IPPS)*, 1997.

[69] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, T Mudge, Rb Brown, and Todd Austin. Mibench: a free, commercially representative embedded benchmark suite. In *IEEE International Symposium on Workload Characterization*, 2001.

[70] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. PacketShader: A GPU-accelerated Software Router. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, 2010.

[71] Carl Hewitt, Peter Bishop, and Richard Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *IJCAI*, 1973.

[72] Amir H. Hormati, Yoonseo Choi, Manjunath Kudlur, Rodric Rabbah, Trevor Mudge, and Scott Mahlke. Flextream: Adaptive Compilation of Streaming Applications for Heterogeneous Architectures. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, 2009.

[73] Shan Shan Huang, Amir Hormati, David F. Bacon, and Rodric Rabbah. Liquid Metal: Object-Oriented Programming Across the Hardware/Software Boundary. In *Proceedings of the 22Nd European Conference on Object-Oriented Programming*, ECOOP '08, 2008.

[74] Richard Hughey. Programming Systolic Arrays. Technical report, Brown University, 1992.

[75] Texas Instruments. MSP430 Ultra-Low-Power MCUs. `http://www.ti.com/microcontrollers/msp430-ultra-low-power-mcus/overview.html`. Accessed: 2018-06-18.

[76] Intel. Reducing Data Center Energy Consumption. Technical report, 2008.

[77] Abhinav Jangda and Greta Yorsh. Unbounded Superoptimization. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2017, 2017.

[78] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soule, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles*, SOSP '17, 2017.

[79] Kurtis T. Johnson, A. R. Hurson, and Behrooz Shirazi. General-Purpose Systolic Arrays. *Computer*, 26(11), November 1993.

[80] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. Compiling Packet Programs to Reconfigurable Switches. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, 2015.

[81] Rajeev Joshi, Greg Nelson, and Keith Randall. Denali: a goal-directed superoptimizer. In *PLDI*, 2002.

[82] Norman P. Jouppi et al. In-Datacenter Performance Analysis of a Tensor Processing Unit. *CoRR*, abs/1704.04760, 2017.

[83] Mahmut Kandemir, N. Vijaykrishnan, and MaryJane Irwin. Compiler Optimizations for Low Power Systems. In *Power Aware Computing*, Series in Computer Science. 2002.

[84] Alan H. Karp. Programming for Parallelism. *Computer*, 20(5):43–57, May 1987.

[85] Antoine Kaufmann, Simon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. High Performance Packet Processing with FlexNIC. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, 2016.

[86] Joongi Kim, Keon Jang, Keunhong Lee, Sangwook Ma, Junhyun Shim, and Sue Moon. NBA (Network Balancing Act): A High-performance Packet Processing Framework for Heterogeneous Processors. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, 2015.

[87] Krzysztof Kuchcinski. Constraints-driven Scheduling and Resource Assignment. *ACM Trans. Des. Autom. Electron. Syst.*, 8(3):355–383, July 2003.

[88] Monica S. Lam. *A Systolic Array Optimizing Compiler*. Kluwer Academic Publishers, Norwell, MA, USA, 1989.

[89] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, 2004.

[90] Eugene L. Lawler. The quadratic assignment problem. *Manage. Sci.*, 9:586–599, 1963.

[91] Edward Ashford Lee and David G. Messerschmitt. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Trans. Comput.*, 36(1):24–35, January 1987.

[92] Walter Lee, Rajeev Barua, Matthew Frank, Devabhaktuni Srikrishna, Jonathan Babb, Vivek Sarkar, and Saman Amarasinghe. Space-time Scheduling of Instruction-level Parallelism on a Raw Machine. In *ASPLOS*, 1998.

[93] R. Leupers and P. Marwedel. Time-constrained code compaction for DSPs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 5(1):112–122, March 1997.

[94] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, 2017.

[95] Bojie Li, Kun Tan, Layong (Larry) Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, 2016.

[96] Jialin Li, Ellis Michael, and Dan R. K. Ports. Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, 2017.

[97] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016.

[98] Ben Liblit, Alex Aiken, and Katherine Yelick. Type Systems for Distributed Data Sharing. In *Proceedings of the 10th International Conference on Static Analysis*, SAS'03, 2003.

[99] Linki Tools. S10 Superoptimization Framework. `http://linki.tools/pages/s10`. Accessed: 2018-08-08.

[100] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. IncBricks: Toward In-Network Computation with an In-Network Cache. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, 2017.

[101] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Provably Correct Peephole Optimizations with Alive. In *PLDI*, 2015.

[102] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '42, 2009.

[103] Henry Massalin. Superoptimizer: a look at the smallest program. In *ASPLOS*, 1987.

[104] Mellanox Technologies. BlueField Multicore System on Chip. `http://www.mellanox.com/related-docs/npu-multicore-processors/PB_Bluefield_SoC.pdf`, 1018. Accessed: 2018-04-25.

[105] Mellanox Technologies. Innova - 2 Flex Programmable Network Adapter. `http://www.mellanox.com/related-docs/npu-multicore-processors/PB_Bluefield_SoC.pdf`, 1018. Accessed: 2018-04-25.

[106] David Menendez and Santosh Nagarakatte. Alive-Infer: Data-driven Precondition Inference for Peephole Optimizations in LLVM. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, 2017.

[107] John Merlin. Techniques for the Automatic Parallelisation of 'Distributed Fortran 90', 1992.

[108] P. Merolla, J. Arthur, F. Akopyan, N. Imam, R. Manohar, and D.S. Modha. A digital neurosynaptic core using embedded crossbar memory with 45pJ per spike in 45nm. In *Custom Integrated Circuits Conference (CICC), 2011 IEEE*, 2011.

[109] Microchip. Microchip AVR MCUs. `http://www.microchip.com/design-centers/8-bit/avr-mcus`. Accessed: 2018-06-18.

[110] Sparsh Mittal and Jeffrey S. Vetter. A Survey of CPU-GPU Heterogeneous Computing Techniques. *ACM Comput. Surv.*, 47(4):69:1–69:35, July 2015.

[111] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. The Click Modular Router. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, SOSP '99, 1999.

[112] Jarek Nieplocha, Bruce Palmer, Vinod Tipparaju, Manojkumar Krishnan, Harold Trease, and Edoardo Aprà. Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit. *Int. J. High Perform. Comput. Appl.*, 20(2):203–231, May 2006.

[113] Edmund B. Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. Helios: Heterogeneous Multiprocessing with Satellite Kernels. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, 2009.

[114] Tony Nowatzki, Michael Sartin-Tarm, Lorenzo De Carli, Karthikeyan Sankaralingam, Cristian Estan, and Behnam Robatmili. A General Constraint-centric Scheduling Framework for Spatial Architectures. In *PLDI*, 2013.

[115] Marek Palkowski. Impact of Variable Privatization on Extracting Synchronization-Free Slices for Multi-core Computers. In Rainer Keller, David Kramer, and Jan-Philipp Weiss, editors, *Facing the Multicore-Challenge III*, volume 7686 of *Lecture Notes in Computer Science*, pages 72–83. 2013.

[116] Jens Palsberg and Mayur Naik. ILP-based Resource-aware Compilation, 2004.

[117] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. NetBricks: Taking the V out of NFV. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI '16, 2016.

[118] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System is the Control Plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014.

[119] Phitchaya Mangpo Phothilimthana, Jason Ansel, Jonathan Ragan-Kelley, and Saman Amarasinghe. Portable Performance on Heterogeneous Architectures. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, 2013.

[120] Phitchaya Mangpo Phothilimthana, Archibald Samuel Elliott, Abhinav Jangda, Bastian Hagedorn, Henrik Barthels, Vinod Grover, and Rastislav Bodik. Swizzle Inventor: Data Movement Synthesis for GPU Kernels. In *Under Submission*, 2019.

[121] Phitchaya Mangpo Phothilimthana, Tikhon Jelvis, Rohin Shah, Nishant Totla, Sarah Chasins, and Rastislav Bodik. Chlorophyll: Synthesis-Aided Compiler for Low-Power Spatial Architectures. In *PLDI*, 2014.

[122] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. Floem: A Programming System for NIC-Accelerated Network Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '18, 2018.

[123] Phitchaya Mangpo Phothilimthana, Michael Schuidt, and Rastislav Bodik. Compiling a Gesture Recognition Application for a Low-Power Spatial Architecture. In *Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, LCTES '16, 2016.

[124] Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. GreenThumb: Superoptimizer Construction Framework. In *Conference on Compiler Construction*, CC '16, 2016.

[125] Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. Scaling Up Superoptimization. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, 2016.

[126] M. Puschel, J. M. F. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, Jianxin Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code Generation for DSP Transforms. *Proceedings of the IEEE*, 93(2):232–275, Feb 2005.

[127] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, 2014.

[128] Wajahat Qadeer, Rehan Hameed, Ofer Shacham, Preethi Venkatesan, Christos Kozyrakis, and Mark A. Horowitz. Convolution Engine: Balancing Efficiency and Flexibility in Specialized Computing. In *ISCA*, 2013.

[129] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, 2013.

[130] Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark Barrett. Counterexample-Guided Quantifier Instantiation for Synthesis in SMT. In *CAV*, 2015.

[131] Arch D. Robison. Impact of Economics on Compiler Optimization. In *Joint ACM-ISCOPE Conference on Java Grande*, 2001.

[132] Christopher J. Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. PTask: Operating System Abstractions to Manage GPUs As Compute Devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, 2011.

[133] Christopher J. Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. Dandelion: A Compiler and Runtime for Heterogeneous Systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, 2013.

[134] Vijay Saraswat et al. X10 language specification - version 2.3. Technical report, 2012.

[135] Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Jubi Taneja, and John Regehr. Souper: A Synthesizing Superoptimizer. *CoRR*, abs/1711.04422, 2017.

[136] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *ASPLOS*, 2013.

[137] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic optimization of floating-point programs with tunable precision. In *PLDI*, 2014.

[138] Thomas Schlömer, Benjamin Poppinga, Niels Henze, and Susanne Boll. Gesture Recognition with a Wii Controller. In *International Conference on Tangible and Embedded Interaction*, 2008.

[139] Rahul Sharma. Personal communication, June 2015.

[140] Hong Shen. Occam implementation of process-to-processor mapping on the Hathi-2 transputer system. *Microprocessing and Microprogramming*, 33(3), 1992.

[141] Pravin Shinde, Antoine Kaufmann, Kornilios Kourtis, and Timothy Roscoe. Modeling NICs with Unicorn. In *Proceedings of the Seventh Workshop on Programming Languages and Operating Systems*, PLOS '13, 2013.

[142] Pravin Shinde, Antoine Kaufmann, Timothy Roscoe, and Stefan Kaestle. We Need to Talk About NICs. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, HotOS '13, 2013.

[143] David Silver. Elon Musk Says Tesla Has A Blazingly Fast Onboard Computer To Aid Autopilot. `https://www.forbes.com/sites/davidsilver/2018/08/03/elon-musk-says-teslas-onboard-computer-is-blazingly-fast`, August 2018. Accessed: 2018-08-21.

[144] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet Transactions: High-Level Programming for Line-Rate Switches. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, 2016.

[145] Jadranka Skorin-Kapov. Tabu Search Applied to the Quadratic Assignment Problem. *INFORMS Journal on Computing*, 2(1):33–45, 1990.

[146] Aaron Smith, Jon Gibson, Bertrand Maher, Nick Nethercote, Bill Yoder, Doug Burger, Kathryn S. McKinle, and Jim Burrill. Compiling for EDGE Architectures. In *CGO*, 2006.

[147] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. Sketching Stencils. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, 2007.

[148] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, 2006.

[149] Venkatesh Srinivasan and Thomas Reps. Synthesis of machine code from semantics. In *PLDI*, 2015.

[150] Venkatesh Srinivasan, Tushar Sharma, and Thomas Reps. Speeding Up Machine-code Synthesis. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '16, 2016.

[151] Venkatesh Srinivasan, Ara Vartanian, and Thomas Reps. Model-assisted Machine-code Synthesis. *Proc. ACM Program. Lang.*, 1(OOPSLA):61:1–61:26, October 2017.

[152] Weibin Sun and Robert Ricci. Fast and Flexible: Parallel Packet Processing with GPUs and Click. In *Proceedings of the Ninth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '13, 2013.

[153] Mikio Takeuchi, Yuki Makino, Kiyokuni Kawachiya, Hiroshi Horii, Toyotaro Suzumura, Toshio Suganuma, and Tamiya Onodera. Compiling X10 to Java. In *ACM SIGPLAN X10 Workshop*, 2011.

[154] Shan Tang. Deep Learning Processor List. `https://basicmi.github.io/Deep-Learning-Processor-List`. Accessed: 2018-09-12.

[155] Trefis Team. Can Intel Challenge ARM's Mobile Dominance?, 2012.

[156] The Linley Group. Processor Watch: Getting Way Out of Box. `http://www.linleygroup.com/newsletters/newsletter_detail.php?num=5038`, 2013. Accessed: 2014-11-13.

[157] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. StreamIt: A Language for Streaming Applications. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, 2002.

[158] Frank Tip. A Survey of Program Slicing Techniques. Technical report, 1994.

[159] Emina Torlak and Rastislav Bodik. Growing Solver-Aided Languages with Rosette. In *Symp. on New Ideas in Programming and Reflections on Software (Onward!)*, 2013.

[160] Emina Torlak and Rastislav Bodik. A Lightweight Symbolic Virtual Machine for Solver-aided Host Languages. In *PLDI*, 2014.

[161] Peng Tu and David A. Padua. Automatic Array Privatization. In *International Workshop on Languages and Compilers for Parallel Computing*, 1994.

[162] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin, and Rajeev Alur. TRANSIT: Specifying Protocols with Concolic Snippets. In *PLDI*, 2013.

[163] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, 1999.

[164] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *CoRR*, abs/1802.04730, 2018.

[165] Henry S. Warren. *Hacker's Delight*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[166] Henry S. Warren. A Hacker's Assistant. October 2008.

[167] Wikipedia. List of ARM microarchitectures. `http://en.wikipedia.org/wiki/List_of_ARM_microarchitectures`, 2014. Accessed: 2014-11-13.

[168] Kent Wilken, Jack Liu, and Mark Heffernan. Optimal Instruction Scheduling Using Integer Programming. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, 2000.

[169] Tom Wilson, Gary Grewal, Ben Halley, and Dilip Banerji. An Integrated Approach to Retargetable Code Generation. In *Proceedings of the 7th International Symposium on High-level Synthesis*, ISSS '94, 1994.

[170] R. Glenn Wood and Rob A. Rutenbar. Fpga routing and routability estimation via boolean satisfiability. In *Proceedings of the 1997 ACM Fifth International Symposium on Field-programmable Gate Arrays*, FPGA '97, 1997.

[171] Xilinx. Vivado Design Suite. `http://www.xilinx.com/products/design-tools/vivado/`.

[172] Xilinx. FPGA Design Flow Overview, 2008.

[173] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A High-Performance Java Dialect. In *In ACM*, 1998.

[174] Mingxuan Yuan, Zonghua Gu, Xiuqiang He, Xue Liu, and Lei Jiang. Hardware/Software Partitioning and Pipelined Scheduling on Runtime Reconfigurable FPGAs. *ACM Trans. Des. Autom. Electron. Syst.*, 15(2), March 2010.

[175] Chenxin Zhang. *Dynamically Reconfigurable Architectures for Real-time Baseband Processing*. PhD thesis, Lund University, 2014.

[176] Qi Zheng, Yajing Chen, R. Dreslinski, C. Chakrabarti, A. Anastasopoulos, S. Mahlke, and T. Mudge. WiBench: An open source kernel suite for benchmarking wireless systems. In *Workload Characterization (IISWC), 2013 IEEE International Symposium on*, 2013.

[177] Noa Zilberman, Yury Audzevich, Georgina Kalogeridou, Neelakandan Manihatty-Bojan, Jingyun Zhang, and Andrew Moore. NetFPGA: Rapid prototyping of networking devices in open source. In *ACM SIGCOMM Computer Communication Review*, 2015.