Delft University of Technology

Masters Thesis

An Intermediate Representation for Stateful Dataflows

Author:
Lucas Van Mol

Thesis Advisor:
Dr. Asterios Katsifodimos

Daily Supervisors:
Marcus Schutte MSc
Dr. George Christodoulou

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

in the

Web Information Systems Group
Software Technology

June 23, 2025

Abstract

Building scalable and consistent cloud applications is notoriously difficult due to the challenges of state management and execution consistency in distributed environments. Functions-as-a-Service (FaaS) platforms offer flexible scalability, but weak execution guarantees forces engineers to mix business logic with infrastructure concerns, adding error-handling code, retry mechanisms and consistency checks throughout their applications. At the same time, dataflow systems like Apache Flink offer exactly-once semantics, but their functional APIs often conflict with the imperative, object-oriented style preferred by mainstream developers.

This work aims to address this disconnect, arguing that modern transactional applications, from e-commerce to payment systems to business workflows, naturally form stateful dataflow graphs. By allowing developers to write familiar imperative code that executes on dataflow systems with strong consistency guarantees, we could eliminate the need to handle many infrastructure concerns explicitly.

To this end, we introduce *Cascade*, a compiler pipeline and intermediate representation that bridges the gap by translating imperative Python code into stateful, parallelizable dataflow graphs. Cascade extends prior work by providing a representation that is both expressive and optimizable, and we demonstrate optimizations including parallel execution via data dependency analysis and dynamic value prefetching. Our results show significant performance gains with these optimizations, all while maintaining the strong execution guarantees of the underlying execution target. Finally, we offer avenues for future research by discussing further optimization possibilities and extensions within our proposed framework.

Acknowledgements

Firstly, I would like to thank Asterios, for taking the time to answer a random email about my incoherent thoughts after I followed his excellent course on web-scale data management. His boundless enthusiasm about the subject led to this (hopefully more coherent) thesis. I would also like to thank my supervisors, Marcus and George. Thank you, Marcus, for our fruitful one-on-one conversations, and always being supportive and encouraging. Thank you, George, for your invaluable comments and ideas during thesis meetings. Thanks as well to Kyriakos Psarakis, who helped me understand some of the more technical aspects of experimental evaluation in the subject, and to Soham Chakraborty, for being part of my thesis committee and taking the time to evaluate this work.

Finally, I'm especially grateful to Tilly for her proofreading and helpful feedback, but most of all her unwavering encouragement, even while separated by the tragic reality of differing time zones and unnecessary border controls.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Building scalable and consistent applications in cloud environments remains a significant challenge for developers. While cloud technologies offer abundant computational resources, effectively utilizing these capabilities without compromising reliability introduces complexity that often overwhelms the actual business logic of applications [3]. Function-as-a-Service (FaaS) platforms provide scalable flexibility, but they shift the burden of managing consistency, state, and failure handling to developers. These problems are exacerbated by cloud providers offering proprietary APIs that lack standardization, creating vendor lock-in concerns.

A typical cloud application combines FaaS with external object stores or Backends-as-a-Service (BaaS) [4], but this arrangement lacks the strong execution guarantees needed for many business applications. Existing solutions such as AWS Lambda [5] or Orleans [6] attempt to address these challenges but typically offer only at-most-once or at-least-once processing semantics. As a result, developers are often forced to handle many consistency issues manually. Meanwhile, most developers continue using familiar web application libraries like Java's Spring Boot, Python's Flask, or JavaScript's Next.js, which provide minimal support for distributed execution [7].

Addressing these issues are frameworks such as Cloudburst [8], Hilda [9] and Azure Durable Functions [10], which improve cloud programming abstractions and integrate state management whilst providing stronger guarantees. At the same time, modern batch and streaming dataflow systems such as Apache Flink [11], Apache Spark [12], Naiad [13], and Styx [14] often provide *exactly-once processing* guarantees built-in. This allows developers to avoid dealing with many of the challenges of distributed systems, such as message duplication and partial failures, letting them concentrate on their application's core logic instead. However, these systems typically rely on functional programming APIs that can feel awkward and unnatural to many developers accustomed to traditional imperative coding styles.

At the core of this problem is a straightforward mismatch: developers naturally write imperative, object-oriented code, but the systems that provide strong consistency guarantees are designed for functional programming models. These dataflow systems can deliver exactly-once processing guarantees that would eliminate the need for extensive defensive programming, yet they remain inaccessible to mainstream development practices.

Previous research in dataflow programming and architecture ([15], [16]) has identified effective foundations for parallel execution of dataflows, but translating from imperative code to the dataflow model remained largely unexplored since initial work in the 1990s [17], [18], [19]. However, with the rise of data-parallel processing frameworks and distributed stream processing systems, there has been renewed interest in research addressing this translation gap in recent years [20], [21], [22].

This work addresses the disconnect between how developers write code and how distributed systems execute it. We present Cascade, a compiler pipeline and intermediate representation (IR) that converts imperative, object-oriented code into parallelized dataflow graphs. User code is executed on stream processing systems, allowing for low-latency, exactly-once data processing. Through data dependency analysis, our approach transforms sequential code into parallel execution paths that modern streaming engines can process efficiently. We extend the ideas introduced by StateFlow [22], using Apache Flink [11] as an execution target and demonstrate performance improvements while maintaining strong consistency guarantees.

1.1 Summary of contributions

A summary of our contributions is as follows:

- We extend StateFlow [22] with an updated intermediate representation suitable for further optimization, and an updated approach to compilation of imperative programs into distributed, stateful streaming dataflows.
- We optimize configuration for Apache Flink as an execution target for low-latency applications, benchmarking it on the DeathStar *Hotel Reservation* benchmark [23].
- We describe an algorithm for increasing the parallelism of existing dataflow graphs and evaluate the result on the DeathStar *Movie Review* benchmark [23].
- We describe a more dynamic optimization inspired by traditional compiler optimizations by prefetching certain values, and evaluate this scheme on a novel `NavigationService` benchmark.
- We outline several optimization possibilities for future research, including optimizations related to state size management and handling stateless program sections efficiently.

1.2 Thesis outline

The rest of this thesis is organized as follows. We first provide preliminary background knowledge in Chapter 2. Chapter 3 provides a description for Cascade's intermeditae representation (IR). Compilation into this representation is described in Chapter 4. Execution of this IR on Apache Flink is described in Chapter 5, along with initial experiments to benchmark its performance. We describe and evaluate two optimizations in Chapter 6, including parallelization in Section 6.1 and dynamic prefetching in Section 6.2. A discussion and comparison to related work is described in Chapter 7, and a conclusion with avenues for future work is presented in Chapter 8. Source code for this thesis can be found at https://github.com/delftdata/cascade.

Chapter 2

Preliminaries

This chapter summarizes background knowledge that may be necessary for understanding the concepts and terminology used in the rest of this work. We include background on data streaming systems, namely stream processing in Section 2.1 and event streaming in Section 2.2. Section 2.3 covers programming languages, in particular their grammar and syntactical structure used in compilation.

2.1 Stream processing

Stream processing systems such as Apache Flink [11] are designed to process and analyze large volumes of data in motion, typically in real-time or near real-time. These systems are typically distributed and designed for high-throughput and low-latency workloads [24]. In a dataflow model, computations are modeled as directed graphs, where operators (nodes) consume and produce data via first-in-first-out channels (edges). This model naturally supports parallelism and event-driven execution in low-latency contexts. Flink's core abstraction, the *DataStream API*, enables developers to define event-driven applications using a functional paradigm. As opposed to more traditional, table-based query systems, queries are modelled as static building blocks that ingest and process input data.

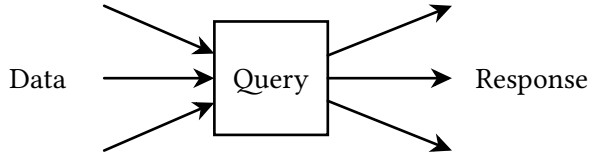


Figure 1: Data ingestion and queries in stream processing systems.

Flink supports *stateful stream processing*, where operators can maintain state across partitions and recover from failures using distributed snapshots. Flink periodically generates *checkpoints*, a consistent snapshot of the current state of an application including operator state, stream offsets and watermark positions. This allows for *exactly-once semantics*, meaning each incoming event affects the final result exactly once, even in the case of machine or software failure.

Flink provides high-level transformations such as `map`, `filter`, `reduce`, `window` and `union`. Data can also be transformed by *operators* called `ProcessFunctions`, that provide fine-grained control over event processing, state management and side outputs. Streams can be partitioned with per-key state using a `KeyedProcessFunction`. It should be noted that this operation may require a *shuffle*: a redistribution of records so that all records with the same key are sent to the same partition. Because shuffling is resource-intensive, it remains a key focus point when optimizing stream processing systems [25].

Flink uses a number of optimizations to achieve high performance in stream processing. Some key optimizations include pruning redundant operations from the dataflow graph, addressing data skew issues, predicate pushdown, partitioning, data redistribution, operator fusion, and many more. We consider these to be primarily **physical optimizations**, falling under the responsibilities of the execution system itself. In contrast, **logical optimizations**, such as the ones explored in Chapter 6, relate directly to the user's program logic. Most stream processing systems, like Flink, are limited in the logical optimizations they can perform due to their shallow program context [26].

2.2 Event streaming

Apache Kafka [27] is a distributed event streaming platform built for high-throughput and fault-tolerant handling of real-time, replayable data streams. One of the fundamental concepts in Kafka is the *topic*, which serves as a logical channel through which events are published and consumed. Within each topic, *partitioning* allows Kafka to distribute data across multiple brokers, enabling parallel processing and improving scalability. Kafka uses a decoupled model of *producers* and *consumers*, where producers publish data to topics and consumers read from them independently.

Kafka's architecture is centered around a distributed commit log, which provides strong ordering guarantees within each partition. This model supports horizontal scalability across a cluster of broker nodes. By replicating data across brokers, Kafka ensures fault tolerance and high availability while maintaining performance in demanding event-streaming environments. Flink and Kafka together provide end-to-end exactly-once semantics thanks to Kafka's support for transactions [28].

2.3 Programming languages

Python is a high-level, interpreted programming language. Example syntax is shown in Listing 1. In order to compile Python source code, Cascade must parse Python syntax by translating source code into its *abstract syntax tree*. Additionally, control flow primitives, such as `if`, `while` and `for` statements can be analyzed using a *control flow graph*.

```
1 def buy_item(user: User, item: Item) -> bool:
2   cost = item.price
3   if cost > user.balance:
4     return False
5   else:
6     user.balance -= cost
7     return True
```

Listing 1: Example of a Python function.

Python uses many object-oriented programming (OOP) principles, allowing for the use of classes that encapsulate state through attributes and functionality through methods. It also supports OOP principles such as abstraction, inheritance and polymorphism.

2.3.1 Abstract syntax trees

An abstract syntax tree (AST) is a tree-structured representation of code that captures its meaningful components while omitting syntactic details like parentheses or whitespace. It's used by Cascade's compiler to parse code into a structured form that's easier to analyze and transform than raw text.

As an example, even a simple assignment as in `cost = item.price` from Listing 1-2 contains multiple elements that need to represented. The assignment contains the left hand side *target*, `cost`, and the right hand side *value*, `item.price`. The right hand side is itself another expression: the attribute `price` on the value `item`. The assignment is represented in the Python AST as in Figure 2.

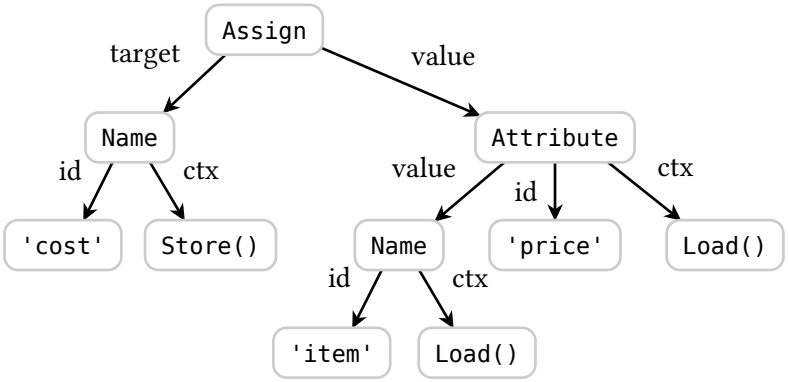


Figure 2: Python AST representation of `cost = item.price`.

An `if` statement would typically be represented as a node with three child nodes: a *predicate* node (the condition being evaluated), a *body* node (the code block to execute if the condition is true), and an optional *orelse* node. Each of these children contain their own subtrees representing more complex expressions or nested statements, allowing the entire structure of the program to be captured hierarchically. The `if` statement in Listing 1-3 creates the branching structure in Figure 3.

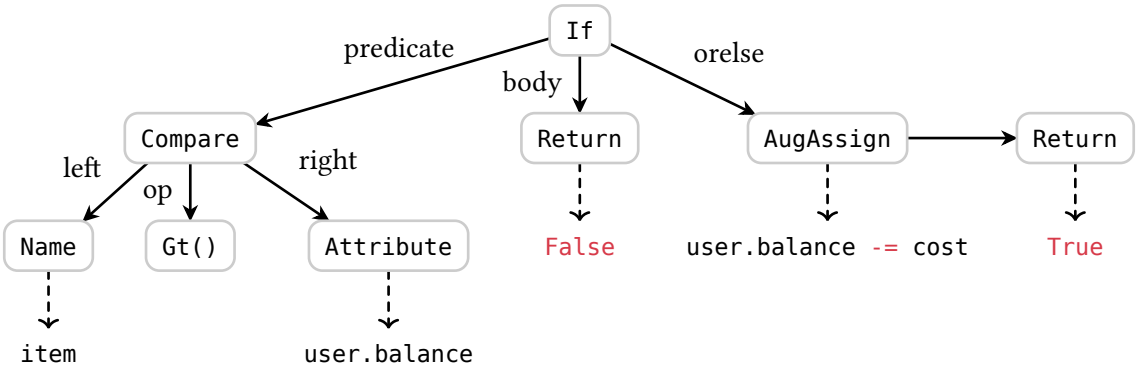


Figure 3: AST representation of the `if` statement in Listing 1-3.

2.3.2 Control flow graphs

A *control flow graph* (CFG) is a representation of all possible execution paths in a program. It consists of nodes representing *basic blocks* of code and edges representing possible flow of control between them, enabling analysis of program execution, optimization opportunities, and identifying unreachable code.

A *basic block* is a sequence of consecutive statements with exactly one entry point (the first instruction) and one exit point (the last instruction). No jumps or other control flow exist within a basic block, and execution always flows from start to finish without branching. Basic blocks form the nodes in a CFG, with edges connecting blocks that can execute one after another.

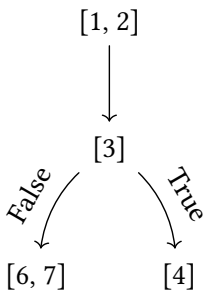


Figure 4: Control flow graph of Listing 1.

2.3.3 SSA & ANF

Static single assignment (SSA) form is an intermediate representation in which each variable is assigned exactly once. SSA simplifies data flow analysis by making dependencies between variables explicit, enabling more effective optimization techniques such as constant propagation, dead code elimination, and register allocation. When control flow merges, φ-functions are introduced to unify variable values coming from different branches, preserving correctness while maintaining the single-assignment property.

A-normal form (ANF) is another intermediate representation where all function arguments are *atomic*: either a variable or a constant. As a result, complex expressions are broken down into a sequence of let-bindings. This form is particularly useful in functional language compilers because it simplifies the control and evaluation order, making transformations like continuation-passing style (CPS) conversion or code generation more straightforward.

Cascade uses a combination of these concepts, using SSA to simplify data dependency analysis and ANF to make the evaluation order of remote calls more explicit for later optimization.

2.3.4 Data dependency graphs

Data dependency graphs represent the flow of data between statements in a program. The graph is composed of nodes (statements) and directed edges that indicate data dependencies, where the output of one operation serves as input to another. For example, consider the statement from Listing 1-3.

```
if cost > user.balance:
```

This statement is dependant on two other lines, notably:

```
cost = item.price
```
and
```
def buy_item(item: Item, user: User) -> bool:
```

These two statements must therefore be evaluated **before** the first. These dependence relations are represented in a graph as in Figure 5. Cascade uses dependency graphs in order to expose the true data dependencies between operations and uncovering opportunities for parallelization between independent statements.

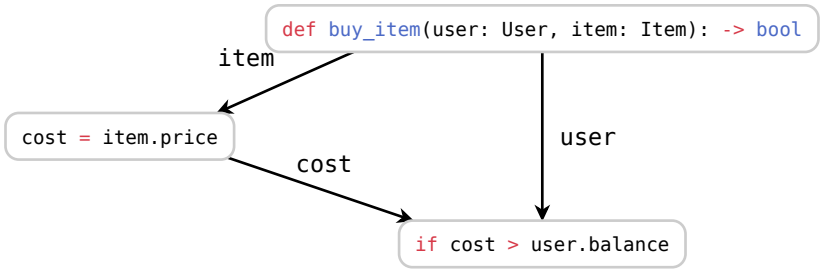


Figure 5: Dependency graph of Listing 1, lines 1-3.

2.3.5 Intermediate representations

In modern compilers, *intermediate representations* (IRs) serve as a bridge between high-level programming languages and machine code. This allows user-written source code to pass through various optimization passes, in addition to allowing an abstraction away from the specific execution system.

Perhaps the most widely used IR is that of LLVM [29], which has allowed unprecedented interoperability between programming languages (e.g. C, Rust, Haskell, Go) and execution targets (e.g. x86, ARM assembly, RISC-V). As a language-agnostic framework, LLVM's optimization passes have had a wide impact across countless programs.

Cascade has similar aims to LLVM in this regard, offering an intermediate representation that compiles from Python and is subsequently optimizable. An important distinction is that Cascade does not *compile* to execution targets; rather, the IR is *interpreted* by an execution target via a process described in Chapter 5.

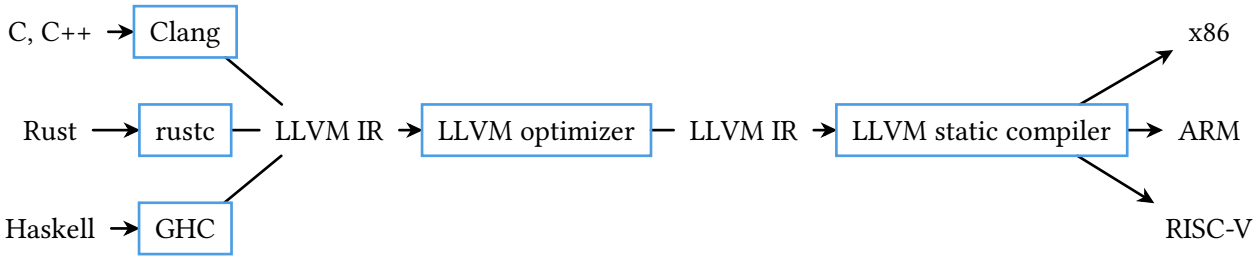


Figure 6: LLVM overview, adapted from [1].

Chapter 3

Cascade's Intermediate Representation

In this chapter, we introduce *Cascade*: a Python framework that can be used to compile imperative code into stateful dataflows. Using Cascade, developers can write and define classes as Python objects, which are then compiled, optimized and executed on a distributed runtime. Cascade's primary contribution consists of an intermediate representation (IR) that has two primary requirements. The first requirement is to be flexible enough to allow the programmer to write and execute any reasonable code within the Cascade framework. Any reasonable code the developer writes should be easily translatable into the Cascade IR. Secondly, we want this representation to be subsequently simple and expressive enough to be both easy to analyze and easy to optimize.

In a broad sense, Cascade's IR consists of *operators* (Section 3.1), written as user-defined Python classes, that may access state and can execute blocks of computation. Functions in Cascade are centered around a directed graph, the *dataflow*, which contains a set of nodes described in Section 3.2. Dataflows are written in code as class methods. *Events* (Section 3.3) propagate along these dataflows, being routed by the Cascade runtime, and utilizing operators to perform computations in order to progress.

We discuss compilation into this IR in Chapter 4 and discuss its execution in Chapter 5. These two components can be referred as to Cascade's frontend and backend respectively in classical compiler design nomenclature.

3.1 Operators

Operators are the building blocks of Cascade's IR. In Cascade, a user-defined class (UDC) can be written as an ordinary Python class. A class becomes an operator when annotated with `@cascade`. Some of these classes contain state in the form of class attributes: these correspond to stateful operators, and their class methods may read or write their own state. Other classes may only define static methods, using Python's `@staticmethod` decorator, and are stateless. An annotated class's methods are compiled into dataflows, and are part of the operator.

In summary, each operator (class) contains a collection of associated dataflows (methods). On stateful operators, these associated dataflows can access operator state. The translation of how UDCs are transformed into Cascade's IR is summarized in Table 1. A class diagram of operators is shown in Appendix A1.

Imperative Code	Cascade IR
Class	Operator
Class attributes	Operator state
Method declaration	Dataflow reference
Method body	Dataflow / `CompiledLocalBlocks`
Method call	Event
Method return	EventResult

Table 1: Translation of imperative code to the Cascade IR.

3.1.1 Stateless operators

Stateless operators represent a collection of one or more related, stateless functions. These functions are represented as dataflows that perform some computation, and which may also call other operators, stateful or not.

```
1 @cascade
2 class Calculator:
3     @staticmethod
4     def add(a: int, b: int) -> int:
5         return a + b
6
7     @staticmethod
8     def subtract(a: int, b: int) -> int:
9         return a - b
```

Listing 2: Stateless operator with two associated dataflows.

Cascade's IR provides no guarantees that these dataflows will be colocated in the underlying execution system. This helps with optimization, as the associated dataflows are not tied to a particular state backend. In fact, certain optimizations could even get rid of these dataflows entirely, opting instead to *inline* the function, as described in Section 8.1.1.

3.1.2 Stateful operators

Stateful operators allow for state retrieval and modification in their associated dataflows. A stateful operator's state is keyed such that each class instance can access its own state using its unique key. As such, class methods become dataflows that can read and write *operator state*. Operator state is represented as a map of class attributes (represented as strings) to their values. This aligns with how Python handles state within its classes, as the state map can be easily retrieved using the built-in `__dict__` attribute on an object. Note that we discern between two types of state. While stateful operators allow access to operator state, any operator can also access *function state*. Function state is saved in Events rather than in operators, and represents the current values of local variables in the dataflow.

In order to initialize operator state, stateful operators contain a special initialization function. This is a stateless function that initializes the state of a new class instance, provided some unique key. In Cascade, this is handled by Python's `__init__()` function. The first positional argument provided (excluding `self`) will correspond to the key of the instantiated class.

User Operator

```python
@cascade
class User:
    def __init__(self, user_id, balance):
        self.user_id = user_id
        self.balance = balance

    def buy_item(self, item: Item):
        self.balance -= item.get_price()
        return self.balance >= 0
```

Item Operator

```python
@cascade
class Item:
    def __init__(self, ...):
        self.item_id = item_id
        self.price = price

    def get_price():
        return self.price
```

Listing 3: Interaction between multiple (stateful) operators.

In the example shown in Listing 3, an instance of `User` might have the following associated state:

```
state = { "user_id": "ca00-fa93-22ef",
          "balance": 4332 }
```

This state is retrieved internally by some choice of key, in this example the `user_id` variable. As we will see in Section 4.1, the `self` identifier will be transformed into a state access via Python's dictionary implementation. In the case above, the call to `self.balance` might be replaced with the subscript `state["balance"]`, where the variable `state` is provided by the Cascade runtime, and is persisted by the underlying execution target.

3.2 Dataflows

A dataflow, in the most general sense, can be represented as a directed graph. These dataflows could also be cyclical, whether directly through control flow primitives, like `for` or `while` loops, or indirectly, through recursive function calls. In this first iteration of the Cascade IR, dataflows consist of five types of nodes.

CallLocal. These nodes execute local blocks. A local block consists of one or more statements, and do not contain control flow primitives or remote calls to other entities. These blocks can read or write function state. On stateful operators, they can also read or write operator state.

CallRemote. These nodes enable nested calling of other dataflows within a dataflow. They contain information such as what dataflow to call, which variables to use as arguments and which key to use (when calling to a stateful operator). The CallRemote node also defines what variable to assign the result to, if any.

IfNode. These nodes allow for branching in the dataflow. Execution will continue in one of two outgoing edges based on the true/false value of a specified boolean variable (the predicate) in function state. Predicates of more complex expressions are handled during compilation, by assigning their value to a temporary boolean variable before proceeding.

CollectNode. These nodes wait on the execution of each of their inputs, acting as a sort of barrier. Internally, they use operator state to save the function state of incoming events. When all expected

input events have arrived, the input event's states are combined and an outgoing event is yielded. These nodes are used only in parallel execution contexts, and are further explained in Section 6.1.

ReturnNode. This node is always a leaf node of the dataflow. Given a name of a variable in function state, it ensures that the dataflow returns the value of that variable when the dataflow is completed. As with the IfNode, return values of more complex expressions are first assigned to a temporary variable in a preceding CallLocal block.

Consider the `buy_item` example from Listing 3. It contains a call to `item.get_price()`, which is translated into a CallRemote node. A CallLocal node is then needed to update `self.balance`. This same node then calculates the return variable expression `self.balance >= 0`, which is returned by a ReturnNode. The resulting dataflow is shown in Figure 7. A class diagram of the different types of nodes is shown in Appendix A2.

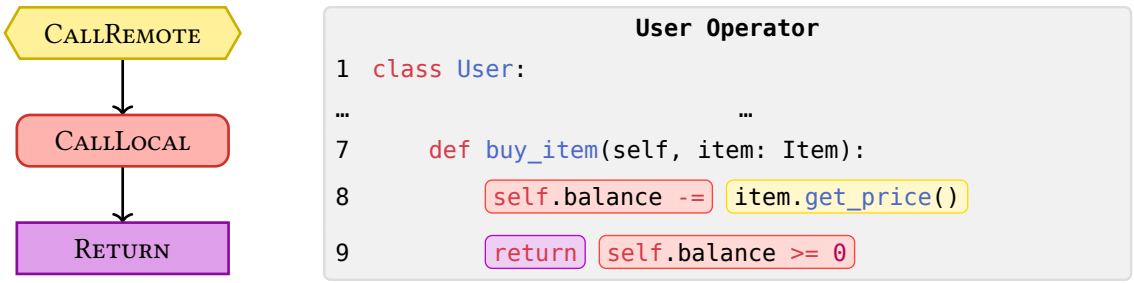


Figure 7: User's `buy_item` dataflow.

3.3 Events

Execution of dataflows is done by routing *events* through the dataflow graph. Events represent partial pieces of function execution and contain information such as:

- their associated dataflow,
- the *target node* they need to go to next,
- active local function variables and their values (*function state*),
- a call stack, used when calling other dataflows, and
- relevant state keys.

Event execution works by *propagating* it through the dataflow. A lot of this propagation, for example propagation related to control flow, is executed by Cascade. Only when a node reaches a CallLocal or CollectNode does execution context switch to the underlying backend to perform the necessary (distributed) computation.

Cascade allows nested dataflows through CallRemote nodes. When an event is created, or whenever it reaches a CallRemote node, information about the called dataflow is pushed onto the call stack. When an event reaches the end of a dataflow, the call stack is popped, and the event can resume execution on the original dataflow by investigating the top of the call stack. A class diagram of Events and how they relate to the call stack and dataflows is shown in Appendix A3.

Cascade	Execution Target
Handling CallRemote nodes	Executing function blocks with CallLocal
Pushing and popping the call stack	Operator state persistence
Navigating control flow, e.g. IfNode	Handling CollectNode with state
Handling ReturnNode	

Table 2: Separation of concerns between Cascade and the underlying runtime.

3.4 Event propagation

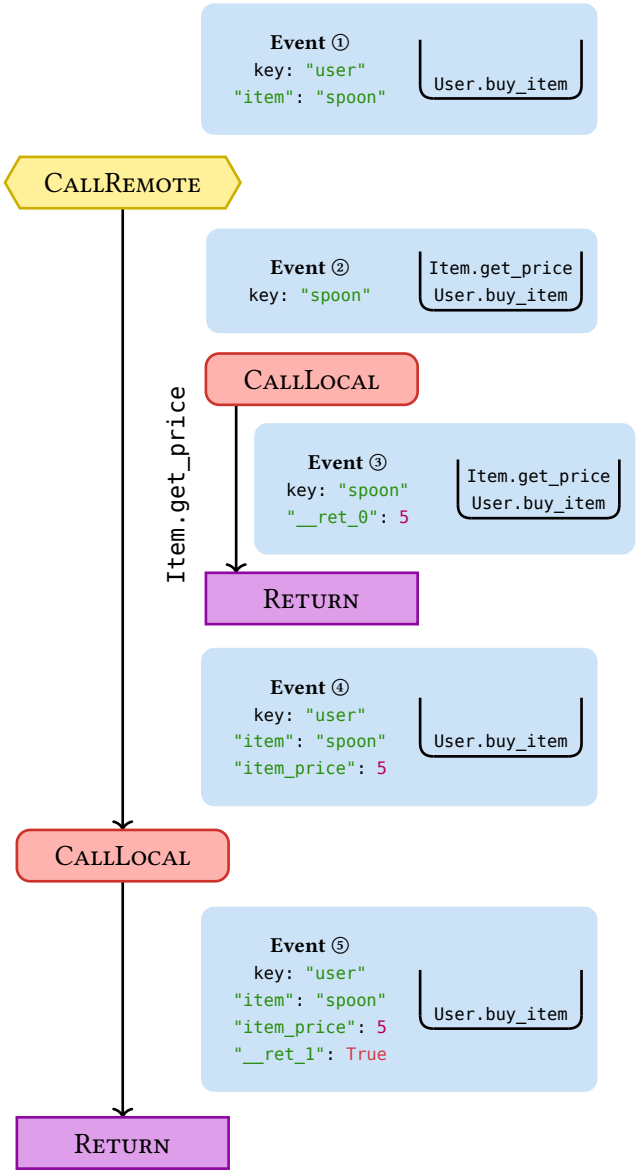


Figure 8: Event propagation for `User.buy_item`.

Figure 8 shows how event propagation works for the `User.buy_item` dataflow from Listing 3. The initial call to the dataflow creates an event containing the dataflow `User.buy_item` on the stack, along with the argument variables (`item`) and a state key `"user"` for the `User` operator ①.

The event starts out with a CallRemote target as per the dataflow graph. Cascade interprets this as a call to the `Item.get_price` dataflow. As such, the new dataflow is put on the stack, and the event is routed to the `Item` operator with a new key, taken from local function state ②.

The CallLocal node uses the backend to execute the `Item.get_price` function, which will retrieve the price from the operator state and save it in the function state with `{"__ret_0": 5}` ③.

The ReturnNode reads this value from function state and assigns it to the specified variable, in this case `"item_price"`. Since the call to `get_price` is over, Cascade pops the dataflow from the stack and continues execution on the `User.buy_item` dataflow with the returned value saved in function state ④.

The event target is now another CallLocal node, which handles the rest of the computation: decrementing the user's balance and setting the correct return value ⑤.

Finally, the event reaches another ReturnNode. Once more, Cascade pops the current dataflow (`User.buy_item`) from the stack. Since the stack is now empty, Cascade can return the final result contained in the variable `"__ret_1"`.

3.5 Limitations

The current iteration of the Cascade IR described in this chapter represents a minimal IR designed to test the optimizations in Chapter 6. As such, certain desirable features are currently missing. Support for `while` and `for` loops is missing, and there's currently no way to query operators with SQL-like `SELECT` queries. Further discussion on future extensions to the IR is provided in Section 8.1.

Chapter 4

Compilation

The Cascade compiler is responsible for transforming ordinary Python source code into a dataflow described by the IR in Chapter 3. This is achieved by a series of steps outlined in Figure 9. First, the source code is transformed into its abstract syntax tree (AST) with the `klara` library. Internally, this uses the same grammar as the `ast` library in the Python standard library, while including extra features such as single static assignment (SSA), used in the preprocessing step (Section 4.1). As well as SSA, this preprocessing step also transforms some of the code into A-Normal Form (ANF).

This AST can then be transformed into an ordered collection of statements that forms the control flow graph (CFG) (Section 4.2.1). We then group these individual statements together into a series of basic blocks (Section 4.2.2), and then split out and highlight the remote calls (Section 4.2.3). Finally, the CFG is transformed into Cascade's IR (Section 4.3) so that it can be optimized in later steps.

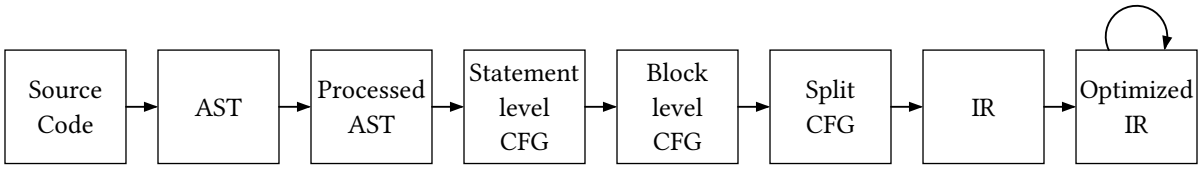


Figure 9: High level overview of the Cascade compiler.

This chapter uses a running example to illustrate the compilation process. We compile the `MovieId.upload_movie` dataflow below. This is part of the DeathStar movie review benchmark used to evaluate the parallelization algorithm in Section 6.1.

```
1  @cascade
2  class MovieId:
3    def __init__(self, title: str, movie_id: str):
4      self.title = title
5      self.movie_id = movie_id
6
7    def upload_movie(self, review: Review, rating: Optional[int]) -> bool:
8      success = True
9      if rating is not None:
10       success = review.upload_rating(rating)
11     movie_id = self.movie_id
12     return success and review.upload_movie_id(movie_id)
```

4.1 Preprocessing

Using the `@cascade` annotation, Python classes and their methods are first registered with an initial pass. This builds a collection of dataflow references, similar to a symbol table. Future passes can reference the table to distinguish remote calls from ordinary function calls.

When Cascade is initialized, the collected classes are each compiled, starting with the preprocessing step. This step transforms the standard Python AST into a form more suited to Cascade's IR, by following a series of sub-steps. These sub-steps including transforming the code into SSA, let-binding of complex expressions, and rewriting state accesses.

4.1.1 SSA

The first step is to transform the source code into single static assignment. This allows for certain subsequent optimizations (especially when doing data dependency analysis, such as during parallelization in Section 6.1) to be implemented in an easier way.

SSA assigns each variably exactly once, by providing a version number for shadowed variables. When multiple control flow paths merge, a *φ-function* determines which value a variable should have by selecting the appropriate value based on which control flow path was taken. For example, the following snippet:

```
1   success = True
2   if rating is not None:
3     success = review.upload_rating(rating)
4   ...
5   return success and ...
```

is transformed into:

```
1   success_0 = True
2   if rating_0 is not None:
3     success_1 = review_0.upload_rating(rating_0)
4   success_2 = phi(success_0, success_1)
5   ...
6   return success_2 and ...
```

4.1.2 Complex expression let-binding

Due to the structure of the IR, certain expressions can only be simple variables or constants, similar to ANF-based IRs. For example, an IfNode requires an identifier to be a boolean variable, and a ReturnNode requires an identifier to a simple local variable (see Section 3.2).

As such, this step transforms complex `if` predicates and `return` expressions into simple variables, changing

```
1   if rating is not None:
2     success = review.upload_rating(rating)
3   movie_id = self.movie_id
4   return success and review.upload_movie_id(movie_id)
```

into the following:

```
1   __cond_0 = rating is not None
2   if __cond_0:
3     success = review.upload_rating(rating)
4   movie_id = self.movie_id
```

```
5    __ret_0 = success and review.upload_movie_id(movie_id)
6    return __ret_0
```

In addition, this will also generate variable names for the results of remote calls. Suppose the following example where we have two instances of the `Item` operator.

```
1  def buy_items(self, item1: Item, item2: Item):
2    self.balance -= (item1.get_price() + item2.get_price())
```

This will be transformed into:

```
1  def buy_items(self, item1: Item, item2: Item):
2    __call_0 = item1.get_price()
3    __call_1 = item2.get_price()
4    self.balance -= (__call_0 + __call_1)
```

Which has the desired property that every remote call will be written on its own line, and makes the instruction order explicit.

4.1.3 State identification

As explained in Section 3.1.2, Cascade stores state as Python dictionary object. This preprocessing step transforms calls to `self` into dictionary accesses:

```
1  movie_id = self.movie_id
```

becomes

```
1  movie_id = state["movie_id"]
```

The `state` variable is handled by the Cascade execution engine, and encapsulates the state that's originally in `self.__dict__`. This process happens in the same way for writes.

The result of the preprocessing step, including SSA, complex expression binding, and state identification is shown in Listing 4.

```
1   def upload_movie(state, review_0: Review, rating_0: Optional[int]) -> bool:
2     success_0 = True
3     __cond_0 = rating_0 is not None
4     if __cond_0:
5       success_1 = review_0.upload_rating(rating_0)
6     movie_id_0 = state["movie_id"]
7     success_2 = phi(success_0, success_1)
8     __call_0 = review_0.upload_movie_id(movie_id_0)
9     __ret_0 = success_2 and __call_0
10    return __ret_0
```

Listing 4: Preprocessed `MovieId.upload_movie` code.

4.2 Control flow graph generation

4.2.1 Statement-level CFG

After preprocessing, the resulting AST is analyzed and transformed into a control flow graph. Each line is transformed into a `Statement`, and corresponds to a node in the CFG. `Statements` carry additional information, such as the variables read and written to, or whether or not it corresponds to a remote call. Control flow statements, such as `if` or `while` statements, create splits and cycles in the control flow graph.

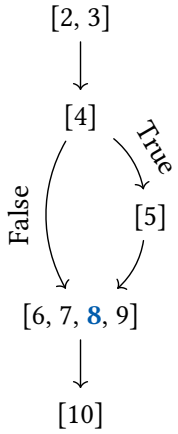
4.2.2 Block-level CFG

In this step, individual statements are grouped together into their basic blocks. For a given *leader* statement, we follow along its path in the CFG either until a branch is detected, or we reach its immediate postdominator. The path taken creates a block of nodes, where each block is a sequence of statements without any branching or other control flow.

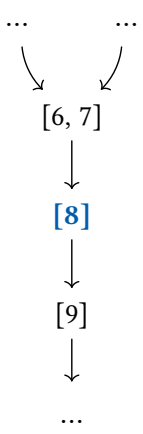
As an example, the `Statements` in the preprocessed `MovieId.upload_movie` method from Listing 4 are grouped together to form the CFG in Figure 10a.

4.2.3 Split CFG

Finally, we analyze the blocks of the block-level CFG, and look for `Statements` that correspond to remote calls. These remote calls are put into their own node in the CFG. For example, the remote call in Listing 4-8 gets split out of the [6,7,8,9] block, as shown in Figure 10b.



(a) Block-level CFG.



(b) Split CFG, zoomed on the [6,7,8,9] block.

Figure 10: CFGs for `MovieId.upload_movie`, referencing line numbers from Listing 4.

4.3 Dataflow graph generation

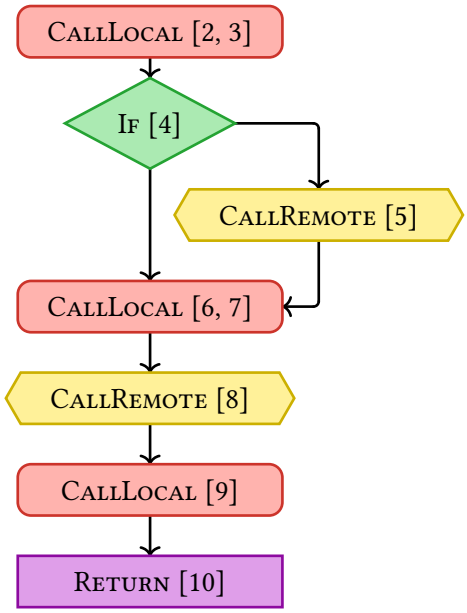
From the split CFG, the translation to the IR described in Chapter 3 is rather simple. In the split CFG, each node falls into three cases:

1. a control flow primitive, e.g. an `if` statement
2. a remote call to another operator
3. a `return` statement
4. a grouping of one or more statements, none of which fall into one of the categories above

each of these nodes are then translated into the equivalent nodes in the dataflow graph, respectively:

1. IfNode
2. CallRemote
3. ReturnNode
4. CallLocal

The result is a directed graph containing the nodes above. The running example in Listing 4 is transformed in the dataflow in Figure 11. In code, the dataflow also contains meta information, such as its unique reference, and the definitions of local blocks (Section 4.3.3).



MovieId Operator

```
1  def upload_movie(state, review_0: Review,
   rating_0: Optional[int]) -> bool:
2      success_0 = True
3      __cond_0 = rating_0 is not None
4      if __cond_0:
5          success_1 =
           review_0.upload_rating(rating_0)
6      movie_id_0 = state["movie_id"]
7      success_2 = phi(success_0, success_1)
8      __call_0 =
       review_0.upload_movie_id(movie_id_0)
9      __ret_0 = success_2 and __call_0
10     return __ret_0
```

Figure 11: `MovieId.upload_movie` (Listing 4) as a dataflow.

4.3.1 If and Return nodes

The transformation to IfNode and ReturnNode is relatively straightforward, as they both only require a single variable identifier. In the running example, `__cond_0` is the predicate variable for the IfNode (Listing 4-4) and `__ret_0` is the return variable for the ReturnNode (Listing 4-10).

4.3.2 Remote calls

CallRemote nodes require a reference to a dataflow, and a variable identifier to assign the answer to. In the running example at Listing 4-5, the corresponding CallRemote will contain a reference to the dataflow `Review.upload_rating`, and will store the answer in the variable `"success_1"` in local function state.

In addition, it requires a mapping from local variable identifiers to dataflow arguments. We can suppose the dataflow being called has the following method signature:

```
Review.upload_rating(self, rating: int) -> bool
```

Since the dataflow is being called with the `rating_0` variable, the correct variable mapping would be `{"rating_0": "rating"}`. Cascade will then be responsible for remapping the local function variable `rating_0` into the argument for the dataflow, `rating`.

4.3.3 Local blocks

For CallLocal nodes, the compilation process requires the creation of local blocks. These are independent functions that perform the computation in the corresponding block by reading and writing

to function state or operator state. These two pieces of state are passed as arguments to the function, thus the function itself is stateless.

By analyzing a statement's read and write sets, we can get the read and write sets of the entire block. For example consider the block at the start of the function in Listing 4, lines 2 and 3. By analysis of the statement's AST, we can extract the read and write sets line by line:

```
2  success_0 = True                          }   W = {success_0},   R = ∅
3  __cond_0 = rating_0 is not None           }   W = {__cond_0},   R = {rating_0}
```

Given the read and write information for each statement, we compute the block's overall read set by taking the union of the individual read sets; the same procedure applies to the write sets. In the case above, the block needs to read one variable from function state (`rating_0`) and it will write two variables (`success_0` and `__cond_0`). We combine this information with the block's original statements to generate the local block as such:

```
1   def upload_movie_block_0(state, func_state):
2     # read from function state
3     rating_0 = func_state["rating_0"]
4
5     # execute the block's statements
6     succes_0 = True
7     __cond_0 = rating_0 is not None
8
9     # write to function state
10    func_state["success_0"] = success_0
11    func_state["__cond_0"] = __cond_0
```

There are still some special cases to consider. If a block reads a variable that is first written to in the same block, then this variable does not need to be read from function state. This is thanks to SSA, which ensures that the read variable can only ever be written in one place, in this case the current block. This effectively reduces the read set of a block, as some variables can remain local only within the block's scope.

More advanced liveness analysis could also reduce the write set in a similar manner. If a variable is written to, but never read outside the local block, than we also do not need to write it to function state. This is related to potential future optimizations that aim to reduce function state discussed in Section 8.1.1.

Chapter 5

Execution

Cascade's IR is designed to be agnostic to the execution target, allowing execution on any dataflow system such as Apache Flink [11], Apache Spark [12], Naiad [13], or Styx [14]. In this work, we use PyFlink 1.20.1 as the execution target, combining it with Kafka as a message queue to enable cyclic dataflow processing.

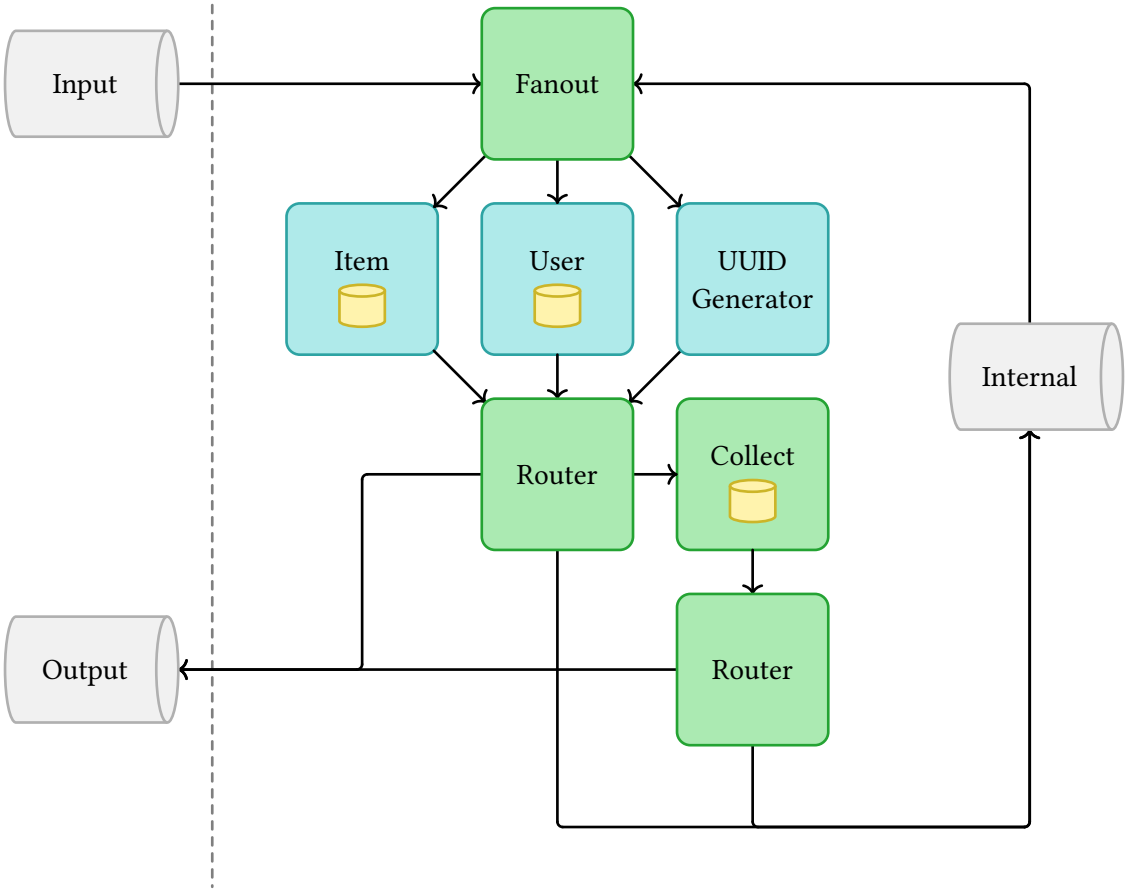


Figure 12: A Flink datastream implementing a Cascade program containing three operators.

5.1 Flink as an execution target

This section lists the major components of it Cascade's implementation of a PyFlink execution target. Figure 12 summarizes how a collection of operators may be represented in Flink through its `ProcessFunctions`.

Operators. In Flink, every `StatefulOperator` is interpreted by a `KeyedProcessFunction`, enabling keyed state storage within the operator. The state is represented as a mapping of class attribute names to their values, equivalent to Python's `__dict__` attribute for a class instance. On the other hand, stateless operators are transformed into `ProcessFunctions` and do not require key-based partitioning.

Fanout. An ingress `ProcessFunction`, called the `FanOutOperator`, is used to ingest and deserialize Events from the message queue and forward them to the correct operators via several output channels. Event targets can include both stateful and stateless operators.

CollectOperator. Events directed toward a CollectNode are routed to a `FlinkCollectOperator`, implemented as a `KeyedProcessFunction`. This operator is keyed using a combination of the event's unique ID and the node ID of the CollectNode. Events arriving at this node are buffered in the operator's state until all expected inputs have been received. Once all required events are available, the CollectNode will yield the collection of events.

Routing. The outputs of all operator datastreams are unioned to a `Router` operator before being sent to a Kafka sink. This operator is responsible for propagating events through the dataflow, setting its next target, and handling its call stack. This propagation step is handled by Cascade itself and does not need to be implemented for every execution target - the `Router` will just need to call the `Event.propogate` method. After propagation, the `Event` will either have a new target (a CallLocal or CollectNode), or it will have reached the end of its execution path. Depending on the event's progress through the dataflow, it is either re-ingested into the system for further processing (via an `internal` Kafka topic) or placed into a designated `output` topic, where it can be read by an external Kafka consumer.

5.2 PyFlink performance benchmark

There are many configuration options available in PyFlink and Kafka that have major impacts on performance. Notably, the heterogenous nature of PyFlink's integration with JVM-based Flink creates inherent performance bottlenecks due to overhead introduced by de/serialization and interprocess communication.

To help alleviate some of this performance impact, Flink introduced *thread mode* in version 1.15, using the PEMJA library (Python Embedded in Java). PEMJA allows Python code to run within the same thread as the JVM, reducing the overhead associated with traditional interprocess communication. Despite this promising implementation, significant parallelization challenges persist, primarily due to Python's Global Interpreter Lock (GIL). The GIL fundamentally restricts true multi-threaded execution in CPython, creating a critical bottleneck for parallel data processing. Notably:

- Only one thread can execute Python bytecode at a time.
- Increased parallelization attempts lead to contention and reduced efficiency.
- Complex synchronization requirements limit scalability.

These issues are not present in process mode, which uses an entirely separate Python process per task slot [30]. In order to determine an optimal setup for further experiments with PyFlink, we compare thread mode and process mode, using the login function of the DeathStar [23] hotel reservation workload as a basis for testing.

Additionally, in order to test multi-threaded performance, we attempt to spread parallelization across multiple taskmanagers by varying the number of task slots per taskmanager. In Flink, *task slots* are the primary unit of resource management and scheduling. Theoretically, having a single task slot per taskmanager would eliminate the problem of GIL contention, but could result in suboptimal resource allocation due to more distributed task management and more significant network overhead.

In this initial experiment, we keep the parallelism at a constant level (16) while varying how this parallelism is spread out across taskmanagers and task slots. On one end of the spectrum, we have the 1x16 configuration, corresponding to one taskmanager with 16 task slots. On the other end we

have the 16x1 configuration, corresponding to 16 taskmanagers, each with a single task slot. We also measure intermediate configurations, namely 2x8, 4x4 and 8x2.

5.2.1 Experimental setup

Experiments are executed on a system with two 64-Core processors with 512 GB RAM, running 64-bit Ubuntu 22.04.5 and Flink version 1.20.1. Each taskmanager is run in its own Docker container, but shares its resources with the host system, including other taskmanagers. Thus, a job with 16 taskmanagers will have proportionally less resources per taskmanager than a job with 4 taskmanagers. The amount of combined resources is constant for all experiments. Taskmanagers are coordinated by a single jobmanager. A single Kafka node is used for the message queue, using 32 partitions for each of the three required topics (`input`, `output` and `internal` events). Additional PyFlink configuration settings are described in Appendix B.

5.2.2 Results

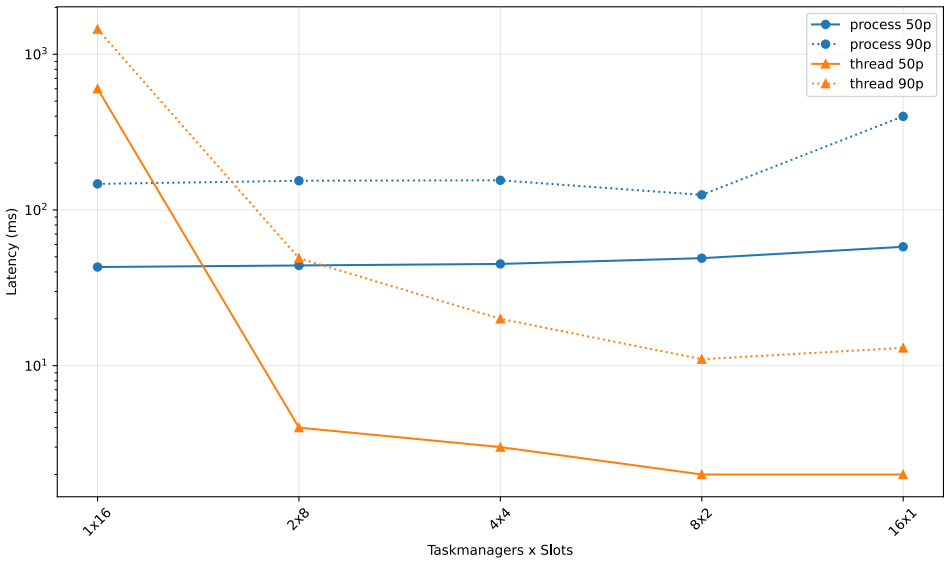


Figure 13: User login latency for 1000 requests per second and parallelism of 16.

Results shown in Figure 13 show that thread mode significantly increased performance under the right conditions. We see that performance in process mode remains relatively constant, with slightly faster latencies in configurations with lower taskmanagers and more slots. We presume that the cause of this trend is due to higher overhead because of additional communication requirements between multiple taskmanagers.

The worst case for thread mode is when one taskmanager shares sixteen slots. We believe this is due to GIL contention. The best performing configuration overall is thread mode with 16 single-slot taskmanagers, reducing the median login latency to just 2 ms. As a result, we use multiple, single-slot taskmanagers in thread mode for all further experiments.

5.2.3 Future implementations

Emerging Python implementations include experimental GIL-free versions[1], which could mitigate some of the issues with PyFlink thread mode, enabling us to increase the number of task slots per taskmanager. Other possibilities include switching to a JVM-based Cascade runtime implementation.

[1]PEP 703: https://peps.python.org/pep-0703/

However, Cascade has the goal to be runtime-independent. As such, other runtimes can be implemented with minimal effort, and could include more suitable runtimes such as Styx or Naiad. Runtimes with support for cyclical dataflows and exactly-once guarantees could also lead to increased performance as they wouldn't rely on an internal message queue with Kafka.

Chapter 6

Optimizations

In this chapter, we present some optimizations that are possible with the current intermediate representation. We focus notably on taking advantage of dataflow's inherent parallelism, both through static code analysis in Section 6.1 and a more dynamic approach in Section 6.2. Further opportunities for optimization are discussed in Section 8.1.1.

6.1 Parallelization

Dataflow programs are often embarrassingly parallel. The data-driven nature of Cascade's IR inherently exposes opportunities for parallel execution, as each CallLocal block is eligible for execution as soon as all their required input data (i.e. `reads`) becomes available. In this section, we provide a method for the automatic parallelization of these dataflow nodes through a data dependency analysis of their reads and writes. We run experiments on the DeathStar [23] movie review workload and evaluate its effectiveness compared to the baseline dataflow.

6.1.1 Method

6.1.1.1 Data dependency analysis

Given a dataflow graph $\mathcal{D}$ consisting of nodes $N = \{n_1, n_2, ..., n_m\}$, we can derive its corresponding dependency graph $G = (N, E)$ by analyzing the read and write sets of nodes. For each node $n \in N$, we define $\texttt{WriteSet}(n)$ as the set of variables written by n and $\texttt{ReadSet}(n)$ as the set of variables read by n. The directed edges in the dependency graph are then defined as

$$E = \{(n_i, n_j) \mid n_i, n_j \in N \text{ and } \texttt{WriteSet}(n_i) \cap \texttt{ReadSet}(n_j) \neq \emptyset\}.$$

This creates the dependence graph G, where an edge from n_i to n_j indicates that n_j depends on n_i because it reads at least one variable written by n_i.

Thanks to the properties of SSA, for each variable v written to by the dataflow, there exists exactly one $n_i \in N$ such that $v \in \texttt{WriteSet}(n_i)$. Therefore, if two nodes n_i, n_j read the same variable v_1, they will both have the same parent n_p such that $v_1 \in \texttt{WriteSet}(n_p)$.

We consider all dependency graphs created this way to be acyclical. Cycles would suggest programming errors, such as reading a variable before it is initialized, and thus are not considered. However, it's possible that such graphs are disconnected. Because of potential side effects, all nodes still need to be executed in the dataflow graph. We solve this issue by adding a CollectNode at the end of disconnected dataflows, ensuring the dataflow only returns when all nodes have been executed. Further analysis into side effects could allow for optimizations that delete these nodes entirely, if they are purely functional. This could be inspired by existing literature such as [31] and [32].

6.1.1.2 Parallelization algorithm

The algorithm for parallelization is based on the dependency graph constructed in the previous section. The parallelized dataflow will largely follow the dependency graph, with one addition: nodes with indegree > 1 will be connected via an intermediate CollectNode as shown in Figure 14. In addition,

dependency graphs that are disconnected will require an additional CollectNode node at the end to join the separate calculations.

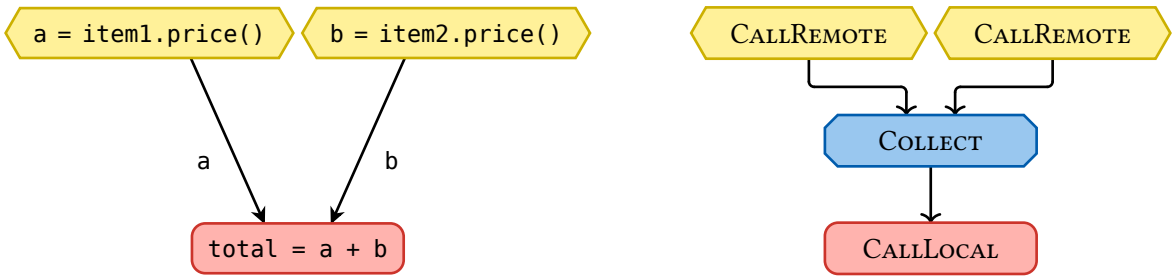


Figure 14: Dependency graph (left) to parallel dataflow (right).

When control flow is introduced, such as with IfNodes, the parallelize algorithm must keep the control flow structure in mind. Instead of running the parallelize algorithm for the entire dataflow, we can restrict it to running only on the basic blocks of the dataflow. These dataflow blocks can be considered sub-dataflows, formed by a subset of the original dataflow. As basic blocks, the CFGs of these sub-dataflows thus form simple path graphs, consisting of CallLocal or CallRemote nodes, and can easily be parallelized by the algorithm described in this section.

6.1.1.3 Local block merging

The parallelized dataflows created by the method above also highlights a different opportunity for optimization. Consider the following example:

```
1 class User:
2   def apply_discount(self, item: Item):
3     b = self.balance
4     o = self.overdraft
5     price = item.price()
6     b_total = b + o
7     DiscountService.apply(b_total, price)
```

The dependency graph and parallelized dataflow is shown in Figure 15. We note that this results in two CallLocal blocks in a row, block [3, 4] and block [6]. These could be merged into a single [3, 4, 6] block, removing a node from the dataflow. The removal of a node can result in performance gains, the underlying assumption being that one node is incurs lower latency than two due to the overhead required to travel the edge. In the Flink + Kafka execution target, this results in one less pass through the Kafka message queue and thus is very effective.

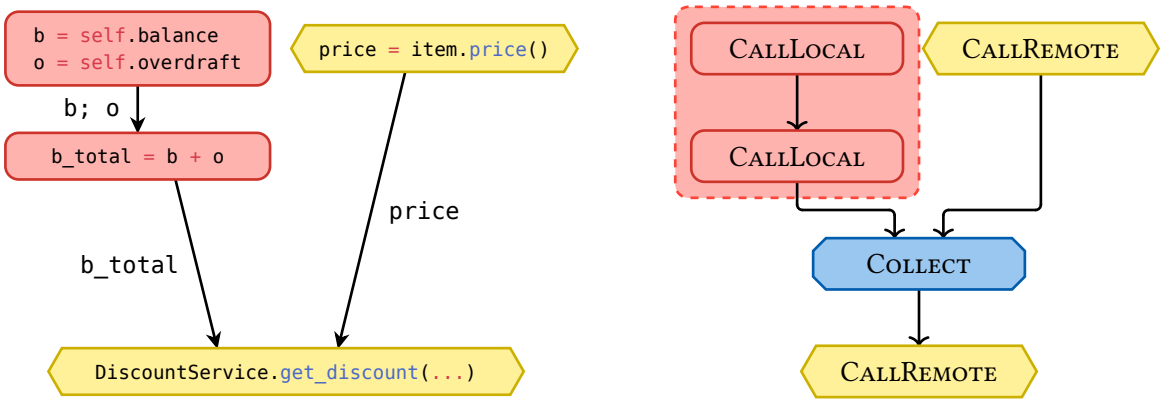


Figure 15: Dependency graph (left) resulting in two subsequent CallLocal nodes (right).

6.1.2 Benchmark

To evaluate the parallelization optimization, we employ the movie review workload from the DeathStar benchmark [23]. Specifically, we simulate requests to the `/review/compose` endpoint by uploading review data to a `Frontend` operator. The workload consists of three stateless operators (`Frontend`, `Text`, `UniqueId`) and three stateful operators (`Review`, `User`, `MovieId`). The `Text`, `UniqueId`, `User` and `MovieId` serve as intermediate operators to the `Frontend` workload, as shown in Listing 5. This dataflow is then parallelized to run the four remote calls in parallel as in Figure 16.

```
1 @cascade
2 class Frontend():
3     @staticmethod
4     def compose(review: Review, user: User, title: MovieId, rating: int, text:
      str):
5         UniqueId.upload_unique_id(review)
6         user.upload_user(review)
7         title.upload_movie(review, rating)
8         Text.upload_text(review, text)
```

Listing 5: DeathStar benchmark workload function.

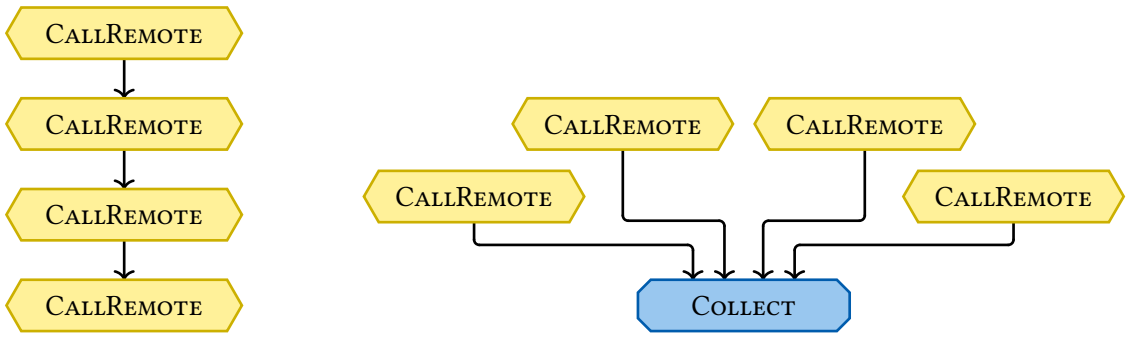
A summary of the relevant dataflows are shown in Table 3. We define the metric *critical path weight* as the *source-to-sink* path with the highest weight. The *weight* of a path is the **number of operators** that an event must go through during propagation. Note that this includes operators inside nested dataflows, i.e. with CallRemote nodes. In other words, it is the total number of CollectNode and CallLocal nodes that an Event travelling through a dataflow will encounter during its propagation through the dataflow.

Since the baseline frontend calls the four nested dataflows sequentially, its critical path weight has the property of being the sum of its parts (2 + 2 + 3 + 1 = 8). In contrast, the parallelized frontend dataflow runs the four dataflows in parallel. This results in a critical path weight of 4; equal to the maximum critical path weight of the four dataflows, **plus one** due to the additional CollectNode. The additional CollectNode also increases the number of operator calls by one relative to the baseline.

Dataflow	Critical path weight	Operator calls
`UniqueId.upload_unique_id`	2	2
`User.upload_user`	2	2
`MovieId.upload_movie`	3	3
`Text.upload_text`	1	1
`Frontend`: baseline	**8**	**8**
`Frontend`: parallel	**4**	**9**

Table 3: Critical path weight and number of function calls for different dataflows.

The benchmark increases the request frequency from 250 to 2500 requests/sec and measures the p50 and p99 round-trip latency of requests. A single request corresponds to an entire run-through of the dataflow, and therefore requires eight operator calls per request (nine for the parallel dataflow), as per Table 3. A throughput of 2500 requests/sec therefore corresponds to 20,000 operator calls/sec, most of which are stateful.



(a) Baseline dataflow. (b) Parallel dataflow.

Figure 16: Dataflows for `Frontend.compose` from Listing 5.

6.1.3 Experimental setup

Experiments are executed on a system with two 64-Core processors with 512 GB RAM, running 64-bit Ubuntu 22.04.5 and Flink version 1.20.1. Due to limitations in PyFlink's thread mode (Section 5.2), we run 24 taskmanagers, each with only one task slot, in local Docker containers. To simulate more realistic distributed conditions, each taskmanager's resources were limited to 4 CPUs and 8 GB RAM. They are coordinated by a single jobmanager. A single Kafka node is used for the message queue, using 32 partitions for each of the three required topics (`input`, `output` and `internal` events).

6.1.4 Results

Introducing the additional `Collect` operator increases the total number of operator calls by one (Table 3). Nonetheless, Cascade reduces the critical path length from 8 to 4, with the other three paths being even shorter. Since the parallel dataflow runs the four remote calls in parallel, we can expect around 4x speedup, assuming each intermediate call has similar latency and the CollectNode latency is negligeable.

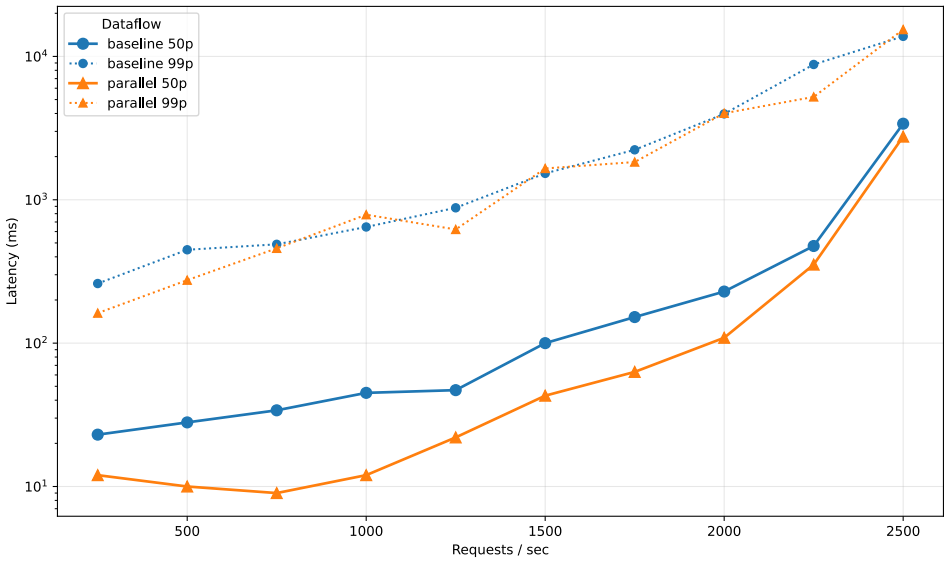


Figure 17: `Frontend` latency for baseline and parallel dataflows.

The results depicted in Figure 17 support this hypothesis, demonstrating a 2x speedup at 250 req/s for the p50 latency, rising to 2.8x at 500 req/s and 3.8x at 750-1000 req/s when compared to the baseline. From 1000-2000 req/s, the speedup is closer to 2x, and we notice that the speedup drops to around 1.3x for 2250-2500 req/s. We believe this break in the scaling behavior is a consequence of the backpressure due to increased time spent in the Kafka queue for both dataflows. There may also be some effects of key contention, as the number of initialized users and movies does not scale with the increased requests/second.

The 99p latency also shows some improvement throughout, demonstrating 1.6x speedup for low throughputs of 250-500 req/s. However, this improvement is not as visible at higher throughputs, suggesting that parallelization may have some adverse effect on the 99p latency under increased load.

In summary, our key takeaway is that parallelizing computation effectively shortens the critical path weight, leading to beneficial speedups at scale. We observe speedups for median latencies of up to Px, where P is the number of parallelized calls.

6.2 Dynamic prefetching

Prefetching is a technique to retrieve data from a cache or external object store by predicting that it will be used in the future. Since waiting on the results of I/O operations is inefficient, accurate predictions will be able to perform I/O operations in the background whilst performing calculations in the foreground, making better use of available resources. Through dynamic analysis of a program, one could start to predict which variables could be used in the future with reasonable accuracy. We introduce the concept of dynamic prefetching in the context of dataflows by first providing a motivating example.

Consider the `NavigationService` from Listing 6. It provides directions from an origin location to a destination through a `MapService`, and also serves advertisements to some users via an `Advertiser` service. A small subset of users are paying costumers, and will not be served ads.

```
class NavigationService:
  @staticmethod
  def get_directions(origin: Geo, destination: Geo, user: User):
    directions = MapService.get_route(origin, destination)
    if not user.is_ad_free():
      recc = Advertiser.get_recommendations(destination, user)
      return (directions, recc)
    else:
      return (directions, None)
```

Listing 6: `NavigationService` code.

Suppose we somehow knew that the none of the `User`s are ad free and thus will always get served recommendations by the `Advertiser` service. In this case, the `if` predicate will always evaluate to `True`. In this situation, parallelizing the remote call to `Advertiser` could be an effective way to increase performance, as suggested by the parallelization experiment in Section 6.1.

Consider now that 10% of `User`s are paying and thus ad free. In this case, it might still be worth it to parallelize these calls, as we will only be doing extra work 10% of the time. As a result, 90% of the time, the prediction will be correct and we may have saved some time by parallelizing the `Advertiser` call. If the parallelism is high enough, the only extra work needed is the overhead induced by the extra CollectNode, as the calls will be run in parallel anyway. Once the `if` predicate is reached and the predicate condition becomes known, Cascade could either continue waiting for the prefetched call, or discard it if it is not needed.

This is the idea behind dynamic prefetching. It has similarities to branch prediction in computer architecture, where the branch predictor plays a critical role in achieving high performance [33]. In this section, we investigate its effectiveness in the context of dataflow execution.

6.2.1 Method

In order to test whether prefetching could be a viable strategy for optimization, we first simulate it by running two different versions of the same dataflow, shown in Figure 18. In this experiment, the creation of the prefetched dataflow was done manually. The actual branch prediction is out of scope for this experimen, but one could refer to elements of literature [33], [34] for inspiration on strategies to implement the dynamic analysis. We envision some sort of continous probabilistic analysis to be a good starting point for measuring the expected value of prefetching, as described in Section 7.2.

6.2.2 Benchmark

To test the potential of dynamic prefetching, we benchmark usig the code from Listing 6. This example contains one stateful operator, `User`, and three stateless operators: `NavigationService`, `MapService` and `Advertiser`. The `MapService` operator is responsible for giving directions between two coordinates. For the purposes of this experiment, this call is mocked and replaced by a waiting time of 10 ms. The `Advertiser` operator returns advertisements tailored to the preferences for a certain `User` around their destination. The `NavigationService` combines the two previous operators, giving both directions and advertisements to the user. However, if the user is a paying member, they will not get served advertisements.

Figure 18 highlights the difference between the baseline and prefetched versions of the `NavigationService.get_directions` dataflow. The baseline dataflow is able to run the calls to `MapService` and `User` in parallel. The prefetched dataflow also includes a call to `Advertiser` in parallel, that is saved in the function's local state but which may or may not be used.

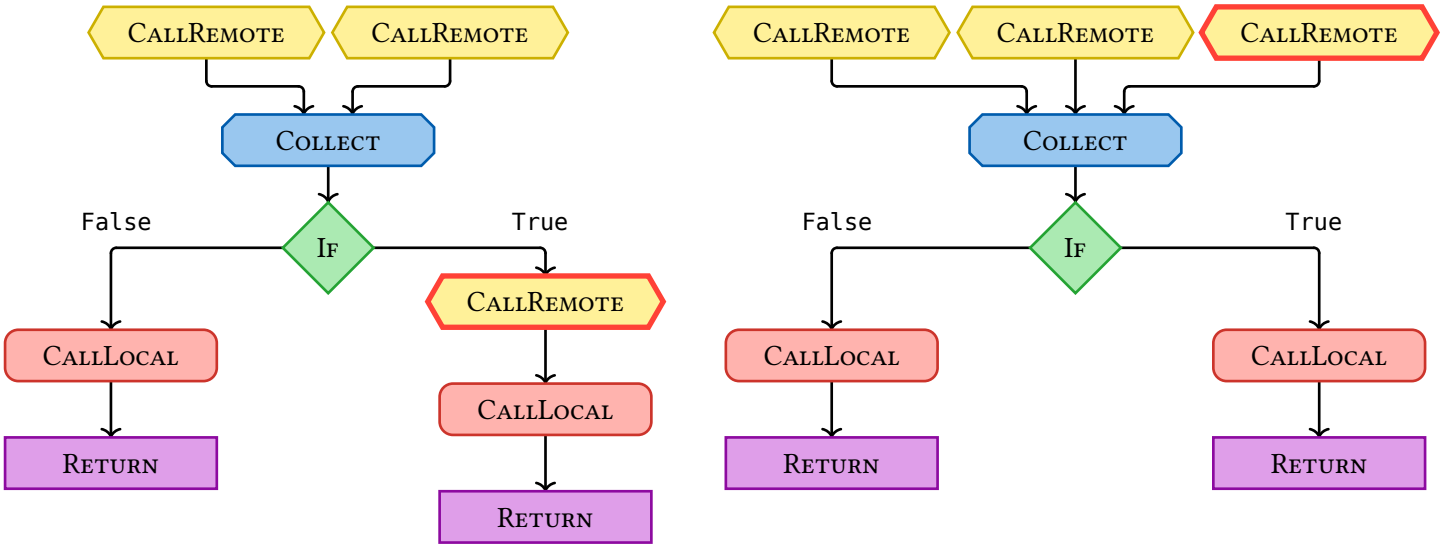


Figure 18: Baseline dataflow vs prefetching dataflow.

To run the experiment, we initialize 1000 users and vary the ratio of ad free users. This has a direct impact on the branch probability, as the ratio of ad free users is equal to the probability to take the `if` branch in Listing 6-5.

6.2.3 Experimental setup

Experiments are executed on a system with two 64-Core processors with 512 GB RAM, running 64-bit Ubuntu 22.04.5 and Flink version 1.20.1. We run 8 taskmanagers, each with one task slot, in local Docker containers. Each taskmanager's resources are limited to 4 CPUs and 8 GB RAM and are coordinated by a single jobmanager. A single Kafka node is used for the message queue, using 32 partitions for each of the three required topics (`input`, `output` and `internal` events).

6.2.4 Results

We vary the ratio of ad free users between 10%, 50% and 90%, and measure the round-trip latency of 500 requests per second over 30 seconds.

The results for the `NavigationService` workload are summarized in Figure 19. To display these results, we remove the top 5% latencies, and show the mean latency for the baseline and prefetched dataflows, similar to a winsorized mean. We justify the removal of tail-end latencies since we expect a bimodal long-tail distribution. We argue removing the tail-end of latencies and taking the mean generates a more representative average than simply taking the mean (which would be affected by extreme values) or the median (which isn't as representative for bimodal distributions).

We can clearly see the difference in clustering that highlights the difference in approach between the two experiments. Whilst the prefetched dataflow shows a single cluster and a long tail distribution, the baseline dataflow sees two clusters of latencies, roughly weighted by the branch probability. The two clusters of latencies indicate the important difference in round-trip latency depending on the outcome of the branch predicate. This problem is mitigated by the prefetched version of the dataflow, which, given some overhead cost, is faster on average when the branch probability is sufficiently high.

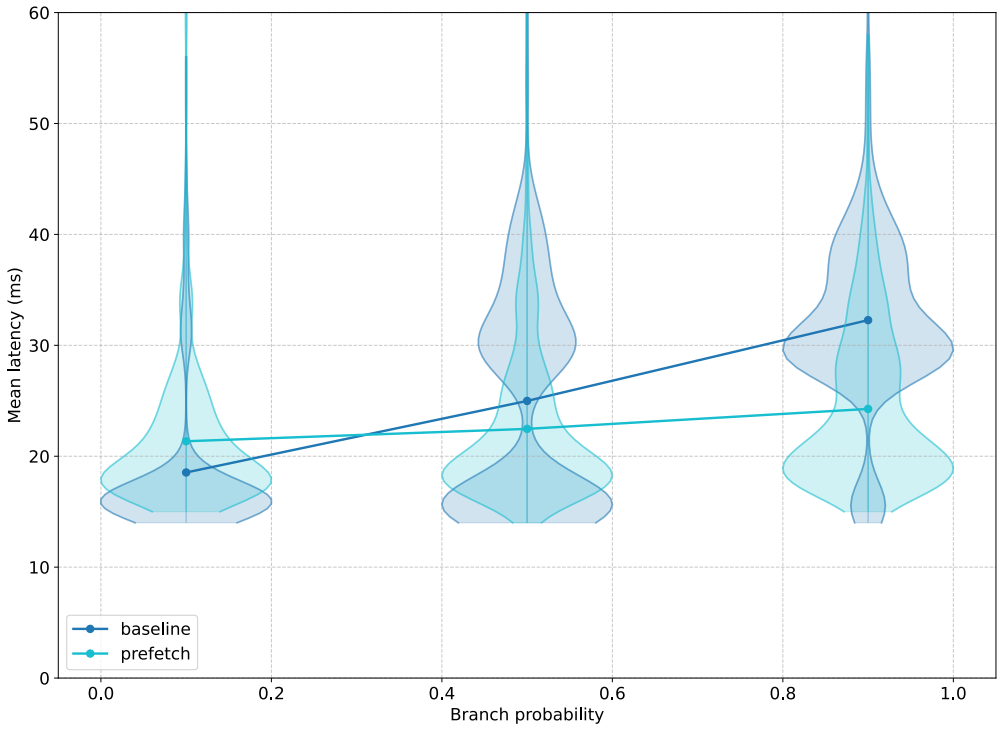


Figure 19: Baseline vs prefetching results for the `NavigationService` workload.

6.2.5 DeathStar benchmark

As an aside, we also run the experiments on the `MovieId` workload of the DeathStar benchmark. This is possible because the call to `Review.upload_rating` (see Listing 4-5) can be done speculatively, because in this case a write of `None` corresponds to a null-op. The branch being tested is therefore Listing 4-4, and we have used a custom workload that varies the branch probability between 10%, 50% and 90% by controlling the `rating` argument. We measure the round-trip latency for 500 requests per second over 30 seconds. An important distinction is that we add an additional collect node for the baseline dataflow that isn't necessary. This is to provide a more direct comparison between the baseline and prefetched dataflow by eliminating overhead related to the collect operator.

Results, shown in Figure 20, shows a similar trend as the previous experiment, with a relatively flat prefetch curve that is lower than the mean latency for the baseline at higher branch probabilities. Noticeably, we observe that there is no obvious bimodal distribution for the baseline dataflow. We speculate that this could be due to latency numbers being quite low already. When compared to the previous experiment, the DeathStar benchmark suggests that dynamic prefetching would be effective at lower branch probabilities compared to the `NavigationService` workload. This could be due to the fact that the latency of the `Review.upload_rating` dataflow is very close the latency of the `Review.upload_movie_id` dataflow, resulting in higher relative benefits from parallelization.

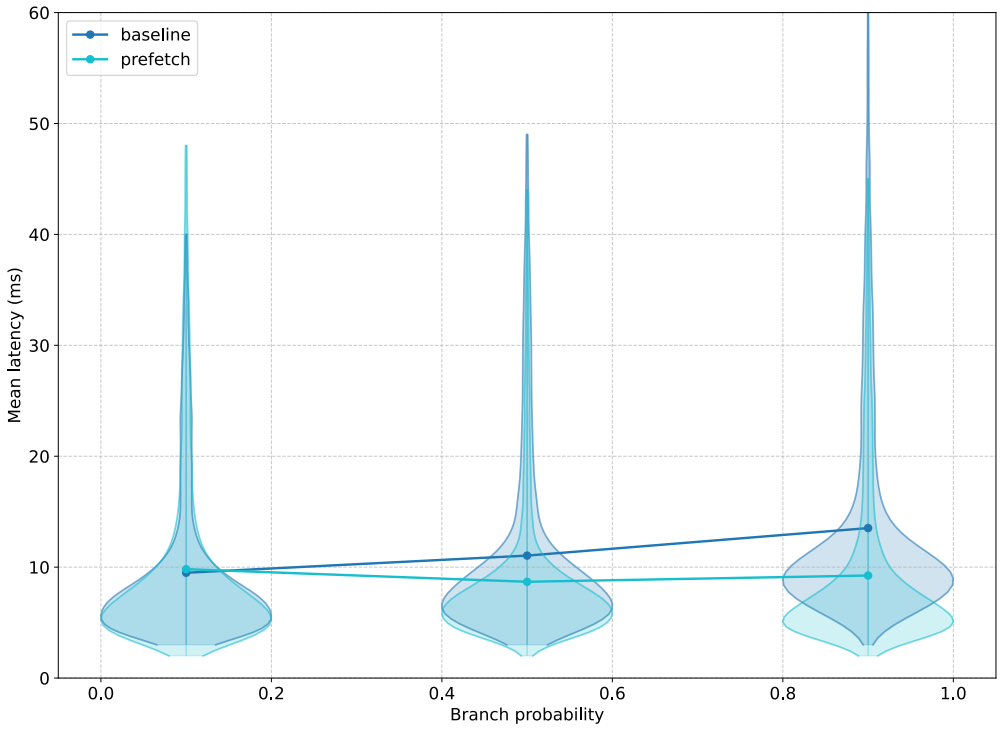


Figure 20: Baseline vs prefetching results for the DeathStar `MovieId` workload.

6.2.6 Limitations

An important limitation with the current implementation is that the prefetched dataflows cannot choose to discard unwanted results. By connecting together with a CollectNode, dataflows could be waiting on a result that they don't even need, limiting performance if prefetching takes a long time. A future iteration could allow the dataflow to discard the result of a prefetched node, and immediately continue execution rather than waiting on a useless result. This could also help alleviate some overhead induced by introducing an extra CollectNode in some cases.

Further discussion on dynamic analysis and speculative writes is contained in Section 7.2.

Chapter 7

Discussion

7.1 Parallelization

Our results in Figure 17 show that parallelization can be very effective for workloads with high degrees of parallelism. Our approach, described in Section 6.1, follows similar to approaches used in compiler optimization [35]. Both Cascade and other approaches rely on the *fundamental theorem of dependence*: any reordering transformation that preserves every dependence in a program preserves the meaning of that program. While loop-oriented parallelization is common, Cascade aims to be more aggressive, aiming for parallelization at the block level.

To the best of our knowledge, Cascade is unique in parallelizing dataflows originating from imperative code for execution on stream processing systems. In order to provide a comparison with existing approaches to parallelization, we refer to elements of research related to parallelizing imperative programs for parallel architectures, such as multi-core hardware [36], [37], [38] or dataflow hardware [39]. The approach by Gasmi and Hasanaoui [36] models the problem as a mapping of a dependency graph to a multi-core computer architecture. The mapping is solved using a Satisfiability Modulo Theory (SMT) solver that takes into account constraints such as number of processors and optimizes for execution time. We argue that our problem is simpler, as we do not have limits on parallelization degree. Cascade targets transactional cloud applications, and thus we consider only applications that are reasonably parallel. Excessively parallel programs, such as image processing, machine learning, or scientific computing tasks are more suited to high-performance computing systems, and are not considered suitable for Cascade.

A significant challenge that was unaddressed in dataflow parallelization involves handling side effects. Cascade's current approach does not consider the ordering of side-effects, and considers that nodes can be safely re-ordered as long as their dependencies are met. Further refinements could incorporate side effect analysis techniques as proposed by Callahan and Kennedy [31], and Herhut et al. [32]. This could also enable aggressive optimizations when nodes are proven to be free of side effects, for example by eliminating unnecessary nodes, speculative execution, and more flexible reordering of operations.

7.1.1 Limitations

While the parallelization method presented demonstrates noteworthy performance improvements, several limitations should be acknowledged.

Critical path optimization. While the approach reduces critical path weight, it does not explicitly optimize for minimizing the time duration of the critical path, and does not consider varying execution times of different operations. For example, it may not be worth to parallelize a CallLocal block if it can be merged with another block instead. The current algorithm does not take this into account.

Control flow complexity. As noted in the chapter, the algorithm is restricted to basic blocks when control flow is introduced. More sophisticated techniques would be needed to handle complex control flow patterns while maintaining parallelism. Dynamic prefetching from Section 6.2 is an attempt to address part of this issue, but this still does not consider loops. Remote calls inside a loop body are a very good target for parallelization, as the entire loop could be run in parallel in some cases.

7.2 Dynamic prefetching

The results in Figure 19 show that, in `NavigationService` case, the prefetched dataflow is faster even if the branch is taken only half the time. We argue that this might not always be the case, and that the efficacity of prefetching and branch prediction heavily relies on a numerous factors. For example, if the prefetched remote call takes a considerable amount of time relative to the dataflow, this would require a higher branch probability in order for the prefetching to be effective.

Another consideration is the overhead introduced by the CollectNode operator. This example already included a CollectNode in the baseline dataflow, and thus adding an extra parallel call introduces minimal overhead. However, introducing a CollectNode call in a different dataflow may result in too high of an overhead to be worth it, depending on the backend implementation.

A potential solution to these issues is a dynamic approach, that measures both the branch probability P_b and the latency difference between the baseline and prefetched dataflows as the system is running. Future versions could calcualte the expected latency reduction in order to determine when prefetching is worth it:

$$\text{Expected latency reduction} = \mathbb{E}[t_{\text{base}}] - \mathbb{E}\left[t_{\text{prefetch}}\right]$$

where

$$\mathbb{E}[t_{\text{base}}] = P_b\mathbb{E}[t_{\text{true}}] + (1 - P_b)\mathbb{E}[t_{\text{false}}]$$

and we can reasonably assume that all latencies are independent and identically distributed from some normal distribution

$$t_i \sim \mathcal{N}_i$$

and that values for P_{branch}, $\mathbb{E}\left[t_{\text{prefetch}}\right]$, $\mathbb{E}[t_{\text{true}}]$ and $\mathbb{E}[t_{\text{false}}]$ can be approximated dynamically through continous measurement. Such a system could also adapt to changing branch probabilities and latency changes in the long term.

This technique is similar to the approach by [40], which occasionally performs profiling on running programs to determine hot data streams that can be prefetched. Machine learning techniques, perhaps inspired by [33], could be a starting point for deeper dynamic analysis.

Finally, the example in Section 6.2 only discussed prefetching of values that may or may not be read. We argue that this technique could also apply to writes as well. One could speculatively write and update values of an operator when a branch prediction is made, but this would require some compensating action similar to the saga pattern [41] if the prediction is wrong. A more isolated process, inspired by techniques in thread level speculation [42], could also be possible in order to increase consistency at the cost of availability.

7.3 Related work

7.3.1 Dataflow programming

Dataflow programming represents a paradigm where program execution is modeled as a directed graph. In this graph, nodes symbolize operations that accept inputs and produces outputs, while directed edges signify the flow of data, channeling the output of one operator as the input to another. Early research in dataflow programming explored dataflow computer architectures that contrasted

with the control-flow model prevalent in von Neumann architectures [43]. In von Nuemann machines, static data is acted upon by a sequence of instructions driven by some control flow. The main benefit of the dataflow model is that the data-driven nature of its execution inherently exposes opportunities for parallel processing, as operations become eligible for execution as soon as all their required input data becomes available.

Early dataflow architectures were designed to exploit the inherent parallelism and implicit synchronization in programs by eliminating the sequential program counter of von Neumann machines. Dataflow programming semantics were first introduced in the 1970s [44] in order to permit concurrent execution of independent program parts. The Manchester Dataflow Machine [15] and MIT Tagged Token architecture [16] compiled functional programs into dynamic dataflow graphs for parallel execution on specialized architecture. However, the adoption of dataflow architectures never became widespread, due to issues such as the large overhead of fine-grained parallelism on realistic programs [45].

A resurgence to the ideas developed in this early era of computing came with the advent of big data. Software systems like MapReduce [46] express computations that map trivially to dataflow graphs. Further evolution of the field lead to projects like Apache Flink [11], Apache Spark [12] and Naiad [13], extending the possibilities of the dataflow graphs with more complex operations (such as joins, stateful computations and iterations), and embracing continuous stream processing. These novel systems leverage principles of distributed computing to achieve scalability and fault tolerance, allowing them to process very large datasets across clusters of machines. These capabilities opened up new application domains for dataflow programming, such as real-time analytics and complex event processing. With Cascade, we argue that cloud applications also fall under these application possibilities, and that they can be expressed as dataflow graphs that can be distributed, scaled and optimized.

7.3.2 Compiling imperative code to dataflows

The systems mentioned above have generally been executed via a functional-style language, such as through the Tagged Token *Id* language, or Flink's *Datastream API*. Compilation from imperative code to dataflows was initially explored by the theoretical work of Beck et al. [17], with a focus on execution on dataflow architectures. Their translation makes both data and control dependencies explicit, and provides basis for Cascade's approach to compilation. The end goal of Cascade's IR is not to execute on dataflow architectures, but rather to provide a practical, expressive IR that can be optimized and executed on modern dataflow systems.

More recent approaches, such as Labyrinth [21], compile imperative code to modern stream processing systems instead. However, Labyrinth is limited in scope, focused on compilation of imperative control flow constructs such as `while` loops. Outside of control flow, Labyrinth retains many characteristics of functional programming. Second order functions such as `map`, `reduce` and `filter` are still required for describing computation and are not suited for an object-oriented approach to dataflow programming.

7.3.3 Intermediate representations in parallel dataflow systems

Optimizing dataflows directly in stream processing systems like Flink [11] and Spark [12] is challenging due to limited program context. Dataflow graphs often contain user-defined functions (UDFs) which are treated as black boxes, making it difficult to apply optimizations past the physical level. Many real-life dataflows are dominated by UDFs and some works have attempted to look under these black boxes, for example through syntactical dataflow modification, dataflow transformations, and inferring semantics such as read/write sets [26], [47], [48].

Within dataflow systems, IRs have started to emerge as a valuable abstraction layer between high-level programming languages and underlying execution engines. For example, Emma [49] is an ANF-

based, embedded domain-specific language in Scala. This gives the Emma IR a much wider program context, allowing for a separate class of optimizations that would not be possible using systems like Flink or Spark alone. These optimizations would usually be left to the programmer to find and write, and include automatic caching, join order optimization, and partial aggregation. Similarly, DryadLINQ [50] uses LINQ, a domain-specific language that treats data as parallel collections. The Mitos runtime extends this idea to introduce optimizations for iterative algorithms, such as loop pipelining and loop-invariant hoisting [21], [51]. However, the scope of these systems remains largely confined to machine learning and big data analytics workloads, and programming is typically still done in a functional style.

Another example is Tensorflow [2], wherein high-level control-flow constructs are compiled to dataflow graphs using a handful of primitives. These primitives have some similarities with Cascade's IR, such as the `Merge` operation (CollectNode) or the `Switch` operation (IfNode). Another primitive, the `Next` operator, is used for advancing iterators in for/while loops and could eventually have an equivalent implementation in Cascade when loops are formally introduced. TensorFlow's representation of a `while` loop is shown in Figure 21. TensorFlow is also responsible for the distributed execution of the dataflow, mapping nodes in the graph to a given set of devices and partitioning the graph into subgraphs - something Cascade leaves to the runtime execution engine. However, TensorFlow's API uses functions such as `cond(pred, true_fn, false_fn)` and `while_loop(pred, body, inits)` to express control-flow constructs, resulting in a more awkward developer experience. Cascade aims to be more natural to use for developers accustomed to programming in a more object-oriented style. Unlike systems that expect user-written dataflow graphs, the Cascade compiler targets standard imperative code and generates the dataflow, complementing prior work by raising the abstraction level.

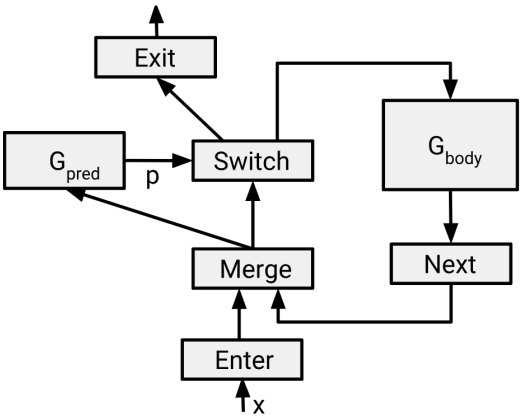


Figure 21: TensorFlow's representation of a `while` loop, from [2].

Finding a suitable intermediate representation, that is fit for general purpose, remains challenging. A general-purpose IR, such as LLVM [29], enables efficient execution for a wide set of source languages and execution models, even when they impose conflicting guarantees or semantics, by offering a unified and extensible representation. Recently, Flo [52] has attempted to unify stream processing semantics, recognizing that existing streaming languages do not agree on what constitutes a stream, and vary in semantics for state persistence. Cascade could benefit from Flo's approach in order to provide more formal guarantees, especially when complex optimizations are applied. At the same time, we also argue that Cascade's IR serves a purpose that is orthogonal to stream processing. While execution is done in Flink, a stream processing system, Cascade's IR is based solely on dataflows, and its execution model is more closely related to actor models rather than stream processing.

7.3.4 Stateful computation models

The actor model is a conceptual framework for designing and building concurrent and distributed systems [53]. It defines a system as a collection of independent software entities called actors, which interact with each other exclusively through message passing. Each actor contains mutable state that can only be modified by the actor itself. This framework has been successfully implemented in distributed systems languages such as Akka [54], Erlang [55] and Orleans [6]. The actor model maps rather easily to both object-oriented programming and the operators used in dataflow systems such as Flink. While the Cascade programming model is similar to the actor framework, we differ by applying program analysis and optimizations at a much finer granularity.

Azure's Durable Functions [10] is a conceptual framework for designing and building concurrent and distributed Stateful-Functions-as-a-Service (SFaaS) systems. The key characteristic of the system is that it enhances FaaS with actors, workflows and critical sections. Durable Functions and Cascade share similar goals: both attempt to deal with compute-storage separation problems seen in FaaS systems by incorporating exactly-once processing, synchronization and concurrency control. However, Cascade aims to be agnostic of the execution target, instead leveraging the execution target for exactly-once guarantees and persistence, whilst incorporating our own optimizations and providing a general-purpose intermediate representation.

7.3.5 StateFlow

This work is an extension of StateFlow [22], a compiler pipeline and dataflow system built for low-latency cloud applications. Cascade improves upon StateFlow by providing an IR that's more expressive and easier to optimize. We list some improvements here.

Dataflow optimizations. This thesis has extended StateFlow through automatic optimization of dataflow graphs, allowing for fine-grained optimization of user code. We introduce two optimizations, parallelization and dynamic prefetching, and lay the groundwork for other possible optimizations.

Nested dataflows. Cascade's IR allows for dataflows to call other dataflows without any restrictions thanks to the inclusion of the Event's call stack. This means that the IR presented in this thesis is much closer to what the developer expects, as it parallels with nested function calls in Python. This also means Cascade allows for recursive functions, something previously impossible in StateFlow's framework. Importantly, nested dataflows through CallRemote nodes has allowed for more flexibility with regards to optimization. Each remote call is seen as a separate node, rather than being inlined in the original dataflow. These explicit remote calls have allowed for better reasoning about optimizations such as parallelization. In contrast, StateFlow's linear dataflow approach fails to exploit the inherently parallel nature of dataflow graphs, limiting performance in real-world applications.

Backend tuning. StateFlow focuses on showcasing a wide variety of backends, including not also PyFlink, but also Flink, Apache Beam, AWS Lambda and Cloudburst. This thesis uses only PyFlink as a backend. In Zorgdrager's master's thesis on StateFlow [56], PyFlink is shown to be the worst performing backend, and was omitted from experiments requiring throughputs of >100 requests per second. After more extensive performance fine-tuning, not to mention four years worth of development on PyFlink, this latency was decreased by several orders of magnitude. While a direct performance comparison is hard to make, the user login function from the DeathstarBench hotel reservation workload was able to be executed at single-digit millisecond latencies at 1000 requests per second in Figure 13. In the StateFlow thesis ([56], Figure 6.15), this was >300ms at 10 requests per second, albeit with a slightly different, but comparable, experimental setup.

Syntax limitations. We improve upon StateFlow in the compilation step, by providing a preprocessing stage that includes steps to address some of the limitations of StateFlow, such as being able to use multiple remote calls in a single statement, or using remote calls in `if` predicates.

System	Coding Style	Execution Model	UDF Optimization	Domain
Flink	Functional	Stream processing	No	Stream analytics
Spark	Functional	Batch dataflow	No	Stream analytics
Durable Functions	Imperative	Event-driven orchestration	No	Stateful cloud workflows
Emma/Mitos	Functional	Runtime-dependent	Yes	Big data processing
Labyrinth	Functional with imperative loops	Runtime-dependent	Yes	Big data processing
DryadLINQ	Declarative or functional	Batch dataflow	No	Distributed data-parallel execution
Naiad	Functional	Timely dataflow	No	Distributed data-parallel execution
Orleans	Object-oriented imperative	Actor model	No	Distributed applications
Tensorflow	Functional	Dataflow	Yes	AI/Machine learning
StateFlow	Object-oriented imperative	Linear stateful dataflow	No	Transactional cloud apps
Cascade	Object-oriented imperative	Stateful dataflow	Yes	Transactional cloud apps

Table 4: Comparison of selected stateful dataflow processing systems.

Chapter 8

Conclusion

We present Cascade, a compiler pipeline that transforms imperative code into stateful dataflows. We propose an intermediate representation and demonstrate how static code analysis and optimization can enhance performance whilst simplifying cloud application development. We improve the performance of a predecessor system, StateFlow [22], by providing a new IR structure and fine-tuning the PyFlink backend in Chapter 5. For optimizations, we focus on those that enhance parallelization in order to take full advantage of the dataflow model. Our experiments show that we can increase performance up to 3.8x in the DeathStar [23] movie review benchmark by running 4 remote calls in parallel in Section 6.1. Additionally, we show promising results for dynamic code analysis and branch prediction optimizations in Section 6.2.

8.1 Future work

We provide avenues for future work below. This section is split into two parts. Section 8.1.1 describes additional optimizations that could be performed with the IR described in this thesis. Section 8.1.2 describes extensions to the current IR that would allow for more developer flexibility and create further opportunities for optimization.

8.1.1 Additional optimizations

There are many more opportunities for optimizations with the current IR. We describe a handful here that we consider to be the most salient.

Stateless function inlining. Consider a small, stateless operator that has a single dataflow that generates unique ids using `uuid4`. A call to such a simple dataflow could be directly inlined into existing blocks, rather than using a CallRemote node. Such an inlining optimization could thus remove a CallRemote node from dataflows.

Stateless local block dislocation. When a stateful dataflow is split into blocks as in Section 4.2.3, some of these blocks could happen to be stateless even if the dataflow isn't. Currently, these blocks are colocated as normal with the original stateful operator, even though they don't access any state. Instead, they could be move to a different, stateless operator, reducing overhead associated with state access and keyby partitioning.

Function state size reduction. Currently, function state is tied to an `Event` object that travels through the dataflow. It saves all variables written to function state but does not delete them. Live variable analysis could be used to delete dead variables from function state, which could be important for large, complex functions. This optimization could reduce the size of `Event` objects, generating speedups in (de)serialization and network latency.

IfNode removal. Current limitations to the CFG generation algorithm will always create an IfNode when an `if` statement is encountered. However this is not necessary when the statement's branches do not contain remote calls. In this case, the `if` statement and its branches could be kept inside a local block rather than generating an IfNode. An optimization pass could analyze a Cascade dataflow in the

current IR and determine when this optimization is possible by looking at the branches of an IfNode, as in Figure 22.

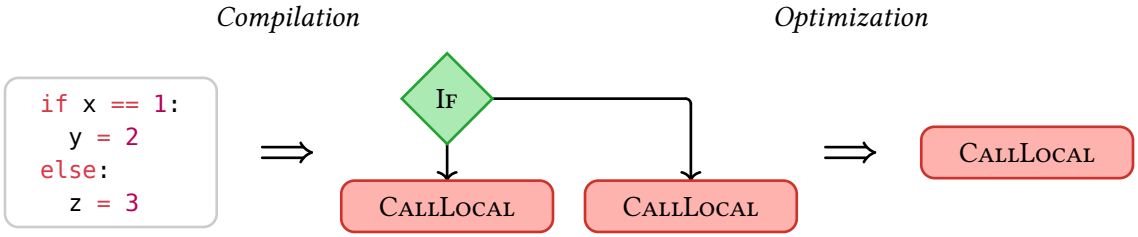


Figure 22: Proposed local `if` block optimization.

8.1.2 IR extensions

Loops. Cascade currently lacks a looping primitive, designed for supporting `for` and `while` loops. This could take inspiration from existing systems such as TensorFlow's `Next` operation, responsible for advancing iterators in loops [2]. We argue that implementing loops is mostly an engineering effort, but that they open the doors to a wide range of additional optimizations, such as loop-invariant code motion and loop unrolling/parallelization [57].

`SELECT` queries. Cascade currently does not support SQL-like `SELECT` queries. Certain applications, like the DeathStar [23] hotel reservation benchmark require this type of functionality. The `/search/` endpoint requires searching through all hotels and returning those within a certain distance from a point. A prototype for such a system was explored by saving existing keys in a dedicated `Indexer` operator, which could then be queried to return all the keys for a certain stateful operator, essentially mimicking a `SELECT key FROM operator` query. However, further work would be required to provide efficient data retrieval operations that are competitive with existing database-based approaches, and ideally would support efficient implementations of common query operations such as range selection, sorting and joins.

To the best of our knowledge, built-in support for secondary indexing in existing dataflow systems is severely limited. This may be partly due to the unstructured nature of state in dataflow systems, as building and maintaining indexes can require significant overhead: [58] finds that unstructured data is a limitation when incorporating efficient indexing into dataflow systems and [59] explores the trade-off of building indexes in the context of monetary cost optimization for dataflow systems in the cloud. We argue that state in Cascade is similar to *NoSQL* systems such as Cassandra [60], MongoDB [61] and DynamoDB [62], that have flexible schemas. MongoDB defaults to B+ trees for indexing, which follows the approach of traditional relational databases. However, the optimal data structure for indexing also depends on the type of query performed, and there are many trade-offs to consider [63]. For example, `WHERE` clauses are more suited to hash-based indexing whereas geospatial indexes may be used for location-based queries [64].

Transactions. Cascade currently lacks support for transactions, which could lead to unexpected consequences, for example when the same state is accessed and modified by two different dataflows. Support for transactional workflows could lock pieces of accessed state in critical sections and prevent such issues. This would serve to isolate concurrent operations from each other and ensure that an event would see a consistent application state. Maintaining these properties is not always simple. Parallel dataflows could easily lead to atomicity violations, and durability guarantees are different depending on the underlying execution target. Orleans [6] is a similar system to Cascade that has implemented ACID transactions, and has addressed some of these issues.

Event generation. Currently, dataflows are executed by constructing the necessary `Event` objects somewhat manually, and sending them to a `CascadeClient` object. Future versions could raise the abstraction level and automatically transform calls such as `user.buy_item(item)` into an Event sent to the `CascadeClient`, through some program analysis. Automatic event generation and sending was already implemented in StateFlow, and Cascade could take the same approach [56].

Bibliography

[1] E. Lumunge, "LLVM - An Overview." Accessed: May 27, 2025. [Online]. Available: https://iq.opengenus.org/llvm-overview/

[2] Y. Yu *et al.*, "Dynamic Control Flow in Large-Scale Machine Learning," in *Proceedings of the 13th EuroSys Conference, EuroSys 2018*, Association for Computing Machinery, Inc, 2018. doi: 10.1145/3190508.3190551.

[3] A. Cheung, N. Crooks, J. M. Hellerstein, and M. Milano, "New directions in cloud programming," *arXiv preprint arXiv:2101.01159*, 2021.

[4] H. B. Hassan, S. A. Barakat, and Q. I. Sarhan, "Survey on serverless computing," *Journal of Cloud Computing*, vol. 10, pp. 1–29, 2021.

[5] Amazon Web Services, Inc., "AWS Lambda." Accessed: May 20, 2025. [Online]. Available: https://aws.amazon.com/lambda/

[6] S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin, "Orleans: cloud computing for everyone," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*, 2011, pp. 1–14.

[7] Stack Overflow, "Stack Overflow Developer Survey 2024." Accessed: May 22, 2024. [Online]. Available: https://survey.stackoverflow.co/2024/technology#1-web-frameworks-and-technologies

[8] V. Sreekanti *et al.*, "Cloudburst: Stateful functions-as-a-service," *arXiv preprint arXiv:2001.04592*, 2020.

[9] F. Yang, J. Shanmugasundaram, M. Riedewald, and J. Gehrke, "Hilda: A high-level language for data-drivenweb applications," in *22nd International Conference on Data Engineering (ICDE'06)*, 2006, p. 32.

[10] S. Burckhardt, C. Gillum, D. Justo, K. Kallas, C. McMahon, and C. S. Meiklejohn, "Durable functions: Semantics for stateful serverless," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–27, 2021.

[11] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *The Bulletin of the Technical Committee on Data Engineering*, vol. 38, no. 4, 2015.

[12] M. Zaharia *et al.*, "Apache spark: a unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[13] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: A timely dataflow system," in *SOSP 2013 - Proceedings of the 24th ACM Symposium on Operating Systems Principles*, 2013, pp. 439–455. doi: 10.1145/2517349.2522738.

[14] K. Psarakis, G. Christodoulou, G. Siachamis, M. Fragkoulis, and A. Katsifodimos, "Styx: Transactional Stateful Functions on Streaming Dataflows," *Proceedings of the ACM on Management of Data*, vol. 3, no. 3, pp. 1–28, 2025.

[15] J. R. Gurd, "The manchester dataflow machine," *Computer Physics Communications*, vol. 37, no. 1–3, pp. 49–62, 1985.

[16] Arvind and R. S. Nikhil, "Executing a program on the MIT tagged-token dataflow architecture," *IEEE Transactions on computers*, vol. 39, no. 3, pp. 300–318, 1990.

[17] M. Beck, R. Johnson, and K. Pingali, "From control flow to dataflow," *Journal of Parallel and Distributed Computing*, vol. 12, no. 2, pp. 118–129, 1991, doi: 10.1016/0743-7315(91)90016-3.

[18] K. J. Ottenstein, R. A. Ballance, and A. B. MacCabe, "The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages,"

in *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, 1990, pp. 257–271.

[19] R. Johnson and K. Pingali, "Dependence-based program analysis," in *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, 1993, pp. 78–89.

[20] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Making state explicit for imperative big data processing," in *2014 USENIX annual technical conference (USENIX ATC 14)*, 2014, pp. 49–60.

[21] G. E. Gévay, T. Rabl, S. Breß, L. Madai-Tahy, and V. Markl, "Labyrinth: Compiling imperative control flow to parallel dataflows," *arXiv preprint arXiv:1809.06845*, 2018.

[22] K. Psarakis, W. Zorgdrager, M. Fragkoulis, G. Salvaneschi, and A. Katsifodimos, "Stateful entities: object-oriented cloud applications as distributed dataflows," *arXiv preprint arXiv:2112.00710*, 2021.

[23] Y. Gan *et al.*, "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems," in *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, Association for Computing Machinery, 2019, pp. 3–18. doi: 10.1145/3297858.3304013.

[24] M. Fragkoulis, P. Carbone, V. Kalavri, and A. Katsifodimos, "A survey on the evolution of stream processing systems," *The VLDB Journal*, vol. 33, no. 2, pp. 507–541, 2024.

[25] A. Davidson and A. Or, "Optimizing shuffle performance in spark," *University of California, Berkeley-Department of Electrical Engineering and Computer Sciences, Tech. Rep*, vol. 10, 2013.

[26] F. Hueske *et al.*, "Opening the Black Boxes in Data Flow Optimization," *Proceedings of the VLDB Endowment*, vol. 5, no. 11, pp. 1256–1267, 2012, doi: 10.14778/2350229.2350244.

[27] J. Kreps, N. Narkhede, and J. Rao, "Kafka: A distributed messaging system for log processing," in *Proceedings of the NetDB*, 2011, pp. 1–7.

[28] P. Nowojsk and M. Winters, "An Overview of End-to-End Exactly-Once Processing in Apache Flink (with Apache Kafka, too!)." Accessed: Apr. 28, 2025. [Online]. Available: https://flink.apache.org/2018/02/28/an-overview-of-end-to-end-exactly-once-processing-in-apache-flink-with-apache-kafka-too/

[29] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International symposium on code generation and optimization, 2004. CGO 2004.*, 2004, pp. 75–86.

[30] X. Huang and D. Fu, "Exploring the Thread Mode in PyFlink." Accessed: May 15, 2025. [Online]. Available: https://flink.apache.org/2022/05/06/exploring-the-thread-mode-in-pyflink/

[31] D. Callahan and K. Kennedy, "Analysis of interprocedural side effects in a parallel programming environment," in *International Conference on Supercomputing*, 1987, pp. 138–171.

[32] S. Herhut, S.-B. Scholz, and C. Grelck, "Controlling chaos: on safe side-effects in data-parallel operations," in *Proceedings of the 4th workshop on Declarative aspects of multicore programming*, 2009, pp. 59–67.

[33] S. Mittal, "A survey of techniques for dynamic branch prediction," *Concurrency and Computation: Practice and Experience*, vol. 31, no. 1, p. e4666, 2019.

[34] R. Joseph, "A survey of deep learning techniques for dynamic branch prediction," *arXiv preprint arXiv:2112.14911*, 2021.

[35] K. Kennedy and J. R. Allen, *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., 2001.

[36] K. Gasmi and S. Hasnaoui, "Dataflow-based automatic parallelization of MATLAB/Simulink models for fitting modern multicore architectures," *Cluster Computing*, vol. 27, no. 5, pp. 6579–6590, 2024.

[37] M. W. Hall, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. S. Lam, "Interprocedural parallelization analysis in SUIF," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 27, no. 4, pp. 662–731, 2005.

[38] J. Gu and Z. Li, "Efficient interprocedural array data-flow analysis for automatic program parallelization," *IEEE Transactions on Software Engineering*, vol. 26, no. 3, pp. 244–261, 2000.

[39] N. Korolija, "Fine grain algorithm parallelization on a hybrid control-flow and dataflow processor," *Journal of Big Data*, vol. 12, no. 1, p. 42, 2025.

[40] T. M. Chilimbi and M. Hirzel, "Dynamic hot data stream prefetching for general-purpose programs," in *Proceedings of the ACM SIGPLAN 2002 Convference on Programming language design and implementation*, 2002, pp. 199–209.

[41] H. Garcia-Molina and K. Salem, "Sagas," *ACM Sigmod Record*, vol. 16, no. 3, pp. 249–259, 1987.

[42] A. Estebanez, D. R. Llanos, and A. Gonzalez-Escribano, "A survey on thread-level speculation techniques," *ACM Computing Surveys (CSUR)*, vol. 49, no. 2, pp. 1–39, 2016.

[43] J. Von Neumann, "First Draft of a Report on the EDVAC," *IEEE Annals of the History of Computing*, vol. 15, no. 4, pp. 27–75, 1993.

[44] J. B. Dennis, "First version of a data flow procedure language," in *Programming Symposium: Proceedings, Colloque sur la Programmation Paris, April 9–11, 1974*, 1974, pp. 362–376.

[45] B. Lee and A. R. Hurson, "Issues in dataflow computing," *Advances in computers*, vol. 37. Elsevier, pp. 285–333, 1993.

[46] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[47] E. Jahani, M. J. Cafarella, and C. Ré, "Automatic optimization for MapReduce programs," *arXiv preprint arXiv:1104.3217*, 2011.

[48] A. Rheinländer, U. Leser, and G. Graefe, "Optimization of Complex Dataflows with User-Defined Functions," *ACM Computing Surveys (CSUR)*, vol. 50, no. 3, 2017, doi: 10.1145/3078752.

[49] A. Alexandrov, G. Krastev, and V. Markl, "Representations and optimizations for embedded parallel dataflow languages," *ACM Transactions on Database Systems*, vol. 44, no. 1, 2019, doi: 10.1145/3281629.

[50] Y. Yu *et al.*, "DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language," 2009.

[51] G. E. Gevay, T. Rabl, S. Bres, L. Madai-Tahy, J. A. Quiane-Ruiz, and V. Markl, "Efficient control flow in dataflow systems: When ease-of-use meets high performance," in *Proceedings - International Conference on Data Engineering*, IEEE Computer Society, 2021, pp. 1428–1439. doi: 10.1109/ICDE51399.2021.00127.

[52] S. Laddad, A. Cheung, J. M. Hellerstein, and M. Milano, "Flo: A Semantic Foundation for Progressive Stream Processing," *Proceedings of the ACM on Programming Languages*, vol. 9, no. POPL, pp. 241–270, 2025.

[53] C. Hewitt, "The challenge of open systems: current logic programming methods may be insufficient for developing the intelligent systems of the future," *Byte*, vol. 10, no. 4, pp. 223–242, 1985.

[54] D. Wyatt, *Akka concurrency*. Artima Incorporation, 2013.

[55] R. Virding, C. Wikström, and M. Williams, *Concurrent programming in ERLANG*. Prentice Hall International (UK) Ltd., 1996.

[56] W. Zorgdrager, "Anyone Can Cloud: Democratizing Cloud Application Programming," 2021.

[57] W. Pugh, "Uniform techniques for loop optimization," in *Proceedings of the 5th international conference on Supercomputing*, 1991, pp. 341–352.

[58] D. Logothetis and K. Yocum, "Data indexing for stateful, large-scale data processing," in *Proceedings of NetDB*, 2009.

[59] H. Kllapi, I. Pietri, V. Kantere, and Y. E. Ioannidis, "Automated Management of Indexes for Dataflow Processing Engines in IaaS Clouds.," in *EDBT*, 2020, pp. 169–180.

[60] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS operating systems review*, vol. 44, no. 2, pp. 35–40, 2010.

[61] S. Bradshaw, E. Brazil, and K. Chodorow, *MongoDB: the definitive guide: powerful and scalable data storage*. " O'Reilly Media, Inc.", 2019.

[62] S. Sivasubramanian, "Amazon dynamoDB: a seamlessly scalable non-relational database service," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012, pp. 729–730.

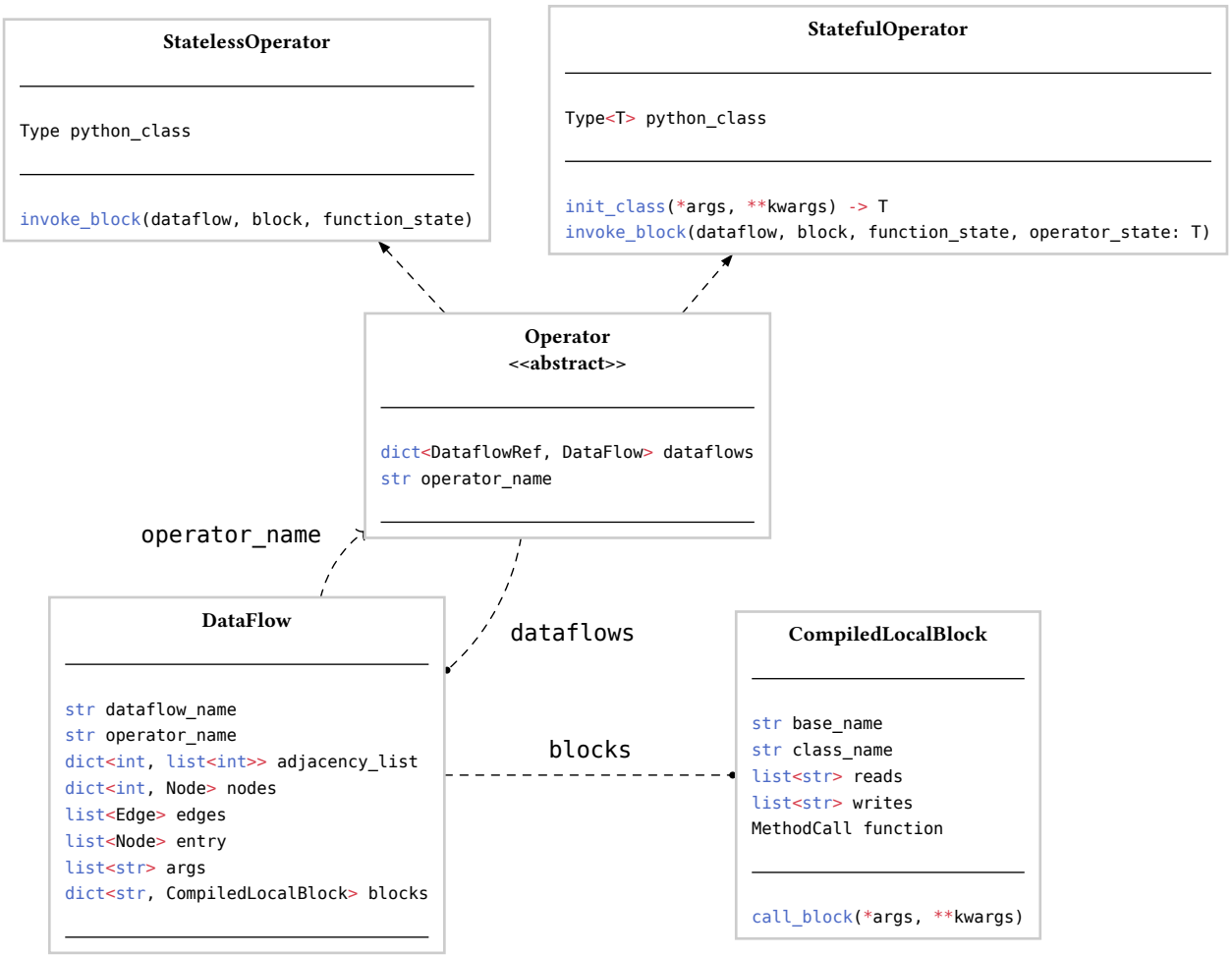
[63] M. A. Qader, S. Cheng, and V. Hristidis, "A comparative study of secondary indexing techniques in LSM-based NoSQL databases," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 551–566.

[64] M. Nuriev, R. Zaripova, O. Yanova, I. Koshkina, and A. Chupaev, "Enhancing MongoDB query performance through index optimization," in *E3S Web of Conferences*, 2024, p. 3022.

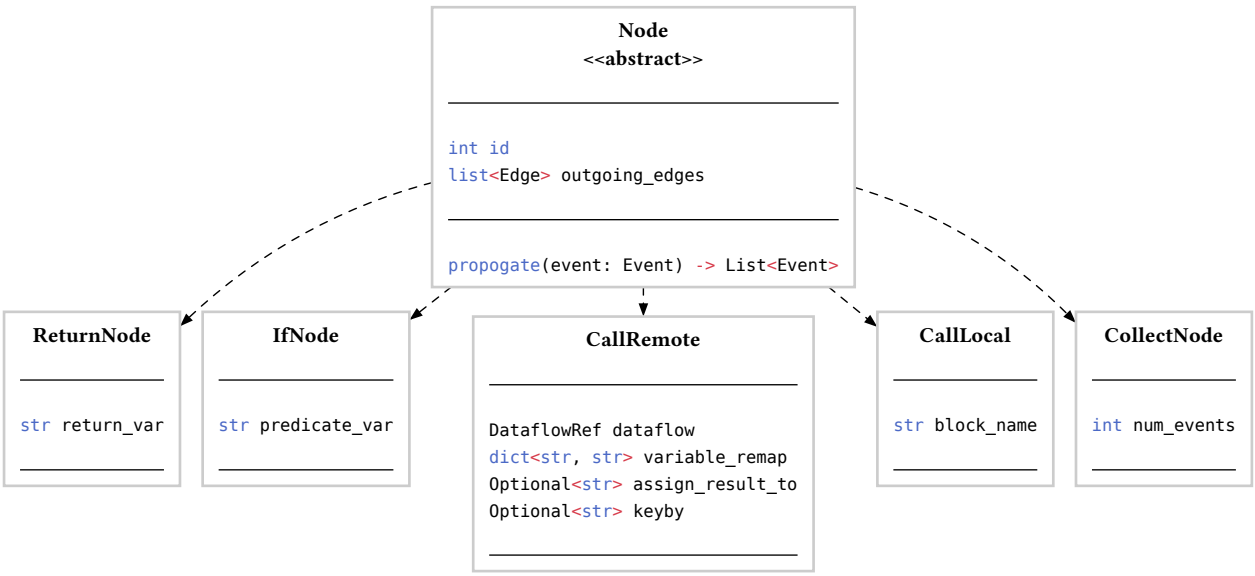
Appendix A

Class Diagrams

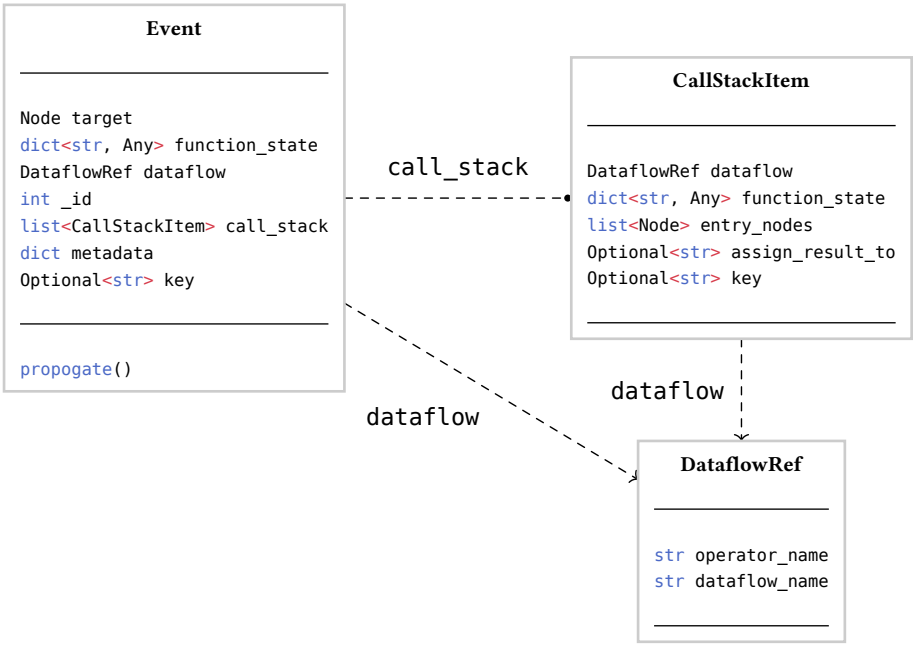
A1 Operators and blocks



A2 Dataflow node types



A3 Events and the call stack



Appendix B

PyFlink Configuration Settings

In order to optimize PyFlink for low-latency, the following settings where used:

```
python.execution-mode: "thread"
python.fn-execution.bundle.time: 5
python.fn-execution.bundle.size: 1
execution.batch-shuffle-mode: "ALL_EXCHANGES_PIPELINED"
execution.buffer-timeout: "0 ms"
```

In addition, internal Kafka sources where given the property:

```
fetch.min.bytes: 1
```

while internal Kafka sinks set:

```
linger.ms: 0
acks: 1
```