PerfGen: Automated Performance Benchmark Generation for Big Data Analytics

Jiyuan Wang
Tulane University
USA
wjiyuan@tulane.edu

Jason Teoh
University of California, Los Angeles
USA
jteoh@cs.ucla.edu

Muhammad Ali Gulzar
Virginia Tech
USA
magulza@vt.edu

Qian Zhang
University of California, Riverside
USA
qzhang@cs.ucr.edu

Miryung Kim
University of California, Los Angeles
USA
miryung@cs.ucla.edu

Abstract

Many symptoms of poor performance in big data analytics such as *computational skews*, *data skews*, and *memory skews* are input dependent. However, due to lack of inputs that can trigger such performance symptoms, it is hard to debug and test big data analytics.

We design PERFGEN to automatically generate inputs for the purpose of performance testing. PERFGEN overcomes three challenges when naively using automated fuzz testing for the purpose of performance testing. First, typical greybox fuzzing relies on coverage as a guidance signal and thus is unlikely to trigger interesting performance behavior. Therefore, PERFGEN provides performance monitor templates that a user can extend to serve as a set of guidance metrics for grey-box fuzzing. Second, performance symptoms may occur at an intermediate or later stage of a big data analytics pipeline. Thus, PERFGEN uses a *phased fuzzing* approach. This approach identifies symptom-causing intermediate inputs at an intermediate stage first and then converts them to the inputs at the beginning of the program with a pseudo-inverse function generated by a large language model. Third, PERFGEN defines sets of skew-inspired input mutations, which increases the chance of inducing performance problems. We evaluate PERFGEN using four case studies. PERFGEN achieves at least 43X speedup compared to a traditional fuzzing approach when generating inputs to trigger performance symptoms. Additionally, identifying intermediate inputs first and then converting them to original inputs by pseudo-inverse functions, which only takes up to 2 prompting iterations using a large language model, enables PERFGEN to generate such workloads in less than 0.004% of the iterations required by a baseline approach.

CCS Concepts

• **Software and its engineering → Software testing and debugging**.

Keywords

Performance Testing, Big Data

FSE Companion '26, Montreal, QC, Canada

ACM ISBN 979-8-4007-2636-1/2026/07
https://doi.org/10.1145/3803437.3806095

ACM Reference Format:
Jiyuan Wang, Jason Teoh, Muhammad Ali Gulzar, Qian Zhang, and Miryung Kim. 2026. PerfGen: Automated Performance Benchmark Generation for Big Data Analytics. In *34th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE Companion '26), July 05–09, 2026, Montreal, QC, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3803437.3806095

1 Introduction

As the capacity to store and process data has increased remarkably, large scale data processing has become an essential part of software development. Data-intensive scalable computing (DISC) systems, such as MapReduce [16], Apache Hadoop [1], and Apache Spark [5], have shown great promises to address the *scalability* challenge. These systems abstract away the complex execution models, deep software stacks, and numerous customizable configurations from users. However, this comes at the cost of *performance incomprehensibility*, *i.e.,* novice developers lack the necessary knowledge to prevent and correct performance issues. This problem is exacerbated when *performance variability is input dependent* and *existing test data fails to expose pathological performance symptoms*.

Prior work [45] has discussed several major sources of performance slowdowns in DISC applications. Figure 1 visualizes three kinds of performance problems. *Data skew* [25] happens when the input is unevenly distributed across computation nodes, leading to a few nodes taking a much bigger workload and becoming stragglers. *Computational skew* [45] occurs when a normal input triggers an extremely long, intensive computation (*e.g.,* finding the n-th Fibonacci number). *Memory Skew* [11] occurs when a certain record triggers repetitive, memory-intensive operations (*e.g.,* `malloc` or object creation). Performance skews are often *input dependent*. Therefore, our goal is to automatically generate test inputs that can trigger performance skews. Such inputs could guide developers in correcting the root cause of performance bottlenecks.

Challenges. The standard practice for testing data-intensive applications today is to select a subset of inputs based on the developers' hunch with the hope that it will reveal possible performance issues. For example, to test big data applications, developers may use a small sample of data selected via random sampling or top k sampling. Not surprisingly, such sampling is unlikely to yield performance skews described above. Developers could always increase the number of samples, or modify the samples in different ways; however, such adhoc test input generation will significantly increase testing time.

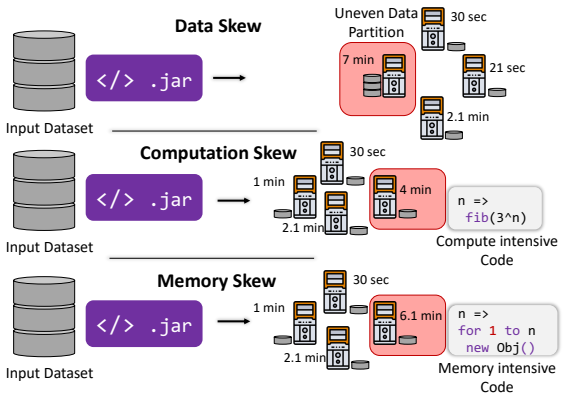

Figure 1: Three sources of performance skews

Fuzz testing has been proven to be highly effective in revealing a diverse set of bugs, including performance defects [26, 35, 42, 48], correctness bugs [9, 40, 41], and security vulnerabilities [12, 15, 19, 43]. Generally speaking, fuzzing techniques start from a seed input, run the program on the selected input, generate new inputs by mutating the previous input, and add new inputs to the queue if they improve a given guidance metric such as branch coverage. However, it is nontrivial to apply such traditional fuzzing to data-intensive applications for the following reasons.

First, the *long latency* of big data analytics prohibits repetitively invoking the entire application from the beginning—a common assumption for fuzz testing techniques. Inherently, these are two conflicting goals: eliminating long-latency during testing vs. encountering performance skews that are rare during testing. In particular, when the performance skew symptoms appear at a later computation stage, it is difficult to generate inputs in the beginning of the program, which induce performance problems in the later stage.

Second, to discover pathological inputs that trigger performance symptoms, *feedback metrics* must account for underlying systems-level metrics such as CPU time and memory usage. Prior work PerfFuzz [26] is limited to simply counting the number of exercised control flow graph edges as feedback (i.e., inducing a longer execution path). To trigger various performance skews in big data analytics, a user should be able to inject performance monitors that can watch partition-level runtime, peak memory usage, and garbage collection time and compare these metrics across multiple partitions.

Third, in terms of *input mutation operations*, random bit-level input mutations used in traditional fuzzers like AFL [51] may not be effective at exposing real performance problems. This is because the entire running time is strongly influenced by the *collective properties of the dataset* (e.g., key distribution, garbage collection that depends on the entire dataset's in-situ memory needs, differences in parallel task execution latency). In other words, input mutations for performance fuzzing must ensure that each record compiles with the input schema and that either the content of data set should be modified in ways that target specific performance symptoms.

Our Solution. PERFGEN automatically generates test inputs to trigger a performance skew. In PERFGEN, a user can easily specify a performance symptom of interest using pre-defined performance monitors. PERFGEN then automatically inserts the corresponding performance monitor and uses it as feedback guidance for grey-box fuzzing. To overcome the aforementioned challenges of adapting fuzz testing for performance workload generation—long latency, lack of performance feedback, and low-level input mutations, PERFGEN combines three technical innovations:

First, to trigger a performance symptom appearing in the later computation stage, PERFGEN uses a *phased fuzzing* approach. Deeper program paths cannot be easily reached through input generation; however, it is relatively easier to target a single user-defined function (UDF) in isolation. Our phased fuzzing targets an individual user-defined function at a given stage to gain knowledge about the intermediate inputs that trigger a performance symptom. Then using a pseudo-inverse function that is auto-generated by a large language model GPT-5 [38], PERFGEN converts the intermediate inputs to corresponding inputs at the beginning of the program, which are then used as improved seeds for fuzzing the entire program.

Second, PERFGEN enables users to specify performance symptoms by implementing customizable monitor templates which provide useful guidance metrics for fuzzing. PERFGEN currently supports performance outlier detecting templates for computation, memory, and data skew symptoms. These templates relate symptoms with relevant performance metrics such as partition runtime, memory usage, and shuffle sizes.

Third, PERFGEN improves its chances of constructing meaningful inputs with sets of skew-inspired mutations. PERFGEN supports a variety of input mutation operations including data replication, column/field-level mutations for multi-value inputs, and mutations that exploit key-value distributions.

Evaluation. We compare the test generation time and the number of iterations required for triggering performance symptoms with and without phased fuzzing. Across our four case studies, PERFGEN achieves more than 43X speedup in time and requires less than 0.004% iterations compared to PERFGEN without phased fuzzing. Additionally, PERFGEN's template-inspired mutation probabilities result in a 1.81X speedup in input generation time compared to a uniform sampling configuration.

Most fuzzing techniques focus on correctness testing with crash symptoms. Compared to crashes, performance problems are not explored much in automated fuzzing. PERFGEN is the first automated fuzzer that can trigger data, memory, and computational skews.

- We present an automated test generation framework that supports various kinds of performance monitors for big data analytics. PERFGEN is built on Apache Spark and its key idea generalizes to other data-intensive scalable computing applications such MapReduce and Hadoop.
- We are the first to employ a *targeted, phased fuzzing* approach to expedite fuzz testing by generating intermediate inputs at a later stage first and then mapping them to the improved seeds by a pseudo-inverse function at the beginning of a program. It only takes up to 2 prompting iterations to generate the pseudo-inverse function and achieves 43X compared to the baseline.
- We are the first to define skew-inducing input mutations for data-intensive analytics. These input mutations duplicate data rows, adjust key-value distributions, or add additional records with the same key while mixing values from other rows. These input mutations differ from typical bit-level mutations and

are designed to trigger data, memory, and computation skews easily.

2 Motivating Example

```
1  val inputs = sc.textFile("collatz.txt") // read input
2  val trips = inputs
3    .flatMap(line => line.split(" ")) // split by space
4    .map(s=>(Integer.parseInt(s),1))// parse ints into pair
5  val groups=trips.groupByKey(4)//group into 4 partitions
6  val solved = groups.map { s => // apply UDF to generate
7     (s._1, solve_collatz(s._1)) } // new pair values
8  val sum=solved.reduceByKey((a, b) => a + b)//sum by key
```

Figure 2: The *Collatz* program which applies the `solve_collatz` function to each input integer and sums the result by distinct integer input.

```
1  def solve_collatz(m:Int): Int ={
2     var k=m, i=0
3     while (k>1) { // compute collatz sequence length, i
4       i=i+1
5       if (k % 2==0) k = k/2 else {k=k*3+1}}
6     var a=i+0.1
7     for (j<-1 to i*i*i*i){ // O(i^4) computation loop
8       a = (a + log10(a))*log10(a)}
9     a.toInt
10 }
```

Figure 3: The `solve_collatz` function used in Figure 2 to determine each integer's Collatz sequence length and compute a polynomial-time result based on the sequence length. For example, an input of *3* has a Collatz length of 7 and calling `solve_collatz(3)` takes 1 ms to compute, while an input of *27* has a length of 111 and takes 4.9 s to compute.

To demonstrate the challenges of performance debugging and how PERFGEN addresses such challenges, we present a motivating example using a program inspired by [49]. In this example, a developer uses the *Collatz* program shown in Figure 2. The *Collatz* program consumes a string dataset of space-separated integers to compute a mathematical result for each distinct integer based on its *Collatz* sequence length and number of occurrences. For each parsed integer, the program applies a mathematical function `solve_collatz` (Figure 3) to compute a numerical result based on each integer's Collatz sequence length, in polynomial time with respect to that length. After applying `solve_collatz` to each integer, the program then aggregates across each integer and returns the summed result per distinct integer.

Suppose the developer is interested in exploring the performance of this program, particularly the *solved* variable which applies the `solve_collatz` funtion. They want to generate an input dataset that will induce performance skew by causing a single data partition to require at least five times the computation time of other partitions. In other words, they wish to find an input that meets the following symptom predicate:

$$\frac{SlowestPartitionRuntime}{SecondSlowestPartitionRuntime} \geq 5.0$$

As a starting point, the developer generates an initial input consisting of four single-record partitions: "1", "2", "3", and "4". However, this simple input does not result in any significant performance skew within the *Collatz* program.

The developer initially turns to traditional fuzzing techniques for help in generating a skew-inducing input dataset. However, such approaches either flip individual bits or some bytes in the dataset in an attempt to produce new inputs. Because *Collatz*'s string inputs are parsed into integers, such inputs may not be program-compatible and therefore could not reach the `solve_collatz` function and induce performance skew. Furthermore, traditional fuzzing techniques typically use code branch coverage as guidance and thus do not consider performance metrics.

```
1  def inverse(udfInput: RDD[(Int, Iterable[Int])]): RDD[String]
       = {
2     udfInput.flatMapValues(identity).map(s => s._1.toString)
3  }
```

Figure 4: A pseudo-inverse function to convert *solved* inputs into inputs for the entire *Collatz* program (Figure 2, lines 1-7).

The developer decides to use PERFGEN to generate an input that produces performance skew for the *Collatz* program. Suppose that the developer suspects that *solved* UDF may have a performance problem. PERFGEN's phased fuzzing approach first targets this function to generate intermediate inputs. It then uses a pseudo-inverse function to convert the *solved*'s intermediate inputs to improved seeds for *Collatz*. For example, Figure 4 shows such a pseudo-inverse function that guesses seed inputs from the intermediate inputs. Such a pseudo-inversion function does not need to be an exact inverse function as its goal is to generate improved seeds for subsequent fuzzing iterations. PERFGEN applies the LLM to generate a pseudo-inverse function. As shown in Figure 9, it took only 1 prompting to create this pseudo inverse function using ChatGPT.

Next, the developer defines their symptom in PERFGEN by selecting a *monitor template* and performance *metric* from Tables 1 and 2. Based on the symptom predicate described earlier, they choose a *NextComparison(5.0)* monitor template and the *Runtime* metric to create the following symptom monitor. $[Runtime]$ is the collection of partition runtimes for a given job execution:

$$\frac{max([Runtime])}{max([Runtime] - \{max([Runtime])\})} \geq 5.0$$

This predicate inspects the partition runtimes for a given job execution and checks if the longest partition runtime is at least five times as long as all other partition runtimes.

Using this symptom monitor, PERFGEN produces mutations from Table 3 for both intermediate *solved* inputs as well as *Collatz* program inputs. For example, the mutations for *solved*'s *(Int, Iterable[Int])* inputs include mutations which randomly replace the integer values in record keys or values, or alter the distribution of data by appending newly generated records. In addition to producing mutations, PERFGEN also defines mutation sampling probabilities by assigning sampling weights to each mutation based on their alignment with the symptom definition; for example, mutations associated with computation skew have higher sampling probabilities when PERFGEN is given a computation skew symptom.

The user-specified target UDF, monitor template, and metric are shown in Figure 5. Using this configuration, PERFGEN begins its phased fuzzing approach. It first executes *Collatz* with the input data until reaching inputs to *solved*, producing the partitioned UDF input shown in Figure 6. Next, it uses the derived mutations to fuzz *solved* first. After a few iterations, it produces the symptom-triggering UDF inputs starred in Figure 6 by adding the bolded record. As a result of the key's long Collatz length and the `solve_collatz` function,

```
val config = PerfGenConfig(
   programOutput = sum, // HybridRDD output of entire program
   targetUDF = solved, // HybridRDD output of target UDF
   monitorTemplate = nextComparison, // Monitor Template /
         Symptom definition
   inputMutationMap, // Program mutations
   udfMutationMap, // UDF mutations
   seed = (("1"),("2"), ("3"), ("4")), // initial seed input
   inverse // pseudo-inverse function from Collatz program.
)
PerfGen.run(config)
```

Figure 5: Code demonstrating how a user can configure PERFGEN for the Collatz program

this intermediate input executes slowly for only one of the data partitions and satisfies the performance skew definition.[1]

Next, PERFGEN applies the pseudo-inverse function to this UDF input to produce the *Collatz* program input shown in Figure 6. Upon testing, PERFGEN finds that the converted inputs also exhibit performance skew for the full *Collatz* and returns the dataset to the user for further analysis. At this point, the user now possesses a *Collatz* program input triggering a performance skew symptom of interest.

3 Approach

As shown in Figure 6, PERFGEN takes as input a DISC application built on Apache Spark, an initial input seed, and a symptom of interest defined using a monitor template.

PERFGEN extends a traditional fuzzing workflow with four novel contributions. Section 3.1 describes PERFGEN's *HybridRDD* extension to the Spark RDD API in order to support execution of individual UDFs for more precise fuzzing. Section 3.2 enables a user to specify a desired symptom via execution *metrics* and predefined *monitor templates* which define patterns to detect symptoms. Section 3.3 leverages type knowledge from the isolated UDF as well as the symptom definition to define a set of *skew-inspired mutations* designed to generate syntactically valid inputs geared towards producing the performance symptom of interest. Finally, Section 3.4 combines these techniques to find symptom-reproducing UDF inputs first and to produce improved seeds for fuzzing the entire program end to end.

3.1 Enabling an entry point to UDFs

DISC applications have longer latency than other applications, making them unsuitable for iterative fuzz testing end to end [52]. Triggering a performance skew is also an extremely rare event to induce by chance via low-level mutation, when the symptom occurs deep in the program. Fuzzing is easier for a single UDF in isolation than fuzzing an entire application to reach a deep execution path. However, fuzzing a UDF requires having an entry point to the UDF.

Suppose that a user desires to have a direct entry point to a specific UDF of interest (Figure 6 label 1). Existing DISC systems such as Spark define datasets in terms of transformations (including UDFs) directly applied to previous datasets. As a result, such programs do not support decoupling UDFs from input datasets without manual refactoring or system modifications. It is nontrivial to execute a program (or subprogram) with new inputs.[2]

To enable a direct entry point to a UDF, PERFGEN wraps Spark *RDD*s with its own *HybridRDD*s. While *HybridRDD*s are functionally equivalent to RDDs, they internally separate transformations from the datasets on which they are applied and store information about the corresponding input and output data types. Using this new *HybridRDD* interface, a user can easily specify a target UDF of interest. Figures 7a and 7b illustrate the API changes required to leverage PERFGEN's *HybridRDD* for the *Collatz* program discussed in Section 2. PERFGEN automatically decouples the *map* transformation of `solved` from its predecessor (`grouped`) to produce a function of type `RDD[(Int, Iterable[Int])] => RDD[(Int, Int)]` which captures the `solve_collatz` function used in the `map` transformation.

3.2 Modeling performance symptoms

Detecting performance skews often requires monitoring task execution time, the number of records read or written during a shuffle, and memory usage. To guide test generation towards exposing performance skews, PERFGEN provides a set of eight customizable *monitor templates* and ten performance *metrics* that we constructed using Spark's Listener API. They are listed in Tables 1 and 2 respectively. These metrics are measured at the level of each partition and each stage.

Our insight behind these templates is that *DISC performance skews often follow patterns* and each performance symptom could be modeled using *monitor template* with a choice of *performance metrics* (Figure 6 label 2). PERFGEN uses these monitoring templates as both an oracle and a guidance feedback, as opposed to detecting crashes as an oracle and monitoring branch coverage for grey-box fuzzing.

Consider a symptom where any partition's runtime during a program execution exceeds 100 seconds. This symptom can be defined by using the *Runtime* metric and *MaximumThreshold* monitor template, which evaluates the following predicate using the collected partition runtimes from Spark to determine if the performance symptom is triggered: $\max([Runtime]) \geq 100s$. In addition to detecting symptoms, the monitor template also provides a feedback score corresponding to the largest metric (runtime) value observed.

While PERFGEN models many symptoms via the definitions in Tables 1 and 2, other symptoms may require additional patterns or metrics. PERFGEN enables users to define custom templates by implementing a monitor template interface.

3.3 Skew-Inspired Input Mutation Operations

Consider the Collatz program from Section 2, which parses strings as space-separated integers. When bit-level or byte-level mutations are applied to such inputs, they can hardly generate meaningful data that drives the program to a deep execution path since bit-flipping is likely to destroy the data format or data type. For example, modifying an input "10" to "1a" would produce a parsing error since an integer number is expected. Additionally, DISC applications include distributed performance bottlenecks such as data shuffling that is dependent on the characteristics of the entire dataset and may be difficult or impossible to trigger with only record-level mutations. Designing mutations to detect performance skews in

[1] "474680340" has a Collatz sequence length of 192, while the remaining records' lengths are no more than 7.

[2] For example, Spark's various *RDD* implementations including *MapPartitionsRDD* and *ShuffledRDD* capture information about transformations via private, operator-specific objects such as iterator-to-iterator functions or Spark *Aggregator* instances. Reusing these transformation definitions with new inputs requires direct access to Spark's internal classes.

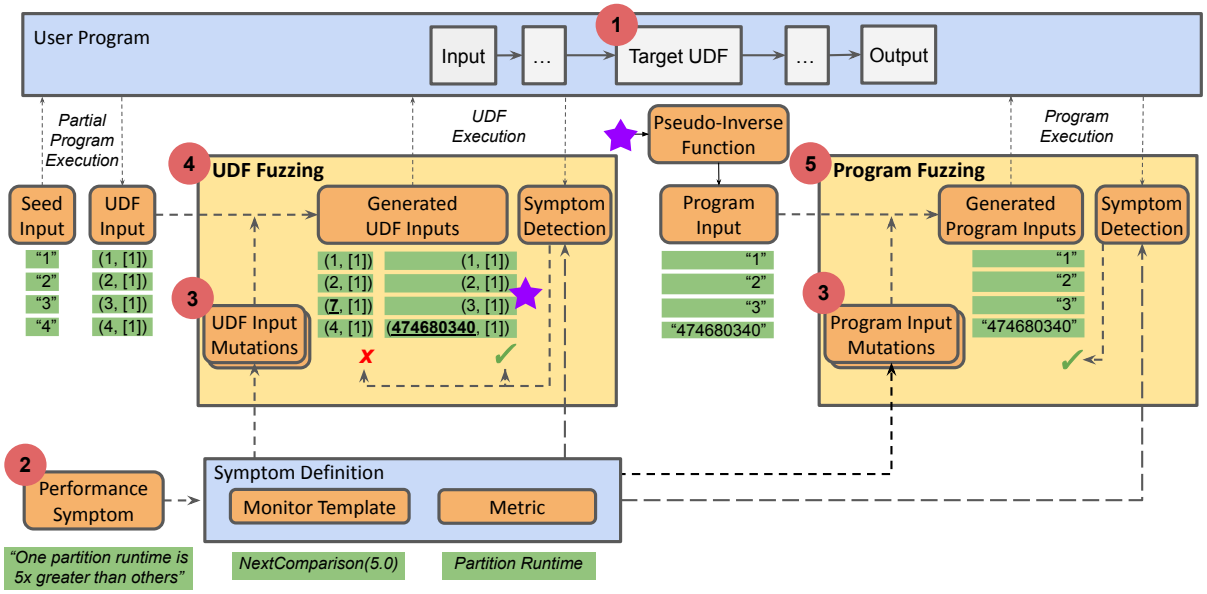

Figure 6: An overview of PERFGEN's phased fuzzing approach. Suppose that a user wants to generate test data that can induce a performance symptom of interest at a given stage. PERFGEN generates a set of mutations for both UDF and program input fuzzing. It first fuzzes the target UDF to reproduce the desired performance symptom. It then applies a pseudo-inverse function to generate improved seeds. These seeds are used to fuzz the entire program to induce the performance symptom of interest.

```
1 val inputs = sc.textFile("collatz.txt")
2 val trips = inputs
3   .flatMap(line => line.split(" "))
4   .map(s=>(Integer.parseInt(s),1))
5 val grouped = trips.groupByKey(4)
6 val solved = grouped.map { s =>
7    (s._1, solve_collatz(s._1)) }
8 val sum = solved.reduceByKey((a, b) => a + b)
```

(a) A DISC Application `Collatz.scala`

```
1 val inputs = HybridRDD(sc.textFile("collatz.txt"))
2 val trips: HybridRDD[String, (Int, Int)] = inputs
3  .flatMap(line => line.split(" "))
4  .map(s=>(Integer.parseInt(s),1))
5 val grouped: HybridRDD[(Int, Int), (Int, Iterable[Int])] =
6  trips.groupByKey(4)
7 // RDD corresponding to the target UDF
8 val solved: HybridRDD[(Int, Iterable[Int]), (Int, Int)] =
9  grouped.map { s =>(s._1, solve_collatz(s._1)) }
10 val sum: HybridRDD[(Int, Int), (Int, Int)] = solved.reduceByKey((a, b)
     => a + b)
```

User-defined function

(b) Transformed `Collatz` with `HybridRDD`

Figure 7: PERFGEN's *HybridRDD* supports extraction and reuse of individual UDFs

DISC applications requires that (1) mutations must ensure type-correctness, and (2) mutations should be able to manipulate input datasets in ways that comprehensively exercise the performance-sensitive aspects of distributed applications. For example, mutations should shuffle or redistribute data.

For example, PERFGEN targets data skew symptoms by defining mutations that alter the distribution of keys and values in tuple inputs, as well as mutations that extend the length of collection-based fields (which might be flattened into multiple records and contribute to data skew later in the application). PERFGEN's mutations alter specific values or elements in tuple and collection datasets. *AppendSameKey* mutation (M13 in Table 3) targets data skew by appending new records for a pre-existing key.

PERFGEN adjusts the sampling probability of each mutation based on the skew category associated with the symptom of interest (Figure 6 label 3). Table 3 describes PERFGEN's mutations that target different skew categories.

3.4 Phased Fuzzing

PERFGEN's *phased fuzzing* technique generates intermediate inputs to a given UDF and produces improved seeds for fuzzing the entire program, sketched in Figure 8.

Step 1. UDF Fuzzing. PERFGEN generates an initial UDF input by partially executing the original program. Using this intermediate result as a seed, it then fuzzes the target UDF using the procedure outlined in Figure 8. The process is illustrated in Figure 6 label 4 with concrete inputs from the motivating example. Two nontrivial outcomes exist for each fuzzing loop iteration: (1) the monitor template detects that the desired symptom is triggered and terminates the fuzzing loop or (2) the monitor template does not detect skew but returns a feedback score that is better than previously observed, so PERFGEN adds saves the mutated input, updates the best observed feedback score, and resumes fuzzing with the updated input queue.

Step 2. Pseudo-Inverse Function generation with the Large Language Model. While targeted UDF fuzzing enables PERFGEN to generate symptom-triggering intermediate inputs, the final objective

Template(parameters)	Predicate	Description
MaximumThreshold(X,t)	$\max(X) \geq t$, where t = value threshold	Compares the maximum value of X to a threshold t.
NextComparison(X,t)	$\frac{\max(X)}{\max(X - \{\max(X)\})} \geq t$, where t = ratio threshold	Computes the ratio between the two largest metric values in X and compares it to a threshold t.
IQROutlier(X,t)	$\frac{\max(\max(X) - Q_3, Q_1 - \min(X))}{Q_3 - Q_1} \geq t$, where Q_1, Q_3 = first and third quartiles of X, t = IQR distance threshold (default 1.5)	Computes the largest interquartile range (IQR) [2] distance in X and compares it to a threshold t.
Skewness(X,t)	$\frac{m_3}{\sigma^3} \geq t$, where m_3 = third central moment of X, σ = standard deviation of X, t = skewness threshold (default 1.0)	Computes the skewness [4] of X and compares it to a threshold t.
ZScore(X,t)	$\frac{\max(X) - \mu}{\sigma} \geq t$, where μ = mean of X, σ = standard deviation of X, t = z-score threshold	Computes the largest z-score [6] in X and compares it to a threshold t.
ModZScore(X,t)	$\frac{\max(X) - M}{1.486 * MAD} \geq t$, where M = median of X, MAD = median absolute deviation of X, t = modified z-score threshold	Computes the largest modified z-score [3] in X and compares it to a threshold t.
LeaveOneOutRatio(X,t)	$\frac{\max(X)}{\text{mean}(X - \{\max(X)\})} \geq t$, where t = target ratio threshold	Computes the ratio between the largest metric and the average of all other metrics, and compares it to a threshold t.
ErrorDetection(X,s, mt)	error is thrown and error message contains substring s	Monitors for thrown exceptions with error messages containing the specified substring s.

Table 1: *Monitor Templates* define *predicates* that are used to (1) detect specific symptoms and (2) calculate feedback scores, given a collection of values X derived using performance metrics definitions such as those from Table 2.

PerformanceMetric	Skew Category
Job Execution Time	Computation & Data
Garbage Collection Time	Memory
Peak Memory Usage	Memory
Memory Bytes Spilled on Disk	Memory
Input Read Records Count	Data
Output Write Records Count	Data
Shuffle Read Records Count	Data
Shuffle Read Bytes	Data
Shuffle Write Records Count	Data
Shuffle Write Bytes	Data

Table 2: Performance metrics captured by PERFGEN

```
1  def phasedFuzzing[I, U, O](conf: PerfGenConfig[I, U, O]):
       RDD[I] = {
2  // Step 1: Fuzz the target UDF to produce symptom-triggering
       intermediate inputs
3    // partially run program up until UDF
4    val udfSeed: RDD[U] = computeUDFInput(conf.seed)
5    val udfSymptomInput: RDD[U] = fuzz(conf.udfProgram,
6      udfSeed, conf.monitorTemplate, conf.udfMutations)
7  // Step 2: Use pseudo-inverse function to generate program seed
8    val programSeed: RDD[I] =
         conf.inverseFn.apply(udfSymptomInput)
9  // Step 3: Fuzz the full program to produce symptom-triggering
       program inputs
10   val programSymptomInput: RDD[I] = fuzz(conf.fullProgram,
         programSeed, conf.monitorTemplate,
         conf.programMutations)
11   return programSymptomInpu}
12
13 def fuzz[T,U](progFn: RDD[T] => RDD[U], seed: RDD[T], monitor:
       MonitorTemplate, mutations: MutationMap[T]) = {
14   val seeds = List(seed)
15   var maxScore = 0.0
16   while(true) { // not timed out
17     // select a seed and apply a mutation
18     val base = sample(seeds)
19     val newInput = mutations.sample().apply(base)
20     val programOutput = progFn(newInput) // test the new input
21     // Use monitor template to check if symptom was reproduced
22     // or if feedback score was increased.
23     val metrics = config.metric.getLastExecutionMetrics()
24     val (meetsCriteria, feedbackScore) =
           monitor.checkSymptoms(metrics)
25     if(meetsCriteria) { // symptom reproduced
26       return newInput}
27     else if (feedbackScore>maxScore) {// score increased
28       maxScore = feedbackScore
29       seed.append(newInput)}}}
```

Figure 8: PERFGEN's *phased fuzzing* approach for generating symptom-reproducing inputs, using feedback scores from monitor templates to guide fuzzing.

is to identify inputs to the entire program. PERFGEN uses a *pseudo-inverse function* to convert intermediate UDF inputs to improved seeds. Such pseudo-inverse function does not need to be an exact inverse function as its goal is to bootstrap seed input generation. *Collatz* pseudo-inverse function is shown in Figure 4.

As illustrated in Figure 9, a pseudo-inverse function can be generated by an LLM such as GPT-5. For example, PERFGEN prompts the GPT-5 model to generate an inverse function by submitting the source code of *Collatz*; the request could be framed as, "Could you generate an inverse function that transitions from the RDD `grouped` to the original input? There is no need to provide the precise reverse function, just try your best." In response, the AI model will strive to deliver an appropriate inverse function.

Take, for instance, a dataset comprising student grades with an aggregation that computes the mean grade per course. Given an intermediate dataset displaying courses and their average grades, GPT synthesizes a pseudo-inverse function that outputs a single student grade per course, rounded to the nearest integer equivalent of the average grade. While such a function can only approximate average grades and consequently may not precisely replicate the provided intermediate inputs, the function can still facilitate the production of enhanced seeds.

A pseudo-inverse functions for the portion of a program that precedes the target User Defined Function (UDF), is independent of the

ID	Name	Data Type	Target Skew(s)	Description
M1	ReplaceInteger	Integer	Computation	Replace the input integer with a randomly generated integer value within a configurable range (default: [0, `Int.MaxValue`)).
M2	ReplaceDouble	Double	Computation	Replace the input double with a randomly generated double value within a configurable range (default: [0, `Double.MaxValue`))
M3	ReplaceBoolean	Boolean	Computation	Replace the input boolean with a random boolean value.
M4	ReplaceSubtring	String	Computation	Mutate a string by replacing a random substring (including either empty or the full string) with a newly generated random string of random length within a configurable range (default: [0, 25)).
M5	ReplaceCollectionElement	Collection	Computation	Randomly select and mutate a random element within a collection according to its type.
M6	AppendCollectionCopy	Collection	Computation, Data, Memory	Extend a collection by appending a copy of itself.
M7	ReplaceTupleElement	2-Element Tuple	Computation	Randomly mutate an element within a two-element tuple according to its type.
M8	ReplaceTripleElement	3-Element Tuple	Computation	Randomly mutate an element within a three-element tuple according to its type.
M9	ReplaceQuadrupleElement	4-Element Tuple	Computation	Randomly mutate an element within a four-element tuple according to its type.
M10	ReplaceRandomRecord	Dataset	Computation	Randomly select a record and mutate it according to one of the mutations applicable to the dataset type. For example, this mutation could choose a random integer out of an integer dataset and apply the *ReplaceInteger* mutation.
M11	PairKeyToAllValues	2-Element Tuple Dataset	Data, Memory	Randomly select a random record. For each distinct value within that record's partition, append a new record to the partition consisting of the the selected record's key and the distinct value, such that the key is paired with every value in the partition.
M12	PairValueToAllKeys	2-Element Tuple Dataset	Data	Similar to *PairKeyToAllValues* but instead pairing a random record's value with all distinct keys in a partition.
M13	AppendSameKey	2-Element Tuple Dataset	Data, Memory	Randomly select a random record. Append additional records consisting of that record's key paired with mutations of its value some number of times (default: up to 10% of partition size).
M14	AppendSameValue	2-Element Tuple Dataset	Data	Similar to *AppendSameKey* but instead with a fixed value and mutated keys.

Table 3: Skew-inspired mutation operations implemented by PERFGEN for various data types and their typical skew categories. Some mutations depend on others (*e.g.*, due to nested data types).

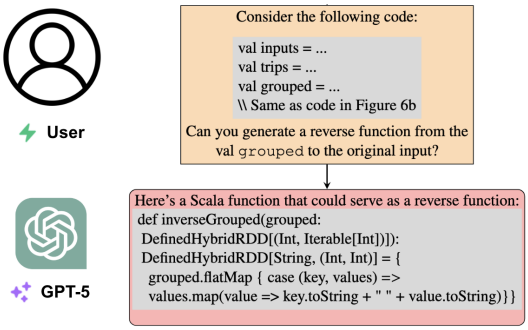

Figure 9: An example of PERFGEN's pseudo-inverse function.

target UDF implementation. For example, Figure 4 for the *Collatz* program in Figure 2 excludes the target UDF `solve_collatz`. Additionally, a pseudo-inverse function remains unaffected by target symptoms. Therefore, it is external to PERFGEN's *phased fuzzing* process and in most cases, is simple enough for a large language model to automatically generate an accurate version. In our evaluation, pseudo-inverse functions averaged 6 lines of code.

Step 3. End-to-End Fuzzing with Improved Seeds. As a final step, PERFGEN tests the pseudo-inverse function result to see if it is a symptom-triggering input. If not, it uses the derived program input as an improved seed for fuzzing the entire application as shown in Figure 6 label 5. This step resembles UDF fuzzing (Figure 8) and reuses the same monitor template, but initializes with the pseudo-inverse function output as a seed and utilizes a different set of mutations suitable for the entire program's input data type. PERFGEN achieves at least 11x speedup compared to a traditional fuzzing approach when generating inputs to trigger performance symptoms in our case studies.

4 Evaluation

We evaluate PERFGEN with the following research questions:

RQ1 How much speedup in total execution time can PERFGEN achieve by phased fuzzing?

RQ2 How much reduction in the number of fuzzing iterations does PERFGEN provide through improved seeds derived from phased fuzzing?

RQ3 How much improvement in speedup is gained by PERFGEN's adjustment of mutation sampling probabilities based on the target symptom?

RQ1 assesses overall time savings in using PERFGEN, while RQ2 measures the reduction in the number of required fuzzing iterations. RQ3 explores the effects of mutation sampling probabilities on test input generation time.

Evaluation Setup. As a baseline, we compare against a simplified version of PERFGEN that does not apply phased fuzzing. This is

because existing dataflow application fuzzers such as BigFuzz [52] are unable to detect performance symptoms or monitor underlying performance characteristics of Spark programs. The baseline configuration fuzzes the original program with the same monitor template but invokes the entire program with the initial seed input. Similar to the PERFGEN setup, the baseline fuzzes the program until a skew-inducing input is identified. As pseudo-inverse functions are not tied to a specific symptom and can be potentially reused, we do not include their derivation times in our results; in practice, we found that each pseudo-inverse function definition required no more than five minutes to implement.

Each evaluation is run for up to four hours, using Spark 2.4.4's local execution mode on a single machine with 16GB RAM and 2.6 GHz 6-core Intel Core i7 processor.

4.1 Case Study: Collatz Conjecture

The *Collatz* case study is based on the description in Section 2. It parses a dataset of space-separated integers and applies a *Collatz*-sequence-based mathematical function to each integer. This case study's symptom definition differs from that in Section 2, while other details including pseudo-inverse function and generated datasets remain the same.

The developer is interested in inputs that will exhibit severe computation skew in which one outlier partition takes more than 100 times longer to compute than others due to the `solve_collatz` function. As this function is called in the transformation that produces *solved* variable, they specify *solved* as function of interest for PERFGEN's phased fuzzing. The developer defines their performance symptom by using the *Runtime* metric with an *IQROutlier* monitor template, specifying a target threshold of 100.0.

Based on Section 3.3, PERFGEN produces the following mutations and weights for *solved* variable and specified skew symptom:

Mutation Operators	Weight	Sampling %
M10 + M7 + M1	1.0	11.1%
M10 + M7 + M5 + M1	5.0	55.5%
M10 + M7 + M6	1.0	11.1%
M11	0.5	5.6%
M12	0.5	5.6%
M14	1.0	11.1%

PERFGEN begins to generate test inputs and produces the datasets shown in Figure 6. Its UDF fuzzing phase requires 3 iterations and 41 seconds. Its end-to-end fuzzing phase requires no iterations after the pseudo-inverse function is applied, as improved seeds immediately trigger the symptom of interest.

For comparison, we also evaluate the *Collatz* program under our baseline configuration and it produces a symptom-triggering input after 12,166 iterations and 15.6 min by changing the "4" record to "338", which has a *Collatz* length of 50.

Collatz evaluation results are summarized in Table 4, with the progress of *IQROutlier* feedback scores plotted in Figure 10. Compared to the baseline, PERFGEN's approach produces a 11.17X speedup and requires 0.008% of the program fuzzing iterations. Additionally, PERFGEN spends 49.14% of its total input generation time on the UDF fuzzing process. While both configurations can successfully generate inputs that trigger the performance symptom of interest, PERFGEN can do more efficiently because its type knowledge allows it to focus on generating integer inputs. The baseline is restricted to string-based mutations which often fail the integer parsing process.

4.2 Case Study: StockBuyAndSell

Suppose a developer is interested in the *StockBuyAndSell* program, which is based on the LeetCode problem [7]. Using a dataset of comma-separated strings in the form *"Symbol,Date,Open,High,Low,Close,Volume,OpenInt"*, the *StockBuyAndSell* calculates each stock's maximum achievable profit with at most three transactions (using a dynamic programming implementation adapted from [8]) by grouping closing prices by stock symbol and sorting within each symbol. As an initial dataset, the developer samples 1% of the 20 largest stock symbols from a Kaggle dataset [32]. The dataset consists of 2,389 records across 20 partitions.

The developer wishes to generate an input that triggers a symptom where one partition increases the maximum observed profit of the dynamic programming loop at least five times more frequently than other partitions. For the target UDF, they specify the RDD variable that is produced from the transformation in which this loop occurs. The developer may use Spark's Accumulator API [5] to count the number of branch executions to model an increase of the maximum observed profit for each partition. This metric is then passed to a *NextComparison* monitor template with a ratio of 5.0.

The target UDF takes *(String, Iterable[Double])* tuples as input. The developer uses their knowledge of the *StockBuyAndSell* program to disable key-based mutations for these inputs, as well as impose a restriction that UDF input keys must be unique due to an earlier aggregation. As a result, PERFGEN produces the following two mutations: M10 + M7 + M5 + M2 and M10 + M7 + M6.

Finally, PERFGEN generates a pseudo-inverse function with GPT-4. The generated function first assigns a chronological date to each price within a stock group. Next, it populates arbitrary values for unused program input fields. Lastly, it joins all values into the comma-separated string format required by *StockBuyAndSell*.

```
1  def inverse(udfInput: RDD[(String,
       Iterable[Double])]): RDD[String] = {
2    val datePrice = udfInput.flatMapValues(prices
       => {
3      prices.map(price => {
4        val dateStr = getNextDate()(dateStr,
           price)})})
5    val stringJoin = datePrice.map({
6    case (key, valueTuple) => val (date,
       price)=valueTuple
7     //
          "Symbol,Date,Open,High,Low,Close,Volume,OpenInt"
8     Seq(key, date, price, price, price, 100000,
          0).mkString(",")})
9    return stringJoin}
```

PERFGEN starts generating test inputs by first partially executing *StockBuyAndSell* on the provided input dataset to produce a UDF input consisting of stock symbols and their chronologically ordered prices. It then applies mutations to this intermediate input and, after 3.4 mins and 4,775 iterations, produces an input which satisfies the monitor template. The result is produced from a *M5* which directly affects the developer's custom metric by modifying individual values in the grouped stock prices.

Next, PERFGEN applies the pseudo-inverse function to the UDF input, tests the resulting *StockBuyAndSell* input, and finds that it also triggers the symptom of interest. As a result, no additional fuzzing iterations are necessary.

Program	UDF Fuzzing			Program Fuzzing			PerfGen Total	Baseline		PerfGen vs. Baseline		
	Seed Init. (ms)	Duration (ms)	# Iter.	P-Inv. Func. Appl. (ms)	Duration (ms)	# Iter.	Duration (ms)	Duration (ms)	# Iter.	Speedup	Iter. % Program Fuzzing	Time % Phased Fuzzing
Collatz	1,259	41,221	3	310	41,095	1	83,888	937,071	12,166	11.17	0.008%	49.14%
WordCount*	4,299	378,946	357	986	544	1	384,778	14,401,990	46,884	37.43	0.002%	98.48%
StockBuyAndSell*	1,450	205,084	4,775	601	208	1	207,346	14,402,428	40,010	69.46	0.002%	98.91%
DeptGPAsMedian_courseGpas*	2,282	259,205	1,519	736	638	1	262,864	14,405,503	21,575	54.80	0.005%	98.61%
DeptGPAsMedian_courseGpaAvgs*	2,616	251,612	1,306	636	638	1	255,502	14,405,503	21,575	56.38	0.005%	98.23%
DeptGPAsMedian_deptGpas*	1,707	271,538	2,306	107	638	1	273,990	14,405,503	21,575	52.57	0.005%	98.09%
DeptGPAsMedian_median*	881	292,555	1,387	852	638	1	294,926	14,405,503	21,575	49.02	0.005%	97.96%

Table 4: Fuzzing times and iterations for each case study program. For programs marked with a "*", the baseline evaluation timed out after 4 hours and was unsuccessful in reproducing the desired symptom.

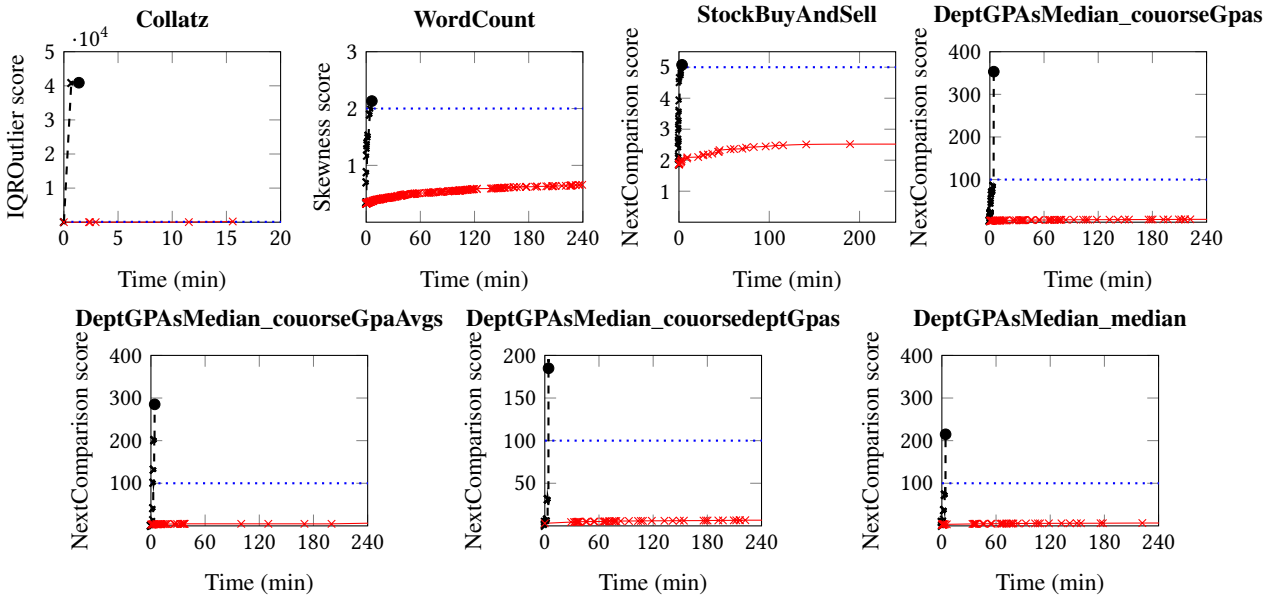

Figure 10: Time series plots of each case study's monitor template feedback score against time. PerfGen results are plotted in black with the final program result indicated by a circle, while baseline results are plotted in red crosses. The target threshold for each case study's symptom definition is represented by a horizontal blue dotted line.

For comparison, we evaluate the *StockBuyAndSell* program using the initially provided input dataset and the baseline configuration discussed at the start of Section 4. After approximately 4 hours and 40,010 iterations, no inputs trigger the symptom of interest.

StockBuyAndSell evaluation results are summarized in Table 4, with the progress of the best observed *NextComparison* ratios plotted in Figure 10. Compared to the baseline which times out after four hours, PerfGen leads to at least 69.46X speedup and requires at most 0.002% of the program fuzzing iterations. Additionally, 98.91% of PerfGen's execution time is spent on UDF fuzzing alone.

While PerfGen is able to trigger the performance symptom of interest–a *Next Comparison* ratio greater than 5.0, the baseline only reaches a ratio of approximately 2.5, indicating a substantial gap in the two approaches' effectiveness. This is because the baseline is unable to handle fields that are unused or parsed into numbers, nor is it able to significantly affect the distribution of data across each key. In contrast, PerfGen overcomes these challenges through its phased fuzzing and tailored mutations.

4.3 Improvement in RQ1 and RQ2

Table 4 presents each case study's evaluation results, and Figure 10 shows each case study's progress over time. Averaged across all four case studies,[3] PerfGen leads to a speedup of at least 43X while requiring no more than 0.004% of the program fuzzing iterations required by the baseline. Additionally, PerfGen's UDF fuzzing process accounts for an average 86.28% of its total execution time.

4.4 RQ3: Effect of Mutation Weights

Using the *DeptGPAsMedian* program, we experiment with the mutation sampling probabilities to evaluate their impact on PerfGen's ability to generate symptom-triggering inputs. As discussed in section 3.3, mutation sampling probabilities are determined by weighted random sampling. In addition to the original weight of 5.0 in the case study, we also experimented with weights of 0.1, 0.5, 1.0, 2.5, 7.5,

[3]As three of the four case study baselines timed out after four hours, numbers are reported as bounds.

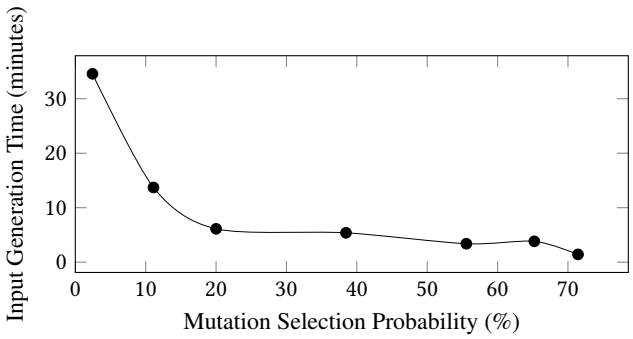

Figure 11: Plot of PERFGEN input generation time against varying sampling probabilities for the *M13* and *M14* mutations used in the *DeptGPAsMedian* program.

and 10.0, which resulted in individual mutation probabilities ranging from 2.44% to 71.43%. For each value, we average over 5 executions and report the total time required for PERFGEN to generate an input that triggers the original *DeptGPAsMedian* symptom.

Execution times for each sampling weight are plotted in Figure 11. We find that PERFGEN's template-dependent weight of 5.0 leads to a speedup of 1.81X compared to a configuration in which no extra weight is assigned (i.e., uniform weights of 1.0). More generally, we also observe that the total time required to generate a satisfying input appears to be inversely proportional to the weights of the aforementioned mutations.

In total, the range of execution times for each of the evaluated sampling weights ranged between 23.32% and 564.74% of the time taken for an unweighted evaluation. Compared to an unweighted scenario, reducing the same weights to 0.5 and 0.1 resulted in 123.97% and 464.74% increases in total time.

5 Related Work

Performance Analysis of Big Data Systems. Today, big data analytics applications are always composed of both UDF and dataflow operators. Performance analysis has been a long challenge in big data systems [22, 34, 46, 50]. Ernest [46], ARIA [47], and Jockey [17] model job performance by observing system and job characteristics. Starfish [23] constructs performance models and proposes system configurations that either meet the budget or deadline requirements. DAC [50] is a data-size aware auto-tuning approach to efficiently identify the high dimensional configuration for a given Apache Spark program to achieve optimal performance on a given cluster. It builds the performance model based on both the size of input dataset and Spark configuration parameters. Cheng et al. [14] incorporate up to 180 Spark configuration parameters to predict Spark application performance for a given application and dataset size. They do so by training Adaboost ensemble learning models to predict performance at each stage, while minimizing required training data through a data mining technique known as projective sampling.

Kwon et al. present a survey of various sources of performance skew and identify different kinds of data skews [24]. Several works aim to mitigate the performance impacts of data skew by proposing automatic skew mitigation approaches that leverage techniques such as data redistribution and adaptive repartitioning [10, 13, 25, 29, 31, 44]. Mishra et al. [33] conduct a survey of Hadoop-based data skewness mitigation techniques and categorize them based on each technique's support for map-side and reduce-side data skew.

All these performance analysis techniques assume that benchmarks provide insights about workload characteristics and performance depends on data size and cluster configurations. However, many ignore the fact that performance is often also dependent on input content. They do not auto-generate inputs to trigger performance skews. In contrast, PERFGEN automatically generates such performance workload to reproduce a desired performance symptom.

Test Generation for DISC Applications. State of the art test generation techniques for DISC applications fall into two main categories: symbolic-execution based approaches [21, 27, 36] and fuzzing-based approaches [52]. Gulzar et al. model the semantics of these operators in first-order logical specifications alongside with the symbolic representation of UDFs [21] and generate a test suite to reveal faults. Prior DISC testing approaches either do not model the UDF or only model the specifications of dataflow operators partially [27, 36]. Li et al. propose a combinatorial testing approach to bound the scope of possible input combinations [28]. All these symbolic execution approaches generate path constraints up to a given depth and are thus ineffective in generating test inputs that can lead to deep execution and trigger performance skews. To reduce fuzz testing time for dataflow-based big data applications, BigFuzz [52] rewrites dataflow APIS with executable specifications; however, its guidance metric concerns branch coverage only and thus cannot detect performance skews. Additionally, there is no guarantee that the rewritten program preserves the original DISC application's performance behaviors.

Fuzz Testing for Performance. Fuzzing has gained popularity in both academia and industry due to its black/grey box approach with a low barrier to entry [51]. For example, AFL mutates a seed input to discover previously unseen branch coverage [51]. Instead of using fuzzing for code coverage, several techniques have investigated how to adapt fuzzing for performance testing. PMFuzz [30] generates test cases to test the crash consistency guarantee of programs designed for persistent memory systems. It monitors the statistics of PM paths that consist of program statements with PM operations. PerfFuzz [26] uses the execution counts of exercised instructions as fuzzing guidance to explore pathological performance behavior. MemLock [48] employs both coverage and memory consumption metrics to guide fuzzing; however, it does not leverage a phased fuzzing approach to expedite fuzzing time and is limited to uncontrolled memory consumption bugs only.

Unlike these works, PERFGEN tackles the performance testing challenge of DISC applications. It leverages a phased fuzzing approach to expedite the test generation process, uses performance skew symptoms as a fuzzing objective, and generates inputs to replicate these symptoms through custom performance monitoring and skew-inspired mutations.

Program Synthesis for Data Transformation. Inductive program synthesis [20] learns a *program* (i.e., a procedure) from incomplete specifications such as input and output examples. FlashProfile [39] adapts this approach to the data domain, presents a novel domain-specific language (DSL) for patterns, defines a specification over a given set of strings, and learns a syntactic pattern automatically. PADS [18] provides a data description language allowing users to describe their ad-hoc data for various fields in the data and their

corresponding type. The data description is then generated automatically by an inference algorithm. Oncina et al. [37] propose a new algorithm that learns a DFA compatible with a given sample of positive and negative examples. However, a key limitation of prior work is that such data transformation synthesis is not used to speed up fuzz testing. PERFGEN addresses this gap and is the first to employ OpenAI's ChatGPT [38] to generate a pseudo inverse function to produce improved seeds to be used in the beginning of the program and thus reduce the overall fuzz testing time. This approach not only integrates program synthesis with test input generation but also utilizes the power of advanced large language models, providing a more robust and effective solution for data transformation tasks.

6 Conclusion

This paper presents PERFGEN, a tool that automatically generates inputs for reproducing a desired performance symptom in a DISC application. PERFGEN's novel phased fuzzing approach addresses the challenge of latency reduction, symptom-specific performance feedback monitoring, and skew-inspired mutations. PERFGEN marks the first instance of a tool leveraging a large language model to enhance performance testing, achieved by generating pseudo-inverse functions. Our evaluation shows that PERFGEN achieves an average speedup of at least 43X compared to traditional fuzzing approaches when generating workloads that exhibit specified performance skew symptoms. Furthermore, it does so while requiring at most 0.004% of the fuzzing iterations when phased fuzzing is disabled. Finally, our evaluation explores the impact of mutation sampling probabilities on input generation efficiency and finds that PERFGEN's template-inspired mutation probabilities produce a 1.81X speedup in input generation time compared to a uniform sampling approach.

Acknowledgments

This work is supported by the National Science Foundation under grant numbers 2426162, 2106838, 2426161, and 2106404. It is also supported in part by funding from Amazon and Samsung, and startup funding offered by Tulane University. We want to thank the anonymous reviewers for their constructive feedback that helped improve the work.

References

[1] [n. d.]. Hadoop. http://hadoop.apache.org/.
[2] [n. d.]. Interquartile_range. https://en.wikipedia.org/wiki/Interquartile_range.
[3] [n. d.]. Modified Z-Score. https://www.ibm.com/docs/en/cognos-analytics/11.1.0?topic=terms-modified-z-score.
[4] [n. d.]. Skewness. https://en.wikipedia.org/wiki/Skewness.
[5] [n. d.]. Spark Documentation. http://spark.apache.org/docs/1.2.1/.
[6] [n. d.]. Z-Score. https://en.wikipedia.org/wiki/Standard_score.
[7] 2022. https://leetcode.com/problems/best-time-to-buy-and-sell-stock-iii/.
[8] 2022. https://leetcode.com/problems/best-time-to-buy-and-sell-stock-iii/discuss/39608/A-clean-DP-solution-which-generalizes-to-k-transactions.
[9] Domagoj Babić, Stefan Bucur, Yaohui Chen, Franjo Ivančić, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. 2019. FUDGE: Fuzz Driver Generation at Scale. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 975–985. doi:10.1145/3338906.3340456
[10] Laurent Bindschaedler, Jasmina Malicevic, Nicolas Schiper, Ashvin Goel, and Willy Zwaenepoel. 2018. Rock You like a Hurricane: Taming Skew in Large Scale Analytics. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 20, 15 pages. doi:10.1145/3190508.3190532
[11] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. 2004. Myths and Realities: The Performance Impact of Garbage Collection. *SIGMETRICS Perform. Eval. Rev.* 32, 1 (June 2004), 25–36. doi:10.1145/1012888.1005693
[12] Tegan Brennan, Seemanta Saha, and Tevfik Bultan. 2020. JVM Fuzzing for JIT-Induced Side-Channel Detection. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1011–1023. doi:10.1145/3377811.3380432
[13] Qi Chen, Jinyu Yao, and Zhen Xiao. 2014. Libra: Lightweight data skew mitigation in mapreduce. *IEEE Transactions on parallel and distributed systems* 26, 9 (2014), 2520–2533.
[14] Guoli Cheng, Shi Ying, Bingming Wang, and Yuhang Li. 2021. Efficient performance prediction for apache spark. *J. Parallel and Distrib. Comput.* 149 (2021), 40–51.
[15] Joeri De Ruiter and Erik Poll. 2015. Protocol State Fuzzing of TLS Implementations. In *Proceedings of the 24th USENIX Conference on Security Symposium* (Washington, D.C.) *(SEC'15)*. USENIX Association, USA, 193–206.
[16] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. doi:10.1145/1327452.1327492
[17] Andrew D. Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. 2012. Jockey: Guaranteed Job Latency in Data Parallel Clusters. In *Proceedings of the 7th ACM European Conference on Computer Systems* (Bern, Switzerland) *(EuroSys '12)*. ACM, New York, NY, USA, 99–112. doi:10.1145/2168836.2168847
[18] Kathleen Fisher and David Walker. 2011. The PADS Project: An Overview. In *Proceedings of the 14th International Conference on Database Theory* (Uppsala, Sweden) *(ICDT '11)*. ACM, New York, NY, USA, 11–17. doi:10.1145/1938551.1938556
[19] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen. 2018. CollAFL: Path Sensitive Fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*. 679–696. doi:10.1109/SP.2018.00040
[20] Sumit Gulwani. 2010. Dimensions in Program Synthesis. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming* (Hagenberg, Austria) *(PPDP '10)*. Association for Computing Machinery, New York, NY, USA, 13–24. doi:10.1145/1836089.1836091
[21] Muhammad Ali Gulzar, Madanlal Musuvathi, and Miryung Kim. 2020. BigTest: A Symbolic Execution Based Systematic Test Generation Tool for Apache Spark. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 61–64. doi:10.1145/3377812.3382145
[22] Carson Hanel, Arif Arman, Di Xiao, John Keech, and Dmitri Loguinov. 2020. Vortex: Extreme-Performance Memory Abstractions for Data-Intensive Streaming Applications. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 623–638.
[23] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. 2011. Starfish: A Self-tuning System for Big Data Analytics. In *In CIDR*. 261–272.
[24] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. 2011. A study of skew in mapreduce applications. *Open Cirrus Summit 11* (2011).
[25] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. 2012. SkewTune: Mitigating Skew in Mapreduce Applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. ACM, New York, NY, USA, 25–36. doi:10.1145/2213836.2213840
[26] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. Perffuzz: Automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 254–265.
[27] Kaituo Li, Christoph Reichenbach, Yannis Smaragdakis, Yanlei Diao, and Christoph Csallner. 2013. SEDGE: Symbolic example data generation for dataflow programs. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE, 235–245.
[28] Nan Li, Yu Lei, Haider Riaz Khan, Jingshu Liu, and Yun Guo. 2016. Applying Combinatorial Test Data Generation to Big Data Applications. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE 2016)*. Association for Computing Machinery, New York, NY, USA, 637–647. doi:10.1145/2970276.2970325
[29] Guipeng Liu, Xiaomin Zhu, Ji Wang, Deke Guo, Weidong Bao, and Hui Guo. 2018. SP-Partitioner: A novel partition method to handle intermediate data skew in spark streaming. *Future Generation Computer Systems* 86 (2018), 1054–1063. doi:10.1016/j.future.2017.07.014
[30] Sihang Liu, Suyash Mahar, Baishakhi Ray, and Samira Khan. 2021. PMFuzz: test case generation for persistent memory programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 487–502.

[31] Zhihong Liu, Qi Zhang, Mohamed Faten Zhani, Raouf Boutaba, Yaping Liu, and Zhenghu Gong. 2015. Dreams: Dynamic resource allocation for mapreduce with data skew. In *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. IEEE, 18–26.

[32] Boris Marjanovic. 2017. Huge Stock Market Dataset | Kaggle. https://www.kaggle.com/borismarjanovic/price-volume-data-for-all-us-stocks-etfs

[33] Subhankar Mishra, Namita Sethi, and Ayes Chinmay. 2019. Various Data Skewness Methods in the Hadoop Environment. In *2019 International Conference on Recent Advances in Energy-efficient Computing and Communication (ICRAECC)*. 1–4. doi:10.1109/ICRAECC43874.2019.8994979

[34] Khanh Nguyen, Lu Fang, Christian Navasca, Guoqing Xu, Brian Demsky, and Shan Lu. 2018. Skyway: Connecting managed heaps in distributed big data systems. *ACM SIGPLAN Notices* 53, 2 (2018), 56–69.

[35] Yannic Noller, Rody Kersten, and Corina S. Păsăreanu. 2018. Badger: Complexity Analysis with Fuzzing and Symbolic Execution. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 322–332. doi:10.1145/3213846.3213868

[36] Christopher Olston, Shubham Chopra, and Utkarsh Srivastava. 2009. Generating Example Data for Dataflow Programs. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data* (Providence, Rhode Island, USA) *(SIGMOD '09)*. ACM, New York, NY, USA, 245–256. doi:10.1145/1559845.1559873

[37] Jose Oncina and Pedro Garcia. 1992. Identifying Regular Languages In Polynomial Time. In *ADVANCES IN STRUCTURAL AND SYNTACTIC PATTERN RECOGNITION, VOLUME 5 OF SERIES IN MACHINE PERCEPTION AND ARTIFICIAL INTELLIGENCE*. World Scientific, 99–108.

[38] OpenAI. 2025. OpenAI GPT-5 System Card. arXiv:2601.03267 [cs.CL]

[39] Saswat Padhi, Prateek Jain, Daniel Perelman, Oleksandr Polozov, Sumit Gulwani, and Todd D Millstein. 2017. FlashProfile: Interactive Synthesis of Syntactic Profiles. *CoRR* (2017).

[40] Rohan Padhye, Caroline Lemieux, and Koushik Sen. 2019. JQF: Coverage-Guided Property-Based Testing in Java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 398–401. doi:10.1145/3293882.3339002

[41] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic Fuzzing with Zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 329–340. doi:10.1145/3293882.3330576

[42] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. 2017. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2155–2168. doi:10.1145/3133956.3134073

[43] Juraj Somorovsky. 2016. Systematic Fuzzing and Testing of TLS Libraries. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 1492–1504. doi:10.1145/2976749.2978411

[44] Zhuo Tang, Wei Lv, Kenli Li, and Keqin Li. 2021. An Intermediate Data Partition Algorithm for Skew Mitigation in Spark Computing Environment. *IEEE Transactions on Cloud Computing* 9, 2 (2021), 461–474. doi:10.1109/TCC.2018.2878838

[45] Jason Teoh, Muhammad Ali Gulzar, Guoqing Harry Xu, and Miryung Kim. 2019. PerfDebug: Performance Debugging of Computation Skew in Dataflow Systems. In *Proceedings of the ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '19)*. Association for Computing Machinery, New York, NY, USA, 465–476. doi:10.1145/3357223.3362727

[46] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. 2016. Ernest: Efficient Performance Prediction for Large-scale Advanced Analytics. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation* (Santa Clara, CA) *(NSDI'16)*. USENIX Association, Berkeley, CA, USA, 363–378. http://dl.acm.org/citation.cfm?id=2930611.2930635

[47] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. 2011. ARIA: Automatic Resource Inference and Allocation for Mapreduce Environments. In *Proceedings of the 8th ACM International Conference on Autonomic Computing* (Karlsruhe, Germany) *(ICAC '11)*. ACM, New York, NY, USA, 235–244. doi:10.1145/1998582.1998637

[48] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yichang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. 2020. MemLock: Memory Usage Guided Fuzzing. In *ICSE 2020* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 765–777. doi:10.1145/3377811.3380396

[49] Hui Xu, Zirui Zhao, Yangfan Zhou, and Michael R. Lyu. 2020. Benchmarking the Capability of Symbolic Execution Tools with Logic Bombs. *IEEE Transactions on Dependable and Secure Computing* 17, 6 (2020), 1243–1256. doi:10.1109/TDSC.2018.2866469

[50] Zhibin Yu, Zhendong Bei, and Xuehai Qian. 2018. Datasize-aware high dimensional configurations auto-tuning of in-memory cluster computing. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. 564–577.

[51] Michał Zalewski. 2021. American Fuzz Loop. http://lcamtuf.coredump.cx/afl/.

[52] Qian Zhang, Jiyuan Wang, Muhammad Ali Gulzar, Rohan Padhye, and Miryung Kim. 2020. BigFuzz: Efficient Fuzz Testing for Data Analytics using Framework Abstraction. In *The 35th IEEE/ACM International Conference on Automated Software Engineering*. doi:10.1145/3324884.3416641