Efficient Control Flow in Dataflow Systems: When Ease-of-Use Meets High Performance

Gábor E. Gévay* Tilmann Rabl∓,1 Sebastian Breß◇,1 Loránd Madai-Tahy*
Jorge-Arnulfo Quiané-Ruiz*,⋆ Volker Markl*,⋆

*Technische Universität Berlin (TU Berlin) ∓Hasso Plattner Institute, Uni Potsdam ⋆DFKI, Berlin ◇Snowflake Inc.
{gevay, madai-tahy, jorge.quiane, volker.markl}@tu-berlin.de, tilmann.rabl@hpi.de, sebastian.bress@snowflake.com

***Abstract*—Modern data analysis tasks often involve control flow statements, such as iterations. Common examples are PageRank and K-means. To achieve scalability, developers usually implement data analysis tasks in distributed dataflow systems, such as Spark and Flink. However, for tasks with control flow statements, these systems still either suffer from poor performance or are hard to use. For example, while Flink supports iterations and Spark provides ease-of-use, Flink is hard to use and Spark has poor performance for iterative tasks. As a result, developers typically have to implement different workarounds to run their jobs with control flow statements in an easy and efficient way.**

We propose Mitos, a system that achieves the best of both worlds: it achieves both high performance and ease-of-use. Mitos uses an intermediate representation that abstracts away specific control flow statements and is able to represent any imperative control flow. This facilitates building the dataflow graph and coordinating the distributed execution of control flow in a way that is not tied to specific control flow constructs. Our experimental evaluation shows that the performance of Mitos is more than one order of magnitude better than systems that launch new dataflow jobs for every iteration step. Remarkably, it is also up to 10.5 times faster than Flink, which has native iteration support, while matching the ease-of-use of Spark.

***Index Terms*—Iterative dataflow, Loop pipelining, Loop-invariant hoisting**

I. Introduction

Modern data analytics typically achieve scalability by relying on dataflow systems, such as Spark [1] and Flink [2], [3]. Besides this scalability need, many data analysis algorithms require support for control flow statements. For example, many graph analysis tasks are iterative, such as PageRank [4] and computing connected components [5]. Other data science pipelines are also mainly composed of iterative programs [6]. K-means clustering [6] and gradient descent [7] are just two of the most commonly occurring iterative tasks. Additionally, control flow is just getting more complex: An iterative machine learning training task can be inside another loop for hyperparameter optimization; programs may contain if statements inside loops, such as in simulated annealing [8].

However, despite that control flow statements are at the core of modern data analytics, supporting control flow efficiently and effectively is the biggest weakness of dataflow systems: They either suffer from poor performance or are hard to use. On the one hand, in some systems, such as Spark, users express iterations inside the driver program, using the standard, *imperative* control flow constructs. Although this imperative approach is easy to use, it launches a new dataflow job for every iteration step, which hurts performance because of a high inherent job-launch overhead. On the other hand, some other systems, such as Flink, provide *native control flow* support [9], i.e., users can include iterations in their (cyclic) dataflow jobs. This removes the job-launch overhead, which is present in Spark, resulting in much better performance. However, this high performance comes at a price: Users have to express iterations by calling higher-order functions, which are harder to use than the imperative control flow of Spark.

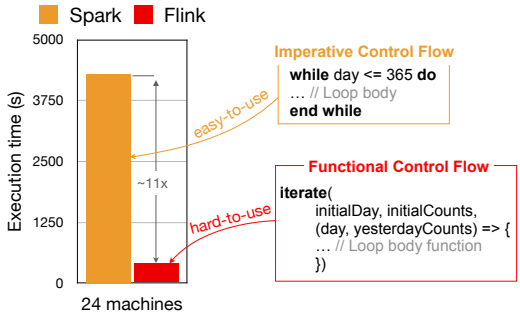


Fig. 1: Imperative vs. functional control flow.

To better illustrate this problem, we ran an experiment to evaluate Spark and Flink, using a program that computes the visit counts from a year of page visit logs. This program has a loop that reads a different file at each iteration step and compares the visit counts with the previous day[2]. Figure 1 shows the results of this experiment. We observe that Spark is more than an order of magnitude slower than Flink because it does not support native iterations. Spark launches a new dataflow job for every iteration step, incurring a high overhead. However, on the other side, Flink is harder to use than Spark. In Flink users call the *iterate* higher-order function and give the loop body as an argument (see the functional control flow box in Figure 1). The loop body is a function that builds the dataflow job fragment representing the actual loop body operations. This API is clearly not easy at all for non-expert users, such as data scientists[3]. In contrast, users prefer the imperative control flow present in Spark, similar to, e.g., Python, R, or Matlab (see

[1]Work done while the author was at TU Berlin.

[2]We provide the details of this experiment in Section VI and provide the code for both the imperative and functional control flow APIs in the Appendix.

[3]A simple search on stackoverflow.com for the terms *Flink iterate* or *TensorFlow while_loop* shows that a large number of users are indeed confused by such a functional control flow API.

the imperative control flow box in Figure 1).

Ideally, the system should allow users to express control flow using simple imperative control flow statements, while matching the performance of native control flow. In other words, we want a system that marries the ease-of-use of Spark with the high efficiency of Flink. The research community has paid attention to this problem and recently proposed a number of solutions [10], [11], [12]. For example, Emma [10] can translate imperative control flow to Flink's native iterations, but only when there is a single while-loop without any other control flow statement in its body. This makes it not suitable for many tasks in modern data analytics, such as hyper-parameter optimization, simulated annealing, and strongly connected components [13]. AutoGraph [11] and Janus [12] compile imperative control flow to TensorFlow's native iterations [14]. However, they do not support general data analytics other than machine learning.

Supporting general imperative control flow (e.g., iterative tasks) without sacrificing performance is challenging for two main reasons. First, normally a dataflow job is built from just the method calls (e.g., *map*, *join*) that the user program makes to the system. However, to build a complete cyclic dataflow job from imperative control flow, the system also needs to inspect other parts of the user code, such as the control flow statements: It also has to insert special nodes and edges into the dataflow job for such parts of the code. Second, to fully take advantage of the entire program being in a single dataflow job, we want to support loop pipelining, i.e., overlapping subsequent executions of a loop body. This means that we cannot simply insert a full synchronization barrier between iteration steps, and just reset all operators at the barrier. Instead, we need to deal with different operators (and their different physical instances) processing different iteration steps at the same time.

We propose Mitos[4], a system where control flow support matches Spark's ease-of-use, and that significantly outperforms both Spark and Flink. Specifically, it outperforms Spark because of native iterations, and it outperforms Flink's native iterations because of loop pipelining. Mitos uses compile-time metaprogramming to parse an imperative user program into an intermediate representation (IR) that abstracts away specific control flow constructs. This IR facilitates the building of a single (cyclic) dataflow job from any program with imperative control flow. At runtime, Mitos coordinates the distributed execution of control flow using a novel coordination algorithm that leverages our IR to handle any general imperative control flow. In summary, we make three major contributions:

(1) We propose a compilation approach based on metaprogramming to build a single dataflow job of a distributed dataflow system from a program with general imperative control flow statements. Specifically, we leverage Scala macros [15] to inspect and rewrite the user program's abstract syntax tree such that the system can produce a single dataflow job. By this, we can bring the power of native control flow to data scientists, who like to use high-level languages that have imperative control flow statements. (Section IV)

(2) We devise a mechanism that coordinates and communicates the control flow decisions between machines in a non-intrusive manner. In particular, our coordination mechanism enables two core optimizations that speed up the dataflow job execution: loop pipelining, i.e., overlapping iteration steps, and loop-invariant hoisting, i.e., reusing loop-invariant (static) datasets during subsequent iteration steps. As a result, our system not only supports any control flow statement but also outperforms dataflow systems with native control flow support. (Section V)

(3) We experimentally evaluate Mitos using real tasks (Visit Count and PageRank) and microbenchmarks. We mainly compare its performance to Flink (as a system supporting native control flow), Spark (as a system providing ease-of-use). Our results show the superiority of Mitos over all baselines: It is more than one order of magnitude faster than Spark, and, surprisingly, it is also up to $10.5\times$ faster than Flink (the system with native control flow support). (Section VI)

II. Motivating Example

We now illustrate through an example the problems of current dataflow systems when faced with imperative control flow.

Consider a program that computes the visit counts for each page per day in a year of page visit logs. Assume that the log of each day is read from a separate file and that each log entry is a page ID, which means that someone has visited the page.

```
1: for day = 1 .. 365 do
2:    visits = readFile("PageVisitLog_" + day) // page IDs
3:    counts = visits.map(x => (x,1)).reduceByKey(_ + _)
4:    counts.writeFile("Counts_" + day)
5: end for
```

We cannot express this simple program in Flink's native iterations, because Flink does not support reading and writing files inside native iterations. However, not using native iterations would cause each iteration step to launch a new dataflow job, which has an inherent high overhead[5] (see Spark in Figure 1).

This simple program can easily become more complicated. Imagine that instead of just writing out the visit counts for each day separately, we want to compare the visit counts of consecutive days. For this, we replace Line 4 with the following:

```
4: if day != 1 then
5:    diffs =
6:        (counts join yesterdayCounts)
7:        .map((id,today,yesterday) => abs(today - yesterday))
8:    diffs.sum.writeFile("diff" + day)
9: end if
10: yesterdayCounts = counts
```

If it is not the first day, we join the current counts with the previous day's counts (Line 6). We then compute pairwise differences (Line 7), sum up the differences (Line 8), and write the sum to a file. At the end, we save the current counts so that we can use them the next day (Line 10). We can see that it is

[4]The name comes from Greek mythology: Mitos is the thread that Ariadne gave to Theseus to help him get out of the labyrinth.

[5]Note that a new job is not launched if there is no *action* inside the loop body. However, actions are needed in most iterative algorithms to compute a loop exit condition from the current state of the algorithm. Moreover, Spark's job-launch overhead is mostly the task launch overhead, which will still be present at each iteration step even without actions (see Section VI-E).

natural to use an if statement inside the loop. On top of that, we could replace the computation of visit counts (Line 3) with a more complex computation that itself involves a loop, such as PageRank [4]. This would result in having nested loops. Unfortunately, Flink does not provide native support for either nested loops or if statements inside loops. On the other side, Spark does not have native support for any control flow at all.

Yet, this program can become even more complex. Imagine we are interested only in a certain page type. As the logs do not contain information about the page type (each log line is just a page ID), we have to read a dataset containing the types of all pages before the loop. Inside the loop, we then add the line below before Line 3, which performs a join between the visits and page type datasets, and filters based on page type:

```
3: visits = (visits join pageTypes).filter(p => p.type=...)
```

It is worth noting that the *pageTypes* dataset does not change between iteration steps, i.e., it is *loop-invariant*. This clearly opens an opportunity for optimization: Even though the join method is called inside the loop, we can build the hash table of the join only once before the loop and probe it at every iteration step. This is only possible if the system implements the loop as a native iteration. This is because all iteration steps are in a single dataflow job, which enables the join operator to keep the hash table throughout the entire loop. Nevertheless, we cannot express this program using Flink's native iterations because of the aforementioned issues.

Note that iterations are at the core of machine learning training algorithms and hyperparameter search. This makes Mitos an important piece in modern analytics, such as the ones targeted by Agora [16].

III. Mitos Overview

We present Mitos, a system that compiles a data analysis program with imperative control flow statements into a *single* dataflow job for distributed execution on a dataflow system. The main goal of Mitos is to bring ease-of-use to users while achieving high efficiency for their programs. Overall, users write their programs using imperative control flow. The system, in turn, parses an *imperative program* into an intermediate representation, from which it builds a single (cyclic) dataflow job. At runtime, the system coordinates the distributed execution of control flow statements among workers in the underlying dataflow system. Below, we describe these steps in more detail.

Figure 2 illustrates the general architecture of Mitos. A user provides a data analysis program in a high-level language with imperative control flow support. We use the Emma language [17], [10] because of its metaprogramming infrastructure and because it is similar to the languages of typical dataflow systems, such as Flink and Spark: The user expresses a data analysis program in Scala using a scalable collection type, which we call *bag* henceforth. Given an imperative program, Mitos first simplifies it to make each assignment statement have only a single bag operation (e.g., a *map*). It then parses this simplified imperative program to an intermediate representation (IR). From there, it creates a dataflow job of a distributed dataflow system (Section IV). Recall that running

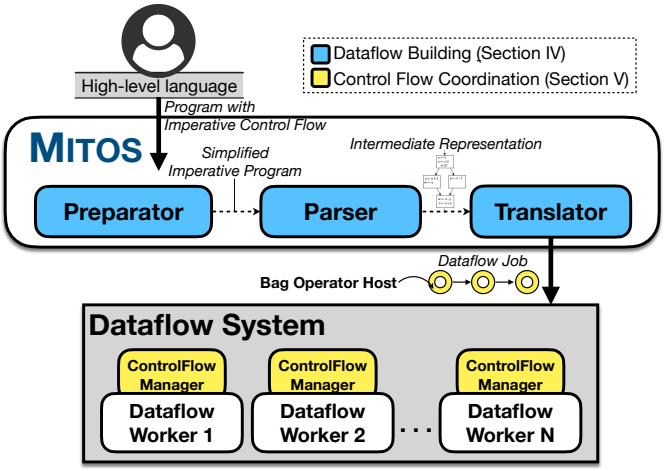


Fig. 2: Mitos architecture.

many dataflow jobs sequentially significantly deteriorates the execution time of a program as illustrated in Figure 1 (the Spark case). Thus, it is crucial to generate as few dataflow jobs as possible: Mitos creates a single job for the entire program.

Next, the system sends the job for execution on the underlying dataflow system. Then, Mitos coordinates the distributed execution of control flow via two components: the *Control Flow Manager* and the *Bag Operator Host* (Section V). The control flow manager communicates control flow decisions among the worker machines. The bag operator host bridges the gap between Mitos' and the underlying dataflow system's operators. While Mitos' operators take input bags and compute output bags, the underlying dataflow system's operators do not know about bags. The bag operator host provides an interface for implementing Mitos' operators at the level of bags instead of directly with the dataflow system's operator interface. Note that our control flow coordination enables loop pipelining, i.e., overlapping different iteration steps.

Generality for Backends. Although we use Flink as our target dataflow system, Mitos is designed to be as general as possible, i.e., not closely tied to a specific dataflow system. It only requires a dataflow system that allows for arbitrary stateful computations in the dataflow vertices, and supports arbitrary cycles in the dataflow graph. Examples of systems that support cycles are Flink, Naiad [18], Dandelion [19], and TensorFlow. Note that, for Mitos' loop pipelining to have a significant effect, the system should support pipelined data transfers. It is also possible to integrate Mitos into Rheem [20], [21], [22] (now Apache Wayang), to run over multiple dataflow systems.

Generality for Languages. Although we use the Emma language [17], [10] for Mitos, one could use other high-level data analytics languages that have imperative control flow support. Importantly, the language should provide the system with means to get information about the imperative control flow statements. In the case of Emma, this is achieved by compile-time metaprogramming. Specifically, we use Scala macros [15]. Julia [23] and Python also have the required metaprogramming capabilities. Alternatively, SystemML [24] could also be integrated with Mitos. SystemML's language is an *external* [10] domain-specific language, and thereby

SystemML's compiler can naturally inspect the control flow.

Background (Compiler Concepts). We rely on a couple of basic compiler concepts: *static single assignment form* (SSA) and *basic blocks*. SSA [25] is often used in compilers to represent imperative control flow. When a program is in SSA form, each variable has exactly one assignment statement to it. Another important characteristic of SSA is that it abstracts away from specific control flow constructs: The program is divided into so-called basic blocks. A basic block is a contiguous sequence of instructions with no control flow instructions, except at the end, where they conditionally jump to the beginning of the same (or another) basic block. For example, consider a loop body consisting of a single basic block. The last instruction jumps either back to the beginning of the loop body block or to the basic block that is after the loop, depending on the loop exit condition. We later provide more details of SSA and other compiler concepts where necessary.

IV. Building Dataflows from Imperative Programs

Our goal is to produce a *single* dataflow job from a user's imperative program that has arbitrary imperative control flow constructs. Doing so is far from being a trivial task. We need to inspect control flow statements and add extra edges. For example, in iterative algorithms, there is typically a dataflow node near the end of the loop body whose output has to be fed into the next iteration step. A more specific example is passing the current PageRanks from one step to the next. Additionally, we need to include non-bag variables into our dataflow jobs.

We leverage compile-time metaprogramming to overcome the above-mentioned challenges and hence create a dataflow job containing all the operations of an imperative program. Specifically, we leverage Scala macros [15] to inspect and rewrite the user program's abstract syntax tree. In more detail, we first simplify the imperative program (Section IV-A), and then parse it into an intermediate representation (Section IV-B). Both of these facilitate the translation of the user's program into a single dataflow job (Section IV-C).

A. Simplifying an Imperative Program

As a first step, we split those assignment statements that have more than one operation on their right-hand side. For example, we split $b = a.map(...).filter(...)$ into two assignments: $tmp = a.map(...)$; $b = tmp.filter(...)$. For instance, Lines 8 & 9 in Figure 3a are the splitted version of Line 3 in Section II.

Next, we take care of non-bag variables, e.g., an *Integer* loop counter or a *Double* learning rate. We wrap all these variables into one-element bags. This normalization step simplifies later dataflow-building by ensuring that it needs to deal with only bag operations instead of introducing special cases for non-bag variables. More specifically, we perform the following transformations: any operation that creates a non-bag value is substituted with an equivalent operation that puts the same value inside a one-element bag (e.g., creating a constant, such as $a = 1$ becomes $a = newBag(1)$); a unary function f that acts on a non-bag value is substituted with a *map* operator, whose user-defined function (UDF) is f (e.g., $b = -a$ is substituted by $b = a.map(x => -x)$); a binary function that acts on two non-bag values is substituted by a cross product and a *map*. The cross product creates a one-element bag that contains a pair with the elements of the two input bags. The *map* operates on this pair and has f as its UDF (e.g., $c = a + b$ is substituted by $c = (a \ cross \ b).map(\_ + \_)$). Note that we can apply further simplifications in some cases. For example, $b = a + 1$ can be transformed into $b = a.map(x => x + 1)$ instead of $tmp = newBag(1); b = a.cross(tmp).map((x, y) => x + y)$.

B. Intermediate Representation for General Control Flow

To handle all imperative control flow statements uniformly, Mitos transforms the program into an IR that is based on SSA [25]. As part of this transformation, Mitos introduces a different variable for each assignment statement: if a variable in the original program had more than one assignment statement, we rename the left-hand sides of all these assignments to unique names. At the same time, we update all references to these variables with the new names. However, this updating step is not directly possible if there are different control flow paths that assign different values to a variable. In this case, the different assignments in the different control flow paths are renamed to different names and hence there is no single name to change a reference into. For example:

```
1: if ... then
2:    a = ...
3: else
4:    a = ...
5: end if
6: b = a.map(...)
```

Note that after we change the left-hand sides of the assignments in Line 2 and 4 to different names, we cannot simply change the variable reference in Line 6 to just one of them at compile time. Therefore, we have to choose the value to refer to at runtime, based on the actual control flow path that the program execution takes. SSA solves this problem by introducing Φ-functions, which make this runtime choice explicit (Line 6):

```
1: if ... then
2:    a1 = ...
3: else
4:    a2 = ...
5: end if
6: a3 = Φ(a1,a2)
7: b = a3.map(...)
```

We explain how Mitos tracks the control flow and thus how Φ-functions choose between their inputs at runtime in Section V.

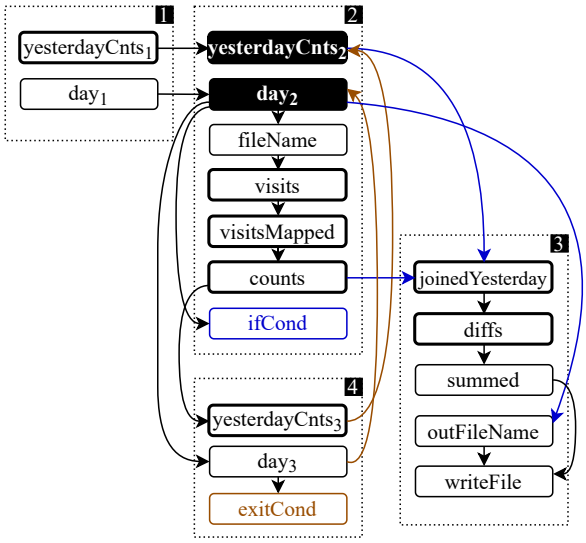
By relying on SSA, we abstract away from specific control flow constructs, and thus handle all control flow uniformly: Control flow constructs are translated into basic blocks and conditional jumps at the end of basic blocks.

C. Translating an Imperative Program to a Single Dataflow

After simplifying an imperative program and putting it into our intermediate representation, the final step to build a dataflow job is now simple: We create a single dataflow node from each assignment statement and a single dataflow edge from each variable reference. For example, from $c = a \ join \ b$, we create

```
1:  yesterdayCnts1 = EmptyBag                                   [1]
2:  day1 = newBag(1)
3:  do
4:    yesterdayCnts2 = Φ(yesterdayCnts1,yesterdayCnts3)         [2]
5:    day2 = Φ(day1,day3)
6:    fileName = day2.map(x => "pageVisitLog" + x)
7:    visits = readFile(fileName)
8:    visitsMapped = visits.map(x => (x,1))
9:    counts = visitsMapped.reduceByKey(_ + _)
10:   ifCond = day2.map(x => x != 1)
11:   if ifCond then
12:     joinedYesterday = counts join yesterdayCnts2            [3]
13:     diffs = joinedYesterday.map(...)
14:     summed = diffs.reduce(_ + _)
15:     outFileName = day2.map(x => "diff" + x)
16:     summed.writeFile(outFileName)
17:   end if
18:   yesterdayCnts3 = counts                                   [4]
19:   day3 = day2.map(x => x + 1)
20:   exitCond = day3.map(x => x ≤ 365)
21: while exitCond
```

(a)



(b)

Fig. 3: (a) SSA representation of Visit Count and (b) its Mitos dataflow: The basic blocks are marked with dotted rectangles; The small rectangles are dataflow nodes, corresponding to variables in SSA; The variables corresponding to the thick-bordered nodes are bags; The colored nodes make control flow decisions and influence the same-colored edges.

a *join* node, whose two input edges come from the nodes of the a and b variables.

To better illustrate this final translation step, we use our Visit Count running example program (Section II). Figure 3a shows the program's intermediate representation, with the basic blocks as dotted rectangles, and Figure 3b shows the corresponding Mitos dataflow. Note that the join with the page types is not included for simplicity. As explained in Section IV-A, we wrap non-bag variables in one-element bags. We show the extra code for this in *italic* in Figure 3a. The corresponding nodes in Figure 3b have thin borders. We also create the nodes with the black background from assignments whose right-hand sides are Φ-functions (Lines 4–5). Unlike other nodes, the origins of their inputs depend on the execution path that the program has taken so far: In the first iteration step, they get their values from outside the loop (Lines 1 & 2), but then from the previous iteration step (Lines 18 & 19). This choice is represented by Φ-functions of the SSA form. The blue node corresponds to the *ifCond* variable (Line 10), and the brown node to the loop exit condition (Line 20). These *condition nodes* determine the control flow path. Edges with corresponding colors are *conditional edges*. A condition node determines whether a conditional edge with the same color transmits data in a certain iteration step, as we explain in the following section.

V. Control Flow Coordination

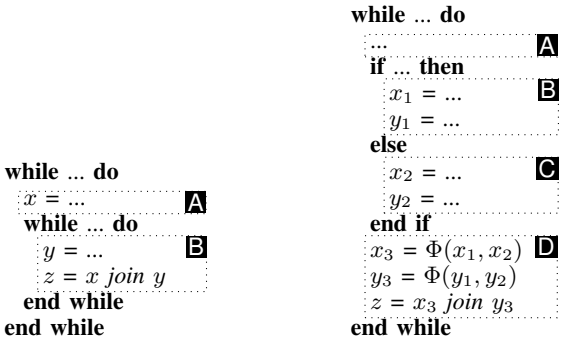
Once a job is submitted for execution in an underlying dataflow system, Mitos has to coordinate the distributed execution of control flow constructs. It communicates control flow decisions between worker machines, gives appropriate input bags to operators for processing, and handles conditional edges. One of the difficulties in doing so is that we must do it in a non-intrusive manner, i.e., with minimal changes or additions inside the underlying dataflow system. This allows Mitos to be as general as possible. We achieve this via two components: the *control flow manager* and the *bag operator host*. The control flow manager communicates control flow decisions among machines. Thus, there is one instance per machine. Next, each operator is wrapped inside a bag operator host, which implements the coordination logic from the operators' side. We refer to these two components together as the *Mitos runtime* (runtime, for short), and we detail them in the following.

Before diving into the runtime, we first give some required preliminaries. We will use the terms "logical" and "physical" to refer to parallelization: A dataflow system parallelizes a dataflow graph (job) by creating multiple *physical* instances of each *logical* operator. A *logical edge* between two logical operators is also multiplied into *physical edges*. Note that if an operator requires a shuffle (e.g., joins), then one physical instance of the operator has p physical input edges corresponding to one logical input edge, where p is the degree of parallelization.

A. Challenges for the Runtime

Devising an algorithm for coordinating the distributed execution of control flow is challenging for three main reasons:

Challenge 1. Input elements from different bags can get mixed. Mitos aims at pipelining loop execution for efficiency reasons. This means that different iteration steps can potentially overlap. That is, different operators or different physical instances of the same operator may be processing different bags that belong to different iteration steps. An example is the Visit Count program's file reading: When any instance of the file-reading operator is done reading the file of the current iteration step, the instance can start working on the file that belongs to the next step. The difficulty is that the output from these different instances get mixed when the next operator is connected by a shuffle. This is because in case of a shuffle, each instance of the next operator receives input from all instances



Listing 1: Programs with non-trivial control flow structures.

of the previous operator. This means that the runtime has to separate input elements that belong to different steps, so that appropriate inputs are used for computing an output bag.

Challenge 2. The matching of input bags of binary operators is not always one-to-one. In the case of binary operators (e.g., join), the runtime gives a pair of bags to an operator at a time. To form a pair, we have to match bags arriving on one logical input edge to bags arriving on the other logical input edge. This matching is not always one-to-one, e.g., sometimes one bag has to be used several times, each time matching it with a different bag. The example program in Listing 1a demonstrates such a case. Input x of the join is from outside the loop, while input y is from inside the loop. This means that when the runtime provides the join with pairs of input bags, it has to use a bag from x several times, matching it with different bags from y each time.

Challenge 3. First-come-first-served does not work for choosing the input bags to process. Even when the matching of bags between the two logical input edges is one-to-one, the following naive algorithm for matching them up does not work: Assume we order bags in the same order as their first elements arrive. In this case, we could match bags from each of the inputs in the order they arrived, i.e., match the first bag from one input with the first bag from the other input, then match the second bags from both inputs, and so on. However, doing so might lead to errors. Suppose that the control flow in Listing 1b reaches the basic blocks in the following order: $ABDACD$. It is then possible that, due to irregular processing delays, the operator of x_3 gets data from x_1 first and then from x_2, while the operator of y_3 gets data from y_2 first and then from y_1. This can happen because the operators in the different *if* branches are not synchronized, i.e., they do not agree on a global order in which to process bags. This would clearly lead to an incorrect result: The operator of z has to match the bag that originates from x_1 with the bag that originates from y_1, and match the bag that originates from x_2 with the bag that originates from y_2. Note that this issue can arise only if we perform loop pipelining. Otherwise, all operators finish the processing of one step before any operator starts the next step.

B. Coordination Based on Bag Identifiers

We tackle the aforementioned challenges by introducing a *bag identifier* (Section V-B1). We make sure that the same bags and same bag identifiers are created during the distributed execution as they would be in a non-parallel execution. More specifically, we show how a physical operator instance can determine during a distributed execution: (i) the identifier of the output bag that it should compute next (Section V-B2); (ii) the identifier of the input bags that it should use to compute a particular output bag (Section V-B3), and; (iii) on which conditional output edge it should send a particular output bag (Section V-B4). Note that the Mitos runtime is designed for allowing operators to start computing an output bag as soon as its inputs start to arrive. The runtime achieves loop pipelining via this feature, i.e., an operator can start a later step while some other operators are still working on a previous step.

1) Bag Identifiers with Execution Paths: A bag identifier encapsulates both the identifier of the logical operator that created the bag and the execution path of the program up to the creation of the bag. The execution path is a sequence of basic blocks that the execution reached. In a distributed execution, the execution path is determined by the condition nodes. A condition node appends a basic block to the path when it evaluates its condition. Condition nodes let all other operators know about these decisions through the control flow manager. The local control flow manager broadcasts the decision to all remote control flow managers through TCP connections (which are independent from dataflow edges). This way every physical instance of every operator knows how the execution path evolves. The bag identifiers are also used to separate elements that belong to different bags (Challenge 1): we tag each element with the bag identifier that it belongs to.

2) Choosing Output Bags: By watching how the execution path evolves, operators can choose the identifiers of output bags to be computed: When the path reaches the basic block of the operator, the operator starts to compute the bag whose bag identifier contains the current path. For example, in Challenge 3, this means that the physical operator instances of both x_3 and y_3 choose to compute the output bag with path ABD in its identifier first, and then $ABDACD$.

3) Choosing Input Bags: When an operator O_2 decides to produce a particular output bag g_2 next, it also needs to choose input bags for it (Challenges 2 & 3). This choice is made independently for each logical input.

In a non-parallel execution, the operator would use the latest bag that was written to the variable that the particular input refers to. We mirror this behavior in the distributed execution, by examining the execution path while keeping in mind the operator's and input's basic blocks. More specifically, for a logical input i of O_2, let O_1 be the operator whose output is connected to i, b_1 and b_2 be the basic blocks of O_1 and O_2, and c be the execution path in the identifier of g_2. To determine the identifier of a bag coming from i to compute an output bag g_2, we consider all the prefixes of c. Among these prefixes, we choose the longest one such that it ends with b_1. For example, in Listing 1a when we are computing z and choosing an input bag from x, we always choose the bag that the latest run of the outer loop computed. Concretely, if we are computing the bag with the path $ABBABBB$, then the prefix we choose is $ABBA$.

Recall that Φ-nodes need to choose between their inputs at each run. We, thus, specially treat Φ-nodes: For each particular output bag, a Φ-node reads a bag from only one input. Therefore, we adapt the above procedure to choose between the inputs by looking at the above-mentioned prefixes for each input, and choosing the longer one.

It is worth noting that in some cases we need to materialize input bags. This happens in two cases: First, when an arriving input bag is not the bag that is currently being processed; Second, when the operator might need the same input bag later (for example, see Challenge 2 in Section V-A). In both of these cases, the bag operator host saves the arriving input elements and provides them (possibly multiple times) to the bag operator at an appropriate time. Note that Mitos saves the elements in a serialized form to reduce the pressure on the Java garbage collector. It discards such saved input bags when they are not needed anymore. This happens when the execution path reaches a block b_3, such that b_1 dominates[6] b_2 from b_3. This is because in that case, the variable of O_1 will necessarily have a new value before O_2 would want to read it.

4) Choosing Conditional Outputs: Operators look at how the execution path evolves after a particular output bag and send the bag on such conditional output edges whose target is reached by the path before the next output bag is computed. Specifically, let O_1 be an operator that is computing output bag g, e be a conditional output edge of O_1, O_2 be the operator that is the target of e, b_1 be the basic block of O_1, b_2 be the basic block of O_2, and c be the execution path of the identifier of g. Note that the last element of c is b_1. O_1 should examine each new basic block appended to the execution path and send g to O_2 when the path reaches b_2 for the first time after c but before it reaches b_1 again. This means that instances of O_1 can discard their partitions of g once the execution path reaches such a basic block from which every path to b_2 on the control flow graph goes through b_1. If O_2 is a Φ-function, then we also need to consider the basic blocks of the other O_2's inputs.

C. Bag Operator Host

To separate the above coordination logic from the semantics of operators (i.e., performing a join, aggregation, etc.), we introduced the *bag operator host*. This provides a standard, push-based interface for implementing the logic of bag operators: First, the operator's *open* method is called by the bag operator host so that the operator can initialize its state; Then, the operator is given input elements by *pushInElement* method calls; Finally, the operator is *close*d by the bag operator host, at which point it can emit its final output, e.g., all the results of a per-group aggregation. In other words, each bag operator instance is wrapped by a bag operator host, which performs the coordination logic described in the previous subsection on behalf of the bag operator. It provides the bag operator with appropriate input bags, separates input elements belonging to different input bags, and so forth.

[6]On the control flow graph, a node d is said to *dominate* [25] a node n from node s, when all paths from s to n go through d. The *control flow graph's* [26] nodes are the basic blocks and its edges are the possible control flow transitions between the blocks.

D. Optimization: Loop-Invariant Hoisting

We now show how to incorporate loop-invariant hoisting into our dataflows. That is, we show how to improve performance when an iteration involves a loop-invariant (static) dataset, which is reused without updates during subsequent iteration steps. We can see an example of this in our running example in Section II: The *pageTypes* dataset is read from a file outside the iteration and is used in a join inside the iteration. Another example is any iterative graph algorithm that performs a join with a static dataset containing the edges of the graph.

It is a common optimization to pull those parts of a loop body that depend on only static datasets outside of the loop, and thus execute them only once [9], [27], [28]. However, launching new dataflow jobs for every iteration step prevents this optimization in the case of binary operators where only one input is static. For example, if a static dataset is used as the build-side of a hash join, then the system should not rebuild the hash table at every iteration step. Mitos operators can keep such a hash table in their internal states among iteration steps. We make this possible by having a single cyclic dataflow job, where the lifetime of operators spans all the steps.

We now show how to incorporate this optimization into Mitos. Normally, the bag operators drop the state that they have built up during the computation of a specific output bag. However, to perform loop-invariant hoisting, the runtime lets the bag operators know when to keep their state that they build up for an input (e.g., the hash table of a hash join). Assume, without loss of generality, that the first input of the bag operator is the one that does not always change between output bags, and the second input changes for every output bag. Between two output bags, the runtime tells the operator whether the next bag coming from the first input changes for the next output bag. If it changes, the operator should drop the state built-up for the first input. Otherwise, the operator implementation should assume that the first input is the same bag as before. For our example in Listing 1a, the first input bag changes at every step of the outer loop, but not between steps of the inner loop.

E. Fault Tolerance

Mitos comes with its own fault-tolerance mechanism as it cannot directly use Flink's Asynchronous Barrier Snapshotting algorithm [29]. This is because the communication among control flow managers happens independently of the dataflow edges that Flink knows about. Mitos provides a *snapshotting* mechanism that is tied to basic blocks in the execution path. A snapshot contains the values of all the variables of a program at a certain point in the execution path, e.g. after every 10th basic block. In detail, Mitos takes snapshots as follows. First, it designates one control flow manager to be the *coordinator*. The coordinator selects the points in the execution path where snapshots should be taken and broadcasts these decisions. Each operator can then individually determine when it reaches such a snapshot point and write its latest output bag into the appropriate snapshot. Once it is done, it sends a *'done'*

message to the coordinator. When the coordinator receives all the *'done'* messages, it writes its state (the execution path) into the snapshot and marks the snapshot as complete. Note that this is an asynchronous algorithm, because different operators can reach a certain snapshot point at different wall-clock times. To restore from a snapshot, all operators read their bags from the snapshot and send these on their output edges. Additionally, the control flow managers read the execution path and tell it to the operators. Normal execution then resumes.

F. Integration with the Underlying Dataflow System

We rely on Flink's streaming API because it allows us to add any arbitrary cycle to the dataflow graph. Note that we do not use any other streaming-specific features. As mentioned before, we aimed for minimal changes in Flink, so that Mitos is as general as possible to be able to sit on top of any dataflow system. We made only one non-superficial change in Flink to enable operators to flush output network buffers at will, which is needed at the end of output bags.

VI. Evaluation

We implemented Mitos on Java 8 and Scala 2.11 and used Flink 1.6 as an underlying dataflow system. We evaluate Mitos with six main questions in mind: (i) How well does Mitos perform vis-a-vis state-of-the-art systems? (Section VI-B) (ii) Can one efficiently bring the ease-of-use of Spark to Flink without Mitos? (Section VI-C) (iii) How well does Mitos scale with respect to the input dataset size? (Section VI-D) (iv) What is the iteration step overhead of Mitos? (Section VI-E) (v) How effective is Mitos' loop-invariant hoisting optimization? (Section VI-F) and (vi) What is the performance impact of the loop pipelining feature of Mitos? (Section VI-G)

A. Setup

Hardware. We ran our experiments on a cluster of 26 machines, each with 2×8-core AMD Opteron 6128 CPUs, 32 GB memory, 4×1 TB disks, a 1 Gb network card, and Ubuntu Linux 18.04.
Tasks and Datasets. We used the Visit Count example introduced in Section II, where we compare visit counts of subsequent days. We used two versions: one with and one without the join of the *pageTypes* dataset. We also used the per-day PageRank task, i.e., we inserted PageRank into the Visit Count example in place of the *reduceByKey* in Line 3. This resulted in nested loops, as explained in Section II. For Visit Count, we have generated random inputs, with the visits uniformly distributed. The page types filter's selectivity is 0.5. For PageRank, we took a real graph[7] [30], and randomly sampled its edges for each day. We have also performed microbenchmarks to isolate the iteration step overhead.
Baselines. We performed most of our experiments against Spark 3.0 and Flink 1.6, with both running on OpenJDK 8. We stored input data on HDFS 2.7.1. We also performed microbenchmarks against Naiad [18] and TensorFlow [14].
Repeatability. We report numbers for the average of three runs. We also provide the code for Mitos[8].

[7]http://law.di.unimi.it/webdata/webbase-2001/
[8]https://github.com/ggevay/mitos

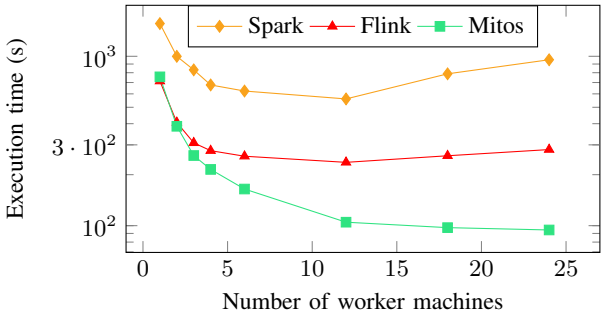


Fig. 4: Strong scaling for Visit Count.

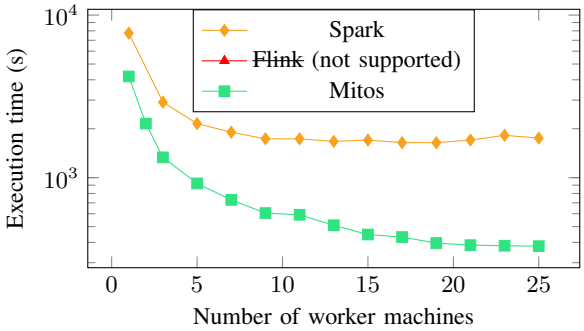


Fig. 5: Strong scaling for per-day PageRank.

B. Strong Scaling

We start by evaluating how well Mitos scales with respect to the number of worker machines as well as how well it performs vis-a-vis the two state-of-the-art systems: Spark and Flink.

1) Visit Count: Figure 4 shows the results for the Visit Count task. The size of the input for one day is 21 MB, and there are 365 days, i.e., the total input size is 7.6 GB. We observe that Mitos scales gracefully with the number of machines. However, Spark and Flink show a surprising *increase* in execution time as we give more machines to the system. This is because of their overhead in each iteration step increases with the number of machines, and thereby becoming a dominant factor in the execution time. We study this iteration overhead in Section VI-E. In particular, we observe that with the maximum number of machines, Mitos is 10× faster than Spark and 3× faster than Flink. The latter is an interesting result as Flink provides native control flow support. Our system improves over Flink because it performs loop pipelining.

2) PageRank: Figure 5 shows the results for PageRank. Note that Flink does not support this task with its native iterations API. We observe that Mitos scales gracefully, while Spark stops getting faster beyond 9 machines. Our system reaches an improvement factor of 4.6× over Spark with 25 machines.

> Mitos performs and scales better than Spark and Flink. It achieves speedups of 4.6–10× compared to Spark while matching Spark's ease-of-use, and 3× compared to Flink while being easier to use than Flink.

C. Ease-of-Use vs. Performance in Flink

It is worth noting that implementing Visit Count using Flink's native iterations was quite challenging. This is because Flink does not have built-in support for file I/O or if statements inside native iterations. It took us almost 10 hours to implement such

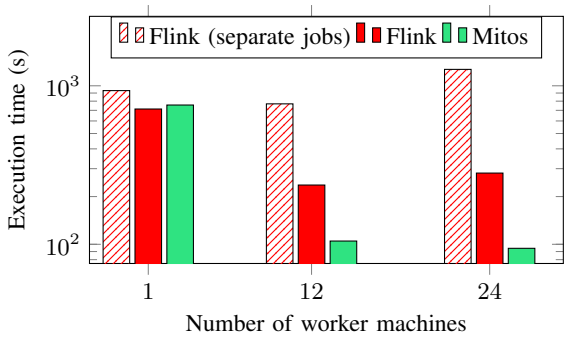


Fig. 6: Easy-to-use Flink workaround.

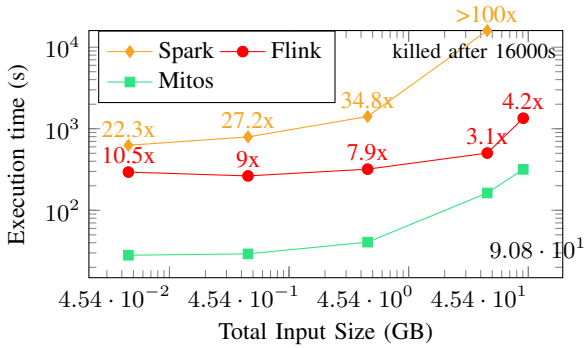


Fig. 7: Visit Count (with the *pageTypes* dataset) when varying the input size. The factors are relative to Mitos.

a task on Flink compared to less than 1 hour for its Spark counterpart. Thus, Flink users (including expert users) would typically resort to the workaround of an imperative loop in the driver program (similarly as in Spark), which launches a separate job per iteration. However, this comes at the price of poor performance. We implemented Visit Count using this workaround, Flink (separate jobs), to show this problem.

Figure 6 shows the results. Note that, as a reference, we also show the numbers for Mitos and Flink (native iterations) from Figure 4. We observe that launching separate Flink jobs from the driver program results in a big performance hit. For 24 machines, this approach is $4.5\times$ slower than Flink native iteration, and $13.5\times$ slower than Mitos. We also observe that the performance of this approach gets worse as we increase the number of machines due to its inherent job-launch overhead. This result shows the high effectiveness and efficiency of our system: it allows users to write control flow imperatively, i.e., it matches the ease-of-use of this approach (as well as of Spark), while still achieving $13.5\times$ better performance.

When users resort to an easy-to-use workaround in Flink due to the limitations of Flink's functional API, Mitos outperforms this approach by more than one order of magnitude.

D. Scalability With Respect to Input Size

Our goal is now to analyze how well Mitos performs with different input dataset sizes for Visit Count. Figure 7 shows the results of this experiment. We observe that our system significantly outperforms Spark and the performance gap increases with the dataset size: it goes from $23\times$ to more

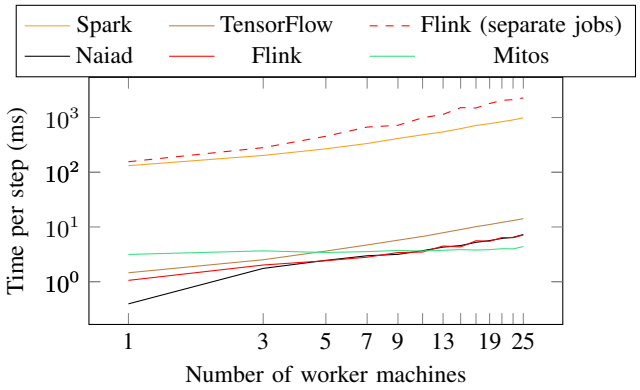


Fig. 8: Log-log plot for the per-step overhead.

than two orders of magnitude. This is because of the loop-invariant hoisting optimization (see Section VI-F for a detailed evaluation). Mitos outperforms also Flink, by 3.1–$10.5\times$, while being easier to use due to its imperative control flow interface. The surprisingly large improvement factor over Flink for small data sizes is due to Flink's native iteration having a large per-step overhead due to a technical issue[9].

Mitos can achieve more than two orders of magnitude speedup compared to Spark for large input datasets.

E. Iteration Step Overhead

We now dive into studying the step overhead. First, we isolate the step overhead from the actual data processing in a microbenchmark: a simple loop with minimal actual data processing in each step. In this experiment, we also considered TensorFlow and Naiad as baselines to better evaluate the efficiency of Mitos. Figure 8 shows the results. We observe that the native iteration of Mitos is about two orders of magnitude faster than launching new jobs for each step, i.e., Spark and Flink (separated jobs). It is interesting to note that the job launch overhead increases linearly with the number of machines. Importantly, this means that scaling out to more machines makes the step overhead problem of Spark worse. Furthermore, we can also see that Mitos matches the performance of other systems with native iterations, i.e., Flink, TensorFlow, and Naiad, despite being able to handle more general control flow. Note that even systems with native control flow have some step overhead (1–10ms). This is because they need to 1) broadcast control flow decisions, and 2) track progress, i.e., determine when operator input for a certain step is complete.

We now investigate the composition of Spark's step overhead. Since in typical cases each step launches a new dataflow job, we have considered so far Spark's step overhead to be the job-launch overhead (task-launch overhead included). However, if a loop body does not contain an action (which is an uncommon case), then Spark can execute the entire loop in a single dataflow job. One might think that this eliminates Spark's step overhead. However, the number of tasks per step is still the same. Therefore, we have to focus on the task-launch overhead (including the initiation of shuffle-reads) to know

[9]https://issues.apache.org/jira/browse/FLINK-3322

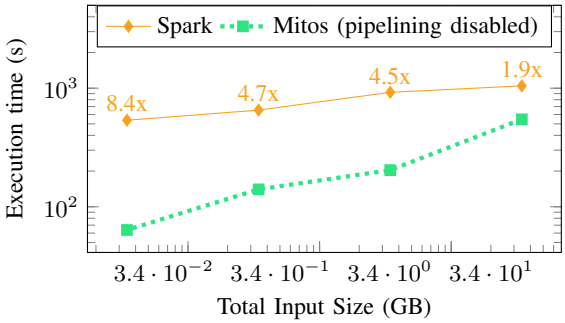


Fig. 9: Visit Count (w/o *pageTypes*) when varying input size.

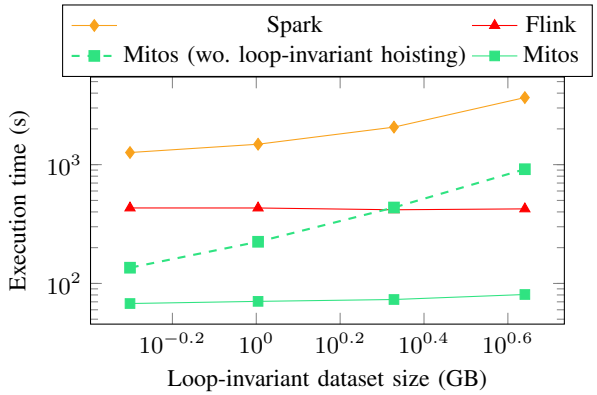


Fig. 10: Varying the loop-invariant dataset size.

the step overhead in this case. We ran a microbenchmark that compares a loop with an action to the same loop without an action, but with the same number of tasks. In our experiments, we observed only a 10% speedup from removing the action. Therefore, we can conclude that most of Spark's step overhead actually comes from launching tasks. In other words, Mitos' performance advantage would not significantly diminish even in the case of a loop with no action.

We now examine how much effect the iteration step overhead has on a real program. As this depends on the amount of actual data processing per step, we ran an experiment where we varied the input size of the Visit Count program. In this experiment, we isolated the effect of removing the job-launch overhead from Mitos' other optimizations: The join with the *pageTypes* dataset is not present in the program, and thus Mitos' loop-invariant hoisting optimization is not applicable. Furthermore, we disabled the loop pipelining optimization of Mitos. Figure 9 shows the result. We observe that increasing the input dataset size decreases the effect of the job-launch overhead, and thereby the improvement factor of Mitos over Spark For a 34 MB input, Mitos is $8.4\times$ faster than Spark. However, even for a 34 GB input, Mitos is still $1.9\times$ faster than Spark. In practice, many real datasets fall into this size range [31].

The overhead of Mitos is two orders of magnitude less than launching separate dataflow jobs per step, which, in real programs, can result in a 1.9–$4.5\times$ speedup over Spark, even when Mitos' other optimizations are disabled.

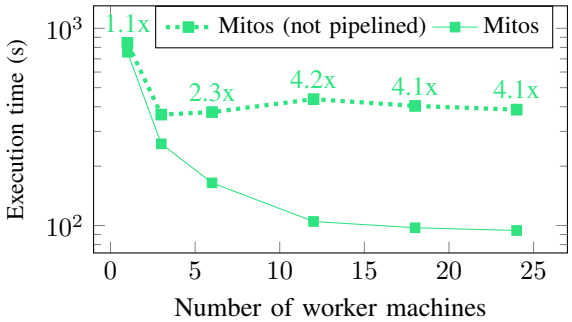


Fig. 11: Loop pipelining with varying worker machine count.

F. Loop-Invariant Hoisting

We proceed to evaluate the loop-invariant hoisting optimization in Mitos. For this, we used the version of the Visit Count example that has the join with the *pageTypes* dataset at every iteration step. The *pageTypes* dataset does not change between steps, and therefore the loop-invariant hoisting optimization can improve performance. Figure 10 shows the results when varying the size of the loop-invariant dataset, while keeping the other part of the input constant (13 GB). We observe that increasing the loop-invariant dataset size has very little effect on Mitos and Flink. This is because they perform the loop-invariant hoisting optimizations i.e., they build the hash table for the join only once and then just probe the hash table at every iteration step. Still, Mitos is 5–$6\times$ faster than Flink.

On the other hand, the execution time of Spark (and the speedup of Mitos over Spark) linearly increases because Spark does not perform this loop-invariant hoisting optimization. Note that, in our Spark implementation, we manually inserted a repartitioning of the *pageTypes* dataset once before the loop. This way, the join does not need to repartition at every iteration step. However, this does not eliminate all redundancy: (1) Matching partitions might still be on different machines, and thus network transfer still happens redundantly at each step; (2) The join's hash table building also still happens redundantly. As a result, Mitos is up to $45\times$ faster than Spark.

To isolate the effect of loop-invariant hoisting from other differences between Spark and Mitos, we also ran Mitos with loop-invariant hoisting switched off. In this case, its execution time increases linearly with the size of the loop-invariant dataset, similarly to Spark. Therefore, Mitos is up to $11\times$ faster than Mitos without loop-invariant hoisting.

Mitos performs loop-invariant hoisting, which improves its performance by up to $45\times$ compared to Spark.

G. Loop Pipelining

We now analyze the loop pipelining feature of Mitos, which allows it to outperform Flink. Recall that, even though Flink also provides native iteration support, our system is up to $3\times$ faster in Figure 4, 3.1–$10.5\times$ faster in Figure 7, and 5–$6\times$ faster in Figure 10. As one might think that this performance difference could come from other factors, we ran an experiment to better isolate the effect of loop pipelining. We ran Visit Count (without the *pageTypes* dataset) in Mitos with and

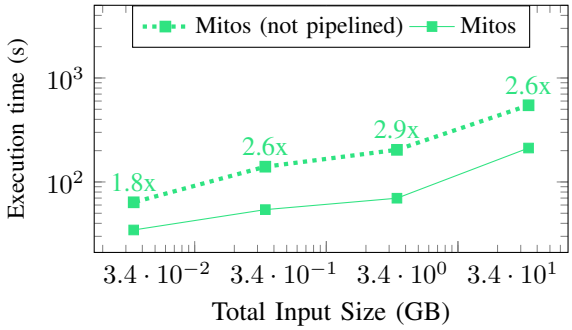


Fig. 12: Effect of loop pipelining when varying the input size.

without the loop pipelining optimization. Figure 11–12 show the results. Overall, we clearly observe the benefits of loop pipelining: Our system can be up to $4\times$ faster with than without loop pipelining, which is made possible by our control flow coordination mechanism. Varying the input size does not have a significant effect on the speedup achievable by loop pipelining.

The control flow coordination algorithm of Mitos allows for loop pipelining, which results in speedups of up to $4\times$.

H. Fault Tolerance

To test Mitos' snapshotting mechanism, we used the Visit Count program (without the *pageTypes* dataset) with an input data size of 34.4 GB. We configured Mitos to snapshot every 10th iteration step. We observed that the execution without Mitos' snapshotting is 205s, while with Mitos' snapshotting is 222s. This represents an overhead of 8.3%, which shows the high efficiency of Mitos' snapshotting algorithm.

VII. Related Work

The dataflow model of computing has a long history [32]. Arvind et al. [33] include control flow into dataflow graphs through the *switch* and *merge* primitives (operations), which TensorFlow recently adopted [14]. Mitos, in contrast to TensorFlow, applies to general data analytics in addition to machine learning. The recent AutoGraph [11] and Janus [12] systems compile imperative control flow to TensorFlow, which makes them not directly applicable for general data analytics. Hirn et al. [34] compile from PL/SQL's imperative control flow to recursive SQL queries.

Several systems can natively support a limited number of control flow constructs, such as Flink [9], and Naiad [18]. However, they rely on functional-style APIs, where each control flow construct is a higher-order function. For example, in TensorFlow, users call the *while_loop* method and provide two functions: one for building the dataflow of the loop body and another for building the dataflow of the loop exit condition. Similarly, in Flink, users call the *iterate* method and supply the loop body as a function that builds the dataflow job fragment representing the loop body. A simple search for these Flink and TensorFlow methods on *stackoverflow.com* shows many users being confused by this API. Mitos allows users to write imperative control flow constructs, such as regular while-loops and if statements, which makes it more accessible to a larger number of programmers. See the Appendix for more discussion comparing functional and imperative control flow APIs.

Other works have added iteration to systems that do not support control flow natively. HaLoop [27] and Twister [28] extend MapReduce to provide support for iterations. Nonetheless, in contrast to Mitos, the programming model of these systems is directly based on MapReduce rather than building complex programs using a collection-based API. Moreover, although loop-invariant hoisting is a well-known optimization in the context of distributed data analytics systems [9], [18], [27], [28], none of these works supports programs with imperative control flow constructs. SystemML [24] does, but it cannot perform it on a binary operator having only one static input, e.g., the hash join that we used in Section V-D.

VIII. Conclusion

Despite modern data analysis requires complex control flow constructs, dataflow systems either suffer from poor execution times for programs with control flow or are hard to use. We presented Mitos, a system that allows users to express control flow by easy-to-use imperative constructs, and still executes these programs efficiently as a single dataflow job. Mitos uses an intermediate representation that abstracts away from specific control flow constructs and that facilitates both building dataflows and coordinating the execution of control flow statements. Our coordination mechanism enables loop pipelining and loop-invariant hoisting. The experimental evaluation shows that Mitos outperforms Spark by up to $45\times$ thanks to native control flow. Interestingly, the results also show that Mitos outperforms Flink, which supports iterations natively, by up to a factor of $10.5\times$ (thanks to loop pipelining and less per-step overhead) while also being easier to use.

Acknowledgments

We thank Alexander Alexandrov for pointing our attention to SSA, and Eleni Tzirita Zacharatou for the system name. This work was funded by the German Ministry for Education and Research as BIFOLD – Berlin Institute for the Foundations of Learning and Data (ref. 01IS18025A and ref. 01IS18037A), and German Research Foundation – Project-ID 414984028 – SFB 1404.

References

[1] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets." *HotCloud*, vol. 10, 2010.
[2] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl *et al.*, "The Stratosphere platform for big data analytics," *The VLDB Journal*, vol. 23, no. 6, 2014.
[3] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache Flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.
[4] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: Bringing order to the web," Stanford InfoLab, Tech. Rep., 1999.
[5] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "Pegasus: mining peta-scale graphs," *Knowledge and information systems*, vol. 27, no. 2, 2011.
[6] X. Wu, V. Kumar, J. R. Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, A. Ng, B. Liu, S. Y. Philip *et al.*, "Top 10 algorithms in data mining," *Knowledge and information systems*, vol. 14, no. 1, 2008.
[7] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola, "Parallelized stochastic gradient descent," in *Advances in neural information processing*, 2010.
[8] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *science*, vol. 220, no. 4598, pp. 671–680, 1983.

[9] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl, "Spinning fast iterative data flows," *Proceedings of the VLDB Endowment*, vol. 5, 2012.

[10] A. Alexandrov, G. Krastev, and V. Markl, "Representations and optimizations for embedded parallel dataflow languages," *ACM Transactions on Database Systems (TODS)*, vol. 44, no. 1, p. 4, 2019.

[11] D. Moldovan, J. Decker, F. Wang, A. Johnson, B. Lee, Z. Nado, D. Sculley, T. Rompf, and A. B. Wiltschko, "AutoGraph: Imperative-style coding with graph-based performance," in *SysML*, 2019.

[12] E. Jeong, S. Cho, G.-I. Yu, J. S. Jeong, D.-J. Shin, and B.-G. Chun, "JANUS: Fast and flexible deep learning via symbolic graph execution of imperative programs," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019, pp. 453–468.

[13] S. M. Orzan, "On distributed verification and verified distribution," Ph.D. dissertation, Vrije Universiteit Amsterdam, 2004.

[14] Y. Yu, M. Abadi, P. Barham, E. Brevdo, M. Burrows, A. Davis, J. Dean, S. Ghemawat, T. Harley, P. Hawkins *et al.*, "Dynamic control flow in large-scale machine learning," in *EuroSys*. ACM, 2018, p. 18.

[15] E. Burmako, "Scala macros: let our powers combine!: on how rich syntax and static types work with metaprogramming," in *Proceedings of the 4th Workshop on Scala*. ACM, 2013.

[16] J. Traub, Z. Kaoudi, J.-A. Quiané-Ruiz, and V. Markl, "Agora: Bringing together datasets, algorithms, models and more in a unified ecosystem [vision]," *SIGMOD Record*, vol. 49, no. 4, pp. 6–11, 2020.

[17] A. Alexandrov, A. Kunft, A. Katsifodimos, F. Schüler, L. Thamsen, O. Kao, T. Herb, and V. Markl, "Implicit parallelism through deep language embedding," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015.

[18] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: a timely dataflow system," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013.

[19] C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly, "Dandelion: a compiler and runtime for heterogeneous systems," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013, pp. 49–68.

[20] D. Agrawal, S. Chawla, B. Contreras-Rojas, A. K. Elmagarmid, Y. Idris, Z. Kaoudi, S. Kruse, J. Lucas, E. Mansour, M. Ouzzani, P. Papotti, J. Quiané-Ruiz, N. Tang, S. Thirumuruganathan, and A. Troudi, "RHEEM: enabling cross-platform data processing - may the big data be with you!" *PVLDB*, vol. 11, no. 11, pp. 1414–1427, 2018.

[21] D. Agrawal, S. Chawla, A. K. Elmagarmid, Z. Kaoudi, M. Ouzzani, P. Papotti, J. Quiané-Ruiz, N. Tang, and M. J. Zaki, "Road to Freedom in Big Data Analytics," in *EDBT*, 2016, pp. 479–484.

[22] D. Agrawal, M. L. Ba, L. Berti-Équille, S. Chawla, A. K. Elmagarmid, H. Hammady, Y. Idris, Z. Kaoudi, Z. Khayyat, S. Kruse, M. Ouzzani, P. Papotti, J. Quiané-Ruiz, N. Tang, and M. J. Zaki, "Rheem: Enabling Multi-Platform Task Execution," in *SIGMOD*, F. Özcan, G. Koutrika, and S. Madden, Eds., 2016, pp. 2069–2072.

[23] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM review*, vol. 59, no. 1, pp. 65–98, 2017.

[24] M. Boehm, M. W. Dusenberry, D. Eriksson, A. V. Evfimievski, F. M. Manshadi, N. Pansare, B. Reinwald, F. R. Reiss, P. Sen, A. C. Surve *et al.*, "SystemML: Declarative machine learning on Spark," *VLDB*, 2016.

[25] F. Rastello, *SSA-based Compiler Design*. Springer, 2016.

[26] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, and tools*, 2nd ed. Addison-wesley Reading, 2007.

[27] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "HaLoop: efficient iterative data processing on large clusters," *VLDB*, 2010.

[28] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: a runtime for iterative MapReduce," in *Proceedings of the 19th ACM international symposium on high performance distributed computing*. ACM, 2010, pp. 810–818.

[29] P. Carbone, S. Ewen, Gy. Fóra, S. Haridi, S. Richter, and K. Tzoumas, "State management in Apache Flink®: consistent stateful distributed stream processing," *Proceedings of the VLDB Endowment*, vol. 10, 2017.

[30] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," in *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. Manhattan, USA: ACM Press, 2004.

[31] G. Piatetsky, "Largest Dataset Analyzed Poll shows surprising stability, more junior Data Scientists," https://www.kdnuggets.com/2016/11/poll-results-largest-dataset-analyzed.html, 2016, [accessed 14-Oct-2020].

[32] P. G. Whiting and R. S. Pascoe, "A history of data-flow languages," *IEEE Annals of the History of Computing*, vol. 16, no. 4, pp. 38–59, 1994.

[33] Arvind and D. E. Culler, "Dataflow architectures," *Annual review of computer science*, vol. 1, no. 1, pp. 225–253, 1986.

[34] D. Hirn and T. Grust, "PL/SQL without the PL," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 2677–2680.

APPENDIX

Listing 2 compares functional control flow APIs and Mitos' imperative API through the Visit Count example program (Section II). For the functional version, we show an idealized version of Flink's API: we extend it with 1) file I/O inside iterations, 2) if statements, 3) support for multiple loop variables, and 4) a *Scalar* type for wrapping non-bag values. However, even all these extensions cannot hide the inconvenience of the functional API, as we can see in the listing.

```
1: pageTypes = readFile("pageTypes")
2: yesterdayCounts = null
3: day = 1
4: while day ≤ 365 do
5:   // Read all page-visits for this day
6:   visits = readFile("pageVisitLog" + day) // pageIDs
7:   // We want to examine only pages of a certain type, so
8:   // we get the page types from a large lookup table:
9:   visits = visits.join(pageTypes).filter(p => p.type=...)
10:  // Count how many times each page was visited:
11:  counts = visits.map(x => (x,1)).reduceByKey(_ + _)
12:  // Compare to previous day (but skip the first day)
13:  if day != 1 then
14:    diffs =
15:      (counts join yesterdayCounts)
16:      .map((id,today,yesterday) => abs(today - yesterday))
17:    diffs.reduce(_ + _).writeFile("diff" + day)
18:  end if
19:  yesterdayCounts = counts
20:  day = day + 1
21: end while
```

(a) Imperative control flow (Mitos).

```
1: pageTypes = readFile("pageTypes")
2: initialCounts = EmptyBag
3: initialDay = Scalar(1) Wrap non-bag values in system-provided types
4: whileLoop( // Higher-order function call
5:   // First two arguments are the initial values of the loop variables:
6:   initialDay, initialCounts,
7:   // Third arg is the function building the dataflow for the body:
8:   (day, yesterdayCounts) => {
9:    fileName = day.map(d => "pageVisitLog" + d)
10:   visits = readFile(fileName)
11:   visits = visits.join(pageTypes).filter(p => p.type = ...)
12:   counts = visits.map(x => (x,1)).reduceByKey(_ + _)
13:   if( // Higher-order function call
14:     // First arg is the function building the dataflow for the condition:
15:     () => day.map(d => d != 1),
16:     // 2nd arg is the function building the dataflow for then-branch:
17:     () => (counts join yesterdayCounts)
18:       .map((id,today,yesterday) => abs(today - yesterday))
19:       .reduce(_ + _).writeFile("diff" + day)
20:   )
21:   day = day.map(d => d + 1)
22:   exitCond = day.map(d => d ≤ 365)
23:   // next values of the loop vars and exit cond:
24:   return (day, counts, exitCond)
25:  }
26: )
```

(b) Functional control flow.

Listing 2: A comparison of control flow APIs through the Visit Count example program (explained in Section II).