Optimizing Data Processing Through Verified Lifting

Maaz Bin Safeer Ahmad

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2022

Reading Committee:
Alvin Cheung, Chair
Rastislav Bodik
Shoaib Kamil

Program Authorized to Offer Degree:
Computer Science & Engineering

University of Washington

Abstract

Optimizing Data Processing Through Verified Lifting

Maaz Bin Safeer Ahmad

Chair of the Supervisory Committee:
Affiliate Professor Alvin Cheung
Computer Science & Engineering

Some of the most exciting and impactful software systems being built today, from data analytics to artificial intelligence, rely on computationally expensive algorithms. A key hurdle in bringing these technologies to end-users is finding highly-optimized implementations for these algorithms. Unfortunately, the ever-increasing complexity of hardware architectures coupled with the diverse set of available hardware backends makes writing highly efficient and portable code challenging.

Over the past decade, a variety of domain-specific languages (DSLs) have been developed to automatically generate near-optimal device-specific implementations from high-level specifications. However, existing software written in general-purpose programming languages such as C++ or Java does not automatically benefit from these new DSL compilers. In fact, legacy software must first be re-written using the DSL's domain-specific application interface (API). This presents an enormous engineering burden as the amount of code that needs to be re-written may be large. Any optimizations in legacy software can obfuscate the embedded algorithms, making the code difficult to understand and translate. In addition, re-writing code using an entirely different interface risks the introduction of bugs and unintended changes to the semantics of the application. Finally, as hardware continues to evolve, new DSLs continue to emerge leaving developers perpetually chasing the state-of-the-art.

This thesis addresses the problem of how to automatically translate legacy data-processing

code to high-level domain-specific APIs. To do so, we build compilers that use *verified lifting* to first generate a *semantic summary* of the legacy code. The program summary, written using a high-level intermediate representation (IR), describes the semantics of the algorithm implemented in the legacy code. The compiler then uses the generated summary to produce new code in the target DSL's API. Our verified-lifting-based compilers use program synthesis to infer the program summaries without needing any re-write rules and verify that each summary is an exact semantic match to the input legacy code.

To demonstrate the feasibility and efficacy of our approach, we introduce three verified-lifting-based compilers. *Casper* is a tool that automatically re-writes legacy Java code to MapReduce frameworks such as Hadoop or Apache Spark. Since different implementations of the same algorithm are often possible within MapReduce frameworks, Casper uses a domain-specific cost model to identify highly efficient implementations of the legacy code. *Dexter* is a tool designed to automatically re-write legacy image-processing functions written in C++ to Halide, a modern DSL for image processing. Dexter introduces a novel algorithm that deconstructs the synthesis of program summaries into three distinct stages, allowing it to scale to complex real-world code. Finally, *Rake* is a tool that uses our verified lifting methodology to perform instruction selection within the Halide DSL. Modern hardware accelerators often implement complex domain-specific patterns in their instruction-set (ISA) and rule-based instruction selection is not always able to detect the best usage of these higher-level instructions. Rake uses verified lifting to re-write input expression into an IR of *uber-instructions* before lowering the IR representation to the target backend.

The tools presented in this thesis have been used to translate tens of thousands of lines of code, including code from real-world applications such as Adobe Photoshop. Together, they demonstrate the value of using program syntheses to implement provably correct lifting transformations to optimize data-processing code. We believe this thesis could be the prelude to a new generation of compilers, ones that can infer higher-level semantics of the input code

and port algorithms across different abstractions to unlock the best optimizations.

TABLE OF CONTENTS

Page

List of Figures iii

List of Tables vi

Chapter 1: Introduction 1
1.1 Modernizing Legacy Applications 3
1.2 Vector Instruction Selection for New Hardware Backends 6
1.3 Dissertation Outline & Contributions 7

Chapter 2: Background & Related Work 9
2.1 Reasoning About Program Semantics 9
2.2 Program Synthesis 10
2.3 Source-to-Source Compilers 12

Chapter 3: Leveraging MapReduce Frameworks for Data-Intensive Applications . . 14
3.1 Introduction 14
3.2 Overview 17
3.3 Synthesizing Program Summaries 21
3.4 Improving Summary Search 27
3.5 Finding Efficient Translations 32
3.6 Implementation 34
3.7 Evaluation 36
3.8 Related Work 49
3.9 Conclusion 49

Chapter 4: Translating Image Processing Libraries to Halide 51
4.1 Introduction 51

4.2 Related Work & Background . . . 55
4.3 Overview . . . 57
4.4 Finding Summaries For Image Processing Operations . . . 62
4.5 Implementation . . . 69
4.6 Evaluation . . . 75
4.7 Conclusion . . . 83

Chapter 5: Vector Instruction Selection using Verified Lifting . . . 85
5.1 Introduction . . . 85
5.2 Background & Overview . . . 88
5.3 Dynamic Expression Decomposition . . . 97
5.4 Abstracting Data Movement . . . 100
5.5 Synthesizing Swizzles . . . 104
5.6 Implementation . . . 106
5.7 Evaluation . . . 108
5.8 Related Work . . . 116
5.9 Conclusion . . . 117

Chapter 6: Reflections, Future Work and Conclusions . . . 119
6.1 Insights & Future Work . . . 119
6.2 Conclusion . . . 121

Bibliography . . . 123

Appendix A: Casper . . . 132
A.1 Proof Sketch For Soundness and Completeness . . . 132
A.2 Intermediate Representation Specification . . . 133
A.3 Code Generation Rules . . . 135
A.4 Program Analyzer Outputs . . . 136
A.5 Supplementary Experiments . . . 138

Appendix B: Dexter . . . 142
B.1 Symmetry Elimination and Memoization . . . 142
B.2 Analysis Based Suggestions . . . 144

LIST OF FIGURES

Figure Number Page

3.1 Translating the row-wise mean benchmark to MapReduce (Spark). 19

3.2 Casper's system architecture. Sequential code fragments (green) are translated into MapReduce tasks (orange). 21

3.3 Excerpt of Casper's IR for program summaries (PSs), a full description of which is provided in Appendix A.2. 23

3.4 Proof of soundness for the row-wise mean benchmark. 25

3.5 Incremental grammar generation. Casper generates a hierarchy of grammars to optimize search. 30

3.6 A runtime comparison of Casper-generated implementations against reference implementations. 41

3.7 StringMatch benchmark: Casper dynamically selects the optimal implementation for execution at runtime. 47

4.1 Dexter parses the input C++ function (shown on the left) into a DAG of smaller stages, then uses our 3-step synthesis algorithm to infer the semantics of each stage, expressed in a high-level IR (middle). Finally, code generation rules compile the IR specifications into executable Halide code (right). 52

4.2 Using Dexter to translate a 3×3 box blur filter from C++ to Halide, with casting removed for clarity. 60

4.3 Dexter's system architecture. Image processing functions in C++ (orange) are translated into Halide (green). 61

4.4 Search grammars used to synthesize summaries for image processing operations. Each summary represents a possible Halide tranlsation for the input operation. 63

4.5 Dexter synthesizes the ROI of an image processing operation by constructing a reduced version of the input code fragment. 65

4.6 Once the ROI has been determined, Dexter synthesizes a mapping for terminals found in the input code. 66

4.7 User annotation helps DEXTER determine that each index of the array is updated only once. 73
4.8 Synthesizing the mapping for terminal `noiseData` requires synthesizing the hash function. 74
4.9 A subset of DEXTER's code generation function `Gen`(). 76
4.10 An example of a tiled implementation that DEXTER can successfully deschedule to recover the program summary. 78
4.11 Speedup obtained from automatic translation followed by autoscheduling. For x86, we allow the autoscheduler to explore 32 candidates, while for ARM we allow the autoscheduler to augment its model using the x86 performance, but do not explore multiple schedules. Benchmarks are ordered by their x86 speedup; the ARM chart shows fewer benchmarks because we cannot execute hand-written SSE for comparison purposes. 81

5.1 Compilation Overview. RAKE intercepts Halide's compilation pipeline to synthesize device-specific implementations for vector expressions. 88
5.2 The Sobel Filter expressed in Halide. 89
5.3 Target-independent Halide IR vector expression produced by the Sobel filter algorithm. 90
5.4 An illustration of key differences in the HVX code generated by Halide's current optimizer and Rake for the Sobel filter. We do not count broadcasts of loop-invaraint expressions towards latency, as they will be moved outside the loop by LLVM. 92
5.5 The Sobel filter expression lifted to HVX uber-instructions. 94
5.6 A sample of HVX uber-instructions. 95
5.7 A swizzle-free sketch of an HVX expression. Data-movement is abstracted away using `??load` and `??swizzle`. 96
5.8 Synthesized data movement replaces `??load` and `??swizzle` to yield complete HVX implementations. 96
5.9 An illustration of how RAKE uses bottom-up program synthesis to lift the Sobel filter to the Uber-Instruction IR. 101
5.10 Definition of `??load` and `??swizzle`. 102
5.11 Speedups for RAKE over the default Halide HVX backend. Across the 22 benchmarks, RAKE improves performance by an average of 18% over the existing highly-optimized backend, which has been developed over years by Qualcomm and Google engineers. 110

5.12 RAKE uses search to find new optimization opportunities not considered by the existing Halide HVX backend. Here, `wild_{i|u}bb{x|}` represents a subtree of signed (`i`) or unsigned (`u`) `bb`-bit value that is either a scalar (no suffix) or vector (`x`). 112

A.1 The top 2 benchmarks with the most performance along with the bottom 2. The x-axis plots the size of input data, while the y-axis plots the runtime speedup over sequential implementations. 141

B.1 Four possible traces of DEXTER's expression generation algorithm. The steps in green indicate successful termination of the algorithm, whereas the steps in red indicate a pre-emptive rejection of the trace due to violation of symmetry breaking rules. 143

LIST OF TABLES

Table Number Page

3.1 Number of code fragments translated by Casper and their mean and max speedups compared to sequential implementations. 39

3.2 Summary of Casper's compilation performance. Values for the reference implementations are shown in parentheses. 44

3.3 With incremental grammar generation, Casper produces far less redundant summaries. 45

5.1 Compilation statistics for the Hexagon HVX backend. 115

A.1 The correlation of data shuffle and execution. 139

LIST OF ALGORITHMS

1 The counter-example guided inductive synthesis (CEGIS) algorithm. 11

2 CASPER's search algorithm. 29

3 Lifting expressions from Halide IR to the Uber-Instruction IR. 98

4 Lowering expressions from the Uber-Instruction IR to the target ISA. 105

ACKNOWLEDGMENTS

I have been exceedingly fortunate in my life to have found, time and again, people that have provided unparalleled love, support, and encouragement. I would like to express my sincerest gratitude to all my family, mentors, friends, and collaborators, without whom this dissertation would not be possible.

First and foremost, I'd like to thank my advisor, Alvin Cheung, for his mentorship and guidance throughout my Ph.D. It is difficult to overstate the care and attention Alvin has invested in my growth and success, both as a person and as a researcher. In addition to his incredible technical insights and influence on this work, he was a constant source of optimism, energy, and encouragement.

I would also like to thank Shoaib Kamil for his mentorship throughout my Ph.D. and during my internships at Adobe. In addition to his many intellectual contributions to this work, Shoaib's ardent advocacy of our vision was pivotal to its eventual realization.

I must thank the many great collaborators, co-authors, mentors, and colleagues I've had the privilege of working with during my Ph.D. None of my work would have been as exciting or well-rounded without insights from Ras Bodik, Andrew Adams, Jonathan Ragan-Kelley, Justin Gottschlich, AJ Root, Sam Kauffman, Jacob Van Geffen, Julie Newcomb, Mangpo Phothilimthana, Sarah Chasins, Chenglong Wang, Krzysztof Drewniak and everyone else in the PLSE, Databases and ICTD groups.

During the course of graduate school I've enjoyed the privilege of meeting so many incredible people, many of whom I am now lucky enough to now call my friends. I'd specifically like to thank Austin Schumacher, Anna Wehowsky, Deepali Nijhawan, Aman Nijhawan, Shrainik Jain, Srini Iyer, Sachin Mehta, Guna Prasaad, Blaine Taylor, Antoine Bosselut,

Kiron Lebeck, Niel Lebeck, Jiechen Chen, Dave Wadden, Terra Blevins, Jacob Schreiber, Menghsa Li, Mandar Joshi, Gagan Bansal, Chandrakana Nandi, Cong Yan, Jared Roesch, Sam Kauffman, and Chenglong Wang. Lastly, this Ph.D. would have never happened if it weren't for the encouragement, support, and generosity of Fahad Pervaiz, an incredible person and an even better friend.

Thank you to Sarfraz Raza, my undergraduate professor who nurtured my passion for computer science and was an island of encouragement when I felt most disillusioned. Thank you to Abid Mahmood, Naveed Butt and Munir Ahmad for taking me under their wing and providing valuable mentorship and support throughout my undergraduate studies. My friends Omar Farooq, Asad Naeem, Ammad Anwar, Bilal Waheed, Sarib Mahmood, Fahad Noor, Mohammed Ahmed, and Muhammed Abubakr, who always went above and beyond whenever I needed help.

Lastly but most importantly, I must thank my family: my parents, Safeer and Sameera, whose love and untold sacrifices gave me the confidence and the opportunity to succeed, and my siblings Musa and Maryam, who are okay.

DEDICATION

to my parents, Safeer Ahmad and Sameera Safeer.

Chapter 1

INTRODUCTION

Traditionally, performance-critical software applications were implemented in general-purpose programming languages (such as C++ or Fortran) and optimized through a combination of automatic compiler transformations and low-level manual performance tuning. However, the increased complexity in both hardware and software has rendered this approach difficult. On the one hand, general-purpose language compilers fail to automatically generate highly-optimized code [56], often leaving multiple orders of magnitude performance on the table. On the other hand, the increased diversity and heterogeneity of available hardware backends has made manually writing optimized device-specific low-level code more challenging now than ever before.

The inability of general-purpose compilers to discover optimal implementations can be attributed to three key factors. First, program analysis techniques are imperfect and cannot always determine whether an optimizing transformation, such as parallelizing a loop, is safe to perform. Second, the set of optimizations that yield optimal performance is highly domain-specific and sensitive to application variables, such as the size of input data. Finally, syntax-driven transformation rules are brittle, and compilers often explore only a small subset of possible implementations that are reachable through the set of available transformation rules.

The effectiveness of general-purpose compilers is limited by two factors: the inability ofprogram analysis techniques to determine what transformations are legal and the lack of accurateperformance models to determine what code should be generated from among the legal possibilities.Even simple, easily-analyzable loop nests that operate on arrays, such as the three nested loops ofmatrix multiplication, can suffer from the latter

The impracticality of manually optimizing large code bases in a landscape of diverse, rapidly evolving hardware has motivated the development of many domain-specific frameworks for building high-performance software. Under the hood, these frameworks are powered by either a library of finely-tuned hand-written functions that application developers can use to compose their algorithms or domain-specific compilers that automatically translate developers' high-level specifications to efficient low-level device code. While general-purpose language compilers suffer from their inability to pick the best optimizations due to a lack of insight into the program's high-level intent, or the inability of program analysis techniques to determine which optimizations are legal, domain-specific frameworks circumvent these issues by requiring programmers to express their algorithms using a domain-specific language (DSL) or application interface (API).

When developing new applications, domain-specific frameworks are a compelling solution that allow us to write maintainable and portable code, while enjoying high levels of performance. However, for legacy applications (i.e., applications written in general-purpose languages) to access the optimizations offered by these frameworks, they must first be re-written using the framework's API or DSL. Re-writing large parts of an existing application using an entirely new abstraction can be a daunting task requiring substantial time and effort. Furthermore, low-level optimizations in the legacy code, such as loop-tiling or the use of device-specific intrinsics, can obfuscate the algorithm embedded in the code. Not only does this make the legacy code harder to understand but an incorrect understanding may lead to critical bugs being introduced into the translated implementation. There are also practical concerns that cannot be discounted, such as having to train developers to use the new framework's API or DSL effectively. Finally, as hardware architectures and APIs evolve, so do the domain-specific frameworks evolve with them, requiring developers to perpetually re-write code in the latest and greatest APIs. This motivates the need to build tools that can automatically re-write legacy code into newer APIs.

Re-writing legacy code into newer APIs typically requires *lifting* programs written in low-level general-purpose languages (like C++ or Java) to higher-level DSLs or APIs. While

lowering programs expressed in a higher-level abstraction to a lower-level abstraction is well understood, *lifting* programs from a lower-level abstraction to a higher-level abstraction remains difficult. This is because the latter requires a way to infer higher-level semantic information about a program from the input code. Classical compilers rely heavily on syntax-directed translation (SDT) [6], which utilizes pre-defined code transformation rules to compile programs. Unfortunately, this technique is not very suitable for lifting programs since there is often a myriad of ways that an operation in the target DSL may be implemented in the input language. Designing a rule-set that is robust enough to match all possible syntactic patterns that may be encountered in the source code is challenging. In addition, maintaining a large corpus of rules and proving the correctness of the re-write system are also major burdens on the compiler developer.

In this thesis, we present a more scalable way to build lifting compilers by leveraging program synthesis and verification. Our *verified lifting* based approach uses search, rather than re-write rules, to automatically discover the high-level algorithms embedded in the input code and formally proves the correctness of each re-write using automated program verification techniques. The inferred algorithms are then used to generate executable code in the target framework's API or DSL. We introduce three new tools that implement verified lifting for three distinct domains. Casper and Dexter are verified lifting compilers that rejuvenate legacy software for the domains of large-scale data processing and high-performance image processing respectively. *Rake* is an instruction-selector that optimizes general-purpose vector expressions in the Halide DSL [73] intermediate representation (IR) by re-writing them using higher-level device intrinsics.

1.1 Modernizing Legacy Applications

High-performance software written in low-level general-purpose languages is highly susceptible to bit-rot. Even slight changes in the target hardware architecture, such as changes in the cache size or the introduction of new device intrinsics, can adversely impact performance and make past optimizations sub-optimal. Modern domain-specific frameworks offer robustness

against such architectural changes by using domain-specific optimizing compilers or allowing developers to easily and safely change the set of optimizations without affecting program semantics. However, porting legacy software to new frameworks can be a slow, expensive, and error-prone process. Automating this process is an ambitious task because, for a code translation tool to be viable, it must meet the following criteria:

- The translated code must be *correct*. Developers must be able to trust that the revised implementation in the target framework is semantically equivalent to the original program.

- The translated code must be *peformant*. There are often many ways to express a program in the target framework's API or DSL, with significant differences in the runtime performance. An effective code translator must be able to generate implementations that are not just correct but also maximize performance.

- The code translator must be *robust*. There are often many different ways to syntactically express an algorithm in low-level languages. Moreover, different optimization choices made by developers in the past can lead to dramatically different code structures for the same algorithm. An effective code translator must be robust to such syntactic variations.

- The code translator must be *scalable*. Real-world legacy software may require tens of thousands of lines of code to be translated; an effective translator should be able to scale to such tasks.

1.1.1 Casper*: Translating Sequential Java to MapReduce*

Our first case study focuses on the domain of large-scale data processing. We present a new tool Casper that modernizes sequential Java code by re-writing computation using MapReduce frameworks [35], such as Apache Spark [14]. In addition to simplifying the codebase and

improving code readability and maintainability, lifting sequential loops to the MapReduce programming paradigm enables frameworks to parallelize the computation across multiple cores or even multiple nodes across a distributed cluster. Since CASPER relies on program synthesis to lift the input program into a high-level IR, only simple code-generation rules are necessary to achieve the translation, making it highly robust. Additionally, CASPER ensures that the translation is sound by using a theorem prover to prove the semantic equivalence between the original and the translated code. Since there are often multiple valid MapReduce implementations of an input program, CASPER incorporates a cost-model into the search procedure to identify translations with the best-expected performance.

Our results show that CASPER can effectively translate a diverse set of real-world benchmarks, with the translated benchmarks performing up to 48.2× faster compared to the original implementations and were competitive even with other distributed implementations, including manual ones.

1.1.2 DEXTER*: Translating Image Processing Pipelines from C++ to Halide*

Our second case study focuses on the domain of image processing. We present DEXTER, a tool designed to translate legacy C++ implementations of image processing operations and pipelines to Halide [70], a high-performance DSL for image computation. Halide allows image processing pipelines to be expressed in two parts: an algorithm describing *what* needs to be computed, and a schedule that describes *how* it must be computed. The former describes the semantics of the program in a clean, portable representation. The latter describes the set of optimizations that must be performed for each target hardware backend. This separation of concerns makes Halide programs easier to maintain since developers can easily explore different optimizations without worrying about accidentally changing the program semantics. Moreover, Halide ships with a powerful auto-schedular that can automatically infer the set of necessary optimizations for a broad class of programs and different hardware backends.

Like CASPER, DEXTER also uses program synthesis to translate the input code and thus enjoys the same soundness and robustness advantages over syntax-driven compilers.

However, due to the size and complexity of the image processing pipelines, state-of-the-art synthesis algorithms are unable to reliably find a translation from the space of candidate Halide programs. Therefore, DEXTER introduces a new domain-specific synthesis algorithm that infers the semantics of input algorithms in three distinct stages. Our experiments show that DEXTER can automatically translate 264 functions from a set of 353 functions from the source code of Adobe Photoshop. These functions, implemented using over 36,000 lines of C++ code, include complexities such as vectorization, loop-tiling, type-casting, bitwise operations, reductions, and conditionals, all of which were beyond the scope of prior work [47]. By leveraging Halide and its auto-scheduler, our translated functions are not only more portable but perform up to 73× faster than the original implementations.

1.2 Vector Instruction Selection for New Hardware Backends

The ability to lift low-level code to a higher abstraction has applications beyond modernizing legacy code. The advent of specialized hardware architectures designed to accelerate specific workloads has presented a new challenge for optimizing compilers. These accelerators, such as the Qualcomm Hexagon DSP [29] that is now found on-die in millions of Android mobile phones [91], offer domain-specific optimizations in the form of intrinsics that implement important higher-level compute patterns. To fully utilize such accelerators, compilers must map the expressions in their low-level IR onto the exotic instruction sets provided by the accelerator. While mapping computations from programmer expressions into a sequence of processor instructions can be formulated as *instruction selection* as part of program compilation, choosing the optimal sequence of instructions for a given computation is especially difficult when considering vector instructions, which enable fine-grained parallel computation.

1.2.1 RAKE*: Mapping Halide IR Vector Expressions to Hexagon HVX*

In our third and final case study, we tackle the problem of mapping Halide IR expressions to the Hexagon HVX vector instruction set [29]. We introduce RAKE, a new instruction selec-

tor that leverages program synthesis to perform instruction selection for vectorized Halide expressions. Unlike prior work, Rake does not rely on manually crafted code patterns to match the input code. Given an input code sequence represented in the Halide IR, Rake first uses synthesis to *lift* the input code sequence into an intermediate representation (IR). This IR, called Uber-Instruction IR, is a high-level abstracted version of the target instruction set. Once lifted, Rake then lowers the Uber-Instruction IR into the concrete syntax of the target instruction set using synthesis. Like Casper and Dexter, all code transformations discovered by Rake are provably correct. We evaluate Rake by using it to generate code for the HVX accelerator and show that Rake-generated code can produce speedups of up to 2.1× over that generated by the existing Halide and LLVM [52] HVX pipeline (which has been developed and tuned by Qualcomm, Google, and LLVM developers), as Rake finds instruction sequences that are not considered by the existing Halide pattern-matching rules and the instruction selection pass in LLVM.

1.3 Dissertation Outline & Contributions

The remainder of this dissertation is outlined below:

- Chapter 2 presents the necessary background on program synthesis and verification to contextualize the work in this thesis. We also discuss related work on building source-to-source compilers and motivate the need for program synthesis based code translators.

- Chapter 3 presents Casper, a verified lifting compiler for modernizing large-scale data-processing Java code. It details key design decisions that make Casper robust and scalable, including dynamic grammar generation, incremental grammar expansion, and the incorporation of a cost model into the search process. We present evidence that our synthesis-based approach is broadly applicable and performs more performant code than rule-based approaches, matching even the performance of manually translated code.

- Chapter 4 presents DEXTER. DEXTER expands on the methodology introduced in CASPER and demonstrates how domain-specific insights can be used to perform algorithm inference incrementally, decomposing a single intractable algorithm inference problem into multiple tractable sub-problems. We demonstrate how DEXTER can scale to real-world use-cases, translating tens of thousands of lines of Adobe Photoshop source code.

- Chapter 5 describes how verified lifting can be leveraged to perform vector instruction selection within DSL compilers. We present RAKE, an instruction selector for the Hexagon HVX instruction set implemented within the Halide DSL. We explain how Uber-Instructions can be used to reveal high-level code patterns implemented in the input programs. Our results demonstrate that our synthesis-powered instruction selection algorithm can detect optimization opportunities missed by Halide's and LLVM's existing pattern-matching infrastructure, yielding performance improvements as high as 2.1× using the same optimization schedules.

Together, these chapters support the thesis at the core of this dissertation: (1) synthesis powered code-translators can be used to automatically modernize legacy code by leveraging modern domain-specific frameworks, and (2) verified lifting can be used within DSL compilers to perform vector instruction selection for modern hardware accelerators.

Chapter 2
BACKGROUND & RELATED WORK

In this chapter, we first discuss how to verify equivalence between two programs automatically. Then, we introduce the concept of program synthesis and formalize the code-translation problem within the framework of syntax-guided program synthesis. Finally, we discuss relevant literature on source-to-source compilation of programs.

2.1 Reasoning About Program Semantics

To understand how legacy data-processing code can be automatically ported to a new DSL or API, we must first understand how to reason about the semantics of the input code. Formally speaking, given a legacy program statement p in a general-purpose programming language and a candidate summary ps that describes the algorithm implemented in the input code, we want to decide whether p is a valid *postcondition* for the input code. A postcondition is a predicate statement that is always true at the end of a block of code, under all possible executions.

The validity of a postcondition with respect to a piece of code can be established by constructing Hoare style *verification conditions* [45]. Verification conditions are Boolean predicates that state what must be true *before* p is executed in order for ps to be a valid postcondition of p. Verification conditions can be systematically generated for imperative program statements, including those processed by Casper and Dexter [92, 58]. However, each loop statement requires an extra *loop invariant* to construct an inductive proof. Loop invariants are Boolean predicates that are true before and after every execution of the loop body regardless of how many times the loop executes. Given a set of verification conditions vc and any necessary loop invariants $inv_1, \ldots, inv_n$, we can check the validity of the

postcondition over all possible input program states σ by querying an SMT solver as follows:

$$\forall \sigma.\ vc(p, ps, inv_1, \ldots, inv_n, \sigma) \tag{2.1}$$

We share concrete examples of postconditions and the verification conditions used to verify them against a piece of input program code in Chapter 3 and Chapter 3.

2.2 Program Synthesis

Program Synthesis is the task of searching for programs that satisfy some user-provided constraints [42]. In our setting, the programs that we search for are the semantic summaries (postconditions) and the constraints are the automatically generated verification conditions. As discussed earlier, when dealing with input code that contains loops, verification conditions require loop invariants to construct an inductive proof. These loop invariants therefore must either be provided by the user as an input or they must also be synthesized along with the postcondition.

2.2.1 Syntax-Guided Program Synthesis

Syntax-guided synthesis (SyGuS) [8] is a search-based technique for constructing programs. As input, it takes a set of semantic constraints known as the *specification* as well as a set of syntactic constraints, which we refer to as the *grammar*. As output, it generates programs or expressions from the grammar that satisfy the semantic constraints. We can frame the code translation problem as a SyGuS task as follows:

$$\exists\, ps,\ inv_1, \ldots, inv_n \in \mathcal{G}.\ \ \forall \sigma.\ vc(p, ps, inv_1, \ldots, inv_n, \sigma) \tag{2.2}$$

In other words, our goal is to find a program summary ps and any required invariants $inv_1, \ldots, inv_n$ from the provided grammar $\mathcal{G}$ such that for all possible program states σ, the verification conditions vc for the input program p are true.

Input: (g) A grammar of candidate program summaries

Input: (vc) The set of verification conditions derived from the input program

Output: (ps) A program summary that is valid over the bounded verification domain

```
1  Function Synthesize(g, vc)
2      Φ ← {...}                                   ▷ Set of random program states
3      ps, inv_{1..n} ← GenerateCandidate(g, vc, Φ)
4      while ps ≠ null do
5          φ ← BoundedVerify(ps, inv_{1..n}, vc)   ▷ Attempt bounded verification
6          if phi ≠ null then
7              return ps, inv_{1..n}               ▷ No counter-example found; summary is correct
8          else
9              Φ ← Φ ∪ φ                           ▷ Add counter-example to the set of program states
10     return null
```

Algorithm 1: The counter-example guided inductive synthesis (CEGIS) algorithm.

2.2.2 Counter-Example Guided Inductive Synthesis

The primary hurdle towards solving synthesis problems, such as the one described in 2.2 is scalability. The space of candidate postconditions and loop invariants defined by $\mathcal{G}$ may be exponentially large. A number of approaches have been developed to solve such synthesis problem [18, 40] using different algorithms, such as constraint-based search [84], enumerative search [65], or stochastic search [79]. The work presented in this thesis primarily builds upon the counter-example guided inductive synthesis (CEGIS) algorithm [84].

Algorithm 1 shows the core CEGIS algorithm. The algorithm is an iterative interaction between two modules: a candidate program summary generator and a bounded model checker. The candidate summary generator takes as input the IR grammar g, the verification conditions for the input code fragment vc, and a set of concrete program states Φ. To start the process, the synthesizer populates Φ with a few randomly chosen states, and generates

program summary candidate ps and any needed invariants $inv_1, \ldots, inv_n$ from g such that $\forall \sigma \in \Phi$. $vc(ps, inv_1, \ldots, inv_n, \sigma)$ is true. Next, the bounded model checker verifies whether the candidate program summary holds over the bounded domain. If it does, the algorithm returns ps as the solution. Otherwise, the model checker returns a counter-example state ϕ such that $vc(ps, inv_1, \ldots, inv_n, \phi)$ is false. The algorithm adds ϕ to Φ and restarts the program summary generator. This continues until either a program summary is found that passes bounded model checking or the search space is exhausted.

2.3 Source-to-Source Compilers

Many efforts translate programs from low-level languages into high-level DSLs. MOLD [69], a source-to-source compiler, relies on syntax-directed rules to convert native Java programs to Apache Spark. Many source-to-source compilers have been built similarly for other domains [61, 47, 9, 17]. [93] employs syntax-driven techniques to translate image processing code in Python by transforming the Python abstract syntax tree into lower-level Cython [32] code, which is a mixture of Python and C, while performing several program transformations. Similarly, SEJITS specializers [48, 23] use syntax-driven code generation for translating subsets of Python code into various languages. Such systems invariably handle only a subset of ways input programs can encode their operations and do not provide the kinds of correctness guarantees possible with program verification. Furthermore, the rules that they rely on are difficult to devise and brittle to code pattern changes.

There is also prior work on using synthesis to generate efficient implementations and optimize programs. [83] synthesizes MapReduce solutions from user-provided input and output examples. QBS [28, 26, 27] and STNG [47] both use synthesis to convert low-level languages to specialized high-level DSLs for database applications and stencil computations, respectively. This dissertation takes inspiration from these prior approaches by applying verified lifting to construct compilers. Unlike prior work, however, we (1) address the problem of verifier failures and designs a grammar hierarchy to prune away non-performant summaries, (2) incorporate a dynamic cost model and runtime monitoring module for adaptively choosing

from different implementations at runtime, (3) propose domain-specific algorithms to incrementally synthesize program summaries, and (4) extend verified lifting to new applications such as vector instruction selection.

Chapter 3

LEVERAGING MAPREDUCE FRAMEWORKS FOR DATA-INTENSIVE APPLICATIONS

MapReduce is a popular programming paradigm for developing large-scale, data-intensive computation. Many frameworks that implement this paradigm have recently been developed. To leverage these frameworks, however, developers must become familiar with their APIs and rewrite existing code. In this chapter, we introduce Casper, a new tool that automatically translates sequential Java programs into the MapReduce paradigm. Casper identifies potential code fragments to rewrite and translates them in two steps: (1) Casper uses *program synthesis* to search for a program summary (i.e., a functional specification) of each code fragment. The summary is expressed using a high-level intermediate language resembling the MapReduce paradigm and verified to be semantically equivalent to the original using a theorem prover. (2) Casper generates executable code from the summary, using either the Hadoop, Spark, or Flink API. We evaluated Casper by automatically converting real-world, sequential Java benchmarks to MapReduce. The resulting benchmarks perform up to 48.2× faster compared to the original.

3.1 Introduction

MapReduce [35], a popular paradigm for developing data-intensive applications, has varied and highly efficient implementations [11, 14, 10, 63]. All these implementations expose an application programming interface (API) to developers. While the concrete syntax differs slightly across the different APIs, they all require developers to organize their computation into *map* and *reduce* stages in order to leverage their optimizations.

While exposing optimization via an API shields application developers from the com-

plexities of distributed computing, this approach contains a major drawback: for legacy applications to leverage MapReduce frameworks, developers must first understand the existing code's function and subsequently re-organize the computation using mappers and reducers. Similarly, novice programmers, unfamiliar with the MapReduce paradigm, must first learn the different APIs in order to express their computation accordingly. Both require a significant expenditure of time and effort. Further, each code rewrite or algorithm reformulation opens another opportunity to introduce bugs.

One way to alleviate these issues is to build a compiler that translates code written in another paradigm (e.g., imperative code) into MapReduce. Classical compilers, like logical to physical query plan compilers [49], use pattern matching rules, i.e., the compilers contain a number of rules that recognize different input code patterns (e.g., a sequential loop over lists) and translate the matched code fragment into the target (e.g., a single-stage map and reduce). As in query compilers, designing the rules is challenging: they must be both *correct*, i.e., the translated code should have the same semantics as the input, and sufficiently *expressive* to capture the wide variety of coding patterns that developers use to express their computations. We are aware of only one such compiler that translates imperative Java programs into MapReduce [69], and the number of rules involved in that compiler makes it difficult to maintain and modify.

This chapter describes a new tool, Casper, that translates sequential Java code into semantically equivalent MapReduce programs. Rather than relying on rules to translate different code patterns, Casper is inspired by prior work on *cost-based query optimization* [80], which considers compilation to be a dynamic search problem. However, given that the inputs are general-purpose programs, the space of possible target programs is much larger than it is for query optimization. To address this issue, Casper leverages recent advances in program synthesis [40, 18] to search for MapReduce programs into which it can rewrite a given input sequential Java code fragment. To reduce the search space, Casper searches over the space of *program summaries*, which are expressed using a *high-level intermediate language (IR)* that we designed. As we discuss in §3.3.1, the IR's design succinctly expresses

computations in the MapReduce paradigm yet remains sufficiently easy to translate into the concrete syntax of the target API.

To search for summaries, CASPER first performs lightweight program analysis to generate a description of the space of MapReduce programs that a given input code fragment *might* be equivalent to. The search space is also described using our high-level IR. CASPER then uses an off-the-shelf *program synthesizer* to perform the search, but it is guided by an *incremental search algorithm* and our *domain-specific cost model* to speed the process. A *theorem prover* is used to check whether the found program summary is indeed semantically equivalent to the input. Once proved, the summary is translated into the concrete syntax of the target MapReduce API. Since the performance of the translated program often depends on input data characteristics (e.g., skewness), CASPER generates multiple semantically equivalent MapReduce programs for a given input and produces a monitor module that switches among them based on runtime statistics; the monitor and switcher are automatically generated during compilation.

Compared to prior approaches, CASPER does not require compiler developers to design or maintain any pattern matching rules. Furthermore, the entire translation process is completely automatic. We evaluated CASPER using a number of benchmarks and real-world Java applications and demonstrated both CASPER's ability to translate an input program into MapReduce equivalents and the significant performance improvements that result.

In summary, our work makes the following contributions:

- We propose a new high-level intermediate representation (IR) to express the semantics of sequential Java programs in the MapReduce paradigm. The language is succinct to be easily translated into multiple MapReduce APIs, yet expressive to describe the semantics of many real-world benchmarks written in a general-purpose language. Furthermore, programs written in our IR can be automatically checked for correctness using a theorem prover (§3.4.1). The IR, being a high-level language, also lets us perform various *semantic optimizations* using our cost model (§3.5).

- We describe an efficient search technique for program summaries expressed in the IR without requiring any pattern matching rules. Our technique is both *sound* and *complete* with respect to the input search space. Unlike classical compilers, which rely on pattern matching to drive translation, our technique leverages program synthesis to dynamically search for summaries. Our technique is novel in that it incrementally searches for summaries based on cost. It also uses verification failures to systematically prune the search space and a hierarchy of search grammars to speed the summary search. This lets us translate benchmarks that have not been translated in any prior work (§3.4.1).

- There are often multiple ways to express the same input as MapReduce programs. Therefore, our technique can generate multiple semantically equivalent MapReduce versions of the input. It also automatically inserts code that collects statistics during program execution to adaptively switch among the different generated versions (§3.5.2).

- We implemented our methodology in Casper, a tool that converts sequential Java programs into three MapReduce implementations: Spark, Hadoop, and Flink. We evaluated the feasibility and effectiveness of Casper by translating real-world benchmarks from 7 different suites from multiple domains. Across 55 benchmarks, Casper translated 82 of 101 code fragments. The translated benchmarks performed up to 48.2× faster compared to the original ones and were competitive even with other distributed implementations, including manual ones (§3.7).

3.2 Overview

This section describes how we model the MapReduce programming paradigm and demonstrates by example how Casper translates sequential code into MapReduce programs.

3.2.1 MapReduce Operators

MapReduce organizes computation using two operators: *map* and *reduce*. The map operator has the following type signature:

$$\boldsymbol{map} : (mset[\tau],\ \lambda_m) \longrightarrow mset[(\kappa, \nu)]$$
$$\boldsymbol{\lambda_m} : \tau \longrightarrow mset[(\kappa, \nu)]$$

Input into *map* is a multiset (i.e., bag) of type τ and a unary transformer function λ_m, which converts a value of type τ into a multiset of key-value pairs of types κ and ν. The map operator then concurrently applies λ_m to every element in the multiset and returns the union of all multisets generated by λ_m.

$$\boldsymbol{reduce} : (mset[(\kappa, \nu)],\ \lambda_r) \longrightarrow mset[(\kappa, \nu)]$$
$$\boldsymbol{\lambda_r} : (\nu, \nu) \longrightarrow \nu$$

Input into *reduce* is a multiset of key-value pairs and a binary transformer function λ_r, which combines two values of type ν to produce a final value. The reduce operator first groups all key-value pairs by key (also known as shuffling) and then uses λ_r to combine, in parallel, the bag of values for each key-group into a single value. The output of *reduce* is another multiset of key-value pairs, where each pair holds a unique key. If the transformer function λ_r is commutative and associative, then *reduce* can be further optimized by concurrently applying λ_r to pairs of values in a key-group.

Casper's goal is to translate a sequential code fragment into a MapReduce program that is expressed using the *map* and *reduce* operators. The challenges in doing so are: (1) identify the correct sequence of operators to apply, and (2) implement the corresponding transformer functions. We next discuss how Casper overcomes these challenges.

3.2.2 Translating Imperative Code to MapReduce

Casper takes in Java code with loop nests that sequentially iterate over data and translates the code into a semantically equivalent MapReduce program to be executed by the target

```
// Program Summary:
//    m = map(reduce(map(mat, λm1), λr), λm2)
//    λm1 : (i, j, v) → {(i, v)}
//    λr : (v1, v2) → v1 + v2
//    λm2 : (k, v) → {(k, v/cols)}
int[] rwm(int[][] mat, int rows, int cols) {
    int[] m = new int[rows];
    for (int i = 0; i < rows; i++) {
        int sum = 0;
        for (int j = 0; j < cols; j++)
            sum += mat[i][j];
        m[i] = sum / cols;
    }
    return m;
}
```

(a) Input: Sequential Java code

```
RDD rwm(RDD mat, int rows, int cols) {
  RDD m = mat.mapToPair(e -> new Tuple(e.i, e.v));
  m = m.reduceByKey((v1, v2) -> (v1 + v2));
  m = m.mapValues(v -> (v / cols));
  return m;
}
```

(b) Output: Apache Spark code

Figure 3.1: Translating the row-wise mean benchmark to MapReduce (Spark).

framework. To demonstrate, we show how CASPER translates a real-word benchmark from the Phoenix suite [74].

As shown in Figure 3.1a, the benchmark takes as input a matrix (`mat`) and computes, using nested loops, the column vector (`m`) containing the mean value of each row in the matrix. Assume the code is annotated with a *program summary* that helps with the translation into MapReduce. The program summary, written using a high-level intermediate representation (IR), describes how the output of the code fragment (i.e., `m`) can be computed using a series of *map* and *reduce* stages from the input data (i.e., `mat`), as shown in lines 1 to 5 in Figure 3.1a. While the summary is not executable, translating from that IR into the concrete syntax of a MapReduce framework (say, Spark) would be much easier than translating from the original input code. This is shown in Figure 3.1b where the *map* and *reduce* primitives from our summary are translated into the corresponding Spark API calls.

Unfortunately, the input code does not have such a summary, which must therefore be inferred. CASPER does this via program synthesis and verification, as we explain in §3.3.

3.2.3 System Architecture

Figure 3.2 shows CASPER's overall design. We now discuss the three primary modules that make up CASPER's compilation pipeline.

First, the *program analyzer* parses the input code into an Abstract Syntax Tree (AST) and uses static program analysis to identify code fragments for translation (§3.6.1). In addition, for each identified code fragment, it prepares: (1) a *search space description* encoded using our high-level IR that lets the synthesizer search for a valid program summary (§3.3.1), and (2) *verification conditions* (VCs) (§3.3.3) to automatically ascertain that the induced program summary is semantically equivalent to the input.

Next, the *summary generator* synthesizes and verifies program summaries (§3.3.4 and §3.4.1). To speed up the search, it partitions the search space so that it can be efficiently traversed using our incremental synthesis algorithm (§3.4.2).

Once a summary is inferred, the *code generator* translates it into executable code. CASPER

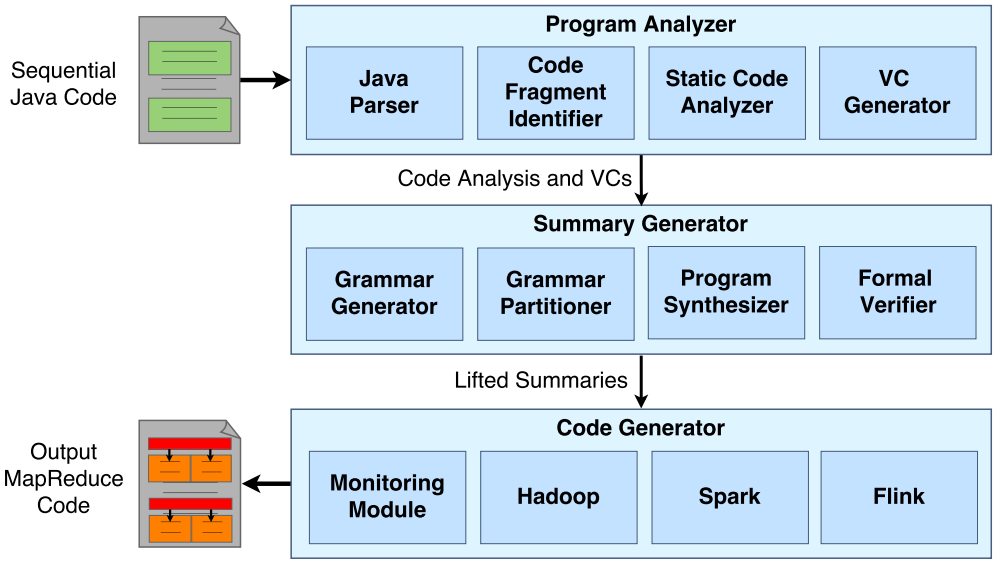


Figure 3.2: Casper's system architecture. Sequential code fragments (green) are translated into MapReduce tasks (orange).

currently supports three MapReduce frameworks: Spark, Hadoop, and Flink. Additionally, this component also generates code that collects data statistics to adaptively choose among different implementations during runtime (§3.5.2).

3.3 Synthesizing Program Summaries

As discussed, Casper discovers a program summary for each code fragment before translation. Technically, a program summary is a *postcondition* [45] of the input code that describes the program state after the code fragment is executed. In this section, we explain: (1) the IR Casper uses to express summaries, (2) how Casper verifies a summary's validity, and (3) the search algorithm Casper uses to find valid summaries given a search space description.

3.3.1 A High-level IR for Program Summaries

One approach to synthesize summaries directly searches in programs written in the target framework's API. This does not scale well; Spark alone offers over 80 high-level operators,

even though many of them have similar semantics and differ only in their implementation or syntax (e.g., map, flatMap, filter). To speed up synthesis, we instead search in programs written in a high-level IR that abstracts away syntactical differences and describes only the functionality of a few essential operators. The goals of the IR are: (1) to express summaries that are translatable into the target API, and (2) to let the synthesizer efficiently search for summaries that are equivalent to the input program. To address these goals, CASPER's IR models two MapReduce primitives that are similar to the *map* and *fold* operators in Haskell (see §3.2.1). In addition, our IR models the *join* primitive, which takes as input two multisets of key-value pairs and returns all pairs of elements with matching keys. The IR does not currently model the full range of operators across different MapReduce implementations; however, it already lets CASPER capture a wide array of computations expressible using the paradigm and is sufficiently general to be translatable into different MapReduce APIs while keeping the search problem tractable, as we demonstrate in §3.7.

Figure 3.3 shows a subset of CASPER's IR, used to express both program summaries and the search space. The IR assumes that program summaries are expressed in the stylized form shown in Figure 3.3 as *PS*, which states that each output variable v (i.e., a variables updated in the code fragment), must be computed using a sequence of *map*, *reduce* and *join* operations over the inputs (e.g., the arrays or collections being iterated). While doing so ensures that the summary is translatable into the target API, the implementations of λ_m and λ_r for the *map* and *reduce* operators depend on the code fragment being translated. We leave these functions to be synthesized and restrict the body of λ_m to a sequence of *emit* statements, where each *emit* statement produces a single key-value pair, and the body of λ_r is an expression that evaluates to a single value of the required type. Besides *emit*, the bodies of λ_m and λ_r's can consist of conditionals and other operations on tuples, as shown in Figure 3.3. The output of the MapReduce expression is an associative array of key-value pairs; the unique key v_{id} for each variable is used to access the computed value of that variable. Appendix A.2 lists the full set of types and operators that our IR supports.

$$
\begin{aligned}
PS &:= \forall v.\ v = MR \mid \forall v.\ v = MR[v_{id}] \\
MR &:= map(MR, \lambda_m) \mid reduce(MR, \lambda_r) \mid join(MR, MR) \mid data \\
\lambda_m &:= f : (val) \rightarrow \{Emit\} \\
\lambda_r &:= f : (val_1, val_2) \rightarrow Expr \\
Emit &:= emit(Expr, Expr) \mid if(Expr)\ emit(Expr, Expr) \mid \\
&\quad\ if(Expr)\ emit(Expr, Expr)\ else\ Emit \\
Expr &:= Expr\ op\ Expr \mid op\ Expr \mid f(Expr, Expr, ...) \mid \\
&\quad\ n \mid var \mid (Expr, Expr)
\end{aligned}
$$

$v \in Output\ Variables$ $\qquad$ $v_{id} \in Variable\ ID,$

$op \in Operators$ $\qquad$ $f \in Library\ Methods$

Figure 3.3: Excerpt of Casper's IR for program summaries (PSs), a full description of which is provided in Appendix A.2.

3.3.2 Defining the Search Space

In addition to program summaries, Casper also uses the IR to describe the search space of summaries for the synthesizer. It does so by generating a *grammar* for each input code fragment, like the one shown in Figure 3.3. The synthesizer traverses the grammar by expanding on each production rule and checks whether any generated candidate constitutes a valid summary (as explained in §3.3.3).

To generate the search space grammar, Casper analyzes the input code to extract the following properties and their type information:

1. Variables in scope at the beginning of the input code
2. Variables that are modified within the input code
3. The operators and library methods used

The code analyzer extracts these properties using standard program analyses. It computes (1) and (2) using live variable and dataflow analysis [6], and it computes (3) by scanning functions that are invoked in the input code. We currently assume that input variables are not aliased to each other and put guards on the translated code to ensure that is the case.[1] Appendix A.4 shows the analysis results for the TPC-H Q6 benchmark, and we discuss the limitations of our program analyzer module implementation in §3.6.1.

Given this information, the summary generator builds a search space grammar that is specialized to the code fragment being translated. Figure 3.5 shows sample grammars that are generated for the code shown in Figure 3.1a.[2] The input code uses addition and division; hence, the grammar includes addition and division in its production rules for λ_m and λ_r. Furthermore, Casper also uses type information of variables to prune invalid production rules in the grammar. For instance, if the output variable v is of type int, the final operation in the synthesized MapReduce expression must evaluate to a value of type int. Since the output type of a reduce operation is inferred from the type of its input, we can propagate this information to restrict the type of values the reduce operation accepts. To make synthesis tractable and the search space finite, Casper imposes recursive bounds on the production rules. For instance, it limits the number of MapReduce operations a program summary can use and the number of *emit* statements in a single transformer function. In §3.4.2, we discuss how Casper further specializes the search space by changing the set of production rules available in the grammar or specifying different recursive bounds.

3.3.3 Verifying Program Summaries

To search for a valid summary within the search space, Casper requires a way to check whether a candidate summary is semantically equivalent to the input code. It does so using

[1]Thus, if variable handles `v1` and `v2` are both inputs into the same code fragment, Casper wraps the translated code as: `if (v1 != v2) { [Casper translated code] } else { [original code] }`. Computing precise alias information requires more engineering [82] and does not impact our approach.

[2]Refer to Appendix A.4 to see how a grammar can be encoded in our IR.

$$invariant(m, i) \equiv \quad 0 \leq i \leq rows \quad \wedge \quad m = map(reduce(map(mat[0..i], \lambda_{m1}), \lambda_r), \lambda_{m2})$$

(a) Outer loop invariant

Initiation	$(i = 0) \rightarrow Inv(m, i)$
Continuation	$Inv(m, i) \wedge (i < rows) \rightarrow Inv(m[i \mapsto sum(mat[i])/cols], i + 1)$
Termination	$Inv(m, i) \wedge \neg(i < rows) \rightarrow PS(m, i)$

(b) Verification conditions to ascertain the correctness of the program summary PS given loop invariant Inv.

Figure 3.4: Proof of soundness for the row-wise mean benchmark.

standard techniques in program verification, namely, by creating *verification conditions* based on Hoare logic, introduced in section 2.1.

The general problem of inferring the strongest loop invariants or postconditions is undecidable [58, 92]. Unlike prior work, however, two factors make our problem solvable: first, our summaries are restricted to only those expressible using the IR described in §3.3.1, which lacks many problematic features (e.g., pointers) that a general-purpose language would have. Moreover, we are interested only in finding loop invariants that are *strong enough* to establish the validity of the synthesized program summaries.

As an example, Figure 3.4a shows an outer loop invariant Inv, which can be used to prove the validty of the program summary shown in Figure 3.1a. Figure 3.4b shows the verification conditions Casper constructs to state what the program summary and invariant must satisfy. We can check that this loop invariant and program summary are indeed valid based on Hoare logic as follows. First, the *initiation* clause asserts that the invariant holds before the loop, i.e., when `i` is zero. This is true because the invariant asserts that the MapReduce expression is true only for the first i rows of the input matrix. Hence, when `i` is zero, the MapReduce expression is executed on an empty dataset, and the output value for each row is 0. Next, the *continuation* clause asserts that after one more execution of

the loop body, the i^{th} index of output vector `m` should hold the mean for the i^{th} row of the matrix `mat`. This is true since the value of `i` is incremented inside the loop body, which implies that the mean for the i^{th} row has been computed. Finally, the *termination* condition completes the proof by asserting that if the invariant is true, and `i` has reached the end of the matrix, then the program summary PS must now hold as well. This is true since `i` now equals the number of rows in the matrix, and the loop invariant asserts that `m` equals the MapReduce expression executed over the entire matrix, which is the same assertion as our program summary.

Casper formulates the search problem for finding program summaries by constructing the verification conditions for the given code fragment and leaving the body of the summary (and any necessary invariants for loops) to be synthesized. For the program summary and invariants, the search space is expressed using the same IR as discussed in §3.3.1. After the synthesizer has identified a candidate summary and invariants, Casper sends them and the verification conditions to a theorem prover (see §3.4.1), and to the code generator to generate executable MapReduce code if the program summary is proven to be correct.

3.3.4 Search Strategy

Casper uses an off-the-shelf program synthesizer, Sketch [81], to infer program summaries and loop invariants. Sketch takes as input: (1) a set of candidate summaries and invariants encoded as a grammar (e.g., Figure 3.3), and (2) the correctness specification for the summary in the form of verification conditions. It then attempts to find a program summary (and any invariants needed) using the provided grammar such that the verification conditions hold true.

The universal quantifier in Eq.2.2 makes the synthesis problem challenging. Therefore, Casper uses a two-step process to ensure that the found summary is valid. First, it leverages Sketch's *bounded model checking* to verify the candidate program summary over a finite (i.e., "bounded") subset of all possible program states. For example, Casper restricts the maximum size of the input dataset and the range of values for integer inputs. Finding a

solution for this weakened specification can be done very efficiently by the synthesizer. Once a candidate program summary can be verified for the bounded domain, CASPER passes the summary to a theorem prover to determine its soundness over the entire domain, which is more expensive computationally. CASPER currently translates the summary along with an automatically generated proof script to Dafny [53] for full verification. This two-step verification makes CASPER's synthesis algorithm sound, without compromising efficiency.

Synthesis Algorithm

CASPER builds upon the CEGIS algorithm (Algorithm 1) introduced in 2.2.2. A limitation of the CEGIS algorithm is that, while efficient, the found program summary might be true only for the finite domain and thus will be rejected by the theorem prover when checking for validity over the entire domain. In this case, CASPER dynamically changes the search space grammar to exclude the candidate program summary that does not verify and restarts the synthesizer to generate a new candidate summary using the preceding algorithm. We discuss this process in detail in §3.4.1.

3.4 Improving Summary Search

We now discuss the techniques CASPER uses to make the search for program summaries more robust and efficient.

3.4.1 Leveraging Verifier Failures

As mentioned, the program summary that the synthesizer returns can fail theorem prover validation due to the bounded domain used during search. For instance, assume we bound the integer inputs to have a maximum value of 4 in the synthesizer. In this bounded domain, the expressions `v` and `Math.min(4,v)` (where `v` is an input integer) are deemed to be equivalent even though they are not equal in practice. While prior work [28, 47] simply fails to translate such benchmarks if the theorem prover rejects the candidate summary, CASPER uses a two-

phase verification technique to eliminate such candidates. This ensures that CASPER's search is complete with respect to the search space defined by the grammar.

To achieve completeness, CASPER must first prevent summaries that failed the theorem prover from being regenerated by the synthesizer. A naive approach would be to restart the synthesizer until a new summary is found, assuming that the algorithm implemented by the synthesizer is non-deterministic. However, this approach is incomplete because the algorithm may never terminate since it can continually return the same incorrect summary. Instead, CASPER modifies the search space to ensure forward progress. Recall from §3.3.4 that the search space for candidate summaries $\{c_1, \ldots, c_n\}$ is specified using an input grammar that is generated by the program analyzer and passed to the synthesizer. Thus, to prevent a candidate c_f that fails the theorem prover from being repeatedly generated from grammar G, CASPER simply passes in a new grammar $G - \{c_f\}$ to the synthesizer. This is implemented by passing additional constraints to the synthesizer to block a summary from being regenerated.

Theorem. CASPER's algorithm for inferring program summaries is sound and complete with respect to the given search space.

A proof sketch for this theorem is provided in Appendix A.1.

Algorithm 2 shows how CASPER infers program summaries and invariants. CASPER calls the synthesizer to generate a candidate summary `c` (Line 7) and attempts to verify `c` by passing it to the theorem prover (Line 9). If verification fails, `c` is added to Ω, the set of incorrect summaries, and the synthesizer is restarted with a new grammar $G - \Omega$. We explain the full algorithm in §3.4.3.

In §3.7.3, we provide experimental results that illustrate how our two-phase verification algorithm effectively finds program summaries even when faced with verification failures.

3.4.2 Incremental Grammar Generation

Although CASPER's search algorithm is complete, the space of possible summaries to consider remains large. To address this, CASPER incrementally expands the search space for program

Input: (a) The results of program analysis on the input code

Input: (vc) The set of verification conditions derived from the input program

Output: (Δ) A set of valid program summaries for the input code

```
1  Function InferSummary(a, vc)
2      g ← GenerateGrammar(a)          ▷ Generate a grammar of candidate summaries
3      Γ ← GenerateClasses(G)          ▷ Partition grammar into hierarchical classes
4      Ω ← {}                          ▷ Set of summaries that failed verification
5      Δ ← {}                          ▷ Set of summaries that passed verification
6      for γ ∈ Γ do
7          c ← Synthesize(γ − Ω − Δ, vc)
8          while c ≠ null do
9              if Verify(c, vc) then                  ▷ Attempt full verification
10                 Δ ← Δ ∪ c
11             else
12                 Ω ← Ω ∪ c
13         if Δ ≠ {} then
14             return Δ                       ▷ Search completes successfully
15     return null              ▷ Search space exhausted, no valid summaries found
```

Algorithm 2: CASPER's search algorithm.

summaries to speed up the search. It does this by: (1) adding new production rules to the grammar, and (2) increasing the number of times that each product rule is expanded.

The benefits of this approach are twofold. First, since the search time for a valid summary is proportional to search space size, CASPER often finds valid summaries quickly, as our experiments show. Second, since larger grammars are more syntactically expressive, the found summaries are likely to be more expensive computationally. Hence, biasing the search towards smaller grammars likely produces program summaries that run more efficiently. Although this is not sufficient to guarantee optimality of generated summaries, our experiments

Property	G_1	G_2	G_3
MapReduce Sequence	m	m → r	m → r → m
Number of Emits in λ_m	1	2	2
Key-Value Type	int	int	int or Tuple<int,int>

$$G1 \coloneqq map(mat, \lambda_m)$$
$$\lambda_m \coloneqq \begin{cases} (i,j,v) \rightarrow [(i,j)] \\ (i,j,v) \rightarrow [(i,v)] \\ (i,j,v) \rightarrow [(j, v+i)] \\ (i,j,v) \rightarrow [(i+j, v)] \\ \vdots \end{cases}$$

$$G2 \coloneqq reduce(map(mat, \lambda_m), \lambda_r)$$
$$\lambda_m \coloneqq \begin{cases} (i,j,v) \rightarrow [(i,v)] \\ (i,j,v) \rightarrow [(j, v+i)] \\ (i,j,v) \rightarrow [(i,j),(v,1)] \\ \vdots \end{cases}$$
$$\lambda_r \coloneqq \begin{cases} (v_1, v_2) \rightarrow v_1 \\ (v_1, v_2) \rightarrow v_2 + 4 \\ (v_1, v_2) \rightarrow v_1 + v_2 \\ \vdots \end{cases}$$

$$G3 \coloneqq map(reduce(map(mat, \lambda_{m1}), \lambda_r), \lambda_{m2})$$
$$\lambda_{m1} \coloneqq \begin{cases} (i,j,v) \rightarrow [(i,v)] \\ (i,j,v) \rightarrow [\big(i,(v,i)\big)] \\ (i,j,v) \rightarrow [(i+1, j-v),(i,v)] \\ \vdots \end{cases}$$
$$\lambda_r \coloneqq \begin{cases} (v_1, v_2) \rightarrow v_1 \\ (v_1, v_2) \rightarrow v_1 + v_2 \\ (v_1, v_2) \rightarrow (v_1.1, v_2.2) \\ \vdots \end{cases}$$
$$\lambda_{m2} \coloneqq \begin{cases} (k,v) \rightarrow [(k,v),(v,k)] \\ (k,v) \rightarrow [\big((v.1,k), v.2\big)] \\ (k,v) \rightarrow [(k, v/cols)] \\ (k,v) \rightarrow if(v > i)[(k,v)] \\ \vdots \end{cases}$$

Figure 3.5: Incremental grammar generation. Casper generates a hierarchy of grammars to optimize search.

show that in practice Casper generates efficient solutions (§3.7.2).

To implement incremental grammar generation, Casper partitions the space of program summaries into different *grammar classes*, where each class is defined based on these syntactical features: (1) the number of MapReduce operations, (2) the number of *emit* statements in each *map* stage, (3) the size of key-value pairs emitted in each stage, as inferred from the types of the key and value, and (4) the length of expressions (e.g., $x + y$ is an expression of length 2, while $x + y + z$ has a length of 3). All of these features are implemented by altering production rules in the search space grammar. A grammar hierarchy is created such that all

program summaries expressible in a grammar class G_i are also expressible in a higher level class, i.e., G_j where $j > i$.

3.4.3 CASPER*'s Search Algorithm for Summaries*

Algorithm 2 shows CASPER's algorithm for searching program summaries. The algorithm begins by constructing a grammar G using the results of program analysis A on the input code. First, CASPER partitions the grammar G into a hierarchy of grammar classes Γ (Line 3). Then, it incrementally searches each grammar class $\gamma \in \Gamma$, invoking the synthesizer to find summaries in γ (Line 7). Each summary (and invariants) the synthesizer returns is checked by a theorem prover (Line 9); CASPER saves the set of correct program summaries in Δ and all summaries that fail verification in Ω. Each synthesized summary (correct or not) is eliminated from the search space, forcing the synthesizer to generate a new summary each time, as explained in §3.4.1. When the grammar γ is exhausted, i.e., the synthesizer has returned null, CASPER returns the set of correct summaries Δ if it is non-empty. Otherwise, no valid solution was found, and the algorithm proceeds to search the next grammar class in Γ. If Δ is empty after exploring every grammar in Γ, i.e., no summary could be found in the entire search space, the algorithm returns null.

3.4.4 Row-wise Mean Revisited

We now illustrate how `InferSummary` searches for program summaries using the row-wise mean benchmark discussed in §3.2.2. Figure 3.5 shows three sample (incremental) grammars CASPER generated as a result of calling `GenerateClasses` (Algorithm 2, Line 3) along with their properties. For example, the first class, G_1, consists of program summaries expressed using a single *map* or *reduce* operator, and the transformer functions λ_m and λ_r are restricted to emit only one integer key-value pair. A few candidates for λ_m are shown in the figure. For instance, the first candidate, $(i, j, v) \rightarrow [(i, j)]$, maps each matrix entry to its row and column as the output.

If `InferSummary` fails to find a valid summary in G_1 for the benchmark, it advances to the next grammar class, G_2. G_2 expands upon G_1 by including summaries that consist of two *map* or *reduce* operators, and each λ_m can emit up to 2 key-value pairs. The search next moves to G_3, where G_3 expands upon G_2 with summaries that include up to three *map* or *reduce* operators, and the transformers can emit either integers or tuples. As shown in Figure 3.1a, a valid summary is finally found in G_3 and added to Δ. Search continues in G_3 for other valid summaries in the same grammar class. The search terminates after all valid summaries in G_3, i.e., those returned by the synthesizer and fully verified, are found. This includes the one shown in Figure 3.1a.

3.5 Finding Efficient Translations

There often exist many semantically equivalent MapReduce implementations for a given sequential code fragment, with significant performance differences. Many frameworks come with optimizers that perform low-level optimizations (e.g., fusing multiple map operators). However, performing *semantic transformations* is often difficult. For instance, at least three different implementations of the StringMatch benchmark exist in MapReduce, and they differ in the type of key-value pairs the *map* stage emits (see §3.7.4). Although it is difficult for a low-level optimizer to discover these equivalences by syntax analysis, Casper can perform such optimization because it searches for a high-level program summary expressed using the IR. We now discuss Casper's use of a cost model and runtime monitoring module for this purpose.

3.5.1 Cost Model

Casper uses a cost model to evaluate different semantically equivalent program summaries that are found for a code fragment. Because Casper aims to translate data-intensive applications, its cost model estimates data transfer costs as opposed to compute costs.

Each synthesized program summary is a sequence of *map*, *reduce* and *join* operations. The semantics of these operations are known, but the transformer functions that they use

(λ_m and λ_r) are synthesized and determine the operation's cost. We define the cost functions of the *map*, *reduce* and *join* operations below:

$$cost_m(\lambda_m, N, W_m) = W_m * N * \sum_{i=1}^{|\lambda_m|} sizeOf(emit_i) * p_i \tag{3.1}$$

$$cost_r(\lambda_r, N, W_r) = W_r * N * sizeOf(\lambda_r) * \epsilon(\lambda_r) \tag{3.2}$$

$$cost_j(N_1, N_2, W_j) = W_j * N_1 * N_2 * sizeOf(emit_j) * p_j \tag{3.3}$$

The function $cost_m$ estimates the amount of data generated in the *map* stage. For each *emit* statement in λ_m, the size of the key-value pair emitted is multiplied by the probability that the *emit* statement will execute (p_i). The values are then summed to get the expected size of the output record. The total amount of data emitted during the map stage equals to the product of expected record size and the number of times λ_m is executed (N). The cost function for a *reduce* stage, $cost_r$, is defined similarly, except that λ_r produces only a single value and the cost is adjusted based on whether λ_r is commutative and associative. The function ϵ returns 1 if these properties hold; otherwise, it returns W_{csg}. The cost function for *join* operations, $cost_j$, is defined over: the number of elements in the two input datasets (N_1 and N_2), the selectivity of the join predicate (p_j), and the size of the output record. W_m, W_r and W_j are the weights assigned to the map, reduce and join operations. W_{csg} is the penalty for a non-commutative associative reduction. In our experiments, we used the values 1, 2, 2 and 50 for these weights, respectively based on our empirical studies.

To estimate the cost of a program summary, we simply sum the cost of each individual operation. The first operator in the pipeline takes symbolic variables $N_{0..i}$ as the number of records. For each subsequent stage, we use the number of key-value pairs generated by the current stage, expressed as a function over $N_{0..i}$:

$$cost_{mr}([(op_1, \lambda_1), (op_2, \lambda_2), \ldots], N_{0..i}) = cost_{op1}(\lambda_1, N, W) + cost_{mr}([(op_2, \lambda_2), \ldots], count(\lambda_1, N_{0..i}))$$

The function *count* returns the number of key-value pairs generated by a given stage. For *map* stages, this equals $\sum_{i=1}^{|emits|} p_i$; for *reduce* stages, it equals the number of unique key values on which the reducer was executed; for joins, it equals $N_1 * N_2 * p_j$.

3.5.2 Dynamic Cost Estimation

The cost model computes the cost of a program summary as a function of input data size N. We use this cost model to compare the synthesized summaries both statically and dynamically. First, calling `findSummary` returns a list of verified summaries that were found. Casper then uses the cost model to prune summaries when a less costly one exists in the list. Not all summaries can be compared that way, however, since they could depend on the value distribution of the input data or how frequently a conditional evaluates to true, as shown in the candidates for grammar G_3's λ_{m1} in Figure 3.5.

In such cases, Casper generates code for all remaining summaries that have been validated, and it uses a runtime monitoring module to evaluate their costs dynamically when the generated program executes. As the program executes, the runtime module samples values from the input dataset (Casper currently uses first-k values sampling, although different sampling method may be used). It then uses the samples to estimate the probabilities of conditionals by counting the number of data elements in the sample for which the conditional will evaluate to true. Similarly, it counts the number of unique data values that are emitted as keys. These estimates are inserted into Eqn 3.1 and Eqn 3.2 for each program summary to get comparable cost values. Finally, the summary with the lowest cost is executed at runtime. Hence, if the generated program is executed over different data distributions, it will run different implementations, as illustrated in §3.7.4.

3.6 Implementation

We implemented Casper using the Polyglot framework [66] to parse Java code into an abstract syntax tree (AST). Casper traverses the program AST to identify candidate code fragments, performs program analysis, and generates target code. We now describe the Java

features supported by our compiler front-end. We also discuss how CASPER identifies code fragments for translation and generates executable code from the verified program summary.

3.6.1 Supported Language Features

To translate a code fragment, CASPER must first successfully generate verification conditions for that fragment (as explained in §3.3.3). CASPER can currently do this for basic Java statements, conditionals, functions, user-defined types, and loops.

Basic Types CASPER supports all basic Java arithmetic, logical, and bit-wise operators. It can also process reads and writes into primitive arrays and common Java Collection interfaces, such as `java.util.{List, Set,Map}`. CASPER can be extended to support other data structures, such as `Stack` or `Queue`.

User-defined Types CASPER traverses the program AST to find declarations of all types that were used in the code fragment being translated. It then dynamically translates and adds these types to the IR as `structs`, as shown in Appendix A.2.

Loops CASPER computes VCs for different types of loops (`for`, `while`, `do`), including those with loop-carried dependencies [6], after applying classical transformations [6] to convert loops into the `while(true){...}` format.

Methods CASPER handles methods by inlining their bodies. Polymorphic methods can be supported similarly by inlining different versions with conditionals that check the type of the host object at runtime. Recursive methods and methods with side-effects are not currently supported because they are unlikely to gain any speedup by being translated to MapReduce.

External Library Methods CASPER supports common library methods from standard Java libraries (e.g., `java.lang.Math` methods) by modeling their semantics explicitly using

the IR. Users can similarly provide models for other methods that Casper currently does not support. [3]

3.6.2 Code Fragment Identification

Casper traverses the input AST to identify code fragments that are amenable for translation by searching for loops that iterate one or more data structures (e.g., a list or an array). We target loops since they are most likely to benefit from translation to MapReduce. We have kept our loop selection criteria lenient to avoid false negatives.

3.6.3 Code Generation

Once an identified code fragment is translated, Casper replaces the original code fragment with the translated MapReduce code. It also generates "glue" code to merge the generated code into the rest of the program. This includes creating a `SparkContext` (or an `ExecutionEnvironment` for Flink), converting data into `RDD`s (or Flink's `DataSet`s), broadcasting required variables, etc. Since some API calls (such as Spark's `reduceByKey`) are not defined for non-commutative associative transformer functions, Casper uses these API calls only if the generated code is indeed commutative and associative (otherwise, Casper uses safe, albiet less efficient, transformations, such as `groupByKey`). Finally, Casper also generates code for sampling input data and dynamic switching, as discussed in §3.5.2. Appendix A.3 presents a subset of code-generation rules for the Spark API.

3.7 Evaluation

In this section, we present a comprehensive evaluation of Casper on a number of dimensions, including its ability to: (1) handle diverse and realistic workloads, (2) find efficient translations, (3) compile efficiently, and (4) extend to support other IRs and cost-models in the future. All experiments were conducted on an AWS cluster of 10 m3.2xlarge instances

[3] We provide examples of library function and type models in Appendix A.2.

(1 master node, 9 core nodes), where each node contains an Intel Xeon 2.5 GHz processor with 8 vCPUs, 30 GB of memory, and 160 GB of SSD storage. We used the latest versions of all frameworks available on AWS: Spark 2.3.0, Hadoop 2.8.3, and Flink 1.4.0. The data files for all experiments were stored on HDFS.

3.7.1 ***Feasibility Analysis***

We first assess CASPER's ability to handle a variety of data-processing applications. Specifically, we determine whether: (1) CASPER can generate verification conditions for a syntactically diverse set of programs, (2) our IR can express summaries for a broad range of data-processing workloads, and (3) CASPER's ability to find such summaries. To this end, we used CASPER to optimize a set of 55 diverse benchmarks from real-world applications that contained a total of 101 translatable code fragments.

Basic Applications For benchmarking, we assembled a set of small applications from prior work and online repositories. These applications, summarized below, contain a diverse set of code patterns commonly found in data-processing workloads (e.g., aggregations, selections, grouping, etc), as follows:

- *Bigλ* [83] consists of several data analysis tasks such as *sentiment analysis*, *database operations* (e.g., selection and projection), and *Wikipedia log processing*. Since Bigλ generates code from input-output examples rather than from an actual implementation, we recruited computer science graduate students in our department to implement a representative subset of the benchmarks from their textual descriptions. This resulted in 211 lines of code across 7 files.

- *Stats* is a set of benchmarks CASPER automatically extracted from an online repository for the statistical analysis of data [55]. Examples include *Covariance*, *Standard Error* and *Hadamard Product*. The repository contains 1162 lines of code across 12 Java files, mostly consisting of vector and matrix operations.

- *Ariths* is a set of simple mathematical functions and aggregations collected from prior work [25, 31, 75, 37]. Examples include *Min*, *Max*, *Delta*, and *Conditional Sum*. The suite contains 245 lines of code than span 11 files.

Across the 3 suites, CASPER identified 38 code fragments, of which 35 were successfully translated. One code-fragment that CASPER failed to translate used a variable-sized kernel to convolve a matrix; two others required broadcasting data values to many reducers during the map stage, but such mappers are currently inexpressible in our IR due to the absence of loops.

Traditional Data-Processing Benchmarks Next, we used CASPER to translate a set of well-known, data-processing benchmarks that resemble real-world workloads:

- We manually implemented Q1, Q6, Q15 and Q17 from the *TPC-H* benchmark using sequential Java and used CASPER to translate the Java implementations to MapReduce. The selected queries cover many SQL features, such as aggregations, joins and nested queries.

- *Phoenix* [74] is a collection of standard MapReduce problems—such as *3D Histogram*, *Linear Regression*, *KMeans*, etc.—used in prior work [69]. Since the original sequential implementations were written in C, we used the sequential Java translations of the benchmarks from prior work in our experiments. The suite consists of 440 lines of code across 7 files.

- *Iterative* represents two popular iterative algorithms that we manually implemented into sequential versions: *PageRank* and *Logistic Regression Based Classification*.

CASPER successfully translated all 4 TPC-H queries and both iterative algorithms. It successfully translated 7 of 11 from the Phoenix suite. Three of the 4 failures were due to the IR's lack of support for loops inside transformer functions. One benchmark failed to synthesize within 90 minutes, causing CASPER to time out.

Benchmark Suite	Number Translated	Mean Speedup	Max Speedup
Phoenix	7 / 11	14.8x	32x
Ariths	11 / 11	12.6x	18.1x
Stats	18 / 19	18.2x	28.9x
Bigλ	6 / 8	21.5x	32.2x
Fiji	23 / 35	18.1x	24.3x
TPC-H	10 / 10	31.8x	48.2x
Iterative	7 / 7	18.4x	28.8x

Table 3.1: Number of code fragments translated by Casper and their mean and max speedups compared to sequential implementations.

Real-World Applications Fiji [38] is a popular distribution of the ImageJ [46] library for scientific image analysis. We ran Casper on the source code of four Fiji packages (aka plug-ins). *NL Means* is a plugin for denoising images via the non-local-means algorithm [21] with optimizations [33]. *Red To Magenta* transforms images by changing red pixels to magenta. *Temporal Median* is a probabilistic filter for extracting foreground objects from a sequence of images. *Trails* averages pixel intensities over a time window in an image sequence. These packages, authored by different developers, contain 1411 lines of code that span 5 files. Of the 35 candidate code fragments identified across all 4 packages, Casper successfully optimized 23. Three of the failures were caused by the use of unsupported types or methods from the ImageJ library since we did not model them using the IR, and the search timed out for the remaining 9.

Table 3.1 summarizes the results of our feasibility analysis. Of the 101 individual code fragments identified by the compiler across all benchmarks, Casper translated 82. We manually inspected all code files to ensure that Casper's code fragment identifier missed no translatable code fragments. Overall, the benchmarks form a syntactically diverse set of

applications.[4]

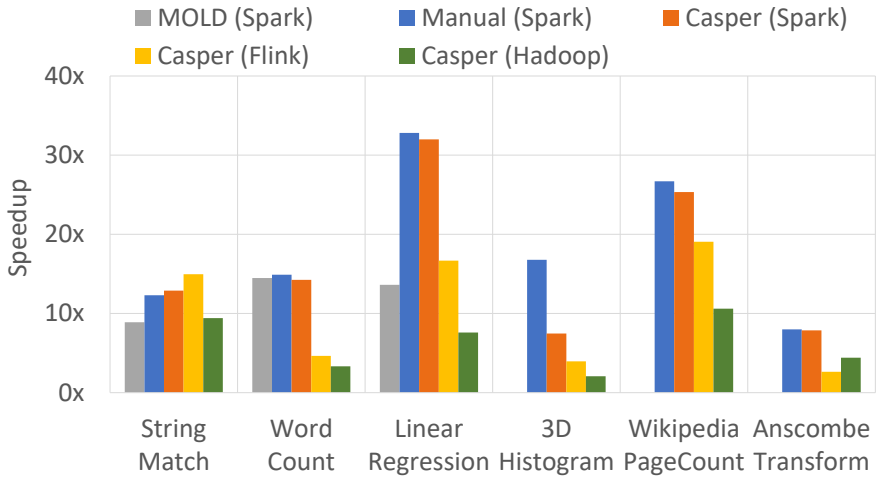
Because MOLD is not publicly available, we obtained the generated code from the MOLD authors for the benchmarks used in its evaluation [69]. Of the 7 Phoenix benchmarks, MOLD could not translate 2 (*PCA* and *KMeans*). Another 2 (*Histogram* and *Matrix Multiplication*) generated semantically correct translations that worked well for multi-core execution but failed to execute on the cluster because they ran out of memory. For the remaining 3 benchmarks (*Word Count*, *String Match* and *Linear Regression*), MOLD generated working implementations. In contrast, CASPER translated 4 of the 7 Phoenix benchmarks. For *PCA* and *KMeans*, CASPER translated and successfully executed a subset of all the loops found, while translation failed for the other loops and the *Matrix Multiplication benchmark* for reasons explained above.

3.7.2 Performance of the Translated Benchmarks

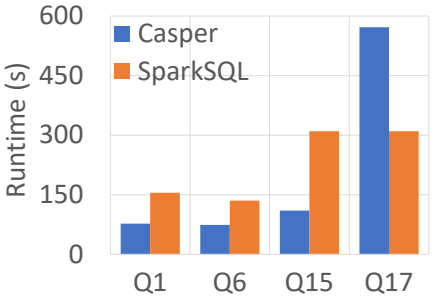
CASPER helps an application leverage the optimization and parallelization provided by MapReduce implementations by translating their code. Therefore, in this section, we examine the quality of the translations CASPER produced by comparing their performance to that of reference distributed implementations.

We used CASPER to translate summaries for these benchmarks to three popular implementations of the MapReduce programming model: Hadoop, Spark, and Flink. The translated Spark implementations, along with their original sequential implementations, were executed on three synthetic datasets of sizes 25GB, 50GB, and 75GB. Overall, the Spark implementations CASPER generated are 15.6× faster on average than their sequential counterparts, with a max improvement of up to 48.2×. Table 3.1 shows the mean and max speedup observed for each benchmark suite using Spark on a 75GB dataset. We also executed the Hadoop and Flink implementations generated by CASPER for a subset of 10 benchmarks, some of which are shown in Figure 3.6a. The average speedups observed (over

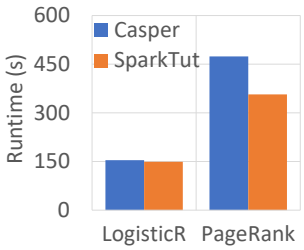
[4]We summarize the syntactic features of the code fragments in Appendix A.5.1.



(a) Casper achieves speedup competitive with manual translations



(b) TPC-H benchmarks



(c) Iterative algorithms

Figure 3.6: A runtime comparison of Casper-generated implementations against reference implementations.

the 10 benchmarks) by these implementations are 6.4× and 10.8×, respectively. These results show that Casper can effectively improve the performance of applications by an order of magnitude by retargeting critical code fragments for execution on MapReduce frameworks.

Figure 3.6a plots the speedup achieved by the MOLD-generated implementations for *String Match*, *Word Count*, and *Linear Regression*. The Spark translations MOLD generated for these benchmarks performed 12.3× faster on average than the sequential versions. The solutions generated by Casper for *String Match* and *Linear Regression* were faster than those generated by MOLD by 1.44× and 2.34×, respectively. For *String Match*, Casper found an

efficient encoding to reduce the amount of data emitted in the map stage (see §3.7.4), whereas MOLD emitted a key-value pair for every word in the dataset. Furthermore, MOLD used separate MapReduce operations to compute the result for each keyword; Casper computed the result for all keywords in the same set of operations. For *Linear Regression*, MOLD discovered the same overall algorithm as Casper except its implementation zipped the input RDD with its index as a pre-processing step, almost doubling the size of input data and hence the amount of time spent in data transfers.

For the *Ariths*, *Stats*, *Bigλ*, and *Fiji* benchmarks, we recruited Spark developers through UpWork.com to manually rewrite the benchmarks since reference distributed implementations were not available.[5] Figure 3.6a compares the performance of (a subset of) Casper-generated implementations to handwritten benchmark implementations over the 75GB dataset. Results show that the Casper-generated implementations perform competitively, even with those manually written by developers. In fact, of the 42 hand-translated benchmark implementations, 24 used the same high-level algorithm as the one generated by Casper, and most of the remaining ones differ by using framework-specific methods instead of an explicit map/reduce (e.g., using Spark's built-in filter, sum, and count methods). However, these variations did not cause a noticeable performance difference. One interesting case was the 3D Histogram benchmark, where the developer exploited knowledge about the data to improve runtime performance. Specifically, the developer recognized that since RGB values always range between 0-255, the histogram data structure would never exceed 768 values. Therefore, the developer used Spark's more efficient *aggregate* operator to implement the solution. Casper, not knowing that pixel RGB values are bounded, assumed that the number of keys could grow to be arbitrarily large and that using the aggregate operator could cause out-of-memory errors, hence it generated a single stage map and reduce instead.

For *PageRank* and *Logistical Regression*, we compared Casper against the implementations found in the Spark Tutorials [86] (see Figure 3.6c). The reference PageRank implemen-

[5]Appendix A.5.2 describes the hiring criteria.

tation was 1.3× faster than the one CASPER generated on a dataset of about 2.25 billion graph edges and running 10 iterations. This is because CASPER currently does not generate any `cache()` statements, nor does it co-partition data. Deciding when to cache can lead to further performance gains. Prior work [20] suggested heuristics for inserting such statements into Spark algorithms that could be integrated into CASPER's code generator to improve performance for iterative workloads. For *Logistical Regression*, we found no noticeable difference in performance.

For TPC-H queries, we compared the performance of Spark code generated by CASPER against SparkSQL's implementation. Figure 3.6b plots the results of this experiment. For Q1, Q6 and Q15, CASPER implementations executed 2×, 1.8× and 2.8× faster, respectively, than SparkSQL on a scale factor of 100. For Q1 and Q6, we attribute this to the extra data shuffling performed by the SparkSQL query plan. In Q15, SparkSQL's query plan scanned the *lineitem* relation twice, whereas CASPER's implementation did so only once, resulting in worse runtime performance. For Q17, SparkSQL executed 1.7× faster because it performed better scheduling of the query operators than the CASPER-generated implementation. In sum, results show that the CASPER-generated implementations the TPC-H benchmarks have comparable performance to those implemented directly using the MapReduce frameworks. Yet, developers need not learn different MapReduce APIs by using CASPER.

3.7.3 Compilation Performance

We next evaluate CASPER's compilation performance. We discuss the time taken by CASPER to compile the benchmarks, the effectiveness of CASPER's two-phase verification strategy, the quality of the generated code, and incremental grammar generation.

Compile Time

On average, CASPER took 11.4 minutes to compile a single code fragment. However, the median compile time for a single benchmark was only 2.1 minutes: for some benchmarks, the synthesizer discovered a low-cost solution during the first few grammar classes, letting

Benchmark Suite	Mean Time (s)	Mean LOC	Mean # Op	Mean TP Failures
Phoenix	944	13.8 (13.1)	2.3 (2.1)	0.35
Ariths	223	9.4 (7.6)	1.6 (1.2)	4
Stats	351	7.6 (5.8)	1.8 (1.8)	0.6
Bigλ	112	13.6 (10)	1.8 (2.0)	0.4
Fiji	1294	7.2 (7.4)	1.4 (1.6)	0.1
TPC-H	476	5.9 (n/a)	7.25 (n/a)	0
Iterative	788	3.3 (3.7)	4.5 (3.5)	2

Table 3.2: Summary of Casper's compilation performance. Values for the reference implementations are shown in parentheses.

Casper terminate search early. Table 3.2 shows the mean compilation time for a single benchmark by suite.

Two-Phase Verification

In our experiments, the candidate summary generator produced at least one incorrect solution for 13 out of the 101 successfully translated code-fragments. The synthesizer proposed a total of 76 incorrect summaries across all benchmarks. Table 3.2 lists the average number of times the theorem prover rejected a solution for each benchmark suite. As an example, the *Delta* benchmark computes the difference between the largest and smallest values in the dataset. It incurred 7 rounds of interaction with the theorem prover before the candidate generator found a correct solution due to errors from bounded model checking (discussed in §3.4.1).

Generated Code Quality

In addition to measuring the runtime performance of Casper-generated implementations, we manually inspected the code generated by Casper and compared it to the reference imple-

Benchmark	With Incremental Grammar	Without Incremental Grammar
WordCount	2	827
StringMatch	24	416
Linear Regression	1	94
3D Histogram	5	118
YelpKids	1	286
Wikipedia PageCount	1	568
Covariance	5	11
Hadamard Product	1	484
Database Select	1	397
Anscombe Transform	2	78

Table 3.3: With incremental grammar generation, Casper produces far less redundant summaries.

mentations for two code quality metrics: lines of code (LOC) and the number of MapReduce operations used. Table 3.2 shows the results of our analysis. Implementations generated by Casper were comparable and did not use more MapReduce operations or LOC than were necessary to implement a given task. Note that the LOC pertain to individual code fragments, not entire benchmarks.

Incremental Grammar Generation

We also measured the effectiveness of incremental grammar generation in optimizing search. To measure its impact on compilation time, we used Casper to translate benchmarks without incremental grammar generation and compared the results. The synthesizer was allowed to run for 90 minutes, after which it was manually killed. The results of this experiment are summarized in Table 3.3. Exhaustively searching the entire search space produced hundreds of more expensive solutions. The cost of searching, verifying, and sorting all these superfluous solutions dramatically increased overall synthesis time. In fact, Casper timed

out for every benchmark in that set (which represents a slowdown by at least one order of magnitude).

3.7.4 Dynamic Tuning

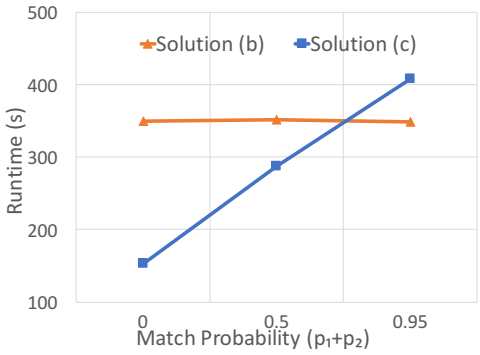
The final set of experiments evaluated the runtime monitor module and whether the dynamic cost model could select the correct implementations. As explained in §3.5.2, the performance of some solutions depends on the distribution of the input data. Therefore, we used CASPER to generate different implementations for the StringMatch benchmark (Figure 3.7a). Figure 3.7d shows three (out of 400+) correct candidate solutions, with their respective costs based on the formula described in §3.5.1 and the following values for data-type sizes: 40 bytes for String, 10 bytes for Boolean and 28 bytes for a tuple of Boolean Objects. Solution (a) can be disqualified at compile time because it will have a higher cost than solution (b) for all possible data distributions. However, the cost of solutions (b) and (c) cannot be statically compared due to the unknowns p_1 and p_2 (the respective probabilities that the conditionals will evaluate to true and a key-value pair will be emitted). The values of p_1 and p_2 depend on the input data, i.e., how often the keywords appear in the text, and thus can be determined only dynamically at run-time.

CASPER handles this by generating a runtime monitor in the output code. The monitor samples the input data (first 5000 values) in each execution to estimate values for unknown variables in the cost formulas. The estimated values are then plugged back into the original cost functions (Eqn 3.1 and 3.2), and the solution with the lowest cost is then executed.

We executed solutions (b) and (c) on three 75GB datasets with different amounts of skew: one with no matching words (i.e., (c) emits nothing), one with 50% matching words (i.e., (c) emits a key-value pair for half of the words in the dataset), and one with 95% matching words (i.e., (c) emits a key-value pair for 95% of the words in the dataset). Figure 3.7c shows the dynamically computed final cost of solution (c) using p_1 and p_2 estimates calculated using sampling. Figure 3.7b shows the actual performance of the two solutions. For datasets with very high skew, it is beneficial to use solution (b) due to the smaller size of its key-value pair

```
1 key1_found = false
2 key2_found = false
3 for word in text:
4   if word == key1:
5     key1_found = true;
6   if word == key2:
7     key2_found = true;
```

(a) Sequential code for StringMatch



(b) Performance of solutions over datasets with different levels of skew

Input Dataset	Cost of Solutionn (c)	Optimal Solution
0% match	0	(c)
50% match	$75N$	(c)
95% match	$142.5N$	(b)

(c) Dynamic selection of optimal algorithm

	Solution	Static Cost
a	$output = reduceByKey(map(text, \lambda_m), \lambda_r)$ $\lambda_m : (word) \rightarrow \{(key1, word = key1), (key2, word = key2)\}$ $\lambda_r : (v_1, v_2) \rightarrow v_1 \vee v_2$	$\lambda_m : 2 * (40 + 10) * N$ $\lambda_r : 2 * 2 * 50 * N$ Total : $300N$
b	$output = reduce(map(text, \lambda_m), \lambda_r)$ $\lambda_m : (word) \rightarrow \{(word = key1,\ word = key2)\}$ $\lambda_r : (t_1, t_2) \rightarrow (t_1[0] \vee t_2[0],\ t_1[1] \vee t_2[1])$	$\lambda_m : 1 * 28 * N$ $\lambda_r : 2 * 28 * N$ Total : $84N$
c	$output = reduceByKey(map(text, \lambda_m), \lambda_r)$ $\lambda_m : (w) \rightarrow \{if\ (w = key1) : (key1, true), if\ (w = key2) : (key2, true)\}$ $\lambda_r : (v_1, v_2) \rightarrow v_1 \vee v_2$	$\lambda_m : (p_1 + p_2) * 50 * N$ $\lambda_r : (p_1 + p_2) * 2 * 50 * N$ Total : $150(p_1 + p_2)$

(d) Candidate solutions and their statically computed costs

Figure 3.7: StringMatch benchmark: CASPER dynamically selects the optimal implementation for execution at runtime.

emit. Otherwise, solution (c) performs better. CASPER, with the help of the dynamic input from the runtime monitor, makes this inference and selects the correct solution for all three datasets.

Dynamic cost estimation is particularly impactful in workloads with multiple join operations. The size of each relation participating in the join in addition to the selectivity of the join predicate dictate the most cost-efficient join ordering. To demonstrate this, we translated a simple query based on the TPC-H schema that implements a 3-way join between the *part*, *supplier*, and *partsupplier* relations. Query parameters are the name of the supplier and the customer_id, and outputs are the customer's name, email address, and the sum of discount savings across all sales between the two parties. We executed this query over two parameter configurations: one where the cardinality of *join(sales, supplier)* was much greater than *join(sales, customer)* and one where it was much smaller. On compilation, CASPER generated two semantically equivalent implementations for the query with different join orderings; which one to use depends on the cardinality of the input data. Upon execution, the CASPER runtime estimated the cost of each join ordering and executed the faster solution for both configurations, showing the effectiveness of our dynamic tuning approach. We discuss the accuracy of the cost-functions we used in Appendix A.5.3

3.7.5 System Extensibility

The translation techniques CASPER uses are not coupled to our IR or the target frameworks used. To demonstrate CASPER's extensibility, we implemented the Fold-IR in prior work [36] in our system. Adding the `fold` construct to our IR required just 5 lines of code. An additional 43 lines of code were required to implement compilation of the fold operator to Dafny for verification of synthesized summaries. Since operations such as `min`, `max`, `set.insert` and `list.append` were already available in our IR, hence no extra work was needed. We did not implement any incremental grammar exploration for Fold-IR and used a constant bound to restrict the maximum size of summary expressions. With this minimal amount of work, we synthesized summaries expressed in Fold-IR for all benchmarks in the

Ariths set. We believe it should be easy to extend CASPER's code generator to output the same code as in the original work.

We also explored using WeldIR [62] to express summaries. Although WeldIR is an excellent abstraction for data-processing workloads, we believe it is not suited for synthesis because it is too low-level. However, since both our IR and Fold-IR are conceptually subsets of WeldIR, summaries expressed using them can be translated to Weld through simple rewrite rules. To demonstrate, we successfully translated the summary for TPC-H Q6 expressed in our IR to Weld and used the Weld compiler to produce vectorized, multi-threaded code.

3.8 Related Work

Implementations of MapReduce MapReduce [35] is a popular programming model that has been implemented by various systems [12, 14, 13]. These systems provide their own high-level DSLs that developers must use to express their computation. In contrast, CASPER works with native Java programs and infers rewrites automatically.

Query Optimizers and IRs Modern frameworks usually ship with sophisticated query optimizers [7, 49, 30, 50, 15] for generating efficient execution plans. However, these tools make users express their queries in the provided APIs. Our objective is orthogonal, i.e., to find the best way to express program semantics using the APIs provided by these tools. We essentially enable these tools to optimize code *not* written in their API. Furthermore, unlike our IR, most IRs meant to capture data-processing workloads [36, 62] are not designed with synthesis in mind. This makes it difficult both to find and verify programs expressed in them.

3.9 Conclusion

This chapter introduced CASPER, a new compiler that identifies and converts sequential Java code fragments into MapReduce frameworks. Rather than defining pattern-matching rules

to search for convertible code fragments, CASPER instead automatically discovers high-level summaries of each input code fragment using program synthesis and retargets the found summary to the framework's API. Our experiments show that CASPER can convert a wide variety of benchmarks from both prior work and real-world applications and can generate code for three different MapReduce frameworks. The generated code performs up to 48.2× faster compared to the original implementation, and is competitive with translations done manually by developers.

Chapter 4

TRANSLATING IMAGE PROCESSING LIBRARIES TO HALIDE

In this chapter, we present DEXTER, a new tool that automatically translates image processing functions from a low-level general-purpose language to a high-level domain-specific language (DSL), allowing them to leverage cross-platform optimizations enabled by DSLs. Rather than building a classical syntax-driven compiler to do this translation, DEXTER leverages recent advances in program synthesis and program verification, along with a new domain-specific synthesis algorithm, to translate C++ image processing code to the Halide DSL, while guaranteeing semantic equivalence. This new synthesis algorithm scales and generalizes to much larger and more complex functions than state-of-the-art, including the ability to handle tiling, conditionals, and multi-stage pipelines in the original low-level code. To demonstrate the effectiveness of our approach, we evaluate DEXTER using real-world image processing functions from Adobe Photoshop, a widely used multi-platform image processing program. Our results show that DEXTER can translate 264 out of 353 functions in our test set, with the original implementations ranging from 20 to 150 lines of code. By leveraging Halide's advanced auto-scheduling capabilities, we get median speedups of 7.03× and 4.52× for DEXTER-translated functions as compared to the original implementations on Intel and ARM architectures, respectively.

4.1 Introduction

Domain specific languages (DSLs) for image processing [3, 39, 73] enable high performance, portability, and maintainability, but extending these benefits to existing low-level code is difficult. Rewriting entire legacy applications in DSLs requires huge amounts of human

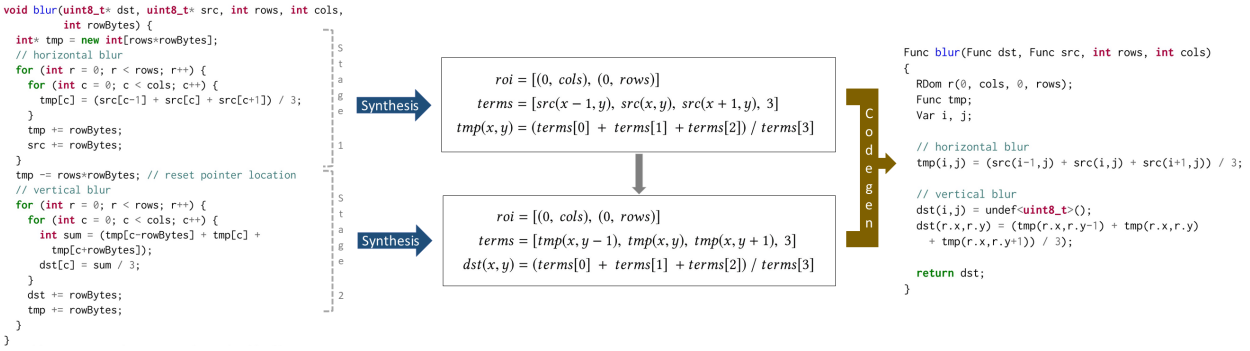


Figure 4.1: DEXTER parses the input C++ function (shown on the left) into a DAG of smaller stages, then uses our 3-step synthesis algorithm to infer the semantics of each stage, expressed in a high-level IR (middle). Finally, code generation rules compile the IR specifications into executable Halide code (right).

effort and risks adding bugs. Building compilers to automatically translate low-level image processing code to high-performance DSLs using traditional code rewriting techniques such as syntax-directed translation [6] is fragile, since these methods are prone to failure when code does not exactly match expected syntactic patterns. These techniques also provide no guarantees that the translated code is correct.

In Chapter 3, we proposed a different way to solve the translation problem. Rather than constructing syntax matching rules, we consider the translation problem as a *search*: given the input code, we build and solve a search problem that helps us find a provably semantically equivalent program in the target language, finding solutions with the aid of recent advances in program synthesis [40, 18] and automatic formal verification techniques. In this chapter, we extend this approach to the domain of image processing. We take image processing operations and pipelines written in C++ and translate them to Halide [70], a high-performance DSL for image computation.

Translating image processing code presents unique challenges. First, although Halide is not a Turing-complete general purpose language, the number of programs expressible is nevertheless huge. In addition, image processing works with arrays, but the lack of first-class

multidimensional arrays in languages like C++ leads to low-level code including pointer arithmetic and using single-dimensional arrays to represent multi-dimensional data. Prior work in other domains has also not considered operations common in low-level image processing code like bit-shifting, widening and narrowing casts, and exact integer arithmetic over multiple data types, each of which present challenges for program synthesis and verification. Combined with various low-level optimizations common in C++ image processing code (e.g., loop tiling or vectorization), these characteristics make reasoning about, and hence translating, image processing code difficult.

We rely on two insights to make the problem tractable. First, rather than searching directly over the textual representation of Halide programs, we search over an intermediate representation (IR) that closely resembles Halide, but without details such as type annotations. Second, to make the search scalable, we decompose the search problem into three parts: given an image processing operation written in C++, we first reason about the number of arrays modified by the computation, along with their dimensionality. We analyze how the arrays are traversed to determine the region of interest (ROI) of the operation. Next, we identify the inputs to the operation: the input array reads and scalars used to compute the output. Finally, we use the information inferred from the first two steps to reason about the actual computation performed over the input image to generate the output. Our algorithm infers full specifications for image processing functions and pipelines, which can then be straightforwardly used to generate executable Halide code. We argue that our new search-based algorithm is both simpler and more general than designing ad hoc syntax matching rules in a traditional compiler.

To evaluate the effectiveness of our approach, we have implemented our translation algorithm in Dexter, a translator for rewriting C++ image processing functions in Halide. Dexter performs translation by first synthesizing a summary program written in our IR that decomposes the image processing operation into the three components described above. Then, it uses the synthesized summary to generate executable Halide code. Finally, Dexter leverages the Halide auto-scheduler [1] to generate efficient schedules for the translated func-

tions. We show that our DEXTER prototype can translate 264 image processing functions, developed over decades, from Adobe Photoshop source code performing various blend and filter operations. In addition, we show that DEXTER can also translate complex, difficult-to-understand optimized implementations containing vector intrinsics and loop tiling optimizations, along with multi-stage image processing pipelines.

Overall, this chapter presents the following contributions:

- We describe a three-stage search algorithm for specification inference of image processing functions. Our algorithm is much more scalable than existing techniques, both in terms of the ability to infer specifications from complex code and the time needed to infer them. Furthermore, the algorithm expands the types of operations supported (e.g., bit-wise operations, which are not supported in existing synthesizers).

- We describe how DEXTER translates larger image-processing functions, such as those implementing multi-stage pipelines, by parsing the functions into a directed-acyclic graph (DAG), where each node in the DAG corresponds to a loop-nest implementing an individual operation in the pipeline.

- We implement a prototype called DEXTER[1] based on our algorithm, and show that it can automatically translate 264 functions from a set of 353 functions from the source code of Adobe Photoshop. These functions, implemented using over 36k lines of C++ code, include complexities such as vectorization, loop-tiling, type-casting, bitwise operations, reductions and conditionals, all of which were beyond the scope of prior work [47]. By leveraging Halide and its auto-scheduler, our translated functions are not only more portable, but perform up to 73× faster than the original implementations.

In the rest of the chapter, we first discuss prior work and background in §4.2. Then in §4.3, we give an overview of DEXTER using an example, and discuss our three-stage algorithm

[1] http://dexter.uwplse.org

in §4.4. We describe the implementation of DEXTER in §4.5, followed by experimental results in §4.6.

4.2 Related Work & Background

4.2.1 Automatically Translating Image Processing Code

We discuss related techniques that optimize either image processing or stencil code via automatic rewriting or compilation in three categories: dynamic analyses that use runtime techniques for optimization, hybrid analyses that use both compilation and runtime mechanisms to optimize input code, and classical compilation (i.e., static analysis).

Dynamic Analysis

Dynamic analysis based techniques perform runtime profiling of existing code to derive equivalent translations. Helium [59], an example of such a tool, identifies and converts image processing kernels from stripped binaries to Halide. Helium uses dynamically generated program traces to learn the shapes and values of the input and output buffers, generalizing the computation into a symbolic expression tree that is then used to generate Halide code. Such runtime techniques are fast and can be used even if the code is available only in binary form. However, the reconstructed kernels are merely an approximation of the original code based on the observed set of traces and such techniques do not offer any soundness guarantees. Furthermore, if the traces only exercise a specific set of parameters (for example, a single blur radius for a filter that supports user-definable blur radii), the translated function will only support the specific observed parameters, limiting the tool's usefulness.

Static-Dynamic Hybrid Analysis

To compensate for the lack of soundness guarantees in dynamic techniques, hybrid analysis uses static compilation techniques in addition to runtime profiling. For instance, STNG [47] is a compiler for translating FORTRAN stencils to Halide. It statically analyzes the input

code to ensure that the generated Halide implementations are semantically equivalent to the original over all possible inputs.

Like DEXTER, STNG uses program synthesis to find a valid translation given the input. In the absence of a scalable synthesis algorithm, such as the one described in this chapter, STNG restricts its search to a space of candidate Halide programs defined by a template. STNG constructs these templates through dynamic analysis of program traces, similar to Helium. Therefore, STNG's approach, although sound, suffers from many of the same limitations as Helium. The generated templates are often over-fitted to the set of traces observed and can exclude valid translations from the search space. While any translation found by STNG is guaranteed to be correct, STNG is limited to translating only simple functions as each runtime trace can capture only a single path through the complex control flow in a program, and reconstructing the original control flow through a small set of traces remains challenging. In addition, STNG cannot handle many important operations found in image processing, including casting and bitwise arithmetic.

Similar hybrid approaches are used by systems outside of image processing that synthesize programs based on input-output examples, such as Scythe [89], which synthesizes database queries based on user-provided input-output pairs, and FlashFill [41], a feature in Microsoft Excel that uses examples to guess user-intended transformations.

Syntax-Driven Compilation

Classical source-to-source transformations have been utilized to generate optimized code from higher-level descriptions, based on syntax-driven transformations, which enable fast performance when input code matches expected syntactic forms. As discussed in 2.3, such compilers are based on syntax matching rules to translate input programs, and developing such rules requires major engineering effort. Such approaches have been used to find optimal schedules for image processing operations [60, 71, 19], along with compiling image processing operations to hardware [43, 44].

4.2.2 Program Synthesis and Verification

In DEXTER, we use program synthesis to infer a *summary* for each image processing operation in the input library. Program synthesizers take in two inputs: a search space of candidate program summaries written in our IR, and a way to verify if a candidate is semantically equivalent to the input code. The former is described using a grammar over our IR, to be discussed in §4.4. For the latter, we leverage Hoare-style verification conditions (Section 2.1) that are readily expressible in forms understood by solvers such as Z3 [34].

DEXTER relies on the Sketch [81] program synthesizer to generate and search through the candidate program summaries, in conjunction with a solver for validation. Internally, Sketch solves the search problem using the CEGIS algorithm (Algorithm 1) introduced in 2.2.2.

Unfortunately, standard algorithms for synthesis (like CEGIS) fail to solve our translation problem, as the number of candidate programs is simply too large to be considered by an existing synthesizer. To make the search efficient, specialized algorithms have been developed for different application domains, and we discuss how we address the issue for image processing operations in §4.4.

4.3 Overview

We now describe how DEXTER translates image processing functions in C++ to Halide, using an example to illustrate the workflow.

4.3.1 Image Processing Functions

DEXTER targets image processing functions written in standard C++. Such functions are often expressed using a sequence of loop nests that iterate over the input buffers to compute intermediate or output buffers. Each loop nest iterates through a region of interest (ROI) and, for each point i within the ROI, computes the corresponding value in the output buffer using a neighborhood of values around i and invoking different kinds of operators, such as

arithmetic, bitwise, and conditional expressions (i.e., Halide's `select` operator); array reads using i; and reductions. The input image can be stored using arrays, vectors, or even user-defined types (UDTs). We outline the set of C++ features supported by our implementation in §4.5.1. DEXTER only targets code that implements image processing logic, and does not translate setup or logging code present in image processing applications (e.g., memory allocation, I/O, etc), as such code does not yield performance improvement even if expressed in Halide.

4.3.2 Translating Image Processing Functions to Halide

The input to DEXTER is a library of image processing functions implemented in C++. As output, DEXTER generates a new, semantically equivalent version of the input library implemented using Halide. DEXTER translates the input by parsing each function as a directed acyclic graph (DAG), with each node in the DAG corresponding to a loop nest in the input code, and synthesizing a semantically equivalent Halide function for each node in the DAG. Each translated function then becomes a stage in the overall Halide pipeline.

To demonstrate this process, we show how DEXTER translates a 3×3 box blur to Halide. As shown in Figure 4.2a, the original implementation uses the composition of a 1×3 and a 3×1 blur filter, each implemented as a pair of nested loops, to compute the overall 3×3 blur. The first loop nest iterates the source image `src` and saves the intermediate output in `tmp`. Each iteration of the outer `for` loop (Lines Line 4–Line 10) uses the inner `for` loop (Lines Line 5–Line 7) to compute the `r`'th row of the `tmp` buffer, then moves the input and output buffer pointers to the first column of the next row (Lines Line 8–Line 9) using pointer arithmetic. Similarly, the second loop nest iterates over the intermediate buffer and performs a 3×1 vertical blur to compute the final output stored in `dst`. Unfortunately, generating a Halide implementation using syntax-driven rules directly from this C++ implementation is challenging given the myriad of ways the same computation can be expressed in low-level languages like C++.

DEXTER's goal is to rewrite the code into Halide by inferring a summary expressed using

```
1  void blur(uint8_t* dst, uint8_t* src, int rows, int cols, int rowBytes) {
2    int* tmp = new int[rows*rowBytes];
3    // horizontal blur
4    for (int r = 0; r < rows; r++) {
5      for (int c = 0; c < cols; c++) {
6        tmp[c] = (src[c-1] + src[c] + src[c+1]) / 3;
7      }
8      tmp += rowBytes;
9      src += rowBytes;
10   }
11   tmp -= rows*rowBytes; // reset pointer location
12   // vertical blur
13   for (int r = 0; r < rows; r++) {
14     for (int c = 0; c < cols; c++) {
15       int sum = (tmp[c-rowBytes] + tmp[c] +
16         tmp[c+rowBytes]);
17       dst[c] = sum / 3;
18     }
19     dst += rowBytes;
20     tmp += rowBytes;
21   }
22 }
```

(a) Input: C++ Implementation of a 3×3 box blur filter.

$$roi = [(0,\ cols),\ (0,\ rows)]$$
$$terms = [src(x-1, y),\ src(x, y),\ src(x+1, y),\ 3]$$
$$tmp(x, y) = (terms[0]\ +\ terms[1]\ + terms[2])\ /\ terms[3]$$

$$roi = [(0,\ cols),\ (0,\ rows)]$$
$$terms = [tmp(x, y-1),\ tmp(x, y),\ tmp(x, y+1),\ 3]$$
$$dst(x, y) = (terms[0]\ +\ terms[1]\ + terms[2])\ /\ terms[3]$$

(b) Summary expressed using Dexter's IR that describes the blur function.

```
1  Func blur(Func dst, Func src, int rows, int cols) {
2    RDom r(0, cols, 0, rows);
3    Func tmp; Var i, j;
4    tmp(i,j) = (src(i-1, j) + src(i, j) + src(i+1, j)) / 3;
5    dst(i,j) = undef<uint8_t>();
6    dst(r.x,r.y) = (tmp(r.x, r.y - 1) + tmp(r.x, r.y) + tmp(r.x, r.y + 1)) / 3);
7    return dst;
8  }
```

(c) Output: Halide implementation of the blur function, as generated from the IR summary shown in (b).

Figure 4.2: Using DEXTER to translate a 3×3 box blur filter from C++ to Halide, with casting removed for clarity.

an intermediate representation (IR) based on Halide, as shown in Figure 4.2b. The summary describes the blur function as a DAG of two operations, where each operation in the DAG is described using three components: its ROI, which describes the range and dimensionality over which the operation is realized, its *terms*, which is the set of all array reads, constants, and scalar variables used in the computation, and finally how each point in the output image is computed using the set of available terms.

Generating Halide code from the IR summary is now straightforward, as the summary eliminates all scheduling information as well as any low-level optimizations (like vectorization) that may exist in the input code, leaving only a declarative description of each output point. We show the generated Halide code for the blur function in Figure 4.2c. Once expressed in Halide, the pipeline can then be optimized by the auto-scheduler, for instance, by merging the two stages when iterating over each location in the source image.

Instead of relying on pattern-matching translation rules, which are brittle and difficult to construct manually, the key insight in DEXTER is to use a search-based technique known as *program synthesis* [40, 18] to find the equivalent IR summary, as described in §4.2.2. Unfortunately, searching through the space of all possible summaries is prohibitively expensive,

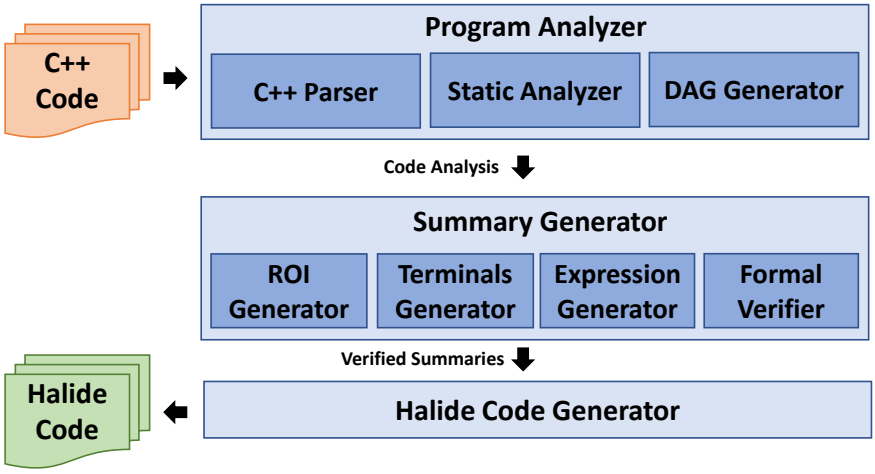


Figure 4.3: DEXTER's system architecture. Image processing functions in C++ (orange) are translated into Halide (green).

and has not been addressed by state-of-the-art program synthesizers. Hence, DEXTER uses a new algorithm (discussed in §4.4) to make the search efficient.

4.3.3 System Architecture

Figure 4.3 shows DEXTER's overall architecture comprising three modules that make up its compilation pipeline.

First, the *program analyzer* parses input C++ code into an Abstract Syntax Tree (AST) and statically analyzes each library function to identify important features about the code, such as the set of variables that are read (input) or modified (output). Then, it parses each function's AST into a DAG of smaller operations before sending it along with the analysis results to the next module.

Next, the *summary generator* synthesizes summaries expressed using DEXTER's IR for each operation in the DAG. First, the ROI generator synthesizes the dimensionality and ROI for each operation. Then, the *terminals generator* synthesizes a mapping for all terminals (variables and array reads) found in the input code into a normalized iteration space. Finally, the *expression generator* synthesizes an expression that encodes the computation performed

in each operation using the previously synthesized terminals. To ensure semantic equivalence to the input, any summary candidate identified by the summary generator is passed to the verifier for validation. We discuss the search and verification procedure in detail in §4.4.

Once a summary is inferred and verified, the *code generator* translates the summary from Dexter's IR into executable Halide code. The translation is straightforward given the resemblance between the IR and Halide. The code generator traverses the summary and generates equivalent Halide code for each expression, outputting a compilable Halide generator that can be combined with an optimized schedule to produce high performance code. We provide more details of the code generation process in §4.5.4.

4.4 Finding Summaries For Image Processing Operations

Dexter uses program synthesis to find translations of image processing operations. To search for translations, we define the search space of Halide programs using a *grammar*, an excerpt of which is shown in Figure 4.4. Given this grammar, the synthesizer will conceptually enumerate all programs that can be constructed using it by randomly choosing a production rule up to a fixed number of times, and check if any of the constructed Halide programs is semantically equivalent to the input.

Unfortunately, this process is prohibitively expensive: even if we limit the synthesizer to expand only up to 5 production rules, the grammar shown in Figure 4.4 expands to tens of thousands different Halide programs; such a search space is at least an order of magnitude larger than what any state-of-the-art synthesizer can handle, making this approach infeasible without further optimizations.

Our key insight to make the search problem tractable in Dexter is to exploit domain-specific knowledge about image processing operations. In particular, we observe that we can decompose many such operations into three components:

- A *Region of Interest* (ROI) that describes the dimensionality of the operation and the bounds for each dimension within which the output is realized.

$$
\begin{aligned}
ROI &:= [B_0,\ B_1,\ \ldots,\ B_D] \\
B_i &:= (IntExpr,\ IntExpr) \\
IntExpr &:= intvars \mid const \mid IntExpr\ Op\ IntExpr \\
Op &:= + \mid - \mid \times
\end{aligned}
$$

(a) Grammar for synthesizing an operation's region of interest.

$$
\begin{aligned}
Term &:= intvar \mid floatvar \mid boolvar \mid const \\
&\mid\ arrvar(Index, \ldots) \\
Index &:= intvar \mid intvar \pm const \mid const \\
&\mid\ arrvar(Index, \ldots)
\end{aligned}
$$

(b) Grammar for synthesizing terminal mappings.

$$
\begin{aligned}
Expr &:= terms \mid iden \mid Expr\ BOp\ Expr \mid UOp\ Expr \\
&\mid\ (Expr\ ?\ Expr\ :\ Expr) \mid f(Expr, \ldots) \\
&\mid\ cast¡Type¿(Expr) \\
Type &:= float \mid uint8\_t \mid int8\_t \mid uint16\_t \mid \ldots \\
BOps &:= + \mid - \mid * \mid / \mid << \mid \ \& \ \mid != \mid \ldots \\
UOps &:= \sim \mid - \mid !
\end{aligned}
$$

(c) Grammar for synthesizing the computation performed in an operation.

Figure 4.4: Search grammars used to synthesize summaries for image processing operations. Each summary represents a possible Halide tranlsation for the input operation.

- The set of *terminals* used to compute the value of each point within the ROI. Such terminals can consist of numeric constants, program variables, or array reads.

- The *computation* performed using the aforementioned set of terminals to compute the values inside the ROI.

DEXTER exploits this insight to decompose the overall summary synthesis problem into three separate synthesis sub-problems, each targeting one component above.

4.4.1 Synthesizing the Region of Interest

An image processing operation's region of interest (ROI) describes its dimensionality and bounds. In this section, we describe how DEXTER synthesizes the ROI for each image processing operation.

ROI Grammar

Like other synthesis problems, DEXTER synthesizes the ROI for each image processing operation by encoding the search space of candidate ROIs using the grammar shown in Figure 4.4a. In the grammar, each ROI consists of D *bound expressions* $(B_i \dots B_D)$, one for each dimension. Each bound expression consists of an upper and lower bound $IntExpr$ that is made up of integer constants ($const$), the set of integer variables and pointers read or updated ($intvars$) extracted through static analysis of each input function, and combinations of such expressions using arithmetic operators.

For example, Figure 4.2b shows the ROI for each operation in the box-blur function, shown in Figure 4.2a. It describes the 1x3 row-blur as two-dimensional, with bounds $0 \leq d_1 < cols$ and $0 \leq d_2 < rows$ for the first and second dimension, respectively. The synthesizer can construct this ROI from the grammar by first setting D to be 2, and then applying the appropriate production rules shown in Figure 4.4a to construct the bound expressions for each dimension.

```
RDom r(0, cols, 0, rows);
Func Var i, j;
dst(i, j) = undef<int>(); // ROI's contents undefined
dst(r.x, r.y) = ⊥; // except for locations within r
```

(a) A candidate ROI expressed in Halide.

```
for (int r = 0; r < rows; r++) {
  for (int c = 0; c < cols; c++) {
    dst[c] = ⊥;
  }
  dst += rowBytes;
  tmp += rowBytes;
}
```

(b) A reduced version of the 3x1 column-blur used to synthesize the ROI.

Figure 4.5: DEXTER synthesizes the ROI of an image processing operation by constructing a reduced version of the input code fragment.

ROI Verification

To synthesize the ROI, we need a way to check whether a candidate ROI is correct. Recall that we have not yet synthesized the set of terminals used or the actual computation performed by the input code. Hence, to verify a candidate ROI, we create a "reduced" version of the input code fragment, where all statements in the fragment's body are removed, except for those (if any) that update loop counters, array pointers, or array contents. We replace all array updates with the special value $\perp$ to indicate that the array entry has been updated, but using an expression that we do not yet know (to be synthesized in the last step).

Consider again the 3x3 blur function from Figure 4.2a. To synthesize the ROI for the second operation in the pipeline (3x1 column-blur), DEXTER prepares a reduced version of the loop nest, shown in Figure 4.5b. Given this code, DEXTER generates ROI candidates using the grammar shown in Figure 4.4a. To check the validity of a candidate, DEXTER creates a

$$roi = [(0,\ cols),\ (0,\ rows)]$$

$$dst(x, y) = \bot(??)$$

(a) The goal is to synthesize the arguments required to compute $\bot$.

```
1  RDom r(0, cols, 0, rows);
2  Var i, j;
3  dst(i,j) = undef<int>();
4  dst(r.x,r.y) = ⊥(src(x,y-1), src(x,y), src(x,y+1), 3);
```

(b) A candidate mapping expressed in Halide.

```
1  for (int r = 0; r < rows; r++) {
2    for (int c = 0; c < cols; c++) {
3      dst[c] = ⊥(src[c-rowBytes], src[c], src[c+rowBytes], 3);
4    }
5    dst += rowBytes;
6    src += rowBytes;
7  }
```

(c) A reduced version of the 3x1 column-blur used to synthesize terminal mappings.

Figure 4.6: Once the ROI has been determined, DEXTER synthesizes a mapping for terminals found in the input code.

skeletal Halide program; one that corresponds to the ROI candidate $[(0, cols), (0, rows)]$ is shown in Figure 4.5a, where the special value $\bot$ is written to the ROI defined by the reduction domain (`r`) on Line Line 4. The validity of a candidate ROI is determined by checking the equivalence of the candidate (Figure 4.5a) and the reduced input code (Figure 4.5b) through program verifiers.

4.4.2 Synthesizing the Terminal Mappings

Once an operation's ROI is synthesized, DEXTER next infers the computation performed by the code fragment for each location within the ROI. As discussed earlier, DEXTER partitions this problem into two further steps. First, DEXTER learns the terminals used in the computation, such as variables, constants, and array reads. Recall from §4.4.1 that we replaced the values of all array updates with the special value $\perp$. Conceptually, the goal of this step is to learn the arguments that are needed to compute $\perp$, as shown in Figure 4.6a.

Extracting Terminals

To extract the set of terminals, DEXTER statically analyzes the input code for each operation in the function. The analysis starts at each statement that updates the output image (such as Line Line 17 in Figure 4.2a), and extracts all terminals involved in these assignments, i.e., `sum` and `3`. Then, it traverses the code backwards to recursively extract all terminals used to compute the extracted values. Since `sum` is in the extracted set, the terminals `src[c-rowBytes]`, `src[c]` and `src[c+rowBytes]` replace `sum` in the extracted set after Line Line 15 in Figure 4.2a is analyzed. The final set of terminals extracted for the 3x1 column-blur operation is {`src[c-rowBytes]`, `src[c]`, `src[c+rowBytes]`, `3`}. These serve as the inputs to $\perp$ as shown in Figure 4.6c.

Mapping Terminals

The terminals extracted through static analysis are defined in the context of loops found in the original code, which can contain low-level optimizations (such as tiling and array flattening) that use different indexing than the Halide code to be synthesized. For example, the input code shown in Figure 4.2a stores the input image `src` as a 1-D array, while the translated Halide code stores the input as a 2-D array as determined by the ROI. Hence, the terminal `src[c-rowBytes]` in the input code can be mapped to $src(x, y-1)$ in the Halide summary, where x and y are the loop induction variables bound to the two dimensions of the ROI. Determining the mappings for constants is trivial: all constants map to themselves. For all other terminals, DEXTER synthesizes their mappings through program synthesis.

Grammar for Terminal Mapping

Figure 4.4b describes the grammar used to synthesize terminal mappings. A terminal ($Term$) can map to scalar values or array reads. While generating array indexing expressions ($Index$), the grammar allows offsetting integer variables, such as the induction variables, by a constant. This allows the synthesizer to explore reads from neighboring indices. Finally, the grammar can also express indirect array accesses to handle code that use pre-computed lookup tables, for instance, the terminal `histogram[src[i]]` that looks up from a pre-computed histogram based on the current location's pixel value.

Verifying Mappings

DEXTER again constructs a reduced version of the input to check the correctness of any synthesized mapping. Like ROI synthesis, DEXTER removes all statements in the input code fragment, except for loops and assignments to output arrays. Rather than changing array assignments to $\perp$, DEXTER instead changes array assignments to a special $\perp$ function with parameters being the extracted terminals. DEXTER then generates a similar skeletal Halide program, an example of which is shown in Figure 4.6b, with the special assignment shown on Line Line 4. Verification, like before, is done by checking the equivalence of the two programs.

4.4.3 Synthesizing the Computation

The final step in synthesizing summaries is to infer how the terminals combine to compute the values used to update the locations within the ROI. To do so, DEXTER replaces the special value $\perp$ (as shown in Figure 4.6a) with an actual Halide expression.

Expressions Grammar

Figure 4.4c shows the grammar to used to synthesize Halide expressions. Besides simple expressions such as arithmetic expressions, DEXTER also supports conditionals in the form

of ternary operations, as well as type-casting to different integer bit-widths, between integer and floating point representations, along with signed and unsigned representations. The only terminals available in the grammar ($Terms$) are the terminals synthesized in the second stage and the special *iden* terminal which represents no-op. The no-op operator is useful for describing operations such as threshold blends, where the input data values control whether a point in the ROI is modified or not.

Verifying the Summary

Replacing $\perp$ with a candidate Halide expression yields a candidate summary of the input code. Similar to the previous stages, DEXTER verifies a candidate by constructing the corresponding Halide program and then testing the equivalence of the generated candidate program with the *original* code fragment. If verification succeeds, then we have found a valid translation of the input code. If not, the synthesizer attempts to generate another candidate, until it exhausts the search space (i.e., the search space encoded by the grammar is not expressive enough), or it times out.

As explained in §4.2.2, DEXTER uses the Sketch synthesizer to sample the search space for each of the three sub-problems created by our algorithm. In Appendix B, we provide supplementary details on the finer optimizations used by DEXTER to optimize the search process. Once synthesized, the summary is sent to the code generator to produce executable Halide code. We discuss the details in §4.5.4.

4.5 Implementation

We implemented DEXTER's program analyzer using the Clang [54] compiler's `libTooling` library to parse C++ code into an abstract syntax tree (AST). The analyzer traverses the AST to perform static analysis and DAG generation, and sends the results to the summary generator. DEXTER's summary generator, implemented in Java, uses an off-the-shelf synthesizer called Sketch [81], along with Z3 [34] for verification. The code generator for parsing the synthesizer's output and generating the output Halide code is also implemented in Java.

In the remainder of this section, we first outline the subset of C++ that DEXTER supports. Then, we discuss how users can interact with, extend, and fine-tune DEXTER. Last, we provide details about Halide code generation.

4.5.1 Supported C++ Constructs

To translate any input code fragment, DEXTER must parse the input and generate search grammars for the different components (as described in §4.4). DEXTER currently supports a core set of C++ constructs, such as basic assignment and declaration statements, conditionals, loops, functions, and user-defined types.

Types Supported

DEXTER supports all built-in primitive C++ data types and operators. It also processes reads and writes into primitive arrays or `std::vector` types. DEXTER only supports pointers that represent dynamically sized primitive arrays, and internally models them as a data array and an integer offset that represents the pointer's location within the array. This enables supporting pointer de-referencing and arithmetic when generating search grammars.

To support user-defined types, DEXTER traverses the program AST to find declarations of all user-defined `structs` used in the code being translated. It then adds these types to the underlying synthesizer's and verifier's type systems. This is useful especially when planar image data is stored in a `struct` with arrays for each channel. DEXTER can also generate search grammars for code that involves user-defined types, including the use of constructors and methods.

Loops

DEXTER can process different types of loops (`for`, `while`, `do`), including those with loop-carried dependencies after applying classical transformations [6] to convert them into `while(true){...}` loops.

Functions

DEXTER handles function calls by inlining the function bodies, except if the function being called is pure and computes a scalar quantity from other scalar quantities. DEXTER translates such functions to equivalent pure functions in our IR and adds them to the search space to keep the generated code clean and understandable. For example, in Figure 4.7, the output `dst` is computed using the function `Mul8x8Div255`, which multiplies the two input 8-bit values, divides by 255, and returns the result. Since `Mul8x8Div255` is a pure function, DEXTER will add it to the expressions grammar in Figure 4.4c. DEXTER currently does not support recursive functions and functions with side-effects other than array writes, since neither are expressible in Halide.[2]

External Library Functions

Users can provide semantic models of external library functions used in the input code by implementing them using Dexter's IR. DEXTER already comes with built-in support for a number of common functions from the C++ standard library (e.g., `min`, `max`, `abs` etc).

4.5.2 DAG Construction

Image processing functions often implement a pipeline of operations computing multiple intermediate and output values. Summaries expressed in our IR that describe the output of an entire multi-stage pipeline are not only difficult to synthesize (due to their potentially large size), but they often do not exist as not all pipelines can be inlined into a single operation. This is the case, for example, of a pipeline where an earlier stage computes a histogram that a later stage uses. To address this issue, DEXTER parses the input functions into a DAG, where each node in the graph represents a single loop-nest found in the code and is treated as an operation in a larger pipeline. This allows us to introduce ordering between different

[2]Functions with side effects can be called from Halide by using extern functions, but such translations are beyond the scope of this work.

fragments of computation within the function, all of which can be expressed using our IR and therefore be translated to Halide to produce a Halide pipeline.

DEXTER generates the DAG through a forward traversal of the function's statements, assigning each statement to a stage in the DAG. At the start, DEXTER initializes the DAG with a single stage that has no statements. It then adds all basic program statements, such as variable assignments and declarations, to the current stage of the DAG until it reaches a loop nest. Once a loop is encountered, DEXTER adds the loop nest to the current stage and creates a new stage as a child of the previous stage. DEXTER then resumes the process until either another loop is encountered or all of the function statements have been assigned. A special case is made for conditional statements (e.g., `if`) that contain a loop in either branch of the control flow path (or both). Each branch of the conditional is recursively parsed into a DAG, with the heads of each sub-DAG connected to the current graph as child nodes (representing a fork in the DAG). The last stages in each of the sub-DAGs are merged back together just as the original program control flow merges.

As an illustration, the blur function in Figure 4.2a is parsed by DEXTER into two consecutive stages, where stage 1 contains all statements from Line Line 2 to Line 10, and stage 2 contains all statements from Line Line 11 to Line Line 21. Dividing the input code this way replaces one difficult synthesis problem (finding a summary that involves 10 terms) to two much simpler synthesis problems (finding summaries involving only 4 terms).

4.5.3 User Interaction

In this section, we discuss the miscellaneous inputs a user may provide to DEXTER and how users may tune or extend the system in the future.

Code Annotations

Occasionally the functions in a library make assumptions about the input parameters that are not explicitly expressed in the source code and yet are essential to its correctness. For

```
1 void adjustOpacity(uint8_t* dst, int opacity, int rows, int cols, int rowBytes) {
2   assert (cols <= rowBytes); // required user annotation
3   for (int r = 0; r < rows; r++) {
4     for (int c = 0; c < cols; c++) {
5       dst[c] = Mul8x8Div255(dst[c], opacity);
6     }
7     dst += rowBytes;
8   }
9 }
```

Figure 4.7: User annotation helps DEXTER determine that each index of the array is updated only once.

instance, the code shown in Figure 4.7 takes as input variables `cols` and `rowBytes` representing the number of columns to compute in each row and the width of the output buffer row in bytes, respectively. The code implicitly assumes that the number of columns is less than or equal to the row-width; otherwise, the assignment on Line Line 5 would be executed multiple times for some locations in `dst`. Because of this possibility, DEXTER will fail to translate the code fragment as there exist inputs where the fragment is not equivalent to a two-dimensional Halide assignment. Users can help DEXTER translate such kernels by adding annotations, such as the `assert` statement on Line Line 2, to clarify the intent of the code.

Tuning Search Grammar

The default grammar used by DEXTER represents a broad class of image processing operations. Users may alternatively want to specialize the grammar to the library they intend to translate. For instance, if the library only includes point-wise operations, the grammar could be adjusted to not explore neighboring points when synthesizing point mappings. Similarly, users may want to compose Halide expressions from a set of custom higher-level library-specific operations. DEXTER is designed to make such modifications easy: it allows users to express grammars by writing them in a format similar to those shown in §4.4. Our default

```
1  for (int row = 0; row < rows; row++) {
2    for (int col = 0; col < cols; col++) {
3      int x = msk[row*cols + col];
4      x = 255 - x + noiseData[HashFunction(row,col)];
5      if (x < 256)
6        msk[row*cols + col] = 255;
7      else
8        msk[row*cols + col] = 0;
9    }
10 }
```

Figure 4.8: Synthesizing the mapping for terminal `noiseData` requires synthesizing the hash function.

grammar is expressed using fewer than 250 lines of code.

Extending DEXTER

Dexter is designed to be highly extensible. For instance, to support custom types or external library functions in the input source code, users only need to provide models for said types and functions using Dexter's IR. To demonstrate the ease of extending DEXTER, we discuss two patches to the system that enable the translation of benchmarks that DEXTER failed to translate during our evaluation: a dissolve blend and an addition blend.

The dissolve blend uses a specialized hash-function over the loop counters to pseudo-randomly read noise data from a pre-computed table, as shown in Figure 4.8. To find the mapping for terminal `noiseData[HashFunction(row,col)]`, DEXTER would have to synthesize this hash function using our points grammar, which is very challenging. A straightforward solution is to implement `HashFunction` in DEXTER using the IR and update the *Index* rule in the default points grammar (Figure 4.4b) as follows:

$$\begin{aligned} Index \;&:=\; intvar \;\mid\; intvar \pm const \;\mid\; const \;\mid\; arrvar(Index, \ldots) \\ &\mid\; arrvar(HashFunction(Index, Index), \ldots) \end{aligned}$$

The addition blend fails since it calls the function `UDIV255`, to perform an unsigned divide-by-255, that is implemented using hand-written assembly, a feature currently not supported by DEXTER. Providing DEXTER with a semantic model of the `UDIV255` function is sufficient to translate this benchmark:

```
uint_t UDIV255(uint_t x) { return x / 255; }
```

4.5.4 Code Generation

Since the synthesizer outputs code using a stylized subset of Halide, code generation is straightforward and is done via a small set of rules. The ROI described by the summary is used to declare the set of induction variables, one for each dimension, as well as constructing the reduction domain (`Halide::RDom`) to iterate over, which defines the set of points over which the stencil executes. The expressions synthesized in the final step describe how each location in the output buffer is computed, and has a one-to-one correspondence with Halide's `Func` assignment statements.

Figure 4.9 lists a part of DEXTER's code generation function `Gen`(), which takes in a DEXTER IR construct and generates executable Halide code. `Gen`() is recursively called: for instance, calling `Gen`() on $e_1 + e_2$ will recursively call `Gen`() to translate each operand. Expressions such as variables and constants represent the base cases, as they trivially map to themselves. Translating the required declarations works similarly; Line 1 in Figure 4.9 converts the synthesized ROI description into an `RDom` declaration.

4.6 Evaluation

In this section, we present a comprehensive evaluation of DEXTER's ability to: (1) translate complex and diverse image processing code, and (2) translate code efficiently. Furthermore, we investigate the performance of the compiled Halide library against the original C++ implementation in various contexts.

All benchmarks in our evaluation were compiled on a high-performance server with 4 Intel

$$
\begin{aligned}
\texttt{Gen}(roi = [(lb_0,\ ub_0),\ \ldots]) &= \texttt{RDom}(\texttt{Gen}(lb_0),\ \texttt{Gen}(ub_0), \ldots) \\
\texttt{Gen}(e_1 = e_2) &= \texttt{Gen}(e_1) = \texttt{Gen}(e_2) \\
\texttt{Gen}(\ (e_1\ ?\ e_2\ :\ e_3)\ ) &= \texttt{select}(\texttt{Gen}(e_1),\ \texttt{Gen}(e_2),\ \texttt{Gen}(e_3)) \\
\texttt{Gen}(\ cast\texttt{<}\tau\texttt{>}(e)\ ) &= \texttt{Halide} :: \texttt{cast} <\tau\texttt{>}(\texttt{Gen}(e)) \\
\texttt{Gen}(e_1 + e_2) &= \texttt{Gen}(e_1)\ \texttt{+}\ \texttt{Gen}(e_2) \\
\texttt{Gen}(var) &= var
\end{aligned}
$$

Figure 4.9: A subset of DEXTER's code generation function Gen().

Xeon E7-4890v2 2.8 GHz 15-core processors, 1 TB of memory, running Ubuntu OS 16.04. For synthesis, DEXTER utilized Sketch 1.7.5 with a parallelism factor of 100. Z3 version 4.8.3 was used for verification. Runtime performance evaluation for compiled code was performed using a 15-inch Apple Macbook Pro (2018) with a 6-core 2.6 GHz Intel Core i7 processor and 16 GB of memory, running macOS 10.14.2; and a 2018 12-inch iPad Pro with a 2.5 GHz Apple A12X processor[3] (ARM64 architecture) running iOS 12.2. The Intel machine supports AVX2 vectorization, and the ARM machine supports NEON vector instructions. We use Git commit cf73bfe6 of Halide for all tests, using the default auto-scheduler weights.

4.6.1 Code for Evaluation

We evaluate DEXTER on 3 suites of image processing functions from Photoshop by Adobe, containing a total of 353 performance critical functions. These functions are called when performing a variety of essential image processing operations, including compositing layers and basic transformations such as rotations and blurs. Due to their importance, some essential functions have been hand-optimized with vectorized x86 implementations; however, due to the difficulty of hand-optimization, only a small subset has been optimized, and these implementations do not take advantage of the latest vectorization capabilities of x86

[3]Multithreaded performance is limited to 2.3 GHz.

processors.

Blend Suite consists of 186 functions across approximately 13k lines of C++ code, which perform point-wise image blending operations such as Normal, Multiply and Dissolve blends. For the most basic operation, i.e. the Normal blend mode [67], two image layers A, B are combined based on a per-pixel weight W, such that the output pixel c_{output} is a linear combination of input pixels c_A, c_B:

$$c_{output} = W \times c_a + (1 - W) \times c_B$$

The set of functions supports a large number of blend modes, but also includes a number of other operations. In addition, the suite contains specialized implementations for specific bit-widths and color formats, as well as specializations where weights are constant.

SSE Blend Suite is a set of hand-optimized blending operations, containing 36 functions implemented in 4.5k lines of code, with a mix of SSE2 intrinsics and hand-written assembly. These are highly non-portable implementations, making it difficult to run Photoshop efficiently on non-x86 hardware.

Filter Suite contains 131 functions implementing various image filtering algorithms that convolve an image with a filter, written in 19k lines of code. These include filters with specific radii, filters for which the radius is an input, as well as specializations for specific image formats.

4.6.2 Feasibility Analysis

DEXTER was able to automatically translate 264 out of 353 functions to Halide, achieving a coverage of 88% for the Blend Suite, 100% for the SSE Blend Suite and 50% for the Filter Suite. The total time required by DEXTER to compile all three suites was 182 hours, an average of 47 minutes per function. The compilation time essentially equals the synthesis time, since synthesis dominates the process. Time spent in all other stages, such as parsing, DAG generation and code generation is insignificant.

```
1  void Darken (uint8_t *dst, uint8_t *src, uint8_t *msk,
2  int rows, int cols, int rowBytes)
3  {
4    for (int rowOut=0; rowOut < rows; rowOut += 16)
5      for (int colOut=0; colOut < cols; colOut += 32)
6        for (row = rowOut; row < min(rows, rowOut+16); row++)
7          for (col = colOut; col < min(cols, colOut+32); col++) {
8            uint16_t delta = (src[row * rowBytes + col]) -
9                             (dst[row * rowBytes + col]);
10           if (delta < 0)
11             (dst[row * rowBytes + col]) -= Mul8x8Div255(
12                                 (msk[row * rowBytes + col]), -delta);
13         }
14 }
```

Figure 4.10: An example of a tiled implementation that DEXTER can successfully de-schedule to recover the program summary.

Of the 89 code fragments that DEXTER failed to translate, 51 failed due to lack of front-end support of language constructs in our current implementation, such as embedded assembly instructions, recursive functions or switch statements. Another 38 benchmarks took too long to synthesize and timed-out after 6 hours of search. See §4.5.3 for examples of such failures, as well as a discussion on how DEXTER can be patched to translate them. The relatively lower coverage of the Filter Suite is due to the increased complexity of the input code. However, this complexity does not stem from the convolutional nature of the functions but instead from how they are implemented. For example, the Filter suite contains recursive implementations and pointer type-casts, i.e., language features that either our current prototype cannot reason about or are unsupported by Halide.

Photoshop executes the code fragments in our test suite on individual tiles of the image that fit into processor caches. As such, the loops inside the operations do not benefit from

tiling optimizations common in image processing. To demonstrate that DEXTER can also translate tiled implementations, we manually modified one image operation implementation to execute in tiles of 16×32, as shown in Figure 4.10. DEXTER successfully translated the loops to recover the correct region of interest (ROI) and the untiled implementation. However, synthesizing the ROI for the tiled implementation took approximately 6× longer since two additional invariants were required due to the additional `for` loops.

Compared to Helium [59], which uses dynamic execution traces to perform translation, DEXTER translates many more operations. We ran Helium[4] on Photoshop and were able to fully-translate 7 operations; part of the difficulty in applying Helium is that the operations must be triggered after starting tracing, which is then manually stopped after the operation is complete. Attempting to translate compositing operations fails under this scenario, because the composite calls many different operations, causing Helium's heuristics to fail; thus, all of the successful translations are from the Filter Suite. Of the 7 translated operations, one (`boxBlur`) cannot be translated by DEXTER due to its recursive nature; in addition, Helium only translates the radius 1 specialization of this function. The other 6 functions are also translated by DEXTER successfully.

4.6.3 Translation Performance

In this section, we discuss two experiments to evaluate the effectiveness of DEXTER's three-stage search algorithm. STNG, which lifts Fortran code to Halide, uses monolithic search combined with traces generated by symbolic execution to limit the search space. Though not a direct comparison between the systems (since DEXTER does not use execution traces, and since the systems target different front-end languages), the experiments in this section compare the monolithic search strategy of STNG against the modular search of DEXTER.

For our first experiment, we compare the performance of DEXTER's modular search against naive monolithic search over a set of 5 library functions using the same synthesizer.

[4]Git commit id 139a4a95

We hand-picked the simplest functions to give naive search the best chance of completion. Each of the five benchmarks were successfully translated in less than 10 seconds by our three-stage algorithm, whereas monolithic search timed out after 15 minutes for all five functions. This demonstrates a speedup of roughly 100× in synthesis time.

Our second experiment aims to investigate how uniformly the synthesis problem is partitioned. To do so, we reviewed the amount of time spent in each of the three stages of synthesis. Across the 264 benchmarks that were successfully translated, DEXTER spent 23% of the translation time during the first stage, 34% of the time during the second stage and the remaining 43% of the time was spent during the third stage, showing that dividing the original synthesis problem into three parts improves overall search efficiency.

4.6.4 Runtime Performance

To demonstrate the possible benefits of applying DEXTER to existing image processing code, we compare the performance of the original reference code to the generated Halide code. We apply the newest Halide autoscheduler [1] to each translated pipeline; this newest autoscheduler uses a combination of learned models and auto-tuning to obtain the best performance. For our x86 testbed, we allow the autoscheduler to explore 32 potential schedules and utilize the best-performing one. Because the current tooling for the autoscheduler does not support exploring GPU schedules or executing on iPads [5], we allow the autoscheduler to use a model augmented by obtained performance on the x86 candidates. We do not argue that these are the best schedules, but they give a sense of what kind of performance can be obtained fully automatically.

In Photoshop, low-level image operations are called on image tiles of a fixed size, instead of on the entire image or layer. The default tile size is 1024 × 1024, and, depending on the operation, tiles may be interleaved or planar. In the actual application, a separate subsystem subdivides tiles among different threads for parallel execution; for these experiments, we run

[5]Apple requires code signing for execution, and the current tooling for the autoscheduler does not implement this requirement.

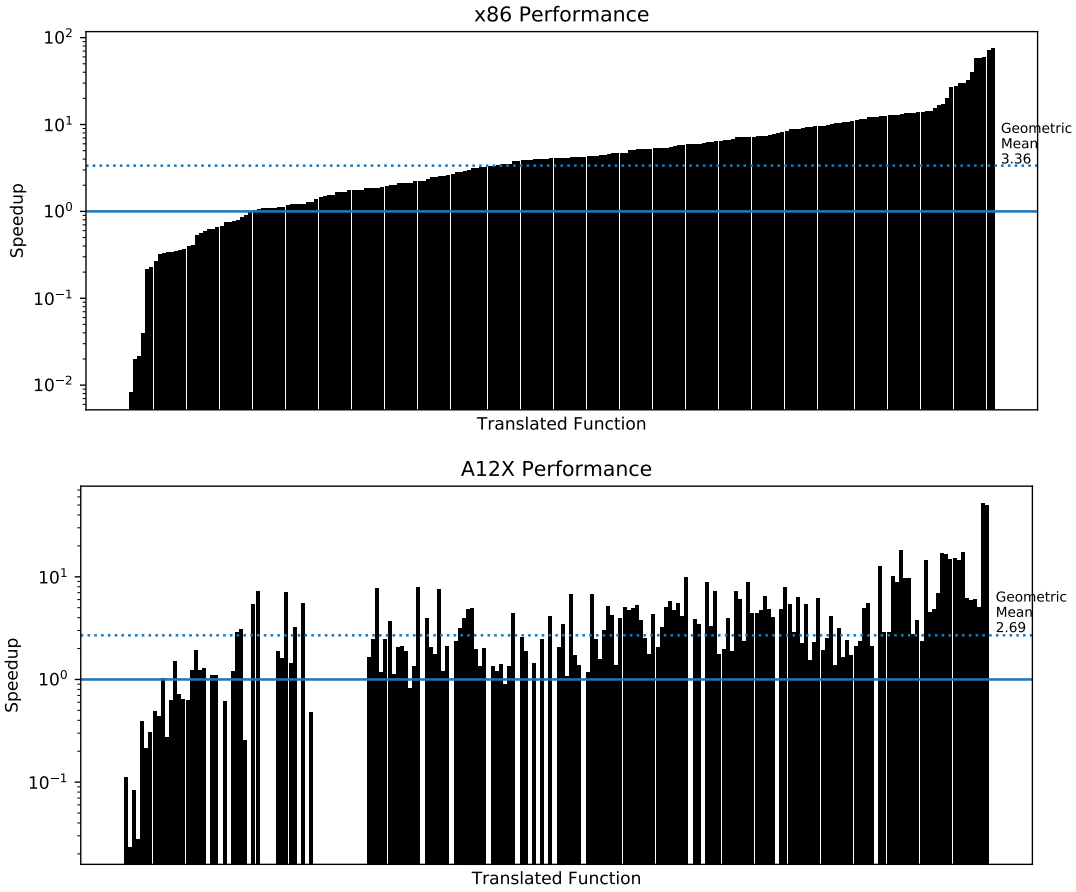


Figure 4.11: Speedup obtained from automatic translation followed by autoscheduling. For x86, we allow the autoscheduler to explore 32 candidates, while for ARM we allow the autoscheduler to augment its model using the x86 performance, but do not explore multiple schedules. Benchmarks are ordered by their x86 speedup; the ARM chart shows fewer benchmarks because we cannot execute hand-written SSE for comparison purposes.

the original code in isolation, without parallelism, since the operations themselves do not contain parallel directives[6]. We measure performance when the tile is present in cache, as this scenario is the most common in Photoshop.

Figure 4.11 shows the performance improvements from applying Dexter to Photoshop image processing source code[7]. The median performance improvement on our x86 test machine is 7.03×, with 70% of all benchmarks achieving a speedup of at least 2×. While compilers are able to vectorize some of these functions automatically, the use of Halide enables a much more efficient vectorized and parallelized schedule with little effort, demonstrating the usefulness of Dexter in bringing the benefits of Halide to legacy image processing code.

Porting to Different Architectures

Photoshop currently only runs on Intel architectures, so for this experiment we demonstrate the usefulness of porting legacy code to Halide automatically to enable cross-platform performance. For our ARM testbed, the median speedup is 4.52×. In some cases, especially for interleaved benchmarks, Figure 4.11 shows that the autoscheduler chooses a suboptimal schedule; this could easily be fixed by changing just a line or two in the generated schedule, unlike the case when hand-optimizing C++ code. To test this, we explored why some of the benchmarks are more than 10× slower, and discovered that the autoscheduler attempted to parallelize over the channel dimension for the interleaved benchmarks; in essence, this schedule forces fine-grained synchronization between cores by sharing cache lines. By writing a simple schedule by hand, we obtain 1.4–4.4× speedups for these benchmarks.

---

[6]Thus, within Photoshop, the performance is often higher than shown here. However, the purpose of this experiment is to show runtime performance of translated kernels, not to compare against Photoshop performance.

[7]Because some translated functions are difficult to test in isolation (e.g. they are leaf operations in a multi-step pipeline), we show performance results for only functions with unit tests that execute them in isolation (88% of the test set).

4.6.5 Composing Translated Functions into Pipelines

A document in Photoshop, like in many image processing programs, consists of *layers* of image data, which are composited together using *blend modes* and filters. The overall document can be thought of as a directed acyclic graph (DAG), with multiple source nodes representing the layer data, intermediate nodes representing blending or filtering operations, and a single sink node representing the final, composited document. In Photoshop, each intermediate node calls multiple low-level routines from the different benchmark suites. In this section, we construct a simple document-specific Just-In-Time (JIT) compiled pipeline that performs the full composite for a small document, and compare its performance with calling the translated (and optimized) routines individually. In both cases, we utilize the autoscheduler for optimization.

The document consists of two layers that are blended together using a normal blend (which calls three different translated functions), and then blurred using a radius of one. Calling individual routines yields a performance of 1.02 Gpixels/sec, while a combined function obtains 3.13 Gpixels/sec, a speedup of over 3×. This speedup is almost completely due to avoiding unnecessary memory traffic, since no intermediates need to be written to memory. We anticipate that JIT-compiling document-specific pipelines for GPUs will yield even greater speedups. This demonstrates that translation to Halide opens up new possibilities for optimization that would be difficult using the legacy C++ code, and DEXTER enables such optimization by automatically translating legacy C++ code into Halide.

4.7 Conclusion

In this chapter, we presented DEXTER, a tool that automatically translates image processing functions written in C++ to the Halide DSL. Unlike traditional compilers, DEXTER uses a novel domain-specific synthesis algorithm to infer the summaries from the input image processing operations. Our prototype can translate many real-world image processing operations, and the translated code performs significantly better when compared to the original

implementation.

Chapter 5

VECTOR INSTRUCTION SELECTION USING VERIFIED LIFTING

Instruction selection, whereby input code expressed in an intermediate representation is translated into executable instructions from the target platform, is often the most target-dependent component in optimizing compilers. Current approaches include pattern matching, which is brittle and tedious to design, or search-based methods, which are limited by scalability of the search algorithm. In this chapter, we propose a new algorithm that first abstracts the target platform instructions into high-level uber-instructions, with each uber-instruction unifying multiple concrete instructions from the target platform. Program synthesis is used to lift input code sequences into semantically equivalent sequences of uber-instructions and then to lower from uber-instructions to machine code. Using 21 real-world benchmarks, we show that our synthesis-based instruction selection algorithm can generate instruction sequences for a hardware target, with the synthesized code performing up to 2.1× faster as compared to code generated by a professionally-developed optimizing compiler for the same platform.

5.1 Introduction

We have witnessed the rise of hardware accelerators across different application domains. These accelerators, such as the Qualcomm Hexagon DSP [29] that is now found on-die in millions of Android mobile phones, offer domain-specific optimization for applications, such as performance improvement and energy savings as compared to general processors. However, to fully utilize such accelerators, applications must either be written against libraries or exotic instruction intrinsics provided by the accelerator, or in a domain-specific language

(DSL) altogether.

While mapping computations from programmer expressions into a sequence of processor instructions can be formulated as *instruction selection* as part of program compilation, choosing the optimal sequence of instructions for a given computation is especially difficult when considering vector instructions, which enable fine-grained parallel computation. Modern vector instruction sets, such as Intel's AVX-512 and VNNI, ARM's Advanced SIMD, or Hexagon's HVX, offer a rich set of complex vector instructions. These include lane-parallel vector instructions such as single-instruction multiple-data (SIMD) instructions, non-isomorphic instructions that apply different operations to different lanes of the input vector in parallel, cross-lane vector instructions that implement reductions such as dot-products, and sliding window instructions that use intersecting sets of input vector lanes to compute the output values in parallel.

Fully exploiting these instructions is difficult for compilers. Most approaches (such as LLVM [52] and Halide [73]) utilize some variant of pattern-matching rewrites, which transform templatized sequences of operations into hardware-specific vector instructions. Indeed, Halide utilizes its own matching machinery, consisting of ad-hoc patterns and rewrites built by programmers experienced in writing code for each specific hardware backend, due to LLVM's inability to fully exploit complex vector instruction sets such as Hexagon's HVX. While work has been done to enhance LLVM's library of patterns [24, 77], matching-based approaches in general suffer from the limitations of the underlying greedy algorithm to match code patterns and thus can miss rewriting opportunities. Meanwhile, prior work that formulates instruction selection as dynamic programming [51] or exhaustively enumerates instruction sequences up to a fixed length guided by a cost model [57, 79] struggle to scale to large input instruction sequences, or require complex cost models to make the search efficient.

In this chapter, we describe Rake, a system that leverages *program synthesis* to perform instruction selection for vectorized Halide expressions. Unlike prior work, Rake does not rely on manually crafted code patterns to match on the input code. Given an input code sequence represented in the Halide IR, Rake instead synthesizes a sequence of target

platform instructions that is provably semantically equivalent to the input. Like Casper and Dexter, Rake also implements verified lifting to enable synthesis-based code translation. It *lifts* the input code sequence into an intermediate representation (IR). This IR, called Uber-Instruction IR, is a high-level abstracted version of the target instruction set. Once lifted, Rake then lowers the Uber-Instruction IR into the concrete syntax of the target instruction set using synthesis. Lowering is done *incrementally* by first synthesizing a combination of hardware intrinsics that performs the computation while ignoring any data movement operations (a *swizzle-free sketch*).

Once the swizzle-free sketch is synthesized, the data movement (i.e., swizzle) instructions are concretized via another synthesis query, and the finished sequence of instructions is grafted back into the Halide IR. Rake makes it possible to integrate backend-specific rewrites into a domain-specific compiler without explicitly specifying transformation rules or their priority order. Instead, compiler developers only need to specify the semantics of the target instructions, modifying the Uber-Instruction IR if necessary, and Rake will then synthesize target instruction sequences automatically. As our experiments show, Rake's synthesis-driven "lift-then-lower" approach makes instruction selection scalable, without compromising the quality of the generated code.

To summarize, our key contributions are:

- A methodology that decomposes instruction selection for mixed vector/scalar IR into a series of tractable program synthesis queries, by searching for high-level Uber-Instructions, then lowering to sketches without data movement operations, and finally synthesizing the required data movement.

- An implementation of our methodology within the Halide DSL compiler and evaluation using real-world benchmarks.[1]

We evaluate Rake by using it to generate code for the HVX accelerator and show that

[1]Source code available at https://github.com/uwplse/rake/

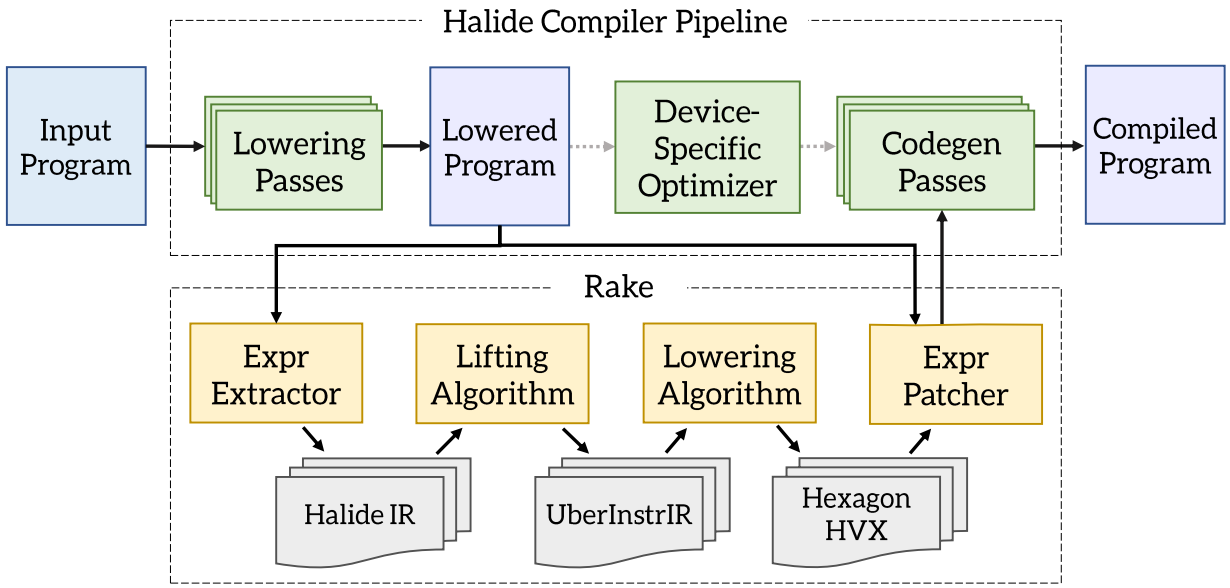


Figure 5.1: Compilation Overview. Rake intercepts Halide's compilation pipeline to synthesize device-specific implementations for vector expressions.

Rake-generated code can produce speedups of up to 2.1× over that generated by the existing Halide and LLVM HVX pipeline (which has been developed and tuned by Qualcomm, Google, and LLVM developers), as Rake finds instruction sequences that are not considered by the existing Halide pattern-matching rules and the instruction selection pass in LLVM.

Next, we give background on the Halide domain-specific compiler and describe how Rake works in three phases: lifting to the Uber-Instruction IR (§5.3), swizzle-free sketch synthesis (§5.4), and swizzle synthesis (§5.5). Then, in §5.7 we show the efficacy of Rake using 21 real-world benchmarks.

5.2 Background & Overview

In this section, we give an overview of the entire compilation process illustrated in Figure 5.1, while providing background on the methodologies underlying Rake's synthesis-based instruction selection algorithm.

```
1  void sobel3x3(Buffer<uint8_t> input, Buffer<uint8_t> output) {
2    Var x, y, xi, yi;
3    Func in16, x_avg, y_avg, sobel_x, sobel_y;
4
5    // The Algorithm
6    in16(x, y) = cast<uint16_t>(input(x, y));
7
8    x_avg(x, y) = in16(x - 1, y) + 2 * in16(x, y) + in16(x + 1, y);
9    sobel_x(x, y) = absd(x_avg(x, y - 1), x_avg(x, y + 1));
10
11   y_avg(x, y) = in16(x, y - 1) + 2 * in16(x, y) + in16(x, y + 1);
12   sobel_y(x, y) = absd(y_avg(x - 1, y), y_avg(x + 1, y));
13
14   output(x, y) = cast<uint8_t>(clamp(sobel_x(x, y) + sobel_y(x, y), 0, 255));
15
16   // The schedule
17   output.hexagon()
18       .prefetch(input, y, 2)
19       .tile(x, y, xi, yi, 128, 4)
20       .vectorize(xi);
21 }
```

Figure 5.2: The Sobel Filter expressed in Halide.

We implement Rake within Halide [73], a domain-specific language for computations on images and dense tensors. Halide enables programmers to concisely describe the algorithm they wish to implement separately from details of how that algorithm should be executed. This separation between *algorithm* and *schedule* allows the compiler to generate optimized code for CPUs, GPUs, and DSPs from the same algorithm specification. Halide enjoys wide adoption in industry, including usage by Google, Adobe, and others [72].

```
// Syntax guide:
// - uint8x128(...) and uint16x128(...) are vector casts
// - x128(c) broadcasts scalar c to a 128-lane vector
// - input(x,y) denotes a 1024-bit vector load from location (x,y)
// - min, max, absd (absolute difference), + and * are vector ops
uint8x128(
  max(
    min(
      absd(
        uint16x128(input(x - 1, y - 1)) +
          uint16x128(input(x, y - 1)) * x128(2) +
          uint16x128(input(x + 1, y - 1)),
        uint16x128(input(x - 1, y + 1)) +
          uint16x128(input(x, y + 1)) * x128(2) +
          uint16x128(input(x + 1,  y + 1))
      )
      +
      absd(
        uint16x128(input(x - 1, y - 1)) +
          uint16x128(input(x - 1, y)) * x128(2) +
          uint16x128(input(x - 1, y + 1)),
        uint16x128(input(x + 1, y - 1)) +
          uint16x128(input(x + 1, y)) * x128(2) +
          uint16x128(input(x + 1, y + 1)),
      ),
    x128(0)),
  x128(255)))
```

Figure 5.3: Target-independent Halide IR vector expression produced by the Sobel filter algorithm.

5.2.1 Motivating Example

The Sobel filter [68] is a well-known algorithm used in image processing and computer vision for approximating the gradient of an image intensity function. This approximation is particularly useful in edge-detection algorithms.

Figure 5.2 shows an implementation of the Sobel filter[2] expressed in Halide. To compile this algorithm for a given architecture, Halide requires a schedule that describes high-level device-specific optimizations, such as how to tile and vectorize the loops in the output program. Lines 17 to 20 specify a Halide schedule for compiling the Sobel filter to the Hexagon DSP architecture [29]. The schedule directs Halide to offload the computation to Hexagon, prefetch data into the cache two iterations before it is needed, and compute the output in tiles of 4×128 elements. The schedule also directs Halide to vectorize the inner x-loop. Since the schedule specifies no directives for any of the intermediate results, such as `sobel_x` or `sobel_y`, by default Halide will inline their computation. Thus, after all scheduling is applied, the lowered program in Halide IR is a single tiled loop-nest, where the body of the inner-most loop computes a 1×128 element tile of the output using a target-independent vector-expression, shown in Figure 5.3.

Once the program is lowered into Halide's IR, the next step is code-generation. Currently, Halide relies on two mechanisms to map Halide's IR to machine instructions: for most operations, Halide uses LLVM's built-in vector types and operations, but in addition, the Halide compiler uses a pattern-matching pass (the Device-Specific Optimizer in Figure 5.1) to rewrite sequences of Halide IR operations into calls to LLVM backend-specific intrinsics. This transformation is essential for obtaining performance on architectures like Hexagon, where LLVM fails to automatically discover mappings from generic LLVM IR to semantically-rich backend vector instructions such as those in the HVX ISA.

While Halide's pattern-matching approach is fast, it is also brittle. If a program fragment could map to an HVX-specific instruction, but doesn't conform to the pre-written patterns,

[2]This implementation obtained from the Halide repository is a modified version that does not take the square root of the gradient.

	Halide IR Expr	Halide Codegen (HVX)	Rake Codegen (HVX)

(a)	`uint16x128(input(x-1,y+1))`
	`+`
	`(`
	` uint16x128(input(x,y+1))`
	` *`
	` x128(2)`
	`)`
	`+`
	`uint16x128(input(x+1,y+1))`
		`/* Latency: 4,`
		`  Loads: 3 */`
		`// 2-multiply-add`
		`vmpa(`
		`  input(x, y+1),`
		`  input(x+1, y+1),`
		`  0x2, 0x1) +`
		`vzxt(`
		`  input(x-1, y+1)`
		`)`
			`/* Latency: 2,`
			`  Loads: 2 */`
			`// Sliding window`
			`// 3-point reduction`
			`vtmpy(`
			`  input(x - 1, y + 1),`
			`  input(x, y + 1),`
			`  0x1, 0x2)`

(b)	`uint16x128(input(x-1, y-1))`
	`+`
	`(`
	` uint16x128(input(x-1, y))`
	` *`
	` x128(2)`
	`)`
	`+`
	`uint16x128(input(x-1, y+1))`
		`/* Latency: 4 */`
		`// 2-multiply-add`
		`vmpa(`
		`  input(x-1, y),`
		`  input(x-1, y+1),`
		`  0x2, 0x1) +`
		`vzxt(`
		`  input(x-1, y-1))`
		`)`
			`/* Latency: 3 */`
			`// 2-multiply-add`
			`// accumulate`
			`vmpa.acc(`
			`  vzxt(`
			`  input(x-1, y-1)`
			`  ),`
			`  input(x-1, y),`
			`  input(x-1, y+1),`
			`  0x2, 0x1)`

(c)	`uint8x128(`
	`  max(`
	`  min(`
	`  absd(...) +`
	`  absd(...),`
	`  x128(0)`
	`  ),`
	`  x128(255)`
	`  )`
	`)`
		`/* Latency: 11 */`
		`// Extract lower byte`
		`vshuffeb(`
		`  vmax(`
		`  vabsdiff(...) +`
		`  vabsdiff(...),`
		`  vsplat(255)),`
		`  vmax(`
		`  vabsdiff(...) +`
		`  vabsdiff(...),`
		`  vsplat(255)))`
			`/* Latency: 9 */`
			`// Saturate`
			`vsat(`
			`  vabsdiff(...) +`
			`  vabsdiff(...),`
			`  vabsdiff(...) +`
			`  vabsdiff(...)`
			`)`

Figure 5.4: An illustration of key differences in the HVX code generated by Halide's current optimizer and Rake for the Sobel filter. We do not count broadcasts of loop-invaraint expressions towards latency, as they will be moved outside the loop by LLVM.

Halide will use lower-performing generic vector instructions. This leaves valuable performance on the table. Figure 5.4 highlights three instances in the Sobel filter benchmark where Halide's pattern-matching approach fails to discover the best instruction sequence. In (a), Halide uses the more general `vmpa` (sum of two widening multiplies) and `vadd` instructions to implement the 3-point horizontal convolution. This computation can be implemented more efficiently as a single `vtmpy` instruction (sliding-window-sum of two widening multiplies with an additional accumulation). In (b), although the `vtmpy` instruction is not applicable as this expression does not implement a sliding-window reduction, we can replace the `vmpa` and `vadd` instructions with a single `vmpa.acc` instruction (a variant of `vmpa` that accumulates into the target register). Finally, in expression (c), Halide fails to infer that the *min* and *cast* operations on an unsigned input can be replaced by a single *saturate* operation. In contrast, Rake can discover all three optimizations without the need for any re-write rules, resulting in a 27% runtime performance improvement over Halide's existing optimizer.

5.2.2 Instruction Selection using Rake

As illustrated in Figure 5.1, Rake intercepts Halide's compilation pipeline after the input program has been lowered to Halide's IR and all scheduling optimizations, including vectorization, have been applied. Rake then extracts the set of vectorized expressions found in the lowered program and uses program synthesis to discover mappings from Halide IR to backend-specific intrinsics using program synthesis. Rake decomposes the instruction selection problem into multi-step program synthesis by first lifting the input expressions to high-level Uber-Instructions before lowering. Rake lowers uber-instruction sequences into executable instructions by synthesizing the computational instructions followed by data movement guided by a simple, explainable cost model. Together, these strategies enable Rake to scale up to large loop bodies, rather than the scale of a few instructions as in prior superoptimizers [79, 65]. Finally, Rake patches the lowered program, replacing target-independent Halide IR vector expressions with optimized target-aware instruction sequences.

```
(narrow
  (vs-mpy-add
    '((abs-diff
       (vs-mpy-add (load-data) '(2 1 1) #f uint16)
       (vs-mpy-add (load-data) '(2 1 1) #f uint16))
      (abs-diff
       (vs-mpy-add (load-data) '(2 1 1) #f uint16)
       (vs-mpy-add (load-data) '(2 1 1) #f uint16)))
    '(1 1) #f uint16)
  #t #f uint8)
```

Figure 5.5: The Sobel filter expression lifted to HVX uber-instructions.

Lifting to Uber-Instruction IR

Rake begins the instruction selection process by lifting the input Halide IR expressions (such as the one shown in Figure 5.3) into a target-specific IR of uber-instructions, which we call the Uber-Instruction IR.

The Uber-Instruction IR is a condensed version of the target ISA, where each uber-instruction unifies a set of related intrinsics in the target ISA by implementing the common higher-level compute pattern. For example, Figure 5.5 shows the expression encountered in the Sobel filter (Figure 5.3) expressed using the uber-instructions derived from the HVX ISA. The `vs-mpy-add` uber-instruction unifies all available HVX intrinsics that implement vector-scalar multiply-add patterns, such as `vadd` (addition), `vmpy` (widening multiply) or `vmpa` (sum of two widening multiples). Similarly, the set of HVX intrinsics that downcast integer values in the input vector to a narrower integer type are consolidated into an uber-instruction called `narrow`. Figure 5.6 shows pseudo-code describing the semantics of HVX uber-instructions that appear in Figure 5.5.

The translation from Halide IR to Uber-Instruction IR is performed using a bottom-up enumerative synthesis algorithm (described in §5.3) that *lifts* the lower-level Halide IR to the higher-level Uber-Instruction IR. The lifting algorithm attempts to rewrite the input

```
(define (narrow vec saturate? round? outT)
  (let a (if round? (round vec) vec))
  (if saturate? (sat_cast<outT> a) (cast<outT> a)))

(define (abs-diff vec0 vec1)
  (- (max vec0 vec1) (min vec0 vec1)))

(define (vs-mpy-add vec weights saturate? outT)
  (let a (if saturate? (cast<int64> vec) (cast<outT> vec)))
  (let b (convolve a weights))
  (if saturate? (sat_cast<outT> b) b))
```

Figure 5.6: A sample of HVX uber-instructions.

expressions using the fewest number of uber-instructions possible. This, in effect, clusters operations in the input expression that implement a single high-level compute pattern by rewriting them into an uber-instruction. Rake then synthesizes target ISA implementations for the lifted expression by lowering the sequence of uber-instructions into a sequence of target ISA instructions. Lifting expressions to Uber-Instruction IR makes synthesis-based instruction selection scalable in two ways. First, it breaks the synthesis problem down to easier sub-problems since larger expressions require many uber-instructions to implement, and the algorithm builds the sequence of uber-instructions bottom-up. Second, for each uber-instruction only a subset of the target ISA is relevant, so we can specialize the grammar to just those instructions.

Lowering to the Target ISA

In the second stage of instruction selection, Rake uses a recursive backtracking algorithm, listed in Algorithm 4, to lower the lifted expressions to the target ISA. Given an expression e in the Uber-Instruction IR, Rake first recursively lowers each sub-expression to the the target ISA (Lines 5 to 7). Then, Rake uses the lowered sub-expressions $\mathcal{S}$ as building-blocks

```
1 (??swizzle
2   (vtmpy (??load [vec-pair? #t]), 1, 2)
3   [vec-pair? #t])
```

Figure 5.7: A swizzle-free sketch of an HVX expression. Data-movement is abstracted away using `??load` and `??swizzle`.

```
1 (let [x (vtmpy
2           (vcombine
3             (vread (- x 1) (- y 1))
4             (vread (+ x 1) (- y 1)))
5           1, 2)]
6     (vshuffvdd (hi x) (lo x) -2))
```

Figure 5.8: Synthesized data movement replaces `??load` and `??swizzle` to yield complete HVX implementations.

to synthesize $\mathcal{I}$, the lowered implementation for e. We explain Rake's lowering algorithm in more detail in §5.4 and §5.5.

Despite the incremental approach outlined above, synthesizing efficient vector implementations remains challenging. The best implementations often involve an interweaving of data-movement (i.e., loading/storing data across memory hierarchies or re-arrangement of vector lanes into different permutations) and computation (operations that produce new values) to exploit intrinsics that offer the greatest throughput. Directly synthesizing such implementations is expensive due to the sheer number of candidates that can be enumerated. Therefore, Rake first synthesizes a swizzle-free sketch τ of the output expression. A swizzle-free sketch is a partial implementation of the input expression in the target ISA that specifies the computation concretely using intrinsics from the target ISA but abstracts away the necessary data movement using special placeholder terms. For example, consider the lifted multiply-add expression from line 4 of Figure 5.5. Figure 5.7 shows a valid swizzle-free sketch for this expression. The computations performed by the sketch (multiplications, addi-

tions and widening-casts) are implemented using intrinsics from the HVX ISA, such as `vtmpy` (3-point fused widening-multiply-add), but the loading and swizzling of data are expressed abstractly using special constructs `??load` and `??swizzle`. In §5.4, we define the semantics of the constructs used by Rake to represent data movement in swizzle-free sketches and explain how Rake uses them to verify partial implementations for correctness during synthesis. At a high level, these allow instructions in a swizzle-free sketch to load data from any memory location or vector lane in a register without needing to reason about the cost or sequence of data movement operations required.

Synthesizing Data Movement

Once a valid swizzle-free sketch is synthesized, Rake attempts to complete the implementation by synthesizing the missing data movement. Figure 5.8 shows a complete implementation synthesized using the sketch from Figure 5.7. Each `??load` and `??swizzle` term has been replaced by a sequence of vector-reads and HVX shuffling instructions (such as `vcombine` and `vshuffvdd`) to yield a fully lowered HVX expression.

5.3 Dynamic Expression Decomposition

Synthesizing low-level device-specific expressions from IR code is an expensive task. For example, the relatively simple IR expression found in the Sobel filter (Figure 5.3) requires a sequence of 38 HVX intrinsics to implement. On top of that, the HVX ISA offers hundreds of intrinsics to choose from at every step when building the sequence. Prior work attempted to scale synthesis to large instruction sets by constraining the length of the input instruction sequence considered at each step [65, 16]. Other prior work attempted to scale the input instruction sequence length by carefully extracting a directed acyclic graph (DAG) of operations to consider with a single output [77] while restricting the scope of the target instructions to middle-end IR rather than concrete instructions. While these strategies are effective, they have mostly been applied to scalar code (or, in the case of [77], directly to LLVM IR); vectorized code introduces not just a larger variety of potential instructions

Input: An expression in the Halide IR

Output: An equivalent expression in the Uber-Instruction IR

```
1 Function Lift(e)
2     S ← { Lift(se) | se ∈ Subexprs(e) }
3     for s ∈ S do
4         if ℓ ← UpdateInstr(s, e) ≠ unsat then return ℓ
5     for s ∈ S do
6         if ℓ ← ReplaceInstr(s, e) ≠ unsat then return ℓ
7     return ExtendExpr(S, e)
```

Algorithm 3: Lifting expressions from Halide IR to the Uber-Instruction IR.

but additional complications due to potentially needing swizzling between operations. The goal of Rake is to scale synthesis to input sequences sufficiently large enough to optimize real-world code, as well as scaling reasoning to handle a large, complex vector instruction set.

5.3.1 Uber-Instruction IR

In modern ISAs, the number of high-level compute patterns implemented is typically much smaller than the number of intrinsics offered; many intrinsics can be viewed as specializations of a more general compute pattern. An uber-instruction is a function that implements one such high-level compute pattern and therefore consolidates the semantics of many related intrinsics in the target ISA. The Uber-Instruction IR is simply a collection of all uber-instructions manually derived from the target ISA. To make synthesis scalable, Rake re-writes large input expressions as a sequence of the derived uber-instructions, before lowering each uber-instruction incrementally. In §5.6, we discuss the key design concerns when designing the Uber-Instruction IR for a given target ISA, and how we derived the set of uber-instructions for the HVX ISA.

5.3.2 Lifting Algorithm

Rake uses a bottom-up enumerative search algorithm, listed as Algorithm 3, to lift expressions from the Halide IR to the Uber-Instruction IR. Given a Halide IR expression e, Rake first recursively lifts each sub-expression of e to the Uber-Instruction IR (Line 2). Then, Rake applies a sequence of *update*, *replace* and *extend* steps to the set of lifted sub-expressions ($\mathcal{S}$) to construct the lifted representation of e.

Update

In the update step, Rake tries to lift the input expression by updating the inputs to an instruction in one of the lifted sub-expressions. Suppose e is the input expression in Halide IR and $\mathcal{S}$ is the set of sub-expressions of e lifted to the Uber-Instruction IR. Then, for each $s \in \mathcal{S}$, We formulate the following query to the SMT solver:

$$\exists\, i \in s. \quad interpret(e) = interpret(s[i \rightarrow i'])$$

We want to check if there exists an uber-instruction i in the sub-expression s, such that updating the parameters to i makes the output of s and the output of e equal. For instance, if i is the `narrow` uber-instruction, we may update the `saturate?` flag to also perform saturation while narrowing.

Replace

The replace step is similar to the update step, except that instead of updating an instruction, Rake attempts to replace an instruction with a different uber-instruction:

$$\exists\, i \in s.\, j \in \text{UberIR}. \quad interpret(e) = interpret(s[i \rightarrow j])$$

We want to check if there exists an uber-instruction i in the sub-expression s, such that when we replace i with another uber-instruction j, the output of s and e are equal.

Extend

Finally, if neither update or replace steps are successful, Rake lifts e by extending the lifted sub-expressions with a new uber-instruction:

$$\exists\, i \in \text{UberIR}.\ s_0, ..., s_n \in \mathcal{S}.\quad interpret(e) = interpret(i(s0, ..., s_n))$$

Demonstrative Example

Figure 5.9 illustrates the lifting process for the Sobel filter expression in Figure 5.3. For brevity, we do not show the numerous synthesis queries that are *unsat* and instead focus on the successful queries to demonstrate how the lifted expression evolves. Steps 1, 2, 3 and 4 show the base case for the recursive algorithm. Since there are no vector sub-expressions for leaf nodes, Rake extends the expression by adding a new uber-instruction. Step 5 shows an application of the *replace* step: by replacing the uber-instruction `widen` with the uber-instruction `vs-mpy-add`, Rake is able to construct an equivalent expression without increasing the total number of uber-instructions. Finally, steps 6 and 7 demonstrate applications of the update rule: additional sum operations can simply be folded into the existing `vs-mpy-add` instruction by updating the weight matrix. The weight matrix for the `vs-mpy-add` instruction specifies both the length of the multiply-add pattern, as well as the scalar weights.

Rake's lifting algorithm greedily folds each new Halide IR operation encountered during the bottom-up traversal into the existing Uber-Instruction IR expression. As a result, it may not always discover the Uber-Instruction IR representations with the fewest number of instructions. However, the greedy approach is scalable as each synthesis query attempts to add or modify at most a single uber-instruction.

5.4 Abstracting Data Movement

When lowering an expression from Uber-Instruction IR to the target ISA, Rake first synthesizes a *swizzle-free sketch* that expresses the computation using target ISA intrinsics, while

Step	Halide Expr	Lifted Sub-Exprs	Rule	Lifted Expr

1	`cast<uint16_t>(`
	`  input(x-1, y-1)`
	`)`
		∅	Extend	`(widen`
				`  (load-data)`
				`  int16)`

2	`cast<uint16_t>(`
	`  input(x, y-1)`
	`)`
		∅	Extend	`(widen`
				`  (load-data)`
				`  int16)`

3	`2`	∅	Extend	`(broadcast 2)`

4	`cast<uint16_t>(`
	`  input(x+1, y-1)`
	`)`
		∅	Extend	`(widen`
				`  (load-data)`
				`  int16)`

5	`cast<uint16_t>(`
	`  input(x, y-1)`
	`)`
	`*`
	`2`
		`[`
		` (widen (load-data) int16),`
		` broadcast(2)`
		`]`
			Replace	`(vs-mpy-add`
				` (load-data)`
				` [kernel: '(2)]`
				` [saturating: #f]`
				` [output-type: int16]`
				`)`

6	`cast<uint16_t>(`
	`  input(x-1, y-1)`
	`) +`
	`cast<uint16_t>(`
	`  input(x, y-1)`
	`) * 2`
		`[`
		` (widen (load-data) int16),`
		` (vs-mpy-add`
		`  (load-data)`
		`  [kernel: '(2)]`
		`  [saturating: #f]`
		`  [output-type: int16])`
		`]`
			Update	`(vs-mpy-add`
				` (load-data)`
				` [kernel: '(2 1)]`
				` [saturating: #f]`
				` [output-type: int16])`

7	`cast<uint16_t>(`
	`  input(x-1, y-1)`
	`) +`
	`cast<uint16_t>(`
	`  input(x, y-1)`
	`) * 2 +`
	`cast<uint16_t>(`
	`  input(x+1, y-1)`
	`)`
		`[`
		` (widen (load-data) int16),`
		` (vs-mpy-add`
		`  (load-data)`
		`  [kernel: '(2 1)]`
		`  [saturating: #f]`
		`  [output-type: int16])`
		`]`
			Update	`(vs-mpy-add`
				` (load-data)`
				` [kernel: '(2 1 1)]`
				` [saturating: #f]`
				` [output-type: int16])`

...	...	...	...	...

Figure 5.9: An illustration of how Rake uses bottom-up program synthesis to lift the Sobel filter to the Uber-Instruction IR.

```
(define (??load live-data elemT vec-pair?)
  (λ (i) (choose* ;; The synthesizer can pick any value from the
                  ;; live-data set after filtering
    (filter (λ (v) (eq? (type v) elemT))  live-data))))

(define (??swizzle exprs elemT vec-pair?)
  (λ (i) (choose* ;; The synthesizer can pick any value from exprs
                  ;; after filtering
    (filter (λ (v) (eq? (type v) elemT)) (get-vals exprs)))))
```

Figure 5.10: Definition of `??load` and `??swizzle`.

abstracting away all data movement. The goal of synthesizing this sketch is to simplify the synthesis problem by identifying sequences of compute intrinsics that produce the correct output while assuming all required data is present in registers in the correct layout required for each instruction. To prove the correctness of a swizzle-free sketch, it is sufficient to show that there exists a sequence of loads and swizzles for which the sketch produces the correct output. Identifying the most optimal set of data movement instructions in the target ISA can be deferred until a correct sketch is found. Therefore, when synthesizing a swizzle-free sketch, we introduce two additional constructs to the search grammar that are used to represent data movement: `??load` and `??swizzle`.

Figure 5.10 shows the definitions of `??load` and `??swizzle` operations in pseudo-code. The `??load` operation implements the initial loading of data from memory into a vector register or vector register pair. There are three inputs to the `??load` operation: (1) the set of data values read from memory by the input expression (`live-data`), (2) a boolean flag indicating whether the operation loads a vector or a vector-pair (`vec-pair?`) and (3) the required type for elements in the output vector (`elemT`). The `??load` operation then returns a *symbolic* vector (or vector-pair), where each lane of the vector holds one of the live data values of the requested element type. A symbolic vector encodes the set of all possible vectors (of the requested type) that can be constructed from swizzling live data. This allows the

synthesizer to concretize the symbolic vector into any one of its possible values, allowing us to represent all possible swizzling patterns. The `??swizzle` operation similarly implements the re-arrangement of data produced by one or more sub-expressions. Instead of returning a symbolic vector of data read from memory, it returns a symbolic vector populated with data produced by sub-expressions.

5.4.1 Verifying Sketches

In order to prove the validity of a candidate swizzle-free sketch, Rake must select a concrete instantiation for each symbolic vector produced by the `??load` and `??swizzle` operations, such that the output of the overall expression matches the output of the input expression we are trying to lower. The verification problem can be formally specified as follows:

$$\exists\, v_0, \ldots, v_n.\ \ \forall i \in lanes.\ \ input[i] = sketch[i]$$

where $v_0, \ldots, v_1$ represent the concrete instantiations for each symbolic vector.

The amount of work a synthesizer must do to find concrete instantiations for each vector is proportional to the number of lanes in that vector since the synthesizer must pick a value to populate each lane. As vectors grow larger (HVX vectors have up to 128 lanes), this becomes expensive. Additionally, more lanes in the output vector generally mean the set of live-data values to choose from is also larger. Fortunately, both of these challenges can be addressed by verifying incrementally for each lane of the output vector. For instance, we can simplify the verification query to only verify for the first lane of the vector:

$$\exists\, v_0, \ldots, v_n.\ \ input[0] = sketch[0]$$

The synthesizer must now only instantiate lanes of the symbolic vectors $v_0, \ldots, v_n$ that are required to compute the first lane of the overall output. While such a query does not guarantee the sketch is correct, it allows Rake to quickly reject obviously incorrect sketches, since if the two expressions produce unequal outputs for the first lane of the vector, the sketch cannot be correct. The more expensive verification query is then reserved for swizzle-free sketches that pass this initial pruning step.

5.5 Synthesizing Swizzles

Once a valid swizzle-free sketch is found, Rake synthesizes an implementation to replace each `??load` or `??swizzle` operation in the sketch. The synthesis problem can be formulated as follows:

$$\forall\, v_0, \ldots, v_n \in \tau.\ \exists\, e_0, \ldots, e_n \in \mathcal{G}_{sw}.$$
$$interpret(\tau[v_i \rightarrow e_i]) = interpret(e)$$

For all symbolic vectors in the sketch $v_0, \ldots, v_n$ (each representing an abstract swizzle), we search for swizzle expressions $e_0, \ldots, e_n$ constructed using the grammar of swizzle intrinsics G_{sw} in the target ISA, such that replacing the symbolic vectors with the swizzle expression still produces the correct output. In practice, Rake will not try to replace all abstract swizzles at once, but do so one at a time.

5.5.1 Backtracking and Intermediate Data Layouts

The incremental approach described thus far for lowering expressions from Uber-Instruction IR to the target ISA, although scalable, runs the risk of introducing inefficiencies in the final implementation.

Sub-optimal Sketches The most efficient swizzle-free sketches do not necessarily yield the most performant implementations. The cost overhead of the required data movement may outweigh the benefit of using fewer compute instructions. To address this, we introduce backtracking to our lowering algorithm, shown in Algorithm 4. Whenever a lowered implementation ϵ is synthesized for the input expression, Rake updates the expression cost upper bound β (initially set to infinite) and then backtracks to synthesize another implementation. With each new implementation, the cost upper bound is tightened until a better implementation cannot be found.

Input: An expression in the uber-instruction IR

Output: An equivalent expression in the target ISA

```
1  Function Lower(e, ℓ)
2      I ← null                                  ▷ Best lowered implementation
3      β ← ∞                                     ▷ Expression cost upper-bound
4      for sl ∈ SubexprLayouts(e, ℓ) do
5          S ← ∅                                 ▷ Set of lowered sub-exprs
6          for se ∈ Subexprs(e) do
7              S ← S ∪ Lower(se, sl)
8          while τ ← SynthesizeSketch(e, S, β) ≠ unsat do
9              υ ← InferCost(τ)
10             if ε ← SynthesizeSwizzles(e, τ, ℓ, β − υ) ≠ unsat then
11                 I ← ε
12                 β ← InferCost(ε)
13     return I
```

Algorithm 4: Lowering expressions from the Uber-Instruction IR to the target ISA.

Intermediate Data Layouts Since Rake builds the output expressions bottom-up by lowering one uber-instruction at a time, the lowered sub-expressions are verified against sub-expressions in the input lifted expressions. This is problematic since it forces lowered sub-expressions to produce their output in the same layout as the input lifted sub-expressions. For example, since the HVX `vtmpy` intrinsic produces deinterleaved output, the implementation in Figure 5.8 interleaves the output produced by the intrinsic to undo the implicit deinterleaving. However, if the expression being lowered is producing an intermediate output, then it may be beneficial to not interleave the output for two reasons: 1) just as how `vtmpy` produces deinterleaved output, other instructions down the pipeline may interleave the output, thus eliminating the need to do the swizzle at all, 2) even if the swizzle is re-

quired, it may be less expensive to do it later. For example, it is much cheaper to swizzle the output of a large reduction than to swizzle all of its inputs. To address this concern, we parameterize our lowering algorithm over the data layout ℓ of the output we wish to lower to. This allows Rake to synthesize a variety of lowered implementations for each sub-expressions, each producing different permutations of the same intermediate output. For HVX, we consider interleaved and deinterleaved layouts since HVX instructions only perform these permutations implicitly. Simpler ISAs that do not have implicit data movement in compute instructions may not require this step.

5.6 Implementation

Rake's core synthesis algorithm is implemented in Racket, using the Rosette 4.0 framework [87] with z3 [34] as the back-end solver. We've also added C++ code inside Halide to extract qualifying Halide IR expressions and compile them to Racket syntax. We have also implemented within Halide an S-Expression parser to convert the S-Expressions synthesized by Rake back to Halide IR. Finally, we manually implemented an interpreter (in Racket) for both the HVX intrinsics provided by LLVM as well as the derived uber-instructions.

Deriving Uber-Instruction IR The quality of code generated by our lift-then-lower approach is sensitive to the design of the Uber-Instruction IR used. For instance, if the uber-instructions are too general (coarse Uber-Instruction IR), the set of ISA instructions that need to be enumerated when lowering an uber-instruction can get large. This in turn makes synthesis slow or at times intractable. On the other hand, if the uber-instructions are too low-level, there can be many different ways to express the input expression in the Uber-Instruction IR and very few ways to lower the lifted representation down to the target ISA. Our instruction-selection algorithm then effectively becomes a greedy instruction selector, which can easily generate sub-optimal implementations since it explores a very small subspace of all possible implementations. The goal is to design an Uber-Instruction IR coarse enough for the greedy lifting algorithm to find the correct representation, yet not

so coarse that lowering becomes intractable. To derive the set of uber-instructions for HVX, we identified clusters of intrinsics that had related or overlapping semantics. This was fairly straightforward since the HVX documentation already lists similar instructions together. Then, we manually defined an uber-instruction that implemented the common higher-level compute pattern. The uber-instructions were designed such that each intrinsic in the target ISA was expressible by at least one uber-instruction. For example, we can express the HVX vector-addition intrinsic *vadd* using the `vs-mpy-add` uber-instruction, since addition is simply a multiply-add with multiplicative weights of `(1 1)` (that is, a multiply-add where each input is first multiplied by 1 then added together).

Cost Model Our implementation uses a variation of instruction-count to estimate the cost of an HVX expression. Since HVX has multiple hardware resources (such as multiply, shift or permute) and different instructions can execute on different hardware resources within the same cycle, we count the number of instructions per resource and take the maximum of the computed values. This biases our cost model towards implementations that distribute the computation across resources.

Extending to other ISAs At a high-level, extending Rake to a new target ISA requires three inputs:

- An interpreter for the intrinsics available in the new target backend, implemented in Racket.
- An Uber-Instruction IR implementing the high-level patterns available in the new backend.
- A grammar mapping each uber-instruction to the subset of the ISA that must be explored during lowering.

Follow-up research to this work has successfully extended Rake to target ARM's Neon vector instructions [76]. The Uber-Instruction IR used to generate ARM expressions is largely derived from our HVX Uber-Instruction IR but introduces some new high-level compute patterns that were missing in HVX, such as `add-high-narrow` (an add followed by a high-half narrow of the result). This is rather unsurprising since both ARM and HVX target the same high-level domain of fixed-point arithmetic. Furthermore, they were able to automatically generate a large portion of the required ARM Neon interpreter by leveraging Halide's ARM-specific code optimization rules. These re-write rules, originally intended to map Halide IR expressions to ARM instructions, can be used to map each ARM instruction to an equivalent Halide IR expression, which Rake can already interpret. For some backends, interpreters can be automatically generated from the pseudo-code provided in the documentation, as demonstrated by Vegen [24] for Intel x86 SIMD instructions.

With the aforementioned changes, Rake was able to achieve a mean speedup of 12% over Halide across 11 real-world benchmarks, with up to 65% faster runtimes observed in some benchmarks. In general, we believe our algorithm is suitable for any target backend that, similar to HVX, has a large number of instructions containing many variants of relatively few compute patterns. In contrast, backends that expose only a small number of very distinct instructions would not benefit much from our approach.

5.7 Evaluation

We evaluate Rake using a suite of 21 benchmarks, listed in Table 5.1. The benchmarks consist of open-source applications taken from the Halide repository as well as sample Halide programs provided in the Hexagon Software Development Kit (SDK) v3.5.2. These benchmarks, summarized below, span a range of image processing, computational photography, computer vision and machine learning workloads.

- *Image Processing.* Our test suite includes image processing operations, implementing fundamental operations such as blurs (box blur, gaussian blur, median filter), edge

detection (Sobel filter), image dilation and general 3×3 convolutions. For Gaussian blur, we include implementations for three different radii: 3, 5 and 7. For general convolutions, we include implementations for both 16-bit and 32-bit accumulators.

- *Machine Learning.* This subset contains Halide implementations of core Tensorflow operations, including normalization layers (l2norm, softmax), elementwise layers (add, mul), pooling layers (average_pool, max_pool), reduction (mean), fully connected layers, and convolutional layers (conv, depthwise_conv).

- *Camera Pipeline.* This is the Frankencamera pipeline [2] for processing raw data from an image sensor into a color image. The pipeline performs hot-pixel suppression, demosaicking, color correction, gamma correction, and contrast.

- *Matrix Multiplication.* This benchmark implements quantized matrix-multiplication of two unsigned 8-bit matrices.

For all benchmarks, we use the existing Halide schedules found in the implementations; Rake only changes instruction selection and does not alter the overall structure of the generated code. The set of benchmarks outlined above generates a total of 450 qualifying vector expressions that Rake attempts to optimize. Rake currently ignores all scalar expressions as well as all trivial vector expressions, such as a single variable, non-strided vector-loads, or scalar broadcasts. Rake assumes these are handled correctly by LLVM.

The benchmarks were compiled on a Windows 10 desktop machine with an AMD Ryzen Threadripper 2950X 3.5GHz 16-core CPU and 128GB of RAM. All HVX runtime performance numbers were computed using the reported cycle counts from Qualcomm's Hexagon Simulator v8.3.07.

5.7.1 Runtime Performance

To evaluate the effectiveness of Rake's instruction selection algorithm, we compare the runtime performance of all benchmarks for the Rake HVX backend, using the existing

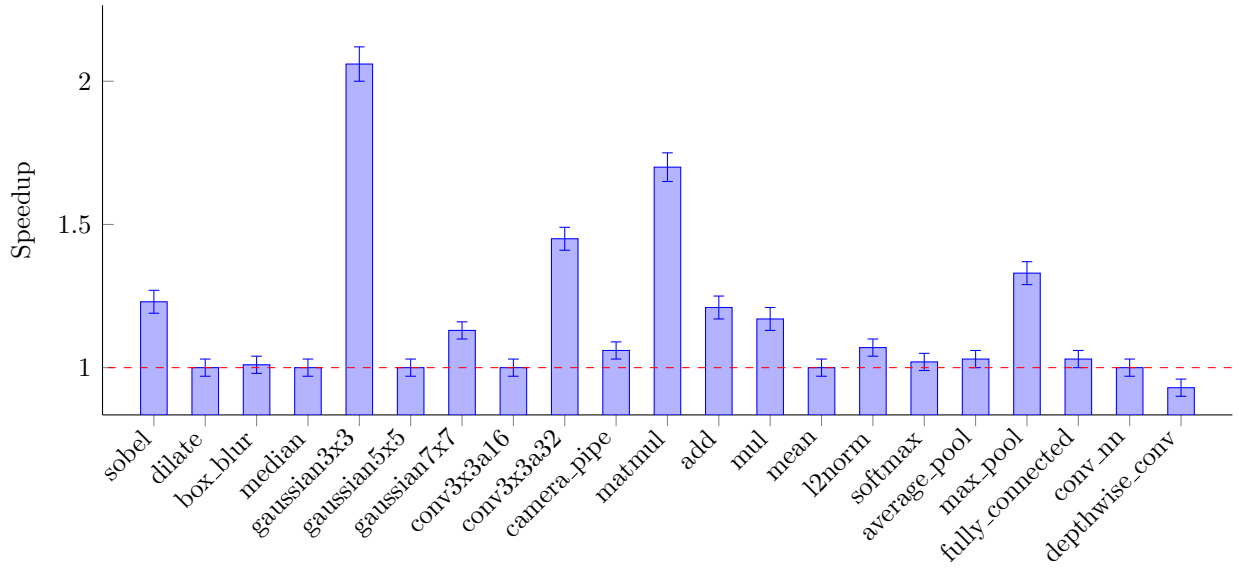


Figure 5.11: Speedups for Rake over the default Halide HVX backend. Across the 22 benchmarks, Rake improves performance by an average of 18% over the existing highly-optimized backend, which has been developed over years by Qualcomm and Google engineers.

Halide 12.0 HVX backend as the baseline. The existing HVX backend is developed by Qualcomm and Google engineers, representing years of developer effort, and has been used for compiling production Android code distributed to millions of devices [91, 72].

Figure 5.11 graphs the results of our experiment. On average, Rake improved the overall runtime performance by 18%, with a maximum observed speedup of 2.1× in the `gaussian3x3` benchmark and the lowest observed speedup of 0.93× in `depthwise_conv`. 10 of the 21 benchmarks demonstrated a performance improvement beyond the 3% margin of error introduced by the simulator, with another 10 benchmarks showing identical performance. Upon manual inspection of the generated code, we discovered that Rake did improve instruction selection in some of these benchmarks. However, these optimizations did not result in overall performance improvement either because the benchmarks were memory-bound or because the optimizations were not in the critical code path. Rake performed worse than Halide's optimizer on only a single benchmark.

To illustrate the breadth of optimizations discovered by Rake, we now discuss a handful

of representative examples, shown in Figure 5.12. The figure shows the Halide IR expression, as well as the optimized code generated by Halide and by Rake.

Missing Optimization Patterns

The first class of improvements made by Rake over Halide's existing Hexagon optimizer relate to identifying missing optimization patterns. Through search, Rake considers a much larger space of implementations and discovers optimizations not handled by any of Halide's existing rewrites. We already highlighted three such instances from the Sobel filter in §5.2.1. In Figure 5.12, we provide three more examples from other benchmarks in our test suite.

- `average_pool`**:** The input code implements an addition between vectors of type `uint16` and `uint8`. The Halide implementation first zero-extends the `uint8` vector and then performs vector addition to complete the implementation. Rake, by contrast, uses a single widening multiply-add instruction with a multiplicative weight of 1.

- `camera_pipe`**:** In this example, Rake removes the `vmax` instruction since the instruction `vpackub` already saturates the value to an unsigned byte, making the max with zero operation redundant.

- `add`**:** Rake is able to fold the shift operation into a widening multiply-add, implemented using the single `vmpy-acc` intrinsic. Halide instead zero-extends the input before implementing the shift-left and addition using a non-widening multiply-add using `vmpyi-acc`. Since `vmpy-acc` generates a vector-pair as output, two `vmpyi-acc` instructions are needed to compute the equivalent tile.

Semantic Reasoning

In addition to discovering optimization patterns missing in Halide's rule-set, Rake can also discover *context-specific* optimizations that require semantic reasoning about the expression,

	Benchmark	Input (Halide IR)	Halide Codegen (HVX)	Rake Codegen (HVX)

Missing Patterns	average_pool	`wild_u16x`
		`+`
		`uint16x128(wild_u8x)`
			`/* Latency: 3 */`
			`wild_u16x + vzxt(wild_u8x)`
				`/* Latency: 2 */`
				`vmpy-acc(`
				` wild_u16x,`
				` wild_u8x,`
				` 1)`

	camera_pipe	`uint8x128(`
		`  max(`
		`  min(`
		`  wild_i16x,`
		`  x128(127)),`
		`  x128(0)))`
			`/* Latency: 4 */`
			`// saturate to uint8`
			`vpackub(`
			`  vmax(`
			`  vmin(`
			`  wild_i16x,`
			`  vsplatb(127)),`
			`  vsplatb(0)))`
				`/* Latency: 3 */`
				`vpackub(`
				`  vmin(`
				`  wild_i16x,`
				`  vsplat(127)))`

	add	`(`
		` int16x128(wild_u8x)`
		` <<`
		` 6`
		`)`
		`+`
		`x128(`
		`  int16(wild_u8)`
		`  *`
		`  -64`
		`)`
			`/* Latency: 6 */`
			`x = vzxt(wild_u8x)`
			`vmpyi-acc( // Multiply-add`
			`  vsplat(`
			`  int16(wild_u8) * -64),`
			`  (lo x), // Lower 64 lanes`
			`  64)`
			`vmpyi-acc(`
			`  vsplat(`
			`  int16(wild_u8) * -64),`
			`  (hi x), // Upper 64 lanes`
			`  64)`
				`/* Latency: 2 */`
				`// Widening multiply-add`
				`vmpy-acc(`
				`  vsplat(`
				`  int16(wild_u8) * -64`
				`  ),`
				`  wild_u8x,`
				`  64)`

Semantic Reasoning	l2norm	`x64(wild_i32)`
		`*`
		`int32x64(wild_i16x)`
			`/* Latency: 6 */`
			`// Mul i32s with odd i16s`
			`vmpyio(`
			`  vsplat(wild_i32),`
			`  wild_i16x)`
			`vmpyio(`
			`  vsplat(wild_i32),`
			`  // Shift-left`
			`  vaslw(wild_i16x, 16))`
				`/* Latency: 4 */`
				`// Mul i32s with odd i16s`
				`vmpyio(`
				`  vsplat(wild_i32),`
				`  wild_i16x)`
				`// Mul i32s with even i16s`
				`vmpyie(`
				`  vsplat(wild_i32),`
				`  wild_i16x)`

	gaussian3x3	`uint8x128(`
		`  (`
		`  wild_i16x`
		`  +`
		`  x128(8)`
		`  )`
		`  >>`
		`  x128(4))`
			`/* Latency: 8 */`
			`// extract lower byte`
			`vshuffeb(`
			`  vasr( // shift-right`
			`  wild_i16x + vsplat(8),`
			`  4),`
			`  vasr(`
			`  wild_i16x + vsplat(8),`
			`  4))`
				`/* Latency: 2 */`
				`// Fused shift right, round`
				`// and saturate`
				`vasr-rnd-sat(`
				`  wild_i16x,`
				`  4)`

Figure 5.12: Rake uses search to find new optimization opportunities not considered by the existing Halide HVX backend. Here, `wild_{i|u}bb{x|}` represents a subtree of signed (`i`) or unsigned (`u`) `bb`-bit value that is either a scalar (no suffix) or vector (`x`).

such as inferring the range of values possible for an intermediate output. Figure 5.12 shows two examples from the benchmarks `l2norm` and `gaussian3x3`.

- **`l2norm`:** The input IR multiplies a vector of words with a vector of halfwords. Halide generates the results in two steps: first, it multiplies all odd halfwords with the vector of words using the `vmpyio` instruction, and then it uses the shift-left instruction `vaslw` to move the even halfwords into the odd indices and repeats the first step. Rake, on the other hand, avoids the shift-left operation and instead uses the `vmpyie` instruction to directly multiply the even halfwords with the vector of words. Interestingly, HVX only offers the `vmpyie` instruction for unsigned halfwords. Therefore, to use this instruction safely, Rake must prove that the sub-expression producing the input to this pattern will never produce negative values (i.e, the most significant bit is always 0).

- **`gaussian3x3`:** Rake uses a fused instruction to implement the rounding-shift-right as well as the lowering cast. This transformation is only safe if the upper-most 8-bits of the input values are always 0. In other words, performing a truncating cast or a saturating cast produces the same outputs.

Data Movement

The third major category of perfomance improvements comes from improved data movement.

- **`gaussian3x3`, `conv3x3a32`:** In some benchmarks, such as the Sobel filter and `conv3x3a32`, Rake exploits fused multiply-add sliding-window instructions such as `vtmpy` (3-wide reduction) or `vrmpy` (4-wide reduction). A major advantage of of these instructions is that they reduce the number of vector loads necessary. For instance, row (a) in Figure 5.4 shows that in addition to the smaller compute latency, the `vtmpy` instruction requires one fewer vector load.

- `add`, `mul`, `average_pool`, `max_pool`, `matmul`: We found that Rake frequently avoids unnecessary data shuffling operations introduced by Halide's optimizer. The most common example of this was Halide adding an interleave operation to undo the implicit deinterleaving of a prior compute instruction (or vice versa). While Halide's optimizer has an optimization pass dedicated specifically to eliminating such unnecessary interleaves and deinterleaves, it is not always able to do so.

5.7.2 Compilation Performance

Table 5.1 shows a breakdown of compilation times for each benchmark. On average, Rake took 62 minutes to compile each benchmark, with a median compilation time of roughly 21 minutes. The largest expression optimized by Rake required a sequence of 103 HVX intrinsics to implement.

The mean time spent lifting to the Uber-Instruction IR was 154 seconds, which equates to 9% of the compilation time. Synthesizing swizzle-free sketches was more expensive, taking on average 397 seconds or 21% of the total compilation time. Synthesizing data movement accounted for the majority of compilation time, taking on average 53 minutes and making up almost 70% of the total synthesis time. There are three reasons why considerably more time is spent on synthesizing data movement than on synthesizing swizzle-free sketches. First, the search space for swizzles suffers from *symmetry*, resulting in a larger set of candidate expressions. In program synthesis, symmetries arise when multiple expansions of a grammar result in the same program; the synthesizer unnecessarily considers the same program multiple times. Second, Rake can specialize the search space for compute instructions by leveraging semantic information exposed by lifting. The same cannot be done for swizzling, so Rake always considers the full set of shuffling instructions. Lastly, due to the backtracking nature of our instruction selection algorithm, Rake often spends considerable time trying to prove that the required swizzle cannot be implemented within a given instruction budget.

Compared to pattern-matching optimizers, such as Halide's existing HexagonOptimizer,

Table 5.1: Compilation statistics for the Hexagon HVX backend.

Benchmark	Optimized Exprs	Lifting Queries	Sketching Queries	Swizzling Queries	Lifting Time (s)	Sketching Time (s)	Swizzling Time (s)	Total Time (s)
sobel	4	348	252	3748	12	27	7274	7313
dilate	8	560	1376	1168	45	48	114	207
box_blur	4	76	820	597	4	538	1033	1575
median	40	1440	5256	5360	119	202	397	718
gaussian3x3	4	368	100	376	46	52	137	235
gaussian5x5	8	364	98	1636	22	20	214	256
gaussian7x7	20	864	320	6835	50	173	1358	1581
conv3x3a16	4	320	73	1254	96	2552	953	3601
conv3x3a32	4	404	41	1560	204	1628	750	2582
camera_pipe	44	2037	6177	9002	1153	429	13782	15364
matmul	10	594	456	6224	104	175	6835	7114
add	4	306	403	638	47	118	431	596
mul	4	492	427	576	129	184	1559	1872
mean	2	43	37	298	11	82	186	279
l2norm	4	262	163	301	49	13	77	139
softmax	18	755	679	1311	221	352	683	1256
average_pool	6	186	476	2246	6	42	248	296
max_pool	6	24	156	126	1	6	8	14
fully_connected	39	608	198	416	63	160	512	735
conv_nn	140	3673	4165	1612	558	1002	19335	20895
depthwise_conv	77	2926	3542	7941	301	538	10593	11432

Rake presents a different performance to compilation time trade-off. As an offline optimizer, Rake can be used to fine-tune applications for a given hardware platform. In addition, Rake can be a valuable tool to inform compiler developers of patterns and optimizations that are both missing in their machinery and important for performance. Furthermore, since Rake uses SMT solvers for search and verifies its transformations, it can even highlight bugs in the rule-based compiler. In fact, during our manual inspection of the generated code, we found three bugs in Halide's HVX code-generation involving unsafe instruction selection. We have communicated these bugs to the Halide developers and the appropriate fixes have been merged into the master branch. We intend to keep working with Halide developers and share the optimization patterns we discovered.

5.7.3 Limitations

While Rake's algorithm is designed to generalize to other ISAs, such as ARM and Intel's AVX, our prototype currently only supports Hexagon's HVX backend. Our experiments also highlighted two significant limitations in the way Rake is integrated into Halide. Unlike Halide's existing optimizer, which may modify the layout in which data is stored in an intermediate buffer to enable better instruction selection across multiple expressions, Rake optimizes each expression individually. This was the key reason behind the performance degradation observed in the `depthwise_conv` benchmark, but also reduced the speedup observed in other benchmarks like `average_pool`. Secondly, in some cases, better instruction selection was possible if Rake had access to certain loop-invariants. For example, when the Halide schedule re-uses reads in a rotating buffer, Rake is unaware of the relationship between data read in this iteration and the data read in previous iterations, preventing it from using more optimal sliding window instructions.

5.8 Related Work

Superoptimization is the task of using search to optimize low-level programs based on instruction semantics. Traditionally superoptimizers like Optgen [22], Lens [65] and the one

proposed by Bansal and Aiken [16] have focused on peephole optimization over short instruction sequences, and target a broader class of optimizations than just instruction selection. Rake by contrast is designed to be a target-specific instruction-selector that can operate on much larger instruction sequences. Souper [77], a middle-end LLVM IR superoptimizer, performs rewrites on its own IR via dataflow analysis. Unlike Rake, Souper does not support vector instructions or hardware-specific intrinsics.

Autovectorization is the related task of converting scalar code into vectorized implementations. Vegen [24] is an auto-vectorizer that jointly performs instruction-selection for complex vector ISAs. Unlike Rake, Vegen uses automatically generated pattern matching rules for instruction selection. While automatically generated, the pattern matching rules suffer from the same limitations as Halide's optimizer: they cannot perform semantic reasoning on expressions and may miss creative applications of instructions. Diospyros [88] is another auto-vectorizer designed to synthesize efficient implementations of kernels on DSP architectures. It uses equality saturation to identify creative swizzles and vectorization patterns in its own IR. When lowering the discovered shuffles to the backend target, it delegates the instruction-selection process to the vendor-supplied DSP compiler toolchain.

Data Swizzling is the task of inferring permutations of data and computation to optimize performance. Swizzle Inventor [64] is a tool that infers swizzles to optimize applications for GPU memory hierarchies. Unlike Rake, Swizzle Inventor can only synthesize data-movement and requires the users to provide a sketch describing where swizzles can be added; on the other hand, integrating Swizzle Inventor into Rake could improve the quality of our swizzle synthesis.

5.9 Conclusion

In this chapter, we described Rake, which takes code in the Halide intermediate representation and uses program synthesis to perform target-specific instruction selection for the Hexagon HVX digital signal processor. Rake scales to real-world vectorized expressions

by decomposing the task into three different synthesis queries. On a suite of 21 real-world benchmarks, Rake improves performance an average of 18% and up to 2.1× over the existing combination of Halide and LLVM, which use human-created pattern-matching rules to perform instruction selection.

Chapter 6

REFLECTIONS, FUTURE WORK AND CONCLUSIONS

The work presented in this thesis has demonstrated the potential of using verified lifting to implement the next-generation of compiler optimizations, enabling us to automate tasks such as re-writing legacy code to new DSLs that were not feasible before. In this final chapter, we step back and reflect on the three case studies presented in chapters 2, 3, and 4. We highlight key insights derived from our work and propose possible directions for the future. We end by summarizing the contributions of this dissertation.

6.1 Insights & Future Work

We believe this thesis could be the prelude to a new generation of programming tools. We envision the following future designs and systems to make verified lifting more accessible and ubiquitous in compilers.

6.1.1 A Framework for Building Verified-lifting Compilers

Verified lifting compilers do not need a set of complex pre-defined translation rules to operate. However, they do require the semantics of the target API or DSL expressed in a form that the underlying synthesizers and verifiers can understand. Furthermore, the compiler developer must implement Hoare style verification condition generation for the language the input source code was written in. Neither of these tasks is trivial and can require specialized formal methods knowledge on behalf of the compiler designer.

Fortunately, much of the complexities associated with building verified lifting compilers can either be abstracted away or eliminated by designing a meta-framework for building compilers. Such a framework could define a new solver-compatible language for expressing

DSL semantics or implement verification condition generation for a common IR (such as LLVM's IR) to enable re-use across different verified lifting compilers. Such a platform would also be fertile grounds for further research, enabling quick prototyping of domain-specific lifting algorithms or specifying domain-specific axioms to aid verification.

6.1.2 Memoizing Optimizations to Improve Compilation Performance

A limitation of our synthesis-based approach, especially when compared against syntax-driven methods, is the longer compilation times. This is exacerbated by the fact that our current methodology does not learn from prior compilations. Even if the same expression or program is compiled twice using our compilers, it would perform the search from scratch each time. We believe there is ample opportunity to memoize past optimizations either in the form of grammar specializations (sketches) or in the form of heuristics for guiding the search-space exploration.

We believe this is particularly applicable in the case of instruction selection, where our synthesis-based instruction selector could be used as an oracle to discover missing re-write rules in the classical pattern-matching optimizer. The more expensive search-based optimizations could then be reserved only for final production builds, when lower compilation times may be sacrificed for better optimization.

6.1.3 Bringing Humans Into the Loop

In both Casper and Dexter, a significant portion of benchmarks that failed to translate were due to the compiler's inability to establish program properties that would be easy for the human to specify. For example, the compiler may not be able to infer whether two input points are aliases but this may be obvious to the programmer or even part of the legacy code's documentation. In such cases, one can imagine an interactive process where the compiler queries the developer or requests annotations on the input code to aid with compilation.

Bringing humans into the synthesis loop is not trivial and runs the risk of putting too much burden on the developers in the form of numerous queries. However, we believe that

with the right balance of human interaction and automated reasoning, more robust compilers could be built with a much higher success rate.

All of these possibilities indicate a bright future for the synthesis-aided development of high-performance software.

6.2 Conclusion

Lifting compiler transformations, whereby lower-level code is re-written in a higher-level programming abstraction, are essential to building the next generation of optimizing compilers. In this thesis, we presented three compilers that use verified lifting to optimize the performance of data-processing code by leveraging modern data-processing APIs.

Concretely, the contributions of this thesis include:

- A methodology for modernizing legacy large-scale data-processing applications by translating them to MapReduce frameworks (Chapter 3). We present Casper, a new tool that implements our methodology to produce provably correct translations, that perform up to 48.2× faster. Our results show that code generated by Casper performs comparably to hand-translated code by experts.

- In 3.4, we describe strategies for making synthesis-based code rewrites robust to verification failures and improving search scalability through incremental grammar expansion.

- In 3.5, we explain how cost-models can be incorporated into verified lifting to discover not just valid but optimal translations.

- In Chapter 4, we present a methodology for re-writing image processing pipelines in C++ to the Halide DSL. We describe a new three-stage synthesis algorithm for inferring program semantics incrementally and illustrate how domain-specific insights can be used to scale up the search where state-of-the-art synthesis algorithms are insufficient.

- We implement our methodology for lifting image-processing pipelines in a new tool called Dexter. We use Dexter to rejuvenate over 36,000 lines of code taken from the popular Adobe Photoshop software. Our results demonstrate the real-world applicability and benefits of this approach, with over 70% of the translated operations performing at least 2× faster.

- In Chapter 5, we discuss a methodology for re-writing general-purpose vector expressions in the Halide DSL IR to exotic hardware instruction sets. We introduce the concept of uber-Instructions and explain how they can be used to reveal high-level compute patterns implemented in the input expressions.

- We implement our instruction selection methodology into a new tool called Rake. Rake is implemented within Halide and can synthesize Hexagon HVX and ARM Neon expressions without any instruction-selection rules. Our results demonstrate that Rake is often able to discover optimizations that both Halide and LLVM's pattern matching infrastructure fails to detect.

BIBLIOGRAPHY

[1] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. Learning to optimize halide with tree search and random programs. *ACM Transactions on Graphics (TOG)*, 38(4), 2019.

[2] Andrew Adams, Eino-Ville Talvala, Sung Hee Park, David E. Jacobs, Boris Ajdin, Natasha Gelfand, Jennifer Dolson, Daniel Vaquero, Jongmin Baek, Marius Tico, Hendrik P. A. Lensch, Wojciech Matusik, Kari Pulli, Mark Horowitz, and Marc Levoy. The frankencamera: An experimental platform for computational photography. In *ACM SIGGRAPH 2010 Papers*, SIGGRAPH '10, New York, NY, USA, 2010. Association for Computing Machinery.

[3] Adobe. Pixel bender language reference, 2010.

[4] Maaz Bin Safeer Ahmad and Alvin Cheung. Automatically leveraging mapreduce frameworks for data-intensive applications. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1205–1220, New York, NY, USA, 2018. ACM.

[5] Maaz Bin Safeer Ahmad, Jonathan Ragan-Kelley, Alvin Cheung, and Shoaib Kamil. Automatically translating image processing libraries to halide. *ACM Transactions on Graphics*, 38:1–13, 11 2019.

[6] 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.

[7] Alexander Alexandrov, Asterios Katsifodimos, Georgi Krastev, and Volker Markl. Implicit parallelism through deep language embedding. *SIGMOD Rec.*, 45(1):51–58, June 2016.

[8] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 1–17, 2013.

[9] Saman P. Amarasinghe, Jennifer-Ann M. Anderson, Monica S. Lam, and Chau-Wen Tseng. An overview of the SUIF compiler for scalable parallel machines. In *PPSC*, pages 662–667, 1995.

[10] `https://flink.apache.org/`, 2022. Accessed on: 2022-05-27.

[11] `http://hadoop.apache.org`, 2022. Accessed on: 2022-05-27.

[12] `http://hive.apache.org`, 2022. Accessed on: 2022-05-27.

[13] `https://pig.apache.org/`, 2022. Accessed on: 2022-05-27.

[14] `https://spark.apache.org`, 2022. Accessed on: 2022-05-27.

[15] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1383–1394, New York, NY, USA, 2015. ACM.

[16] Sorav Bansal and Alex Aiken. Automatic generation of peephole superoptimizers. *SIGOPS Oper. Syst. Rev.*, 40(5):394–403, October 2006.

[17] William Blume, Rudolf Eigenmann, Jay Hoeflinger, David A. Padua, Paul Petersen, Lawrence Rauchwerger, and Peng Tu. Automatic detection of parallelism: A grand challenge for high performance computing. *IEEE P&DT*, 2(3):37–47, 1994.

[18] Rastislav Bodík and Barbara Jobstmann. Algorithmic program synthesis: introduction. *International Journal on Software Tools for Technology Transfer*, 15:397–411, 2013.

[19] Pedro Boechat, Mark Dokter, Michael Kenzel, Hans-Peter Seidel, Dieter Schmalstieg, and Markus Steinberger. Representing and scheduling procedural generation using operator graphs. *ACM Trans. Graph.*, 35(6):183:1–183:12, 2016.

[20] Matthias Boehm, Michael W. Dusenberry, Deron Eriksson, Alexandre V. Evfimievski, Faraz Makari Manshadi, Niketan Pansare, Berthold Reinwald, Frederick R. Reiss, Prithviraj Sen, Arvind C. Surve, and Shirish Tatikonda. Systemml: Declarative machine learning on spark. *Proc. VLDB Endow.*, 9(13):1425–1436, September 2016.

[21] Antoni Buades, Bartomeu Coll, and Jean-Michel Morel. A non-local algorithm for image denoising. In *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, CVPR '05, pages 60–65, Washington, DC, USA, 2005. IEEE Computer Society.

[22] Sebastian Buchwald. Optgen: A generator for local optimizations. In *International Conference on Compiler Construction*, pages 171–189. Springer, 04 2015.

[23] Bryan Catanzaro, Shoaib Kamil, Yunsup Lee, Krste Asanović, James Demmel, Kurt Keutzer, John Shalf, Kathy Yelick, and Armando Fox. Sejits: Getting productivity and performance with selective embedded jit specialization. In *PMEA*, 2009.

[24] Yishen Chen, Charith Mendis, Michael Carbin, and Saman Amarasinghe. Vegen: A vectorizer generator for simd and beyond. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 902–914, New York, NY, USA, 2021. Association for Computing Machinery.

[25] Yu-Fang Chen, Lei Song, and Zhilin Wu. The commutativity problem of the mapreduce framework: A transducer-based approach. *CoRR*, abs/1605.01497, 2016.

[26] Alvin Cheung, Samuel Madden, Armando Solar-Lezama, Owen Arden, and Andrew C. Myers. Using program analysis to improve database applications. *IEEE Data Eng. Bull.*, 37(1):48–59, 2014.

[27] Alvin Cheung and Armando Solar-Lezama. Computer-assisted query formulation. *Foundations and Trends in Programming Languages*, 3(1):1–94, 2016.

[28] 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, pages 3–14, New York, NY, USA, 2013. ACM.

[29] L. Codrescu, W. Anderson, S. Venkumanhanti, M. Zeng, E. Plondke, C. Koob, A. Ingle, C. Tabony, and R. Maule. Hexagon DSP: An architecture optimized for mobile multimedia and communications. *IEEE Micro*, 34(02):34–43, mar 2014.

[30] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Ugur Çetintemel, and Stanley B. Zdonik. Tupleware: Redefining modern analytics. *CoRR*, abs/1406.6667, 2014.

[31] Przemyslaw Daca, Thomas A. Henzinger, and Andrey Kupriyanov. Array folds logic. *CoRR*, abs/1603.06850, 2016.

[32] L. Dalcin, R. Bradshaw, K. Smith, C. Citro, S. Behnel, and D. S. Seljebotn. Cython: The best of both worlds. *Computing in Science & Engineering*, 13:31–39, 09 2010.

[33] Jerome Darbon, Alexandre Cunha, Tony F. Chan, Stanley Osher, and Grant J. Jensen. Fast nonlocal filtering applied to electron cryomicroscopy. In *ISBI*, pages 1331–1334. IEEE, 2008.

[34] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[35] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[36] K. Venkatesh Emani, Karthik Ramachandra, Subhro Bhattacharya, and S. Sudarshan. Extracting equivalent sql from imperative code in database applications. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1781–1796, New York, NY, USA, 2016. ACM.

[37] Grigory Fedyukovich, Maaz Bin Safeer Ahmad, and Rastislav Bodik. Gradual synthesis for static parallelization of single-pass array-processing programs. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 572–585, New York, NY, USA, 2017. ACM.

[38] `https://github.com/fiji`, 2022. Accessed on: 2022-05-27.

[39] Brian Guenter and Diego Nehab. The neon image processing language. Technical report, Microsoft Research, March 2010.

[40] Sumit Gulwani. Dimensions in program synthesis. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, PPDP '10, pages 13–24, New York, NY, USA, 2010. ACM.

[41] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 317–330, 2011.

[42] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Foundations and Trends in Programming Languages*, 4(1-2):1–119, 2017.

[43] James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. Darkroom: compiling high-level image processing code into hardware pipelines. *ACM Trans. Graph.*, 33(4):144:1–144:11, 2014.

[44] James Hegarty, Ross G. Daly, Zachary DeVito, Mark Horowitz, Pat Hanrahan, and Jonathan Ragan-Kelley. Rigel: flexible multi-rate image processing hardware. *ACM Trans. Graph.*, 35(4):85:1–85:11, 2016.

[45] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[46] `https://imagej.net/Welcome`, 2022. Accessed on: 2022-05-27.

[47] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. Verified lifting of stencil computations. *SIGPLAN Not.*, 51(6):711–726, June 2016.

[48] Shoaib Kamil, Derrick Coetzee, Scott Beamer, Henry Cook, Ekaterina Gonina, Jonathan Harper, Jeffrey Morlan, and Armando Fox. Portable parallel performance from sequential, productive, embedded domain-specific languages. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 303–304, New York, NY, USA, 2012. ACM.

[49] Alfons Kemper, Thomas Neumann, Florian Funke, Viktor Leis, and Henrik Mühe. Hyper: Adapting columnar main-memory data management for transactional AND query processing. *IEEE Data Eng. Bull.*, 35(1):46–51, 2012.

[50] Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. Building efficient query engines in a high-level language. *Proc. VLDB Endow.*, 7(10):853–864, June 2014.

[51] David Ryan Koes and Seth Copen Goldstein. Near-optimal instruction selection on dags. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '08, page 45–54, New York, NY, USA, 2008. Association for Computing Machinery.

[52] 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, page 75, USA, 2004. IEEE Computer Society.

[53] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, pages 348–370, Berlin, Heidelberg, 2010. Springer-Verlag.

[54] Bruno Cardoso Lopes and Rafael Auler. *Getting Started with LLVM Core Libraries*. Packt Publishing, Birmingham, UK, 2014.

[55] `https://github.com/thisMagpie/Analysis`, 2022. Accessed on: 2022-05-27.

[56] Saeed Maleki, Yaoqing Gao, Maria J. Garzar´n, Tommy Wong, and David A. Padua. An evaluation of vectorizing compilers. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 372–382, 2011.

[57] Henry Massalin. Superoptimizer: A look at the smallest program. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems*, ASPLOS II, pages 122–126, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[58] John Matthews, J. Strother Moore, Sandip Ray, and Daron Vroon. Verification condition generation via theorem proving. In *Proceedings of the 13th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'06, pages 362–376, Berlin, Heidelberg, 2006. Springer-Verlag.

[59] Charith Mendis, Jeffrey Bosboom, Kevin Wu, Shoaib Kamil, Jonathan Ragan-Kelley, Sylvain Paris, Qin Zhao, and Saman Amarasinghe. Helium: Lifting high-performance stencil kernels from stripped x86 binaries to halide dsl code. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 391–402, New York, NY, USA, 2015. ACM.

[60] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. Automatically scheduling halide image processing pipelines. *ACM Trans. Graph.*, 35(4):83:1–83:11, 2016.

[61] Cedric Nugteren and Henk Corporaal. Introducing 'bones': A parallelizing source-to-source compiler based on algorithmic skeletons. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, GPGPU-5, pages 1–10, New York, NY, USA, 2012. ACM.

[62] Shoumik Palkar, James J. Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, and Matei Zaharia. Weld: A common runtime for high performance data analytics. *CIDR 2017 - 8th Biennial Conference on Innovative Data Systems Research*, January 2017.

[63] Spiros Papadimitriou and Jimeng Sun. Disco: Distributed co-clustering with map-reduce: A case study towards petabyte-scale end-to-end mining. In *Proceedings of the 2008 Eighth IEEE International Conference on Data Mining*, ICDM '08, pages 512–521, Washington, DC, USA, 2008. IEEE Computer Society.

[64] Phitchaya Mangpo Phothilimthana, Archibald Samuel Elliott, An Wang, Abhinav Jangda, Bastian Hagedorn, Henrik Barthels, Samuel J. Kaufman, Vinod Grover, Emina Torlak, and Rastislav Bodik. Swizzle inventor: Data movement synthesis for gpu kernels. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 65–78, New York, NY, USA, 2019. Association for Computing Machinery.

[65] 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, pages 297–310, New York, NY, USA, 2016. ACM.

[66] `http://www.cs.cornell.edu/Projects/polyglot/`, 2022. Accessed on: 2022-05-27.

[67] Thomas Porter and Tom Duff. Compositing digital images. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '84, pages 253–259, New York, NY, USA, 1984. ACM.

[68] William K. Pratt. *Digital Image Processing: PIKS Scientific Inside*. Wiley-Interscience, USA, 2007.

[69] Cosmin Radoi, Stephen J. Fink, Rodric Rabbah, and Manu Sridharan. Translating imperative code to mapreduce. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 909–927, New York, NY, USA, 2014. ACM.

[70] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph.*, 31(4):32:1–32:12, July 2012.

[71] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman P. Amarasinghe, and Frédo Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph.*, 31(4):32:1–32:12, 2012.

[72] Jonathan Ragan-Kelley, Andrew Adams, Dillon Sharlet, Connelly Barnes, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Halide: Decoupling algorithms from schedules for high-performance image processing. *Commun. ACM*, 61(1):106–115, December 2017.

[73] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *PLDI*, pages 519–530, Seattle, WA, USA, 2013. ACM.

[74] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society.

[75] Veselin Raychev, Madanlal Musuvathi, and Todd Mytkowicz. Parallelizing user-defined aggregations using symbolic execution. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 153–167, New York, NY, USA, 2015. ACM.

[76] Alexander J. Root. Optimizing vector instruction selection for digital signal processing. Master's thesis, Massachusetts Institute of Technology, 2022.

[77] Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Gratian Lup, Jubi Taneja, and John Regehr. Souper: A synthesizing superoptimizer, 2018.

[78] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic optimization of floating-point programs with tunable precision. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 53–64, New York, NY, USA, 2014. ACM.

[79] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic program optimization. *Commun. ACM*, 59(2):114–122, January 2016.

[80] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, SIGMOD '79, pages 23–34, New York, NY, USA, 1979. ACM.

[81] `https://people.csail.mit.edu/asolar/`, 2022. Accessed on: 2022-05-27.

[82] Yannis Smaragdakis and George Balatsouras. Pointer analysis. *Found. Trends Program. Lang.*, 2(1):1–69, April 2015.

[83] Calvin Smith and Aws Albarghouthi. Mapreduce program synthesis. *SIGPLAN Not.*, 51(6):326–340, June 2016.

[84] Armando Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, MIT, Berkeley, CA, USA, 2008.

[85] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 404–415, New York, NY, USA, 2006. ACM.

[86] `https://github.com/apache/spark/tree/master/examples/src/main/scala/org/apache/spark/examples`, 2018. Accessed on: 2018-01-20.

[87] Emina Torlak and Rastislav Bodik. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, Onward! 2013, page 135–152, New York, NY, USA, 2013. Association for Computing Machinery.

[88] Alexa VanHattum, Rachit Nigam, Vincent T. Lee, James Bornholt, and Adrian Sampson. Vectorization for digital signal processors via equality saturation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 874–886, New York, NY, USA, 2021. Association for Computing Machinery.

[89] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. Interactive query synthesis from input-output examples. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1631–1634, New York, NY, USA, 2017. ACM.

[90] Kaiyuan Wang, Allison Sullivan, Manos Koukoutos, Darko Marinov, and Sarfraz Khurshid. Systematic generation of non-equivalent expressions for relational algebra. In *ABZ*, volume 10817 of *Lecture Notes in Computer Science*, pages 105–120. Springer, 2018.

[91] Kyle Wiggers. Qualcomm hexagon 685 dsp is a boon for machine learning, 2017.

[92] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, MA, USA, 1993.

[93] Yuting Yang, Sam Prestwood, and Connelly Barnes. Vizgen: accelerating visual computing prototypes in dynamic languages. *ACM Trans. Graph. (TOG)*, 35(6):206:1–206:13, 2016.

Appendix A

CASPER

A.1 Proof Sketch For Soundness and Completeness

Here, we first formalize the definitions of soundness and completeness, and then we present a proof sketch to show that CASPER's synthesis algorithm for program summaries has these properties. We use terms and acronyms defined in Chapter 3 without explaining them again here.

Definition 1. *(Soundness of Search)* An algorithm for generating program summaries is sound if and only if, for all program summary ps and loop invariants $inv_1, \ldots, inv_n$ generated by the algorithm, the verification conditions hold over all possible program states after we execute the input code fragment p. In other words, $\forall \sigma.\ vc(p, ps, inv_1, \ldots, inv_n, \sigma)$.

Definition 2. *(Completeness of Search)* An algorithm for generating program summaries is complete if and only if when there exists $ps, inv_1, \ldots, inv_n \in G$, then $\forall \sigma.\ vc(p, ps, inv_1, \ldots, inv_n, \sigma)) \rightarrow (\Delta \neq \emptyset)$. Here, G is the search space traversed, p is the input code fragment, vc is the set of verification conditions, and Δ is the set of sound summaries found by the algorithm. In other words, the algorithm will never fail to find a correct program summary as long as one exists in the search space.

Proof of Soundness. The soundness guarantee for CASPER's synthesis algorithm is derived from the soundness guarantees offered by Hoare-style verification conditions. The proof is constructed using a *loop-invariant*, namely, a statement that is true immediately before and after each loop execution. Hoare logic dictates that in order to prove correctness of a

given postcondition (i.e., program summary) for a given loop, we must prove the following holds over all possible program states:

1. The invariant is true before the loop.

2. Each iteration of the loop maintains the invariant.

3. Once the loop has terminated, the invariant implies the postcondition.

This is essentially an inductive proof. The first two constraints prevent CASPER from finding a loop invariant strong enough to imply an incorrect program summary. Our correctness guarantee is, of course, subject to the correct implementation of our VC generation module and of the theorem prover we use (Dafny). Establishing that the summary is a correct postcondition is sufficient to establish that it is a correct translation. This is so because summaries in our IR must describe the final value of *all* output variables (i.e., variables that were modified) as a function over the inputs (see Figure 3.3).

Proof of Completeness. To understand that CASPER's algorithm is complete with respect to the search space, we first show that that the algorithm always terminates. Recall that we use recursive bounds to finitize the number of solutions expressible by our IR's grammar. As explained in §3.4.1, we prevent the same solution from being regenerated, thus ensuring forward progress in search. These two facts imply that our algorithm always terminates. It is important to note that our search algorithm is complete only for *verifiably correct* summaries. If a correct summary exists in the search space but cannot be proven correct using the available automated theorem prover, it will not be returned. Therefore, the completeness of the algorithm is modulo the completeness of the theorem prover.

A.2 Intermediate Representation Specification

Here, we list the full set of types available in our IR and provide examples to demonstrate how they may be used to express models for library methods and types.

Primitive Data Types	
Scalars	`bool`, `int`, `float`, `string`, `char`, ...
Structures	`class(id:Type, id2:Type2, ..)`
List	`list(Type)`
Array	`array(dimensions, Type)`
Functions	`name(arg1:Type1, ...) : Type -> Body`
Conditionals	`if` *cond* `then` e_1 `else` e_2
Synthesis Construct	`choose(`e_1`,` e_2`, ... ,` e_n`)`

Built-in operations	
Arithmetic	$+$, $-$, $*$, $/$, $\%$, ...
Bitwise	$<<$, $>>$, $\&$, ...
Relational	$<$, $>$, $\leq$, $\geq$, ...
Logical	$\&\&$, $\|\|$, $==$, $!=$
List	`len, append, get, equals, concat, slice`
Array	`select, store`

To provide support for a datatype found in a Library, users must define the type of the object using our IR and annotate it with the fully qualified name, as follows:

```
@java.awt.Point
class Point(x:int, y:int)
```

Similarly, users may also provide support for library methods, for instance the following defines a model for the absolute value function:

```
@java.lang.Math.abs
abs(val: int) : int ->
  if val < 0 then val * -1 else val
```

Using the core IR described above, we implemented in CASPER the *map*, *reduce* and *join* primitives used to synthesize summaries. We have also implemented commonly used methods from Java standard libraries such as `java.util.Math,String,Date` and other essential data-types, along with methods that were needed to translate the Fiji plugins.

The *choose* operator in the IR is a special construct that enables us to express a search space using the IR. The parameters to *choose* are one or more expressions of matching types. The synthesizer is then free to select any expression from the list of choices in order to satisfy the correctness specification.

A.3 Code Generation Rules

To generate target DSL code from the synthesized program summary, we implemented in CASPER a set of translation rules that map the operators in our IR to the concrete syntax of the target DSL. Here, we list a subset of such code-generation rules for the Spark RDD API.

$$
\begin{aligned}
TR\texttt{Gen}(\textbf{map}(l,\ \lambda_m : T \rightarrow list(Pair))) &= \texttt{l.flatMapToPair(Gen}(\lambda_m)\texttt{);} \\
TR\texttt{Gen}(\textbf{map}(l,\ \lambda_m : T \rightarrow list(U))) &= \texttt{l.flatMap(Gen}(\lambda_m)\texttt{);} \\
TR\texttt{Gen}(\textbf{map}(l,\ \lambda_m : T \rightarrow Pair)) &= \texttt{l.mapToPair(Gen}(\lambda_m)\texttt{);} \\
TR\texttt{Gen}(\textbf{map}(l,\ \lambda_m : T \rightarrow U)) &= \texttt{l.map(Gen}(\lambda_m)\texttt{);} \\
TR\texttt{Gen}(\textbf{reduce}(l : list(Pair),\ \lambda_r)) &= \texttt{l.reduceByKey(Gen}(\lambda_r)\texttt{);} \\
TR\texttt{Gen}(\textbf{reduce}(l : list(U),\ \lambda_r)) &= \texttt{l.reduce(Gen}(\lambda_r)\texttt{);}
\end{aligned}
$$

$$
\begin{aligned}
TR\texttt{Gen}(\lambda_m(e) \rightarrow e_b) &= \texttt{(e -> Gen}(e_b)\texttt{)} \\
TR\texttt{Gen}(e_1 + e_2) &= \texttt{Gen}(e_1)\ \texttt{+}\ \texttt{Gen}(e_2)
\end{aligned}
$$

The translation function TR takes as input an expression in our IR language and maps it to an equivalent expression in Spark. Since Spark provides multiple variations for the operators defined in our IR, such as *map*, we can select the appropriate variation by looking at the type information of the λ_m function used by *map*. For example, if λ_m returns a list of Pairs, we translate to `JavaRDD.flatMapToPair`. If it instead returns a list of a non-Pair type, we use the more general rule that translates *map* to `JavaRDD.flatMap`. Translation for the other expressions proceeds similarly.

A.4 Program Analyzer Outputs

Here, we use TPC-H Query 6 to illustrate the outputs computed by Casper's program analyzer. Since the queries are originally in SQL, we have manually translated them to Java as follows:

```
1  double query6(List<LineItem> lineitem){
2    Date dt1 = Util.df.parse("1993-01-01");
3    Date dt2 = Util.df.parse("1994-01-01");
4    double revenue = 0;
5    for (LineItem l : lineitem) {
6      if (
7        l.l_shipdate.after(dt1) &&
8        l.l_shipdate.before(dt2) &&
9        l.l_discount >= 0.05 &&
10       l.l_discount <= 0.07 &&
11       l.l_quantity < 24
12     )
13       revenue += (l.l_extendedprice * l.l_discount);
14   }
15   return revenue;
16 }
```

First, CASPER's program analyzer normalizes the loop starting on Line 6 into an equivalent `while(true){..}` loop, and then traverses the loop to identify the set of input/output variables and operators used:

Program Analysis Results	
Inputs Vars	`l: list(LineItem), dt1: Date, dt2: Date`
Output Vars	`revenue: double`
Constants	`[(24, int), (0.05, double), (0.07, double)]`
Operators	$+, -, *, \geq, \leq, <$
Methods	`Date.before, Date.after`

With this information, CASPER generates verification conditions like those shown in Figure 3.4b for the row-wise mean benchmark. Next, the program analyzer defines a search space within which CASPER searches for summaries and the needed loop-invariant. Since the full search space description is too large to show, we only show a small snippet below:

```
1  generator doubleExpr(val:LineItem, depth:int) : double ->
2    if depth = 0 then
3      choose(val.l_quantity, val.l_extendedprice, val.l_discount, 0.05, 0.07, 24)
4    else
5      choose(
6        doubleExpr(val, 0),
7        doubleExpr(val, depth-1) + doubleExpr(val, depth-1),
8        doubleExpr(val, depth-1) * doubleExpr(val, depth-1),
9        doubleExpr(val, depth-1) / doubleExpr(val, depth-1)
10     )
```

The `doubleExpr` is the part of the grammar used to construct expressions that evaluate to *double*. The `generator` keyword indicates that this is a special type of function, one that can select a different value from the `choose` operators on each invocation. The depth parameter

controls how large the generated expression is allowed to grow. The `choose` construct is used to present a set of possible productions to the synthesizer. This grammar is tailored specifically to our implementation of TPC-H Query 6.

A.5 Supplementary Experiments

A.5.1 Benchmark Details

The benchmarks Casper extracted form a diverse and challenging problem set. As shown in the table below, they vary across programming style as well as the structure of their solutions.

Benchmark Properties	**# Extracted**	**# Translated**
Conditionals	26	19
User Defined Types	14	10
Nested Loops	40	22
Multiple Datasets	22	18
Multidim. Dataset	38	23

A.5.2 Developer Selection Criteria

To get reference Spark implementations for non-SQL benchmarks, we hired developers through the online freelancing platform UpWork.com. While hiring, we ensured all candidates met the following basic criteria:

- At least an undergraduate or equivalent degree in computer science.
- Minimum 500 hours of work logged at the platform.
- Minimum 4 star rating for previous projects (scale of 5).

- A portfolio of at least one or more successfully completed contracts using Spark.

Finally, applicants were required to answer three test questions regarding Spark API internals to bid on our contract.

A.5.3 Evaluating Cost Model Heuristics

We present here some experiments that measure whether CASPER's cost model model can effectively identify efficient solutions during the search process.

Program	**Emitted (MB)**	**Shuffled (MB)**	**Runtime (s)**
WordCount 1	105k	30	254
WordCount 2	105k	58k	2627
StringMatch 1	16	0.7	189
StringMatch 2	90k	0.7	362

Table A.1: The correlation of data shuffle and execution.

As discussed in §3.5.1, CASPER uses a data-centric cost model. The cost model is based on the hypothesis that the amount of data generated and shuffled during the execution of a MapReduce program determines how fast the program executes. For our first experiment, we measured the correlation between the amount of data shuffled and the runtime of a benchmark to check the validity of the hypothesis. To do so, we compared the performance of two different Spark WordCount implementations: one that aggregates data locally before shuffling (WordCount 1) using combiners [35], and one that does not (WordCount 2). Although both implementations processed the same amount of input data, the former implementation significantly outperformed the latter, as the latter incurred the expensive overhead of moving data across the network to the nodes responsible for processing it. Table A.1 shows the amount of data shuffled along with the corresponding runtimes for both implementations

using the 75GB dataset. As shown, the implementation that used combiners to reduce data shuffling was almost an order of magnitude faster.

Next, we verified the second part of our hypotheses by measuring the correlation of the amount of data generated and the runtime of a benchmark. To do so, we compared two solutions for the StringMatch benchmark (sequential code shown in Figure 3.7a). The benchmark determines whether certain keywords exist in a large body of text. Both solutions use combiners to locally aggregate data before shuffling. However, one solution emits a key-value pair only when a matching word is found (StringMatch 1), whereas the other always emits either `(key, true)` or `(key, false)` (StringMatch 2). Since the data is locally aggregated, each node in the cluster only generates 2 records for shuffling (one for each keyword) regardless of how many records were emitted during the map phase. As shown in Table A.1, the implementation that minimized the amount of data emitted in the map-phase executed almost twice as fast.

In sum, the two experiments confirm that the heuristics used in our cost model are accurate indicators of runtime performance for MapReduce applications. We also demonstrated the need for a data-centric cost model; solutions that minimize data costs execute significantly faster than those that do not.

A.5.4 Evaluating Scalability of Generated Implementations

To observe how implementations generated by Casper scale, we executed our benchmarks on different amounts of data and measured the resulting speedups. As shown in Figure A.1, the Casper-generated Spark implementations exhibited good data parallelism and showed a steady increase in speedups across all translated benchmarks as the input data size increased, until the cluster reached maximum utilization.

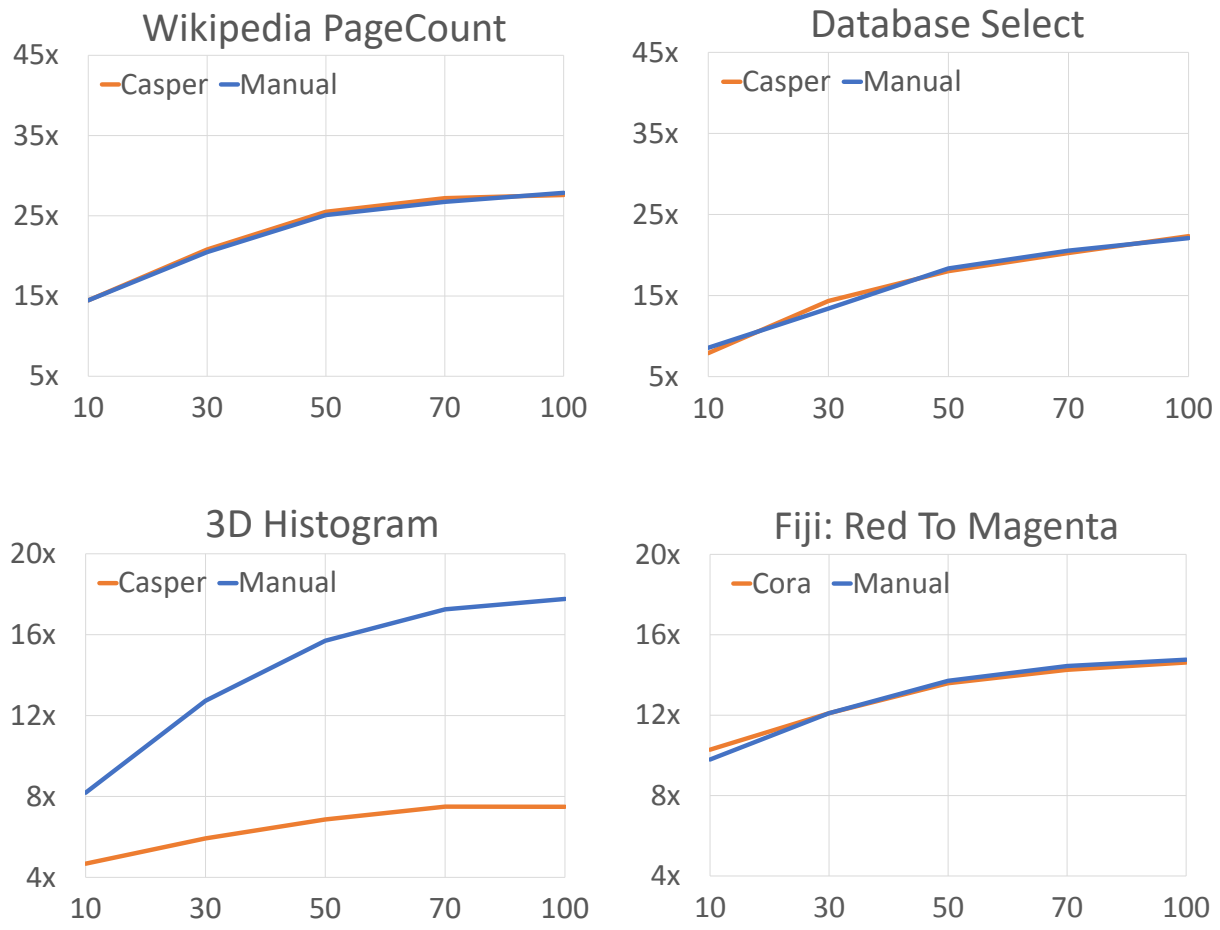


Figure A.1: The top 2 benchmarks with the most performance along with the bottom 2. The x-axis plots the size of input data, while the y-axis plots the runtime speedup over sequential implementations.

Appendix B

DEXTER

B.1 Symmetry Elimination and Memoization

The space of possible expressions encoded by the grammar in Figure 4.4c contains a large amount of symmetry: it can generate syntactically different expressions that are semantically equivalent due to the presence of commutative and associative operations (for instance, $a*(b+c) \equiv a*b+a*c$). Furthermore, larger functions frequently contain recurring sub-expressions, especially across different branches of control flow. A naive search over this grammar would consider far too many redundant expressions with the same semantics. Therefore, DEXTER, inspired by prior work in relational algebra [90], uses a bottom-up expression generator that prunes away redundant expressions, while memoizing already generated sub-expressions for reuse.

The expression generator maintains a list of expressions, initially instantiated with the set of available terminals. To construct new expressions, the generator first chooses an operator from the set of operators available in the grammar. Next, it chooses operands for the operator from the list of expressions to construct a new expression. Finally, the generator checks whether this newly generated expression is the correct expression for the summary by invoking the solver. If so, the algorithm terminates and the expression is returned. If it does not verify, the expression is memoized by appending it to the end of the list and the process is repeated until the correct expression is found.

We illustrate our algorithm with an example. Suppose our goal is to generate the following expression: $(msk(i) == 1\ ?\ src_1(i) * src_2(i)\ :\ src_1(i) * src_2(i) * msk(i))$. In this expression, src_1 and src_2 represent two layers that we want to blend, and msk is the blend mask. Figure B.1 shows four possible traces (decision paths) of our algorithm for producing

	Input Terminals				Step 1	Step 2	Step 3	Step 4	Step 5
Trace 1	msk(i)	src1(i)	src2(i)	1	1 == msk(i)	src1(i) * src2(i)	src1(i) * src2(i) * msk(i)	(1==msk(i) ? src1(i) * src2(i) : src1(i) * src2(i) * msk(i))	-
Trace 2	msk(i)	src1(i)	src2(i)	1	msk(i) == 1	src1(i) * src2(i)	src1(i) * src2(i) * msk(i)	(msk(i)==1 ? src1(i) * src2(i) : src1(i) * src2(i) * msk(i))	-
Trace 3	msk(i)	src1(i)	src2(i)	1	msk(i) == 1	src1(i) * src2(i)	msk(i) * src1(i) * src2(i)	(msk(i)==1 ? src1(i) * src2(i) : msk(i) * src1(i) * src2(i))	-
Trace 4	msk(i)	src1(i)	src2(i)	1	msk(i) == 1	src1(i) * src2(i)	src1(i) * src2(i)	msk(i) * src1(i) * src2(i)	(msk(i)==1 ? src1(i) * src2(i) : msk(i) * src1(i) * src2(i))

Figure B.1: Four possible traces of DEXTER's expression generation algorithm. The steps in green indicate successful termination of the algorithm, whereas the steps in red indicate a pre-emptive rejection of the trace due to violation of symmetry breaking rules.

semantically equivalent expressions. Not all traces are viable as they contain steps violating our symmetry breaking rules (marked in red), which we explain later. To demonstrate our algorithm, we walk through trace 3, which successfully generates this expression in four steps. In step 1, the algorithm combines terminals $msk(i)$ and 1 using the equality operator to construct a boolean expression, but finds that the generated expression is not the desired expression. In step 2, it combines $src_1(i)$ and $src_2(i)$ using multiplication. In the third step, the expression generated in step 2 ($src_1(i) * src_2(i)$) is combined with the terminal $msk(i)$ using multiplication to get the alternate expression. The fourth and final step uses these three generated expressions as operands to the ternary operator to construct the desired output expression.

There are several benefits of DEXTER's memoization approach. First, it maintains a total ordering over the generated expressions, based on the list index at which they are stored. This is useful for eliminating symmetries in the search space: for commutative and associative binary operators, the expression generator only allows binary expressions e_1 *op* e_2 where e_1 is stored at a lower index in the expressions array than e_2, and likewise for the test, consequent, and alternate expressions used in a conditional. To see the benefit, consider traces 1, 2, and 3 in Figure B.1. All three traces generate semantically equivalent expressions, yet since they are syntactically different, the synthesizer would enumerate all three in its search. With our order-based pruning constraints, however, both trace 1 and trace 2 would be rejected as they violate the constraints at step 1 and step 3 respectively. Furthermore, since the generated sub-expressions are stored in the list, they can be reused subsequently. This reduces the

number of steps the synthesizer must take to generate expressions that contain recurring sub-expressions. This is illustrated in trace 4 in Figure B.1. This trace also generates the correct expression, but since it does not reuse the sub-expression $src_1(i) * src_2(i)$, the algorithm requires an extra step to build the expression compared to trace 3.

B.2 Analysis Based Suggestions

The optimizations discussed so far are not particular to a specific input kernel. DEXTER also analyzes the input code to populate the starting list of expressions with input-specific recommendations to the synthesizer. For example, in the blur kernel, static analysis can extract that the value being written into the `dst` array is `(tmp[c-rowBytes] + tmp[c] + tmp[c+rowBytes]) / 3`. By substituting our synthesized terminal mappings, we can get the equivalent IR expression: $(tmp(x, y-1) + tmp(x, y) + tmp(x, y+1))$ / 3. DEXTER therefore adds this expression as one of the initial expressions to consider during synthesis. If the suggestion is correct, or is a sub-expression of the correct expression, the synthesizer can construct the result in fewer steps. If the suggestion is incorrect, the synthesizer can simply ignore it and construct the correct expression using the set of terminals. The overhead of providing these recommendations is minimal and the benefits of a correct recommendation are significant (over 100x faster synthesis).