DiSh: Dynamic Shell-Script Distribution

Tammam Mustafa
MIT

Konstantinos Kallas
University of Pennsylvania

Pratyush Das
Purdue University

Nikos Vasilakis
Brown University

Abstract

Shell scripting remains prevalent for automation and data-processing tasks, partly due to its dynamic features—*e.g.*, expansion, substitution—and language agnosticism—*i.e.*, the ability to combine third-party commands available in any programming language. Unfortunately, these characteristics hinder automated shell-script distribution, often necessary for dealing with large datasets that do not fit in a single computer. This paper introduces DISH, a system distributing the execution of dynamic shell scripts operating on distributed filesystems. DISH is designed as a shim that applies program analyses and transformations to leverage distributed computing, while delegating all execution to the underlying shell available on each computing node. As a result, DISH avoids modifications to shell scripts and maintains compatibility with existing shells and legacy functionality. We evaluate DISH against several options available to users today: (i) Bash, a single-node shell-interpreter baseline, (ii) PASH, a state-of-the-art automated-parallelization system, and (iii) Hadoop Streaming, a MapReduce system that supports language-agnostic third-party components. Combined, our results demonstrate that DISH offers significant performance gains, requires no developer effort, and handles arbitrary dynamic behaviors pervasive in shell scripts.

1 Introduction

Unix and Linux shell scripting remains prevalent—8th most popular language on GitHub [16]—for data processing, system orchestration, and other automation tasks. Part of this prevalence can be attributed to a unique combination of features: (1) powerful and language-agnostic primitives for composing components available in any programming language; (2) dynamic features such as command substitution, variable expansion, and state reflection on the file system; and (3) a wide range of useful components called commands, available in the broader environment and tailored to specific tasks. These features enable succinct and concise program composition on a single computer (§2).

Tab. 1: Available options for scaling out shell programs. Compatibility: support existing shell scripts without modifications. Granularity: support for fine-grained distribution. Expressiveness: support arbitrary behaviors, including the dynamic ones supported in the shell. Agnosticism: support components written in any programming language. Correctness: behavior equivalence with existing shells.

Approach	Compatibility	Granularity	Expressiveness	Agnosticism	Correctness	Examples
Distributed Shells	□	■	■	■	□	[11, 14, 57]
POSH	■	■	□	■	□	[43]
Cluster Comp. Frameworks (CCF)	□		□	□	□	[38, 51, 62, 65]
Language-agnostic CCFs	□		□	■	■	[21, 25]
Job Scheduling Tools	■	□	□	■	■	[15, 24, 52, 63]
Other languages	□	■	■	□	□	[13, 54, 60]
DISH	■	■	■	■	■	

Unfortunately, these features also hinder automated shell-script scale-out to multiple computers. Such scale-out is often necessary not only to accelerate computations, but also to compute over data that either do not fit on a single computer or are naturally distributed across multiple computers.

State of the art: Shell users dealing with large datasets that do not fit in a single computer are left with only a few options (see §8 for details). One option is to use a distributed shell [11, 14, 57]. Distributed shells require rewriting scripts manually and only support a small subset of UNIX features—often with limited, if any, dynamic features and varying support for composition constructs. A recent distributed shell named POSH [43] can handle a subset of shell scripts without rewriting—although that subset is limited to dataflow-only computations and also does not include arbitrary dynamic shell behaviors. In addition, since POSH is a shell reimplementation, it is not behaviorally equivalent with existing shells and thus risks breaking ported scripts. A second option is to rewrite (parts of) the script in a cluster-computing framework [8,38,62,65]. These support only computational subsets (*e.g.*, batch, stream) of the shell, require manual rewriting, and only rarely [21,25] support language-agnostic components. Another option is job scheduling tools [15,24,52,63], but these operate at a coarse granularity and usually do not leverage parallelism available in a single script. Yet another option is to rewrite scripts in languages that support distribution [1,34,36,60], foregoing the shell's succinctness and language agnosticism. Summarizing in Tab. 1, these options operate on a subset of the shell, require significant manual effort, risk of breaking correctness, or—most often—suffer from a combination of these limitations.

Dynamic shell-script distribution: This paper presents DISH, a system designed to scale out shell scripts operating on distributed filesystems while maintaining full POSIX compatibility. DISH satisfies all requirements in Table 1: it operates on existing shell scripts without manual rewriting; it distributes scripts at a fine granularity of individual commands; it handles arbitrary dynamic shell features such as substitution and expansion; it allows the use of commands and utilities of any language; and, most importantly, it is behaviorally equivalent to Bash.

Given a script to distribute, DISH orchestrates the script and identifies regions that may benefit from distribution. At runtime, it compiles these regions to an intermediate representation which it then optimizes to introduce appropriate parallelism, buffering, communication, and coordination. DISH then executes each compiled region in a distributed fashion using the same shell interpreter, components, and data as the original script.

Implementation and results: DISH is implemented as a shim layer (rather than a shell) that wraps and orchestrates the (completely unmodified) user shell, delegating all execution to the underlying shell available on each computing node. This design hides parallelization and distribution from the user without modifying the underlying shell interpreter: the user thinks that their original script is being executed (but faster), and each underlying shell is given a part of the distributed script to execute. As a result, DISH achieves a new milestone in automated shell-script distribution: it offers significant performance benefits, it avoids modifications to shell scripts, and it maintains full POSIX compatibility. Additionally, this modular design allows further research and improvements without modifications in the underlying shell.

We characterize DISH's performance on a 4-node on-premise cluster and a 20-node cloud deployment using 76 scripts—including ones not trivially expressible in modern distributed computing frameworks, such as scripts with `for` loops, side-effects, and complex third-party components. DISH surpasses the speedups achieved by production-grade systems on existing benchmarks and extends speedups to new ones: it accelerates significantly (1) over Bash (avg: 13.6×; max: 136.3×), a single-node shell-interpreter baseline; (2) over PASH (avg: 8.9×; max: 108.8×), a shell-script parallelization system; and (3) over Hadoop Streaming (avg: 7.2×; max: 32.3×), a cluster computing framework that supports language-agnostic components and shell scripts. Moreover, whereas Hadoop Streaming does not support 27/76 scripts and requires rewriting 7/76 scripts, DISH runs all scripts without any modifications; in fact, DISH is able to execute the entire POSIX shell test suite, only diverging in one error code out of thousands of assertions.

Paper outline and contributions: The paper begins with an example and overview (§2) of DISH's use and techniques. Sections 3–6 present DISH's key components:

- Dynamic orchestration (§3): DISH parses, pre-processes, expands, and orchestrates its input script to enable dynamic distribution at runtime.
- Compilation (§4): During script execution, DISH compiles certain regions to an intermediate representation and applies a series of optimizations.
- Distribution (§5): DISH distributes each region to a set of workers in a way that promotes co-location of processing primitives and the data blocks they process.
- Runtime support (§6): DISH bundles additional runtime primitives supporting correct and efficient communication in the context of distributed shell script execution.

The paper then presents DISH's evaluation (§7) and related work (§8), before concluding (§9).

DISH limitations: DISH currently does not tolerate failures such as worker aborts or network partitions. In such occasions, users are expected to rerun their scripts similar to how they do in non-distributed executions: due to the shell's dynamic features and its support for third-party components, users often re-run failing scripts from start. The current DISH prototype additionally does not implement support for security features such as encryption and containment.

Availability: All the work described in this paper has been implemented and incorporated into PASH—an MIT-licensed project available by the Linux Foundation at https://github.com/binpash/.

2 Background, Example, and Overview

DISH allows everyday shell scripts to reap the benefits of distributed computing: execute on data that would not fit into any single machine, often also speeding up expensive computations.

Intended use: DISH is designed to support a variety of use cases, depending on the details of the distributed environment on which the system is executing. The most common case is one where input data are downloaded and stored in a distributed file system such as HDFS and then processed using various analyses. This is useful for datasets that do not fit on a single computer, that are naturally distributed across multiple computers, or that can be processed faster in a data-parallel fashion. DiSh will distribute the computation appropriately, often running data-parallel instances on multiple machines and multiple processors per machine. DiSh also supports distributed operation where data resides on both distributed and local file systems; this is useful for computations that contain CPU-intensive stages over datasets that do not necessarily reside on distributed file systems.

Example script: Fig. 1 shows a shell script that calculates maximum and average temperatures across the US, on datasets hosted on the National Oceanic and Atmospheric Administration (NOAA). The script is split into three parts: (p. 1) an 11-stage pre-processing pipeline to download data from NOAA and store them on HDFS, with the data range controlled upon invocation via dynamic arguments `$1` and `$2`; (p. 2, 3) two 5-stage pipelines calculating and storing maximum and average temperatures to the local file system.

HDFS is a distributed file system for handling large data sets on commodity hardware. Scripts like the one in Fig. 1 that process files stored in distributed file systems spend most of their execution time moving files across the network. In a 4-node cluster (§7) and with 3.6GB of input, running just `hdfs dfs -cat` takes 1020s; computing pipeline 2 (maximum temperature) adds only 17s. This phenomenon is due to pipeline parallelism: the execution time of all concurrently executing commands is mostly shadowed by `hdfs dfs -cat`.

Opportunities for scale-out: There are ample opportunities for improving the performance of this script. Since all parts contain stages that operate on large datasets, we should be able to execute (at least some of) their stages in a data-parallel fashion. For example, we should parallelize commands such as `cut` and `grep` that process their input independently by having them operate in parallel over partial inputs.

Additionally, carefully colocating computation and data could also improve performance. For example, we should schedule the data-parallel execution of the aforementioned `cut` and `grep` instances on machines that store the data segments used by each instance. Directly operating on distributed file segments, rather than gathering and processing data on a subset of the machines, eliminates most data-movement overheads.

Finally, the execution of program fragments that do not depend on each other could become concurrent: since parts 2 and 3 are independent on each other, we should be able to overlap their execution in a task-parallel fashion.

Key challenges: Unfortunately, exploiting these opportunities to scale out execution automatically is particularly challenging in the context of the shell. First, exposing opportunities at the level of individual commands such as `cut` and `grep` is challenging—and this is why prior systems often focused on coarser, script-level or job-level granularity [15,63].

```
NOAA=${NOAA:-http://ndr.md/data/noaa/}
TEMPS=${TEMPS:-/noaa/temps.txt}
hdfs dfs -mkdir /noaa

## Pipeline 1: Download temperature data
##             and store to HDFS
seq $1 $2 | sed "s;^;$NOAA;" |
 sed 's;$;/;' | xargs -r -n 1 curl -s | grep gz |
 tr -s ' \n' | cut -d ' ' -f9 |
 sed 's;^\(.*\)\(20[0-9][0-9]\).gz;\2/\1\2\.gz;' |
 sed "s;^;$NOAA;" | xargs -n1 curl -s |
 gunzip | hdfs dfs -put - $TEMPS

## Pipeline 2: Compute maximum temperature
##             over all data
hdfs dfs -cat $TEMPS | cut -c 89-92 | grep -v 999 |
 sort -rn | head -n1 > max.txt

## Pipeline 3: Compute average temperature
##             over all data
hdfs dfs -cat $TEMPS | cut -c 89-92 | grep -v 999 |
 awk "{ t += \$1; i++ } END { print t/i }" > avg.txt
```

Fig. 1: Example script. Downloading a temperature dataset, storing on a distributed file system, and running analysis to extract statistics.

Second, pervasive dynamic features, file-system introspection, and other side-effects impede traditional distribution approaches based on static transformation—this is why prior shell-script distribution work [21,43] focuses on pure dataflow subsets. These challenges are compounded by the presence of more elaborate control flow such as **`for`** loops, `break`, and `trap` statements present in ordinary shell scripts.

Third, behavioral equivalence with existing shells is practically unattainable, especially with shell reimplementations—after all, even production-grade shells such as Bash and zsh diverge subtly in their POSIX behavior [19]. A new distributed shell [11,43] has little hope of automatically exploiting scale-out opportunities without breaking scripts in subtle ways.

DiSh overview: To overcome these challenges, DiSh is designed to dynamically orchestrate, compile, schedule, and support the execution of shell scripts (Fig. 2). DiSh orchestrates user scripts so that the underlying shell executes a distributed version of the user script (§3). This orchestration saves a snapshot of the user's shell environment (variables, configuration) and invokes the DiSh compiler with candidate script regions (Fig. 2a). The compiler takes these regions and, if possible, translates them to dataflow graphs—which it then optimizes to introduce parallelization, buffering, *etc.* (§4). The compiler either determines that a region is pure (Fig. 2b), applying transformations, expanding HDFS paths, and generating a parallel dataflow graph; or aborts compilation (Fig. 2e) because it cannot guarantee purity. The scheduler (§5) divides the compiled dataflow graph into different subgraphs which it sends

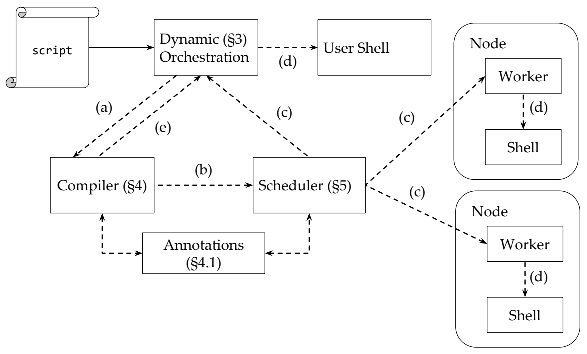

Fig. 2: DISH architecture overview. DISH is designed to dynamically orchestrate, compile, schedule, and support the execution of shell scripts.

to available cluster workers (Fig. 2c). In response to these execution requests (Fig. 2d), workers apply a second pass of optimizations to better utilize available resources, translate the dataflow graph back to a shell script, offer additional runtime support, and execute it using the local, unmodified shell interpreter (§6).

Applying DISH: DISH preprocesses the script in Fig. 1 to identify script regions that could benefit from distribution—in this case, all three pipelines. It then replaces each of these regions with calls to the dynamic orchestrator, which will attempt to distribute them at runtime. During execution, the orchestrator queries the DISH compiler to determine whether a region is pure and thus distributable: if the compiler succeeds, it translates the region to a dataflow graph. Since regions contain arbitrary black-box commands, DISH cannot analyze them directly. Instead, it employs a command specification framework that contains partial specifications of command invocations such as their inputs and outputs. For example, DISH's compiler uses these specifications to determine that `hdfs dfs -cat /noaa/temps.txt` reads from the HDFS file `/noaa/temps.txt` and writes to `stdout`. Once a region is in dataflow form, DISH applies transformations to distribute it.

Fig. 3 shows the distribution stages for pipeline 2 (maximum temperature). DISH first detects operations on HDFS files (*i.e.*, HDFS `cat`) and expands each distributed file to its segments (datablocks), often stored on different physical machines. Informed by command specifications, DISH applies parallelization transformations: commands like `cut` and `grep` are parallelizable directly and can be executed on the machine with the raw input datablock. The scheduler then splits the compiled graph into subgraphs and maps them to workers in a data-aware fashion. Finally, each worker translates the graph back to a shell script, adds additional runtime primitives (commands), and executes it locally.

The result? DISH drops the execution of pipeline 2 from 1037s to 6s (Fig. 1 in §7) while maintaining full behavioral equivalence and requiring no modifications to the user shell.

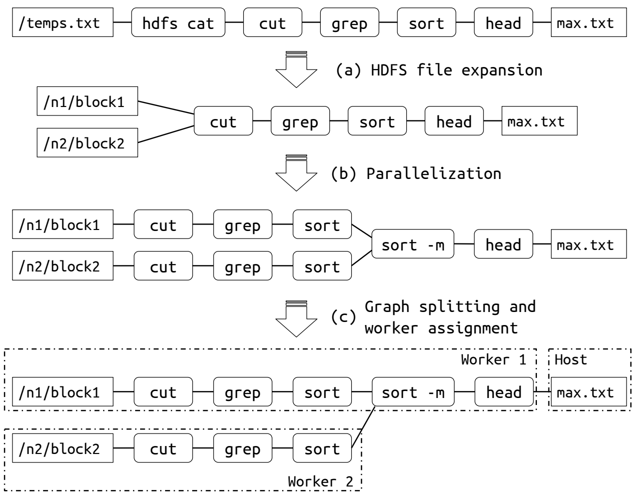

Fig. 3: DISH dataflow graph stages. (a) HDFS files are expanded to sequences of blocks. (b) the graph is parallelized based on the command specifications. (c) the scheduler splits the graph and assigns subgraphs to workers.

3 Dynamic Shell Orchestrator

An important desideratum in the design of DISH is to avoid requiring any modifications to the user shell. DISH is thus not designed to operate as another shell, but rather wraps the user's existing shell interpreter and the shell interpreters on the worker machines orchestrating their execution. As a result, DISH hides parallelization and distribution from both the user and the underlying shells: the user thinks that what is being executed is the original script—it just executes faster—and each underlying shell simply executes a standard non-distributed shell script. This allows DISH to achieve exceedingly high compatibility with the underlying shell implementation (§7.3), while also minimizing maintenance costs since updates and modifications on the underlying shell are reflected in DISH without any change.

Fig. 4 shows an overview of the structure of DISH's dynamic orchestration. To achieve dynamic shell script orchestration without any shell-interpreter modification, DISH opts for a light-weight script instrumentation pre-processing step: it instruments *potentially* distributable regions with invocations to the orchestration engine. It chooses regions with the goal of maximizing distribution benefits: intuitively, it focuses on commands and pipelines rather than control flow statements and variable assignments. However, the choice of these region boundaries is not binding—the orchestrator just needs to be precise enough to determine potential regions, but DISH will eventually decide whether or not (and if yes, how) to distribute a potentially distributable region at runtime. The orchestrator first parses the original script, it then replaces the relevant program regions with orchestration prefixes, and then un-parses (emits) it back as an instrumented script that is given for execution to the user's shell interpreter.

The instrumented script then makes calls to the orchestration engine. The orchestration engine is itself a shell script coordinating with the compiler and worker manager and attempting to parallelize the upcoming region (see §4 and 5 for a detailed description of parallelization and distribution). If it succeeds, it runs the distributed version of the region. If it aborts, it just falls back to the original region, executing it normally. Reasons for aborting include the region being side-effectful, *e.g.*, modifying some environment variable, or lacking relevant annotations.

Preprocessor: The preprocessor searches for maximal potentially distributable regions by processing the AST bottom-up, combining distributable subtrees when they are composed using constructs that do not introduce scheduling constraints (*e.g.*, `&`, `|`). When a region cannot outgrow a certain subtree, DISH replaces it with a call to the orchestrator. If successfully compiled, this region will be transformed to a dataflow graph—a convenient and well-studied computation model amenable to transformation-based optimizations [22]. The instrumented AST resulting from the compilation is finally translated (unparsed) back to shell code and sent over to the underlying shell for execution.

Parsing library: DISH invokes parsing and unparsing routines frequently, and therefore needs them to be very efficient. To that end, it uses an internal Python implementation of POSIX-shell-script parsing and unparsing based on `libdash` [18, 19]. The DISH parser contains several optimizations such as caching, inlining, and careful array appending to achieve improved performance.

Orchestration engine: DISH's orchestration engine is designed to maintain the original script behavior and minimize runtime overhead—as it is invoked multiple times per script. The engine is a reflective shell script: it coordinates transparently with the compiler to determine whether or not to parallelize a script by inspecting the state of the shell and that of the broader system. DISH constantly switches between two execution modes when executing scripts: (1) conventional shell mode, where scripts execute in the original shell context, and (2) DISH mode, where the runtime reflects on shell state and invokes a compiler to determine whether to execute the original or an optimized version of the target region. To switch from shell mode to DISH mode, the engine saves the state of the user's shell; to switch back, it restores the state of the user's shell. The state of a shell is quite complex: apart from saving and restoring variables, DISH must account for various shell flags along with other internal shell state (*e.g.*, the previous exit status, working directory). The engine first switches to DISH mode, communicates with the compiler and scheduler to determine whether a region can be safely distributed, and it then switches back to shell mode to execute the original or distributed version of the script.

Environment sharing: The distributed version of the script

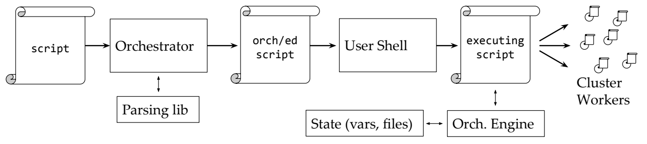

Fig. 4: Dynamic orchestration overview. DISH instruments scripts with calls to the orchestration engine, which passes program fragments to the worker manager at run-time.

region might execute on a different shell (or even machine). Therefore, a challenge that DISH needs to address is to make sure that all regions execute in the correct environment—including access to the latest variable values and function definitions. To achieve that the engine takes a snapshot of the environment right before execution. It then transfers the snapshot to the distributed workers, which they can load to execute the incoming script fragment in the correct environment. This is safe to do, since successful distribution of a region implies that it is pure (and therefore does not affect the environment), and thus the snapshot will be valid until the region finishes execution.

String expansion: In order for the compiler to correctly determine if a script is safe to distribute, all strings in a script region need to be completely expanded. Since DISH performs distribution dynamically, at runtime, it has access to all the latest variables and system state to perform expansion correctly but early, so that it passes a fully expanded script to the compiler. DISH only needs to implement a pure subset of all available expansions, and avoids implementing expansions that have the risk of affecting the environment. DISH's expansion takes the host shell's configuration and expands common, safe expansions in as many positions as possible—in simple commands, pipelines, and other parallelizable regions.

4 Compiler

This section describes the compiler of DISH, which builds on the PASH parallelizing compiler [58]. The compiler is given the AST of an input script fragment and information about the commands in that fragment (§4.1). It then attempts to transform it to a dataflow graph (§4.2), an intermediate representation amenable to parallelization transformations. If the compiler succeeds in transforming a script region to a parallel dataflow graph, that graph is then passed to the scheduler which then decides how to map subgraph components to the available worker nodes. As the compiler operates at runtime, it exploits ample opportunities for parallelization even across subgraphs (§4.3).

4.1 Command Annotations

DiSh needs to support analyses and transformations over third-party commands, without access to their source code. To achieve this, DiSh uses annotations á la PaSh [58] and POSH [43], capturing information about a command invocation's parallelizability class, inputs, and outputs.

A command annotation in DiSh encodes information at the level of individual command invocations, *i.e.*, precise instantiations of a command's flags, options, and arguments. Among other information, annotations determine whether a specific command invocation is pure and what its inputs and outputs are—information which DiSh uses when translating commands to and from dataflow nodes (§4.2). For example, the annotation for `grep` can be used to extract that the script fragment `grep -f dict.txt src.txt > out.txt` contains two input files `dict.txt` and `src.txt` and one output file `out.txt`. This knowledge of input and output files is used by DiSh to enable location-aware distribution, by scheduling the computation on nodes that contain relevant data blocks. Additionally, annotations describe parallelization opportunities—*e.g.*, that `grep "pattern" src.txt` processes each line of `src.txt` independently, and thus it can be parallelized at a line boundary.

4.2 Dataflow Model

The core of DiSh's compiler is an order-aware dataflow model that captures pure shell script regions that read from a well-defined set of input files and write to a well-defined set of output files—*i.e.*, they do not modify their environment in any other way. This model is expressive enough to capture a shell subset used pervasively in data processing scripts [22].

In this model, nodes represent commands and edges represent files, pipes, named FIFOs, and file descriptors. The model is order-aware in the sense that it keeps information about the order in which nodes read from their inputs, which is important for the script's semantics. For example, `grep "pattern" in1.txt - in2.txt` first reads from `in1.txt`, then from its standard input, and then from `in2.txt`. This order awareness allows DiSh to perform transformations that optimize execution of a script, *e.g.*, by exposing parallelism, but preserve its original behavior.

Translation workflow: Given an AST representation of an input script region, the compiler uses annotations to deduce whether commands are pure and attempts to transform them to dataflow nodes. If all commands in the region are pure, *i.e.*, they only affect their environment through a well-defined set of output files, then the compiler successfully transforms the region to a dataflow graph. It then applies transformations (described below), optimizing the graph to expose parallelism and improve the script's performance. Finally, it serializes the graph back to a (now optimized) shell script, by translating every node back to a command and connecting them all together with named FIFOs and the necessary redirections.

Transformations: DiSh's transformations duplicate nodes in the graph for which it is possible to enable data-parallel execution. The transformations also add appropriate split and merge nodes before and after the duplicated nodes, They apply a pass over the graph to remove pairs of opposing nodes—*i.e.*, pairs of nodes whose semantic effects cancel out but whose performance effects are additive—for example, a concatenation-style merge followed by a linear split. For commutative commands, *i.e.*, commands that produce the same output irregardless of their input-line order, DiSh applies transformations that pack and unpack metadata across the graph—achieving better performance by avoiding unnecessary blocking and buffering. Finally, to improve the flow of data across the graph, DiSh applies additional transformations that inject hybrid memory-disk buffer nodes in points in the graph that are likely to become bottlenecks.

Remote file resources and HDFS files: To support scripts that perform data analysis on a combination of HDFS and local files, DiSh extends the dataflow model with remote-file resources (RFRs) that encode file blocks in different nodes. RFRs usually represent blocks of files that are partitioned and replicated in HDFS, and contain information about the location of the data in the distributed environment. This information supports multiple locations for the cases of replication, and is used by the scheduler to assign script fragments to different workers. When the DiSh compiler comes across an HDFS file path, it queries HDFS to determine the locations of its file blocks and then expands that file to a sequence of RFRs, each of which represents a block.

4.3 Dynamic Dependency Untangling (DDU)

Scripts often contain fragments that are independent, *i.e.*, they have different file working sets. Independent fragments could in theory run in parallel, better utilizing computational resources and improving the execution times of the scripts in which they belong. However, inferring independence statically and ahead of time is challenging as shell scripts make extensive use of dynamic features.

Figure 5 shows an example script that contains independent fragments but also features dynamic behavior. This script iterates over all files in an HDFS directory, compresses them using `gzip`, and finally stores them as independent files.

Determining independence statically in this script would require inferring values of environment variables (like `IN` and `OUT`) and the state of the file system, *e.g.*, `hdfs dfs -ls`. DiSh's dynamic orchestration (§3) circumvents this challenge by making distribution decisions during the execution of the script when environment variables and the file system state are known. DiSh discovers independent dataflow regions at runtime and executes them in parallel—even if they were not parallel in the original script.

When DiSh successfully compiles a dataflow region, it knows that the region is pure and thus reads and writes only

```
for item in $(hdfs dfs -ls -C ${IN});
do
    output_name=$(basename $item).zip
    hdfs dfs -cat $item |
        gzip -c > $OUT/$output_name
done
```

Fig. 5: Example of independent fragments. This shell script compresses all files in a directory—but each iteration results in an independent body fragment that can be executed in parallel.

certain input and output edges. Since the region is pure, DISH has access to the region's inputs and outputs—and it does so for free, without additional analysis or inference stages. DISH then uses this information to check for read-write or write-write dependency conflicts with regions that are running concurrently. If none is found, DISH passes the dataflow fragment to the scheduler which then orchestrates distributed execution. Whenever the compilation of a dataflow region fails, it means that DISH cannot safely detect the input and output information of this region—and thus it needs to wait until every previous region is done executing.

DDU is precise and gives significant benefits due to improved parallelism and resource utilization—especially for scripts that do not contain highly data-parallelizable commands, such as the commands in the aforementioned compression script (Fig. 5).

5 Distributed Scheduling

This section describes how DISH's scheduler distributes a compiled script to a set of workers. The scheduler is given a dataflow graph that is already parallelized with the HDFS files expanded to sequences of remote file resources (RFRs) representing their blocks. The task of the scheduler is then to distribute this graph with the goal of optimizing performance by both utilizing available resources, and moving computation close to the data to reduce network overhead. Currently the scheduler learns about the workers in the cluster ahead of time, using a configuration file, but there is nothing that prohibits a dynamically changing cluster where workers could be added or removed based on resource availability.

The scheduler makes a decision on how to split the graph based on a policy that optimizes performance through colocation of data blocks and the commands that execution over them. The scheduler processes the top-level dataflow graph to generate a set of subgraphs, one for each worker and one for the machine that the script was executed (the host). It then replaces edges corresponding to communication channels, *i.e.*, named FIFOs or pipes, at the boundaries of each subgraph with remote channel—adding a remote write node on the sender side and a remote read node on the receiver side (see Fig. 6, Top). It also inserts remote reads for subgraph nodes that access files stored on remote workers. The final generated subgraph represents the script fragment that is passed for execution to the user shell running on each worker: the compiled script handles all the redirection to and from local files and the standard input, output, and error streams to and from the worker.

Data-aware scheduling policy: The highest performance overhead when executing distributed shell scripts is networked data movement between workers. DISH addresses this overhead by introducing a greedy scheduling policy that allocates subgraphs in a way that attempts to minimize data movement across workers. If a data file (or block) is available on a worker, then DISH maps the maximal dataflow subgraph that starts from that file to that worker—*i.e.*, scheduling as much of the processing as possible on the worker. The scheduler also tracks the amount of work that each worker currently has scheduled, which can vary due to dynamic dependency untangling (§4.3): if a data file is replicated across multiple workers, DISH chooses the worker with the least amount of pending work to execute that subgraph.

Worker-first aggregation: The distributed dataflow graphs that DISH executes often contain aggregation (*i.e.*, merge) nodes, similarly to the reduce stages in Hadoop Streaming. Regardless of the worker on which the aggregation is performed, data from different workers will need to be combined onto a single worker and thus these dataflow nodes will necessarily result in data movement. DISH prioritizes performing aggregation on one of the participating workers, because workers already contain a subset of the data used in the aggregation (see Fig. 6, Mid). This optimization is particularly beneficial for scripts that filter and aggregate data, *i.e.*, containing commands such as `grep` and `uniq`, because any filtering stages prior to aggregation result in fewer data to be transferred during aggregation.

It is worth noting that, absent additional information about commands [43], the location of aggregators involves challenging trade-offs not addressable with a single optimization policy. For scripts that include aggregators that do not reduce data sizes, DISH's worker-first aggregation optimization risks transferring more data. As DISH's evaluation confirms (§7), however, worker-first aggregation results in performance benefits for most scripts.

Delegated script concretization: DISH's scheduler sends workers dataflow subgraphs, encoded in DISH's intermediate representation, instead of concrete shell scripts ready for execution. Each dataflow subgraph contains holes that workers are expected to fill in, based on the specifics of their local environment. This choice simplifies DISH's distributed execution, as the scheduler does not need to have up-to-date information about several worker details such as the temporary directories they use. Additionally, this choice allows for better resource utilization in a heterogeneous environments with different worker capabilities: a worker can apply another optimization

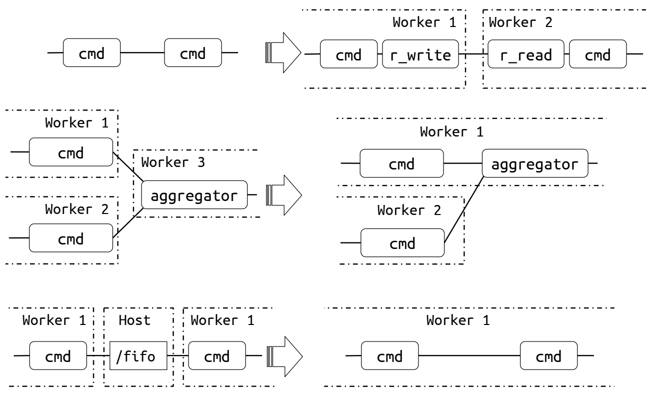

Fig. 6: (Top) Remote writes and reads added during distributed scheduling. **(Mid)** Worker-first aggregation. **(Bot)** Named FIFO teleportation.

pass to the dataflow subgraph it receives to better manage and utilize its resources.

Named FIFO teleportation: Scripts often use named FIFOs to share data between concurrently executing processes. Named FIFOs introduce a performance challenge, because they are local files that reside on the host machine where the script was executed. Therefore, by default, all data that would normally go through named FIFOs in the original execution would now have to go back and forth between workers and the machine for which the script was developed. DISH addresses this challenge by observing that named FIFOs are ephemeral, *i.e.*, they maintain no data after the execution of a dataflow region. Based on this observation, DISH migrates named FIFOs to workers closer to the data, eventually deleting the migrated versions after the dataflow region has finished executing (see Fig. 6, Bot). This transformation, termed FIFO teleportation, improves performance by avoiding unnecessary data movement in scripts that use FIFOs.

6 Runtime Support

DISH has to address several runtime challenges: communication among workers, identification of HDFS data block locations, and correctness in view of HDFS blocks split independently of newlines—a assumption necessary for several dataflow transformations. This section describes several components of DISH's runtime that address the above challenges.

Remote FIFO channel: As described earlier (§5), connections between dataflow nodes are instantiated using UNIX FIFOs in a single-machine setting. Unfortunately, FIFOs do not support networked operation and thus cannot cross worker boundaries. To address that, DISH introduces a remote FIFO primitive (RFIFO) implemented in Go and using socket-based communication. RFIFOs are intended to operate identically to FIFOs, *i.e.*, implement the semantics of dataflow graph edges, but can operate over the network. They have a unique identifier and two ends—a read end and a write end.

Since shell streams are lazy, *i.e.*, a producer blocks until its consumer requests input, the network link is often not fully utilized, lowering throughput, and risks introducing significant latency. To avoid these throughput and latency challenges, DISH adds two buffer nodes to the dataflow graph: one before the write end of the RFIFO, to allow uninterrupted access to data, and one after the read end of the RFIFO, to force the read to request data. This lazy-to-strict optimization maintains correctness and improves performance in most cases, but in rare cases may lead to unnecessary data transfer between nodes—*e.g.*, when there is a `head` command right after the read end of an RFIFO.

Dynamic discovery service: As transformations and optimizations are applied during the execution of a script—contrary to most other distributed environments—DISH's scheduler cannot statically predict which ports will be available at runtime for remote FIFOs at each worker: different scripts and script fragments running concurrently during a single execution may collide on port usage. To address that, DISH workers implement a discovery service that can be accessed by remote FIFOs to (1) advertise their port, and (2) discover the port that their other end uses. The discovery service, implementing in Go and using gRPC [55], is logically centralized and supports a few remote procedure calls (RPCs), central among which are a `put` call for advertisement and a `get` call for discovering a remote end. RFIFOs are extended with gRPC clients to access the dynamic discovery service and identify their other end. Since the ports are not fixed up front, DISH's dynamic discovery service facilitates loose subgraph coupling and simplifies remote subgraph execution on multiple workers.

HDFS data retrieval: During transformations, the DISH compiler (§4) needs to retrieve information about HDFS paths to expand them into block sequences. This expansion happens on a critical runtime path and thus needs to be efficient. DISH's first implementation invoked this expansion on every HDFS path using a shell command—by wrapping `fsck`, a command offered by HDFS API for querying the health of the disk in the cluster, returning information about a file and its partitioning into blocks. This implementation ended up incurring significant latency (> 1s), and thus upon further investigation DISH switched to the web API resulting to sub-10ms latencies.

Enforcing logical block boundaries: A key challenge when processing separate file blocks in HDFS is a mismatch in splitting assumptions: HDFS blocks might not be split on newline boundaries, but to guarantee correctness the DISH compiler (§4) assumes that all blocks are logically separated by newlines. If left unaddressed, this mismatch may and lead to Processing them as they are would invalidate the transformation correctness, leading to wrong results.

Tab. 2: Benchmark summary. Summary of all the benchmarks used to evaluate DiSh, and their characteristics.

Benchmark	Scripts	Pure HS	LOC	Input	Source
1 Classics	10	7/10	123	3G	[3, 4, 26, 33, 53]
2 Unix50	34	30/34	142	21G	[5, 29]
3 COVID-mts	4	4/4	79	3.4G	[56]
4 NLP	21	-	306	120 books	[7]
5 AvgTemp	1	1/1	31	3.6G	[62]
6 MediaConv	2	-	35	0.8 & 0.4G	[43, 50]
7 LogAnalysis	2	-	63	0.7 & 1.3G	[43, 50]
8 FileEnc	2	-	44	1.3G	[35]

DiSh addresses this mismatch by implementing a distributed file reader (DFR) service that runs on every worker. The DFR ensures that parallel dataflow nodes only process batches that are split in newline boundaries, independent of how the actual physical blocks are split—providing the illusion of a logical block that ends at a newline to its consumer. Given a distributed file path, the DFR reads the local file or block from the worker's disk going beyond the first newline character in its block (except for reader 0 that reads from the start). If the block is not terminated with a new line, then it communicates with the reader of the next block (and potentially any readers after that), returning a complete logical block to its consumer. When a compiled dataflow graph is translated back to a script, it prefixes file paths with a command invoking a DFR client that communicates with the relevant DFR service to retrieve the relevant logical block. Both service and client are implemented in Go, communicating using gRPC and protobufs [17].

7 Evaluation

We are interested in evaluating two aspects of DiSh: its performance and its compatibility with respect to Bash. To do so, we perform four experiments using several real-world shell scripts taken from a variety of sources (Tab. 2).

Experiments: The first two experiments focus on the performance gains (§7.1) achieved by DiSh's distribution on (1) a 4-node on-premise cluster, and (2) a 20-node cloud deployment—both evaluated using a variety of benchmarks and workloads. We compare DiSh's performance against (1) GNU Bash [44], the *de facto* sequential shell-script execution environment; (2) Apache Hadoop Streaming [21] (AHS), a production-grade distributed data-processing framework that supports language-agnostic executables. In the case of the 4-node, we additionally compare DiSh with PaSh [27, 58], a shell-script parallelization system from the Linux Foundation and the current state of the art in terms of automation, agnosticism, and correctness (Tab. 1); PaSh's parallelism benefits make it a likely alternative to smaller clusters, where DiSh's anticipated benefits of distribution might be smaller, but this likelihood diminishes as the size of the cluster grows.

The last two experiments evaluate DiSh's dynamic dependency untangling (§7.2) and DiSh's correctness (§7.3), *i.e.*, its compatibility with respect to Bash across all scripts and the POSIX shell test suite.

Benchmarks: We use 8 sets of real-world benchmarks, totaling 76 shell scripts and 823 LOC. Classics and Unix50 contain classic and recent (c. 2019) scripts making heavy use of Unix and Linux built-in commands. COVID-mts contains four scripts used to analyze real telemetry data from mass-transit schedules during a large metropolitan area's COVID-19 response. NLP contains several scripts from Unix-for-poets, a tutorial for developing programs for natural-language processing out of Unix and Linux utilities. AvgTemp contains a large script downloading and processing multi-year temperature data across the US. MediaConv contains two scripts that process, transform, and compress video and audio files. LogAnalysis contains two scripts that apply typical system-administration and network-traffic analyses over log files. Finally, FileEnc contains long aliases that encrypt and compress files.

Baselines and implementations: Bash, PaSh, and DiSh executed every shell script completely unmodified.

Apache Hadoop Streaming (AHS) posed significant expressiveness limitations. Only 42 scripts in Classics, Unix50, COVID-mts, and AvgTemp out of 76 scripts can be implemented natively (Tab. 2, col. Pure HS). Another 7 scripts required manual porting by splitting them into mappers, reducers, and additional components: These components were not available natively by AHS—for example, components for reading from two pipelines for *Diff* and for sorting after the reducer for *Bigrams*. During porting, we put significant care in avoiding limiting AHS's parallelism: we modified 3 AHS scripts in Classics to help HS introduce additional parallelism—for example, we manually expanded `tr -cs` into `tr -c | grep -v` (both stateless). None of the scripts in NLP, MediaConv, or LogAnalysis can be implemented in AHS, as they perform processing in loops and are thus not expressible in AHS. We attempted to replace the body of the loop with an AHS invocation but the startup overhead ended up dwarfing the execution time (on average $> 10\times$ more than the Bash execution time).

Hardware & software setup: The 4-node cluster consists of four 6-core Intel(R) Core(TM) i7-10710U CPU nodes each with 64GBs of RAM, located in the same room and connected with an average bandwidth of 90.8 Mbits/sec. The 20-node deployment consists of xl170 Cloudlab [12] nodes equipped with $10 \times$ Intel Core E5-2640 2.4 GHz CPUs and 8GB of memory. Single-machine shells (Bash & PaSh) were evaluated on a machine with $20 \times$ 2.80GHz Intel(R) Core(TM) i9-10900 CPUs and 32GB of memory.

For ease of deployment and reproducibility, we used Docker `swarm` to deploy (1) HDFS, and (2) the DiSh run-

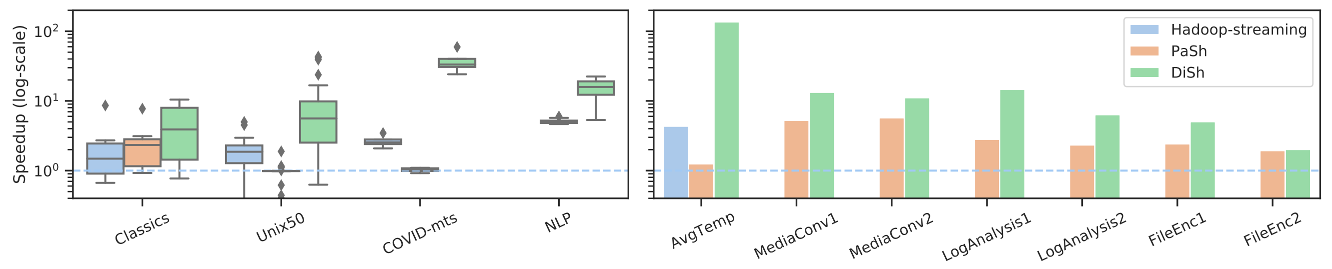

Fig. 7: DISH performance on a 4-node cluster. DISH speedup (*vs.* PASH and Hadoop Streaming whenever possible) over Bash for Tab. 2 rows 1–4 (left, box) and 5–8 (right, bar) (*Cf.*§7.1). (Log y-axis; higher is better.)

time. The containers were created using the standard Ubuntu 18.04 image. We use Bash v.5.0.3, PASH v.6e2ecba, and HDFS/Hadoop streaming version 3.2.2. We explicitly disabled checksum verification from HDFS in all configurations, scripts, and measurements. All scripts were executed completely unmodified, using environment variables, loops, and other shell constructs. To minimize statistical nondeterminism we repeated the experiments several times noticing imperceptible variance.

The DISH implementation comprises 6784 lines of Python (preprocessor, compilation server, expansion, compiler, and parser), 1011 lines of shell code (JIT engine and various utilities), and 1174 lines of C (commutativity primitives, and other runtime components). All counts include only semantically meaningful lines of code.

7.1 Performance

How does DISH's distributed perform on small on-premise clusters and multi-node cloud deployments, and how does it compare to state-of-the-art systems?

Results: Fig. 7 (N.b.: log y-axis) shows the performance of DISH, PASH, and AHS on a 4-node on-premise cluster across all benchmarks of Tab. 2. Box-plots (left) show result quartiles for multi-benchmark suites (Tab. 2, rows 1–4) and bars (right) show results for individual scripts (Tab. 2, rows 5–8). Across all benchmarks, DISH achieves an average speedup of 13.6× (*vs.* 2.55× for PASH and 2.1× for AHS) and a maximum speedup of 136.3× (*vs.* 7.8× for PASH and 8.6× for Hadoop Streaming). DISH is only slower than Bash (737s vs 568s) in the case of `diff.sh` from Classics, for which AHS is even slower (766s). DISH achieves a performance comparable to Bash (1-2s) in `4.sh` and `34.sh` from Unix50, because both perform a short-running `head`.

Fig. 3 shows the speedup of DISH *over AHS* on a 20-node Cloudlab deployment across all scripts implementable with AHS (Classics, Unix50, COVID-mts, AvgTemp). Across all benchmarks, DISH achieves an average speedup of 6.17× and a maximum speedup of 16.95× over AHS. DISH is slower than AHS only for three scripts: `nfa-regex.sh` from Classics (0.92×), `29.sh` and `30.sh` from Unix50 (0.91× and 0.94×).

Tab. 3: DISH performance in 20-node cloud deployment. DISH speedup over Hadoop Streaming for scripts that AHS supports.

DISH speedup over AHS						
Benchmark	Avg	Min	25th	50th	75th	Max
Classics	2.74×	0.92×	2.41×	2.60×	2.85×	6.55×
Unix50	6.64×	0.91×	2.85×	5.38×	10.4×	16.9×
COVID-mts	10.4×	6.64×	8.91×	9.27×	-	16.8×
AvgTemp	7.85×	-	-	-	-	-

Discussion: DISH is faster than Bash, PASH, and AHS across Tab. 2's suites (rows 1–4) with respect to average, and across all of Tab. 2 individual benchmarks (rows 5–8)—often by a significant margin (*e.g.*, 134× for `AvgTemp` against PASH). DISH's (and PASH's) speedup over Bash is due to parallelism. DISH's speedup over PASH is due to DISH' co-location of data and computation: PASH cannot offload computation and thus first gathers all data onto a single machine—a time-consuming stage—and then starts processing in parallel.

DISH's speedup over AHS is due to a few different reasons. One reason is the increased expressiveness of DISH's dataflow model: DISH accepts and parallelizes complete scripts, discovering more opportunities for parallelism. Many of the AHS scripts are broken into multiple map and reduce stages, often leaving pipeline parallelism and data parallelism unexploited. Another reason is DISH's dynamic independence discovery, which allows for additional parallelism and better utilization of resources—in ways that AHS does not support; we zoom into these benefits below (§7.2). In the Cloudlab deployment, DISH is (marginally) slower than AHS in only two cases: (1) a script that is embarrassingly parallel and thus implementable in AHS using only a single mapper (`nfa-regex`), and (2) two scripts in Unix50 that see slightly more benefits from our manual, hand-optimized AHS rewrite than they do from DISH.

We found porting scripts to AHS a serious challenge. Many scripts required manual effort, resulted in multiple error-and-fix cycles, and led to script size increases. Moreover, to over-

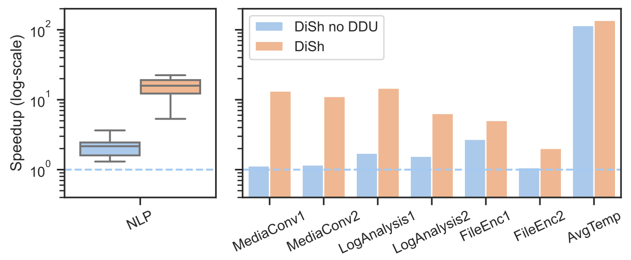

Fig. 8: Dynamic dependency untangling. DISH speedup over Bash when toggling DDU (higher is better).

come AHS's expressiveness limitations, we had to modify a few scripts in unintuitive ways—often combining plain Bash scripts with AHS mappers and reducers. These modifications made scripts significantly more complex and compounded the effort to test and maintain them. Instead, DISH distributed scripts successfully without any of these challenges.

7.2 Dynamic Dependency Untangling

What is the speedup due to dynamic dependency untangling?

Results: Figure 8 shows DISH's speedup over Bash with and without dynamic dependency untangling (DDU, § 4.3). It excludes scripts that contain a single dataflow, for which DDU is not applicable. DISH's average speedup over DISH-w/o-DDU is 6.9×, ranging between 1.2–13.9×.

Discussion: Enabling DDU improves performance significantly across all relevant scripts, by running independent dataflow regions in parallel. This allows DISH to expose parallelism not just within data pipelines but across them, improving utilization. DDU also improves the distributed execution of scripts that operate on many files, many or most of which are small enough to fit in a single HDFS block.

DDU is the main reason why DISH gets an edge over Bash on scripts that (1) have implicit independences that are not highly parallelizable, and (2) operate on small data that incur imperceptible data-movement costs. Examples of such scripts include MediaConv1 and FileEnc2.

7.3 Correctness

What is DISH's output compatibility with respect to Bash?

Results: To check the correctness of DISH across all benchmarks, we check that DISH's stdout and exit status are equivalent to the ones produced by Bash. The output is over 650 million lines (18GB), all of which return correct output and exit status when executed with DISH.

We additionally execute the complete POSIX shell-test suite to evaluate DISH's compatibility with Bash. Out of all relevant tests, DISH diverges from Bash in two cases and only with respect to the exit status it returns: both exit with an error, but Bash returns 1 whereas DISH returns 127. This divergence is due to DISH's invocation of Bash when executing optimized script fragments.

Discussion: The distributed execution of all the benchmarks in Tab. 2, all executed repeatedly in the past several weeks, offers ample opportunities for fragments to execute out of order. Comparing the output on every run provides significant confidence about the correctness of the resulting distributed execution. The POSIX test suite mostly evaluates the correctness dynamic orchestration (§3), as it does not feature many opportunities for parallelization and features zero opportunities for distribution.

8 Related Work

DISH is related to a large body of prior work in several domains.

Distributed data processing: Several environments assist in the development of distributed software systems: distributed computing frameworks [8, 38, 39, 51, 65] and domain-specific languages [1, 6, 10, 34, 36] simplify the development of distributed systems that fall under certain computational classes such as batch processing, stream processing, *etc.* These systems deal with many of the challenges of distribution, but require developers to (re)write their computations manually in models that differ significantly from UNIX shell programming.

Hadoop Streaming and Dryad Nebula are abstractions that allow using third-party language-agnostic components similar to the UNIX shell, atop cluster-computing engines (Hadoop and Dryad, respectively). Both require their users to understand and rewrite their shell scripts using the abstractions provided by each framework. DISH can operate on arbitrary shell scripts automatically, without requiring any manual effort from its users.

Distributed shells and tools: Several packages expose commands for specifying parallelism and distribution on modern UNIXes—*e.g.*, `qsub` [15], SLURM [63], calls to GNU `parallel` [52]. Different from DISH, their effectiveness is predicated upon explicit and careful invocation and is limited to embarrassingly parallel (and short) programs. Often, these commands provide options to support an array of special sub-cases—a stark contradiction to the celebrated UNIX philosophy. For example, `parallel` contains flags such as `--skip-first-line`, `-trim`, and `--xargs`, that a UNIX user can achieve using `head`, `sed`, and `xargs`; it also includes other programs with complex semantics, such as the ability transfer files between computers, separate text files, and parse CSV. DISH embraces the UNIX philosophy, attempting to rewrite shell programs to leverage distributed infrastructure.

Several shells [11, 32, 50] add primitives for non-linear pipe topologies—some of which target distribution. Here too, however, developers are expected to manually rewrite scripts to exploit these new primitives.

POSH [43] is a recent shell for scripts operating on NFS-stored data. It brings pipeline components closer to the data on which they operate, but operates only on shell pipelines that are fully expanded—*i.e.*, ones that do not use dynamic features. DiSh operates on shell scripts that use (1) any POSIX composition primitive, and (2) the full set of dynamic features present in the Unix shell.

Distributed operating systems: There is a long history of operating systems [2, 9, 37, 40, 42, 45–47, 61] for networked and distributed environments. These systems usually offer abstractions that (1) are similar, but not identical, to the ones offered by Unix, (2) operate at a lower level of abstraction (*e.g.*, that of system calls, rather than shell primitives), and (3) often aim at simply hiding the network rather than offering scalability benefits. Instead of implementing full-fledged distributed operating system, DiSh shows that a thin but sophisticated rewriting-based shim can operate on completely unmodified programs, avoid requiring any user input, and achieve significant speedups by executing fragments in parallel across nodes.

Annotation-based transformations: Recent systems [41, 59, 64] lower the developer effort of scaling out program components by performing program transformations based on user-provided annotations. These systems operate in single-language environments, offering declarative DSLs for tuning the semantics of the resulting distributed program. DiSh uses a similar approach, leveraging command annotations from prior projects [43, 58], but operates on-the-fly—within an environment that makes extensive use of dynamic features and that allows combining components from multiple languages.

PaSh-JIT [27] parallelizes scripts by dynamically interposing between a shell script and the underlying shell interpreter. This kind of interposition offers significant performance benefits without jeopardizing correctness, *i.e.*, maintains compatibility with the underlying shell interpreter. DiSh uses similar insights and interposition architecture, but operates on a distributed multi-node setting and addresses challenges that are specific to this setting—such as integration with a distributed file system and distributed environment passing.

Correct distribution of dataflow graphs: The DFG is a prevalent model in several areas of data processing, including batch- and stream-processing. Systems implementing DFGs often perform optimizations that are correct given subtle assumptions on the dataflow nodes that do not always hold, introducing erroneous behaviors. Recent work [23, 28, 31, 48] attempts to address this issue by performing optimizations only in cases where correctness is preserved, or by testing that applied optimizations preserve the original behavior. DiSh uses its dynamic orchestration to achieve compatibility with the underlying shell and then achieves correct distribution on a per-region level by building on prior work on provably correct transformations for order-aware dataflow graphs [22]. Similarly to other automated shell script transformation works [43, 58], DiSh's correctness is predicated upon the correctness of the annotations describing commands.

Resurgence of shell research: Recent shell research [19, 20, 27, 30, 35, 43, 49, 50, 58] highlights renewed interest in shell scripting both as a vehicle for impactful research and as a target worthy of scientific attention. We see DiSh as a natural continuation of the insights and research behind recent systems [20, 22, 27, 43, 58], allowing other researchers to leverage DiSh's POSIX-compliant high-performance dynamic distribution in their future work.

9 Conclusion

DiSh is the first system able to distribute unmodified shell scripts that use (1) any POSIX composition primitive, (2) the full set of dynamic features present in the Unix shell, and (3) distributed file systems such as HDFS. DiSh uses a dynamic orchestration approach that instruments a given script and dynamically distributes it at runtime to then execute it using the underlying shell interpreter. As a result, DiSh avoids modifications to shell scripts and maintains compatibility with existing shells and legacy functionality. Evaluated against several alternatives available to users today, DiSh offers significant speedups, requires no developer effort, and handles arbitrary dynamic behaviors pervasive in shell scripts.

Acknowledgments

We would like to thank Ayush Bhardwaj, Hannah Gross, Lily Tsai, Malte Schwarzkopf, Michael Greenberg, Neil Ramaswamy, the Brown Systems Group, and NSDI23 reviewers for discussions and feedback on the paper. This material is based upon work supported by DARPA contract no. HR00112020013 and no. HR001120C0191, and NSF awards CCF 1763514 and 2008096.

References

[1] Peter Alvaro, Neil Conway, Joseph M Hellerstein, and William R Marczak. Consistency analysis in bloom: a calm and collected approach. In *CIDR*, pages 249–260, 2011.

[2] Amnon Barak and Oren La'adan. The mosix multicomputer operating system for high performance cluster computing. *Future Generation Computer Systems*, 13(4):361–372, 1998.

[3] Jon Bentley. Programming pearls: A spelling checker. *Commun. ACM*, 28(5):456–462, May 1985.

[4] Jon Bentley, Don Knuth, and Doug McIlroy. Programming pearls: A literate program. *Commun. ACM*, 29(6):471–483, June 1986.

[5] Pawan Bhandari. Solutions to unixgame.io, 2020. Accessed: 2020-04-14.

[6] Martin Biely, Pamela Delgado, Zarko Milosevic, and Andre Schiper. Distal: A framework for implementing fault-tolerant distributed algorithms. In *Proceedings of the 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, DSN '13, pages 1–8, Washington, DC, USA, 2013. IEEE Computer Society.

[7] Kenneth Ward Church. Unix™for poets. *Notes of a course from the European Summer School on Language and Speech Communication, Corpus Based Methods*, 1994.

[8] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[9] Sean Dorward, Rob Pike, David L Presotto, Dennis Ritchie, Howard Trickey, and Phil Winterbottom. Inferno. In *Proceedings IEEE COMPCON 97. Digest of Papers*, pages 241–244. IEEE, 1997.

[10] Cezara Drăgoi, Thomas A Henzinger, and Damien Zufferey. Psync: a partially synchronous language for fault-tolerant distributed algorithms. *ACM SIGPLAN Notices*, 51(1):400–415, 2016.

[11] Tom Duff. Rc—a shell for plan 9 and unix systems. *AUUGN*, 12(1):75, 1990.

[12] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[13] Jeff Epstein, Andrew P. Black, and Simon Peyton-Jones. Towards haskell in the cloud. In *Proceedings of the 4th ACM Symposium on Haskell*, Haskell '11, pages 118–129, New York, NY, USA, 2011. ACM.

[14] Jim Garlick. pdsh. https://github.com/chaos/pdsh, 2022. [Online; accessed September 15, 2022].

[15] Wolfgang Gentzsch. Sun grid engine: Towards creating a compute power grid. In *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 35–36. IEEE, 2001.

[16] Inc. GitHub. The 2021 state of the octoverse: Top languages over the years. https://octoverse.github.com/#top-languages-over-the-years, 2021. [Online; accessed June 1, 2022].

[17] Google. Protocol Buffers, 2022. Accessed: 2022-06-01.

[18] Michael Greenberg. libdash. https://github.com/mgree/libdash, 2019. [Online; accessed December 6, 2021].

[19] Michael Greenberg and Austin J. Blatt. Executable formal semantics for the posix shell: Smoosh: the symbolic, mechanized, observable, operational shell. *Proc. ACM Program. Lang.*, 4(POPL):43:1–43:30, January 2020.

[20] Michael Greenberg, Konstantinos Kallas, and Nikos Vasilakis. Unix shell programming: The next 50 years. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 104–111, New York, NY, USA, 2021. Association for Computing Machinery.

[21] Hadoop. Hadoop streaming. https://hadoop.apache.org/docs/r1.2.1/streaming.html, 2022. [Online; accessed September 15, 2022].

[22] Shivam Handa, Konstantinos Kallas, Nikos Vasilakis, and Martin C. Rinard. An order-aware dataflow model for parallel unix pipelines. *Proc. ACM Program. Lang.*, 5(ICFP), aug 2021.

[23] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. A catalog of stream processing optimizations. *ACM Computing Surveys (CSUR)*, 46(4):46:1–46:34, March 2014.

[24] Lluis Batlle i Rossell. *tsp(1) Linux User's Manual*. https://viceverza.homeunix.net/ viric/soft/ts/, 2016.

[25] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72, 2007.

[26] Dan Jurafsky. Unix for poets, 2017.

[27] Konstantinos Kallas, Tammam Mustafa, Jan Bielak, Dimitris Karnikis, Thurston H.Y. Dang, Michael Greenberg, and Nikos Vasilakis. Practically correct, Just-in-Time shell script parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 769–785, Carlsbad, CA, July 2022. USENIX Association.

[28] Konstantinos Kallas, Filip Niksic, Caleb Stanford, and Rajeev Alur. Diffstream: Differential output testing for stream processing programs. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–29, 2020.

[29] Nokia Bell Labs. The unix game—solve puzzles using unix pipes, 2019. Accessed: 2020-03-05.

[30] Aurèle Mahéo, Pierre Sutra, and Tristan Tarrant. The serverless shell. In *Proceedings of the 22nd International Middleware Conference: Industrial Track*, pages 9–15, 2021.

[31] Konstantinos Mamouras, Caleb Stanford, Rajeev Alur, Zachary G. Ives, and Val Tannen. Data-trace types for distributed stream processing systems. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 670–685, New York, NY, USA, 2019. ACM.

[32] Chris McDonald and Trevor I Dix. Support for graphs of processes in a command interpreter. *Software: Practice and Experience*, 18(10):1011–1016, 1988.

[33] Malcolm D McIlroy, Elliot N Pinson, and Berkley A Tague. Unix time-sharing system: Foreword. *Bell System Technical Journal*, 57(6):1899–1904, 1978.

[34] Christopher Meiklejohn and Peter Van Roy. Lasp: a language for distributed, eventually consistent computations with crdts. In *Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data*, page 7. ACM, 2015.

[35] Jürgen Cito Michael Schröder. An empirical investigation of command-line customization. *arXiv preprint arXiv:2012.10206*, 2020.

[36] Adrian Mizzi, Joshua Ellul, and Gordon Pace. D'artagnan: An embedded dsl framework for distributed embedded systems. In *Proceedings of the Real World Domain Specific Languages Workshop 2018*, pages 1–9, 2018.

[37] Sape J Mullender, Guido Van Rossum, AS Tanenbaum, Robbert Van Renesse, and Hans Van Staveren. Amoeba: A distributed operating system for the 1990s. *Computer*, 23(5):44–53, 1990.

[38] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, New York, NY, USA, 2013. ACM.

[39] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. Ciel: A universal execution engine for distributed data-flow computing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 113–126, Berkeley, CA, USA, 2011. USENIX Association.

[40] John K Ousterhout, Andrew R. Cherenson, Fred Douglis, Michael N. Nelson, and Brent B. Welch. The sprite network operating system. *Computer*, 21(2):23–36, 1988.

[41] Shoumik Palkar and Matei Zaharia. Optimizing data-intensive computations in existing libraries with split annotations. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 291–305, New York, NY, USA, 2019. ACM.

[42] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, et al. Plan 9 from Bell Labs. In *Proceedings of the summer 1990 UKUUG Conference*, pages 1–9, 1990.

[43] Deepti Raghavan, Sadjad Fouladi, Philip Levis, and Matei Zaharia. POSH: A data-aware shell. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 617–631, 2020.

[44] Chet Ramey. Bash reference manual. *Network Theory Limited*, 15, 1998.

[45] Richard F Rashid and George G Robertson. *Accent: A communication oriented network operating system kernel*, volume 15. ACM, 1981.

[46] Marc Rozier, Vadim Abrossimov, François Armand, Ivan Boule, Michel Gien, Marc Guillemont, Frédéric Herrmann, Claude Kaiser, Sylvain Langlois, Pierre Léonard, et al. Overview of the chorus distributed operating system. In *Workshop on Micro-Kernels and Other Kernel Architectures*, pages 39–70. Seattle WA (USA), 1992.

[47] Jan Sacha, Jeff Napper, Sape Mullender, and Jim McKie. Osprey: Operating system for predictable clouds. In *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN 2012)*, pages 1–6. IEEE, 2012.

[48] Scott Schneider, Martin Hirzel, Buğra Gedik, and Kun-Lung Wu. Safe data parallelism for general streaming. *IEEE Transactions on Computers*, 64(2):504–517, Feb 2015.

[49] Jiasi Shen, Martin Rinard, and Nikos Vasilakis. Automatic synthesis of parallel unix commands and pipelines with kumquat. corr abs/2012.15443 (2021). *arXiv preprint arXiv:2012.15443*, 2021.

[50] Diomidis Spinellis and Marios Fragkoulis. Extending unix pipelines to dags. *IEEE Transactions on Computers*, 66(9):1547–1561, 2017.

[51] Craig A Stewart, Timothy M Cockerill, Ian Foster, David Hancock, Nirav Merchant, Edwin Skidmore, Daniel Stanzione, James Taylor, Steven Tuecke, George Turner,

et al. Jetstream: a self-provisioned, scalable science and engineering cloud environment. In *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*, pages 1–8, 2015.

[52] Ole Tange. Gnu parallel—the command-line power tool. *;login: The USENIX Magazine*, 36(1):42–47, Feb 2011.

[53] Dave Taylor. *Wicked Cool Shell Scripts: 101 Scripts for Linux, Mac OS X, and Unix Systems*. No Starch Press, 2004.

[54] Elixir Core Team. Elixir. https://elixir-lang.org/.

[55] The gRPC Authors. grpc, 2018. Accessed: 2019-04-16.

[56] Eleftheria Tsaliki and Diomidis Spinellis. The real statistics of buses in athens. https://bit.ly/3s112R5, 2021.

[57] Junichi Uekawa. dsh. https://www.netfort.gr.jp/~dancer/software/dsh.html.en, 2022. [Online; accessed September 15, 2022].

[58] Nikos Vasilakis, Konstantinos Kallas, Konstantinos Mamouras, Achilles Benetopoulos, and Lazar Cvetković. Pash: Light-touch data-parallel shell processing. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 49–66, New York, NY, USA, 2021. Association for Computing Machinery.

[59] Nikos Vasilakis, Ben Karel, Yash Palkhiwala, John Sonchack, André DeHon, and Jonathan M. Smith. Ignis: Scaling distribution-oblivious systems with light-touch distribution. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 1010–1026, New York, NY, USA, 2019. ACM.

[60] Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in ERLANG (2Nd Ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1996.

[61] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The locus distributed operating system. *ACM SIGOPS Operating Systems Review*, 17(5):49–70, 1983.

[62] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 4th edition, 2015.

[63] Andy B Yoo, Morris A Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer, 2003.

[64] Gina Yuan, Shoumik Palkar, Deepak Narayanan, and Matei Zaharia. Offload annotations: Bringing heterogeneous computing to existing libraries and workloads. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 293–306, 2020.

[65] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for inmemory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.