IMPROVING PERFORMANCE OF DATA-CENTRIC SYSTEMS THROUGH FINE-GRAINED CODE GENERATION

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Grégory Essertel

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 2019

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF DISSERTATION APPROVAL

Dr. Tiark Rompf, Chair

Department of Computer Science

Dr. Ananth Grama

Department of Computer Science

Dr. Suresh Jagannathan

Department of Computer Science

Dr. Walid Aref

Department of Computer Science

Approved by:

Dr. Chris Clifton

Head of the Departmental Graduate Program

À mes parents, Colette et Michel.

À mes grands-parents, Augusta, Auguste, Marie-Thérèse et Raymond.

ACKNOWLEDGMENTS

I would like to express my gratitude to my advisor Dr. Tiark Rompf for his constant guidance. His advice and insights helped me to learn a lot, grow, and become a better scholar.

I also want to thank the founding team of the CTF club, Nathan Burow, Scott Carr, Daniele Midi, and Craig West, as well as Dr. Mathias Payer. Our weekends were always fun and allowed me to discover the field of security.

My coworkers and friends, James Decker, Ruby Tahboub, Fei Wang, and Guannan Wei have always been there to discuss, debate, and argue about almost anything.

Last, but certainly not least, words cannot express my gratitude to my family that was back in France rooting for me: my parents Colette and Michel, my grandma Augusta, Baptiste and Elise, Aurélie and Jérôme, Sylvain and Cindy. This extends to all my friends, Alexandre, Flavie and Julien, and la Stimuliste.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Essertel, Grégory PhD, Purdue University, December 2019. IMPROVING PERFORMANCE OF DATA-CENTRIC SYSTEMS THROUGH FINE-GRAINED CODE GENERATION . Major Professor: Tiark Rompf.

The availability of modern hardware with large amounts of memory created a shift in the development of data-centric software; from optimizing I/O operations to optimizing computation. As a result, the main challenge has become using the memory hierarchy (cache, RAM, distributed, etc) efficiently. In order to overcome this difficulty, programmers of data-centric programs need to use low-level APIs such as Pthreads or MPI to manually optimize their software because of the intrinsic difficulties and the low productivity of these APIs. Data-centric systems such as Apache Spark are becoming more and more popular. These kinds of systems offer a much simpler interface and allow programmers and scientists to write in a few lines what would have been thousands of lines of low-level MPI code. The core benefit of these systems comes from the introduction of deferred APIs; the code written by the programmer is actually building a graph representation of the computation that has to be executed. This graph can then be optimized and compiled to achieve higher performance.

In this dissertation, we analyze the limitations of current data-centric systems such as Apache Spark, on relational and heterogeneous workloads interacting with machine learning frameworks. We show that the compilation of queries in multiples stages and the interfacing with external systems is a key impediment to performance because of their inability to optimize across code boundaries. We present Flare, an accelerator for data-centric software, which provides performance comparable to the state of the art relational systems while keeping the expressiveness of high-level deferred

APIs. Flare displays order of magnitude speed up on programs combining relational processing and machine learning frameworks such as TensorFlow. We look at the impact of compilation on short-running jobs and propose an on-stack-replacement mechanism for generative programming to decrease the overhead introduced by the compilation step. We show that this mechanism can also be used in a more generic way within source-to-source compilers. We develop a new kind of static analysis that allows the reverse engineering of legacy codes in order to optimize them with Flare. The novelty of the analysis is also useful for more generic problems such as formal verification of programs using dynamic allocation. We have implemented a prototype that successfully verifies programs within the SV-COMP benchmark suite.

1 INTRODUCTION

1.1 Problem statement

The past decades have seen a significant increase in the amount of memory available for computers. It is possible to buy machines with multiple TB and even 10s of TB of RAM. This development has transformed the way data-centric software is designed. These large main memory machines have eliminated the bottlenecks of I/O to and from secondary storage, and have made computational bottlenecks the main impediment to performance. Therefore using memory efficiently becomes more important; naturally, optimal use of the memory hierarchy (cache, RAM, distributed, etc.) becomes the primary strategy for obtaining high performance.

The current industry standard for implementing efficient parallel and distributed software is to use low-level APIs such as Pthreads, OpenMP, MPI, network primitives, etc. This requires the programmer to optimize the applications manually. However, low-level data-centric programs are difficult to optimize due to their high complexity. The challenge of programming these systems is complicated further by the necessity to use multiple APIs together to achieve good performance. The research community has proposed alternatives to these tedious low-level APIs, for example with performance-oriented Domain Specific Languages (DSLs) [144]. However, introducing new user-facing abstractions always faces the hurdle of adoption. On the other end of the spectrum, systems like Spark optimize for convenience rather than for raw performance and have been widely adopted across different fields.

In its initial version, Spark was nothing more than a distributed collections library, and thus orders of magnitude slower than optimized MPI code. However, developers could express in just a few lines of code what would take hundreds or even thousands of lines using MPI. More recently, new classes of systems such as the DataFrame

subsystem in Apache Spark (Spark SQL [12]) and Tensorflow [4] have emerged, and these recover some of the runtime performances while retaining productivity benefits. These systems, inspired by databases and query plans, build a graph representing the computation that needs to be performed rather than directly performing said computation. Once built, the graph is optimized, and the system generates efficient code that removes interpretative overhead. This pattern is often called *deferred* execution (as opposed to standard, *immediate*, execution), and APIs such as Sparks DataFrame library are referred to as deferred APIs. While many advantages came from this design, there still exists room for improvement:

- While main memory database compilation techniques have been applied to Spark, however, the results are not optimal. For example, the DataFrame computation graph is compiled into multiple components, resulting in communication overhead at the code boundaries. Also, fault-tolerant infrastructure and other such mechanisms add additional overhead. While this can be necessary in a large-scale distributed setting, it is not within a single machine or even small clusters of less than 100 nodes where failures are statistically unlikely.

- Relational processing is not enough for realistic workloads, which often need to combine SQL-style relational processing with procedural code and increasingly training loops of inference of machine learning models, sometimes provided through external frameworks and applications. Spark SQL provides the possibility to use other systems through User Defined Functions (UDFs), however, the computation is a black box to the DataFrame optimizer. Thus, each system is optimized individually; but, when used together, there is no cross-optimization. This results in unnecessary overheads at the boundaries between systems.

- Aggressive compilation does not come for free. For long-running queries, the performance benefit is clear, but the extra compilation time can be a significant overhead for queries with very short running time, which may still be complex

and require substantial optimization and/or dominate the running time of a system because queries are executed many times.

- New platforms are rarely backward-compatible, resulting in large amounts of legacy code which are unable to be utilized on these new systems. In a similar way, arbitrary UDFs are often difficult to integrate, mainly because they are not sharing the programming paradigm.

In this dissertation, we analyze and propose a solution to each of these problems. In order to make our work more easily adopted, we focus on developing accelerators for existing systems rather than proposing a different paradigm or novel user-facing DSL. We seek to improve the performance of systems by generalizing the notion of deferred APIs and generating low-level code that is highly optimized for a given platform. We demonstrate, that in order to achieve this, we need to use deferred API and code generation at different levels of abstraction. For example, Spark SQL query plans contain nodes representing SQL operators, which are then directly transformed to code. We show that pushing down the code generation granularity to the level of primitive operations allows an efficient generation of multi-threaded, NUMA aware, and even distributed code. This also enables cross-optimization with other systems that use the same design, addressing the challenge of interoperability.

While we show that adding an extra compilation step improves query execution time, situations in which the execution time and compilation time are similar do not gain such a benefit. Therefore, we propose a solution to reduce compilation overhead through tiered compilation and on-stack-replacement (OSR) techniques for code generators. We illustrate that OSR has a broader application space in the context of code generation. For example, this can be used to generate code with speculative optimizations. In addition, the capacity to swap code at runtime is an interesting avenue to implement fault tolerance mechanisms within generated code.

Finally, we investigate the possibility of applying the different optimizations that deferred APIs offer to generic user-defined functions (UDFs) and more broadly to

any low-level legacy code. This is achieved by extracting the meaning of programs in terms of computation graphs. However, obtaining this kind of information from a low-level imperative code is not a simple task. We develop a new static analysis method that extracts functional semantics from low-level imperative code. The theory we have established is more general than our current context, for example, it can also be used for formal verification.

1.2 Overview

1.2.1 Flare: Accelerating Data Processing

In recent years, Apache Spark has become the de facto standard for big data processing. Spark has enabled a wide audience of users to process petabyte-scale workloads due to its flexibility and ease of use: users are able to mix SQL-style relational queries with Scala or Python code, and have the resultant programs distributed across an entire cluster, all without having to work with low-level parallelization or network primitives.

However, many workloads of practical importance are not large enough to justify distributed, scale-out execution, as the data may reside entirely in main memory of a single powerful server. Although Spark adopted the state of the art query compilation model, we show that it has to be adapted on order to handle these jobs efficiently. Still, users want to use Spark for its familiar interface and tooling. In such scale-up scenarios, Spark's performance is suboptimal, as Spark prioritizes handling *data size* over optimizing the *computations* on that data. For such medium-size workloads, performance may still be of critical importance if jobs are computationally heavy or need to be run frequently on changing data.

In Chapter 2, we present Flare, an accelerator module for Spark that delivers order of magnitude speedups on scale-up architectures for a large class of applications.

1.2.2 Combining Relational Processing and Machine Learning Frameworks

Realistic contemporary workloads are not purely relational, but for example combine a loading and preprocessing phase (ETL) to feed machine learning pipelines. Inspired by query compilation techniques from main-memory database systems, Flare incorporates a code generation strategy designed to match the unique aspects of Spark and the characteristics of scale-up architectures; in particular, processing data directly from optimized file formats and combining SQL-style relational processing with external frameworks such as TensorFlow. Running machine learning (ML) workloads at scale is as much a data management problem as a model engineering problem. Big performance challenges exist when data management systems invoke ML classifiers as user-defined functions (UDFs) or when stand-alone ML frameworks interact with data stores for data loading and pre-processing. In particular, UDFs can be precompiled or simply a black box for the data management system and the data layout may be completely different from the native layout, thus adding overheads at the boundaries.

In Chapter 3, we show how bottlenecks between existing systems can be eliminated when their engines are designed around runtime compilation and native code generation, which is the case for many state-of-the-art relational engines as well as ML frameworks. We demonstrate an integration of Flare (an accelerator for Spark SQL), and Lantern (an accelerator for TensorFlow and PyTorch) that results in a highly optimized end-to-end compiled data path, switching between SQL and ML processing with negligible overhead.

1.2.3 On Stack Replacement for Program Generators And Source-to-Source Compilers

Aggressive compilation has a cost that must be amortized. For long-running queries the performance are greatly improved. However, in the situation of complex but short-running time queries, the gains are reduced due to the compilation time

overhead. Through on-stack-replacement technique, we reduce this problem with a tiered execution strategy. On-stack-replacement, in addition to solving the compilation time issue, offers a more generic set of techniques that can be used within code generators.

On-stack replacement (OSR) describes the ability to replace currently executing code with a different version, either a more optimized one (tiered execution) or a more general one (deoptimization to undo speculative optimization). While OSR is a key component in all modern VMs for languages like Java or JavaScript, OSR has only recently been studied as a more abstract program transformation, independent of language VMs. Still, previous work has only considered OSR in the context of low-level execution models based on stack frames, labels, and jumps.

With the goal of making OSR more broadly applicable, in Chapter 4 we present a surprisingly simple pattern for implementing OSR in source-to-source compilers or explicit program generators that target languages with structured control flow (loops and conditionals). We evaluate our approach through experiments demonstrating both tiered execution and speculative optimization, based on representative code patterns in the context of a state-of-the-art in-memory database systems that compile SQL queries to C at runtime. We further show that casting OSR as a metaprogramming technique enables new speculative optimization patterns beyond what is commonly implemented in language VMs.

1.2.4 Precise Reasoning with Structured Time, Structured Heaps, and Collective Operations

While Flare can generate very efficient parallel code, the techniques we developed can only be applied on code bases that have been implemented using deferred APIs. We explore the possibilities to use Flare to accelerate some of the legacy code, or UDFs, that were originally implemented in an imperative single-threaded style. Our solution is to extract high-level functional semantics from the legacy code, and use

this representation to extract the essence of the code that can then be accelerated by Flare. The high-level functional representation has the advantage of being very generic, thus we use it to solve other problems, such as program verification.

Despite decades of progress, static analysis tools still have great difficulty dealing with programs that combine arithmetic, loops, dynamic memory allocation, and linked data structures. In this work, we draw attention to two fundamental reasons for this difficulty: First, typical underlying program abstractions are low-level and inherently *scalar*, characterizing compound entities like data structures or results computed through iteration only indirectly. Second, to ensure termination, analyses typically project away the dimension of time, and merge information per program point, which incurs a loss in precision.

As a remedy, we propose to make collective operations first-class in program analysis—inspired by Σ-notation in mathematics, and also by the success of high-level intermediate languages based on `map/reduce` operations in program generators and aggressive optimizing compilers for domain-specific languages (DSLs). We further propose a novel structured heap abstraction that preserves a symbolic dimension of time, reflecting the program's loop structure and thus unambiguously correlating multiple temporal points in the dynamic execution with a single point in the program text.

In Chapter 5, we present a formal model, based on a high-level intermediate analysis language, a practical realization in a prototype tool that analyzes C code, and an experimental evaluation that demonstrates competitive results on a series of benchmarks. Remarkably, our implementation achieves these results in a fully semantics-preserving strongest-postcondition model, which is a worst-case for analysis/verification. The underlying ideas, however, are not tied to this model and would equally apply in other settings, e.g., demand-driven invariant inference in a weakest-precondition model. Given its semantics-preserving nature, our implementation is not limited to analysis for verification, but can also check program equivalence, and translate legacy C code to high-performance DSLs, such as Spark DataFrame.

1.3 Hypothesis

Our thesis hypothesis states that *using fine-grained code generation techniques within the runtime of data-centric systems leads to an increase in performance and productivity.*

1.4 Contributions

The key contributions of our work are the following:

1. We present Flare, an accelerator for data-centric systems. (Chapter 2)

   (a) We identify key performance impediments, such as compiling the query plans in multiple stages, for workloads running on Spark in a shared memory environment, and present a novel code generation strategy able to overcome these impediments. (Section 2.4)

   (b) We present Flare's architecture and discuss key implementation choices. We show how Flare is capable of optimizing data loading, dealing with parallel execution, as well as efficiently working on NUMA systems. This is a result of Flare compiling whole queries as opposed to individual query stages, which results in an end-to-end optimized data path. We evaluate Flare in comparison to Spark on TPC-H, reducing the gap to the best-of-breed relational query engines. In this setting, Flare exhibits order-of-magnitude speedups. Our evaluation spans single-core, multi-core, and NUMA targets. (Section 2.5 and 2.6)

   (c) We show how the work on Flare on a single machine can be extended to a distributed setting while keeping the same architecture and abstraction level. Similarly, we show that the technique used to accelerate Spark can be applied to similar systems such as Flink with little effort. (Section 2.7 and 2.8)

2. We demonstrate the advantages of using compilation and Flare in conjunction with machine learning framework or generic user defined functions. We show that Flare can be used as an efficient database system for machine learning algorithms. (Chapter 3)

   (a) We show that on benchmarks involving external libraries, Flare exhibits order-of-magnitude speedups. (Section 3.6).

3. We analyze the overheads introduced by Flare compilation step, and develop a strategy based on OSR to reduce them. (Chapter 4)

   (a) We propose an extremely simple model of OSR. We believe that this simplicity will make the model useful for future study. In particular, our model provides a simpler correctness story than previous formal models. We do not need to represent OSR primitives in an IR and instead translate away the OSR behavior into a high-level, structured, AST-like program representation. (Section 4.2)

   (b) We show how program generators and source-to-source compilers can emit OSR patterns which enable them to profit from tiered execution and speculative optimization in addition to standard code specialization. (Section 4.3)

   (c) We demonstrate that we can add OSR non-intrusively to a program, without having a JIT setup. Compilation relies only on any of the available ahead-of-time compilers for the desired target language, and requires minimal library support. (Section 4.5 and 4.6)

4. We investigate the challenge of interfacing Flare's code generation with legacy code and external UDFs. We develop a technique to extract high-level functional semantics from low-level imperative code. (Chapter 5)

(a) We present a detailed formal semantics of a source and target language including a structured heap model. We prove the correctness of the translation and target-level simplification rules. The simplification rules we present give rise to a large space of rewriting opportunities that can be realized either deterministically bottom-up or nondeterministically through search. Each rule is guaranteed to be equality-preserving, which leads to a simple but overall *pessimistic* approach if rules are applied one-by-one after the translation step. (Section 5.3)

(b) We extend the pessimistic, equality-preserving, simplification model to an *optimistic* approach that interleaves translation and simplification, based on Kleene-iteration. This model further increases precision by enabling a form of speculation, e.g., assuming that parts of a data structure remain constant throughout a loop, that an arithmetic recurrence has a closed form, or that a write to a data structure is the initialization of a dense array. (Section 5.4)

(c) We present `SIGMA`, which scales up these ideas to be able to analyze C programs. We evaluate `SIGMA` on benchmarks for verification, program equivalence checking, and translation of legacy code to high-performance DSLs. (Section 5.5)

1.4.1 Publications

The work presented in this dissertation has been published in the following papers:

1. **Grégory Essertel**; Tahboub Ruby; Tiark Rompf. *Flare: Optimizing Apache Spark with Native Compilation for Scale-Up Architectures and Medium-Size Data*. OSDI 2018. Presented in Chapter 2, with additional unpublished sections: Section 2.7 and 2.8.

2. **Grégory Essertel**; Ruby Tahboub; Fei Wang; James Decker; Tiark Rompf. *Flare & Lantern: Efficiently Swapping Horses Midstream.* Demo track VLDB 2019. Presented in Chapter 3.

3. **Grégory Essertel**; Ruby Tahboub; Tiark Rompf. *On-Stack Replacement for Program Generators and Source-to-Source Compilers.* Tech report 2019. Presented in Chapter 4.

4. **Grégory Essertel**; Guannan Wei; Tiark Rompf. *Precise Reasoning with Structured Time, Structured Heaps, and Collective Operations.* OOPSLA 2019. Presented in Chapter 5.

The following works are not part of this dissertation, but are related to some of the chapters.

1. Tahboub Ruby; **Grégory Essertel**; Tiark Rompf. *How to Architect a Query Compiler, Revisited.* SIGMOD 2018. Related to Chapter 2.

2. Fei Wang; James Decker; Xilun Wu; **Grégory Essertel**; Tiark Rompf. *Demystifying Differentiable Programming: Shift/Reset the Penultimate Backpropagator.* ICFP 2019. Related to Chapter 3.

1.5 Background

In this section, we provide the necessary background to understand the concepts used in the following chapters: Futamura Projections and Lightweight Modular Staging framework. This section is taken from [149].

1.5.1 Futamura Projections for Query Engines

In 1970, at a time when the hierarchical and network models of data [15] were *du jour*, Codd's seminal work on relational data processing appeared in CACM [39]. One year later, in 1971, Yoshihiko Futamura published his paper "Partial Evaluation

of Computation Process–An approach to a Compiler-Compiler" in the Transactions of the Institute of Electronics and Communication Engineers of Japan [63]. The fundamental insight of Codd was that data could be profitably represented as high-level relations without explicit reference to a given storage model or traversal strategy. The fundamental insight of Futamura was that compilers are not fundamentally different from interpreters, and that compilation can be profitably understood as *specialization* of an interpreter, without explicit reference to a given hardware platform or code generation strategy.

To understand this idea, we first need to understand *specialization* of programs. In the most basic sense, this means to take a generic function, instantiate it with a given argument, and simplify. For example, consider the generic two-argument power function that computes x^n:

```
def power(x:Int, n:Int): Int =
  if (n == 0) 1 else x * power(x, n - 1)
```

If we know the exponent value, e.g., `n = 4`, we can derive a specialized, *residual*, power function:

```
def power4(x:Int): Int = x * x * x * x
```

This form of specialization is also known as *partial evaluation* [81].

Specializing Interpreters. The key idea of Futamura was to apply specialization to interpreters. Like `power` above, an interpreter is a two-argument function: its arguments are the code of the function to interpret, and the input data this function should be called with. Figure 1.1a illustrates the case of databases: The query engine evaluates a SQL query (static input) and data (dynamic input) to produce the result. The effect of specializing an interpreter is shown in Figure 1.1b: if we have a program specializer, or *partial evaluator*, which for historical reasons is often called `mix`, then we can specialize the (query) interpreter with respect to a given source program (query). The result is a single-argument program that computes the query result directly on the data, and runs much faster than running the query through the original interpreter. This is because the specialization process strips away all the "interpretive

overhead", i.e., dispatch the interpreter performs on the structure of the query. In other words, through specialization of the interpreter, we are able to obtain a *compiled* version of the given program!

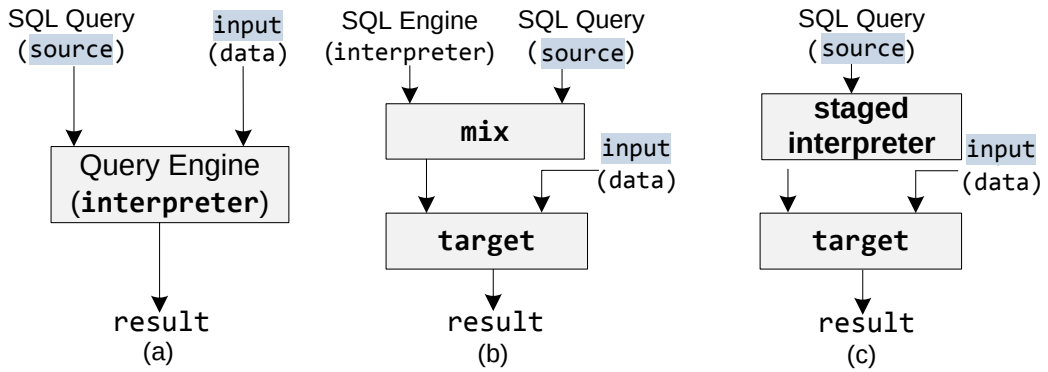


Figure 1.1.: (a) Query interpreter (b) applying the first Futamura projection on a query interpreter (c) Flare realization of the first Futamura projection. Figure taken from [149].

This key result–partially evaluating an interpreter with respect to a source program produces a compiled version of that program–is known as the first Futamura projection. Less relevant for us, the second and third Futamura projections explain how *self-application* of `mix` can, in theory, derive a compiler generator: a program that takes any interpreter and produces a compiler from it.

Codd's idea has been wildly successful, spawning multi-billion-dollar industries, to a large extent thanks to the development of powerful automatic query optimization techniques, which work very well in practice due to the narrow semantic model of relational algebra. Futamura's idea of deriving compilers from interpreters automatically via self-applicable partial evaluation received substantial attention from the research community in the 1980s and 1990s [80], but has not seen the same practical success. Despite partial successes in research, fully automatic partial evaluation has turned out to be largely intractable in practice due to the difficulty of binding-time separation [81]: deciding which expressions in a program to evaluate directly, at specialization time, and which ones to residualize into generated code.

1.5.2 Lightweight Modular Staging

Lightweight Modular Staging (LMS) [130] is a library-based generative programming and compiler framework. LMS maintains a graph-like intermediate representation (IR) to encode *high-level* constructs and operations. Moreover, LMS provides a high-level interface to manipulate the IR graph. The idea in LMS is similar to the Dataset idea in Spark, but is handling a general purpose programing model.

LMS distinguishes two types of expressions; present-stage expressions that are executed normally and future-stage expressions that are compiled into code. LMS defines a special type constructor `Rep[T]` to denote future-stage expressions, e.g., our `MyInt` corresponds to `Rep[Int]` in LMS, and given two `Rep[Int]` values `a` and `b`, *evaluating* the expression `a+b` will *generate* code to perform the addition. LMS provides implementations for all primitive `Rep[T]` types, i.e., strings, arrays, etc. In addition, LMS also provides overloaded control-flow primitives, e.g., `if (c) a else b` where `c` is a `Rep[Boolean]`.

Despite academic success [80–82], the magic `mix` component in Figure 1.1b has turned out to be elusive in practice, however all is not lost. We just have to find another way to implement program specialization, perhaps with some help from the programmer. Going back to the example in 1.5.1, we can implement a self-specializing power function like this:

```
def power(x: Rep[Int], n:Int): Rep[Int] =
  if (n == 0) 1 else x * power(x, n - 1)
```

We changed the type of `x` from `Int` to `Rep[Int]`. When we now call `power` with a symbolic input value:

```
val in: Rep[Int] = fresh("in")
power(in, 4)
```

LMS will emit the desired specialized computation:

```
int x0 = in * 1;  int x1 = in * x0;
int x2 = in * x1; int x3 = in * x2; // = in * in * in * in
```

Based on this core idea of introducing special data types for symbolic or *staged* computation, we suddenly have a handle on making the first Futamura projection immediately practical. Previous works on staging include [36, 147, 161].

2 FLARE: ACCELERATING DATA PROCESSING

This Chapter is based on [54].

2.1 Introduction

Systems like Apache Spark [12] have gained enormous traction thanks to their intuitive APIs and ability to scale to very large data sizes, thereby commoditizing petabyte-scale (PB) data processing for large numbers of users. But thanks to its attractive programming interface and tooling, people are also increasingly using Spark for smaller workloads. Even for companies that *also* have PB-scale data, there is typically a long tail of tasks of much smaller size, which make up a very important class of workloads [40, 134]. In such cases, Spark's performance is suboptimal. For such medium-size workloads, performance may still be of critical importance if there are many such jobs, individual jobs are computationally heavy, or need to be run very frequently on changing data. This is the problem we address in this Chapter. We present Flare, an accelerator module for Spark that delivers order of magnitude speedups on scale-up architectures for a large class of applications. A high-level view of Flare's architecture can be seen in Figure 2.1b.

Inspiration from In-Memory Databases Flare is based on native code generation techniques that have been pioneered by in-memory databases (e.g., HyPer [107]). Given the multitude of front-end programming paradigms, it is not immediately clear that looking at relational databases is the right idea. However, we argue that this is indeed the right strategy: Despite the variety of front-end interfaces, contemporary Spark is, at its core, an SQL engine and query optimizer [12]. Rich front-end APIs are increasingly based on DataFrames, which are internally represented very much

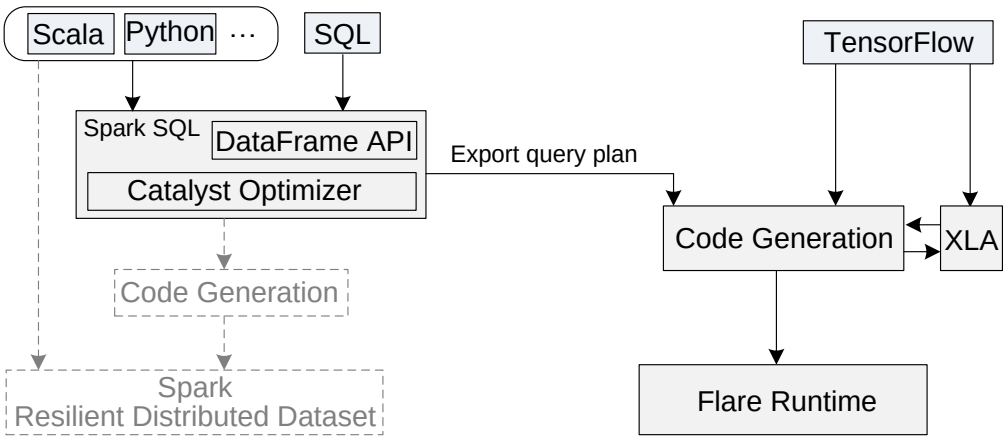


Figure 2.1.: Flare system overview: (a) architecture of Spark adapted from [12]; (b) Flare generates code for entire queries, eliminating the RDD layer, and orchestrating parallel execution optimized for shared memory architectures. Flare also integrates with TensorFlow.

like SQL query plans. Data frames provide a deferred API, i.e., calls only *construct* a query plan, but do not execute it immediately. Thus, front-end abstractions do not interfere with query optimization. Previous generations of Spark relied critically on arbitrarily UDFs, but this is becoming less and less of a concern as more and more functionality is implemented on top of DataFrames.

With main-memory databases in mind, it follows that one may look to existing databases for answers on improving Spark's performance. A piece of low-hanging fruit seems to be simply translating all DataFrame query plans to an existing best-of-breed main-memory database (e.g., HyPer [107]). However, such systems are *full* database systems, not just query engines, and would require data to be stored in a separate, internal format specific to the external system. As data may be changing rapidly, loading this data into an external system is undesirable, for reasons of both storage size and due to the inherent overhead associated with data loading. Moreover, retaining the ability to interact with other systems (e.g., TensorFlow [4] for machine learning) is unclear.

Another logical alternative would be to build a new system which is overall better optimized than Spark for the particular use case of medium-size workloads and scale-up architectures. While some effort has been done in this vein (e.g., Tupleware [40]), such systems forfeit the ability to leverage existing libraries and frameworks built on top of Spark, including the associated tooling. Whereas a system that competes with Spark must replicate all of this functionality, our goal instead was to build a drop-in module capable of handling workloads for which Spark is not optimized, preferably using methodologies seen in these best-of-breed, external systems (e.g., HyPer).

Native Query Compilation Indeed, the need to accelerate CPU computation prompted the development of a code generation engine that ships with Spark since version 1.4, called Tungsten [12]. However, despite following some of the methodology set forth by HyPer, there are a number of challenges facing such a system, which causes Tungsten to yield suboptimal results by comparison. First, due to the fact that Spark resides in a Java-based ecosystem, Tungsten generates Java code. This (somewhat obviously) yields inferior performance to native execution as seen in HyPer. However, generating native code within Spark poses a challenge of interfacing with the JVM when dealing with e.g., data loading. Another challenge comes from Spark's reliance on resilient distributed datasets (RDDs) as its main internal execution abstraction. Mapping query operators to RDDs imposes boundaries between code generation regions, which incurs nontrivial runtime overhead. Finally, having a code generation engine capable of interfacing with external frameworks and libraries, particularly machine-learning oriented frameworks like TensorFlow and PyTorch, is also challenging due to the wide variety of data representations which may be used.

End-to-End Datapath Optimization In solving the problem of generating native code and working within the Java environment, we focus specifically on the issue of data processing. When working with data directly from memory, it is possible to use the Java Native Interface (JNI) and operate on raw pointers. However, when processing data directly from files, fine-grained interaction between decoding logic in

Java and native code would be required, which is both cumbersome and presents high overhead. To resolve this problem, we elect to reimplement file processing for common formats in native code as well. This provides a fully compiled data path, which in turn provides significant performance benefits. While this does present a problem in calling Java UDFs (user-defined functions) at runtime, we can simply fall back to Spark's existing execution in such a case, as these instances appear rare in most use cases considered. We note in passing that existing work (e.g., Tupleware [40], Froid [121]) has presented other solutions for this problem which could be adopted within our method, as well.

Fault Tolerance on Scale-Up Architectures In addition, we must overcome the challenge of working with Spark's reliance on RDDs. For this, we propose a simple solution: when working in a scale-up, shared memory environment, remove RDDs and bypass all fault tolerance mechanisms, as they are not needed in such architectures (seen in Figure 2.1b). The presence of RDDs fundamentally limits the scope of query compilation to individual query stages, which prevents optimization at the granularity of full queries. Without RDDs, we compile whole queries and eliminate the preexisting boundaries across query stages. This also enables the removal of artifacts of distributed architectures, such as Spark's use of `HashJoinExchange` operators even if the query is run on a single core.

Interfacing with External Code Looking now to the issue of having a robust code generation engine capable of interfacing with external libraries and frameworks within Spark, we note that most performance-critical external frameworks are *also* embracing deferred APIs. This is particularly true for machine learning frameworks, which are based on a notion of execution graphs. This includes popular frameworks like TensorFlow [4], Caffe [79], and ONNX [1], though this list is far from exhaustive. As such, we focus on frameworks with APIs that follow this pattern. Importantly, many of these systems already have a native execution backend, which allows for

speedups by generating all required glue code and keeping the entire data path within native code.

In the following section our study focuses on Spark, however most of the drawbacks that we identify are usually present in other big framework as well. Moreover, our solution - using generative programming - can be applied and improve the performances of these systems by following the same idea.

Contributions The main intellectual contribution of this Chapter is to demonstrate and analyze some of the underlying issues contained in the Spark runtime, and to show that the HyPer query compilation model must be adapted in certain ways to achieve good results in Spark (and systems with a similar architecture like Flink [37]), most importantly to eliminate codegen boundaries as much as possible. For Spark, this means generating code not at the granularity of operator pipelines but compiling whole Catalyst operator trees at once (which may include multiple SQL-queries and subqueries), generating specialized code for data structures, for file loading, etc.

We present Flare, an accelerator module for Spark that solves these (and other) challenges which currently prevent Spark from achieving optimal performance on scale-up architectures for a large class of applications. Building on query compilation techniques from main-memory database systems, Flare incorporates a code generation strategy designed to match the unique aspects of Spark and the characteristics of scale-up architectures, in particular processing data directly from optimized file formats and combining SQL-style relational processing with external libraries such as TensorFlow.

2.2 Background

Apache Spark [168,169] is today's most widely-used big data framework. The core programming abstraction comes in the form of an immutable, implicitly distributed, collection called a resilient distributed dataset (RDD). RDDs serve as high-level programming interfaces, while also transparently managing fault-tolerance.

We present a short example using RDDs (from [12]), which counts the number of errors in a (potentially distributed) log file:

```
val lines  = spark.sparkContext.textFile("...")
val errors = lines.filter(s => s.startsWith("ERROR"))
println("Total errors: " + errors.count())
```

Spark's RDD abstraction provides a *deferred* API: in the above example, the calls to `textFile` and `filter` merely construct a computation graph. In fact, no actual computation occurs until `errors.count` is invoked. Because of this property, RDDs are sometimes described as *lazily evaluated*. However, this is somewhat misleading, as a second call to `errors.count` will re-execute the entire computation. [1] However, RDDs do support memoization via explicit calls to `errors.persist()`, which will mark the dataset to be kept in memory for future operations.

2.2.1 Spark SQL and the DataFrame API

The directed, acyclic computation graph represented by an RDD describes the distributed operations in a rather coarse-grained fashion: at the granularity of `map`, `filter`, and so on. While this level of detail is enough to enable demand-driven computation, scheduling, and fault-tolerance via selective recomputation along the "lineage" of a result [168], it does not provide a full view of the computation applied to each element of a dataset. For example, in the code snippet shown above, the argument to `lines.filter` is a normal Scala closure. This makes integration between RDDs and arbitrary external libraries much easier, but it also means that the given closure must be invoked as-is for every element in the dataset.

As such, the performance of RDDs suffers from two limitations: first, limited visibility for analysis and optimization (especially standard optimizations, e.g., join reordering for relational workloads); and second, substantial interpretive overhead, i.e., function calls for each processed tuple. Recent Spark versions have aimed to ameliorate both issues with the introduction of the Spark SQL subsystem [12].

[1] In its original definition, the term "lazy evaluation" means that each term is evaluated only when needed, and *not more than once* [70].

2.2.2 The Power of Multi-Stage APIs and DSLs

The chief addition of Spark SQL is an alternative API based on DataFrames.

A DataFrame is conceptually equivalent to a table in a relational database; i.e., a collection of rows with named columns. However, like RDDs, the DataFrame API records operations, rather than computing the result.

Therefore, we can write the same example as before:

```
val lines  = spark.read.textFile("...")
val errors = lines.filter($"value".startsWith("ERROR"))
println("Total errors: " + errors.count())
```

This is quite similar to the RDD API in that only the call to `errors.count` will trigger actual execution. Unlike RDDs, however, DataFrames capture the *full* computation/query to be executed. We can obtain the internal representation using `errors.explain()`, which produces the following output:

```
== Physical Plan ==
*Filter StartsWith(value#894, ERROR)
+- *Scan text [value#894]
     Format: ...TextFileFormat@18edbdbb,
     InputPaths: ...,
     ReadSchema: struct<value:string>
```

From the high-level DataFrame operations, Spark SQL computes a *query plan*, much like a relational DBMS. Spark SQL optimizes query plans using its relational query optimizer, called Catalyst, and may even generate Java code at runtime to accelerate parts of the query plan using a component named Tungsten.

It is hard to overstate the benefits of this kind of *deferred* API, which generates a complete program (i.e., query) representation at runtime. First, it enables various kinds of optimizations, including classic relational query optimizations. Second, one can use this API from multiple frontends, which exposes Spark to non-JVM languages such as Python and R, and the API can also serve as a translation target from literal SQL:

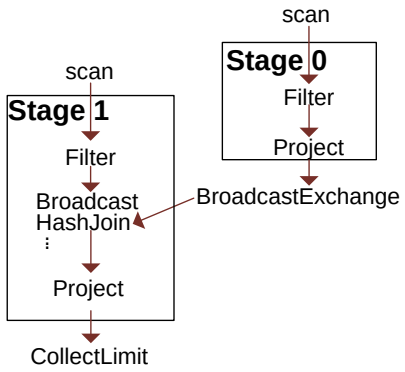
```
lines.createOrReplaceTempView("lines")
val errors = spark.sql("select * from lines
                        where value like 'ERROR%'")
println("Total errors: " + errors.count())
```

Third, one can use the full host language to structure code, and use small functions that pass DataFrames between them to build up a logical plan that is then optimized as a whole.

```
select *
from lineitem, orders
where l_orderkey = o_orderkey
```

```
Spark
  Sort-merge join          14,937 ms
  Broadcast-hash join       4,775 ms
     of which in exchange   2,232 ms
Flare
  In-memory hash join         136 ms
```

(a)



(b)

Figure 2.2.: (a) The cost of Join `lineitem` ⋈ `orders` with different operators (b) Spark's hash join plan shows two separate code generation regions, which communicate through Spark's runtime system.

However, this is only true as long as one stays in the relational world, and, notably, avoids using any external libraries (e.g., TensorFlow). This is a nontrivial restriction; to resolve this, we show in Chapter 3 how the DataFrame model extends to such library calls in Flare.

2.2.3 Catalyst and Tungsten

With the addition of Spark SQL, Spark also introduced a query optimizer known as Catalyst [12]. We elide the details of Catalyst's optimization strategy, as they are largely irrelevant here. After Catalyst has finished optimizing a query plan, Spark's execution backend known as Tungsten takes over. Tungsten aims to improve Spark's performance by reducing the allocation of objects on the Java Virtual Machine (JVM) heap, controlling off-heap memory management, employing cache-aware data structures, and generating Java code which is then compiled to JVM bytecode at

runtime [93]. Notably, these optimizations are able to simultaneously improve the performance of *all* Spark SQL libraries and DataFrame operations [169].

Following the design described by Neumann, and implemented in HyPer [107], Tungsten's code generation engine implements what is known as a "data-centric" model. In this type of model, operator interfaces consist of two methods: `produce`, and `consume`. The `produce` method on an operator signals all child operators to begin producing data in the order defined by the parent operator's semantics. The `consume` method waits to receive and process this data, again in accordance with the parent operator's semantics.

In HyPer (and Tungsten), operators that materialize data (e.g., aggregate, hash join, etc.) are called "pipeline breakers". Where possible, pipelines of operators (e.g., scan, aggregate) are fused to eliminate unnecessary function calls which would otherwise move data between operators. A consequence of this is that all code generated is at the granularity of query *stage*, rather than generating code for the query as a whole. This requires some amount of "glue code" to also be generated, in order to pipeline these generated stages together. The directed graph of the physical plan for a simple join query can be seen in Figure 2.2b. In this figure, we can see that the first stage generates code for scanning and filtering the first table and the second stage generates code for the pipeline of the scan, join, and project operators. In Section 2.4 we discuss the impact of the granularity of code generation and the choice of join algorithm on Spark's performance.

2.3 Spark Performance Analysis

Spark performance studies primarily focus on the scale-out performance, e.g., running big data benchmarks [168] on high-end clusters, performing terabyte sorting [169], etc. However, when considering the class of computationally-heavy workloads that can fit in main-memory, requires multiple iterations, or integrates with external libraries

(e.g., training a machine learning classifier), the performance of Spark becomes sub-optimal.

On a similar note, McSherry, Isard, and Murray have eloquently argued in their 2015 HotOS paper [100] and accompanying blog post [100] that big data systems such as Spark tend to scale well, but often this is because there is a lot of internal overhead. In particular, McSherry et al. demonstrate that a straightforward native implementation of the PageRank algorithm [112] running on a single laptop can outperform a Spark cluster with 128 cores, using the then-current version.

Laptop vs. Cluster Inspired by this setup and the following quote, we are interested in gauging the inherent overheads of Spark and Spark SQL in absolute terms:

> "You can have a second computer once you've shown you know how to use the first one."
>
> — Paul Barham, via [101]

For our benchmark, we pick the simplest query from the industry-standard TPC-H benchmark: Query 6 (shown in Figure 2.3). We define the schema of table `lineitem`, provide the source file, and finally register it as a temporary table for Spark SQL (steps not shown). For our experiments, we use scale factor 2 (SF2) of the TPC-H data set, which means that table `lineitem` is stored in a CSV file of about 1.4 GB. Following the setup by McSherry et al., we run our tests on a fairly standard laptop.[2] Note that all times referenced here may be found in Table 2.1.

Table 2.1.: Running times for Q6 in Spark.

Spark SQL	Preload ms	Query ms
Direct CSV	-	24,400
Preload CSV	118,062	1,418
Hand-Written C / Flare		
Preload CSV	2,847	45

[2]MacBook Pro Retina 2012, 2.6 GHz Intel Core i7, 16 GB 1600 MHz DDR3, 500 GB SSD, Spark 2.0, Java HotSpot VM 1.8.0_112-b16

```
val tpchq6 = spark.sql("""
  select
    sum(l_extendedprice*l_discount) as revenue
  from
    lineitem
  where
    l_shipdate >= to_date('1994-01-01')
    and l_shipdate < to_date('1995-01-01')
    and l_discount between 0.05 and 0.07
    and l_quantity < 24
""")
```

Figure 2.3.: Query 6 from the TPC-H benchmark in Spark.

```
// data loading elided ...
for (i = 0; i < size; i++) {
  double l_quantity = l_quantity_col[i];
  double l_extendedprice = l_extendedprice_col[i];
  double l_discount = l_discount_col[i];
  long l_shipdate = l_shipdate_col[i];
  if (l_shipdate >= 19940101L && l_shipdate < 19950101L &&
      l_discount >= 0.05 && l_discount <= 0.07 &&
      l_quantity < 24.0) {
    revenue += l_extendedprice * l_discount;
  }
} ...
```

Figure 2.4.: Query 6 from the TPC-H benchmark hand-written C code.

We first do a naive run of our query, Q6. As reported in Table 2.1, we achieve a result of 24 seconds, which is clearly suboptimal. In aiming to boost performance, and focus on the computational part we rerun the experiment with the data preloaded.

We note in passing that preloading is quite slow (almost 2 min), which may be due to a variety of factors. With things preloaded, however, we can now execute our query in-memory, and we get a much better result of around 1.4 seconds. Running the query a few more times yields further speedups, but timings stagnate at around 1 second (timing from subsequent runs elided). Using 1s as our baseline, we must now qualify this result.

Hand-Written C Due to the simplicity of Q6, we elect to write a program in C which performs precisely the same computation: mapping the input file into memory using the `mmap` system call, loading the data into an in-memory columnar representation, and then executing the main query loop (see Figure 2.4).

Compiling this C program via `gcc -O3 Q6.c` and running the resultant output file yields a time of 2.8 seconds (including data loading), only 45ms of which is performing the actual query computation. Note that in comparison to Spark 2.0, this is a striking 20× speedup. Performing the same query in HyPer, however, takes only 46.58ms, well within the margin of error of the hand-written C code. This disparity in performance shows that although Tungsten is written with the methodologies prescribed by HyPer in mind, there exist some impediments either in the implementation of these methodologies or in the Spark runtime itself which prevent Spark from achieving optimal performance for these cases.

2.4 Major Bottlenecks

By profiling Spark SQL during a run of Q6, we are able to determine two key reasons for the large gap in performance between Spark and HyPer. Note that while we focus our discussion mainly on Q6, which requires low computational power and uses only trivial query operators, these bottlenecks appear in nearly every query in the TPC-H benchmark.

Data Exchange Between Code Boundaries We first observe that Tungsten must generate multiple pieces of code: one for the main query loop, the other an iterator to traverse the in-memory data structure.

Consider the HashJoin code in Figure 2.5. We can see that Tungsten's produce/-consume interface generates a loop which iterates over data through an iterator interface, then invokes the `consume` method at the end of the loop in order to perform evaluation. HyPer's original codegen model is centrally designed around data-centric pipelines within a given query, the notion of "pipeline-breakers" as coarse-grained

```
case class BroadcastHashJoinExec(/* ... inputs elided ... */)
  extends BinaryExecNode with HashJoin with CodegenSupport {
  // ... fields elided ...
  override def doProduce(ctx: CodegenContext): String =
    streamedPlan.asInstanceOf[CodegenSupport].produce(ctx, this)
  override def doConsume(ctx: CodegenContext, input: Seq[ExprCode],
     row: ExprCode): String = {
    val (broadcastRelation, relationTerm) = prepareBroadcast(ctx)
    val (keyEv, anyNull) = genStreamSideJoinKey(ctx, input)
    val (matched, checkCondition, buildVars) =
       getJoinCondition(ctx, input)
    val numOutput = metricTerm(ctx, "numOutputRows")

    val resultVars = ...
    ctx.copyResult = true
    val matches = ctx.freshName("matches")
    val iteratorCls = classOf[Iterator[UnsafeRow]].getName
    s"""
      |// generate join key for stream side
      |${keyEv.code}
      |// find matches from HashRelation
      |$iteratorCls $matches = $anyNull ? null :
         ($iteratorCls)$relationTerm.get(${keyEv.value});
      |if ($matches == null) continue;
      |while ($matches.hasNext()) {
      |  UnsafeRow $matched = (UnsafeRow) $matches.next();
      |  $checkCondition
      |  $numOutput.add(1);
      |  ${consume(ctx, resultVars)}
      |}
    """.stripMargin
  }
}
```

Figure 2.5.: Spark implementation of inner HashJoin.

boundaries of data flow, and the combination of pre-written code at the boundary between pipelines with generated code within each pipeline. While the particular implementation of this design in HyPer leads to good results in HyPer itself, the direct implementation of HyPers pipeline-focused approach in Spark and similar systems falls short because the overhead of traversing pipeline boundaries is much higher (Java vs C++, RDD overhead, ecosystem integration, etc).

The CPU profile (Figure 2.6) shows that 80% of the execution time is spent in one of two ways: accessing and decoding the in-memory data representation, or moving

between the two pieces of generated code through code paths which are part of the precompiled Spark runtime. In order to avoid this overhead, then, we must replace the runtime altogether with one able to reason about the *entire* query, rather than just the stages.

JVM Overhead Even if the previous indirection is removed and replaced with a unified piece of Java code, the performance remains approximately 30% lower than our hand-written C code. This difference becomes more pronounced in other TPC-H queries which require both memory management and tighter low-level control over data structures. This bottleneck is certainly expected, and choosing a lower level language does alleviate this performance loss greatly.

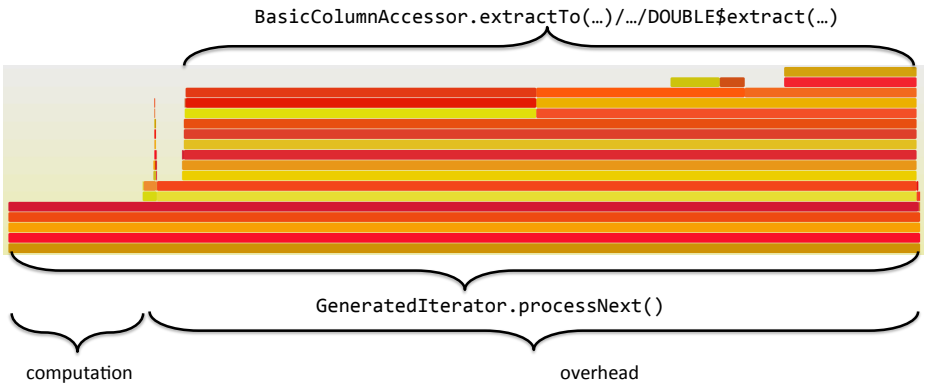


Figure 2.6.: CPU profile of TPC-H Q6 in Spark SQL, after preloading the `lineitem` table. 80% of time is spent accessing and decoding the in-memory data representation.

Other Bottlenecks As shown, even fixing these bottlenecks is not enough. This becomes even more apparent when moving away from Q6. In dealing with more complex queries, concerns regarding granularity of code generation and the necessity to interface with the Spark runtime system become more pronounced than with TPC-H Q6. In fact, queries which require join operations exhibit some unfortunate consequences for main-memory execution due to Spark's design as primarily a cluster-computing framework. Figure 2.2a shows timings for a simple join query that joins

the `lineitem` and `orders` tables of the TPC-H benchmark. Spark's query optimizer picks an expensive sort-merge join by default. Note that this may be the correct choice for distributed or out-of-core execution, but is suboptimal for main memory. With some tuning, it is possible to force Spark's query planner to opt for a hash join instead, which is more efficient for our architecture. However, even this follows a broadcast model with high overhead for the internal exchange operator (2.2s of 4.7s) which is present in the physical plan even when running on a single core.

2.5 Adding Fuel to the Fire

Based on the observations made in Sections 2.3 and 2.4, we formally present Flare: a new backend which acts as an accelerator for Spark. Flare eliminates all previously identified bottlenecks *without* removing the expressiveness and power of its frontends. At its core, Flare efficiently generates code, and brings Spark's performance closer to HyPer and hand-written C. Flare compiles whole queries instead of only query *stages*, effectively bypassing Spark's RDD layer and runtime for operations like hash joins in shared-memory environments. Flare also goes beyond purely relational workloads by adding another intermediate layer between query plans and generated code.

2.5.1 Interface between Spark and Flare

Flare supports most available Spark SQL DataFrame or DataSet operations (i.e., all operations which can be applied to a DataFrame and have a representation as Catalyst operators), though any operators currently missing could be added without compatibility constraints. In the event that a Spark job contains operations that are not part of the SQL frontend, Flare can still be used to accelerate SQL operations and then return the result to the Spark runtime, which will then use the result for the rest of the computation. However, the benefit of doing this may be negated by the communication overhead between the two systems.

Flare can operate in one of two modes. Either users must invoke a function to convert the DataFrame they wish to compute into a Flare DataFrame (a conversion that may fail with a descriptive error), to that end Flare exposes a dedicated API to allow users to pick which DataFrames to evaluate through Flare:

```
val df = spark.sql("...") // create DataFrame (SQL or direct)
val fd = flare(df)        // turn it into a FlareDataFrame
fd.show()                 // execute query plan with Flare
```

Or one can set a configuration item in Spark to use Flare on all queries where possible, and only fall back to the default Spark execution when necessary (optionally emitting a warning when doing so). When Flare is invoked, it first generates C code as explained in the following section, then invokes a C compiler, and finally launches the resulting binary either inside the JVM, or as a separate process. This bypasses Spark's runtime entirely, relying solely on Flare's runtime to trigger execution of the generated code.

2.5.2 Flare: Architecture

In a work published in SIGMOD'18 [149], we presented a principled approach to derive query compilers from query interpreters, and show that these compilers can generate excellent code in a single pass, that is competitive with HyPer [107] and DBLAB [137]. Flare is based on this design: a single pass compiler (See Figure 2.7).

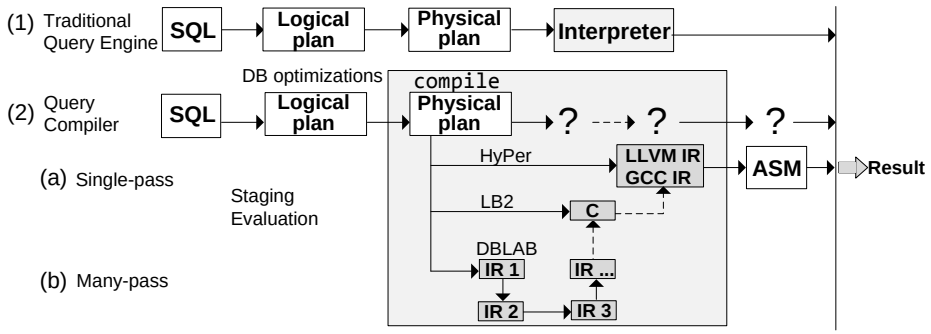


Figure 2.7.: Illustration of (1) query interpreter (2) query compilers (a) single-pass compiler (b) many-pass compiler

A high-level view of Flare's architecture is illustrated in Figure 2.1b. Spark SQL's front-end, the DataFrame API, and the Catalyst optimizer all remain the same. When dealing with relational workloads, the optimized query plan is exported without modification from Catalyst to Flare, upon which Flare performs a compilation pass and creates a code generation object for each of the nodes.

At a closer look, Figure 2.8 illustrates an end-to-end execution path in Flare. Flare analyzes Spark's optimized plan (which possibly embeds external libraries as UDFs) and constructs a computation graph that encodes relational operators, data structures, UDFs, data layout, and any needed configurations.

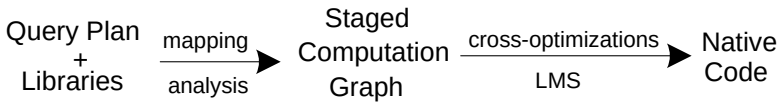


Figure 2.8.: Query compilation in Flare.

Staging and code generation in Flare In the context of query compilation, LMS is used to specialize a query evaluator with respect to a query plan [85, 128]. Based on partial evaluation results (the first Futamura projection [63]), the outcome of programmatic specialization is a compiled target of the query. Figure 2.9 shows an example of compiling a join query in Flare, in which the specialization logic (i.e., staging code using `Rep`) is placed at the granularity of low-level control flow constructs and primitive operators.

Following the `InnerJoin` code generation example in Figure 2.9, a `CodeGen` object is generated from each of the two children, after which the logic of the `Join` operator is implemented: the left child's code generator is invoked and the tuples produced populate a hash map. The right child's code generator is then invoked, and for each of the tuples produced, the matching lines from the left table are extracted from the map, merged, and finally become the produced value of the `Join` operator. LMS performs some lightweight optimizations (e.g., common subexpression

```
case class DataLoop(foreach: (Rep[Record] => Unit) => Unit)
type ThreadCallback = Rep[Int] => DataLoop => Unit
case class CodeGen(gen: ThreadCallback => Unit)

// extract the join hash key functions
def compileCond(cond: Option[Expression]): (Rep[Record] =>
   Rep[Record], Rep[Record] => Rep[Record]) = ...
def compile(plan: LogicalPlan): CodeGen = plan match {
  case HashJoin(left, right, Inner, cond) =>
    val lGen = compile(left); val rGen = compile(right)
    val (lkey, rkey) = compileCond(cond)

    val hmap = new ParHashMap[Record,Record]()
    CodeGen(threadCallback =>
      lGen.gen { tId => dataloop => // start section for left child
        val lhmap = hmap.partition(tId) // Thread local data
        for (ltuple <- dataloop) lhmap += (lkey(ltuple), ltuple)
      }
      rGen.gen { tId => dataloop => // start section for right child
        threadCallback(tId) { callback => // invoke downstream op
          for (rtuple <- dataloop)
            for (ltuple <- hmap(rkey(rtuple)))
              callback(merge(ltuple, rtuple)) // feed downstream op
    } } )
  case ...
}
```

Figure 2.9.: Internal Flare operator that generates code for HashJoin (LogicalPlan and HashJoin are Spark classes).

elimination, dead code elimination), and generates C code that can be compiled and executed by the Flare runtime.

Interestingly, this implementation looks exactly like the implementation of an interpreter. Indeed, this is no coincidence: much like Spark uses multi-stage APIs at the operational level, Flare uses the LMS compiler framework, which implements the same concept, but at a lower level. In the same way that Scala (or Python) is used to build DataFrames in Spark, we use Scala to build a graph which represents the computations needing to be generated. We qualify the code generation of Spark as coarse-grain. The `BroadcastHashJoinExec` operator in Figure 2.5 generates a string that corresponds to the full join computation. This String is generated with regard to some placeholders for the inputs/outputs and join conditons that are

specific to the given query. However, what is hardcoded in the template string will be generated in the same way for every join. Contrast this with Flare's fine-grained code generation: The code in Figure 2.9 also generates code for the Join operator. However, it does not generate one big string; rather, it invokes functions that express the logic of the operator using the full power of the Scala language. The use of `Rep[T]` expressions in judicious places triggers code generation and produces only low-level operations.

With the goal of removing the tightest bottlenecks first, the implementation of Flare has focused on maximizing performance within a single machine. Therefore, Flare does not implement any specific optimizations for distributed execution. Furthermore, Flare is also unable to handle any workloads which require more memory than the machine has available. In either of these cases, we fall back to the Spark runtime.

2.5.3 Optimizing Data Loading

Data loading is an often overlooked factor data processing, and is seldom reported in benchmarks. However, we recognize that data loading from CSV can often be the dominant performance factor for Spark SQL queries. The Apache Parquet [157] format is an attractive alternative, modeled after Dremel [102]. As a binary columnar format, it offers opportunities for compression, and queries can load only required columns instead of all data.

While Parquet allows for irrelevant data to be ignored almost entirely, Spark's code to read Parquet files is very generic, resulting in undue overhead. This generality is primarily due to supporting multiple compression and encoding techniques, but there also exists overhead in determining which column iterators are needed. While these sources of overhead seem somewhat unavoidable, in reality they can be resolved by generating specialized code. In Flare, we implement compiled CSV and Parquet

readers that generate native code specialized to a given schema. As a result, Flare can compile data paths end-to-end. We evaluate these readers in Section 2.5.5.

2.5.4 Indexing Structures

Query engines build indexing structures to minimize time spent in table lookups to speed-up query execution. Small-size data processing is performed efficiently using table scans, whereas very large datasets are executed in latency-insensitive contexts. On the other hand, medium-size workloads can profit from indexes, as these datasets are often processed under tight latency constraints where performing full table scans is infeasible. On that basis, Flare supports indexing structures on primary and foreign keys. At the time of writing, Spark SQL does not support index-based plans. Thus, Flare adds metadata to the table schema that describes index type and key attributes. At loading time, Flare builds indexes as specified in the table definition. Furthermore, Flare implements a set of index-based operators, e.g., scan and join following the methodology described in [149]. Finally, at compilation time, Flare maps Spark's operators to use the index-based operators if such an index is present. The index-based operators are implemented with the same technique described for the basic operators, but shortcut some computation by using the index rather than requesting data from its children.

2.5.5 Experimental Results

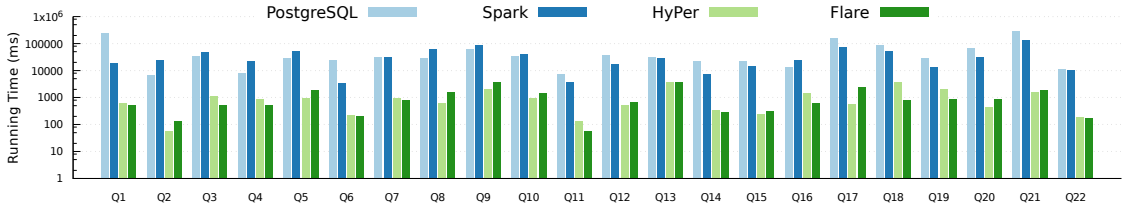
To assess the performance and acceleration potential of Flare in comparison to Spark, we present two sets of experiments. The first set focuses on a standard relational benchmark; the second set evaluates heterogeneous workloads, consisting of relational processing combined with a TensorFlow machine learning kernel. Our experiments span single-core, multi-core, and NUMA targets.

The experiments focuses on a standard relational workload, and demonstrates that the inherent overheads of Spark SQL cause a slowdown of at least 10× compared

to the best available query engines for in-memory execution on a single core. Our experiments show that Flare is able to bridge this gap, accelerating Spark SQL to the same level of performance as state-of-the-art query compiler systems, while retaining the flexibility of Spark's DataFrame API. We also compare parallel speedups, the effect of NUMA optimization, and evaluate the performance benefits of optimized data loading.

Environment We conducted our experiments on a single NUMA machine with 4 sockets, 24 Xeon(R) Platinum 8168 cores per socket, and 750GB RAM per socket (3 TB total). The operating system is Ubuntu 16.04.4 LTS. We use Spark 2.3, Scala 2.11, Postgres 10.2, HyPer v0.5-222-g04766a1, and GCC 5.4 with optimization flags `-O3`.

Dataset We use the standard TPC-H [155] benchmark with scale factor SF10 for sequential, and SF20 and SF100 for parallel execution.



SF10	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11
Postgres	241404	6649	33721	7936	30043	23358	32501	29759	64224	33145	7093
Spark SQL	18219	23741	47816	22630	51731	3383	31770	63823	88861	42216	3857
HyPer	603	59	1126	842	941	232	943	616	1984	967	131
Flare	530	139	532	521	748	198	830	1525	3124	1436	56

SF10	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22
Postgres	37880	31242	22058	23133	13232	155449	90949	29452	65541	299178	11703
Spark SQL	17233	28489	7403	14542	23371	70944	53932	13085	31226	128910	10030
HyPer	501	3625	330	253	1399	563	3703	1980	434	1626	180
Flare	656	3727	278	302	620	2343	823	909	870	1962	177

Figure 2.10.: Performance comparison of Postgres, HyPer, Spark SQL, Flare in SF10.

Single-Core Running Time

In this experiment, we compare the single-core, absolute running time of Flare with Postgres, HyPer, and Spark using the TPC-H benchmark with scale factor SF10. In the case of Spark, we use a single executor thread, though the JVM may spawn auxiliary threads to handle GC or the just-in-time compilation. Postgres and HyPer implement cost-based optimizers that can avoid inefficient query plans, in particular by reordering joins. While Spark's Catalyst optimizer [12] is also cost-based, the default configurations do not perform any kind of join re-ordering. Hence, we match the join ordering of the query plan in Spark SQL and Flare with HyPer's, with a small number of exceptions: in Spark SQL, the original join ordering given in the TPC-H reference outperformed the HyPer plans for Q5, Q9, Q10, and Q11 in Spark SQL, and for Q10 in Flare. For these queries, we kept the original join ordering as is. For Spark SQL, this difference is mainly due to Catalyst picking sort-merge joins over hash joins. It is worth pointing out that HyPer and Postgres plans can use indexes on primary keys, which may give an additional advantage.

Figure 2.10 gives the absolute execution time of Postgres, HyPer, Spark SQL, and Flare for all TPC-H queries. For all systems, data loading time is excluded, i.e., only execution time is reported. In Spark and Flare, we use `persist` to ensure that the data is loaded from memory. At first glance, the performance of Flare and HyPer lie within the same range, and notably outperform Postgres and Spark in all queries. Similarly, Spark's performance is comparable to Postgres's in most of the queries. Unlike the other systems, Postgres does not compile queries at runtime, and relies on the Volcano model [68] for query evaluation, which incurs significant overhead. Hence, we can see that Spark's query compilation does not provide a significant advantage over a standard interpreted query engines on most queries.

At a closer look, Flare outperforms Spark SQL in aggregate queries Q1 and Q6 by 32× and 13× respectively. We observe that Spark is 200× slower than Flare in nested queries (e.g., Q2) After examining the execution plans of Q2, we found that

Catalyst's plan does not detect all patterns that help with avoiding re-computations, e.g., a table which has been previously scanned or sorted. In join queries, e.g., Q5, Q10, Q14, etc., Flare is faster than Spark SQL by 19×-76×. Likewise, in join variants outer join Q13, semi-join Q21, and anti-join Q22, Flare is faster by 7×, 51× and 36× respectively.

The single-core performance gap between Spark SQL and Flare is attributed to the bottlenecks identified in Sections 2.3 and 2.4. First, overhead associated with low-level data access on the JVM. Second, Spark SQL's *distributed-first* strategy that employs costly distributed operators, e.g., sort-merge join and broadcast hash join, even when running on a single core. Third, internal bottlenecks in in-memory processing, the overhead of RDD operations, and communication through Spark's runtime system. By compiling entire queries, instead of isolated query stages, Flare effectively avoids these bottlenecks.

HyPer [107] is a state-of-the-art compiled relational query engine. A precursory look shows that Flare is faster than HyPer by 10%-60% in Q4-Q5,Q7, and Q14-Q16. Moreover, Flare is faster by 2× in Q3, Q11, and Q18. On the other hand, HyPer is faster than Flare by 20%-60% in Q9, Q10, Q12, and Q21. Moreover, HyPer is faster by 2×-4× in Q2, Q8, Q17, and Q20. This performance gap is, in part, attributed to (1) HyPer's use of specialized operators like GroupJoin [104], and (2) employing indexes on primary keys as seen in Q2, Q8, etc., whereas Flare (and Spark SQL) currently does not support indexes.

In summary, while both Flare and HyPer generate native code at runtime, subtle implementation differences in query evaluation and code generation can result in faster code. For instance, HyPer uses proper decimal precision numbers, whereas Flare follows Spark in using double precision floating point values which are native to the architecture. Furthermore, HyPer generates LLVM code, whereas Flare generates C code which is compiled with GCC.

Optimized Data Loading

An often overlooked part of data processing is data loading. Flare contains an optimized implementation for both CSV files and the columnar Apache Parquet format.[3] We show loading times for each of the TPC-H tables in Table 2.2.

Full table read From the data in Table 2.2, we see that in both Spark and Flare, the Parquet file readers outperform the CSV file readers in most scenarios, despite this being a worst-case scenario for Parquet. Spark's CSV reader was faster in only one case: reading `nation`, a table with only 25 rows. In all other cases, Spark's Parquet reader was 1.33×-1.81× faster. However, Flare's highly optimized CSV reader operates at a closer level of performance to the Parquet reader, with all tables except `supplier` having a benefit of less than a 1.25× speedup by using Parquet.

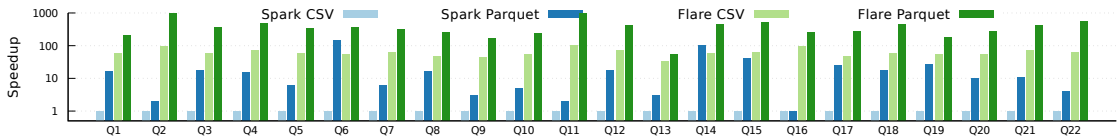


Figure 2.11.: Speedup for TPC-H SF1 when streaming data from SSD on a single thread.

Performing queries Figure 2.11 shows speedups gained from executing queries without preloading data for both systems. Whereas reading an entire table gives Spark and Flare marginal speedups, reading just the required data gives speedups in the range of 2×-144× (Q16 remained the same) for Spark and 60%-14× for Flare. Across systems, Flare's Parquet reader demonstrated between a 2.5×-617× speedup over Spark's, and between 34×-101× over Spark's CSV reader. While the speedup over Spark lessens slightly in higher scale factors, we found that Flare's Parquet reader consistently performed on average at least one order of magnitude faster across each query, regardless of scale factor.

[3]All Parquet files tested were uncompressed and encoded using PLAIN encoding.

In nearly every case, reading from a Parquet file in Flare is approximately 2×-4× slower than in-memory processing. However, reading from a Parquet file in Spark is rarely significantly slower than in-memory processing. These results show that while reading from Parquet certainly provides performance gains for Spark when compared to reading from CSV, the overall performance bottleneck of Spark does not lie in the cost of reading from SSD compared to in-memory processing.

Table 2.2.: Loading time in ms for TPC-H SF10 in Postgres, HyPer, Flare, and SparkSQL.

Table	#Tuples	Postgres CSV	HyPer CSV	Spark CSV	Spark Parquet	Flare CSV	Flare Parquet
CUSTOMER	1500000	7067	1102	11664	9730	329	266
LINEITEM	59986052	377765	49408	471207	257898	11167	10668
NATION	25	1	8	106	110	< 1	< 1
ORDERS	15000000	60214	33195	85985	54124	2028	1786
PART	2000000	8807	1393	11154	7601	351	340
PARTSUPP	8000000	37408	5265	28748	17731	1164	1010
REGION	5	1	8	102	90	< 1	< 1
SUPPLIER	100000	478	66	616	522	28	16

2.6 Parallel and NUMA Execution

Query engines can implement parallelism either explicitly through special *split* and *merge* operators, or internally by modifying the operator's internal logic to orchestrate parallel execution. Flare does the latter, and currently realizes parallelism using OpenMP [41] annotations within the generated C code, although alternatives are possible. On the architectural level, Flare handles splitting the computation internally across multiple threads, accumulating final results, etc. For instance, the parallel `scan` starts a parallel section, which sets the number of threads and invokes the downstream operators in parallel through a `ThreadCallback` (see Figure 2.9). `join` and `aggregate` operators, in turn, which implement materialization points, implement their `ThreadCallback` method in such a way that parallel invocations

are possible without conflict. This is typically accomplished either through per-thread data structures that are merged after the parallel section or through lock-free data structures.

Flare also contains specific optimizations for environments with non-uniform memory access (NUMA), including pinning threads to specific cores and optimizing the memory layout of various data structures to reduce the need for accessing non-local memory. For instance, memory-bound workloads (e.g., TPC-H Q6) perform small amounts of computation, and do not scale up given a large number of threads on a single CPU socket. Flare's code generation supports such workloads through various data partitioning strategies in order to maximize local processing and to reduce the need for threads to access non-local memory as illustrated in Section 2.6.

2.6.1 Experimental Results

The experimental set up is the same as in Section 2.5.5.

In this experiment, we compare the scalability of Spark SQL and Flare. The experiment focuses on the absolute performance and the Configuration that Outperforms a Single Thread (COST) metric proposed by McSherry et al. [101]. We pick four queries that represent aggregate and join variants.

Figure 2.12 presents speedup numbers for Q6, Q13, Q14, and Q22 when scaled up to 32 cores. At first glance, Spark appears to have good speedups in Q6 and Q13 whereas Flare's Q6 speedup drops for high core counts. However, examining the absolute running times, Flare is faster than Spark SQL by 9×. Furthermore, it takes Spark SQL estimated 12 cores in Q6 to match the performance of Flare's single core. In scaling-up Q13, Flare is consistently faster by 8× on all cores. Similarly, Flare continues to outperform Spark by a steady 25× in Q14 and by 20×-80× in Q22 as the number of cores reaches 32. Notice the COST metric in the last two queries is *infinity*, i.e., there is no Spark configuration that matches Flare's single-core performance.

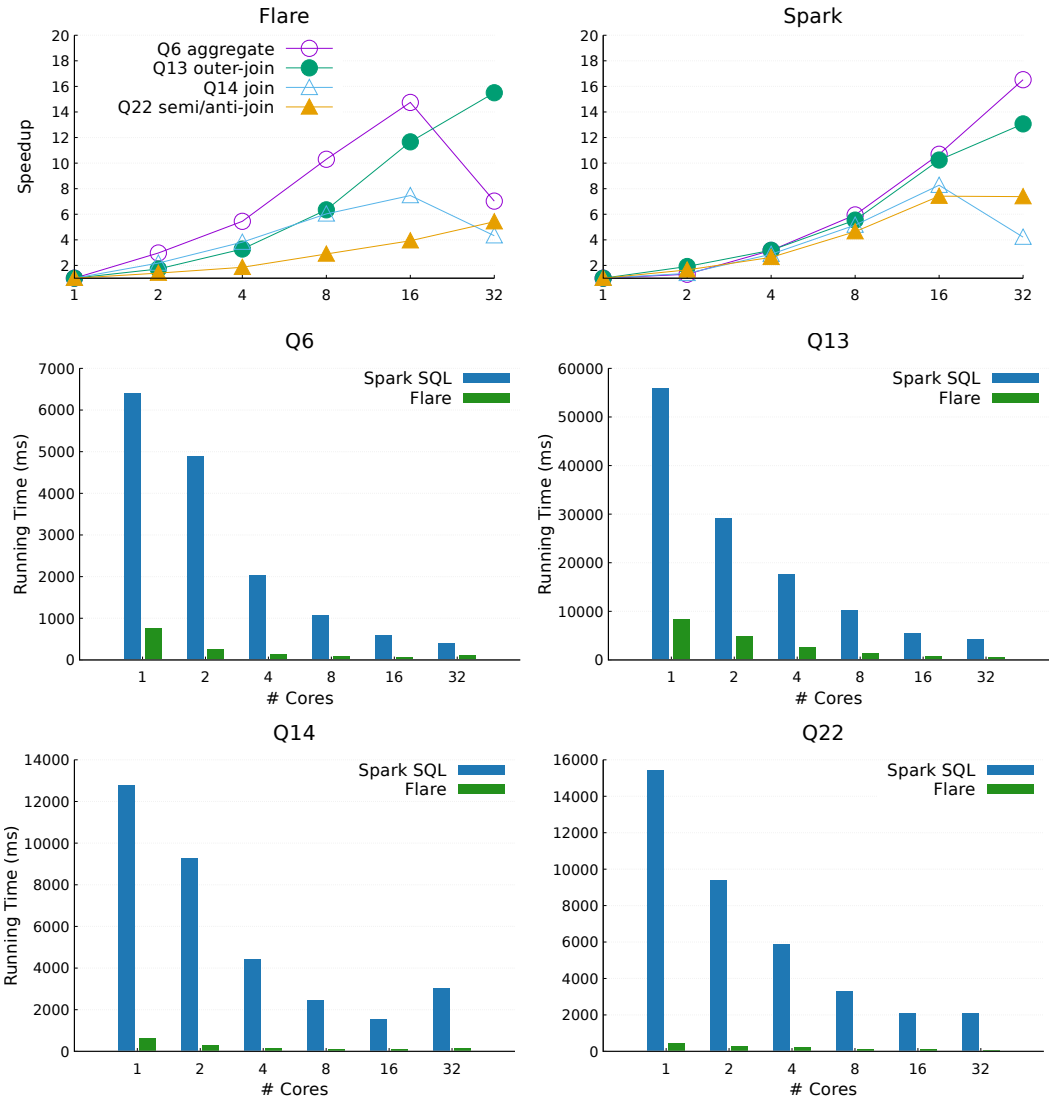


Figure 2.12.: Scaling-up Flare and Spark SQL in SF20, without NUMA optimizations: Spark has good nominal speedups (top), but Flare has better absolute running time in all configurations (bottom). For both systems, NUMA effects for 32 cores are clearly visible (Benchmark machine: 72 cores, 1 TB RAM across 4 CPU sockets, i.e., 18 cores, 256 GB each).

What appears to be good scaling for Spark actually reveals that the runtime incurs significant overhead. In particular, we would expect Q6 to become memory-bound as we increase the level of parallelism. In Flare we can directly observe this effect as a sharp drop from 16 to 32 cores. Since our machine has 18 cores per socket,

for 32 cores, we start accessing non-local memory (NUMA). The reason Spark scales better is because the internal overhead, which does not contribute anything to query evaluation, is trivially parallelizable and hides the memory bandwidth effects. In summary, Flare scales as expected for both of memory and CPU-bound workloads, and reflects the hardware characteristics of the workload, which means that query execution takes good advantage of the available resources – with the exception of multiple CPU sockets, a problem we address next.

As a next step, we evaluate NUMA optimizations in Flare and show that these enable us to scale queries like Q6 to higher core numbers. In particular, we pin threads to individual cores and lay out memory such that most accesses are to the local memory region attached to each socket (Figure 2.13). Q6 performs better when the threads are dispatched on different sockets. This is due to the computation being bounded by the memory bandwidth. As such, when dividing the threads on multiple sockets, we multiply the available bandwidth proportionally. However, as Q1 is more computation bound, dispatching the threads on different sockets has little effect. For both Q1 and Q6, we see scaling up to the capacity of the machine (in our tests, up to 72 cores). This is seen in a maximum speedup of 46× and 58× for Q1 and Q6, respectively.

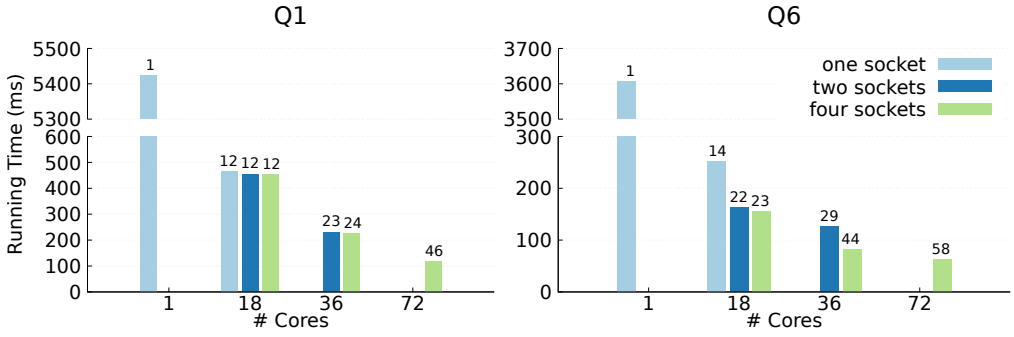


Figure 2.13.: Scaling-up Flare for SF100 with NUMA optimizations on different configurations: threads pinned to one, two, or four sockets. The speedups relative to a single thread are shown on top of the bars (Benchmark machine: 72 cores, 1 TB RAM across 4 CPU sockets, i.e., 18 cores, 256 GB each).

2.7 Distributed Execution

In addition to parallelism, we extend Flare to support distributed execution. While this is still a work in progress, we have good initial results and a proof of concept of a high-level interface.

In Section 2.7.1 we present our abstraction and design, and in Section 2.7.2 we show preliminary experimental results.

2.7.1 Design and Abstraction

Similar to other aspects in Flare, we use specialization to generate efficient code. In the context of metaprogramming with distributed systems, we need to decide which information is going to be known at compile time. We decide to have the number of processes being a constant: the code generated by Flare is thus specialized for this number of processes.

Flare is built around data structures, mainly `Map` for Aggregate, `MultiMap` for Joins, and `Buffer` for almost everything [149]. Therefore, it is natural to look into how to make these data structures distributed. From an execution point of view, each process accumulates partial results in its local data structure, which will later be shuffled to the different processes. While a shuffle operation can be arbitrary complex, we currently propose two useful interfaces: `Mergeable` and `Exchangeable`. `Mergeable` is used for reducing operations and `Exchangable` is used to gather all data at the same location.

Mergeable

```
trait Mergeable[T,U,V] {
  def merge(comm: MergeStrategy)(init: U, mrg: (U, V) => U): (T,
      Rep[Boolean])
}
```

To implement the `Mergeable` trait, the data structure needs to implement a function `merge` that merges the partial results following a `MergeStrategy`. The

`MergeStrategy` will define the source processes from which the partial results will be taken, and the destination processes to which the merged result will be sent. The `merge` function returns the merged result and a flag that indicates if the result is available on the current node.

```
abstract class MergeStrategy {
  def execute(send: Rep[Int] => Unit)(recv: Rep[Int] => Unit):
     Rep[Boolean]
}
```

One useful merge strategy includes an all-to-all merge, in which the partial results of all processes are merged and the final result is made available on all processes.

```
/*
 * Logarithmic merge
 * For 4 processes:
 *  - step 0 (0 <--> 1, 2 <--> 3)
 *  - step 1 (0 <--> 2, 1 <--> 3)
 */
case class AllToAllMerge(nprocs: Int, pid: Rep[Int]) extends
   MergeStrategy {
  assert((nprocs & (nprocs - 1)) == 0) // power of two
  val nstep = (math.log(nprocs) / math.log(2)).toInt
  def execute(send: Rep[Int] => Unit)(recv: Rep[Int] => Unit) = {
    for (step <- 0 until nstep) {
      val binome =
        if ((pid & ((1 << (step + 1)) - 1)) < (1 << step))
          pid + (1 << step)
        else
          pid - (1 << step)
      send(binome)
      recv(binome)
    }
    true
  }
}
```

Another merge strategy is the all-to-one merge, in which the partial results of all processes are merged and the result is made available only on a single process.

```
/*
 * Logarithmic merge (dst 0)
 * For 4 processes:
 * - step 0: 1 --> 0, 3 --> 2
 * - step 1: 2 --> 0
 *
 * In order to merge on a node different from 0, we first rotate
 * the pids.
 */
```

```
case class AllToOneMerge(nprocs: Int, pid: Rep[Int], dst: Rep[Int])
    extends MergeStrategy {
  assert((nprocs & (nprocs - 1)) == 0) // power of two
  val nstep = (math.log(nprocs) / math.log(2)).toInt
  val epid = (pid - dst) & (nprocs - 1) // rotation
  def execute(send: Rep[Int] => Unit)(recv: Rep[Int] => Unit) = {
    for (step <- 0 until nstep) {
      if ((epid & ((1 << step) - 1)) == 0) {
        // idea: if during the next step the node
        // exchanges then it is an aggregator in this round
        if ((epid & ((1 << (step + 1)) - 1)) == 0) {
          recv((pid + (1 << step)) & (nbproc - 1))
        } else {
          send((pid - (1 << step)) & (nbproc - 1))
        }
      }
    }
    pid == dst
  }
}
```

It is important to note that the interface is completely agnostic to the implementation used for the data transfer. We are currently using MPI [61], but it can be easily replaced by another interface offering basic send and receive communication routines.

It is important to notice, our interface does not provide support for synchronization issues. For example, the interface is not completely safe from deadlocks or other common errors. Our MPI interface provides synchronous and asynchronous send/recv methods. For the asynchronous call, we currently offer the concept of asynchronous regions, which ensures that the region can exit only when all asynchronous tasks started in the region are terminated, thus modifying the buffers can be safely done.

This interface can be used as follows; we implement a data structure `IntAgg` that computes an aggregate on integers across processes. In order for the code to be correct for any `MergeStrategy`, we must adhere to the following rules:

1. the send operation needs to be asynchronous

2. for merging, the data needs to have been received

3. the merge operation should not modify the value of the data being sent

We solve this problem by duplicating the data to be sent, and we use a blocking receive call. Of course, this will not always be the correct solution for all data structures. However, it will be solely dependent on the data structure itself and can be optimized accordingly.

```
class IntAgg(value: Rep[Int]) extends Mergeable[Rep[Int], Rep[Int],
    Rep[Int]] {

  def merge(com: MergeStrategy)(init: Rep[Int], up:
      (Rep[Int],Rep[Int]) => Rep[Int]) = {
    var res = up(init, value)
    val gotIt = com.execute { (dst: Rep[Int]) =>
      val tmp = res
      Isend_Int(tmp, dst)
    }{ (src: Rep[Int]) =>
      res += Recv_Int(src)
    }

    (res, gotIt)
  }
}
```

A simple program using that data structure to compute the sum of the pids of 4 processes may look as follows:

```
val nprocs = 4
def main(arr: Rep[Array[String]]) = {
  val pid = getPID
  // Data structure
  val data = new IntAgg(pid)

  asyncMPIRegion {
    // Execute merge
    val (res, gotIt) = data.merge(AllToOneMerge(nprocs, pid, 3))(0,
       {
      (x, y) => x + y
    })

    // print result
    if (gotIt) printf("Aggregate sum is: %d\n", res)
  }
}
```

When executed, this code would generate the following MPI code.

```
int main(int x0, char** x1) {
  int provided;
  MPI_Init_thread(&x0, &x1, MPI_THREAD_FUNNELED, &provided);
  int mpi_pid; MPI_Comm_rank(MPI_COMM_WORLD, &mpi_pid);
  MPI_Request req[100];
  int nreq = 0;
```

```
  int x2 = mpi_pid;
  int x3 = x2;
  int x4 = (x2 - 3) & 3;
  if ((x4 & 1) == 0) {
    int x5;
    MPI_Recv(&x5, 1, MPI_INT, (x2 - 1) & 3, 0, MPI_COMM_WORLD,
       MPI_STATUSES_IGNORE);
    x3 += x5;
  } else {
    int x6 = x3;
    int x7 = nreq++;
    MPI_Isend(&x6, 1, MPI_INT, (x2 + 1) & 3, 0, MPI_COMM_WORLD,
       &req[x7]);
  }
  if ((x4 & 1) == 0) {
    if ((x4 & 3) == 0) {
      int x8;
      MPI_Recv(&x8, 1, MPI_INT, (x2 - 2) & 3, 0, MPI_COMM_WORLD,
         MPI_STATUSES_IGNORE);
      x3 += x8;
    } else {
      int x9 = x3;
      int x10 = nreq++;
      MPI_Isend(&x9, 1, MPI_INT, (x2 + 2) & 3, 0, MPI_COMM_WORLD,
         &req[x10]);
    }
  }
  if (x2 == 3) printf("Aggregate sum is: %d\n", x3);
  MPI_Waitall(nreq, req, MPI_STATUSES_IGNORE);
  MPI_Finalize();
  return 0;
}
```

The power of code generation comes to light with this example. By simply changing the `MergeStrategy`, the same source code will generate vastly different code, while still being correct and without requiring additional effort. An interesting small modification is the unrolling of the loop. In our example, we decided to unroll it because a merge between 4 processes requires only 2 exchange steps. However, this may not be a valid strategy for 256 processes (8 steps), thus generating a loop instead may be better.

Exchangeable

```
trait Exchangeable[T] {
  def gather(dst: Rep[Int]): Unit
  def allGather: Unit
  def exchange(comm: ExchangeStrategy): (T, Rep[Boolean])
}
```

`Exchangeable` is a simpler interface, in part due to the fact that the work is highly dependent on the data structure itself, and also that the whole data structure is shuffled. Thus we will not look at a full example, but only provide the interface:

```
abstract class ExchangeStrategy {
  // exchange strategy. The return value indicates if
  // the current node has access to the global Datastructure
  def execute[T](storage: Exchangeable[T]): Rep[Boolean]
}

case class AllToAll(nprocs: Rep[Int], pid: Rep[Int]) extends
   ExchangeStrategy {
  def execute[T](storage: Exchangeable[T]) = {
    storage.allGather
    true
  }
}

case class AllToOne(nprocs: Rep[Int], pid: Rep[Int], dst: Rep[Int]
   = 0) extends ExchangeStrategy {
  def execute[T](storage: Exchangeable[T]) = {
    storage.gather(dst)
    pid == dst
  }
}
```

Our preliminary work does not take into account the topology of the network, does not propose abstraction for groups and communicators, and does not efficiently handle string values. However, we strongly believe that the design we proposed is still adequate and can be adapted easily in the future.

2.7.2 Experimental Results

For this experiment, we modify the `Aggregate` operator to use a `Map` that implements the `Mergeable` interface and use an `AllToAllMerge` merge strategy. For `Sort` and `Join`, we use `Buffer` and `MultiMaps` that implements the

`Exchangable` interface and the `AllToAll` exchange strategy as well. We also use a CSV reader that mmaps the whole table for each process but only uses a partition of it.

This strategy is not optimal for different reasons. Once the `Map` has been merge in the `Aggregate`, the data structure is partitioned between processes. Thus a better merge procedure will ensure that each process only receives its own partition. Another inefficiency is to always use `AllToAll` without taking into account the size of the data structure and/or the place of the operator in the query plan. For example, Query 1 in TPC-H contains an aggregate that produces 4 values, followed by a sort. Clearly, executing an `AllToAll` merge followed by a partitioning of 4 records for a distributed sort is not optimal. Rather we should do a `AllToOne` merge and terminate the query on the master process. This is something that we plan to do in the future but requires a modification of the query plan. For now, we focus on the code generation, we assume the query plan has been optimized.

With these modifications, we ran five queries of the TPC-H benchmark that do not need to shuffle string values. Figure 2.14 displays the speed-ups for 2, 4, and 8 processes on SF100 (total size of the tables around 100GB).

This experiment illustrates the power of Flare: we manage to support distributed execution without changing the execution engine and only modifying the data structures. Even without the optimal shuffle strategy, we can see that the distributed code executes faster. Queries without joins (Q1 and Q6) almost achieve perfect scaling. While the other queries do not scale as much due to the choice of implementation for the `MultiMap`. The design was initially optimized for single-core machines.

2.8 Beyond Spark

The development of Flare was initially based around Spark. However, we built something more general. The observations made in Section 2.4 hold for many data systems that are JVM based, or written as interpreter. The Flare code base is made up

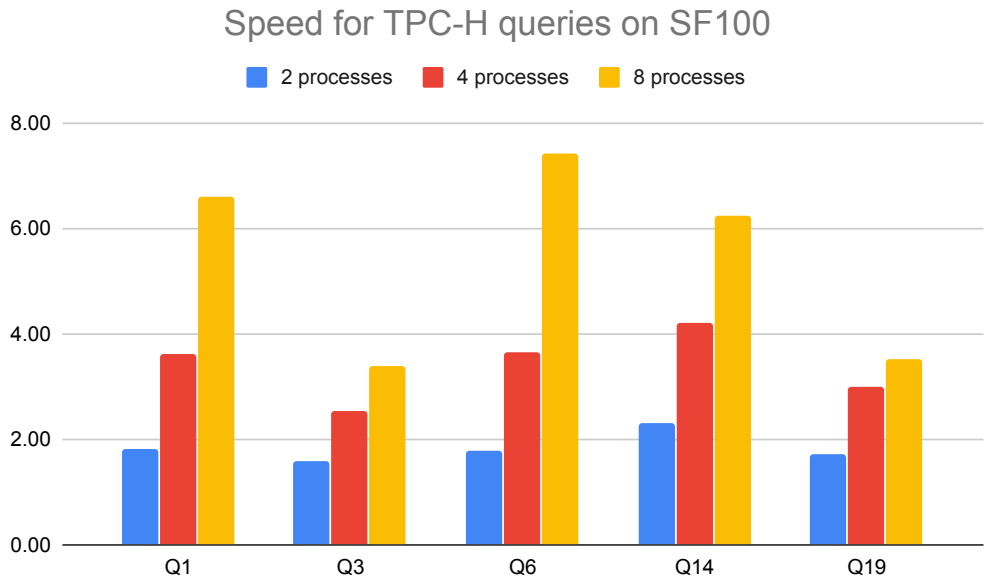


Figure 2.14.: Speedups for Flare on SF100 distributed. (Benchmark machine: 96 cores, 3 TB RAM across 4 CPU sockets, i.e., 24 cores, 750 GB each).

of generic collections (e.g. Buffers, HashMaps, etc.) and Operators (e.g. Aggregates, Joins, Windows, etc.). Thus, it is possible to accelerate many systems that can map their internal query plan to Flare operators. We demonstrated that with the system Flink [6].

We evaluate how Flare accelerates Flink [37] on the full TPCH benchmark. The data is streamed from a CSV file. The performance of Flare is very similar whichever front-end system is used. However, Flink does not generate semi/anti-joins but uses equivalent aggregates on Boolean that are slightly slower (see Q18, Q22).

The engineering effort to write a compiler from Flink to Flare required around 700LOC. The code is very similar to the Spark compiler.

2.9 Future Work

With Flare, we managed to efficiently use a single node to its full potential, and implemented a proof of concept for distributed execution. As future work, it would

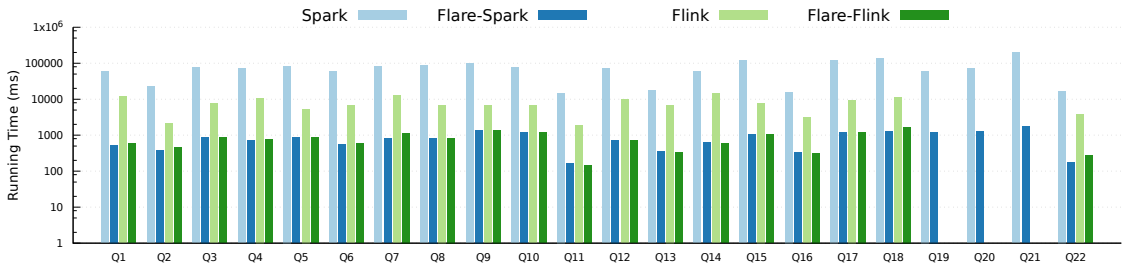


Figure 2.15.: Performance comparison of Spark, Flink, and Flare in SF10

be interesting to explore the remaining abstraction barriers that are still present in most systems. One of the main areas of focus will be to expand our support for generating code that can be executed efficiently on multiple nodes. There are many other challenges to consider, in addition to the ones we already tackled. We also want to explore the world of streaming data processing [37]. In essence, the design of Flare already supports streaming, from a CSV file. Thus we want to add support for timing events that is required for a full streaming data processing.

We have identified the following possible challenges:

- What is the correct level of abstraction for optimal performance? For now we are using MPI primitives. However some recent research [8, 98] showed that generating part of a network stack provides even better performances. In the case of LMS Verify [8], as a bonus, the HTTP parser was formally verified, thus allowing more confidence in the correct implementation of one of the entry points of the system.

- How to handle node failures? Should the code generated be resilient, or should a new piece of code be generated to finish the remaining computation? Both approaches solve the problem, so we want to explore each of them and other possible options. We can already envision that the solution may depend on the workload. As such, we will propose a mechanism and experiments that determine the optimal solution. We explore such avenues in Chapter 4.

- Can we generate code that efficiently balances the load of each node? Would a fully static solution work, or will it require a static/dynamic hybrid method? Load balancing is necessary to achieve good scaling with a distributed system: in the worst case, a distributed application with an unbalanced load could become virtually a single node application.

Our work presented in Chapter 4 is tackling a more generic problem, it was however motivated by the search for a solution to the last two points.

2.10 Related Work

Cluster Computing Frameworks Such frameworks typically implement a combination of parallel, distributed, relational, procedural, and MapReduce computations. The MapReduce model [42] realized in Hadoop [10] performs big data analysis on shared-nothing, potentially unreliable, machines. Twister [51] and Haloop [33] support iterative MapReduce workloads by avoiding reading unnecessary data and keeping invariant data between iterations. Likewise, Spark [168, 169] tackles the issue of data reuse among MapReduce jobs or applications by explicitly persisting intermediate results in memory. Along the same lines, the need for an expressive programming model to perform analytics on structured and semistructured data motivated Hive [156], Dremel [102], Impala [88], Shark [165] and Spark SQL [12] and many others. SnappyData [122] integrates Spark with a transactional main-memory database to realize a unified engine that supports streaming, analytics and transactions. Asterix [19], Stratosphere / Apache Flink [6], and Tupleware [40] are other systems that improve over Spark in various dimensions, including UDFs and performance, and which inspired the design of Flare. While these systems are impressive, Flare sets itself apart by accelerating actual Spark workloads instead of proposing a competing system, and by demonstrating relational performance on par with HyPer [107] on the full set of TPC-H queries. Moreover, in contrast to systems like Tupleware that mainly integrate UDFs on the LLVM level, Flare uses higher-level

knowledge about specific external systems, such as TensorFlow. Similar to Tupleware, Flare's main target are small clusters of powerful machines where faults are statistically improbable.

Query Compilation Recently, code generation for SQL queries has regained momentum. Historic efforts go back all the way to System R [14]. Query compilation can be realized using code templates e.g., Spade [64] or HIQUE [91], general purpose compilers, e.g., HyPer [107] and Hekaton [46], or DSL compiler frameworks, e.g., Legobase [85], DryadLINQ [167], DBLAB [137], and LB2 [149].

Embedded DSL Frameworks and Intermediate Languages These address the compromise between productivity and performance in writing programs that can run under diverse programming models. Voodoo [115] addresses compiling portable query plans that can run on CPUs and GPUs. Voodoo's intermediate algebra is expressive and captures hardware optimizations, e.g., multicores, SIMD, etc. Furthermore, Voodoo is used as an alternative back-end for MonetDB [28]. Delite [144], a general purpose compiler framework, implements high-performance DSLs (e.g., SQL, Machine Learning, graphs and matrices), provides parallel patterns and generates code for heterogeneous targets. The Distributed Multiloop Language (DMLL) [31] provides rich collections and parallel patterns and supports big-memory NUMA machines. Weld [114] is another recent system that aims to provide a common runtime for diverse libraries e.g., SQL and machine learning. Steno [105] performs optimizations similar to DMLL to compile LINQ queries. Furthermore, Steno uses DryadLINQ [167] runtime for distributed execution. Nagel et. al. [106] generates efficient code for LINQ queries. Weld is similar to DMLL in supporting nested parallel structures.

Performance evaluation In data analytics frameworks, performance evaluation aims to identify bottlenecks and study the parameters that impact performance the most, e.g., workload, scale-up/scale-out resources, probability of faults, etc. A recent study [110] on a single Spark cluster revealed that CPU, not I/O, is the source of bot-

tlenecks. McSherry et al. [101] proposed the COST (Configuration that Outperforms a Single Thread) metric, and showed that in many cases, single-threaded programs can outperform big data processing frameworks running on large clusters. TPC-H [155] is a decision support benchmark that consists of 22 analytical queries that address several "choke points," e.g., aggregates, large joins, arithmetic computations, etc. [27].

2.11 Conclusion

Modern data analytics need to make efficient use of modern hardware with large memory, many cores, and NUMA capabilities. We introduce Flare: a new backend for Spark that brings relational performance on par with the best SQL engines. Most importantly, all of this comes without giving up the expressiveness of Spark's high-level APIs. We believe that multi-stage APIs, in the spirit of DataFrames, and compiler systems like Flare, will play an increasingly important role in the future to satisfy the increasing demand for flexible and unified analytics with high efficiency.

3 COMBINING RELATIONAL PROCESSING AND MACHINE LEARNING FRAMEWORKS

This Chapter is based on [53]. The Lantern system was published in [159].

3.1 Introduction

Machine learning, and especially deep neural networks, have been extraordinarily successful in fields such as game play, image recognition, speech processing, etc., and are having a similar impact on business intelligence and related domains. Many data analytics applications require a combination of different programming paradigms, e.g., relational, procedural, and map-reduce-style functional processing. Productionizing ML applications and deploying them at scale often requires interfacing with big datasets stored in data management systems (DBMS) or data stores such as HDFS. For example, a machine learning (ML) application might use relational APIs for the extract, transform, load phase (ETL), and dedicated ML libraries for computations, e.g in Figure 3.1.

```
for epoch in range(100):
  for batch, labels in sql("select ... from t1 join t2 ..."):
     y = model.train(batch, labels)

sql.register_udf("model", model)
sql("select * from input where model(x, y, z, ...) ==
   Cluster1")
```

Figure 3.1.: DBMS used for data loading with ML training algorithm and model used as UDF within a SQL query.

In this chapter, we show how Flare's compilation model efficiently extends to external user-defined functions. Specifically, we discuss Flare's ability to integrate

with other frameworks and domain-specific languages, including in particular machine learning frameworks like Lantern, that target a common intermediate representation, or frameworks like TensorFlow, that provide compilation facilities of their own.

```
# Define linear classifier using TensorFlow
import tensorflow as tf
# weights from pre-trained model elided
mat = tf.constant([[...]])
bias = tf.constant([...])
def classifier(c1,c2,c3,c4):
  # compute distance
  x = tf.constant([[c1,c2,c3,c4]])
  y = tf.matmul(x, mat) + bias
  y1 = tf.session.run(y1)[0]
  return max(y1)
# Register classifier as UDF: dumps TensorFlow graph to
# a .pbtxt file, runs tf_compile to obtain .o binary file
flare.udf.register_tfcompile("classifier", classifier)
# Use compiled classifer in PySpark query with Flare:
q = spark.sql("
  select real_class,
    sum(case when class = 0 then 1 else 0 end) as class1,
    sum(case when class = 1 then 1 else 0 end) as class2,
    ... until 4 ...
  from (select real_class,
               classifier(c1,c2,c3,c4) as class from data)
  group by real_class order by real_class")
flare(q).show()
```

Figure 3.2.: Spark query using TensorFlow classifier as a UDF in Python.

The current state of affairs in ML processing represents a chasm between two classes of systems. Data management systems are highly optimized on the relational side but typically lacking in native ML support, especially regarding automatic differentiation, which is necessary for training via gradient descent. As a result, ML libraries are integrated as external user-defined functions UDFs e.g., a PyTorch or TensorFlow classifier in Spark, Flare, etc. (See Figure 3.2). The performance of ML UDFs varies based on the nature of integration with DBMS evaluation (i.e., black box entirely or with some degree of cross-optimization).

On the other hand, ML frameworks e.g., TensorFlow, PyTorch, etc. provide high-level front-ends on the top of highly-optimized back-end kernels. Programming such frameworks is heavily based on API calls. However ML frameworks are not optimized for data processing when data is local or when accessing distributed datasets.

3.2 User Defined Functions (UDF)

Spark SQL uses Scala functions, which appear as a black box to the optimizer. As mentioned in Section 2.5.2, Flare's internal code generation logic is based on LMS, which allows for multi-stage programming using `Rep` types. Extending UDF support to Flare is achieved by injecting `Rep[A] => Rep[B]` functions into DataFrames in the same way that normal `A => B` functions are injected in plain Spark. As an example, here is a UDF `sqr` that squares a given number:

```
// define and register UDF
def sqr(fc: FlareUDFContext) = { import fc._;
    (y: Rep[Int]) => y * y }
flare.udf.register("sqr", sqr)
// use UDF in query
val df = spark.sql("select ps_availqty from partsupp where
                    sqr(ps_availqty) > 100")
flare(df).show()
```

Notice that the definition of the `sqr` function uses an additional argument of type `FlareUDFContext`, from which we import overloaded operators such as `+`, `-`, `*`, etc., to work on `Rep[Int]` and other `Rep[T]` types. The staged function will become part of the code as well, and will be optimized along with the relational operations. This provides benefits for UDFs (general purpose code embedded in queries), and enables queries to be be optimized with respect to their surrounding code (e.g., queries run within a loop).

Because of that, Flare can integrate with any systems that target the LMS framework. In Chapter 5 we explore solutions to integrate with generic UDFs.

3.3 Lantern

Lantern [159] is a machine learning framework that performs automated differentiation via delimited continuations, and uses LMS to generate efficient low-level C++ and CUDA code

Machine learning relies on backpropagation to compute gradients and update model parameters. Traditional machine learning frameworks such as TensorFlow and PyTorch use auxiliary data structures (computation graphs or traces) to track forward computations for backpropagation. Lantern achieves a language-level backprogation (without auxiliary data structures) through callbacks. That is to say, each computation operation has access to a callback (also called delimited continuation, that represents the rest of the forward propagation and the beginning of the backward propagation). Each computation operation is overloaded to compute the forward computation, trigger the callback with the result of the forward computation as the argument, and compute backward computation after the callback returns. When multiple such operations are stacked together, the forward propagation happens when the stack of callbacks are triggered, and the backward computation happens when the stack of callbacks return. Generation of callbacks is further automated via delimited continuations operators `shift` and `reset`, which in addition achieves backpropagation via code transformation.

After backpropagation via delimited continuations, Lantern stages the intermediate representations and generates low level C++/CUDA code via LMS. The generated code of Lantern has comparable performance compared with TensorFlow and PyTorch [159].

3.4 Flare & Lantern

Figure 3.3 shows the integration of Flare and Lantern on the code generation level using lightweight modular staging (LMS). For ML applications, any of the ML frontends integrated with Lantern e.g. TensorFlow can be used to write ML applications.

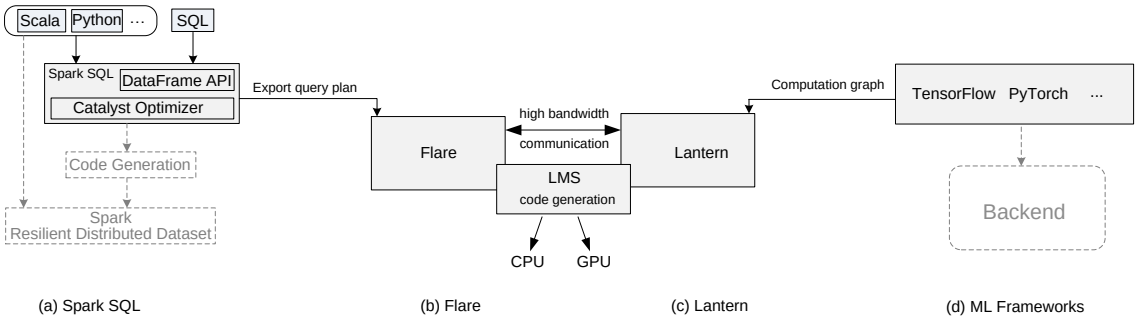


Figure 3.3.: System overview: (a) architecture of Spark [12] (b)-(c) the integration of Flare (a Spark accelerator and query compiler) with Lantern (a machine learning framework) on the code generation level using lightweight modular staging (LMS) (d) ML frameworks

It is also possible to use Lantern primitives directly. The computation graph is processed, staged and the code is generated to target the different back-end of Flare and Lantern. Thus, the code is optimized for both of data and ML processing. Moreover, SQL queries with ML UDFs are written in Spark SQL and optimized as part of Flare's evaluation [54]. Using the example in Figure 3.2, the result of the runs on multiple systems is shown in Figure 3.4.

3.5 Native UDF Example: TensorFlow

Flare has the potential to provide significant performance gains with other machine learning frameworks that generate native code. Figure 3.2 shows a PySpark SQL query which uses a UDF implemented in TensorFlow [3, 4]. This UDF performs classification via machine learning over the data, based on a pretrained model. It is important to reiterate that this UDF is seen as a black box by Spark, though in this case, it is also opaque to Flare.

Calling TensorFlow code from Spark hits a number of bottlenecks, resulting in poor performance (see Section 3.6). This is in large part due to the separate nature of the two programs; there is no inherent way to "share" data without copying back

and forth. A somewhat immediate solution is to use the Java Native Interface (JNI), which enables the use of TensorFlow's ahead-of-time (AOT) compiler, XLA [154]. This already improves performance by over 100×, but even here there is room for improvement.

Using Flare in conjunction with TensorFlow provides speedups of over 1,000,000× when compared with Spark (for concrete numbers, see Section 3.6). These gains come primarily as a result of Flare's ability to link with external C libraries. As mentioned previously, in this example, Flare is able to take advantage of XLA, whereas Spark is relegated to using TensorFlow's less efficient dynamic runtime (which executes a TensorFlow computation graph with only limited knowledge). Flare provides a function `flare.udf.register_tfcompile`, which internally creates a TensorFlow subgraph representing the UDF, saves it to a file, and then invokes TensorFlow's AOT compiler tool `tfcompile` to obtain a compiled object file, which can then be linked against the query code generated by Flare.

Finally, the TensorFlow UDF generated by XLA is pure code, i.e., it does not allocate its own memory. Instead, the caller needs to preallocate all memory which will be used by the UDF. Due to its ability to generate native code, Flare can organize its own data structures to meet TensorFlow's data requirements, and thus does not require data layout modification or extraneous copies.

3.6 Experimental Results

We evaluate the performance of Flare with TensorFlow or Lantern integration with Spark. We run the query shown in Figure 3.2, which embeds a UDF that performs classification via machine learning over the data (based on a pre-trained model). As shown in Figure 3.4, using Flare in conjunction with TensorFlow or Lantern provides speedups of over 1,000,000× when compared to PySpark, and 60× when Spark calls the TensorFlow UDF through JNI. Thus, while we can see that interfacing with an object file gives an important speed-up to Spark, the data loading ultimately becomes

the bottleneck for the system. Flare, however, can optimize the data layout to reduce the amount of data copied to the bare minimum, and eliminate essentially all of the inefficiencies on the boundary between Spark and TensorFlow.

#Data Points	Spark	Spark + JNI	Flare + TF	Flare + Lantern
200	11909	990	0.064	0.070
2000	522471	3178	0.503	0.442

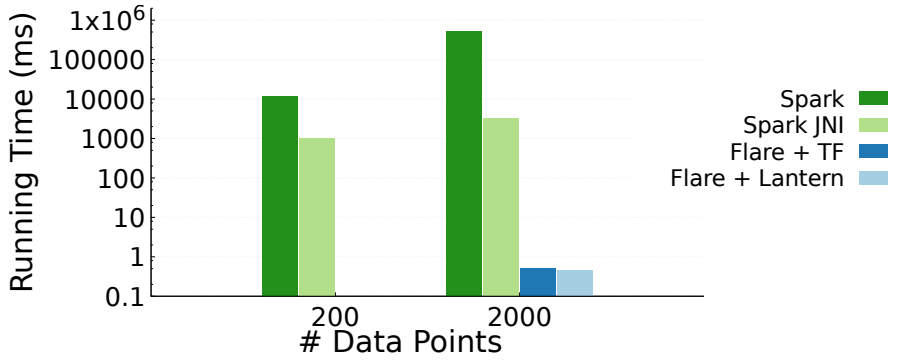


Figure 3.4.: Running time (ms) of query in Figure 3.2 using TensorFlow in Spark and Flare, and Flare with TensorFlow accelerated by Lantern.

3.7 Related Work

Delite [144] is a general purpose compiler framework, implements high-performance DSLs (e.g., SQL, Machine Learning, graphs and matrices), provides parallel patterns and generates code for heterogeneous targets. The Distributed Multiloop Language (DMLL) [31] provides rich collections and parallel patterns and supports big-memory NUMA machines.

Weld [114] is another recent system that aims to provide a common runtime for diverse libraries e.g., SQL and machine learning. Weld is using an IR similar to DMLL that support nested parallel structures. The system is optimizing externally written libraries into a common IR.

Delite and Weld are earlier approaches of integrating ML and data management. Their performances come from the analysis of the IR and loop fusion operations. The

current work is distinguished by demonstrating best of breed performance for both state of the art deep learning models e.g., SqueezeNet and relational benchmarks e.g., TPC-H. In addition, Delite and Weld differ from ours by their multi-pass compilation process. Flare and Lantern emit directly the same IR that can then be optimized in a single pass.

3.8 Conclusion

Modern data analytics need to combine multiple programming models. We illustrated how Flare enables highly optimized heterogeneous workloads with external ML systems, such as Lantern and TensorFlow.

4 ON-STACK REPLACEMENT FOR PROGRAM GENERATORS AND SOURCE-TO-SOURCE COMPILERS

This Chapter is based on a technical report [55].

In Chapter 2, we accelerate systems such as Spark by introducing a compilation step to specialized low-level C code. This process produces very efficient code for SQL-like workloads and offers order of magnitude speed-ups. However, there are situations where the additional compilation step represents a significant part of the total run time. In particular, workloads with short running times (around hundreds of milliseconds) do not benefit from the compilation step as the fraction of time spent in the compilation step is too significant.

In this Chapter, we present a technique that reduces the overhead introduced by the compilation of the low-level code. We introduce generic on-stack-replacement techniques for code generators that open the door to an unexplored space of optimizations and programming techniques.

4.1 Introduction

The idea of on-stack replacement (OSR) is to replace currently executing code on the fly with a different version. There are two main motivations why one would want to do that: *tiered execution* and *speculative optimization*.

Tiered Execution OSR was pioneered in just-in-time (JIT) compilers, concretely in the SELF VM [75] in the early 1990s. Various forms of dynamic compilation were known before. JIT compilers of the zeroth generation compiled whole programs during loading, or individual methods the first time they were called. For large programs, this leads to high overheads in compilation time. Many methods are used only rarely,

so compilation does not pay off. First-generation JIT compilers improved on this model by splitting execution into two tiers: code starts out running in an interpreter, which counts invocations per method and triggers compilation only after a threshold of n calls. While this refined model is effective in focusing compiling efforts on hot methods, it can only switch from interpreted mode to compiled mode on method calls. But not in the middle of long-running methods, i.e., methods that contain loops. As an extreme case, a program consisting of a single main method that runs a billion loop iterations will never be able to profit from compilation. With OSR, however, one can count loop iterations, trigger background compilation once the loop becomes hot, and switch from interpreted to compiled code *in the middle of the running loop*, while the method is still running.

Taking a more general view, code can be executed using a range of interpreters or compilers that make different trade-offs in the spectrum of compilation time vs. running time, i.e., optimization effort vs optimization quality. A typical set up is a low-level interpreter for fast startup, then a simple compiler, then a compiler with more aggressive optimizations.

Speculative Optimization and Deoptimization A compiler can more aggressively optimize code if it is allowed to make some optimistic assumptions. However, if assumptions are violated, a fail-safe mechanism needs to be used that allows the system to *deoptimize*. The overall premise for this technique is that deoptimization cases are infrequent enough so that the incurred overhead is outweighed by the performance gained on the fast path. As a concrete example, Java JIT compilers typically make speculative decisions based on the currently loaded class hierarchy. In particular, if a method defined in a certain class is never overridden in a subclass, all calls to this method can be *devirtualized* since the precise call target is known. Based on the known call target, the compiler can further decide to inline the method. However, if a newly loaded class *does* override the method, this violates the original speculative assumption and all the optimizations that were based on this assumption need to

be rolled back. In this particular example, all running instances of the optimized code have to be aborted since continuing under wrong assumptions about the class hierarchy would be incorrect. In other cases, there is a bit more flexibility when to deoptimize. For example, type feedback with polymorphic inline caches (PIC) [72] caches a limited number of call targets per call site. PIC misses do not necessarily mean that execution of compiled code needs to abort immediately, but deoptimization and recompilation are typically triggered after a certain threshold of misses. It is also interesting to note that PIC schemes benefit from processor-level branch prediction, which can be seen as a low-level speculative optimization approach implemented in silicon.

Today, all cutting edge VMs and JIT compilers (HotSpot, Graal for Java; SpiderMonkey, V8, JSC for JavaScript) support both forms of OSR, tiered execution and speculative optimization with deoptimization. Following recent related work [44, 92], we use the term "OSR" symmetrically to refer to both optimizing and deoptimizing transitions; older work does not recognize deoptimization as a form of OSR.

Generative Programming Generative programming is often used to implement specialized code generation facilities in ways that are out of reach for automatic JIT compilers. While in many cases code is generated and compiled offline and then available for future use, there are also important use-cases where code is generated and compiled on the fly, and then ran once and discarded. This means that the compilation process is part of the runtime of the service, much like zeroth generation JIT compilers. Therefore it seems natural to look into techniques from the JIT compiler and VM space to improve performance. As a key motivating use case for this Chapter, we consider main-memory data processing frameworks such as Flare, and state-of-the-art query compilers based on generative programming techniques [54, 149]. The embedded code generators in such systems often emit C source code for debuggability, portability, and to benefit from the best compiler for a given hardware platform (e.g., Intel's ICC for x86 processors). As we demonstrate in this Chapter, tiered execution

can improve performance for workloads that execute many complex, but quick queries — a scenario that has been identified as key challenge by the database community [87] — through a simpler but faster compiler. In addition, speculative code generation can help the downstream C compiler generate more efficient executables on specialized code paths (e.g., using vectorization). What is needed are OSR techniques adapted to this setting of explicit code generation, that are generic and easy to use but allow the generation of efficient code.

Liberating OSR from Low-Level VMs How can we bring the benefits of OSR to this setting? That is the question we address in this Chapter! The first and obvious idea would be to turn systems like main-memory databases into full-blown VMs. But often that is not practicable. First, implementing all the necessary infrastructure, including a bytecode language and facilities like a high-performance low-level interpreter to deoptimize, represents a huge engineering effort on an aspect that is not the main purpose of the system, and requires deep expertise in areas unfamiliar to database developers. Second, generating structured source code is often important, for optimization (no irreducible control flow) and for debuggability. In addition, there are cases where a given platform dictates a certain target language. For example, such external constraints may require generating Java or JavaScript source for interoperability.

We follow recent work, in particular D'Elia and Demetrescu [44], in viewing OSR as a general way to transfer execution between related program versions, articulated in their vision to "pave the road to unprecedented applications [of OSR] that stretch beyond VMs". Or, in the words of Flückiger et al. [60]: "Speculative optimization gives rise to a large and multi-dimensional design space that lies mostly unexplored". By making speculative optimization meta-programmable and integrating it with generative programming toolkits, in particular LMS (Lightweight Modular Staging) [130], we give programmers a way to explore new uses of speculative optimization without needing to hack on a complex and low-level VM.

Key Ideas Our approach is based on two key ideas. First, we view OSR as a program transformation reminiscent of a data-dependent variant of loop unswitching. We interrupt a loop at the granularity of one or a handful of loop iterations, switch to a different compiled loop implementation and resume at the same iteration count.

The second idea is to treat loops as top-level compilation units. To enable separate compilation for either tiered execution or for speculation with dynamically generated variants, loops, or loop nests, have to become their own units of compilation. This can be achieved elegantly using a form of lambda lifting, i.e., representing the loop body as a function and making the function top-level by turning all the free variables of the loop into parameters of the extracted function. Using the semantics of functions in the target language allows the framework to guarantee correctness without worrying about difficult low-level details such as saving and restoring registers. Furthermore, each compilation unit will have fewer data paths than the original loop, which allows for more aggressive compiler optimizations. It is also important to note that intraprocedural optimizations can be applied before the transformation.

4.2 A Simple Source-to-Source Model of OSR

Let us consider the following simple Scala program, which computes the dot product of two vectors, given as `Float` arrays `x` and `y` along with their size `n`:

```
var res = 0.0f
for (i <- 0 until n) {
  res += x(i) * y(i)
}
println(res)
```

By default, Scala compiles to JVM bytecode, and the JVM's JIT compiler will naturally support a variety of OSR techniques to optimize this code at runtime. But what if we wanted to cross-compile this piece of Scala code to C? Let us first desugar the code into a plain while loop:

```
var res = 0.0f
var i = 0
while (i < n) {
  res += x(i) * y(i)
  i += 1
}
println(res)
```

Now we can transform it into an equivalent OSR-enabled program following the ideas described in the introduction. The `while` loop is lifted into a separate function `loop1`. In addition, there is another function `loop2`, which on the surface is equivalent to `loop1`. But the assumption here is that loop2 is more optimized that loop1, and takes longer to compile. While ignoring for now how the runtime compiles and loads the different pieces of code (there is a mutable variable `loop` that holds a pointer to the currently execution loop body, and at one point `loop2` is going to be assigned to the variable `loop`), we can see that this code is equivalent to the original loop, but may switch between semantically equivalent loop implementations at any time during the execution of the loop.

```
var res = 0.0f
var i = 0
var loop = loop2
def loop1() = {
  while (loop == loop1) {
    if (i >= n) return DONE
    res += x(i) * y(i)
    i += 1
  }
  NOT_DONE
}
```

```
def loop2() = {
  while (loop == loop2) {
    if (i >= n) return DONE
    e2
  }
  NOT_DONE
}

while (loop() != DONE) {}
println(res)
```

In a further lowering step, we can perform lambda lifting to close off and un-nest each of loop functions. A direct mapping to top-level functions in C code is now straightforward. Importantly, each loop function can even be placed in a separate file and compiled independently of the others:

```
// Ref is used to box the value
def loop1(loop: Ref[Func], i: Ref[Int], res: Ref[Float], n: Int, x:
    Array[Float], y: Array[Float]): Int = ...
def loop2(loop: Ref[Func], i: Ref[Int], res: Ref[Float], n: Int, x:
    Array[Float], y: Array[Float]): Int = ...
def main() {
  ...
```

```
  val i = Ref(0); val res = Ref(0.0)
  loop = loop1
  while (loop(loop, i, res, n, x, y) != DONE) {}
  println(res)
}
```

As we will show, based on this simple pattern, we can realize a variety of practically relevant OSR patterns, including lazy tiered compilation (recompile on-demand using a more optimization compilers) and various forms of speculative optimization with deoptimization.

A Generic OSR Transformation Based on these examples, we can describe a generic OSR template in a slightly more formal way. We again consider a suitable subset of Scala as our object language and focus on the representation of while loops. In addition, we introduce an oracle operator *select*. Given a sequence of conditions $< c_i >_n \in \text{Exp}^n$ (where Exp is the syntactic category of program expressions) and a sequence of statements $< t_i >_n \in \text{Exp}^n$, the term $\text{select}(< c_i >_n)\{< t_i >_n\}$ executes a statement t_i for an i such that c_i is true. In addition, *select* returns the result of the evaluation of the statement t_i.

We propose a generic OSR transformation $[\![\, . \,]\!] \setminus < (c_i, f_i) >_n$, where $< (c_i, f_i) >_n \in (\text{Exp} \times \text{Exp} \mapsto \text{Exp})^n$, meeting the following conditions:

- For all i, the statement $f_i(e)$ is equivalent to e under the condition that c_i is true.
- At any point in the program, there is a i such that c_i is true.

With that, the generic OSR transformation is:

$[\![$ `while` $(c)\ e$ $]\!] \setminus < (c_i, f_i) >_n$

```
      = while (c) select (< c_i >_n) { < f_i(e) >_n }
      = while (select (< c_i >_n) { <while (c_i) {
                  if (!c) return DONE; f_i(e)>_n }; NOT_DONE \
        } != DONE) {}
      = def loop_1() = {
          while (c_1) { if (!c) return DONE; f_1(e) }
          NOT_DONE
        }
        ...
        while (select (< c_i >_n) { <loop_i()>_n } != DONE) {}
```

The oracle *select* represents the runtime selection of the code that should be run. In addition, *select* hides the potential compilation and loading of different pieces of code. For each of the OSR situations described earlier, tiered execution and speculation, the transformation sequence $< (c_i, f_i) >_n$ has different characteristics.

We can tie the formalism back to our example: f_1 is the identity function, $c_1 = \text{loop} == \text{loop}_1$, $f_2(e) = e_2$, and $c_2 = \text{loop} == \text{loop}_2$. The only unknown left is how the loop variable is assigned to loop_2. For tiered execution, the transformations f_i are simply the identity function. For any i, the c_i is evaluated to true if the compilation process has terminated for loop_i. For speculation, the transformations f_i can be different in arbitrary ways. In our evaluation (Section 4.6), we will look at two different kinds of speculation. In the first situation, the different transformations make optimistic assumptions about the data handled by the program (e.g., that all values are positive). In this setting, $f_i(e)$ is more optimized than $f_{i+1}(e)$, and c_i is true as long as the assumptions made for the f_i transformation are valid. But once the assumption is invalidated, it can never become valid again. In the second situation, the conditions can be invalidated and be true again later. The conditions c_i are evaluated, like a heuristic, based on data collected during the execution of the program. In this setting, each transformation is more fitted for a kind of data pattern, and the OSR pattern allows the code to adapt to the best possible version.

Limitations We focus on structured loop nests only and do not consider arbitrary recursive functions. In the setting of generative programming and explicit program generation, this is a very sensible choice, as performance-sensitive code tends to be dominated by such coarse-grained loop nests and fine-grained recursion is generally avoided for performance reasons. Moreover, dealing with loops within a function really is the core problem addressed by OSR. With fine-grained recursion, methods are entered and exited all the time so in many cases, code can be fruitfully replaced on a per-method boundary and new invocations will pick it up.

4.3 Metaprogramming Facilities for OSR

The high-level transformations introduced in Section 4.2 can be realized in any suitable way, including directly through the use of a syntactic rewriting framework. In practice, an attractive way to build a (simple) compiler is by programmatically specializing an interpreter using a form of multi-stage programming (staging) [148]. The fact that specializing an interpreter to a given input program yields a compiled version of this program is known as the first Futamura projection [63].

In the remainder of this chapter, we will use Scala as the metaprogramming language, and LMS (Lightweight Modular Staging) as our multi-stage programming framework.

Throughout the rest of the chapter, Scala code examples represent the API that our system provides, as well as the code that a programmer would have to write to use our system. The C code examples represent the generated code that will be compiled and executed, and contains the OSR runtime.

4.3.1 Tiered Execution

High Level Interface As mentioned previously, we aim at providing programmers an easy interface for OSR. In the case of tiered execution, different compilers are used to compile the same piece of code. As a result, the code generator does not require additional information: the programmer must simply mark the loop as an OSR loop. Using LMS and Scala, the interface for programmers to emit a staged `while` loop is given by the following function (note that `=>` signifies a *by-name* parameter in Scala):

```
def __while(cond: => Rep[Boolean])(body: => Rep[Unit]): Rep[Unit]
```

In exactly the same way, we can propose a very simple interface that emits OSR-enabled `while` loops, requiring a minimal amount of changes from the programmer:

```
val validOSR: Boolean
def tieredExecutionOSRWhile(cond: => Rep[Boolean], body: =>
    Rep[Unit]): Rep[Unit] // Provided by our framework
def whileOSR(cond: => Rep[Boolean])(body: => Rep[Unit]): Rep[Unit]
    = {
  if (validOSR) // Verify that OSR can/should be used.
```

```
    tieredExecutionOSRWhile(cond, body)
  else
    while (cond) body // Invokes __while internally
}
```

The Scala syntax allows the `whileOSR` loop to have almost the same syntax as the regular one, which makes its usage virtually transparent to the programmer. The `validOSR` flag is of type `Boolean`; this branch is therefore executed during the code generation phase. When the flag is true, the `tieredExecutionOSRWhile` function is invoked. The function generates the necessary code equivalent to the given `while` loop with the ability to swap to a more optimize version. Otherwise, it calls a regular LMS `while`, which generates a regular loop. It means that through a simple flag, the program is able to toggle OSR; while also having the possibility to use `while` explicitly when a loop should never be transformed. It is of course not always the case that loops will benefit from an OSR transformation. But based on this simple API, programmers can easily build more complex heuristics when necessary. For example, we can use a more complex heuristics than a simple flag by transforming the `validOSR` flag into a function that returns a `Boolean`. Possible heuristics could be to only transform the first loop, or only the top level ones, which would thus disable nested OSR loops. With this interface, we can bring OSR to any program by changing each `while` loop, into a `whileOSR`.

Generated Code The code generator needs to generate the low-level details to support the OSR semantics. As discussed earlier, the loop needs to become its own compilation unit: a single function, potentially in a separate file. The dot product example we used in Section 4.2, would be generated as follows:

```
typedef int
   (function_type*)(int*,
   float*, int,
   float*, float*)
function_type* loaded_osr = NULL;
int osr_region1(int* i_p, float*
   res_p, int n,
   float* x, float* y) {
  int i = *i_p;
  float res = *res_p;
  int eCode = NOT_DONE;
  while (loaded_osr == NULL) {
    if (!(i < n)) { eCode =
       DONE; break; }
    res += x[i] * y[i];
    i++;
```

```
  }
  *i_p = i;
  *res_p = res;
  return eCode;
}
```

```
int main() {
  float res = 0; int i = 0;
  ...
  loaded_osr = osr_region1;
  while (loaded_osr(&i, ...) !=
     DONE);
  printf("%.f\n", res)
}
```

4.3.2 Speculative Optimization

In the case of speculative optimization, there exists a wider range of possibilities than in tiered optimization. We distinguish between low-level and high-level speculative optimizations.

Low-level Speculative Optimization is compiling the same code with different compiler configurations to produce different binaries. All generated binaries produce the correct result; however, their performance may vary depending on the input data (e.g., one version may be more efficient on negative values). Pieces of code with a simple loop traversing an array contiguously aggregating the positive values are a perfect example to illustrate this:

```
float* dist = ...;
float agg = 0;
for (int i = 0; i < N; ++i) {
   if (dist[i] > 0) agg += dist[i];
}
```

This code snippet can be vectorized using Single Instruction Multiple Data (SIMD) instructions. This means that while the code generated can perform four floating points operations in parallel (with SSE 128bits width registers, up to 8 or 16 with AVX2 and AVX-512), some may be executed, but then discarded if the condition does not hold. The vectorized code always has the same performance no matter the selectivity value of the condition. In general, this is very beneficial, as the computation may complete in a quarter of the time. However, in the case of very low selectivity, the non-vectorized code would perform much better. Indeed, at the limit, if the selectivity is 0%, the branch predictor performs perfectly well and no useless computations are

executed. However, at selectivity 100%, vectorized code is exactly 4 times faster than the non-vectorized code. Therefore, there exists a selectivity S such that vectorized code is more efficient for selectivity greater than S, and the opposite is true for the selectivities lower than S. This selectivity S depends on the cost of the operations: the simpler the code, the more it benefits from vectorization and the lower the S value is. The ideal solution is to use the more suitable code based on the selectivity of the condition. However, the selectivity is data-dependent and non-uniform within the array: for example, the selectivity of the condition in our example for this array `[-1, -1, -1, 1, 1, 1]` is 50%, but it is 0% for the first half, and 100% for the second. This means that the correct version must be selected at runtime. In order to provide an abstract interface for this situation, we need to be able to define different configurations and the conditions in which each configuration should be used. We propose a generic interface in Figure 4.1 that allows the programmer to create a loop to be compiled with different configurations, as well as a heuristic to swap between them.

```
type Loop = Rep[Boolean] => Rep[Unit] => Unit
def lowLevelSpeculation(select: => Configuration)
    (loop: Loop => Unit): Unit

// Example
var i: Rep[Int] = 0
var hit: Rep[Int] = 0
val arr: Rep[Array[Float]] = ...
lowLevelSpeculation { if (hit < CUTOFF * i)  // selection heuristic
    Flag("no-tree-vectorize") else Flag("tree-vectorize")
{
   whileSpec =>   // custom while loop to use for the speculative
      work

    whileSpec (i < n) {
      if (arr[i] > 0) { agg += arr[i]; hit += 1 }
      i += 1
    }
}
```

Figure 4.1.: API interface for low level speculative loops on a simple example.

The different configurations are expressed through an abstract `Configuration` class that contains all the information necessary for the compilation. The selection

function `select` is used to pick the configuration that will be used for the following iterations. In our example, we use a single variable for gathering the necessary information for the configuration selection, but the function can be arbitrarily complex.

Similarly to the tiered execution situation, the `whileSpec` loop is transformed into a function. However, in the case of low-level optimization, the generated code cannot check the swapping condition at each iteration. Indeed, if it was the case, the loop could not be vectorized, as it would add a data-dependent exit condition. The idea, then, is to split the workload into segments, and check the switching condition after a full segment has been executed. Once the next configuration is required, the compilation can be started and the execution will jump to it once it becomes available. Figure 4.2 shows a sketch of the generated code.

```
int hit = 0;
int i = 0;
int next = MIN(n, INTERVAL);
while (i < n) {
  int cond = hit < CUTOFF * i;
  if (cond) ... // trigger compilation
  if (osr_region1_nonvect != NULL && cond) {
     // Compiled with -fno-tree-vectorize
     osr_region1_nonvect(i, arr, next, &hit);
  } else {
     // Compiled with -ftree-vectorize
     osr_region1(i, arr, next, &hit);
  }
  i = next;
  next = MIN(n, n + INTERVAL);
}

typedef void (loop*)(int, float*, int, int*);

volatile loop osr_region1_nonvect = NULL;

void osr_region1(int i, float* arr, int n, int* hit_p) {
  int hit = *hit_p;
  for (; i < n; ++i) {
    if (arr[i] > 0) { agg += arr[i]; hit += 1 }
  }
  *hit_p = hit;
}
```

Figure 4.2.: Code generated for the code example in Figure 4.1.

High-level Speculative Optimization is the process of generating different pieces of code for the same task: for each piece of code, some optimistic assumptions that allows the generation of more efficient code are made. One piece of code (the last) is required to be generic, and work without any assumptions. In the situation where the assumption is violated, the program must recover and fall back to the next available segment of code.

We next seek to explain why speculation is highly effective. If the control flow within a loop is more complex than a single loop, compilers have difficulty applying optimizations. However, a compiler could optimize the code if the different paths were within their own compilation units. For example, in the program represented in the right hand part of Figure 4.3, the condition `arr[i] < 0` can be speculated to always evaluate to `true`; the whole of Figure 4.3 represents the program with this speculative optimization applied. The residual program can now potentially be vectorized, and in the event this speculation is correct, the performance would be greatly improved. There are many situations in which the programmer may have good reason to speculate that a condition will likely evaluate to `false`, such as an error checking branch. While the code is required when such an error occurs, the additional code may prevent the compiler from applying some optimizations. Using our high-level speculation interface would provide the opportunity to fully optimize the useful parts of the code, without being blocked by edge cases.

We provide an examination of this situation in our evaluation in Section 4.6.

4.3.3 Implementation Details

In this section, we will use the following example to illustrate the different technical details by focusing on the tiered execution situation. While the program is not representative of a real workload, it *is* representative of a large class of loops. The condition and the body have side-effects, which help highlight the small details that must be considered.

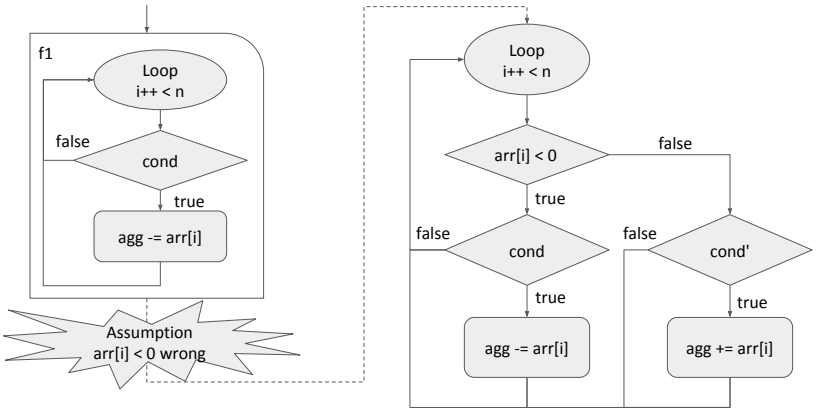


Figure 4.3.: Complex control flow for loop vectorization. Multiple nested branches (right) prevent the compiler from vectorizing the computation. Extracting the hot path with a side exit (left) enables better optimization in general and especially vectorization of the hot path.

```
var i = 0;
while (i++ < n) {
  printf("%d\n", i-1);
}
```

Dynamic Code Loading The implementation of OSR requires some runtime support to be embedded within the generated code in order to load the code which has been dynamically compiled. In the general setting, we assume that there are `N` OSR regions, and they can be compiled using `M` different compilers or compiler configurations. In the case of tiered execution, we also assume that the $(i+1)^{th}$ configuration is *better* than the i^{th} configuration. Therefore, the runtime can prioritize the highest configuration available at any given time by loading the newest available code.

Most languages have support for dynamic loading through library support; therefore the challenges lay on notifying the running program of a newly available code or starting the compilation of a required piece of code.

For the first challenge, there are several possibilities that can be considered; we examine two of them here. The first is to have a polling option. In this option, the worker thread periodically checks if a new version is available. A second possibility is to block until the next version is ready, in which case it is necessary to use an

auxiliary thread that is in charge of the loading step. We present the architectures of these two ideas in Figure 4.4. Theoretically, the background thread option wastes fewer resources as it is mainly waiting on a blocking event while the worker thread only has to check a single condition at each iteration. The polling version, however, must pay the (usually more expensive) cost of polling. If this was done at each loop iteration, the overhead would be too high, with little benefit. Our solution is to check only after `X` iterations; however, the less optimized code may keep running while a new, faster version is available if `X` is too large. In that case, the performance gain is a trade-off between the polling overhead and the waste of using less optimal code.

For the second challenge, our design gives the flexibility to compile the code when it is the most efficient, without adding extra overhead in the computation thread. In the case of tiered execution for fast startup, the fast and slow compilers can start the compilation process at the same time: the code generated by the fast compiler will be executed until the slow compiler terminates and the highest optimized code is available. In the case of speculative optimization, the compilation needs to be triggered based on given criteria, e.g., low-selectivity of a `for-if` construct vectorizing the code would be beneficial. This avoids the waste of resources if the compilation is not necessary.

We evaluate these differences in Section 4.5 and 4.6.

Compensation Code At each boundary of the OSR regions, some extra code needs to be inserted to handle transitions. It is important to structure the code so that the downstream compiler can still fully optimize the code. If some variables are mutated within the loop, they need to be passed as pointers. Because of aliasing issues, the compiler may not be able to apply all optimizations. The idea is to dereference the pointer once when entering the region, save the value into a local variable, execute the region using those variables, and finally assign the current value back to the pointer when exiting the region (See Figure 4.5). In addition, it may be necessary to give more information to the compiler on the aliasing and the alignment of the pointers, as

```
typedef int (osr_regionX_t)(...);
void* loaded_osr_region[M];
volatile int prev_v[M] = {-1 };

// launched as a separate thread
void* load_shared(void* args) {
  for (;;) {
    // Wait for a region to be compiled.
    int osr_reg, version;
    char path[40], name[20];
    wait_until_ready(&osr_reg,
            &version, path, name);

    if (version <= prev_v[osr_reg])
      continue;

    void* plugin = dlopen(path);
    loaded_osr_region[osr_reg] =
        dlsym(plugin, name)

    prev_v[osr_reg] = version;
  }
  return NULL;
}
int osr_regionX(int cur_v, ...) {
  ...
  while (true) {
    if (prev_v[X] > cur_v) {
      ... // Save state
      return NOT_DONE;
    }
    ...
} }
```

(a) Background thread

```
typedef int (osr_regionX_t)(...);
osr_regionX_t loaded_osr_regionX;

int osr_regionX(int cur_v, ...) {
  int test = SWITCH_THRESHOLD;
  int version = cur_v;
  char path[40], name[20];
  ...
  while (1) {
    if (--test == 0) {
      if (poll_osr_regionX(&version, path)
          && version > cur_v) {
        ... // Save state
        void* plugin = dlopen(path);
        loaded_ors_regionX =
            dlsym(plugin, "osr_regionX")
        return NOT_DONE;
      }
      test = SWITCH_THRESHOLD;
    }
    ...
  }
}
```

(b) Polling

```
// Calling site
if (osr_regionX(...) != DONE) {
  while (loaded_osr_regionX(...)
      != DONE);
}
```

(c) Initiation of the loop (for both loading technique)

Figure 4.4.: Different strategies to test if a newest version is available and load it for tiered execution. (a) is using a background thread that blocks until the compilation process notify it. It then loads the newly compiled version. (b) is using a polling method and check every `SWITCH_THRESHOLD` if a newer version is available, if yes it loads it and return.

they are known by the code generator. Generating code with `__restricted__` or `aligned(X)` annotations may be required. This pattern works perfectly for `loop`-tiered execution and low-level speculative optimization.

In the case of high-level speculative optimizations, the data layout between regions may be arbitrarily different; the compensation code would potentially have to transform the already computed data. For example, when using a hashmap, one can speculate that the keys will only be an integer from 0 to 10, and thus use an array instead of a generic hashmap. If the assumption fails, the next OSR region may use

```
int osr_region1(int* i_p, int n) {
  int i = *i_p;
  while (i++ < n) {
    if (loaded[0]) {
      *i_p = i;
      return NOT_DONE;
    }
    printf("%d, ", i-1);
  }
  *i_p = i;
  return DONE;
}

// main
int i = 0;
if (loaded[0] || osr_region1(&i,
   n) != DONE) {
  loaded_osr[0](&i, n);
}
// output if swap arise after
   iteration i = 3
0, 1, 2, 3, 5, 6, 7, ..., `n`
// output if swap arise after
   iteration i = n - 1
0, ..., `n - 1`, `n + 1`, ... //
   miss stop condition
```

(a) Incorrect

```
int osr_region1(int* i_p, int n) {
  int i = *i_p;
  while (!loaded[0]) {
    if (i++ >= n) {
      *i_p = i;
      return DONE;
    }
    printf("%d\n", i-1);
  }
  *i_p = i;
  return NOT_DONE;
}

// main
int i = 0;
if (loaded[0] || osr_region1(&i,
   n) != DONE) {
  loaded_osr[0](&i, n);
}
// output if swap arise after
   iteration i = 3
0, 1, 2, 3, 4, 5, 6, ..., `n`
// output if swap arise after
   iteration i = n - 1
0, ..., `n`
```

(b) Correct

Figure 4.5.: Illustration on the importance of the check order. If the loop condition is carried out before the OSR check, the condition would be executed an extra time when the following OSR region start. It therefore break the language semantics. However doing the OSR check first ensures the correctness.

a generic hashmap; when transitioning, all keys already inserted in the array must be correctly inserted in the hashmap; thus requiring a task-specific compensation code.

Correctness The OSR transformations must preserve the semantics of the original loop. For tiered execution, we can ensure this if we always test the switching condition at the beginning of the loop **before** the loop conditions. Indeed, if the loop condition has side-effects, exiting the loop after executing it and starting a new loop would lead to an incorrect transformation. With this constraint, we ensure that a loop iteration (condition and body) execute completely, or not at all. We show how the checking order can lead to invalid code transformation in Figure 4.5.

For speculative optimizations, the condition to switch to different code can be arbitrarily complex, and depends on the kind of speculation that is made. Such an OSR transformation would be correct if and only if upon an aborted iteration, the program can not have had observable effects; otherwise they need to be rolled back. Whereas it is not possible to define a generic model that would be correct in all situations, a windowed slicing pattern can be used. At the beginning of the loop, the state of the program is saved. Once part of the loop has been executed, the program can check for errors. If there are any, the previous state can be restored, and the OSR region can exit; the next region can then execute the loop from the saved state. This pattern does not work without additional precaution for I/O operations. It is also interesting to note that in a multi-threaded program, restoring the state may not be enough, as another thread may already have observed the modifications that have then been rolled back.

Initialization In order to be as efficient as possible, OSR code needs to jump to the best available code as soon as possible. However, there are some situations where it does not happen. For example, if the code has a lengthy initialization, the compilation of the optimal version may be done. However, as the slower code has not yet reached the OSR region, the swap will arise very late, thus negating the advantage of using OSR altogether. In order to maximize performance, it is important that the initialization step is made of code that is optimally compiled by the fast compiler. For example, it can be made of function calls to libraries which are precompiled with high optimization settings.

Nested Loops In line with the previous paragraph, the case of nested loops can lead to unwanted overhead. Assuming that an outer loop is transformed into an OSR region, and the inner loop is taking more time than the compilation process, there will be a long period between the end of the compilation and the starting of the execution of the fast code. In that situation, it may be preferable to transform the

inner loop into an OSR region. This example shows that it may not always be the most beneficial to transform the outer loop in an OSR region.

On a technical point of view, however, a nested loop does not make the transformation more complex or invalid. The impact on the performance, however, needs to be evaluated. Therefore, leaving the decision up to the programmer is the most logical sense in our situation. Lameed and Hendren [92] discuss the impact of different approaches.

4.4 Case study: Compiling SQL to C

We apply the ideas and tools presented in the preceding sections to a real-world case study, adding OSR to a state-of-the art SQL to C compiler, Flare. While compilation for SQL queries has been an active topic of research for a number of years, optimizing for fast startup and for workloads that execute many complex, but short running, queries has only recently been identified as important challenge by the database community [87]. As we will show, the execution patterns of the SQL language makes it a perfect candidate for OSR compilation.

We show key parts of a simple SQL compiler in Figure 4.6. This code is taken from Rompf and Amin [128] and the complete version, including various join operators and optimized data structures, is implemented in fewer than 500 lines using Scala and LMS.

Flare that forms the basis of our experiments is essentially a scaled-up version of this code. With Flare, we show how generative programming using LMS can be used to design a query compiler by implementing a simple staged interpreter. Interpreters have the advantage of being relatively simple, and can easily be understood by programmers with different backgrounds, even those who are not compiler experts. For example, in Figure 4.6, the function `hasNext` returns a `Rep[Boolean]`; the `while` statement in the `Scan` case will therefore be generated. As such, our interpreter will actually become a code generator.

```
abstract class Op
case class Scan(stream: Stream) extends Op
case class Print(par: Op) extends Op
case class Project(out: Schema, par: Op) extends Op
case class Filter(pred: Predicate, par: Op) extends Op
case class Join(key: Schema,
                  left: Op, right: Op) extends Op

def exec(op: Op)(cb: Record => Unit) = op match {
  case Scan(stream) =>
    while (stream.hasNext) {
      cb(stream.next)
    }
  case Print(par) => exec(parent) { rec =>
    rec.print
  }
  case Filter(pred, par) => exec(par) { rec =>
    if (execPredicate(pred)(rec)) cb(rec)
  }
  case Project(out, par) => exec(par) { rec =>
    rec(out)
  }
  case Join(key, left, right) =>
    val hm = new MultiMap(key, schemaOut(left))

    exec(left) { rec =>
      hm += (rec(key), rec)
    }

    exec(right) { rRec =>
      for (lRec <- hm(rRec(key))) {
        cb(lRec ++ rRec)
      }
    }
}
```

Figure 4.6.: Implementation of relational query engine DSL with the corresponding staged interpreter (`exec`).

Here is a simple query to scan a single table:

```
select * from tweets                    Print(Scan(stream("tweets")))
```

Using Flare (and knowing that the CSV format is "user name, # of likes, tweet content"), this query would be translated to C as shown in Figure 4.7.

```
int main() {
  char* tweets = ...; // open file and mmap it
  int fileLength = ...;
  int pos = 0;
  while (pos < fileLength) {
    int userLength; char* user;
    pos += parseString(tweets + pos, &user, &userLength) + 1;
    int nbLikes;
    pos += parseInt(tweets + pos, &likes) + 1;
    int tweetLength; char* tweet;
    pos += parseString(tweets + pos, &tweet, &tweetLength) + 1;
    printf("%.*s, %d, %.*s\n", userLength, user, likes,
       tweetLength, tweet);
}
```

Figure 4.7.: Code generated for simple query.

Going further, if we add a filtering operation to only display the name and the number of likes:

```
select user, likes from tweets where likes >= 1000
```

The query may be expressed as

```
Print(Project(Seq("user", "likes"), Filter(Geq(Field("likes"),
   Const(1000)), Scan(stream("tweets")))))
```

in the language of relational query operators. For this query, the code generator produces nearly the same code, but rather than printing the whole tuple, it generates code that prints only the expected values, with a simple conditional statement around it:

```
if (likes >= 1000)
  printf("%.*s, %d\n", userLength, user, likes);
```

Here, we can see that the code is comprised of a single while loop that goes through the `tweet` table. This pattern is a key characteristic of code generated for SQL queries. Even in complex queries which include aggregates or joins, the code is

composed of top-level while loops scanning through collections (e.g., tables or data-structures). These loops are called "operator pipelines" [107]; operators such as `aggregate` or `join` are called "pipeline breakers," as they must materialize the tuple of a pipeline and store it in a temporary data-structure, and then produce their result within another pipeline. Figure 4.9 (a)-(c) shows the overall shape of the code with a `Join` operator. This pattern of code is a perfect example where transforming the pipeline data into OSR regions would be beneficial. In our work, an OSR region is a *loop* that will potentially have multiple instructions at runtime, as presented above.

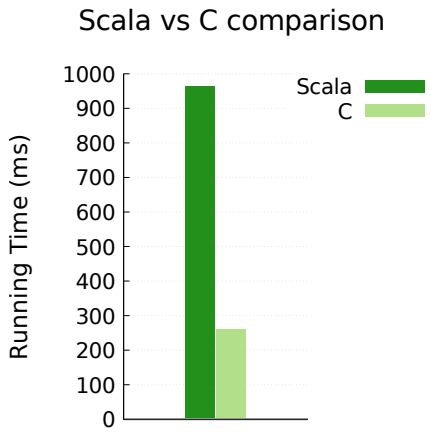


Figure 4.8.: Runtime of the query in Figure 4.9 (a) with different target languages.

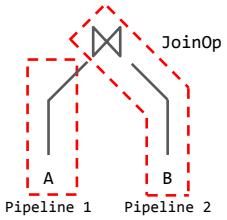
In looking to implement OSR for this style of code, it is instructive to first generate code that is executed on a virtual machine which already possesses all the JIT mechanisms required. We use the example found in Figure 4.9 and generate code to execute the query in Scala. The generation process specializes the data structures (e.g MultiMap) to a very efficient low-level implementation on arrays. The `lineitem` table contains 60k records (a CSV file of $\sim$ 7MB) and the `part` table 2k records (a CSV file of $\sim$ 253 kB). The generated Scala code is compiled and run in 963ms. If we instead generate C code that does not use any OSR constructs, the code is compiled and run in only 260ms (see Figure 4.8). This demonstrates that Scala's JIT compilation was unable to bring the Scala code to the same performance as that of the C

code. In part, this is due to the fact that the I/O operations in C are much more efficient than those in the Scala runtime. The C code can directly `mmap` the input file and parse the csv, whereas the Scala code needs to use a slow Stream interface. For those reasons, we want to be able to generate C code, as well as benefit from OSR.

In the generative programming setting, the programmer can use high-level constructs for generating OSR regions, without having to handle tedious low-level technical details. We examine three different kinds of OSR regions that are available in our framework, and discuss the technical details for each: tiered execution loops, low-level speculation, and high-level speculation.

```
Print(
  Join(Seq("partkey"),
    Filter(Eq(Field("shipdate"),
       Const("1995-09-01")),
      Project(Seq("partkey",
         "shipdate", "quantity"),
        Scan(stream("linetitem")))),
    Project(Seq("type", "partkey"),
      Scan(stream("part"))))
)
```

(a)



(b)

```
// Pipeline 1
val lineitem =
    Stream[Record](...)
val hm = new
    MultiMap("partkey", ...)

while (lineitem.hasNext) {
  val rec = lineitem.next
  val prec = rec("partkey",
      "shipdate", "quantity")
  if (prec("shipdate") ==
      "1995-09-01")
     hm += (rec("partkey"),
        prec)
}

// Pipeline 2
val part = Stream[Record](...)

while (part.hashNext) {
  val rRec = part.next
  val prRec = rRec("type",
      "partkey")
  for (lRec <-
      hm(prRec("partkey")))
    (lRec ++ prRec).print
}
```

(c)

Figure 4.9.: Shape of the generated code for a Join operator. (b) Example of a query with a Join operator, (b) highlight the pipelines and loop structure of the code, (c) is the high level representation of the code generated. The `Stream` and `MultiMap` classes are abstractions that are specialized when the code is generated.

4.5 Tiered Compilation Experiments

In this section, we evaluate the performance of OSR patterns on tiered compilation. The workload targeted in our experiments is based on compiled query processing (Section 4.4). In such situations, OSR has the potential to provide dramatic speedups for fast startup and workloads that are comprised of complex but short-running queries. We use the TPC-H benchmark [155] benchmark that focuses on the performance of practical analytical queries.

Experimental Setup All experiments are conducted on a single NUMA machine with 4 sockets, 24 Intel(R) Xeon(R) Platinum 8168 cores per socket, and 750GB RAM per socket (3 TB total). The operating system is Ubuntu 16.04.4 LTS. We use Scala 2.11, GCC 5.4, Clang 6.0 and TCC 0.9.26.

Dataset We use the standard TPC-H benchmark with scale factor SF0.1, SF0.3, SF1 and SF10 (approximately 100MB, 300MB, 1GB and 10GB of csv files respectively).

Experimental Methodology For all our experiments, the timing is based on the `gettimeofday` function call. We report the median of 5 runs unless stated otherwise.

4.5.1 Tiered Compilation for SQL Queries

In this experiment, we evaluate tiered compilation in the context of compiled query evaluation. The LB2 query compiler [149] uses generative programming to compile SQL queries into optimized C. The back-end of a typical query compiler consists of a small number of structured operators that emit evaluation code on the form of tight, long-running loops that process data and perform various computations (e.g., computing an aggregate over grouped data). The execution path of a compiled SQL query consists of data structure initialization, data loading, and evaluation. In prac-

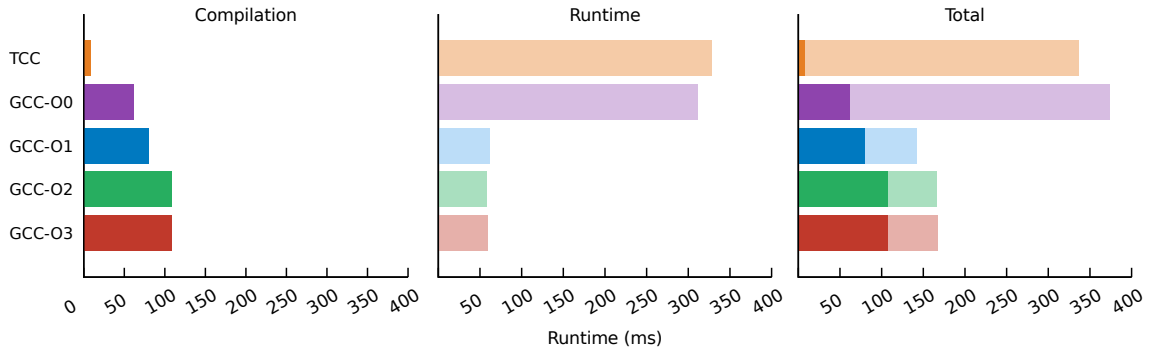
tice, SQL queries in TPC-H take 100-400ms to compile using GCC with the highest optimization flag (-O3). This time is acceptable when processing large datasets, but for small-size workloads, compilation time becomes a rather considerable overhead that may defeat the purpose of compiling SQL queries to low-level code [87]. How, then, to improve query compilation time?

```
select sum(l_extendedprice * l_discount) as revenue
from lineitem
where  l_shipdate >= date '1994-01-01'
 and l_shipdate < date '1995-01-01'
 and l_discount between 0.05 and 0.07
 and l_quantity < 24
```

(a)

```
// data structures initialization & data loading elided.
for (i = 0; i < size; i++) {
  double l_quantity = l_quantity_col[i];
  double l_extendedprice = l_extendedprice_col[i];
  double l_discount = l_discount_col[i];
  long l_shipdate = l_shipdate_col[i];
  if (l_shipdate >= 19940101L && l_shipdate < 19950101L
    && l_discount >= 0.05 && l_discount <= 0.07
    && l_quantity < 24.0) {
    revenue += l_extendedprice * l_discount;
  }
} ...
```

(b)



TCC	GCC-O0	GCC-O1	GCC-O2	GCC-O3
10.00	1.29	1.00	0.74	0.74

GCC-O2	GCC-O3	GCC-O1	GCC-O0	TCC
1.07	1.05	1.00	0.20	0.19

GCC-O1	GCC-O2	GCC-O3	TCC	GCC-O0
1.00	0.86	0.85	0.42	0.38

(c)

Figure 4.10.: TPC-H Q6 in (a) SQL and (b) handwritten C, (c) the compile, runtime and end-to-end execution of Q6. Speedups in the table are relative to GCC -O1.

A first idea is to tune the compilation optimization level without negatively impacting runtime. Thus, we first study the effect of varying the optimization flag levels on compilation time for GCC (we also evaluate Clang in Section 4.5.4). We pick the simplest query, TPC-H Q6, illustrated in Figure 4.10a, with scale factor SF0.1 (the lineitem table size is approximately 71MB). The hand-written Q6 shown in Figure 4.10b is essentially a loop with an if condition that iterates over the lineitem table, filters records, and computes a single aggregate operation (i.e., the sum of a simple computation on each data record). Figure 4.10c shows the time to compile, run, and the end-to-end execution time for Q6 using different optimization levels in GCC. At first glance, we observe that using lower optimization flags improves compilation time by approximately 20-40ms in GCC. Furthermore, the runtimes of -O3, -O2, and -O1 is nearly identical. Hence, for this basic query, GCC-O1 achieves the best compilation time and end-to-end execution time. However, using the lowest level -O0 significantly slows down runtime by 5×. How can the OSR pattern discussed earlier improve the performance of small-size queries?

While using a lower optimization flag *does* improve compilation time, 60-70ms is still perceived as a large compilation time for small-size queries, in our example, it is as much as the runtime itself. An alternative approach would be to use a less-optimized, faster compiler to implement the OSR pattern. The Tiny C Compiler (TCC) [116] is a fast, lightweight compiler that trades performance for speed. For instance, compiling Q6 in TCC takes approximately 7ms. The key idea is to compile and launch the query using TCC until the slow compiler finishes its work, after which OSR switches execution to the fast compiled code. We evaluate this in Section 4.5.3.

Finally, Consider the following breakdown of GCC -O3 and GCC -O0 compile times:

```
gcc -O3 -time tpch6.c gcc -O1 -time tpch6.c gcc -O0 -time tpch6.c
# cc1 0.06 0.01       # cc1 0.05 0.00       # cc1 0.02 0.01
# as 0.00 0.00        # as 0.00 0.00        # as 0.00 0.00
# collect2 0.01 0.00  # collect2 0.01 0.00  # collect2 0.01 0.00
```

We observe the higher optimization levels spend more time in the compilation phase. We also note that the linking time is high as well (the same amount of time TCC

spends in compilation). These observations encourage the use of the OSR pattern, it leaves the chance to the slow compiler to perform more optimizations without "wasting" time as code is already running.

4.5.2 Switching From Slow to Fast OSR Paths

Query evaluation is best described as performing a long-running loop where each iteration evaluates a number of data records. Integrating OSR in query compilers requires stopping a running loop in order to switch from slowly compiled code to one which has been compiled quickly. In this experiment, we evaluate the two switching mechanisms discussed in Section 4.2. Recall, the first approach uses a polling mechanism. We implemented the polling as follows: the compiler process creates a lock file as soon as the new target becomes available. Furthermore, a switching threshold X is configured to determine the frequency at which the code checks for the lock file, for each epoch the code performs X iterations then probes the existence of the lock file. The second approach uses a background thread that blocks until it is notified by the compilation process. Once notified, it dynamically loads the dynamic library. After that, the loading thread updates a volatile variable to signal readiness to the main processing thread. Checking a volatile variable has the advantage to be much less expensive than a polling operation.

Table 4.1 illustrates the impact of using a background thread and various switching thresholds (1, 100, 1000, 10000 and 100000 on OSR runtime in Q6 SF1). We observe that checking the availability of the fast code at each iteration incurs approximately 20ms overhead in runtime compared to the other thresholds. Indeed, performing a check at each iteration uses precious computation time, thus when the switch happens the amount of useful computation that has been performed is lower. In our experiment, the code executes only 67k iterations with a threshold of 1 versus 240k for the others. Similarly, picking a large threshold potentially wastes time depending on when the compiled target becomes ready. Indeed in the worse case, a

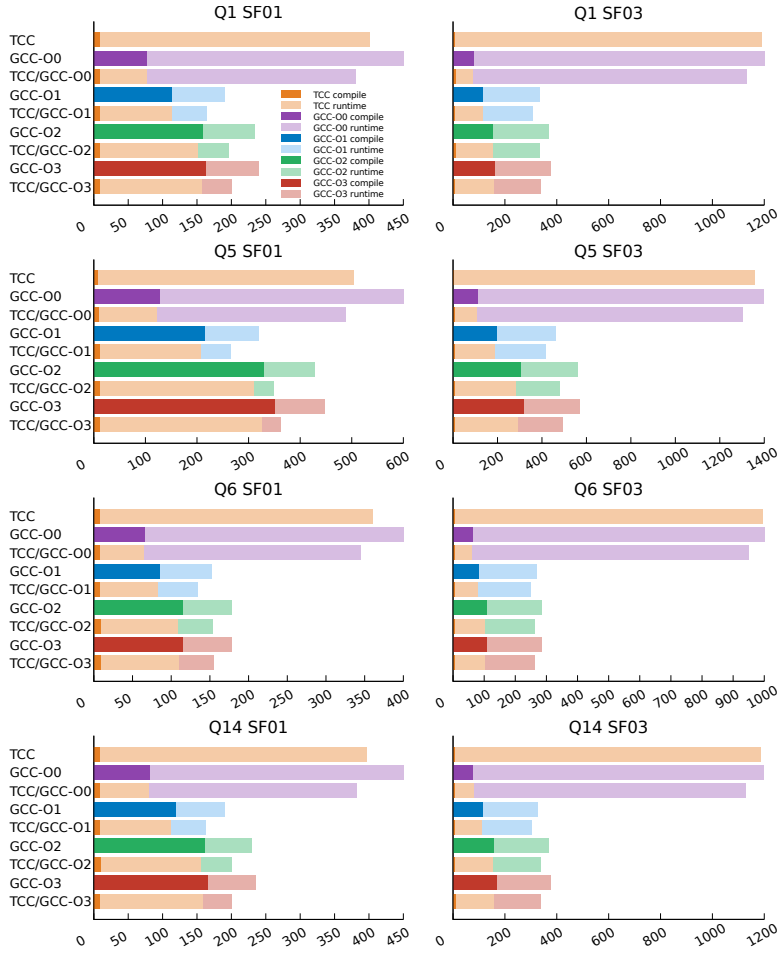
more optimized code could be available at the beginning of an epoch just after the check thus it could be available for a full epoch without being used. This situation is exhibited by our experiment: if the threshold is 100 iterations, the OSR swap arises at iteration 234k, thus for a threshold of 100k the swap occurs at 300k and therefore the code spends more time than necessary in the less optimized code when the threshold is too high (73ms vs 62ms). On the other hand, using a background thread is 25ms faster than the best threshold used in this experiment, making it the best solution if multi-threading is supported.

Table 4.1.: The impact of various switching thresholds on OSR-runtime using Q6 SF1 (see Figure 4.4).

threshold	thread	1	100	1000	10000	100000
runtime (ms)	672	721	698	697	700	705
switched at (ms)	59	62	60	58	61	73
switch iteration	224265	67856	234500	234000	250000	300000

4.5.3 Complex Code with Many OSR Regions

The OSR pattern is applicable on any long-running loop. Applying OSR on TPC-H Q6 is straightforward since it consists of a single loop or code region. For the case of complex programs, each loop is processed as an independent code region where the main program coordinates running code regions. Consider Figure 4.12 that shows TPC-H Q1 in SQL. At a high level, Q1 is an example of an aggregate operation that divides data into groups and computes the sum, average, etc., for each group. The execution breaks down into three distinct code regions as follows. The first region is a loop for inserting data into a hash table. The loop in the second region traverses the hashmap, obtains the computed aggregates and performs sorting. The last region iterates over the sorted buffer and prints results.



SF	TCC/GCC-O1	TCC/GCC-O2	TCC/GCC-O3	TCC/GCC-O0
0.1	1.16	0.97	0.96	0.50
0.3	1.08	0.99	0.99	0.29

Table A

SF	TCC/GCC-O1	TCC/GCC-O2	TCC/GCC-O3	TCC/GCC-O0
0.1	1.21	0.92	0.88	0.66
0.3	1.11	0.96	0.93	0.35

Table B

SF	TCC/GCC-O1	TCC/GCC-O2	TCC/GCC-O3	TCC/GCC-O0
0.1	1.14	0.99	0.99	0.44
0.3	1.07	1.02	1.02	0.28

Table C

SF	TCC/GCC-O1	TCC/GCC-O2	TCC/GCC-O3	TCC/GCC-O0
0.1	1.16	0.95	0.95	0.50
0.3	1.08	0.97	0.97	0.29

Table D

Figure 4.11.: Tiered execution comparison for Q1, Q5, Q6, and Q14 with different compiler configurations. XXX/YYY indicates the run was an OSR execution with first configuration XXX and second YYY. Tables A-D lists the relative speedup of OSR execution over the GCC -O1 configuration.

```
select l_returnflag, l_linestatus, sum(l_quantity),
    sum(l_extendedprice),
       sum(l_extendedprice * (1 - l_discount)),
       sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)),
       avg(l_quantity), avg(l_extendedprice), avg(l_discount),
           count(*)
from   lineitem
where l_shipdate <= date '1998-12-01' - interval '90' day
group by l_returnflag, l_linestatus
order by l_returnflag, l_linestatus
```

Figure 4.12.: TPC-H Q1.

Figure 4.11 shows the execution time of four different TPC-H queries using TCC as the baseline, GCC with various compilation flags, and OSR where TCC and GCC are the fast and slow compilers, respectively. The OSR query time consists of all TCC compilation time, a part of TCC runtime, and GCC runtime. We observe first, TCC compilation is very short (around 10ms), which allows starting execution early. Second, the OSR path is successful in reducing the end-to-end execution time by executing TCC at the beginning. Third, OSR preserves its expected behavior with increasing data size. However, increasing data size reduces the overall benefit of using OSR since the runtime dominates the end-to-end execution time. It may seem surprising that in the OSR context, the code switch appears to happen *before* the gcc compilation terminates. This is due to the fact, that in the OSR context, gcc has less code to compile than in the non OSR context and it does not have to create a full executable but only a shared library. Thus the compilation is slightly faster – around 6% for TPC-H Q1.

For the runs with a scale factor of 0.1 (100MB), the best OSR execution (TCC with GCC -O1) achieves between 14 and 21% speedup over the GCC -O1 configuration. For a larger scale factor of 0.3 (300MB), the speedup is between 7 and 11%. This confirms that OSR will be beneficial, as long as the compilation time is non-negligible compared to the running time. The OSR path in Q1 reduces end-to-end runtime by 20-30ms in comparison with GCC -O3, -O2 and -O1. TPC-H Q5 and Q14 are examples of join operations between five and two tables respectively. The pseudo-code in Figure 4.9 (d) gives a high-level implementation of a hash join operator between two tables. With this experiment, we see that even with a higher number of OSR region in the generated code, the technique improves the runtime.

4.5.4 Shape of Code

As discussed in Section 4.5.1, the highly-optimized compilers spend around two-thirds of compilation time in performing optimizations. Also, the linking time in

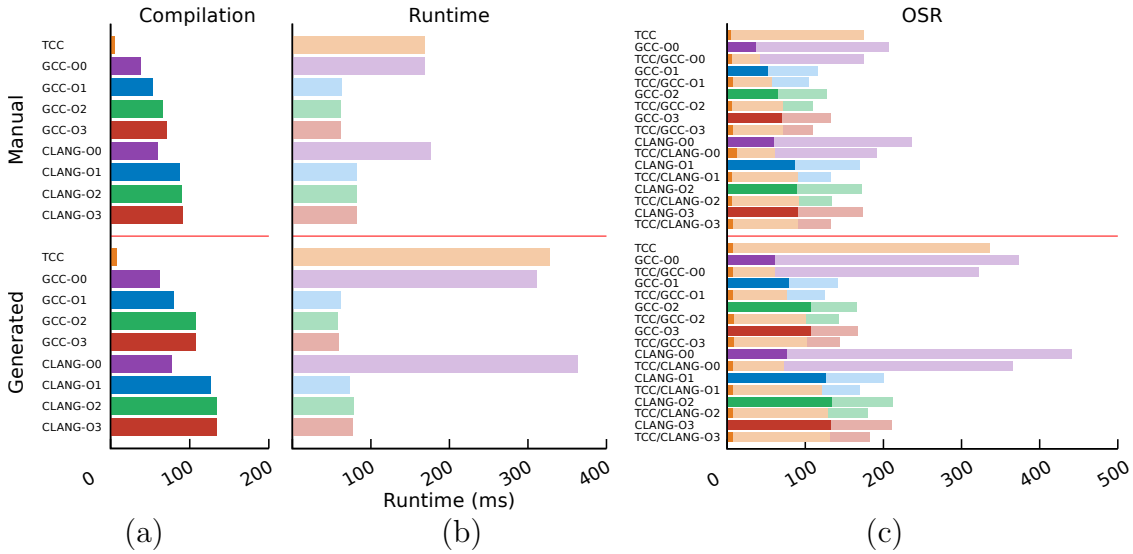


Table A: Speedup of manual over generated implementation of TPC-H Q6

TCC	CLANG-O0	GCC-O0	GCC-O2	GCC-O3	CLANG-O2	GCC-O1	CLANG-O3	CLANG-O1
1.93	1.87	1.80	1.30	1.26	1.23	1.22	1.22	1.18

Table B: Speedup of manual TPC-H Q6 over manual GCC-O1

TCC/GCC-O1	TCC/GCC-O2	TCC/GCC-O3	GCC-O1	GCC-O2	TCC/CLANG-O1	GCC-O3	TCC/CLANG-O2	TCC/CLANG-O3	CLANG-O1	CLANG-O2	CLANG-O3	TCC	TCC/GCC-O0	TCC/CLANG-O0	GCC-O0	CLANG-O0
1.12	1.06	1.05	1.00	0.91	0.88	0.87	0.87	0.87	0.69	0.67	0.67	0.67	0.66	0.61	0.56	0.49

Table C: Speedup of generated TPC-H Q6 over generated GCC-O1

TCC/GCC-O1	GCC-O1	TCC/GCC-O2	TCC/GCC-O3	GCC-O2	GCC-O3	TCC/CLANG-O1	TCC/CLANG-O2	TCC/CLANG-O3	CLANG-O1	CLANG-O3	CLANG-O2	TCC/GCC-O0	TCC	TCC/CLANG-O0	GCC-O0	CLANG-O0
1.14	1.00	0.99	0.99	0.86	0.85	0.84	0.79	0.78	0.71	0.67	0.67	0.44	0.42	0.39	0.38	0.32

Figure 4.13.: Compilation time and execution time of Q6 on SF0.1, for handwritten code and generated code on different compiler configurations. XXX/YYY indicates the run was an OSR execution with first configuration XXX and second YYY.

GCC alone is around the same as TCC's total compilation time. On the other hand, the class of fast compilers (e.g., TCC, GCC -O0 and Clang -O0) perform only a small set of optimizations to minimize compilation time at the expense of performance. For instance, less sophisticated compilers evaluate statements *individually*, whereas optimized compilers process multiple statements together. For example, TCC generates more efficient code for nested expressions than for a cascade of expression (such as ANF form).

```
//nested expression
x = y + z * u;
```

```
//ANF form
x1 = z * u;
x = y + x1;
```

In this experiment, we explore how the shape of code can help fast compilers to generate faster code. We manually implemented TPC-H Q6 using nested expressions and executed the query using TCC, GCC, and Clang. Figure 4.13a-b shows the compile- and runtime of Q6 using the handwritten code and the code generated by LB2. Table A summarizes the key outcome by listing the relative speedup of the manual code over the generated one. We observe that compilers with the slowest compiling times (TCC and Clang) benefited the most with 1.93×-1.87× speedup, respectively.

Figure 4.13c shows the OSR execution using TCC as the fast compiler and various GCC and Clang configurations as slow compilers. For speedup computations, we pick GCC -O1 as the best default (non-OSR) path to measure performance. Tables B and C summarize the speedup or slowdown of manual and generated OSR execution paths over GCC -O1. For the manual case, we see that the OSR configuration TCC/GCC -OX outperforms GCC-O1 by 12%, 6%, and 5% respectively. However, only TCC/GCC-O1 outperforms GCC-O1 (14%) in the generated setting as TCC is much less efficient in this situation. However, the OSR pattern is conserved. Indeed, the OSR configuration always outperforms its corresponding non-OSR configuration.

While the running times between generated and handwritten code are very close for GCC and Clang (with optimization), the compilation time actually changes a lot. This means that compilers manage to optimize the code and converge to the same version but need more time to do it. GCC takes between 50-60% and Clang around 46% more compilation time when the code is generated. For lower optimization level or TCC, the compilation time, as well as the runtime, is increased by almost a 2× factor.

The key insight is code generation frameworks like Lightweight Modular Staging will need to generate code that makes compilation faster, e.g. nested expressions, etc. if they want to benefit fully from OSR.

4.6 Speculative Optimization Experiments

In this section, we evaluate the performance of on-stack replacement patterns in various scenarios of speculative optimizations. We first look at high-level speculations where multiple code snippets are generated for the same task. We then look at low-level speculation where the same generated code is compiled using different optimization options. In both cases, there are multiple OSR regions generated and the program generator adds the logic to swap between them efficiently.

In generic code with many different paths, compilers may have difficulty optimizing each path correctly. Our hypothesis is that if we separate each path into its own compilation unit, the compiler will do a much better job for each path. For example, autovectorization may be ruled out because of complex control-flow, and singling out a single path may permit it. In addition, we assume that it is possible to combine the different paths back together and thus optimize the original program. In the following paragraph, we test this hypothesis on some microbenchmarks and evaluate the possible benefits.

4.6.1 Type Specialization

Dynamic language VMs are all about type specialization. A generic `+` operation could be used on integers, doubles, or even strings, depending on context. For program generators, this is not a typical use case. Since programmatic specialization is one of the prime applications of staging, one would typically try very hard to generate type-specialized code up front.

However, there are related applications that do occur in practice: for example, needing to support for variable-precisions within the same type (see below).

4.6.2 Variable-Size Data

An example of variable-size data is the `mpz_t` datastructure of the GMP library [57]. The space used by the data is runtime dependent, and the performance is linked to its size. A programmer may want to be able to handle all possible scenarios in their program, however, using `mpz_t` when all data could fit into an `int` or a `long` will lead to serious performance penalties.

In this experiment, we look at three different programs that compute the sum of integers of arbitrary size (See Figure 4.14). Program (1) is storing the integer value into `mpz_t`, the program (2) and (3) use a scheme where values between 0 and $2^{63}-1$ are stored as a `long` and other values as `mpz_t`.

Figure 4.15 reports the average running time in milliseconds of twenty runs of these three programs, all of which operate on arrays of 5 million integers. We ran the experiment with inputs having different densities of numbers larger than 2^{63}, 1 in 1 million, 10, 100, and 1000 in 1 million. The higher the density, the lower the index of the first large number will be, thus reducing the advantage of the speculation for programs (2) and (3). The experiments show that in this situation, our assumption was correct. The program with the OSR region performed better than the single loop program. Using the flag that reports successful vectorization, we can confirm that in program (3) the loop is vectorized by GCC, but in program (2) it is not. In order to make the vectorization possible, program (3) needs to be written in a particular manner. Instead of exiting as soon as the assumption is violated, the program computes the aggregate on a fixed window and sets a flag that the assumption is violated. After finishing the window, the program checks the flag and swaps the OSR region upon assumption violation. The following region has to rerun the last window. This is necessary because a loop with multiple exits cannot yet be vectorized by compilers (ICC, GCC, or Clang), but it could potentially in the future and therefore improve this situation even further.

```
mpz_t agg;
mpz_init_set_ui(agg, 0);
for (it = 0; it < length ; ++it) {
  mpz_add(agg, agg, arr[it]);
}
```

(1)

The naive version using `mpz_t` for all number and for the computation.

```
long agg = 0L;
mpz_t bagg;
int changed = 0;
for (it = 0; it < length; it++) {
  int val = arr[it];
  if (val & TAG) {
    if (changed) {
      mpz_add(bagg, bagg,
          storage[val ^ TAG]);
    } else {
      mpz_init_set_ui(bagg, agg);
      mpz_add(bagg, bagg,
          storage[val ^ TAG]);
      changed = 1;
    }
  } else {
    if (changed) mpz_add_ui(bagg,
        bagg, val);
    else agg += val;
  }
}
```

(2)

Stores the value between 0 and $2^{63} - 1$ in a `int` and the other in `mpz_t`. It is a simple loop program that starts to assume that all values are stored as `long` and accumulate into a `long`, if the assumption is violated it continues by accumulating in a `mpz_t`.

```
// OSR region 1
int add_spec(int* arr, int* it_p, long*
    agg_p, int interval) {
  long agg = *agg_p;
  int it = *it_p;
  int fail = 0;
  for (; it < interval; it++) {
    int val = arr[it];
    fail |= val & TAG;
    agg += val;
  }
  if (fail) return 1;
  *agg_p = agg;
  *it_p = it;
  return 0;
}

// OSR region 2
int add(int* arr, mpz_t* storage,
        int* it_p, mpz_t agg, int length)
            {
  int it = *it_p;
  for (; it < length; i++) {
    int val = arr[it];
    if (val & TAG)  mpz_add_ui(agg, agg,
        val);
    else mpz_add(agg, agg, storage[val ^
        TAG]);
  }
}

// Main
int limit = 1000; long agg = 0L;
mpz_t bagg;
while (it < length && !add_spec(arr, &it,
    &agg, limit))
  limit += 1000;
mpz_init_set_ui(bagg, agg);
if (it < length)  add(arr, storage, &it,
    bagg, length);
```

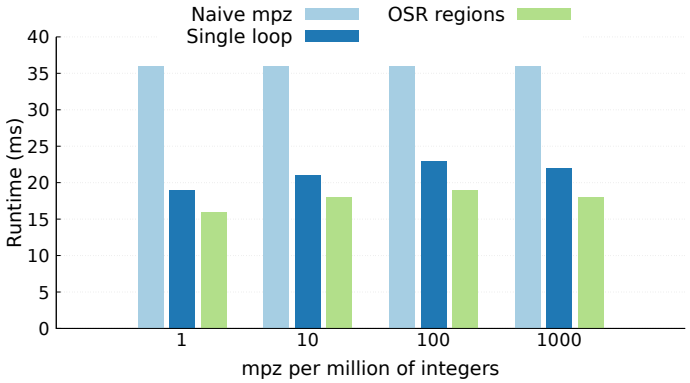
(3)

The code is similar to (2), but instead of being a single loop it has two OSR regions.

Figure 4.14.: Different programs used for experiment on Variable-Size Data.

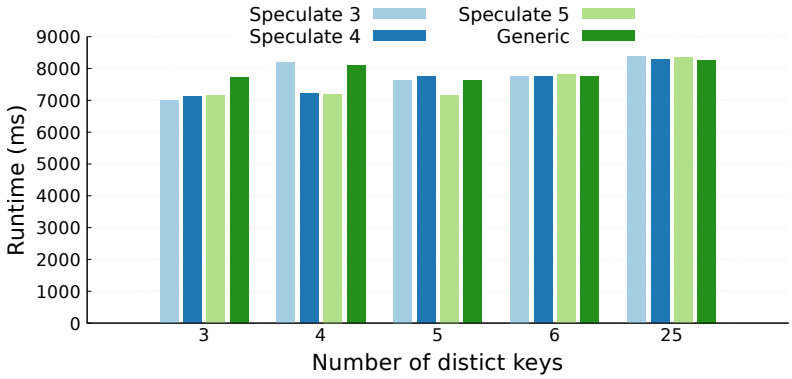
4.6.3 Inline Data Structures

Collections such as hashmaps are used to implement complex algorithms efficiently. They usually have a very good theoretical asymptotic performance; however, there are some specific cases where they are not optimal. For example, Q1 of the TPC-H benchmark has only four different keys for the `group by` operation using the standard TPC-H data. For generality, it is implemented using a hashmap. But in



`mpz_t` per 1M integers	1	10	100	1000
Single loop	1.89	1.71	1.56	1.64
OSR regions	2.25	2	1.89	2

Figure 4.15.: Comparison of the sum of an array of 5 millions potentially large positive integers. Average runtime of 20 runs in microseconds (each run has a different input array randomly generated). The table represent the speedup relative to the Naive `mpz_t` version.



	3	4	5	6	25
speculate 3	1.10	0.99	1.00	1.00	0.99
speculate 4	1.08	1.12	0.98	1.00	1.00
speculate 5	1.08	1.13	1.06	0.99	0.99

Figure 4.16.: Runtime of generated with speculation on the number of distinct keys and a generic hashmap implementation. The x axis represents the actual number of distinct keys. The table display the speed up of each configuration relatively to the generic hashmap implementation.

that context, hashmaps add more overhead than simply using four variable to store the different values. Based on that observation, we test some speculative high-level optimizations. We generated different code for the query: one that assumes there is going to be only 3 distinct keys, another 4, and another 5. For comparison, we also generated a program that is using the `GHashMap` from the GLib library. The code specialized for a given number of keys stores them in local variables instead of a more complex data-structure. Given that the number of keys is small, only a small number of comparisons is needed to find the correct variable to store the data. If the number of keys exceeds the speculated number, the program falls back to the generic implementation with the `GHashMap`. In Figure 4.16, we report the result of our experiment. We ran this program on a table that actually has 3, 4, 5, 6, or 25 distinct keys. We can see that when the assumption was correct (number of key speculated higher than the actual number of keys), the specialized code performs much better than the generic hashmap. But even more importantly, when the speculation is not valid, the code *does not perform worse* than the generic hashmap.

4.6.4 Loop Tiling

A famous optimization pattern in linear algebra algorithms is loop tiling. It is used to improve locality and cache reuse. One could think that OSR could be used to speculate different tiling strategies and allow the program to change on the fly in order to find the optimal tiling setting. However, it is actually difficult to create a safe point (see also [24]). In addition, knowing if or when it is beneficial to switch to a new configuration depends on timing and cache related statistics. This information is not easily accessible during compilation, and thus introduces further overhead in practice.

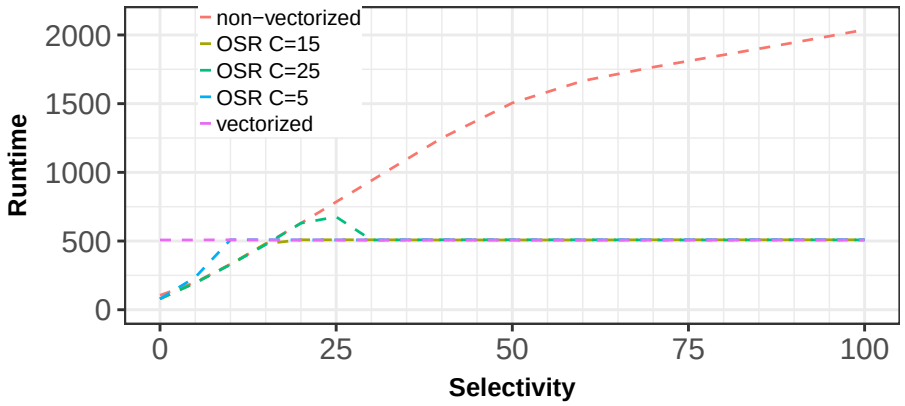


Figure 4.17.: Comparison of vectorized, non-vectorized and OSR code. The OSR code exhibits the best behavior for any selectivity of the data.

4.6.5 Predication vs Branches

Modern hardware supports SIMD, and compilers need to evaluate the benefit of vectorization with some heuristic. However, the benefit can actually be data-dependent. For example, a for-loop computing an aggregate on an array based on some condition (e.g., a basic map-filter-reduce operation) can be vectorized (see Figure 4.1). The instructions emitted use wide registers to compute the result at the same time. In order to handle the conditions, the processor creates a mask that *voids* the results computed that are not needed. While in most cases it results in a sped-up computation, there are some limits. Consider the situation where a SIMD instruction computes two values at the same time. If the computation of these values is significant and the majority of the time the values are discarded because the condition is false, the computation is an overhead and the non-vectorized version may be better. In this experiment, we propose to transform the loop into an OSR region and compile it once with the `-ftree-vectorized` flag and once without it. The loop also includes a counter that keeps track of the selectivity of the data: upon a given cutoff, the code is going to switch between the two different assemblies coded. In Figure 4.17, we present the result of the experiments. Empirically, we can see that the vectorized code and the non-vectorized code intersect when the selectivity is around 15%: using

this for the cutoff value yields the best performance (the OSR cutoff 15 line cannot be seen as it is under the non-vectorized line from 0 to 15 and under the vectorized line from 15 to 100). We can also see that different cutoffs switch too early or too late.

Similarly to the variable-size data experiment, the OSR region is running for a fixed window and checks the swapping condition once it is done; otherwise, it would prevent the vectorization of the loop. There are different ways of computing the swapping condition: the decision can be made based on the last window, or on all the data that has been processed so far. Each of those strategies would have a worst-case scenario that would work for a swap at each window, while not being the optimal choice for the following one.

4.7 Related Work

OSR On-stack-replacement was first prototyped in SELF [75]. The SELF-93 VM was designed to combine interactivity and performance. The SELF code compiles only when needed, instead of performing a lengthy global compilation pass. The technique unwinds the stack and finds the best function to compile, then replaces all the lower stack parts with the stack of the optimized function. The SmallTalk 80 [45] system was implemented using many sophisticated techniques including polymorphic inline caching (PIC) and JIT compilation. In the case of the JIT compilation, the procedures' activation records had different representations for the interpreted code and for the native code: swapping between these representations is the same that swapping between two OSR regions in the speculative setting. Strongtalk [29] provides a type system for the untyped Smalltalk language. An OSR LLVM API is given in [43, 92], similar to our work but focused on a low-level approach within the LLVM IR more targeted toward VM implementation. OSR is also implemented in Hotspot [113] and V8 [65]. The work in [59] uses OSR to switch between garbage collection systems. The work in Skip & Jump [160] presents an OSR API for a low-level virtual machine

based on Swapstack. Our work is different from those as it make available OSR to the programmer explicitly, rather than within a runtime environment as an optimization of the language runtime.

JIT Compilers Examples of modern JIT compilers include the Jalapeo VM [13] that targets server applications; the Oracle HotSpot [113] VM which improves performance through optimization of frequently executed application code; Jikes RVM [7]; the metacircular research VM, Maxine [162]; SPUR [18], a tracing JIT for C#; Google's V8 [65]; Crankshaft [66]; and TurboFan [2]. Truffle [163] is built on top of Graal [109], and optimizes AST interpreters. The PyPy [26,127] framework is written in Python and works on program traces instead of whole methods. The Mu micro VM [160] focuses on JIT compilation, concurrency, and garbage collection.

Optimization, Deoptimization, and Performance Dynamic deoptimization was pioneered in the SELF VM to provide expected behavior with globally-optimized code. The compiler inserts debugging information at interrupt points, while fully optimizing in between [74]. In essence, our OSR regions implemented in this Chapter are similar to [74]. Debugging deoptimization in [73] deoptimizes dependent methods whenever a class loading invalidates inlining or other optimizations. PIC [72] extends the inline caching technique to process polymorphic call sites. The work in [60] deoptimizes code compiled under assumptions that are no longer valid. Bhandari and Nandivada [24] present a generalized scheme to do exception-safe loop optimizations and exception-safe loop tiling.

Staging and Program Generation Delite [31, 144] is a general purpose compiler framework, implements high-performance DSLs, provides parallel patterns, and generates code for heterogeneous targets. LMS [130] is library-based generative programming and compiler framework. LMS uses types instead of syntax to identify binding times, and generates an intermediate representation instead of target code. Code generation examples from relevant domains include Spiral [118] for digital signal

processing (DSP) and Halide [120] for image processing. Haskell is another popular host for embedded DSLs [145]. The work in [89] embeds a DSL that mimics JavaScript in Scala using LMS.

4.8 Conclusion

In this Chapter, we have presented a surprisingly simple pattern for implementing OSR in source-to-source compilers or explicit program generators that target languages with structured control flow (loops and conditionals). We have shown how on-stack-replacement provides the ability to replace currently executing code with a different version, either a more optimized one or a more general one, within a high level program. OSR has been a key component in all modern VMs for languages like Java or JavaScript for a long time, however it has only recently been studied as a more abstract program transformation, independent of language VMs. Our work extends the scope of OSR beyond the context of low-level execution models based on stack frames, labels, and jumps and makes it more broadly applicable.

We have evaluated key use cases and demonstrated attractive speedups for tiered compilation with Flare. We have further shown that casting OSR as a metaprogramming technique enables new speculative optimization patterns beyond what is commonly implemented in language VMs.

5 PRECISE REASONING WITH STRUCTURED TIME, STRUCTURED HEAPS, AND COLLECTIVE OPERATIONS

This Chapter is based on [56].

In Chapter 2, we implement data-centric systems using a defered API: first the system builds a graph representing the computation, then a compiler phase generates efficient code. However, a lot of legacy code is not implemented in this fashion, instead implemented as low-level, imperative code. How can we leverage these pieces of code already implemented and apply Flare optimizations to make them parallel or even distributed? Similarly, Flare is limited to UDFs written with the LMS framework, how can we add support for generic UDFs?

In this chapter, we propose a novel vision to static analysis that allows us to extract high level constructs from a program. The results given by this new analysis allow us to accelerate legacy code with Flare, but, in addition, it also provides an analysis with additional applications such as verification.

5.1 Introduction

Programs like the one in Figure 5.1 are a real challenge for analysis and verification tools. The language features used include arithmetic, dynamic memory allocations, linked heap structures, and loops. Analysis tools need to reason about all these features with high precision. If precision is lost at any point, it may be impossible to obtain any useful result. In practice, many state-of-the-art tools are unable to verify this program, including tools that score highly on software verification competitions such as CPAchecker [23] and SeaHorn [69], as well as Facebook's Infer tool [34].

While it is easy to come up with a long list of individual reasons that make this kind of analysis hard, we make two overarching observations:

```
// build list of numbers < n
x := null; l := 0
while l < n do {
  y := new;
  y.head := l;
  y.tail := x;
  x := y;
  l := l + 1
}
```

```
// traverse list, compute sum
z: = x; s: = 0
while z != null do {
  s := s + z.head;
  z := z.tail
}

// check result (closed form)
assert(s == n*(n-1)/2)
```

Figure 5.1.: Challenge program: in constrast to state-of-the-art tools like CPAchecker, SeaHorn, or Infer, our approach is able to verify the final assertion, as well as the absence of memory errors such as dereferences of `null` or missing fields. Dynamic allocations and linked data structures pose particular difficulties w.r.t. disambiguation of memory references, manifest in potential aliasing and the problem of "strong" updates. Our structured heap model represents allocations inside a loop using a collective form for sequence construction $y = \langle . \rangle (i < n).\ [\texttt{head} \mapsto i, \texttt{tail} \mapsto ...]$, based on which the second loop maps to a collective sum $s = \Sigma(i < n).\ y[i].\texttt{head} = \Sigma(i < n).\ i$ over this sequence. Knowledge about closed forms for certain sums validates the final assert (details, including the definition of $\texttt{tail} \mapsto ...$, are shown in Figure 5.2 and Section 5.2.2).

1. Program abstractions used in common analysis methods are typically *scalar*, i.e., they represent individual program variables, relations between individual variables as in the case of relational abstract domains, array updates at individual positions, and so on. But program abstractions do not typically represent *collective* entities such as "an array that contains the natural numbers from 1 to n" or "the sum of all elements in an array." Instead, such information must be encoded extensionally using quantified and often recursive formulae.

2. Program abstractions typically project away the dimension of *time*. Most analyses gather and collapse information into a single abstract value per *program point* (possibly with some context-sensitivity). This means that in the presence of loops, values computed in different loop iterations are not distinguished. Hence, program abstractions do not typically represent *space-time* information such as "field `tail` of the object allocated here in a given loop iteration points to the object allocated at the same position in the preceding loop iteration."

In this Chapter, we address both points through first-class collective operations and a structured heap representation, coupled with a structured notion of time.

First-Class Collective Operations For the first point, we propose to model *collective operations* such as sums or array formation as first-class entities, without quantifiers or recursive definitions. This idea is inspired by Σ-notation in mathematics, and by recent advances in highly optimizing compilers where high-level IRs based on `map`, `reduce`, and similar collective abstractions have had significant success.

In mathematics, collective forms such as $\langle a_i \rangle_i$ for sequences and $\Sigma_i a_i$ for series are not just compact syntactic sugar, but they give rise to intuitive algebraic laws. Thus, collective forms enable reasoning about sequences and series on a higher level than directly about the underlying recurrences. Introduced by Fourier [62], big-Σ and related operators have rapidly become an integral part of modern mathematics, and they have found their way into programming languages in the form of comprehensions via SETL [136], via Dijkstra's Eindhoven Quantifier Notation [47], and of course as functional operators via APL [77].

If these abstractions help manual reasoning, then it seems only logical that they should also help automated reasoning—so it is surprising that program analysis tools do not in general afford collective forms first-class status and do not try to reverse-engineer low-level code into such higher-level representations. Instead, automated tools usually reason at the level of scalar recurrences, which poses all kinds of challenges.

Compilers: From Optimizing for Performance to Simplifying for Clarity In this aspect, the field of optimizing compilers is ahead of general program analysis. Compiler writers have long recognized that aggressive transformations such as automatic parallelization are very hard to perform on low-level, imperative program representations. Hence, there has been ample work on trying to extract structure from low-level code. For example, the Chains of Recurrences (CoR) model [16, 52] is a collective and closed-form representation for classes of functions including affine,

multivariate polynomial, and geometric functions, which is widely used in optimizing compilers for generating efficient code to compute a given function on an interval of indexes, e.g., in a loop.

Over the last decade, a thriving line of research has demonstrated that even more aggressive transformations such as automatic parallelization are imminently practical for domain-specific languages (DSLs) that restrict mutability and make collective operations such as `map`, `reduce`, `filter`, `groupBy`, etc., first-class, so that the DSL compiler can reason about them algebraically when making optimization decisions, potentially coupled with auto-tuning and/or search for the best implementation based on cost models and dynamic programming. These systems outperform comparable code written in general-purpose languages by orders of magnitude and achieve asymptotically better parallel scaling [31, 32, 120, 131, 133, 141–144]. And while the orignal goal was for programmers to write DSL code directly, recent research has also shown that it is often practical to "decompile" low-level legacy code into high-level DSLs, whose role shifts to that of an intermediate representation [5,83,103,119,124,129,132].

The key thrust of this Chapter is to take this approach further and apply it to more general program analysis settings, including for the purpose of automatic verification. Thus we shift the goal from optimizing programs for performance to simplifying programs for clarity, by extracting high-level collective operators from low-level code. It is not intuitively clear that this reverse-engineering task is easier than the desired analysis itself, but we will show how several techniques come together to make this approach practical, in particular by fusing several simplification and analysis tasks into a single iterative fixed-point computation (Section 5.4).

Structured Time and Structured Heaps A major challenge remains: dynamic memory allocation, coupled with unbounded iteration constructs, may lead to an unbounded number of runtime objects, which need to be mapped to a finite static representation. The crux of this challenge is to find a static representation that still enables effective disambiguation of memory references in order to minimize potential

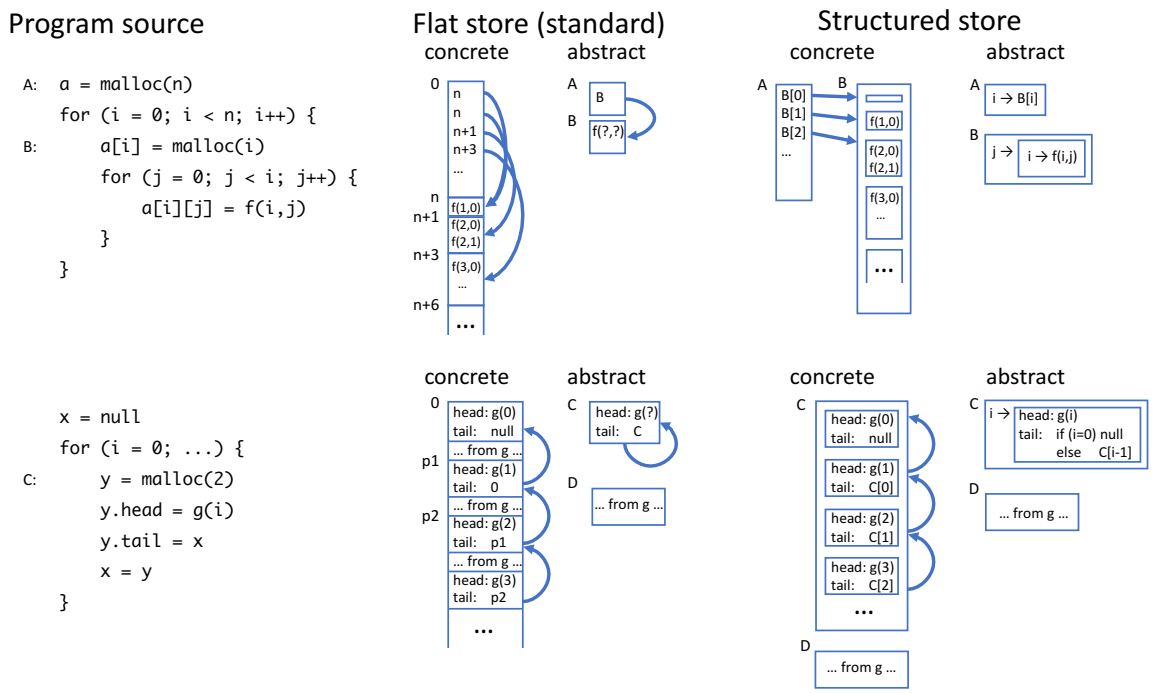


Figure 5.2.: Flat vs. structured stores for two example programs. The standard flat store model assigns consecutive numeric addresses for each allocation. By contrast, addresses in our structured store model consist of a program point and the surrounding loop variables at the time of allocation. `A, B[1], C[42]` are all valid concrete addresses. For program points inside loops (B and C), all objects allocated there are are represented as sequences ($B = \langle \ldots \rangle_i$ and $C = \langle \ldots \rangle_i$) in the store. Going from concrete to abstract stores, these sequences can be abstracted as collective forms, with greatly improved precision over an abstract store that only distinguishes allocations by program point. In particular, it is straightforward to capture the property that elements in a linked list point to the element allocated in the previous loop iteration and the last element points to null.

aliasing of pointers, which, among other detrimental effects, stands in the way of *strong updates*: recognizing when the previous value of a variable or memory location is definitely overwritten. More generally, strong updates are one example of a situation where an analysis needs to reason about the dimension of *time*, i.e., relating values at different points during the dynamic execution of the program. To address this challenge, a key ingredient of our approach is to identify useful collective-form representations for arbitrary-size dynamic heap structures such as arrays and linked lists.

We propose a novel structured heap model that incorporates the dimension of time in the structure of addresses and the allocation policy (see Figure 5.2). This concrete heap model leads directly to an abstract heap model that permits concise and expressive symbolic representation. Instead of abstracting dynamic allocations uniformly per program point, and having the abstract heap map abstract locations to sets of abstract values, we represent objects allocated at a program point inside a loop as a potentially unbounded sequence, indexed by the loop variable. This enables us to reason about all objects allocated at a given program point in a collective way, and assign a concise abstract summary based on a *symbolic* loop index. This in turn gives us fine-grained abilities to reason about objects allocated in *different* loop iterations and about their interactions, including strong updates deep within data structures (see Figure 5.2).

Framework Instantiation We instantiate our framework in a way that is similar to deductive verification with forward reasoning, i.e., a strongest postcondition model. We translate imperative source programs into a functional representation and simplify, preserving correctness and error behavior modulo termination. The translation makes all error conditions explicit, so verification amounts to checking that the `valid` tag that signals the occurrence of errors in the final program state has been simplified to the constant `true`.

In the simplest case, simplification can happen entirely after translation, based on various rewriting strategies that apply simplification rules one by one. The strategies can be deterministic, e.g., apply simplification rules bottom-up, or non-deterministically search for the most profitable simplification based on various heuristics (Section 5.3.3).

From Pessimistic to Optimistic However, even search-based post-hoc simplification strategies are fundamentally limited in that each individual rewrite has to be equality-preserving. The result is essentially a phase ordering problem (typical in compilers). The solution is to *interleave* translation and simplification [95], which en-

ables a form of *speculative* rewriting where chains of rewrites can be tried and either committed, if they are found to preserve equality, or rolled back, if not (Section 5.4). The post-hoc approach is also characterized as *pessimistic*, and the interleaved approach as *optimistic*. To illustrate the difference, the optimistic approach can simplify a loop based on the assumption that a variable remains constant throughout the loop, if it verifies the assumption afterwards. By contrast, the pessimistic approach would need prove that the variable is loop-invariant first, which may not be possible without the simplifications currently on hold.

In both pessimistic and optimistic approaches, the end result is a precise symbolic representation of the program state post execution. Although inefficient, this representation could be used to compute the concrete program output for any given input. This means that our approach leads to more precise information than strictly necessary for verification, and essentially solves a harder problem than demand-driven verification approaches. The fact that we are able to gain this much precision in a strongest-postcondition setting that is typically considered an unworkable verification approach makes us confident that the core of our method will apply just as well in weakest precondition scenarios or analyses based on abstract interpretation that approximate deliberately, and lead to increased precision there as well.

We implement our approach in a prototype system called `SIGMA`, which analyzes C code. Since our approach produces precise symbolic program representations, we can use `SIGMA` not only for verification, but also for checking program equivalence, and for translating legacy code to high-performance DSLs, as we demonstrate in our evaluation.

Contributions To the best of our knowledge, no previous work models first-class collective operations in a general-purpose program analysis setting. There are specialized uses in aggressive optimizing compilers that aim to retarget legacy code to high-performance DSLs [5,83,103,119,124,129,132], but these systems (a) are specific to a given target DSL, and (b) only deal with flat arrays, not linked lists or dynamic

memory allocation. Likewise, simple classes of structured heap models have been used in previous work [150], but these are restricted to separating container instances. The idea of indexing program values by loop iterations has also been proposed in the context of dynamic program analysis [164] and in polyhedral compilation [20]. Our key novel insight, not found in any previous work, is to push execution indexing all the way into the heap and allocation model. We give the heap a structure that uniformly reflects the program's loop structure, with objects allocated at a program point inside a loop represented as a sequence indexed by the loop variable, and use this concrete heap model as a basis for symbolic analysis.

5.2 Collective & Closed Forms, Step-by-Step

We take a program in the imperative source language `IMP` as input (Figure 5.3), and translate it into an equivalent functional program in our intermediate analysis language `FUN` (Figure 5.6), on which we perform symbolic simplification to expose the properties of interest (either after the translation or interleaved with it). For soundness, we require simplification to preserve all potential error conditions, but we do not, for example, need to preserve cases of divergence. `FUN` is a good intermediate representation for several reasons. First, it eschews side effects and makes all data dependencies explicit, which enables simplification through structural rewriting and enables us to represent `IMP`-level error conditions explicitly in the language. Second, it provides both recursive functions and collective forms, which enables us to gradually move from one to the other within the same language. While it can be helpful to think of the translation to `FUN` as a form of abstract interpretation of `IMP` programs, especially w.r.t. the Kleene iteration in Section 5.4, it is important to stress that the basic translation is exact (i.e., fully semantics preserving), without any approximation (details in Section 5.3).

As a running example, let us consider a simple `while` loop, which sums the integers from 0 to `k`-1 in variable `s`:

```
j := 0;
s := 0;
while j < k do {
  s := s + j;
  j := j + 1
}
```

Our goal is to characterize the program state after the loop, i.e., to obtain a mapping of the form `[ j -> ?, s -> ? ]`, from program variables to abstract values (in our case, symbolic expressions).

The first step is to transform this `IMP` program into an equivalent `FUN` program. We make loop indices explicit and represent the values of `j` and `s` *after* a certain loop iteration i by a set of recursive functions, derived from the program text. For example, after the first iteration ($i = 0$), `j` and `s` are equal to 1 and 0 respectively, and are increased by 1 and $j(i-1)$ respectively for each iteration:

$$\texttt{let } j = \lambda(i).\ \texttt{if } i \geq 0 \texttt{ then } j(i-1)+1 \texttt{ else } 0$$
$$\texttt{let } s = \lambda(i).\ \texttt{if } i \geq 0 \texttt{ then } s(i-1)+j(i-1) \texttt{ else } 0$$

Now we can meaningfully talk about values at iterations i and $i-1$, and about their relationship: we reason in both space and time. We describe the trip count of the loop declaratively, as the first n for which the condition is `false`, using the built-in # functional—an example of a collective form:

$$\texttt{let } n = \#(i).\ \neg(j(i) < k)$$

The operational interpretation of $\#(i).\ f(i)$ is to find the smallest $i \geq 0$ for which $f(i)$ evaluates to true or to diverge if no such i exists. Variable i is bound within the term following the dot, i.e, $f(i)$. We are now ready to describe the program state after the loop as a `FUN` term by mapping each variable to a precise symbolic description of how it is computed using the previous definitions: `[ j ->` $j(n-1)$ `, s ->` $s(n-1)$ `]`

We go on by identifying patterns in the recursive definitions. The following chain of rewrites transforms j and s to collective forms: an explicit sum construct, comparable to the mathematical Σ notation. For uniformity with other collective forms, we use the syntax $\Sigma(i < n).\ f(i)$ to denote the sum of all $f(i)$ for all $0 \leq i < n$. Again, variable i is bound in the body of the term. As part of the simplification, β-reduction is performed for non-recursive functions:

```
let j = λ(i). if i ≥ 0 then j(i − 1) + 1 else 0
      = λ(i). Σ(i2 < i + 1). 1
let s = λ(i). if i ≥ 0 then s(i − 1) + j(i − 1) else 0
      = λ(i). Σ(i3 < i + 1). j(i3 − 1)
```

The collective sums for j, s are readily transformed to closed forms, which also provides a closed form loop count n:

```
let j = λ(i). Σ(i2 < i + 1). 1
      = λ(i). if i ≥ 0 then i + 1 else 0
let s = λ(i). Σ(i3 < i + 1). j(i3 − 1)
      = λ(i). Σ(i3 < i + 1). i3
      = λ(i). if i ≥ 0 then (i + 1) ∗ i/2 else 0
let n = #(i). ¬(j(i) < k)
      = #(i). ¬(i + 1 < k)
      = if k ≥ 0 then k else 0
```

With that, we obtain the desired closed form representation for the final program state based on $j(n-1)$ and $s(n-1)$:

```
[ j -> if k ≥ 0 then k else 0, s -> if k ≥ 0 then k∗(k−1)/2 else 0 ]
```

This symbolic representation can be used for multiple purposes at this point, either to verify programmer-specified assertions, as in Figure 5.1, to test equivalence of the source program with another one, or to generate optimized code (Section 5.6).

To keep the presentation high-level, we have deliberately omitted some details above, including exactly *how* recursive relations are converted into collective and closed forms. A simple approach can be realized based on pattern-based rewriting (Section 5.3.3), while the more sophisticated iterative approach based on speculative rewriting is discussed in Section 5.4, using the same running example.

5.2.1 Collective Forms for Arrays

We now turn our attention to dynamic memory operations. The simplest case is arrays. We modify our example program to first store the numbers in an array `a`, and then compute the sum by traversing the array `a`:

```
// build array of numbers < n
a := new;
l := 0;
while l < n do {
  a[l] := l;
  l := l + 1
}
```

```
// traverse array, compute sum
j := 0;
s := 0;
while j < l do {
  s := s + a[j];
  j := j + 1
}
```

The additional challenge now is that our analysis needs to reason about array construction, as well as array traversal. In particular, we need to ensure that all array accesses are safe, and in addition, we need to precisely identify the values each array slot contains after the first loop, without any ambiguity. We solve this challenge by representing the array `a` using a closed form for sequence construction after the first loop, and recognizing that the second loop sums the elements of that sequence in `s`, which enables us again to use a collective sum expression.

The `FUN` representation is as follows. For simplicity, we show `l` and `j` already rewritten to closed forms, and the number of iterations already resolved to $n \geq 0$. The syntax `seq[i -> x]` denotes a copy of sequence `seq`, with the element at position `i` updated to `x`. We extract the recursive dependencies for the first loop:

```
let l = λ(i). if i ≥ 0 then i+1 else 0
let a = λ(i). if i ≥ 0 then a(i−1)[i -> i] else []
```

We can now describe the state after the first loop (a(n−1) refers to the definition above):

```
[ l -> n, a -> a(n−1) ]
```

Alas, this will not allow us to relate the array accesses of both loops. Can we do better? In addition to sums, products, and boolean connectives, our language `FUN` also contains collective form constructors for sequences. The notation `array(`$i < n$`).` $f(i)$ initializes a sequence or array with index range $i = 0, \ldots, n-1$, where each i is mapped to $f(i)$. Simplification proceeds as follows:

```
let a = λ(i). if i ≥ 0 then a(i−1)[i -> i] else []
      = λ(i). array(i₂ < i+1). i₂
```

And we obtain a much more useful description of the state after the first loop:

```
[ l -> n, a -> array(i < n). i ]
```

We can then proceed for the second loop:

```
let j = λ(i). if i ≥ 0 then i+1 else 0
```

```
let s = λ(i). if i ≥ 0 then s(i − 1) + a[j(i − 1)] else 0
      = λ(i). if i ≥ 0 then s(i − 1) + (array(i2 < n+1). i2)[i] else 0
```

Simplifying the array access `(array(`$i_2 < n+1$`). i2)[i]` to i depends on the presence of the enclosing loop precondition $i < n$, i.e., rewriting may be context- and flow-sensitive. Afterwards, simplification of `s` proceeds as before.

```
let s = λ(i). if i ≥ 0 then s(i − 1) + i else 0
      = λ(i). if i ≥ 0 then (i + 1) ∗ i/2 else 0
```

Finally the state at the end of the program is:

```
[ l -> n, j -> n, a -> array(i < n). i, s -> n*(n-1)/2 ]
```

5.2.2 Collective Forms for Linked Structures

To complicate matters further, we might store the numbers in a linked list instead of an array, and build the sum by traversing the list, leading to the code from Figure 5.1:

```
// build list of numbers < n
x := null;
l := 0;
while l < n do {
  y := new;
  y.head := l; y.tail := x;
  x := y;
  l := l + 1
}
```

```
// traverse list, compute sum
z: = x;
s: = 0;
while z != null do {
  s := s + z.head;
  z := z.tail
}
```

Now we need to reason about individual heap cells, allocated in different iterations of the first loop, as well as strong updates to the `head` and `tail` fields in these dynamically allocated objects. At this point, our structured heap model introduced in Figure 5.2 plays a crucial role.

At runtime, there will be one object created for `y` per loop iteration `i`. While variable `y` holds the *address* of an object, we identify the actual object by its location `p` in the program text, indexed by the loop variables of its enclosing loops, i.e., `p[i]`, and refer to the freshly allocated (but deterministically chosen) address as `&new:p[i]`.

Here, we use `p` as an abbreviation for the precise path in the program tree, i.e., `root.snd.snd.while.fst`, and `p[i]` is an abbreviation for the indexed path `root.snd.snd.while[i].fst`.

The notation `p[i]` already suggests that we can treat `p` just like an array of objects. After loop iteration `i`, `x` and `y` contain the address `&new:p[i]` of the latest allocated object:

```
let x = λ(i). if i ≥ 0 then &new:p[i] else null
let y = λ(i). if i ≥ 0 then &new:p[i] else ⊥, // ⊥ = uninitialized
```

The collection (array!) of objects `p` is defined and gets rewritten as follows:

```
let p = λ(i). if i ≥ 0 then p(i − 1)[i->[tail -> x(i − 1), head -> i]] else []
      = λ(i). array(i₂ < i). [tail-> x(i₂-1), head-> i₂]
      = λ(i). array(i₂ < i). [tail-> if i₂ > 0 then &new:p[i₂-1] else
         null, head-> i₂]
```

We can see how the recursive dependency between runtime objects is captured precisely. Note however that after simplification, p is no longer a recursive function: the address `&new:p[`$i_2 - 1$`]` is a purely syntactic term, which can be used to look up an object later by *dereferencing* the address.

After the first loop, the program state represents a proper store with static as well as dynamically allocated objects:

```
[ l -> n,
  x -> if n > 0 then &new:p[n-1] else null,
  y -> if n > 0 then &new:p[n-1] else ⊥
  p -> array(i < n). [tail -> if (i > 0) then &new:p[i-1] else
     null, head -> i]
```

As indicated before, the structure of the store is hierarchical and mirrors the program structure. Dereferencing an address entails accessing the store. We will use the notation σ`[addr]`, but need to keep in mind that for a composite address like `&new:p[0]`, two steps of lookup are necessary: first by `p`, and then by 0.

The second loop leads to the following definitions of `z`, `s`:

```
let z = λ(i). if i ≥ 0 then σ[z(i − 1)][tail] else x
let s = λ(i). if i ≥ 0 then s(i − 1) + σ[z(i − 1)][head] else 0
```

Simplification by rewriting yields the desired closed forms (remember $i < n$ as we are within the loop):

```
let z = λ(i). if n > 0 then { if i ≥ 0 then σ[z(i − 1)][tail] else
    &new:p[n−1] } else null
      = λ(i). if i ≥ 0 then &new:p[n−1−i] else null
let s = λ(i). if i ≥ 0 then (i + 1) ∗ i/2 else 0
```

Thus, we obtain the desired analysis result.

Expressions $\boxed{e \in \text{Exp}}$

$$n \in \text{Nat},\ b \in \text{Bool},\ x \in \text{Name}$$

$e ::=$

$n \mid b \mid \&x$	Constant (nat, bool, addr)
$e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2$	Arithmetic
$e_1 < e_2 \mid e_1 = e_2 \mid e_1 \wedge e_2 \mid \neg e$	Boolean
$e_1[e_2]$	Field read

Statements $\boxed{s \in \text{Stm}}$

$s ::=$

$x := \text{new}$	Allocation
$e_1[e_2] := e_3$	Assignment
if e then s_1 else s_2	Conditional
while e do s	Loop
$s_1;\ s_2$	Sequence
skip	No-op
abort	Error

Syntactic Sugar

x	$\equiv \&x[0]$
$x := e$	$\equiv x[0] := e$
$e.x$	$\equiv e[\text{fieldId}(x)]$
assert e	$\equiv$ if e then skip else abort

Figure 5.3.: `IMP`: Surface language syntax.

Throughout this section, we have glossed over some details. For example, we did not include explicit error checks in our translation, but checks for, e.g., validity of field accesses, need to be accounted for, and the details are described in Section 5.3.2. We also did not bother with variables that were obviously loop invariant. In reality, it is part of our analysis' job to determine which variables are the loop-invariant ones. We will return to this question in Section 5.4.

5.3 Formal Model

Since our approach hinges on translating imperative to functional code and applying transformation and simplification rules, it is imperative to formally establish the correctness of all these components to ensure the overall soundness of the approach.

Runtime Structures

$v \in \text{Val}$	$::= n \mid b \mid l$	Value (nat, bool, ptr)
$l \in \text{Loc}$	$::= \&x \mid \&\text{new}{:}c$	Location (static, dynamic)
$o \in \text{Obj}$	$: \text{Nat} \rightharpoonup \text{Val}$	Object
$\sigma \in \text{Sto}$	$: \text{Loc} \rightharpoonup \text{Obj}$	Store

$c \in \text{Ctx} ::=$	Context path
root	At top level
$c.\text{then} \mid c.\text{else}$	In conditional
$c.\text{fst} \mid c.\text{snd}$	In sequence
$c.\text{while}[n]$	In loop (iteration n)

Expression Evaluation $\boxed{\sigma \vdash e \Downarrow v}$

$$\sigma \vdash n \Downarrow n \ (\text{ENum}) \qquad \frac{\sigma \vdash e_1 \Downarrow n_1 \qquad \sigma \vdash e_2 \Downarrow n_2}{\sigma \vdash e_1 + e_2 \Downarrow n_1 + n_2} \ (\text{EPlus})$$

$$\frac{\sigma \vdash e_1 \Downarrow l_1 \qquad \sigma \vdash e_2 \Downarrow n_2 \qquad \sigma[l_1] = o \qquad o[n_2] = v_3}{\sigma \vdash e_1[e_2] \Downarrow v_3} \ (\text{EField})$$

Loop Evaluation $\boxed{\sigma, c \vdash (e\ s)^n \Downarrow \sigma'}$

$$\sigma, c \vdash (e\ s)^0 \Downarrow \sigma \ (\text{EWhileZero}) \qquad \frac{\sigma, c \vdash (e\ s)^n \Downarrow \sigma' \qquad \sigma' \vdash e \Downarrow \text{true} \qquad \sigma', c.\text{while}[n] \vdash s \Downarrow \sigma''}{\sigma, c \vdash (e\ s)^{n+1} \Downarrow \sigma''} \ (\text{EWhileMore})$$

Statement Evaluation $\boxed{\sigma, c \vdash s \Downarrow \sigma'}$

$$\sigma, c \vdash x := \text{new} \Downarrow \sigma[\&\text{new}{:}c \mapsto [], \&x \mapsto [0 \mapsto \&\text{new}{:}c]] \ (\text{ENew})$$

$$\frac{\sigma \vdash e_1 \Downarrow l_1 \qquad \sigma \vdash e_2 \Downarrow n_2 \qquad \sigma \vdash e_3 \Downarrow v_3 \qquad \sigma[l_1] = o}{\sigma, c \vdash e_1[e_2] := e_3 \Downarrow \sigma[l_1 \mapsto o[n_2 \mapsto v_3]]} \ (\text{EAssign})$$

$$\frac{\sigma \vdash e \Downarrow \text{true} \qquad \sigma, c.\text{then} \vdash s_1 \Downarrow \sigma'}{\sigma, c \vdash \text{if } e \text{ then } s_1 \text{ else } s_2 \Downarrow \sigma'} \ (\text{EIfTrue}) \qquad \frac{\sigma \vdash e \Downarrow \text{false} \qquad \sigma, c.\text{else} \vdash s_2 \Downarrow \sigma'}{\sigma, c \vdash \text{if } e \text{ then } s_1 \text{ else } s_2 \Downarrow \sigma'} \ (\text{EIfFalse})$$

$$\frac{\sigma, c \vdash (e\ s)^n \Downarrow \sigma' \qquad \sigma' \vdash e \Downarrow \text{false}}{\sigma, c \vdash \text{while } e \text{ do } s \Downarrow \sigma'} \ (\text{EWhile})$$

$$\frac{\sigma, c.\text{fst} \vdash s_1 \Downarrow \sigma' \qquad \sigma', c.\text{snd} \vdash s_2 \Downarrow \sigma''}{\sigma, c \vdash s_1;\ s_2 \Downarrow \sigma''} \ (\text{ESeq})$$

$$\sigma, c \vdash \text{skip} \Downarrow \sigma \ (\text{ESkip})$$

Figure 5.4.: `IMP`: Relational big-step semantics (primitive constants and operators other than n and $+$ elided). Note the evaluation rules for `while` loops and the role of program context c for address allocation in rule (ENew).

In addition, our structured store allocation model incurs some subtleties that warrant a formal description that explains the connection with standard store semantics.

Runtime Structures

$$o \in \text{Obj} \;:\; \text{Nat} \to \text{Option Val} \quad \text{Object}$$
$$\sigma \in \text{Sto} \;:\; \text{Loc} \to \text{Option Obj} \quad \text{Store}$$

Monad operations:

$$m \in \text{Option } T \;::=\; \text{None} \mid \text{Some } \tau \;\text{ where } \tau \in T$$
$$x \leftarrow m;\; f(x) \;=\; m \gg\!= f$$
$$\gg\!= \;:\; \text{Option } T \to (T \to \text{Option } U) \to \text{Option } U$$
$$\text{getOrElse} \;:\; \text{Option } T \to T \to T$$
$$\text{toNat} \;:\; \text{Val} \to \text{Option Nat}$$
$$\text{toBool} \;:\; \text{Val} \to \text{Option Bool}$$
$$\text{toLoc} \;:\; \text{Val} \to \text{Option Loc}$$

Iteration primitive:

$$\# \;:\; (\text{Nat} \rightharpoonup \text{Bool}) \rightharpoonup \text{Nat}$$
$$\#f \;=\; g(0) \text{ where } g(i) = \text{if } f(i) \text{ then } i \text{ else } g(i+1)$$

Expression Evaluation $\boxed{[\![\ e\ ]\!](\sigma) = v}$

$$[\![\ . \ ]\!] \;:\; \text{Exp} \to \text{Sto} \to \text{Option Val}$$
$$[\![\ n\ ]\!](\sigma) \;=\; \text{Some } n$$
$$[\![\ e_1 + e_2\ ]\!](\sigma) \;=\; n1 \leftarrow [\![\ e_1\ ]\!](\sigma) \gg\!= \text{toNat};\; n2 \leftarrow [\![\ e_2\ ]\!](\sigma) \gg\!= \text{toNat};\; \text{Some } (n1 + n2)$$
$$\ldots \;=\; \ldots$$
$$[\![\ x\ ]\!](\sigma) \;=\; o \leftarrow \sigma[\&x];\; o[0]$$
$$[\![\ e_1[e_2]\ ]\!](\sigma) \;=\; l \leftarrow [\![\ e_1\ ]\!](\sigma) \gg\!= \text{toLoc};\; n \leftarrow [\![\ e_2\ ]\!](\sigma) \gg\!= \text{toNat};\; o \leftarrow \sigma[l];\; o[n]$$

Loop Evaluation $\boxed{[\![\ e\ s\ ]\!](\sigma, c)(n) = \sigma'}$

$$[\![\ . \ ]\!] \;:\; \text{Exp} \times \text{Stm} \to \text{Sto} \times \text{Ctx} \to \text{Nat} \rightharpoonup \text{Option Sto}$$
$$[\![\ e\ s\ ]\!](\sigma, c)(n) \;=\; f(n) \text{ where } f(0) = \text{Some } \sigma;\; f(n+1) = \sigma' \leftarrow f(n);\; \text{true} \leftarrow [\![\ e\ ]\!](\sigma') \gg\!= \text{toBool};\; [\![\ s\ ]\!](\sigma', c.\text{while}[n])$$

Statement Evaluation $\boxed{[\![\ s\ ]\!](\sigma, c) = \sigma'}$

$$[\![\ . \ ]\!] \;:\; \text{Stm} \to \text{Sto} \times \text{Ctx} \rightharpoonup \text{Option Sto}$$
$$[\![\ x := \text{new}\ ]\!](\sigma, c) \;=\; \sigma[\&\text{new}{:}c \mapsto [],\; \&x \mapsto [0 \mapsto \&\text{new}{:}c]]$$
$$[\![\ e_1[e_2] := e_3\ ]\!](\sigma, c) \;=\; l \leftarrow [\![\ e_1\ ]\!](\sigma) \gg\!= \text{toLoc};\; n \leftarrow [\![\ e_2\ ]\!](\sigma) \gg\!= \text{toNat};\; v \leftarrow [\![\ e_3\ ]\!](\sigma);\; o \leftarrow \sigma[l];\; \sigma[l \mapsto o[n \mapsto v]]$$
$$[\![\ \text{if } (e)\ s_1 \text{ else } s_2\ ]\!](\sigma, c) \;=\; b \leftarrow [\![\ e\ ]\!](\sigma) \gg\!= \text{toBool};\; \text{if } b \text{ then } [\![\ s_1\ ]\!](\sigma, c.\text{then}) \text{ else } [\![\ s_2\ ]\!](\sigma, c.\text{else})$$
$$[\![\ \text{while } e \text{ do } s\ ]\!](\sigma, c) \;=\; \sigma' \leftarrow [\![\ e\ s\ ]\!](\sigma, c)(n);\; \text{false} \leftarrow [\![\ e\ ]\!](\sigma') \gg\!= \text{toBool};\; \text{Some } \sigma' \text{ where } n = \#(\lambda i.(\sigma' \leftarrow [\![\ e\ s\ ]\!](\sigma, c)(i);\; b \leftarrow [\![\ e\ ]\!](\sigma') \gg\!= \text{toBool};\; \text{Some } \neg b)\ \text{getOrElse true})$$
$$[\![\ s_1;\; s_2\ ]\!](\sigma, c) \;=\; \sigma' \leftarrow [\![\ s_1\ ]\!](\sigma, c.\text{fst});\; [\![\ s_2\ ]\!](\sigma', c.\text{snd})$$
$$[\![\ \text{skip}\ ]\!](\sigma, c) \;=\; \text{Some } \sigma$$
$$[\![\ \text{abort}\ ]\!](\sigma, c) \;=\; \text{None}$$

Figure 5.5.: `IMP` Functional semantics with explicit errors, partiality reserved for divergence. Note the use of monad operations throughout and the use of an iteration primitive in the evaluation of `while` loops.

In this section, we formalize the source and target languages and prove correctness of the translation, and the rewrite and simplification rules. This establishes the key result that an analysis engine that applies an arbitrary combination of equality-preserving simplifications will produce a sound result with respect to the semantics of the source language `IMP`. We have implemented our formal model in Coq and mechanized the results in this Section.

5.3.1 Source Language IMP

The syntax of our imperative model language IMP is defined in Figure 5.3. Our version of IMP is similar to imperative model languages found in a variety of textbooks, but is extended with dynamic memory operations and includes the possibility of certain runtime errors which reflect verification scenarios of practical interest.

IMP's syntax is split between expressions e and statements s. The language supports allocation statements $x := \text{new}$, as well as array or field references $e_1[e_2]$ (note that both array indices and fields can be computed dynamically) and corresponding assignments. Addresses of local variables $\&x$ are available as constant expressions, and variable references x are treated as syntactic sugar for dereferencing the corresponding address $\&x[0]$. Named field references $e.x$ are desugared into a numeric index assuming an injective global mapping `fieldId`. The `null` value is not part of the language, but can be understood as a dedicated address that is not otherwise used. It is important to note that there is no syntactic distinction between arithmetic and boolean expressions. Hence, evaluation may fail at runtime due to type errors or undefined fields. The syntax also includes an explicit `abort` statement for user-defined errors with `assert` as syntactic sugar.

Relational Semantics The semantics of IMP is defined in big-step style, shown in Figure 5.4. Many of the evaluation rules are standard: expressions evaluate to values, and statements update the store. A store σ is a partial function from locations l to heap objects o, which are partial functions from numeric field indexes to values. We use square brackets to denote store or object lookup, i.e., $\sigma[l] = o$ and $o[n] = v$, as well as update, i.e., $\sigma[l \mapsto o]$ and $o[n \mapsto v]$. However, two aspects of the semantics deserve further attention.

First, store addresses are not flat but have structure. For dynamic allocations, rule (ENew) deterministically assigns a fresh store address $\&\text{new}{:}c$ where c is the current program context c. This context is maintained throughout all statement rules and uniquely determines the *spatio-temporal* point of execution in the program. It

combines static information, i.e., the location in the program text, with dynamic information, i.e., progress of execution, represented by the current iteration vector of all enclosing loops. This will later enable us to talk about abstract locations from within the same loop, but at different iterations.

Second, `while` loops are executed with the help of an auxiliary judgement $\sigma, c \vdash (e\ s)^n \Downarrow \sigma'$, which characterizes the result of executing a loop body n times. This already hints at what we want to achieve later: replace iteration by a collective form for a given n. Declaratively, the number of iterations a loop will be executed is the particular n after which the condition becomes false. Operationally, rule (EWhile) has to "guess" the correct n. While not necessarily the best fit for deriving an *implementation*, this formulation of while loops renders the semantics compositional [138, 139], a good basis for deriving a semantics-preserving translation.

Given these differences, we first establish equivalence of the given semantics with a more standard formulation that assigns store locations nondeterministically, and defines while loops without explicit reference to iteration numbers.

Definition 5.3.1 (Standard Semantics) *Let $\Downarrow^0$ be the relation derived from $\Downarrow$ by dropping contexts c and replacing rules* (ENew) *and* (EWhile) *with the following rules:*

$$\frac{\&\text{new}{:}n \notin \sigma}{\sigma \vdash x := \text{new} \Downarrow \sigma[\&\text{new}{:}n \mapsto [], \&x \mapsto [0 \mapsto \&\text{new}{:}n]]} \quad \text{(ENewN)}$$

$$\frac{\sigma \vdash e \Downarrow^0 \text{true} \qquad \sigma \vdash s \Downarrow \sigma' \qquad \sigma' \vdash \text{while } (e)\ s \Downarrow^0 \sigma''}{\sigma \vdash \text{while } (e)\ s \Downarrow^0 \sigma''} \quad \text{(EWhileTrue)}$$

$$\frac{\sigma \vdash e \Downarrow^0 \text{false}}{\sigma \vdash \text{while } (e)\ s \Downarrow^0 \sigma} \quad \text{(EWhileFalse)}$$

Proposition 5.3.1 (Adequacy of $\Downarrow$) *$\Downarrow^0$ and $\Downarrow$ are equivalent, up to a bijection between store addrs &new:n and &new:c.*

We now study key properties of our semantics. First, we show that $\Downarrow$ is deterministic, and hence we can understand it as a partial function $\text{eval}_{\Downarrow}$.

Proposition 5.3.2 (Determinism) *The semantics is deterministic: if* $\sigma, c \vdash s \Downarrow \sigma'$ *and* $\sigma, c \vdash s \Downarrow \sigma''$ *then* $\sigma' = \sigma''$.

Definition 5.3.2 (Initial Store) *Let* $\sigma_\emptyset$ *be the store with* $\sigma_\emptyset[\&x] = []$ *and* $\sigma_\emptyset[\&new{:}c]$ *undefined for all* x *and* c.

Definition 5.3.3 *Let* $eval_\Downarrow(s) = \sigma$ *iff* $\sigma_\emptyset, root \vdash s \Downarrow \sigma$, *and undefined otherwise.*

Error Behavior As presented, the semantics does not distinguish error cases from undefinedness due to divergence. If our goal is program verification, then we need to isolate the error cases precisely and introduce a distinction.

Proposition 5.3.3 *For all* s, $eval_\Downarrow(s)$ *is either: (1) a unique result* σ, *(2) undefined due to divergence (i.e., there exists a loop in the program for which the condition is true for all* n *in rule* (EWhile)*), or (3) undefined due to one of the following possible errors: type error (Nat, Bool, Loc), reference to nonexistent store location, reference to nonexistent object field, explicit* `abort`

Proof We show that the property holds up to a given upper bound n on the number of iterations any loop can execute, and do induction over n. ■

Functional Semantics Based on these observations, we define a second semantics that makes all error conditions explicit by wrapping potentially failing computations in the `Option` monad, and which also replaces the nondeterminism in rule (EWhile) with an explicit and potentially diverging search for the correct number of iterations. With these modifications, the semantics can be expressed in a denotational style, directly as partial functions. We show the definition in Figure 5.5. Functions $[\![\, . \,]\!]$ now take the role of the relation $\Downarrow$, and partial functions that could be undefined due to runtime errors are now replaced by total functions that return an `Option` T instance, i.e., either `None` to indicate an error or `Some` τ with a T-value τ to signal success. The evaluation of expressions becomes entirely total. The only remaining

source of partiality is the potentially diverging computation of loop iterations $n = \#(\lambda i. \ldots)$.

Definition 5.3.4 *Let* $eval_{[\![\,]\!]}(s) = [\![\ s\ ]\!](\sigma_\emptyset, root)$.

Proposition 5.3.4 *For all s,* $eval_{[\![\,]\!]}(s)$ *is either: (1) a unique result* `Some` σ*, (2) an explicit error* `None`*, or (3) undefined due to divergence.*

Proposition 5.3.5 (Adequacy) *For all s,* $eval_{\Downarrow}(s)$ *and* $eval_{[\![\,]\!]}(s)$ *agree exactly on their value, error, and divergence behavior.*

Proof By induction on an assumed upper bound on the number of iterations per loop. ■

This functional semantics has the appealing property of denotational formulations that we can directly read it as a translation from `IMP` to mathematics. By mapping the mathematical notation into (a subset of) our target language `FUN`, we obtain a translator from `IMP` to `FUN`.

5.3.2 Target Language `FUN`

The syntax of `FUN` is defined in Figure 5.6. `FUN` is a functional language based on λ-calculus, with expressions g as the only syntactic category. The primitive data types are natural numbers, booleans, store addresses, and objects, i.e., records with numeric keys. In addition, `FUN` has a rich set of collective operators for sums, products, forall, and exists. While the presention here only covers those four monoids, the technique presented in this Chapter can be easily generalized to other monoids.

Figure 5.7 summarizes how the various entities in the definition of `IMP`'s functional semantics in Figure 5.5 map to `FUN` constructs. This enables us to directly read the given `IMP` semantics as translation rules.

Proposition 5.3.6 *Functions* $[\![\ e\ ]\!](\sigma)$ *and* $[\![\ s\ ]\!](\sigma, c)$ *accomplish translation from* ***IMP*** *to* ***FUN****.*

Expressions $\boxed{g \in \text{Exp}}$

$g ::=$			
$n \mid b \mid l \mid []$	Const (nat, bool, loc, obj)		
x	Variable	$e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2$	Arithmetic
$e_1 \ni e_2$	Field exists?	$e_1 < e_2 \mid e_1 = e_2 \mid \neg e$	Logic
$e_1[e_2]$	Field read	if e then s_1 else s_2	Conditional
$e_1[e_2 \mapsto e_3]$	Field update	letrec $x_1 = g_1, \ \ldots$ in g_n	Recursive let
$g_1(g_2)$	Application	$\Sigma(x < g_1).\ g_2$	Sum
$\lambda(x).\ g$	Function	$\Pi(x < g_1).\ g_2$	Product
$\#(x).\ g$	First index	$\forall\ (x < g_1).\ g_2$	Conjunction
$\langle . \rangle(x < g_1).\ g_2$	Sequence	$\exists\ (x < g_1).\ g_2$	Disjunction
		$\ldots$	
$w ::=$	Value		
$n \mid b \mid l$	Constant	$[n_0 \mapsto w_0, \ \ldots]$	Object

Figure 5.6.: `FUN`: Target language syntax.

Translation

$$
\begin{array}{rcl}
\text{None} & = & [\text{valid} \mapsto \text{false}] \\
\text{Some } g & = & [\text{valid} \mapsto \text{true}, \text{data} \mapsto g] \\
g \gg\!= f & = & \text{if } g.\text{valid then } f(g.\text{data}) \text{ else None} \\
g_1 \text{ getOrElse } g_2 & = & \text{if } g_1.\text{valid then } g_1.\text{data else } g_2 \\
\text{Val } n & = & [\text{tpe} \mapsto \text{nat}, \text{val} \mapsto n] \\
\text{Val } b & = & [\text{tpe} \mapsto \text{bool}, \text{val} \mapsto b] \\
\text{Val } l & = & [\text{tpe} \mapsto \text{loc}, \text{val} \mapsto l] \\
\text{toNat } g & = & \text{if } g.\text{tpe} = \mathit{nat} \text{ then Some } g.\text{val else None} \\
\text{toBool } g & = & \text{if } g.\text{tpe} = \mathit{bool} \text{ then Some } g.\text{val else None} \\
\text{toLoc } g & = & \text{if } g.\text{tpe} = \mathit{loc} \text{ then Some } g.\text{val else None} \\
o[n] & = & \text{if } o \ni n \text{ then } [\text{valid} \mapsto \text{true}, \text{data} \mapsto o[n]] \text{else } [\text{valid} \mapsto \text{false}] \\
\sigma[l] & = & \text{if } \sigma \ni l \text{ then } [\text{valid} \mapsto \text{true}, \text{data} \mapsto \sigma[l]] \text{else } [\text{valid} \mapsto \text{false}]
\end{array}
$$

Figure 5.7.: `FUN`: Mathematical notation as syntactic sugar, and value representation for the Option Monad. This enables reading Figure 5.5 as translation from `IMP` to `FUN`. Every monadic value has a `valid` flag that is set to false to indicate an error condition.

The semantics of `FUN` follows the standard call-by-value (CBV) λ rules. The collective operators trivially map to recursive definitions. The only non-trivial addition is the mapping of store locations to numeric keys for record access.

Definition 5.3.5 *Let $\rightarrow_v$ be the standard CBV λ reduction, extended to `FUN` with the following rules:*

$$w[\text{\&new:fst}.p] \to w[\text{fst}][\text{\&new}:p]$$
$$w[\text{\&new:snd}.p] \to w[\text{snd}][\text{\&new}:p]$$
$$w[\text{\&new:while}[n].p] \to w[\text{while}][n][\text{\&new}:p]$$
$$\ldots$$

Here we assume a standard mapping from names like 'fst', 'while', etc., to numbers, as before. We can see now that at the `FUN` level, the `IMP` store assumes a nested structure, mapping all values allocated in a loop into an array indexed by the loop variable. The update and field test rules are analogous to the lookup rules shown.

Proposition 5.3.7 *Evaluating a `FUN` expression via $\to_v^*$ can either: (1) terminate with a value, (2) terminate with a stuck term, or (3) diverge.*

We are now ready to express our main soundness result for the translation from `IMP` to `FUN`.

Definition 5.3.6 *Let $v \cong w$ be the equivalence between `IMP` values v and `FUN` values w induced by the value representation in Figure 5.7. Let $w \cong \sigma$ be the relation extended to `IMP` stores in the nested representation defined above.*

Theorem 5.3.1 *Translation from `IMP` to `FUN` is semantics preserving: For any `IMP` terms e or s translated to `FUN` via $[\![e]\!]$ or $[\![s]\!]$, `FUN` execution via $\to_v^*$ never gets stuck. Values and stores map to their equivalents $v \cong w$ and $\sigma \cong w$, errors map to clearly identified error values, and divergence to divergence.*

Proof Again, by induction on an appropriate upper bound on the number of any loop iterations. ■

5.3.3 Analysis and Verification via Simplification

Based on the sound translation from `IMP` to `FUN`, we now want to simplify `FUN` programs and extract higher-level information. In particular, we want to transform recursive definitions into collective operations. This approach to program analysis

is similar in spirit to deductive verification: we translate the source language into an equivalent representation (or program logic) which can then be solved through a solver procedure (e.g., constraint simplification). In our case, the target language is the functional language FUN, and the solver performs simplification through equality-preserving rewriting of program terms (we discuss a strategy that interleaves translation and simplification in Section 5.4). For the concrete rewriting strategy there is considerable freedom, and we do not fix a particular strategy here.

Structural Equivalences

$$
\begin{array}{rcl}
a[k \mapsto u][j] & \equiv & \text{if } j = k \text{ then } u \text{ else } a[j] \\
C[\text{if } c \text{ then } u \text{ else } v] & \equiv & \text{if } c \text{ then } C[u] \text{ else } C[v] \\
\text{letrec } f = (\lambda(i).[a \mapsto u, b \mapsto v]) \text{ in } e & \equiv & \text{letrec } f = (\lambda(i).[a \mapsto f_a(i), \mapsto f_b(i)]) \\
 & & \quad f_a = (\lambda(i).u), f_b = (\lambda(i).v) \text{ in } e \\
\text{if } 1 < e \text{ then } x \text{ else } y & \equiv & \text{if } 0 < e \text{ then } x \text{ else } y \\
 & & \quad \textit{if } x \equiv y \textit{ when } e = 0 \\
\text{if } a < e \text{ then if } b < e \text{ then } x' \text{ else } y' \text{ else } y & \equiv & \text{if } b < e \text{ then } x' \text{ else } y' \\
 & & \quad \textit{if } y \equiv y' \textit{ and } a \leq b \\
 & \ldots & \\
\end{array}
$$

Collective Forms

$$
\begin{array}{rcl}
\text{letrec } f = (\lambda(i).\ \text{if } 0 \leq i \text{ then} & & \\
\quad f(i-1)[i \mapsto g_i] \text{ else } []) \text{ in } f(a) & \equiv & \langle . \rangle (i < a+1).\ g_i \\
\text{letrec } f = (\lambda(i).\ \text{if } 0 \leq i \text{ then} & & \\
\quad f(i-1) + g_i \text{ else } 0) \text{ in } f(a) & \equiv & \Sigma(i < a+1).\ g_i \\
x + \Sigma(i < a).\ g_i & \equiv & \Sigma(i < a+1).\ g_i \\
 & & \quad \textit{if } x \equiv g_i[i := a] \\
\Sigma(i < n).\ i & \equiv & \text{if } 0 < n \text{ then } n * (n-1)/2 \text{ else } 0 \\
\#(i).\ \neg(i < a) & \equiv & \text{if } 0 < a \text{ then } a \text{ else } 0 \\
\text{if } 0 < \#(i).\ g \text{ then } s1 \text{ else } s2 & \equiv & s1 \quad \textit{if } s1[(\#(i).\ g) := 0] \equiv s2 \\
 & \ldots & \\
\end{array}
$$

Figure 5.8.: Selected equivalences (non-exhaustive) that can be used as simplification rules, using a variety of deterministic or nondeterministic rewriting strategies ($x[y := z]$ means that y is substituted by z in x).

Verification Based on Explicit Errors Values The key property of the IMP to FUN translation was that any runtime error in the IMP program will be reflected as an observable error *value* in the target language, but not trigger erroneous behavior there (Theorem 5.3.1). Based on this property, verification just amounts to checking

that the FUN program cannot produce a failure result. All that is required for this test is a syntactic check that the FUN program after simplification is equal to Some g—in other words, that the `valid` flag according to the value representation in Figure 5.7 statically simplifies to constant `true`. If the `valid` flag is any other symbolic expression, it means that verification did not succeed; i.e., that the program might exhibit erroneous behavior (such as an assertion failure).

Soundness of Simplification Rules The property that any error in the IMP program will be reflected as an observable error value in FUN follows from the semantic preservation of the CBV FUN semantics. For purposes of verification, however, we may settle for a weaker correspondence, and pick a non-strict call-by-name semantics for FUN. This provides more flexibility for simplification, e.g., the ability to rewrite $0 * e \to 0$ even if evaluation of e may not terminate, but it also means that some diverging IMP programs may terminate in their FUN translation. In this case, verification may signal false positive errors. For example, for `while true do skip; assert false`, the analysis might miss that the assert is unreachable. Importantly, this result is still sound.

In the following, we therefore assume a non-strict call-by-name (CBN) or call-by-need semantics for FUN, and recall confluence of λ-calculus and that CBN terminates on more programs that CBV. We need a few other standard results:

Definition 5.3.7 *Let $\to$ be the standard CBN λ reduction, extended to FUN as above. Let $\mathcal{E}[\![\ g\ ]\!]$ be the partial evaluation function induced by $\to^*$.*

Definition 5.3.8 (Behavioral Equivalence) *Let $g_1 \equiv g_2$ iff $\mathcal{E}[\![\ g_1\ ]\!] = \mathcal{E}[\![\ g_2\ ]\!]$.*

Proposition 5.3.8 (Congruence) *For any context C, if $g_1 \equiv g_2$, then $C[g_1] \equiv C[g_2]$*

The congruence property enables us to prove the correctness of individual rewrite rules, and use them to soundly replace parts of an expression with behaviorally equivalent ones.

Simplification Rules in Practice We show selected equivalences that give rise to useful rewrite rules for simplification in Figure 5.8. Besides standard arithmetic simplification, there are structural rules about objects and their fields. In particular, a key simplification is to split recursive functions into individual functions per object field. Since the `IMP` store is represented as a `FUN` object, this rule enables local reasoning about individual `IMP` variables, instead of only about the store as a whole. Combined with β-reduction for non-recursive functions, the original function may be replaced entirely by the component-wise ones. The same pattern also applies to the construction of sequences: instead of creating an array of objects, it is often better to create an object of arrays, one per field. If only parts of an object change, this enables a more detailed characterization of such changes. In addition, it is often helpful to distribute conditionals over other operations, e.g., to push conditionals into object fields. Another important set of rules is concerned with the actual extraction of collective forms for sums, sequences/arrays, etc. The $\#(i)$ rule is key for numeric loop bounds. It is also useful to add standard dead-code and common subexpression rules. The last rewriting rule related to $\#(i)$ in Figure 5.8 is quite interesting. After analyzing a loop, the resulting store will always be of the form: if the loop executed at least once ($0 < \#(i).g$) the new store is $s1$ otherwise it is $s2$. However, if the loop did not execute, $\#(i).g$ must be 0. Therefore if $s1$ is equivalent to $s2$ when the loop does not execute, then the condition can be removed, and the new store is $s1$.

We believe that it is a strong benefit of our approach that the set of simplification rules (Figure 5.8 and beyond) is not fixed and can be extended at any point. The only requirement for a rule is to individually preserve the CBN concrete semantics.

Rewriting Strategies The set of equivalence rules available for simplification, including the rules in Figure 5.8 and beyond, gives rise to a whole space of rewriting opportunities. If each rule individually is proved to preserve semantics, an implementation is free to apply them in any order to reach a sufficiently simplified program. A simple and performance-oriented implementation can use a deterministic bottom-

up strategy, but it would be entirely feasible to use auto-tuning, heuristic search, or strategies based on machine learning.

However, even search-based rewriting strategies are fundamentally limited by their pessimistic nature if they apply simplification rules one by one, due to the requirement that each individual rule preserves the program semantics, as opposed to a set of rules applied at once. Section 5.4 discusses an optimistic strategy based on Kleene iteration that removes this restriction and leads to more precise results in practice.

5.4 Speculative Rewriting & Kleene Iteration

The analysis and verification approach presented in Section 5.3.3 is based on applying equality-preserving simplification rules after a full `IMP` program is translated to `FUN`. While a useful starting point, this approach has clear limitations.

Fundamentally, equality-preserving simplification has to operate with *pessimistic* assumptions around loops and other recursive dependencies. We can only simplify a program if we are *sure* that each individual step will preserve the full extent of the program's semantics. In this section we will refine our approach towards *optimistic* simplification: this approach will simplify loop bodies no matter what, and *check* whether the simplification is indeed valid. If not, we try somewhat less optimistic assumptions, and repeat. This is inspired by Lerner et al.'s work on composing dataflow analyses and transformations in optimizing compilers [95].

Errors and Loop-Invariant Fields Concretely, equality-preserving rewriting works well as long as there are no mutually recursive dependencies, i.e., there is always one recursive function that can be rewritten first, leading to further rewriting opportunities in other functions. But this is not always the case. Consider the following program:

```
j := 0;
while j < n do {
  assert(j >= 0); j := j + 1
}
```

Recall that errors are represented as a `valid` flag in the record representing the overall program state (see Section 5.3). The `valid` flag is equivalent to a variable `v` initialized to `true` at the beginning of the program, and set to `false` if the `assert` fails. To illustrate, the program could be rewritten as follows:

```
j := 0;
v := true;
while j < n ∧ v do {
  if j >= 0 then j := j + 1 else v := false
}
```

To demonstrate the absence of errors, we need to demonstrate that the `valid` flag remains unchanged throughout the loop. However, this is difficult since the derived `FUN` representation contains mutual recursion between variables.

```
let j = λ(i). if i ≥ 0 then { if v(i − 1) ∧ j(i − 1) ≥ 0 then j(i − 1) + 1 else
    j(i − 1) } else 0
let v = λ(i). if i ≥ 0 then { if v(i − 1) ∧ j(i − 1) ≥ 0 then v(i − 1) else
    false } else true
let n = #(i). ¬(j(i) < n ∧ v(i))
```

In this situation, we cannot extract an individual recursive function for `j` (and much less a collective form) because the loop body may raise an error (set `v` to false), and we cannot eliminate `v` because we do not have enough knowledge about `j`. Thus the basic rewriting strategy from Section 5.2 cannot work.

We will explain the process in more detail based on a concrete example of scalar recurrences below, but it is important to note that the approach is more general and applies to all kinds of expressions and data types. To complete the verification of the program above, recall that error conditions are represented as a `valid` flag in the record representing the overall program state (see Section 5.3). To demonstrate the absence of errors, we need to demonstrate that the `valid` flag remains unchanged throughout a loop. Fortunately, identifying loop-invariant parts of data structures is straightforward with the speculative rewriting approach: we make initial optimistic assumptions that all variables and record fields (including the valid flag as special case), are loop-invariant, and roll back these assumptions only if writes to certain vars/fields are observed. With optimistic assumptions, there is no write to `v` in

the loop, and the program verifies. The following table illustrates the simplification process, that terminates once a fixpoint has been reached.

	Before loop	Before ith iter.	After (expected)	After (actual)
	y_0	$\hat{f}(i-1)$	$\hat{f}(i)$	$\Delta(\hat{f}(i-1))$
Step 1	j = 0	0	0	1
	v = true	true	true	true
Step 2	j = 0	i	i + 1	i + 1
	v = true	true	true	true

Scalar Recurrences Consider the example from Section 5.2:

```
j := 0; s := 0;
while j < k do {
  s := s + j;
  j := j + 1
}
```

Our refined approach is as follows: let y_0 be the program state before the loop and let Δ be the transfer function of the loop, describing the effect of one loop iteration on the program state. We use $f(i)$ to denote the program state after iteration i, subject to $f(-1) = y_0$ and $f(i+1) = \Delta(f(i))$. The goal is now to approximate f iteratively by a series of *increasingly pessimistic* functions $\hat{f}_k$ until we reach f.

At the first step $\hat{f}_0$ we assume (maximum optimism) all variables to be loop invariant, i.e., that we can use the following per-variable functions, where `n` is the current symbolic value of k: `let` k `=` $\lambda(i).$`n,` `let` j `=` $\lambda(i).0$, `let` s `=` $\lambda(i).0$

Then, we evaluate the assumed functions to compute the expected value before and after loop iteration `i`, i.e., $\hat{f}_0(i-1)$ and $\hat{f}_0(i)$. We compare the expected post-iteration value with the actual symbolic evaluation of the loop body, using the same expected initial values $\Delta(\hat{f}_0(i-1))$ (Figure 5.9, top). In general, if $\Delta(\hat{f}_k(i)) = \Delta(\hat{f}_k(i-1))$ for a symbolic representation of i, we know that $\hat{f}_k = f$. But in this case, we can see that our assumption about `j` was too optimistic. We need to try a non-loop-invariant transfer function — but which one?

For scalar values, one of our strategies is to focus on polynomials. The observed difference d_j between before and after the loop iteration can be seen as the discrete

Before loop y_0	Before ith iter. $\hat{f}(i-1)$	After (expected) $\hat{f}(i)$	After (actual) $\Delta(\hat{f}(i-1))$
k = n	n	n	n
j = 0	0	0	1
s = 0	0	0	0
k = n	n	n	n
j = 0	i	i + 1	i + 1
s = 0	0	0	i
k = n	n	n	n
j = 0	i	i + 1	i + 1
s = 0	((i-1)*i)/2	(i*(i+1))/2	(i*(i+1))/2

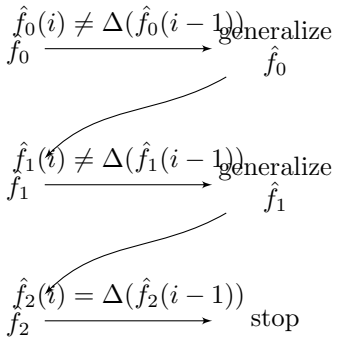


Figure 5.9.: Fixpoint iteration for running example, iterations 0 (top) to 2 (bottom), converging to a 2nd-degree polynomial for `s`. The generalization treats different data types differently: (1) try a higher degree of polynomial for numerics, (2) apply generalization to fields recursively for records, (3) extract the collective form for arrays if writing to the adjacent slot, or (4) create a recursive function for fallback.

derivative of the transfer function we are approximating. In this case, d_j is a constant, i.e., a polynomial in i of degree 0. Thus we try the (uniquely defined) polynomial of degree 1 (a linear function) that matches the observed values for $i = 0$, $j = 1$ and has derivative $d_j = 1$. `let` k `=` $\lambda(i).$`n,` `let` j `=` $\lambda(i).i+1$`,` `let` s `=` $\lambda(i).0$

The computed expected and actual values are shown in Figure 5.9 (middle). Now the representation of j has been settled, but s is no longer correct. We follow the same strategy as before and generalize the transfer function for s. The difference $d_s = i$ is a polynomial of degree 1, and discrete integration yields a quadratic function:

`let` k `=` $\lambda(i).$`n,` `let` j `=` $\lambda(i).i+1$`,` `let` s `=` $\lambda(i).(i*(i+1))/2$

Now we observe convergence, shown in Figure 5.9 (bottom). Therefore, our strategy succeeded and we simultaneously computed sound symbolic representations of k, j, and s. We can now compute the number of iterations executed: $\#(i).\ \neg(j(i) <$ n) = if 0 ¡ n then n else 0. Thus, the last iteration executed was n−1 (or -1 if the loop was not executed at all), and the values of k, j, and s after the loop are therefore:

```
[ k -> if 0 < n then k(n−1) else n = n,
  j -> if 0 < n then j(n−1) else 0 = if (0 < n) then n else 0,
```

```
s -> if 0 < n then s(n−1) else 0 = if (0 < n) then (n-1)*n/2 else
    0 ]
```

In general, polynomials are just one option. Since not all functions can be described as polynomials, we cannot rely on convergence, i.e., we need to stop at a certain degree. In the event that the analysis did not converge, it needs to stop and produce a conservative solution. The fallback (always valid) is to create a recursive definition:

$$\hat{f}_\omega(i) = \text{if } (i \geq 0)\ \Delta(\hat{f}_\omega(i-1)) \text{ else } y_0$$

This solution is the last resort for our analysis. Therefore, we can view the function space as partially ordered, from optimistic to pessimistic: $\hat{f}_0 \sqsubset \hat{f}_1 \cdots \sqsubset \hat{f}_\omega$.

Here, $\hat{f}_0$ can be polynomials of degree 0, $\hat{f}_1$ polynomials of degree 1 etc., with the recursive form $\hat{f}_\omega$ at the top of the chain. While polynomials are useful, other chains of functions would be possible (e.g., Fourier series). The Kleene iteration is subject to the usual conditions, i.e., that sequences of functions $\hat{f}_k$ picked during iteration must be monotonic in k and without infinite chains.

Detecting Sequence Construction Similar to the extraction of closed forms from scalar recurrences, we use speculative rewriting to extract collective forms for sequence construction. Consider the following program:

```
a := new;
j := 0;
while j < k do {
  a[j] := g(j);
  j := j + 1
}
```

Just like we speculate on a closed form for `j`, we recognize that the first loop iteration writes to index 0 in `a`, and we speculate that subsequent loop iterations will write to indexes 1, 2, etc. Hence, for the next Kleene iteration step we propose a collective form for `a`, and verify its validity in the next iteration. During this process, we notice that `j` is equal to the loop index, which means that `a` is being assigned at the loop index. Therefore we can assume that `a` is an array $\texttt{array}(i_2 < i).\ g(i_2)$ and continue

the iteration process. As explained in Section 5.2, extracting collective forms for heap-allocated data structures is key for reasoning about programs like the one in Figure 5.1.

5.5 Scaling up to C

In the preceding sections, we have instantiated our approach for a representative model language `IMP`. To validate this model in practice, we have built a prototype tool called `SIGMA` that applies essentially the same approach to C code. Compared to the formal model, there are several challenges posed by a large and realistic language. Two important features that `IMP` does not include are functions, and intraprocedural control flow other than `if` and `while`. These include in particular `goto`, `break`, `continue`, `switch/case`, etc.

`SIGMA` uses the C parser from the Eclipse project to obtain an AST from C source. `SIGMA` then computes a control-flow graph for each function in the AST, and converts it back into a structured loop form using standard algorithms [123, 166]. We chose this approach for its relative simplicity and consistency with the formal description. It would also be possible to adapt the fixpoint algorithm from Section 5.4 to work directly on control-flow graphs. As part of the iterative translation to a slightly extended `FUN` language, `SIGMA` resolves function calls and inlines the function body at the call site, which provides a level of context-sensitivity. A potentially more scalable and performant alternative would be to compute `FUN` summaries for each function separately, leading to a more modular analysis approach. `SIGMA` currently does not support recursive functions at the C source level, beyond inlining them up to a variable cutoff.

Our simplification approach is based (1) on normalizing rewrites using smart constructors, e.g. pushing a constant in a product to the right, and (2) on an explicit simplification procedure. The main ingredient here is a solver for linear inequalities over integers, which in our case consists of a custom implementation of the Omega test [117]. Other algorithms would also be feasible [48]. Instead of this integrated

implementation one could also consider invoking an external SMT solver. However, care must be taken to faithfully encode `FUN` terms, since the current generation of SMT solvers cannot directly represent collective operators.

Another feature that is required for realistic analysis is dealing with nondeterministic input, often called `havoc` or `r?`. `SIGMA` models this in a manner very similar to dynamic allocations: each call to `r?` is parameterized with the program context `r?(path)`, so that the results of different `r?` calls can be uniquely identified even on the symbolic level. We use this in Figures 5.10, 5.11, 5.12.

While Section 5.3 has focused on a formal soundness property for `IMP`, we do not make such claims for the full C language. In particular, `SIGMA` does not accurately model integer overflow, pointer arithmetic (beyond arrays), floating point computation, concurrency, and undefined behavior.

Analysis and verification of C code using `SIGMA` can currently only be considered sound for programs that do not use such features. These restrictions are not unreasonable, and are, for example, reflected in certain categories of the SV-COMP verification benchmarks.

Figure 5.10 and 5.11 illustrate complex control flow within loops. In Figure 5.10, `SIGMA` manages to infer the polynomial form of the variable `agg` during the approximation phase. However, in Figure 5.11, there is no such polynomial form, thus `SIGMA` generates a generic sum for variable `a` and `b`. While this generic form does not provide a lot of information about either `a` or `b`, it can be used to prove, through the algebraic properties of the sum, that the condition `a + b == 3*n` is always evaluated to true.

5.6 Experimental Results

We evaluate `SIGMA` in three different categories: program verification, program equivalence, and transformation of legacy code to DSLs. We use an Intel Core i7-7700 CPU with 32GB of RAM running Ubuntu 16.04.3 LTS.

```
int main() {
  // path p1
  int n = __VERIFIER_nondet_int();
  // path p2
  int m = __VERIFIER_nondet_int();
  int agg = 0;  int i = 0;
  __VERIFIER_assume(0 < m && 0 <= n);
  while (i < n) {
    // path p3(x18?) = <...>.while[x18?]
    if (i < m) agg += 3;
    else agg += 1;
    i += 1;
  }
  return 0;
}
```

```
Store σ_{x18?} after iteration x18? (constant values elided):
"&i"  ↦  [ (x18? + 1) : "int" ]
"&agg"  ↦  [ if ((x18? < r?(p2))) { ((x18? * 3) + 3) }
             else { ((r?(p2) * 2) + (x18? + 1)) } : "int" ]
Loop termination: u = #(x18?).!(x18? < r?(p1))
Final store σ_f = σ_{u-1} =
"&n"  ↦  [ r?(p1) : "int" ]
"&m"  ↦  [ r?(p2) : "int" ]
"&i"  ↦  [ r?(p1) : "int" ]
"&agg"  ↦  [ if ((r?(p1) < (r?(p2) + 1))) { r?(p1) * 3 }
             else { ((r?(p1) * 2) + r?(p1)) } : "int" ]
"return"  ↦  [ 0 : "int" ]
```

Figure 5.10.: SIGMA analysis result for a program with a conditional in a loop where the result can be expressed as a polynomial. Top: C source code. Bottom: the store inferred within the loop and the final store.

Verification We compare SIGMA with CPAchecker [23] and SeaHorn [69] on programs from or similar to the SV-COMP benchmarks [22]. We used the programs from the *loop-lit*, *loop-invgen*, and *recursive-simple-*$*$ categories of SV-COMP. CPAchecker won the 2018 SV-COMP competition, and both state-of-the-art tools scored highly in previous years. The goal is to assess the reachability of a given function call *__VERIFIER_error()* (an assertion evaluated to false triggers a call to this function as well). The expected result of the analysis is encoded in the filename, e.g., *false-unreach-call* means that the call marked unreachable can actually be executed,

```
int main() {
  int i, n, a, b;
  i = 0; a = 0; b = 0;
  // path p1
  n = __VERIFIER_nondet_int();
  __VERIFIER_assume(n >= 0 && n <= 1000000);
  while (i < n) {
    // path p2(x11?) = <...>.while[x11?]
    if (__VERIFIER_nondet_int()) {
      a = a + 1;  b = b + 2;
    } else { a = a + 2;  b = b + 1; }
    i = i + 1;
  }
  __VERIFIER_assert(a + b == 3*n);
  return 0;
}

Final store:
"&i"  ↦  [ r?(p1) : "int" ]
"valid"  ↦  1
"&a"  ↦  [ sum(r?(p1)) { x11? =>
             if (r?(p2(x11?))) 1 else 2
           } : "int" ]
"&n"  ↦  [ r?(p1) : "int" ]
"&b"  ↦  [ sum(r?(p1)) { x11? =>
             if (r?(p2(x11?))) 2 else 1
           } : "int" ]
"return"  ↦  [ 0 : "int" ]
```

Figure 5.11.: SIGMA analysis result for a program with a conditional in a loop where the result can not be expressed as a polynomial (sv-comp benchmark loop-lit/bhmr2007_true-unreach-call.c.i). We show the C source on the top. At the bottom, we show the final store.

whereas a *true-unreach-call* means that the error can never be triggered. The example in Figure 5.11 is a program from the *loop-lit* category.

First, we highlight nine challenging programs of our benchmark: three programs operate on singly linked lists, four programs use more than one non-nested loop and two other programs have nested loops. The results for these programs are shown in Figure 5.13. At first glance we can see two distinct behaviors between CPAchecker and SeaHorn. CPAchecker, while being quite slow, never gives an incorrect answer. SeaHorn, on the other hand, is fast and can sometimes give an incorrect (false positive) result. All three tools manage to handle the *false-unreachable-call* case, which can

```
int main() {
  // path p1
  int n = __VERIFIER_nondet_int();
  int agg = 0;
  int i = 0;
  __VERIFIER_assume(0 <= n);
  while (i < n) {
    agg += i; i += 4;
  }
  return 0;
}
```

```
Store σ_x11? after iteration x11? (constant values
    elided):
"&agg" ↦ [ (x11? * (x11? * 2)) + (x11? * 2)) : "int"
    ]
"&i" ↦ [ (x11? * 4) + 4: "int" ]
Loop termination: u = #(x11?).!((x11? * 4) < r?(p1))
Final store σ_f = σ_u−1 =
"&n" ↦ [ r?(p1) : "int" ]
"&i" ↦ [ ((r?(p1) + 3) / 4) * 4 : "int" ]
"&agg" ↦ [ (r?(p1) + 3) / 4) * (((r?(p1) + 3) / 4) *
    2)
              + ((r?(p1)+3)/4)* -2 : "int" ]
"return" ↦ [ 0 : "int" ]
```

Figure 5.12.: SIGMA analysis result for a program with a loop increment different from 1. We show the C source on the top. At the bottom, we show the store inferred within the loop and the final store.

Name	CPAchecker	SeaHorn	SIGMA
simple_built_from_end_true-unreach-call.i	TIMEOUT	250	273
list_addnat_false-unreach-call.i	2890	190	215
list_addnat_true-unreach-call.i	305560	170	215
loop_addnat_false-unreach-call.i	2830	190	285
loop_addnat_true-unreach-call.i	TIMEOUT	200	285
loop_addsubnat_false-unreach-call.i	3140	210	364
loop_addsubnat_true-unreach-call.i	TIMEOUT	230	364
nestedloop_mul1_true-unreach-call.i	OUT OF MEMORY	7280	405
nestedloop_mul2_true-unreach-call.i	TIMEOUT	240	365

Figure 5.13.: Results are in ms (TIMEOUT is set at 900s). The red cells indicate incorrect results (false positives).

be seen as the easy problem as it only requires to find a counterexample. However in the case of true-unreachable, the prover needs to check all possible values. The very big search space explains CPAchecker's timeouts. For SeaHorn, the true case appears

to be difficult, as the internal logic may overapproximate the problem and give an incorrect (false positive) result. `SIGMA`, by contrast, is very precise and can verify all the examples while being almost as fast as SeaHorn.

In order to have a more general idea of the performance of `SIGMA`, we look at three SV-COMP suites: *loop-lit*, *loop-invgen*, and *recursive-simple-*$*$. Figure 5.14 shows a summary view of the running time of the analysis, and compares `SIGMA` to CPAChecker and SeaHorn. In these graphs, inspired by a similar visualization by [170], each dot represents one program. The dots are placed at coordinates representing their analysis times. The samples under the identity line are the programs where `SIGMA` is faster then the other tool. The complete list of programs is shown in Figure 5.15. On the *loop-** benchmark, which is composed of programs with loops and computation over scalar values, `SIGMA` performs very well. It proves most of the reachability goals quickly. Out of the 37 programs, it times out on 3 programs that have complex control flow (jumps from within the then branch to the else branch). There are also 4 programs where `SIGMA` cannot make a decision; this is when the valid flag is not simplified to 0 or 1. This situation is indicated by a yellow box in Figure 5.15. For the *recursive-simple-*$*$ benchmark, even though `SIGMA` is not designed to handle recursion, on many of the programs, the verification is successful. `SIGMA` manages to verify the assertions through pure symbolic execution driven by inlining of function calls. We did not implement any additional analysis for recursive functions. For the programs where it fails, the execution dependents on an unknown value thus a simple symbolic execution can not terminate and `SIGMA` gives up after exhausting its internal inlining limit.

Program Equivalence Since `SIGMA` computes exact symbolic descriptions of the program state post-execution, it can also be used to test program equivalence. In the absence of side effects that read external input, we can run `SIGMA` on both programs and verify that the post-execution states have the same symbolic representation, up to symbolic rewriting. Let s_1 and s_2 be the two symbolic states, then we want to test

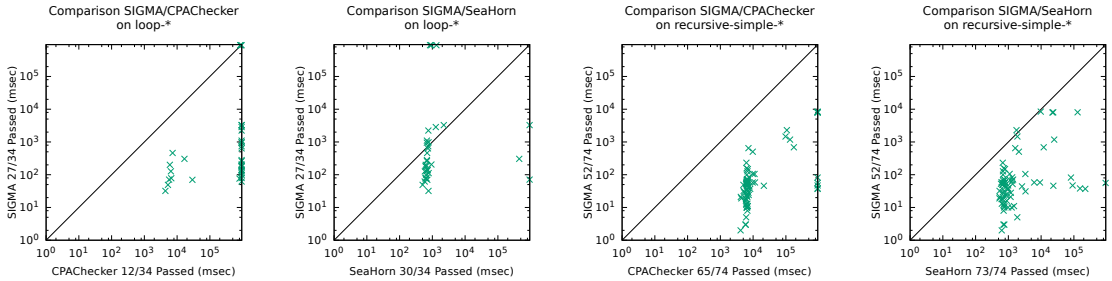


Figure 5.14.: Verification time of CPAChecker vs `SIGMA`, and SeaHorn vs `SIGMA`. Each point represents a program and is placed at coordinates (analysis time CPACHecker/SeaHorn, analysis time `SIGMA`).

whether $s_1 = s_2$ simplifies to true. Since errors are explicit at the symbolic level, this test not only applies to programs that independently verify, but can also relate the error behavior of two programs. In our evaluation, we manage to prove equivalence between the two functions below: one using a while loop, and the other one using GOTOs. For example, given the same random input, these two code snippets produce the same symbolic forms for the return value:

```
int main() {
  int a = nondet_int();
  int b = 0;
  while (b < a)
    b = b + 1;
  return b;
}
```

```
int main() {
  int a = nondet_int();
  int b = 0; goto cond;
more: b = 1 + b;
cond: if (b < a) goto more;
  return b;
}
```

This method can be generalized to global variables and even I/O. In the case of I/O, the different streams of input data can be modeled as data stored on the heap (see earlier discussion of `r?(p)`, and the symbolic forms need to match to prove equivalence.

In addition, we have verified that `SIGMA` can demonstrate equivalence of programs in the style of Section 5.2: (1) a program that computes the sum of n nondeterministic inputs in a loop; (2) a program that stores these value in an array and then computes the sum; (3) a program that stores these values in a linked list and then computes the sum.

Name	CPAchecker	SeaHorn	SIGMA
linv/apache-escape-absolute_true-unreach-call.i	848638	1329	TIMEOUT
linv/apache-get-tag_true-unreach-call.i	TIMEOUT	851	TIMEOUT
linv/down_true-unreach-call.i	TIMEOUT	650	98
linv/fragtest_simple_true-unreach-call.i	TIMEOUT	766	978
linv/half_2_true-unreach-call.i	TIMEOUT	717	142
linv/heapsort_true-unreach-call.i	TIMEOUT	2237	3275
linv/id_build_true-unreach-call.i	6632	661	78
linv/id_trans_false-unreach-call.i	7268	692	462
linv/large_const_true-unreach-call.i	16748	448736	305
linv/MADWiFi-encode_ie_ok_true-unreach-call.i	TIMEOUT	754	2232
linv/nest-if3_true-unreach-call.i	TIMEOUT	746	740
linv/nested6_true-unreach-call.i	TIMEOUT	1272	2853
linv/nested9_true-unreach-call.i	TIMEOUT	TIMEOUT	3267
linv/NetBSD_loop_true-unreach-call.i	TIMEOUT	631	177
linv/sendmail-close-angle_true-unreach-call.i	TIMEOUT	680	279
linv/seq_true-unreach-call.i	TIMEOUT	681	1101
linv/SpamAssassin-loop_true-unreach-call.i	867493	923	TIMEOUT
linv/string_concat-noarr_true-unreach-call.i	TIMEOUT	732	636
linv/up_true-unreach-call.i	TIMEOUT	621	183
llit/afnp2014_true-unreach-call.c.i	29254	700	70
llit/bhmr2007_true-unreach-call.c.i	TIMEOUT	769	112
llit/cggmp2005_true-unreach-call.c.i	4391	761	32
llit/cggmp2005_variant_true-unreach-call.c.i	TIMEOUT	583	62
llit/cggmp2005b_true-unreach-call.c.i	5998	628	201
llit/css2003_true-unreach-call.c.i	6469	664	127
llit/ddlm2013_true-unreach-call.c.i	TIMEOUT	934	205
llit/gj2007_true-unreach-call.c.i	5244	504	49
llit/gj2007b_true-unreach-call.c.i	TIMEOUT	714	136
llit/gr2006_true-unreach-call.c.i	5692	TIMEOUT	71
llit/gsv2008_true-unreach-call.c.i	TIMEOUT	679	952
llit/hhk2008_true-unreach-call.c.i	TIMEOUT	620	79
llit/jm2006_true-unreach-call.c.i	802315	625	76
llit/jm2006_variant_true-unreach-call.c.i	TIMEOUT	590	126
llit/mcmillan2006_true-unreach-call.c.i	TIMEOUT	683	259
rec-sim/afterrec_2calls_false-unreach-call.c	5296	673	29
rec-sim/afterrec_2calls_true-unreach-call.c	4377	534	19
rec-sim/afterrec_false-unreach-call.c	5041	659	49
rec-sim/afterrec_true-unreach-call.c	4151	629	2
rec-sim/fibo_10_false-unreach-call.c	9751	2100	504
rec-sim/fibo_10_true-unreach-call.c	9716	9217	58
rec-sim/fibo_15_false-unreach-call.c	107274	1772	2292
rec-sim/fibo_15_true-unreach-call.c	134059	25115	1182
rec-sim/fibo_20_false-unreach-call.c	TIMEOUT	22148	7934
rec-sim/fibo_20_true-unreach-call.c	TIMEOUT	129764	8049
rec-sim/fibo_25_false-unreach-call.c	TIMEOUT	81518	82
rec-sim/fibo_25_true-unreach-call.c	TIMEOUT	223645	37
rec-sim/fibo_2calls_10_false-unreach-call.c	9731	1096	106
rec-sim/fibo_2calls_10_true-unreach-call.c	11053	3298	105
rec-sim/fibo_2calls_15_false-unreach-call.c	97812	1887	1466
rec-sim/fibo_2calls_15_true-unreach-call.c	175109	11984	686
rec-sim/fibo_2calls_2_false-unreach-call.c	5143	538	32
rec-sim/fibo_2calls_2_true-unreach-call.c	4213	549	21
rec-sim/fibo_2calls_20_false-unreach-call.c	TIMEOUT	9728	8441
rec-sim/fibo_2calls_20_true-unreach-call.c	TIMEOUT	23486	8053
rec-sim/fibo_2calls_25_false-unreach-call.c	TIMEOUT	91274	47
rec-sim/fibo_2calls_25_true-unreach-call.c	TIMEOUT	154521	38

Figure 5.15.: Results are in ms (TIMEOUT is set at 900s). The red cells indicate incorrect results (false positives), and the yellow cells indicate when SIGMA could not decide (valid flag was not simplified to 0 or 1).

Name	CPAchecker	SeaHorn	SIGMA
rec-sim/fibo_2calls_4_false-unreach-call.c	6500	763	46
rec-sim/fibo_2calls_4_true-unreach-call.c	6322	980	10
rec-sim/fibo_2calls_5_false-unreach-call.c	6751	790	57
rec-sim/fibo_2calls_5_true-unreach-call.c	6531	1265	10
rec-sim/fibo_2calls_6_false-unreach-call.c	7328	883	24
rec-sim/fibo_2calls_6_true-unreach-call.c	6679	1349	52
rec-sim/fibo_2calls_8_false-unreach-call.c	8138	946	37
rec-sim/fibo_2calls_8_true-unreach-call.c	7730	2611	44
rec-sim/fibo_5_false-unreach-call.c	6557	795	60
rec-sim/fibo_5_true-unreach-call.c	6134	1873	5
rec-sim/fibo_7_false-unreach-call.c	7209	1190	28
rec-sim/fibo_7_true-unreach-call.c	6745	3332	31
rec-sim/id_b2_o3_true-unreach-call.c	6330	705	65
rec-sim/id_b3_o2_false-unreach-call.c	6915	790	149
rec-sim/id_b3_o5_true-unreach-call.c	6076	668	51
rec-sim/id_b5_o10_true-unreach-call.c	6223	596	53
rec-sim/id_i10_o10_false-unreach-call.c	6513	956	35
rec-sim/id_i10_o10_true-unreach-call.c	6067	678	18
rec-sim/id_i15_o15_false-unreach-call.c	6640	1169	21
rec-sim/id_i15_o15_true-unreach-call.c	6118	631	13
rec-sim/id_i20_o20_false-unreach-call.c	6827	1306	66
rec-sim/id_i20_o20_true-unreach-call.c	6197	624	18
rec-sim/id_i25_o25_false-unreach-call.c	6728	1484	11
rec-sim/id_i25_o25_true-unreach-call.c	6540	681	35
rec-sim/id_i5_o5_false-unreach-call.c	6327	817	24
rec-sim/id_i5_o5_true-unreach-call.c	5832	706	3
rec-sim/id_o10_false-unreach-call.c	7026	996	38
rec-sim/id_o100_false-unreach-call.c	11357	6152	58
rec-sim/id_o1000_false-unreach-call.c	TIMEOUT	TIMEOUT	56
rec-sim/id_o20_false-unreach-call.c	7386	1305	83
rec-sim/id_o200_false-unreach-call.c	21269	23597	46
rec-sim/id_o3_false-unreach-call.c	6454	777	69
rec-sim/id2_b2_o3_true-unreach-call.c	6274	739	55
rec-sim/id2_b3_o2_false-unreach-call.c	6476	723	74
rec-sim/id2_b3_o5_true-unreach-call.c	6169	670	57
rec-sim/id2_b5_o10_true-unreach-call.c	6313	619	59
rec-sim/id2_i5_o5_false-unreach-call.c	6384	731	23
rec-sim/id2_i5_o5_true-unreach-call.c	6038	895	16
rec-sim/sum_10x0_false-unreach-call.c	6437	985	47
rec-sim/sum_10x0_true-unreach-call.c	6678	754	13
rec-sim/sum_15x0_false-unreach-call.c	6805	1125	68
rec-sim/sum_15x0_true-unreach-call.c	6457	745	10
rec-sim/sum_20x0_false-unreach-call.c	6859	1393	71
rec-sim/sum_20x0_true-unreach-call.c	6693	716	8
rec-sim/sum_25x0_false-unreach-call.c	7189	1625	648
rec-sim/sum_25x0_true-unreach-call.c	6853	760	11
rec-sim/sum_2x3_false-unreach-call.c	6182	717	12
rec-sim/sum_2x3_true-unreach-call.c	5959	755	3
rec-sim/sum_non_eq_false-unreach-call.c	6739	687	127
rec-sim/sum_non_eq_true-unreach-call.c	6480	685	100
rec-sim/sum_non_false-unreach-call.c	6228	675	232
rec-sim/sum_non_true-unreach-call.c	6352	727	157

Figure 5.15.: Results are in ms (TIMEOUT is set at 900s). The red cells indicate incorrect results (false positives), and the yellow cells indicate when SIGMA could not decide (valid flag was not simplified to 0 or 1).

Legacy Code to DSLs SIGMA can also be used to analyze legacy optimized C code and translate it to high-level performance-oriented DSLs. In addition, the sym-

bolic representation obtained from the analysis can be used to better understand the program behavior. An interesting case is Stencil codes, which are patterns for updating array elements according to their neighbors. Many stencil codes are written in Fortran or C and heavily optimized for performance on a particular architecture, which precludes porting to GPUs, parallelizing and distributing across a cluster, or in general foward-porting the code to other emerging architectures.

A simple example is Jacobi iteration on a one-dimensional array. At each iteration, the algorithm updates each location with the arithmetic mean of its left-hand side and right-hand side values. For the out-of-bound locations, the default value is 1. From the C program on the left, `SIGMA` will generate the closed form on the right for the computation of a single iteration:

```
int i = 0; int ai = a[0];
a[0] = (1 + a[1])/2;
while (i < n-1) {
  int tmp = (ai + a[i+1])/2;
  ai = a[i]; a[i++] = tmp;
}
a[n-1] = (ai + 1)/2;
```

```
// Extracted FUN code of the
// corresponding C code:
let a = array(i < n).
  if (i == 0) then (1 + a[i+1])/2
  else if (i == n-1)
    then (a[i-1] + 1)/2
    else (a[i-1] + a[i+1])/2
```

From the derived closed form, the Jacobi algorithm is immediately apparent, despite the use of temporaries and loop-carried dependencies in the C source. The closed form `FUN` code is easily mapped to a high-performance DSL such as Halide [83, 103], which supports parallel CPU, GPU, and cluster execution. The process readily generalizes to multi-dimensional Jacobi iteration.

Legacy Code to Flare The motivation behind that work was to develop the capacity to apply Flare optimizations (parallelism, NUMA aware code, distributed, etc.) to legacy code. In this experiment, we illustrate how to achieve that goal. We manually implement a C program that computes the SQL query:

```
select avg(n_nationkey), count(*) from nation where n_regionkey == 0.
```

The nation table is part of the TPC-H benchmark that we used to evaluate Flare. The C version is as follows:

```
int main(int argc, argv** x1) {
```

```
  int n_nationkey_fd = open("../nation_n_nationkey", 0);
  int table_size = fsize(n_nationkey_fd);
  int* n_nationkey_col = (int*)mmap(0, table_size, PROT_READ,
     MAP_FILE | MAP_SHARED, n_nationkey_fd, 0);
  int n_regionkey_fd = open("../nation_n_regionkey", 0);
  int* n_regionkey_col = (int*)mmap(0, table_size, PROT_READ,
     MAP_FILE | MAP_SHARED, n_regionkey_fd, 0);
  struct result res;
  struct result* res_p = &res;
  res_p->n_nationkey_sum = 0;
  res_p->count = 0;
  int idx = 0;
  while (idx < table_size) {
    if (n_regionkey_col[idx] == 0) {
      res_p->n_nationkey_sum = res_p->n_nationkey_sum +
         n_nationkey_col[idx];
      res_p->count = res_p->count + 1;
    }
    idx++;
  }
  printf("%.4f|%ld|\n", (double)res_p->n_nationkey_sum /
     res_p->count, res_p->count);
  return 0;
}
```

We run SIGMA on this program and find the `valid` flag is set to 1, with the output of the program having the following value for "printf" (represents the values printed). We tweaked the output to make it more user-friendly by renaming long variables name and removing the reference to the `fsize` function. We can note the use of the `r?` oracle that we used previously, to indicate that the value returned by `open` or the value read from an mmaped region is considered "unknown". However, it is important to remember that it is uniquely referenced through the structured heap indexing scheme, thus each value can be differentiated.

```
"printf" -> {
  "#1" -> [
    sum(table_size) { x53? =>
      if (r?((r?(((&open:1,"../nation_n_regionkey"),top)),x53?))
         == 0) {
        r?((r?(((&open:0,"../nation_n_nationkey"),top)),x53?))
      } else {
        0
      }
    } / sum(table_size) { x53 =>
      if (r?((r?(((&open:1,"../nation_n_regionkey"),top)),x53)) ==
         0) { 1 } else { 0 }
    } : "double" ],
```

```
  "#2" -> [
    sum(table_size) { x53 =>
      if (r?((r?(((&open:1,"../nation_n_regionkey"),top)),x53)) ==
          0) { 1 } else { 0 }
    } : "int" ]
}
```

We extract from the output a representation of the computation as a Spark Dataframe. The extraction program is using some additional knowledge on the way the table is saved in file. For example, each column is stored in a file with the name: <table_name>_<column_name>. For our example, the DataFrame generated is as follows:

```
nation.filter("n_regionkey == 0").agg(sum(n_nationkey) / count("*"),
count("*"))
```

Once we obtain this DataFrame, Flare can generate code that is parallel, NUMA aware, or even distributed. In addition, the query can be specialized to run on larger datasets or different input formats.

5.7 Related Work

Decompiling to High-Level Languages A key ingredient of our approach is to transform—in a sense, "decompile"—a low-level language into a comparatively higher level language. Our approach has been greatly inspired by previous work in this direction. Some classics are the GOTO-elimination algorithm by Ramshaw [123] (newer work in this direction includes [166]), and the realization that compilers or analyzers for imperative languages based on SSA form are essentially using a functional intermediate language [11]. Various compiler frameworks (e.g., [21]) use rewriting rules to drive simplification and analysis, though these approaches typically do not address challenges such as those introduced by dynamic allocation.

More recently, there has been a flurry of work that aims to translate low-level imperative code to high-performance DSLs. Some works are based on a technique described as verified lifting, which is used to transform stencil codes to the Halide DSL

[83, 103], or to transform imperative Java code to Hadoop for cluster execution [5]. Another line of work uses symbolic execution to parallelize user-defined aggregations [124]. An approach closely related to ours transforms Java code to a functional IR and then to Apache Spark, after a rewriting and simplification process that, e.g., maps loop-carried dependencies to group-by operations [119]. There is also work on synthesizing MapReduce programs from sketches [140], on defining language subsets that are guaranteed to have an efficient translation [129], and work in the space of just-in-time compilers to reverse-engineer Java bytecode at runtime and redirect imperative API calls to embedded DSLs [132].

High-Performance DSLs Some notable works in the DSL space include Delite [32, 94, 133, 143], Halide [120], and Accelerate [38, 99, 145, 146, 158]. Most of these systems come with expressive, functional IRs. Some systems focus explicitly on the intermediate layers, for example Lift [141, 142], PENCIL [17], or the parallel action language [97].

Analysis and Optimization Our optimistic fixpoint approach is directly inspired by Lerner et al.'s work on composing dataflow analyses and transformations [95]. Related work has aimed to automatically prove the correctness of compiler optimizations [96], and on generating compiler optimizations from proofs of equivalence and before/after examples [152]. A related line of work models a space of possible program transformations given by equivalence rules through the notion of equality saturation, based on a program equivalence graphs (PEGs) [153] as IR. The PEG model has heavily inspired early versions of our work. The reasoning-by-rewriting approach and the avoidance of phase-ordering issues is similar, as is the overall goal of a flexible semantics-preserving representation as a basis for various kinds of analysis. However there are important differences: PEGs do not include collective forms except the `pass` operator, which is similar to our `#`. The θ operator in the PEG model describes standard recurrences, not collective forms. Tate et al. [153] also do not discuss specifics about heap-allocated data, and the accompanying Java analysis tool Peggy maps all

heap objects into a single summarization object. Thus, while PEGs can express program equivalence in general, Peggy could not prove the equivalences in Section 5.6, nor verify the linked list program in Figure 5.1. The two innovations we propose, collective forms and structured heaps, could be implemented without difficulty in a PEG setting, and potentially improve precision.

Recurrence Analysis Analyzing integer recurrences has been an active topic of research. Some recent works include compositional recurrence analysis (CRA) [58,84], which aims to derive closed forms for recurrence equations and inequations. The approach is based on an algebraic representation of path expressions [151], refered to as Newtonian program analysis [125]. Earlier works include abstract acceleration of general linear loops [78], and a study of algebraic reasoning about P-solvable loops [90]. Efficient integer linear inequality solvers have been available for some time [48, 117]. Aligators [71] is a tool from the static analysis community, representative for highlighting some of the limitations. Given a simple loop as input, Aligators can extract quantified scalar invariants, using a recurrence solving technique similar to the one used in our framework. However, like many other tools, Aligators has limited applicability in that it only handles linear recurrences (polynomials of degree 1), does not handle nested loops, does not provide collective forms such as symbolic sums, does not handle dynamic allocation of arrays, and does not appear to support complex or nested conditions inside loops. Many compilers provide some form of recurrence analysis as part of their optimization suite, often based on Bachmann's chains of recurrences (CoR) model [16], and sometimes called *scalar evolution*. An example is the SCEV pass in LLVM. These analyses are able to infer closed-form representations for simple counting loops but are limited in ways similar to tools like Aligators with respect to dynamic allocations, collective forms, and complex expressions.

Heap Abstraction Recent work on efficient and precise points-to analysis models the heap by merging equivalent automata [150]. Other works use structured heaps to model container data structures [49], and some techniques have been proposed for

heap abstractions that enable sparse global analyses for C-like languages [108], similar in spirit to SSA form. While SSA is typically used for local variables, techniques under the umbrella name Array SSA exist to extend sparse reasoning to heap data [86]. Our simplificaton rules that break apart heap objects to expose their fields are inspired by such techniques. Abstracting abstract machines [76] described different kinds of allocation policies parameterized by an abstract clock. This line of work has been inspirational for our structured heap representation, which differs in modeling the heap structure after the syntactic structure of the program. Many other directions exist, e.g., predicate abstraction for heap manipulation programs [25].

Shape analysis [135] provides a parametric framework for specifying different abstract interpretations. In each instantiation of the framework, a set of possible runtime stores is represented by a set of 3-valued logical structures. An individual in a 3-valued structure represents a set of runtime objects: each individual represents all objects in a runtime store that have the same values for a chosen set of properties of objects. (Different instantiations of the framework are created by making different choices of which object-properties to use.) A 3-valued structure does not represent a static partition of the runtime objects; for instance, in a loop the properties of a given object o can change from iteration to iteration, and hence the individual that represents o would be different on different iterations. Stated another way, a given individual in a 3-valued structure can represent different objects when considered to be the abstraction of the runtime stores that arise on different iterations. Our work takes a different approach, by indexing objects via an abstract notion of time: all objects allocated in a loop are considered to be a sequence (i.e., a collective form) indexed by the (symbolic) loop variable. It remains to be seen whether the two approaches could be combined, and what the advantages of such a combination might be.

Shape analysis approaches based on separation logic [30,126] improve precision and scale to large codebases [35,50], implemented, e.g., in Facebook's Infer tool [34]. With its support for reasoning about linked list and related structures via bi-abduction [35],

Infer should in principle come close to verifying programs like the one in Figure 5.1; however it still fails on this particular example and several variations we tried on the public Infer web interface. Since Infer does not compute precise symbolic representations, however, it is unsuited for tasks like translating legacy code to DSLs (Section 5.6). An interesting avenue for future research is how our heap representation can form a basis for and interact with separation predicates. This could, e.g., enable support for modular analyses that use a precise partial heap model within a function, and approximate separation predicates for function contracts.

Gopan et al. [67] extend the ideas from shape analysis à la Sagiv et al. [135] to indexed elements in arrays, thereby creating a parametric framework for creating abstract interpreters that can establish certain kinds of relationships among array elements. Their approach is based on splitting the collection of array elements into partitions based on index values that satisfy common properties, e.g., $< i$, $= i$, or $> i$, where i is a loop-counter variable. Additional predicates are introduced to hold onto invariants of elements that have been coalesced into a single partition. Instantiations of the framework are capable of establishing that (i) an array-initialization loop initializes all elements of an array (and that certain numeric constraints hold on the values of the initialized elements); (ii) an array-copy loop copies all elements from one array to another; and (iii) an insertion-sort routine sorts all of the elements of an array.

The idea of representing program values in terms of an execution context that captures the current loop iteration is also present in previous work on dynamic program analysis [164] and on polyhedral compilation [20]. The main difference in our work is that we push the indexing idea all the way into the store model and allocation scheme, which permits effective static reasoning about dynamic allocations and linked data structures, and that we use the indexing scheme as a basis for a generic symbolic representation and static analysis.

Semantics The proofs and formal models presented in this Chapter are largely standard, but make key use of induction over a numeric "fuel metric" that bounds

the amount of work (in this case, loop iterations) a program is allowed to do. Such techniques enable effective proofs for functional formulations of big-step semantics and have only recently received wide-spread interest [9,111]. Existentially quantifying over the number of loop iterations in our `IMP` semantics is very similar to a recent proposal by Siek [138, 139].

5.8 Conclusion

In this Chapter, we identified two key limitations of current progam analysis techniques: (1) the low-level and inherently *scalar* description of program entities, and (2) collapsing information per program point, and projecting away the dimension of time. As a remedy, we proposed first-class collective operations, and a novel structured heap abstraction that preserves a symbolic dimension of time. We have elaborated both in a sound formal model, and in a prototype tool that analyzes C code. The Chapter includes an experimental evaluation that demonstrates competitive results on a series of benchmarks. Given its semantics-preserving nature, our implementation is not limited to analysis for verification, but our benchmarks also include checking program equivalence, and translating legacy C code to Flare optimized code.

6 CONCLUSION

In this dissertation we have demonstrated the advantages of using meta-programming and deferred APIs to accelerate data-centric systems.

In Chapter 2 we analyzed the key impediments to performance for data-centric systems and proposed a solution to remove them through low-level deferred API. We presented Flare, which compiles SQL query plans to efficient low-level C code. We illustrated how Flare maximizes the performance of a single node architecture by generating parallel and NUMA aware code, and we also showed how our work can be adapted for distributed settings. In Chapter 3 we showed how Flare can be integrated with machine learning frameworks efficiently, and provide an efficient end-to-end pipeline. In Chapter 4 we tackled the problem of long compilation times for short execution time programs in the setting of code generation. Furthermore, we showed how our solution can be more broadly applied to other interesting applications that would benefit from on-stack-replacement features. We illustrated the feasibility of speculative optimizations. In Chapter 5 we introduced a new type of static analysis that extracts functional high-level representation from low-level imperative code. We presented a formal model with simplification rules and speculative rewriting that are semantics preserving. We also showed how our theory can be applied to C code for program verification and legacy code analysis.

REFERENCES

REFERENCES

[1] ONNX. https://github.com/onnx/onnx.

[2] Launching Ignition and TurboFan, 2017.

[3] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[4] Martn Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Man, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Vigas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous distributed systems, 2015.

[5] Maaz Bin Safeer Ahmad and Alvin Cheung. Leveraging parallel data processing frameworks with verified lifting. In *SYNT/CAV*, volume 229 of *EPTCS*, pages 67–83, 2016.

[6] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, et al. The Stratosphere platform for big data analytics. *The VLDB Journal*, 23(6):939–964, 2014.

[7] Bowen Alpern, Steve Augart, Stephen M. Blackburn, Maria A. Butrico, Anthony Cocchi, Perry Cheng, Julian Dolby, Stephen J. Fink, David Grove, Michael Hind, Kathryn S. McKinley, Mark F. Mergen, J. Eliot B. Moss, Ton Anh Ngo, Vivek Sarkar, and Martin Trapp. The Jikes research virtual machine project: Building an open-source research community. *IBM Systems Journal*, 44(2):399–418, 2005.

[8] Nada Amin and Tiark Rompf. LMS-Verify: abstraction without regret for verified systems programming. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 859–873, 2017.

[9] Nada Amin and Tiark Rompf. Type soundness proofs with definitional interpreters. In *POPL*, pages 666–679. ACM, 2017.

[10] Apache. Hadoop. `http://hadoop.apache.org/`.

[11] Andrew W. Appel. SSA is functional programming. *SIGPLAN Notices*, 33(4):17–20, 1998.

[12] 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 *SIGMOD*, pages 1383–1394. ACM, 2015.

[13] Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *OOPSLA*, pages 47–65, 2000.

[14] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim N Gray, Patricia P. Griffiths, W Frank King, Raymond A. Lorie, Paul R. McJones, James W. Mehl, et al. System R: relational approach to database management. *TODS*, 1(2):97–137, 1976.

[15] Charles W Bachman. The programmer as navigator. *Communications of the ACM*, 16(11):653–658, 1973.

[16] Olaf Bachmann, Paul S. Wang, and Eugene V. Zima. Chains of recurrences - a method to expedite the evaluation of closed-form functions. In *ISSAC*, pages 242–249. ACM, 1994.

[17] Riyadh Baghdadi, Ulysse Beaugnon, Albert Cohen, Tobias Grosser, Michael Kruse, Chandan Reddy, Sven Verdoolaege, Adam Betts, Alastair F. Donaldson, Jeroen Ketema, Javed Absar, Sven van Haastregt, Alexey Kravets, Anton Lokhmotov, Robert David, and Elnar Hajiyev. PENCIL: A platform-neutral compute intermediate language for accelerator programming. In *PACT*, pages 138–149. IEEE Computer Society, 2015.

[18] Michael Bebenita, Florian Brandner, Manuel Fähndrich, Francesco Logozzo, Wolfram Schulte, Nikolai Tillmann, and Herman Venter. SPUR: a trace-based JIT compiler for CIL. In *OOPSLA*, pages 708–725. ACM, 2010.

[19] Alexander Behm, Vinayak R Borkar, Michael J Carey, Raman Grover, Chen Li, Nicola Onose, Rares Vernica, Alin Deutsch, Yannis Papakonstantinou, and Vassilis J Tsotras. Asterix: towards a scalable, semistructured data platform for evolving-world models. *Distributed and Parallel Databases*, 29(3):185–216, 2011.

[20] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The polyhedral model is more widely applicable than you think. In *CC*, volume 6011 of *Lecture Notes in Computer Science*, pages 283–303. Springer, 2010.

[21] Jan A. Bergstra, T. B. Dinesh, John Field, and Jan Heering. A complete transformational toolkit for compilers. In *ESOP*, volume 1058 of *Lecture Notes in Computer Science*, pages 92–107. Springer, 1996.

[22] Dirk Beyer. Competition on software verification - (SV-COMP). In *TACAS*, volume 7214 of *Lecture Notes in Computer Science*, pages 504–524. Springer, 2012.

[23] Dirk Beyer and M. Erkan Keremoglu. CPAchecker: A tool for configurable software verification. In *CAV*, volume 6806 of *Lecture Notes in Computer Science*, pages 184–190. Springer, 2011.

[24] Abhilash Bhandari and V. Krishna Nandivada. Loop tiling in the presence of exceptions. In *ECOOP*, volume 37 of *LIPIcs*, pages 124–148. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[25] Jesse D. Bingham and Zvonimir Rakamaric. A logic and decision procedure for predicate abstraction of heap-manipulating programs. In *VMCAI*, volume 3855 of *Lecture Notes in Computer Science*, pages 207–221. Springer, 2006.

[26] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 18–25. ACM, 2009.

[27] Peter Boncz, Thomas Neumann, and Orri Erling. TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 61–76. Springer, 2013.

[28] Peter A Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-pipelining query execution. In *CIDR*, 2005.

[29] Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a production environment. In *OOPSLA*, pages 215–230, 1993.

[30] Stephen Brookes and Peter W. O'Hearn. Concurrent separation logic. *SIGLOG News*, 3(3):47–65, 2016.

[31] Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Arvind K. Sujeeth, Christopher De Sa, Christopher R. Aberger, and Kunle Olukotun. Have abstraction and eat performance, too: optimized heterogeneous computing with parallel patterns. In *CGO*, pages 194–205. ACM, 2016.

[32] Kevin J. Brown, Arvind K. Sujeeth, Hyouk Joong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. A heterogeneous parallel framework for domain-specific languages. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 89–100, Washington, DC, USA, 2011. IEEE Computer Society.

[33] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D Ernst. Haloop: efficient iterative data processing on large clusters. *PVLDB*, 3(1-2):285–296, 2010.

[34] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In *NFM*, volume 9058 of *Lecture Notes in Computer Science*, pages 3–11. Springer, 2015.

[35] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6):26:1–26:66, 2011.

[36] Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. Implementing multi-stage languages using ASTs, gensym, and reflection. In *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering*, GPCE '03, pages 57–76, Berlin, Heidelberg, 2003. Springer-Verlag.

[37] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.

[38] Manuel M. T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating Haskell array codes with multicore GPUs. In *DAMP*, pages 3–14. ACM, 2011.

[39] Edgar F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[40] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Carsten Binnig, Ugur Çetintemel, and Stan Zdonik. An architecture for compiling UDF-centric workflows. *PVLDB*, 8(12):1466–1477, 2015.

[41] Leonardo Dagum and Ramesh Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, January 1998.

[42] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150, 2004.

[43] Daniele Cono D'Elia and Camil Demetrescu. Flexible on-stack replacement in LLVM. In *CGO*, pages 250–260. ACM, 2016.

[44] Daniele Cono D'Elia and Camil Demetrescu. On-stack replacement, distilled. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 166–180, New York, NY, USA, 2018. ACM.

[45] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *POPL*, pages 297–302. ACM Press, 1984.

[46] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL server's memory-optimized OLTP engine. In *SIGMOD*, pages 1243–1254. ACM, 2013.

[47] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1976.

[48] Isil Dillig, Thomas Dillig, and Alex Aiken. Cuts from proofs: a complete and practical technique for solving linear inequalities over integers. *Formal Methods in System Design*, 39(3):246–260, 2011.

[49] Isil Dillig, Thomas Dillig, and Alex Aiken. Precise reasoning for programs using containers. In *POPL*, pages 187–200. ACM, 2011.

[50] Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In *TACAS*, volume 3920 of *Lecture Notes in Computer Science*, pages 287–302. Springer, 2006.

[51] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative MapReduce. In *ACM HPCD*, pages 810–818. ACM, 2010.

[52] Robert A. Van Engelen, Johnnie Birch, Yixin Shou, Burt Walsh, and Kyle A. Gallivan. A unified framework for nonlinear dependence testing and symbolic analysis. In *ICS*, pages 106–115. ACM, 2004.

[53] Grégory Essertel, Ruby Y. Tahboub, Fei Wang, James Decker, and Tiark Rompf. Flare & Lantern: Efficiently swapping horses midstream. *Proc. VLDB Endow.*, 12(12):1910–1913, August 2019.

[54] Grégory M. Essertel, Ruby Y. Tahboub, James M. Decker, Kevin J. Brown, Kunle Olukotun, and Tiark Rompf. Flare: Optimizing Apache Spark with native compilation for scale-up architectures and medium-size data. In *OSDI*, pages 799–815. USENIX Association, 2018.

[55] Grégory M. Essertel, Ruby Y. Tahboub, and Tiark Rompf. On-stack replacement for program generators and source-to-source compilers. Technical report, Department of Computer Science, Purdue University, 2019.

[56] Grégory M. Essertel, Guannan Wei, and Tiark Rompf. Precise reasoning with structured time, structured heaps, and collective operations. *Proc. ACM Program. Lang.*, 3(OOPSLA):157:1–157:30, October 2019.

[57] Torbjrn Granlund et al. GNU multiple precision arithmetic library 4.1.2, December 2002.

[58] Azadeh Farzan and Zachary Kincaid. Compositional recurrence analysis. In *FMCAD*, pages 57–64. IEEE, 2015.

[59] Stephen J. Fink and Feng Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *CGO*, pages 241–252. IEEE Computer Society, 2003.

[60] Olivier Flückiger, Gabriel Scherer, Ming-Ho Yee, Aviral Goel, Amal Ahmed, and Jan Vitek. Correctness of speculative optimizations with dynamic deoptimization. *PACMPL*, 2(POPL):49:1–49:28, 2018.

[61] Message P Forum. MPI: A message-passing interface standard. Technical report, Knoxville, TN, USA, 1994.

[62] Joseph Fourier. Extrait dune mémoire sur le refroidissement séculaire du globe terrestre. *Bulletin des Sciences par la Société Philomathique de Paris, April 1820*, pages 58–70, 1820.

[63] Yoshihiko Futamura. Partial evaluation of computation process — an approach to a compiler-compiler. *Transactions of the Institute of Electronics and Communication Engineers of Japan*, 54-C(8):721728, 1971.

[64] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S Yu, and Myungcheol Doo. SPADE: the System S declarative stream processing engine. In *SIGMOD*, pages 1123–1134. ACM, 2008.

[65] Google. The V8 JavaScript VM, 2009.

[66] Google. A new Crankshaft for V8, 2010.

[67] Denis Gopan, Thomas W. Reps, and Shmuel Sagiv. A framework for numeric analysis of array operations. In *POPL*, pages 338–350. ACM, 2005.

[68] Goetz Graefe. Volcano-an extensible and parallel query evaluation system. *TKDE*, 6(1):120–135, 1994.

[69] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The Seahorn verification framework. In *CAV (1)*, volume 9206 of *Lecture Notes in Computer Science*, pages 343–361. Springer, 2015.

[70] Peter Henderson and James H. Morris Jr. A lazy evaluator. In *POPL*, pages 95–103. ACM Press, 1976.

[71] Thomas A. Henzinger, Thibaud Hottelier, Laura Kovács, and Andrey Rybalchenko. Aligators for arrays (tool paper). In *LPAR (Yogyakarta)*, volume 6397 of *Lecture Notes in Computer Science*, pages 348–356. Springer, 2010.

[72] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *ECOOP*, pages 21–38, 1991.

[73] Urs Hölzle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization. In *PLDI*, pages 32–43, 1992.

[74] Urs Hölzle and David Ungar. Reconciling responsiveness with performance in pure object-orieted languages. *ACM Trans. Program. Lang. Syst.*, 18(4):355–400, 1996.

[75] Urs Hölzle and David M. Ungar. A third-generation SELF implementation: Reconsiling responsiveness with performance. In *OOPSLA*, pages 229–243. ACM, 1994.

[76] David Van Horn and Matthew Might. Abstracting abstract machines. In *ICFP*, pages 51–62. ACM, 2010.

[77] Kenneth E. Iverson. Notation as a tool of thought. *Commun. ACM*, 23(8):444–465, 1980.

[78] Bertrand Jeannet, Peter Schrammel, and Sriram Sankaranarayanan. Abstract acceleration of general linear loops. In *POPL*, pages 529–540. ACM, 2014.

[79] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22Nd ACM International Conference on Multimedia*, MM '14, pages 675–678, New York, NY, USA, 2014. ACM.

[80] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial evaluation and automatic program generation*. Peter Sestoft, 1993.

[81] Neil D Jones. An introduction to partial evaluation. *ACM Computing Surveys (CSUR)*, 28(3):480–503, 1996.

[82] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. Mix: A self-applicable partial evaluator for experiments in compiler generation. *LISP and Symbolic Computation*, 2:9–50, 1989.

[83] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. Verified lifting of stencil computations. In *PLDI*, pages 711–726. ACM, 2016.

[84] Zachary Kincaid, Jason Breck, Ashkan Forouhi Boroujeni, and Thomas Reps. Compositional recurrence analysis revisited. In *PLDI*, 2017.

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

[86] Kathleen Knobe and Vivek Sarkar. Array SSA form and its use in parallelization. In *POPL*, pages 107–120. ACM, 1998.

[87] André Kohn, Viktor Leis, and Thomas Neumann. Adaptive execution of compiled queries. In *ICDE*, 2018.

[88] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, Ishaan Joshi, Lenni Kuff, Dileep Kumar, Alex Leblang, Nong Li, Ippokratis Pandis, Henry Robinson, David Rorke, Silvius Rus, John Russell, Dimitris Tsirogiannis, Skye Wanderman-Milne, and Michael and Yoder. Impala: A modern, open-source SQL engine for Hadoop. In *CIDR*, 2015.

[89] Grzegorz Kossakowski, Nada Amin, Tiark Rompf, and Martin Odersky. JavaScript as an embedded DSL. In *ECOOP*, pages 409–434, 2012.

[90] Laura Kovács. Reasoning algebraically about P-solvable loops. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 249–264. Springer, 2008.

[91] Konstantinos Krikellas, Stratis D Viglas, and Marcelo Cintra. Generating code for holistic query evaluation. In *ICDE*, pages 613–624. IEEE, 2010.

[92] Nurudeen Lameed and Laurie J. Hendren. A modular approach to on-stack replacement in LLVM. In *VEE*, pages 143–154. ACM, 2013.

[93] Jacek Laskowski. Mastering Apache Spark 2. https://www.gitbook.com/book/jaceklaskowski/mastering-apache-spark/details, 2016.

[94] HyoukJoong Lee, Kevin J. Brown, Arvind K. Sujeeth, Hassan Chafi, Tiark Rompf, Martin Odersky, and Kunle Olukotun. Implementing domain-specific languages for heterogeneous parallel computing. *IEEE Micro*, 31(5):42–53, 2011.

[95] Sorin Lerner, David Grove, and Craig Chambers. Composing dataflow analyses and transformations. In *POPL*, pages 270–282. ACM, 2002.

[96] Sorin Lerner, Todd D. Millstein, and Craig Chambers. Automatically proving the correctness of compiler optimizations. In *PLDI*, pages 220–231, 2003.

[97] Ivan Llopard, Christian Fabre, and Albert Cohen. From a formalized parallel action language to its efficient code generation. *ACM Trans. Embedded Comput. Syst.*, 16(2):37:1–37:28, 2017.

[98] Ilias Marinos, Robert N.M. Watson, and Mark Handley. Network stack specialization for performance. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 175–186, New York, NY, USA, 2014. ACM.

[99] Trevor L. McDonell, Manuel M. T. Chakravarty, Vinod Grover, and Ryan R. Newton. Type-safe runtime code generation: accelerate to LLVM. In *Haskell*, pages 201–212. ACM, 2015.

[100] Frank McSherry. Scalability! but at what COST?, 2015.

[101] Frank McSherry, Michael Isard, and Derek Gordon Murray. Scalability! but at what COST? In *HotOS*, 2015.

[102] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment*, 3(1-2):330–339, 2010.

[103] Charith Mendis, Jeffrey Bosboom, Kevin Wu, Shoaib Kamil, Jonathan Ragan-Kelley, Sylvain Paris, Qin Zhao, and Saman P. Amarasinghe. Helium: lifting high-performance stencil kernels from stripped x86 binaries to Halide DSL code. In *PLDI*, pages 391–402. ACM, 2015.

[104] Guido Moerkotte and Thomas Neumann. Accelerating queries with group-by and join by groupjoin. *Proceedings of the VLDB Endowment*, 4(11), 2011.

[105] Derek Gordon Murray, Michael Isard, and Yuan Yu. Steno: automatic optimization of declarative queries. In *ACM SIGPLAN Notices*, volume 46, pages 121–131, 2011.

[106] Fabian Nagel, Gavin Bierman, and Stratis D Viglas. Code generation for efficient query processing in managed runtimes. *VLDB*, 7(12):1095–1106, 2014.

[107] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

[108] Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. Design and implementation of sparse global analyses for C-like languages. In *PLDI*, pages 229–238. ACM, 2012.

[109] Oracle. OpenJDK: Graal project, 2012.

[110] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making sense of performance in data analytics frameworks. In *NSDI*, pages 293–307, 2015.

[111] Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. Functional big-step semantics. In *ESOP*, volume 9632 of *Lecture Notes in Computer Science*, pages 589–615. Springer, 2016.

[112] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. In *Proceedings of the 7th International World Wide Web Conference*, pages 161–172, Brisbane, Australia, 1998.

[113] Michael Paleczny, Christopher A. Vick, and Cliff Click. The Java HotSpot server compiler. In *Java Virtual Machine Research and Technology Symposium*, 2001.

[114] Shoumik Palkar, James J Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, Matei Zaharia, and Stanford InfoLab. Weld: A common runtime for high performance data analytics. In *CIDR*, 2017.

[115] Holger Pirk, Oscar Moll, Matei Zaharia, and Sam Madden. Voodoo - a vector algebra for portable database performance on modern hardware. *VLDB*, 9(14):1707–1718, 2016.

[116] Massimiliano Poletto, Wilson C Hsieh, Dawson R Engler, and M Frans Kaashoek. C and TCC: a language and compiler for dynamic code generation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(2):324–369, 1999.

[117] William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *SC*, pages 4–13. ACM, 1991.

[118] M. Puschel, J.M.F. Moura, J.R. Johnson, D. Padua, M.M. Veloso, B.W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R.W. Johnson, and N. Rizzolo. Spiral: Code generation for dsp transforms. *Proceedings of the IEEE*, 93(2):232 –275, feb. 2005.

[119] Cosmin Radoi, Stephen J. Fink, Rodric M. Rabbah, and Manu Sridharan. Translating imperative code to MapReduce. In *OOPSLA*, pages 909–927. ACM, 2014.

[120] 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. ACM, 2013.

[121] Karthik Ramachandra, Kwanghyun Park, K Venkatesh Emani, Alan Halverson, César Galindo-Legaria, and Conor Cunningham. Froid: Optimization of imperative programs in a relational database. *Proceedings of the VLDB Endowment*, 11(4), 2017.

[122] Jags Ramnarayan, Barzan Mozafari, Sumedh Wale, Sudhir Menon, Neeraj Kumar, Hemant Bhanawat, Soubhik Chakraborty, Yogesh Mahajan, Rishitesh Mishra, and Kishor Bachhav. Snappydata: A hybrid transactional analytical store built on Spark. In *SIGMOD*, pages 2153–2156, 2016.

[123] Lyle Ramshaw. Eliminating goto's while preserving program structure. *J. ACM*, 35(4):893–920, 1988.

[124] Veselin Raychev, Madanlal Musuvathi, and Todd Mytkowicz. Parallelizing user-defined aggregations using symbolic execution. In *SOSP*, pages 153–167. ACM, 2015.

[125] Thomas W. Reps, Emma Turetsky, and Prathmesh Prabhu. Newtonian program analysis via tensor product. In *POPL*, pages 663–677. ACM, 2016.

[126] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.

[127] Armin Rigo and Samuele Pedroni. PyPy's approach to virtual machine construction. In *OOPSLA Companion*, pages 944–953, 2006.

[128] Tiark Rompf and Nada Amin. Functional Pearl: A SQL to C compiler in 500 lines of code. In *ICFP*, 2015.

[129] Tiark Rompf and Kevin J. Brown. Functional parallels of sequential imperatives (short paper). In *PEPM*, pages 83–88. ACM, 2017.

[130] Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. *Commun. ACM*, 55(6):121–130, 2012.

[131] Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin Brown, Vojin Jovanovic, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. Optimizing data structures in high-level programs. POPL, 2013.

[132] Tiark Rompf, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Hassan Chafi, and Kunle Olukotun. Surgical precision JIT compilers. In *PLDI*, pages 41–52. ACM, 2014.

[133] Tiark Rompf, Arvind K. Sujeeth, HyoukJoong Lee, Kevin J. Brown, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Building-blocks for performance oriented DSLs. In *DSL*, volume 66 of *EPTCS*, pages 93–117, 2011.

[134] Antony Rowstron, Dushyanth Narayanan, Austin Donnelly, Greg O'Shea, and Andrew Douglas. Nobody ever got fired for using Hadoop on a cluster. In *Proceedings of the 1st International Workshop on Hot Topics in Cloud Data Processing*, HotCDP '12, pages 2:1–2:5, New York, NY, USA, 2012. ACM.

[135] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.

[136] Jack Schwartz. Set theory as a language for program specification and programming. *Courant Institute of Mathematical Sciences, New York University*, 12:193–208, 1970.

[137] Amir Shaikhha, Ioannis Klonatos, Lionel Emile Vincent Parreaux, Lewis Brown, Mohammad Dashti Rahmat Abadi, and Christoph Koch. How to architect a query compiler. In *SIGMOD*, number EPFL-CONF-218087, 2016.

[138] Jeremy G. Siek. Denotational semantics of IMP without the least fixed point. http://siek.blogspot.ch/2016/12/denotational-semantics-of-imp-without.html, 2016.

[139] Jeremy G. Siek. Declarative semantics for functional languages: compositional, extensional, and elementary. *CoRR*, abs/1707.03762, 2017.

[140] Calvin Smith and Aws Albarghouthi. MapReduce program synthesis. In *PLDI*, pages 326–340. ACM, 2016.

[141] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance OpenCL code. In *ICFP*, pages 205–217. ACM, 2015.

[142] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. Lift: a functional data-parallel IR for high-performance GPU code generation. In *CGO*, pages 74–85. ACM, 2017.

[143] A. K. Sujeeth, H. Lee, K. J. Brown, T. Rompf, Michael Wu, A. R. Atreya, M. Odersky, and K. Olukotun. OptiML: an implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on Machine Learning*, ICML, 2011.

[144] Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Trans. Embedded Comput. Syst.*, 13(4s):134:1–134:25, 2014.

[145] Bo Joel Svensson, Mary Sheeran, and Ryan R. Newton. Design exploration through code-generating DSLs. *Commun. ACM*, 57(6):56–63, 2014.

[146] Bo Joel Svensson, Michael Vollmer, Eric Holk, Trevor L. McDonell, and Ryan R. Newton. Converting data-parallelism to task-parallelism by rewrites: purely functional programs across multiple GPUs. In *FHPC/ICFP*, pages 12–22. ACM, 2015.

[147] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. *SIGPLAN Not.*, 32(12):203–217, December 1997.

[148] Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000.

[149] Ruby Y. Tahboub, Grégory M. Essertel, and Tiark Rompf. How to architect a query compiler, revisited. SIGMOD '18, pages 307–322. ACM, 2018.

[150] Tian Tan, Yue Li, and Jingling Xue. Efficient and precise points-to analysis: modeling the heap by merging equivalent automata. In *PLDI*, pages 278–291. ACM, 2017.

[151] Robert Endre Tarjan. Fast algorithms for solving path problems. *J. ACM*, 28(3):594–614, 1981.

[152] Ross Tate, Michael Stepp, and Sorin Lerner. Generating compiler optimizations from proofs. In *POPL*, pages 389–402, 2010.

[153] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: A new approach to optimization. *Logical Methods in Computer Science*, 7(1), 2011.

[154] The XLA Team. XLA – TensorFlow Compiled. Post on the Google Developers Blog, 2017. http://developers.googleblog.com/2017/03/xla-tensorflow-compiled.html.

[155] The Transaction Processing Council. TPC-H Version 2.15.0, 2018.

[156] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *PVLDB*, 2(2):1626–1629, 2009.

[157] Deepak Vohra. Apache spark. In *Practical Hadoop Ecosystem*. Apress, Berkley, CA, 2016.

[158] Michael Vollmer, Bo Joel Svensson, Eric Holk, and Ryan R. Newton. Meta-programming and auto-tuning in the search for high performance GPU code. In *FHPC/ICFP*, pages 1–11. ACM, 2015.

[159] Fei Wang, Daniel Zheng, James Decker, Xilun Wu, Grégory M. Essertel, and Tiark Rompf. Demystifying differentiable programming: Shift/reset the penultimate backpropagator. *Proc. ACM Program. Lang.*, 3(ICFP):96:1–96:31, July 2019.

[160] Kunshan Wang, Stephen M. Blackburn, Antony L. Hosking, and Michael Norrish. Hop, skip, & jump: Practical on-stack replacement for a cross-platform language-neutral VM. In *VEE*, pages 1–16. ACM, 2018.

[161] Edwin Westbrook, Mathias Ricken, Jun Inoue, Yilong Yao, Tamer Abdelatif, and Walid Taha. Mint: Java multi-stage programming using weak separability. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 400–411, New York, NY, USA, 2010. ACM.

[162] Christian Wimmer, Michael Haupt, Michael L. Van de Vanter, Mick J. Jordan, Laurent Daynès, and Doug Simon. Maxine: An approachable virtual machine for, and in, Java. *TACO*, 9(4):30, 2013.

[163] Christian Wimmer and Thomas Würthinger. Truffle: a self-optimizing runtime system. In *SPLASH*, pages 13–14, 2012.

[164] Bin Xin, William N. Sumner, and Xiangyu Zhang. Efficient program execution indexing. In *PLDI*, pages 238–248. ACM, 2008.

[165] Reynold S Xin, Josh Rosen, Matei Zaharia, Michael J Franklin, Scott Shenker, and Ion Stoica. Shark: SQL and rich analytics at scale. In *ACM SIGMOD*, pages 13–24, 2013.

[166] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations. In *NDSS*. The Internet Society, 2015.

[167] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. *OSDI*, 8, 2008.

[168] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[169] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache Spark: a unified engine for big data processing. *Commun. ACM*, 59(11):56–65, 2016.

[170] He Zhu, Stephen Magill, and Suresh Jagannathan. A data-driven CHC solver. In *PLDI*, pages 707–721. ACM, 2018.