Imperative or Functional Control Flow Handling: Why not the Best of Both Worlds?

Gábor E. Gévay
TU Berlin
ggab90@gmail.com

Tilmann Rabl[1]
HPI, Uni Potsdam
Tilmann.Rabl@hpi.de

Sebastian Breß[1]
Snowflake Inc.
sebastian.bress@dfki.de

Loránd Madai-Tahy[1]
mindsquare AG
Lorand.madai@gmail.com

Jorge-Arnulfo Quiané-Ruiz
TU Berlin, DFKI GmbH
jorge.quiane@tu-berlin.de

Volker Markl
TU Berlin, DFKI GmbH
Volker.Markl@dfki.de

ABSTRACT

Modern data analysis tasks often involve control flow statements, such as the iterations in PageRank and K-means. To achieve scalability, developers usually implement these tasks in distributed dataflow systems, such as Spark and Flink. Designers of such systems have to choose between providing imperative or functional control flow constructs to users. Imperative constructs are easier to use, but functional constructs are easier to compile to an efficient dataflow job. We propose Mitos, a system where control flow is both easy to use and efficient. Mitos relies on an intermediate representation based on the static single assignment form. This allows us to abstract away from specific control flow constructs and treat any imperative control flow uniformly both when building the dataflow job and when coordinating the distributed execution.

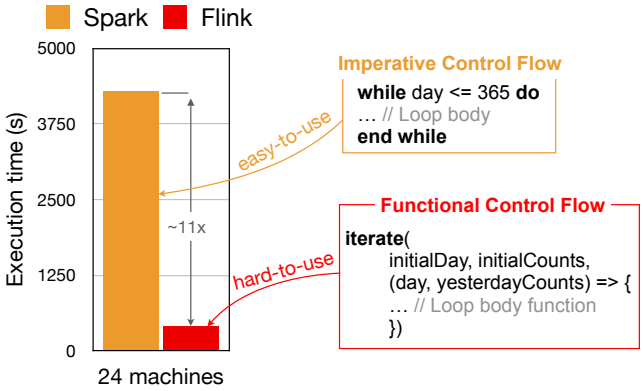


Figure 1: Imperative vs. functional control flow.

1 Introduction

Modern data analytics typically achieve scalability by relying on dataflow systems, such as Spark [23] and Flink [8]. 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 and computing connected components by label propagation. Other data science pipelines are also mainly composed of iterative programs. K-means clustering and gradient descent are two commonly occurring iterative tasks. Additionally, control flow can get more complex: An iterative machine learning training task can be inside another loop for hyperparameter optimization. Nested loops also appear inside a single algorithm, such as the coloring algorithm for computing strongly connected components [18]. Programs may contain if statements inside loops, such as in simulated annealing.

However, despite that control flow statements are at the core of modern data analytics, supporting control flow is still a weak spot 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 and lost optimization opportunities. On the other hand, some other systems, such as Flink and Naiad [17], provide *native control flow* support [10], 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.

To illustrate this problem, we ran an experiment with 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. 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, Flink is harder to use than Spark. In Flink, users call the *iterate* higher-order function and give the loop body as an argument. The loop body is a function that builds the dataflow job fragment representing the actual loop body operations. This API is hard for non-expert

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

 DOI: https://doi.org/10.1109/ICDE51399.2021.00127

users, such as data scientists,[2] who prefer the imperative control flow of Spark, similar to, e.g., Python, R, or Matlab.

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. This is challenging because 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.

We propose Mitos, 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 is based on static single assignment form (SSA). The IR abstracts away specific control flow constructs and thus 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 also 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 use Scala macros [7] to rewrite the user program's abstract syntax tree. (Sec. 4)

(2) We devise a mechanism that coordinates and communicates the control flow decisions between machines. The mechanism supports any imperative control flow uniformly (since it relies on the SSA representation of control flow), and enables two optimizations: loop pipelining, i.e., overlapping iteration steps, and loop-invariant hoisting, i.e., reusing loop-invariant (static) datasets during subsequent iteration steps. (Sec. 5)

(3) We experimentally evaluate Mitos using a real task and microbenchmarks. We mainly compare its performance to Flink (as a system supporting native control flow), and Spark (as a system providing ease-of-use). Mitos is more than one order of magnitude faster than Spark, and, surprisingly, it is also up to 10.5× faster than Flink. (Sec. 6)

2 Motivating Example

We now show an example to illustrate the problems of current dataflow systems 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 each log entry is a page ID, which means that someone has visited the page.

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

```
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 step to launch a new dataflow job, which has an inherent high overhead (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. 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 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. 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 this information, we have to read a dataset containing the page types before the loop. Inside the loop, we then add the line below before Line 3, which joins the visits and page types, 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 [21].

3 Mitos Overview

Mitos compiles a data analysis program with imperative control flow statements into a *single* dataflow job for distributed execution on a dataflow system.

Figure 2 illustrates the general architecture of Mitos. A user provides a data analysis program in a high-level lan-

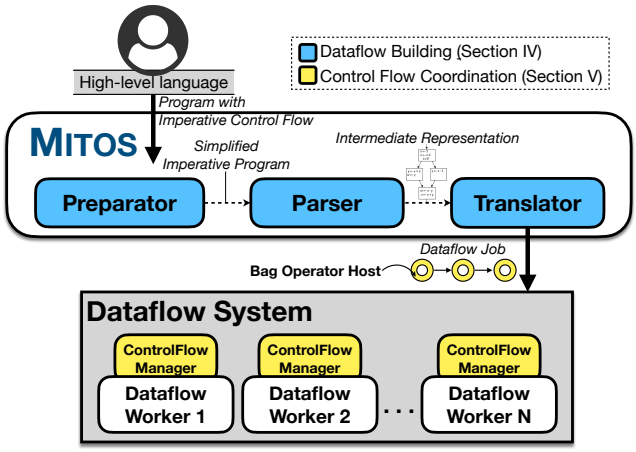


Figure 2: Mitos' architecture.

guage with imperative control flow support. The language has a scalable collection type (akin to RDD), 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 (Sec. 4). Finally, Mitos sends the job for execution on the underlying dataflow system, and coordinates the distributed execution of control flow (Sec. 5).

Generality for Backends. Although we use Flink as our target dataflow system, Mitos is general: 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 [17], Dandelion [20], and TensorFlow. Note that, for Mitos' loop pipelining to have a significant effect, the system should support pipelined data transfers.

Generality for Languages. Although we use the Emma language [2, 1] 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 [7]. Julia [4] and Python also have the required metaprogramming capabilities. Alternatively, SystemDS [5] could also be integrated with Mitos. SystemDS' language is an *external* [1] 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 [19] is often used in compilers to represent imperative control flow. When a program is in SSA, each variable has exactly one assignment statement. 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 a 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.

4 Dataflows Jobs 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 [7] to inspect and rewrite the user program's abstract syntax tree. In more detail, we first simplify the imperative program (Sec. 4.1), and then parse it into an intermediate representation (Sec. 4.2). Both of these facilitate the translation of the user's program into a single dataflow job (Sec. 4.3).

4.1 Simplifying an Imperative Program

First, 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 Sec. 2.

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 we need to deal with only bag operations instead of introducing special cases for non-bag variables.

4.2 Intermediate Representation for General Control Flow

To handle all imperative control flow statements uniformly, Mitos transforms the program into an SSA-based IR [19]. SSA 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:

```
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 makes this runtime choice explicit by introducing Φ-functions (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 in Sec. 5.

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.

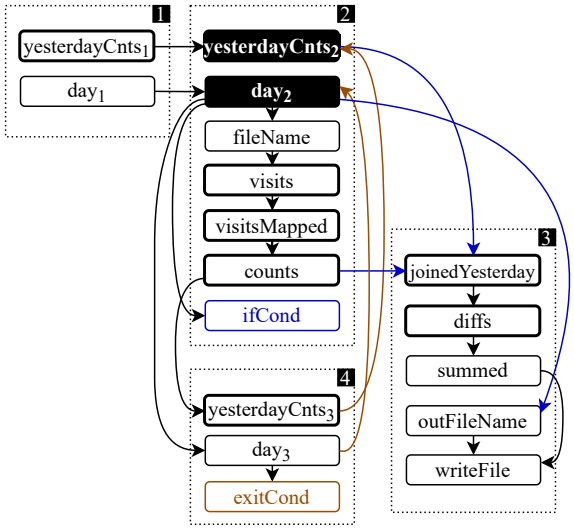
4.3 Translating an Imperative Program to a Single Dataflow

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 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 (Sec. 2). 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 Sec. 4.1, 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.

```
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)

Figure 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.

5 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. 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 (with 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 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 many physical input edges corresponding to one logical input edge.

5.1 Challenges for the Runtime

Challenge 1. Input elements from different bags can get mixed. Mitos performs loop pipelining, i.e., different iteration steps can potentially overlap. That is, at a certain time, 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

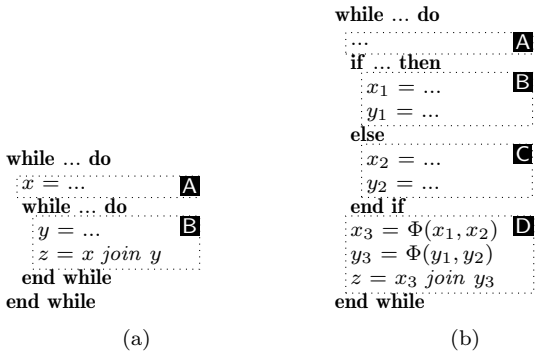


Figure 4: Programs with non-trivial control flow structures.

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 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 Figure 4a 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 Figure 4b 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.

5.2 Coordination Based on Bag Identifiers

We tackle the aforementioned challenges by introducing a *bag identifier* (Sec. 5.2.1). 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 (Sec. 5.2.2); (ii) the identifier of the input bags that it should use to compute a particular output bag (Sec. 5.2.3), and; (iii) on which conditional output edge it should send a particular output bag (Sec. 5.2.4). Note that the Mitos runtime is designed for allowing operators to start computing an output bag as soon as its inputs start to arrive. This enables loop pipelining, i.e., an operator can start a later step while other operators are still working on a previous step.

5.2.1 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.

5.2.2 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 $ABDACD$.

5.2.3 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 Figure 4a 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 com-

puted. Concretely, if we are computing the bag with the path $ABBABBB$, then the prefix we choose is $ABBA$.

5.2.4 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.

5.3 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 Sec. 2: 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 joins with a static dataset containing the graph edges.

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 [10, 6, 9]. 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 across iteration steps. We make this possible by having a single dataflow job, where operator lifetimes span 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 Figure 4a, the first input bag changes at every step of the outer loop, but not between steps of the inner loop.

6 Evaluation

We implemented Mitos on Java 8 and Scala 2.11 and used Flink 1.6 as an underlying dataflow system.

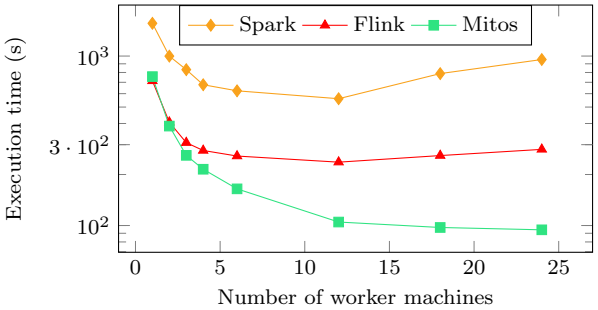


Figure 5: Strong scaling for Visit Count.

6.1 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, Gigabit Ethernet, on Ubuntu 18.04.
Tasks and Datasets. We used the Visit Count example introduced in Sec. 2, where we compare visit counts of subsequent days. We used two versions: one with and one without the join of the *pageTypes* dataset. We have generated random inputs, with the visits uniformly distributed.
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 [17] and TensorFlow [22].
Repeatability. We report numbers for the average of three runs. We also provide the code for Mitos[3].

6.2 Strong Scaling

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

Figure 5 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 Sec. 6.4. 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.

6.3 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 6 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× to more than two orders of magnitude. This is because of the loop-invariant hoisting optimization (see Sec. 6.5 for a detailed evaluation). Mitos outperforms also Flink, by 3.1–10.5×, 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[4].

[3] `https://github.com/ggevay/mitos`
[4] `https://issues.apache.org/jira/browse/FLINK-3322`

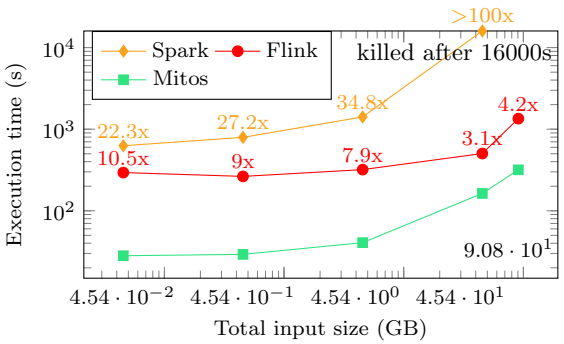


Figure 6: Visit Count (with the *pageTypes* dataset) when varying the input size. The factors are relative to Mitos.

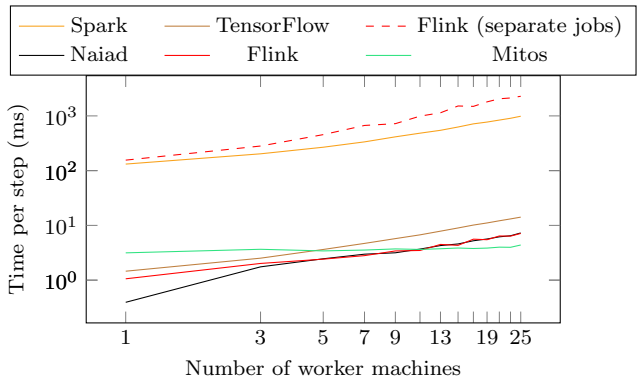


Figure 7: Log-log plot for the per-step overhead.

6.4 Iteration Step Overhead

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 7 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.

6.5 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 loop-invariant *pageTypes* dataset at every iteration step. Figure 8 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.

On the other hand, the execution time of Spark linearly increases because Spark does not perform this loop-invariant hoisting optimization. As a result, Mitos is up to 45× faster

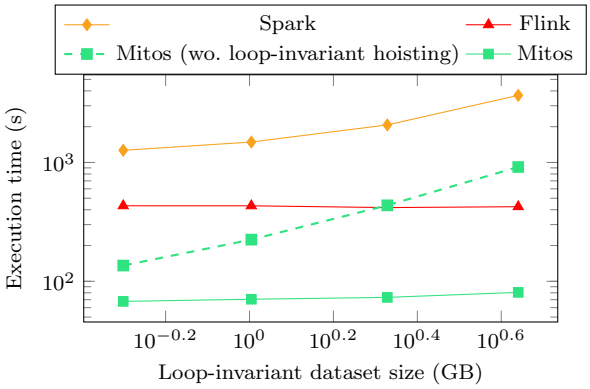


Figure 8: Varying the loop-invariant dataset size.

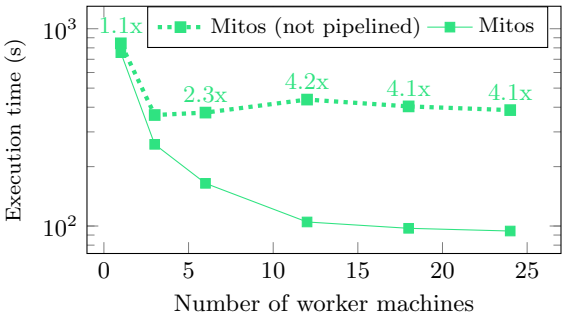


Figure 9: Loop pipelining with varying machine count.

than Spark. Note that, in our Spark implementation, we manually inserted a repartitioning of the *pageTypes* dataset once before the loop.

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× faster than Mitos without loop-invariant hoisting.

6.6 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× faster in Figure 5, 3.1–10.5× faster in Figure 6, and 5–6× faster in Figure 8. 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 without the loop pipelining optimization. Figure 9 shows the results. Overall, we clearly observe the benefits of loop pipelining: Our system can be up to 4× faster with than without loop pipelining, which is made possible by our control flow coordination mechanism.

7 Related Work

Arvind et al. [3] include control flow into dataflow graphs through the *switch* and *merge* primitives (operations), which TensorFlow recently adopted [22]. Mitos, in contrast to TensorFlow, applies to general data analytics in addition to machine learning. The recent AutoGraph [16] and Janus [15] systems compile imperative control flow to TensorFlow, which makes them not directly applicable for general data analytics. Hirn et al. [13, 14] compile from PL/SQL's imperative

control flow to recursive SQL queries. Gévay et al. [12] survey the literature on how various distributed dataflow systems handle control flow. Matryoshka [11] adds control flow support to the flattening technique for nested parallelism.

Several systems natively support a limited number of control flow constructs, such as Flink [10], and Naiad [17]. 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 statements, such as regular while-loops and if statements, which makes it more accessible.

Other works have added iteration to systems that do not support control flow natively. HaLoop [6] and Twister [9] 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.

Although loop-invariant hoisting is a well-known optimization in the context of distributed data analytics [10, 17, 6, 9], none of these works supports programs with imperative control flow constructs. SystemDS [5] does, but it cannot perform it on a binary operator having only one static input, e.g., the hash join that we used in Sec. 5.3.

8 Conclusion

Modern data analysis requires complex control flow constructs, yet dataflow systems either suffer from poor performance 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 based on SSA, which abstracts away from specific control flow constructs. Relying on SSA allows us to handle all imperative control flow in a uniform way, both when building a dataflow job, and when coordinating the distributed execution of control flow statements. Our coordination mechanism enables loop pipelining and loop-invariant hoisting.

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.

9 References

[1] A. Alexandrov, G. Krastev, and V. Markl. Representations and optimizations for embedded parallel dataflow languages. *TODS*, 44(1):4, 2019.

[2] A. Alexandrov, A. Kunft, A. Katsifodimos, F. Schüler, L. Thamsen, O. Kao, T. Herb, and V. Markl. Implicit parallelism through deep language embedding. In *SIGMOD*. ACM, 2015.

[3] Arvind and D. E. Culler. Dataflow architectures. *Annual review of computer science*, 1(1), 1986.

[4] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

[5] M. Boehm, I. Antonov, S. Baunsgaard, M. Dokter, R. Ginthör, K. Innerebner, F. Klezin, S. Lindstaedt, A. Phani, B. Rath, B. Reinwald, S. Siddiqi, and S. B. Wrede. SystemDS: A declarative machine learning system for the end-to-end data science lifecycle. In *CIDR*, 2020.

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

[7] 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.

[8] 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*, 36(4), 2015.

[9] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: a runtime for iterative MapReduce. In *19th ACM international symposium on high performance distributed computing*. ACM, 2010.

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

[11] G. E. Gévay, J.-A. Quiané-Ruiz, and V. Markl. The power of nested parallelism in big data processing–hitting three flies with one slap–. In *SIGMOD*, pages 605–618, 2021.

[12] G. E. Gévay, J. Soto, and V. Markl. Handling iterations in distributed dataflow systems. *ACM Computing Surveys (CSUR)*, 54(9):1–38, 2021.

[13] D. Hirn and T. Grust. PL/SQL without the PL. In *SIGMOD*, pages 2677–2680, 2020.

[14] D. Hirn and T. Grust. One with recursive is worth many GOTOs. In *SIGMOD*, pages 723–735, 2021.

[15] 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 *NSDI*. USENIX Association, 2019.

[16] 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.

[17] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *SOSP*. ACM, 2013.

[18] S. M. Orzan. *On distributed verification and verified distribution*. PhD thesis, Vrije Universiteit Amsterdam, 2004.

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

[20] C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly. Dandelion: a compiler and runtime for heterogeneous systems. In *SOSP*, pages 49–68, 2013.

[21] 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*, 49(4):6–11, 2020.

[22] 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*, page 18. ACM, 2018.

[23] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. *HotCloud*, 10, 2010.