POSH: A Data-Aware Shell

Deepti Raghavan Sadjad Fouladi Philip Levis Matei Zaharia

Stanford University

Abstract

We present POSH, a framework that accelerates shell applications with I/O-heavy components, such as data analytics with command-line utilities. Remote storage such as networked filesystems can severely limit the performance of these applications: data makes a round trip over the network for relatively little computation at the client. Reducing the data movement by moving the code to the data can improve performance.

POSH automatically optimizes *unmodified* I/O-intensive shell applications running over remote storage by offloading the I/O-intensive portions to proxy servers closer to the data. A proxy can run directly on a storage server, or on a machine closer to the storage layer than the client. POSH intercepts shell pipelines and uses metadata called *annotations* to decide where to run each command within the pipeline. We address three principal challenges that arise: an annotation language that allows POSH to understand which files a command will access, a scheduling algorithm that places commands to minimize data movement, and a system runtime to execute a distributed schedule but retain local semantics.

We benchmark POSH on real shell pipelines such as image processing, network security analysis, log analysis, distributed system debugging, and `git`. We find that POSH provides speedups ranging from 1.6× to 15× compared to NFS, without requiring any modifications to the applications.

1 Introduction

The UNIX shell is a linchpin in computing systems and workflows. Developers use many tools at the command-line level for data processing [33], from core `bash` utilities, including `sort`, `head`, `cat` and `grep` to more complicated programs such as `git` [22], ImageMagick [30] and FFmpeg [4]. Network security engineers use shell pipelines to find potential patterns in gigabytes of logs. The shell's continued importance over many decades and generations of computing systems shows just how flexible and powerful a tool it is.

The UNIX shell, however, was designed in a time dominated by local and then LAN storage, when file access was limited by disk access times, such that the overhead of network storage was an acceptable trade-off. Today, however, solid-state disks have reduced access times by orders of magnitude. At the same time, networked attached storage, especially for the enterprise, remains extremely popular [9, 57, 60]. Mounting filesystems across the wide area could incur tens of milliseconds of latency. Furthermore, many applications use wide area storage systems, via cloud blob storage [25, 54] or cloud-backed filesystems [7, 48, 50, 51, 58].

Running I/O-intensive shell pipelines over the network requires transferring huge amounts of data for little computation. For example, consider generating a `tar` archive on NFS. The `tar` utility effectively copies the source files and adds a small amount of metadata: the server reads blocks and sends them over a network to a client, who shifts their offsets slightly and sends them back. NFS mitigates this problem by offering compound operations [29] and server-side support for primitive commands such as `cp` [41]. However, something as simple as `tar` requires large network transfers.

The result of these changing performance trends is that network transfer is an increasingly large overhead on shell scripts. For example, unzipping a dataset of size 0.5 GB with `tar -x` over NFS within a cloud datacenter takes 7× longer than on a local disk. While intra-datacenter networking is fast, it is not as fast as a local flash drive. Some workflows are so slow over the network that they are effectively unusable: running `git status` on the Chromium repository [24] takes 2 seconds locally, but if the repository is stored in a nearby datacenter, it takes over 20 minutes.

The underlying performance problem of using the shell with remote data is locality: because the shell executes locally, it must move large amounts of data to and from remote servers. Data movement is usually the most expensive (time and energy) part of a computation and shell workloads are no exception. Near-data processing [1, 5, 14, 47, 52, 59] can reduce data movement overheads. Data-parallel processing systems such as Spark [61], stored procedures in SQL databases [43, 44], and native data structures in key-value stores such as Redis [40] all bring computation closer to the data. However, many of these systems require applications to use their APIs: they can supplement, but not replace shell pipelines.

To address the shell performance problem of data locality, this paper proposes POSH, the "Process Offload Shell", a system that offloads portions of *unmodified* shell commands to *proxy servers* closer to the data. A proxy server can run on the actual remote fileserver storing the data, or on a different node that is much closer to the data (e.g., within the same datacenter) than the client. POSH improves shell-based I/O workloads through three techniques. First, it identifies parts of complex pipelines that can be safely offloaded to a proxy server. Second, it selects which candidates run on a proxy, in order to minimize network data movement. Finally, it executes the pipeline across an underlying runtime, stitching together the distributed computations while maintaining the exact output semantics expected by a local program. Correctly and efficiently implementing these three techniques has three principal challenges:

1. Correctly understanding the semantics of shell command-line invocations in order to deduce which files each command in the pipeline accesses and determine which

commands can be offloaded.
2. Distributing the entire pipeline across different machines to minimize overall data movement, based on the "closest" execution environment (client or proxy) for each command and its file dependencies.
3. Automatically parallelizing pipelines that access many files while ensuring the output maintains a sequential execution order.

In order to address the challenge of understanding the semantics of shell command-line invocations, POSH uses *annotations*. POSH's key insight is that many shell applications only read and write to files specified in their command-line invocation, so POSH can deduce which files a command accesses from a model of the application's argument structure. Annotations store a model of each command's (e.g., `cat`'s or `grep`'s) semantics and arguments, stored locally at the client's shell. These annotations, inspired by recent proposals to annotate library function calls for automatic pipelining and parallelization [46], assign *types* to the possible arguments of command-line applications. At runtime, POSH can parse which arguments are files and use the underlying storage configuration to determine where those files are located.

Next, POSH must schedule the entire pipeline across the execution engine in a way that reduces data movement as much as possible. However, POSH does not know explicitly how much data is transferred across each pipe in the entire pipeline. If the output of `grep` or `awk` is piped to another command, POSH cannot know how much data will travel over the pipe without running the command. POSH constructs a DAG representation of the entire command and associated metadata and applies a greedy algorithm, that estimates how much data travels across each pipe, to determine the best way to schedule the DAG.

Finally, POSH further improves performance when it knows it can split commands into multiple data-parallel processes. POSH ensures that each split parallel invocation retains the same argument structure as the original command and that the output is stitched together in the correct order. It does so by using information about each argument stored in the annotation to safely split the command. POSH's execution engine serializes output from any parallel processes in the correct order, before writing to the final destination (e.g., client `stdout`).

This paper makes the following contributions:

1. **An expressive annotation language for shell commands**: Annotations capture the dominant grammar of most shell commands and summarize which inputs to command invocations are files as well as semantics to safely split command invocations across inputs.

2. **A greedy scheduling algorithm that reduces data movement for unmodified shell pipelines**: POSH's scheduling mechanism decides which parts of pipelines can be offloaded to proxy servers closer to the data and which parts of pipelines can be parallelized, while preserving the correctness of the output.

We evaluate POSH on a variety of workloads, including an image processing pipeline that creates thumbnails, a `git` command workflow on the Chromium repository, and log processing pipelines from research project analysis. On one hand, for a more compute-heavy log analysis application that includes a data transfer of one large file, POSH provides a 1.6× speedup over running `bash` over NFS. In the best case, for a `git` command workflow, POSH provides a 10-15× speedup over running `bash` over NFS, even when the client and the server are in the same datacenter. Section 8 contains the set of full results.

2 Related Work

Near-data computing (NDP). POSH draws upon previous work that "ships computation closer to the data." Previous approaches to NDP, surveyed in [5], focus on two paradigms: PIM (processing in-memory), where compute is co-located with memory [28, 45, 47], or ISC (in-storage computing), where processors or accelerators are co-located with persistent memory or storage [1, 14, 52, 55, 59]. POSH follows the second approach and pushes portions of arbitrary shell applications to *proxy servers*, compute units running on general-purpose servers, either co-located with storage devices or closer to storage devices than the client.

Barbalace et al. [5] propose an entire operating system architecture for NDP with features such as locality-driven scheduling. Various systems focus on offloading database query computation into smart SSDs [14] or FPGA accelerators [32, 52, 59]. The Metal FS framework [52] allows users to run reusable compute kernels, that perform operations such as encryption or filtering, on FPGA accelerators near the storage. They can be programmed with standard shell syntax such as pipes to chain many near-data operations together. Seshadri et al. [55] propose an extensible programmable SSD interface where users can push specific pieces of functionality, e.g., to run filesystem appends, to the SSD. Finally, many databases allow users to write SQL queries to run as stored procedures [43, 44]. Similarly, many key-value stores, such as Redis [40], Splinter [39] and Comet [21] support extensibility with user-defined functions. POSH, in contrast, focuses just on a locality-aware shell, to enable the extensibility of remote filesystems. POSH pushes computation to proxy servers (that do not require custom hardware), without forcing the user to explicitly decide which operations should be offloaded.

NFS Optimizations and filesystems. NFS, starting in version 4.2 [29], offers support for some server-side operations, such as server-side copy [41]; however, NFS does not have support for offloading arbitrary programs. NFS allows batching operations via compound operations [29]; Juszczak [37] describes techniques to batch writes. vNFS [10] offers a new API for NFS that supports batching and vectorizing filesystem operations. This technique reduces the latency of running commands such as `tar` over NFS, but still requires moving the data across the network, instead of pushing the computation to the data. CA-NFS [6] attempts to improve application performance in a multi-tenant scenario by adaptively scheduling certain client operations to run asynchronously during periods of high system load. POSH currently does not handle multi-tenancy, but could use similar methods to factor system load to adapt scheduling decisions. BlueSky [58] proposes running proxy servers in the cloud that serve data from slower blob storage; these proxy servers can expose NFS access to the data. POSH, in contrast, uses proxy servers to push application code.

Distributed execution engines. Distributed cluster computation systems such as Spark [61], MapReduce [13], Dryad [31], Hadoop [19] also automatically parallelize computation on large datasets, but require that users follow a specific API. Systems such as gg [17] and UCop [15] take "everyday applications," such as software compilation, unit testing and video encoding, and automatically parallelize them in the cloud. These systems focus on compute-intensive workloads, not I/O-intensive, and do not necessarily make decisions about scheduling computation to preserve data locality, unlike POSH.

Code offloading and type systems. Many systems enable *code offloading*, to implement distributed applications on many mobile devices that can benefit from computation on nearby servers [8, 26, 35]. Some of these systems require specific programming language constructs to offload processes and partition programs [8, 27, 34–36]. Recently, Pyxis [11] optimizes database applications by *automatically* offloading procedures to run at the database server. It uses program analysis to determine what to offload, while POSH uses per-command annotations. Split Annotations [46] proposes using per-function annotations to determine how to split and pipeline function calls in data analytics workloads, to enable cross-function cache pipelining and parallelization. These annotations are fundamentally different from POSH's shell annotations: POSH's annotations attempt to understand the command-line semantics of shell commands, which tend to have a much more varied structure than function calls.

Command-line tools. Many command-line tools allow users to automatically parallelize and execute shell commands remotely. `rsh` [3] enables remote execution of shell commands. `pssh` [12] allows users to execute commands over SSH in parallel on other machines. GNU Parallel [56] splits input arguments and executes jobs over these inputs in parallel across one or more machines. POSH also parallelizes commands on remote machines, but *automatically* decides how to offload and schedule commands so users do not need to explicitly program when to offload code. POSH does not rely on SSH access to the storage servers and can be used on top of a service such as NFS, where remote shell access may be prohibited.

3 SYSTEM OVERVIEW

POSH consists of three main components: a shell annotation interface (§4), a parser and scheduler (§5), and an execution engine (§6). This section briefly describes each component and how they link together, pictured in Figure 1.

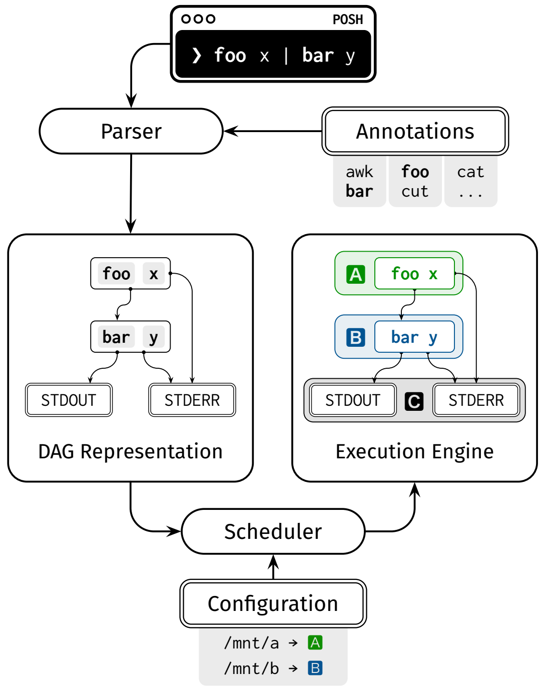


Figure 1: In POSH's main workflow, a shell command is passed to the parser, which uses the annotations to generate and schedule a DAG representation of the command. The DAG includes which machine to run each command on, A, B, or C (client) here. The execution engine finally runs the resulting DAG.

Annotation interface. POSH, on bootup, requires users to provide a file containing a list of annotations for any commands they want POSH to consider offloading. Annotations are written *once per command*, e.g., once for `grep` or once for `awk`, so POSH can then accelerate shell pipelines that combine these commands with standard constructs, such as anonymous pipes. We envision that developers can share annotations for popular programs, so users do not necessarily need to write their own annotations; crowdsourcing annotations has seen success with TypeScript [2, 42, 49].

Parser and scheduler. Given a shell program, the POSH parser turns each pipeline (each line of the program, potentially consisting of several commands combined by pipes and redirects) into a directed acyclic graph (DAG). This graph represents the input-output relationship between commands, the standard I/O streams (`stdin`, `stdout` and `stderr`) and redirection targets, as shown in Figure 1. POSH then parses each individual command and its arguments using the corresponding annotation and completes the DAG by including additional input and output dependencies of the pipeline.

The parser finally runs a scheduling algorithm on the DAG and assigns an execution location to each command in the pipeline. In order to do this, the parser requires extra *configuration information* that specifies a mapping between each mounted client directory and the address for a machine running a proxy server for the corresponding directory (if any).

Execution engine. After POSH has parsed and scheduled a shell pipeline, it executes the command across the underlying execution engine. The execution engine consists of one or more proxy servers, each associated with a specific remote client mount, either at the storage server, or in a nearby node with access to the same data. Additionally, one "proxy server" runs at the client to execute any local computation. POSH ensures that the entire command looks like it has been running locally, even if processes had been offloaded to proxy servers.

4 SHELL ANNOTATIONS

POSH must correctly understand the semantics of shell commands, which can be challenging because of the wide range of syntax allowed by command lines. In this section, we discuss the motivation and design of POSH's shell annotation layer.

4.1 Motivation for Shell Annotations

In order to schedule and execute shell pipelines in a way that minimizes data movement, POSH must understand the semantics of command-line pipelines. Concretely, annotations must reveal enough information that allows POSH to determine:

1. Which commands can be safely offloaded to proxy servers.
2. If any commands in the pipeline filter their input.
3. If any commands can be split in a data-parallel way into multiple processes.

Consider a simple pipeline: `cat A B C D | grep "foo" | tee local_file.txt`. POSH could try to offload any of the three commands: `cat`, `grep`, or `tee`. To determine which commands are safe to offload, POSH must understand which files (if any) `cat`, `grep` and `tee` access, and where these files live. Therefore, POSH must determine which arguments to the three commands represent file paths. However, outside of the program, all of these arguments are seen as generic strings. For example, consider the following four commands:

```
cat A B C D | grep "foo"
tar -cvf output.tar.gz input/
tar -xvf input.tar.gz
git status
```

The `cat` command takes in four input files, while the argument to `grep` is a string. The second command, `tar -cvf`, takes an output file argument followed by `-f`, followed by an input file argument (not preceded by a short option). The third command, also `tar`, takes an input file argument followed by `-f` and implicitly takes its output argument as the current directory. Finally, `git` also implicitly relies on the current directory as a dependency. Without a formal way to model the argument structure for each command, POSH could not determine the file dependencies for each command.

Secondly, in order to produce an execution schedule that reduces data movement, POSH needs to know the relationship between the inputs and outputs of a command. In the `cat | grep` example, if the file argument to `cat` is remote, to minimize data movement, POSH cannot *just* offload the `cat` command. Since `cat` usually produces the same amount of output as input, but `grep` usually filters its input, POSH must also offload `grep`.

Finally, suppose `cat` had multiple file arguments, but these files lived on different mounts (e.g., a pipeline that processes logs from different servers). POSH could not safely offload this command to a single proxy server, as the proxy server may not have access to all the mounts. However, many command-line programs perform map functions over each line in the file in sequence. For example, `cat` prints all lines to the output, and `grep` filters the input line by line. Therefore, these commands can be split into processes across their input files and then offloaded to different proxy servers. However, parallelization is not safe for all commands: `wc`, for example, "reduces" the input. Without a formal model for the command's semantics, POSH could not make optimal scheduling decisions.

4.2 Annotation Interface

Annotations need enough information to allow POSH to deconstruct, parse and schedule each command. Annotations contain two types of information: argument-specific and command-specific information. First, they contain a list of arguments along with a type assignment for each argument. Second, they contain information relevant to parsing the entire command line, either semantic information relevant to scheduling, or custom parsing options. The annotation interface is inspired by the POSIX conventions for command-line arguments and their GNU extensions [23], which are followed by a multitude of UNIX utilities. If a program does not follow these conventions, POSH may not be able to determine how to accelerate it. We describe both parts of the annotation in turn.

4.2.1 Argument-Specific Information

POSH supports the following classes of command-line arguments:

1. A single option with no arguments (e.g., `-d` or `--debug`).
2. An option, followed by one or more parameters (e.g. `-f <file>`).
3. A parameter without a preceding option (e.g. the arguments in `cat A B C D`).

The annotations must specify the following information for each argument that has associated parameters:

1. The **`short`** or **`long`** option name: This is only relevant for arguments preceded by options.
2. The **`type`**: `input_file`, `output_file`, or `string`.
3. The **`size`**: `1`, `specific_size(x)`, or `list` (for variable size).
4. If the argument is **`splittable`**: If an argument has multiple parameters, the **`splittable`** keyword specifies that the command can be split in a data-parallel way across this argument (this is only allowed for up to a single argument).

The above format applies to many popular UNIX commands and core utilities. For example, the annotation for `cat` may look like:

```
cat:
  - PARAMS:
    - type:input_file,splittable,size:list
```

This specifies that `cat` takes in one or more input files, and it can be split across its inputs.

4.2.2 Command-Specific Information

Annotations contain information relevant to the semantics of the entire command, specified by the following keywords:

- `needs_current_dir`: Whether the command implicitly relies on the current directory.
- `splittable_across_input`: Whether the command can be split *across its standard input*. In the example pipeline, if the `cat` is split into separate `cats`, POSH would also need to split the `grep` command into separate commands to truly take advantage of parallelism.
- `filters_input`: Whether the command is *likely* to have a smaller output than input.
- `long_arg_single_dash`: Most programs use double dashes before long arguments (e.g. `--debug`), but some programs require long arguments be preceded by a single dash. (e.g. `-debug`).

4.2.3 Annotation Conflicts

Some commands can be invoked with flags whose behavior changes depending on which other flags are present. For example, the annotation for a `tar` invocation used to *create* an archive could be:

```
tar: [filters_input]
  - FLAGS:
    - short:c
    - short:z
  - OPTPARAMS:
    - short:f,type:output_file,size:1
  - PARAMS:
    - type:input_file,size:list
```

However, developers commonly invoke both `tar -x` and `tar -c`, to extract and create a tarball, respectively. The assignment for the `-f` flag conflicts; it would be an `input_file` in the `-x` case, but an `output_file` in the `-c` case. POSH supports this by allowing multiple annotations per command. In particular, a client can include one annotation *per type of invocation* for every binary, and POSH will try all annotations until it finds one that fits the current invocation.

4.3 Correctness and Coverage

Potential mistakes. POSH depends on correct annotations for optimal execution: if the annotations are incorrect in some way, POSH does not make guarantees about the correctness or the performance of the resulting execution. Annotations with incorrect type semantics (assigning an argument with `str` instead of `input_file`) or parallelization semantics (specifying a command is `splittable` when the command needs all input files concurrently) could cause execution errors. POSH might schedule the command on a machine without access to a necessary file, or incorrectly try to split the command into parallel workers. Annotations might not include potential optimization information, by omitting that a command filters its input or it can be parallelized. In this case POSH might not make the optimal scheduling decision, but execution will still be correct.

Finally, a command, such as `awk`, could either filter or increase its input depending on its invocation. `awk` could include multiple annotations for each separate program string, which separately specify or omit the `filters_input` keyword.

Mitigations. In practice, we expect that a community of developers would maintain and crowdsource a set of "verified" annotations; other annotation-based systems such as Typescript [42] make this assumption. To prevent incorrect type semantics, POSH could use a sandbox on top of the filesystem interface that checks if the program only reads and writes to files specified by the annotation, and crash the execution if the program accesses a file outside of its dependency list [17, 20]. Finally, future work could explore profiling commands to automatically deduce whether command invocations produce less, more or the same amount of input data as output data.

Coverage. The POSH interface covers the command-line syntax for a wide range of command line programs, as specified by the GNU command-line standard syntax and extensions [23]. Along with the source code of POSH, we provide annotations for 21 different commands, which cover a wide range of unmodified shell applications used in our evaluation (§7).

However, POSH will not cover *all* parsing options for all programs. While POSH can interpret wildcards ("*") when listing file paths, it will not do any custom parsing to list paths. For example, `ffmpeg` allows users to provide input files (input frames) based on a pattern, such as "`%04d.jpg`", which corresponds to all files between `0000-9999.jpg`. POSH will not parse commands whose file dependencies are specified *dynamically*, via a pipe: e.g., "`find . -type -iname "*.jpg" | xargs -i mogrify -resize 100x100`" could be used to dynamically list all `jpg` files and resize them with `ImageMagick`. Since POSH does not know what the file inputs to the `mogrify` command are upfront, it cannot decide whether to offload the `mogrify` command or not.

5 POSH'S PARSER AND SCHEDULER

This section discusses how POSH solves two challenges in executing shell commands efficiently across a set of proxy servers: (1) scheduling pipelines to minimize data movement (§5.2) and (2) correctly parallelizing pipelines (§5.3). We begin by discussing how POSH constructs an intermediate program representation of the command that allows it to

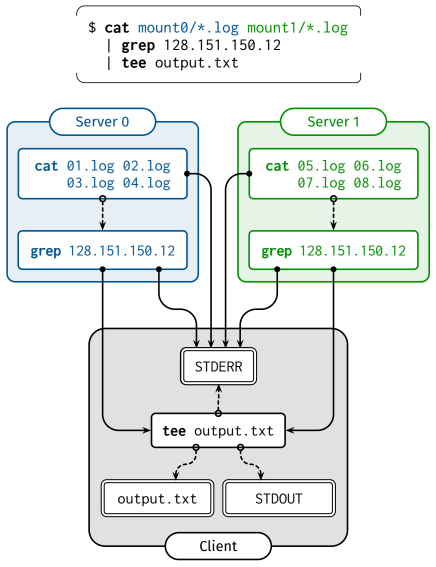


Figure 2: DAG representation of a simple shell program that uses `cat` and `grep` to analyze logs across different mounts. POSH uses its scheduling and parallelization mechanisms to offload the `cat` and `grep` to each server.

effectively solve these two problems.

5.1 POSH's Program Representation

POSH needs a program representation that allows the runtime to see every data source (file that is read), data sink (output file that is written to), and flow (pipe) that connects two commands within the pipeline. Understanding which files commands read and write allows POSH to determine the execution location closest to the data for each command. Understanding the connections in the pipeline allows POSH to see how data flows between commands and allows POSH to preserve dependencies when parallelizing nodes.

POSH represents parsed shell pipelines as directed acyclic graphs (DAGs), which contain *nodes* and *streams*. Nodes represent single commands (command nodes) along with each of their arguments, and parsed annotation metadata (argument types, if it is **`splittable`**, and if it filters its input). Nodes also represent sources and sinks in the entire pipeline: reading from `stdin` (read nodes) or writing to `stdout` or a file (write nodes). Streams represent the data flows in and out of these nodes.

Given a shell pipeline, POSH constructs a DAG that models the dependencies between commands (UNIX pipes) and standard I/O streams (`stdin`, `stdout` and `stderr`). POSH then builds a custom argument parser for each command from the annotations, to determine which arguments actually appear in each command's invocation and what their corresponding types are. For example, if a `tar -c` invocation contained the `-f` argument, POSH knows the string following `-f` is an `output_file`.

Figure 2 shows an example DAG generated for the simple program discussed in §4.1, that runs `cat` and `grep` on files stored in different mounts and pipes the final output to `tee`.

Algorithm 1 POSH Scheduling Algorithm

```
1: function SCHEDULE(dag)
2:   for node ∈ dag.GETNODES do
3:     if CONSTRAINT(node) != NULL then
4:       node.location ← CONSTRAINT(node)
5:   for source_node ∈ dag.GETSOURCES do
6:     path ← GETPATHTOSINK(dag,source_node)
7:     sink_node ← path[path.length() - 1]
8:     if source_node.location == sink_node.location then
9:       for node ∈ path do
10:        if node.location == NULL then
11:          node.location ← source_node.location
12:    else
13:      $min\_weight \leftarrow 1$
14:      edges ← { }
15:      for (node,next) ∈ path do
16:        if ISFILTERNODE(node) then
17:          $min\_weight \leftarrow min\_weight/2$
18:        edges[(node.id,next.id)] = $min\_weight$
19:      for (node,next) ∈ path do
20:        if node.location == NULL then
21:          if edges[(node.id,next.id)] ≥ $min\_weight$ then
22:            node.location ← source_node.location
23:          else
24:            node.location ← sink_node.location
25:        else if node.location != CONSTRAINT(node) then
26:          node.location ← CLIENT
```

POSH schedules and parallelizes the workload by first splitting the command into separate `cat` and `grep` commands that run at each proxy server and then merging the outputs at the client. Sections 5.2 and 5.3 discuss these steps in turn.

5.2 Scheduling

POSH's scheduling algorithm seeks to minimize data movement, as it assumes that in scenarios where the computation is I/O bound rather than CPU bound, minimizing data movement will reduce end-to-end latency. However, POSH does not know how much data a command will produce prior to execution. We describe the exact setup of the problem and POSH's greedy algorithm that uses information from POSH's annotations to estimate how to minimize data movement.

Problem setup. The POSH scheduling algorithm, summarized in Algorithm 1, takes the DAG representation of the command discussed in §5.1 and assigns an execution location to each node. POSH must pay attention to two concerns: *constraints* on where certain nodes can execute and the *number of bytes transferred across edges* in the DAG. The first concern arises because certain proxy servers might not be able to execute certain nodes as they do not have access to all the necessary files. The second concern arises because POSH seeks to minimize the number of bytes transferred between two locations across the network. Concretely, consider a `cat` node (operating on a remote file) that pipes its output to a `grep` node, that filters

Algorithm 2 POSH Scheduling Algorithm Helper Functions

```
 1: /* Returns constrained location assignment for node, if any. */
 2: function CONSTRAINT(node)
 3:   loc ← NULL
 4:   if ISREADNODE(node) then
 5:     loc ← GETREADLOC(node)
 6:   else if ISWRITENODE(node) then
 7:     loc ← GETWRITELOC(node)
 8:   else if ISCMDNODE(node) then
 9:     deps ← Set()
10:     for file ∈ GETFILEDEPENDENCIES(node) do
11:       deps.append(GETLOCATION(file))
12:     if deps.length() > 1 then
13:       loc ← CLIENT
14:     else if deps.length() == 1 then
15:       loc ← deps[0]
16:   return cost
17: /* Given a source node, trace a path to the sink via stdout paths. */
18: function GETPATHTOSINK(dag,node)
19:   path ← [node]
20:   while node.children.length() > 0 do
21:     path.append(GETSTDOUT(node))
22:   return path
```

its input, which in turn writes its output to `stdout` on the client. To schedule the least data movement across the network, POSH should offload both the `cat` and `grep` commands: in the path through the `cat`, `grep` and the `stdout` nodes, the minimum cut (edge with least data transferred) occurs after `grep`.

Step 1: Resolving constraints on each node. Algorithm 2 summarizes the helper functions for the scheduling algorithm, including a function, `Constraint`, that determines whether a node in the DAG has any constraints on execution location (lines 1-16). Read and write nodes are assigned to the location of their input or output data streams (lines 4-7). Command nodes that access files on a single mount are greedily assigned to execute on the proxy server corresponding to that mount (lines 14-15). Finally, command nodes that access files from multiple different mounts are always assigned to execute on the client (lines 12-13). Only the client has access to all mounts; by default, proxy servers only serve requests for a single mount. We discuss an optimization to this decision in §5.3, for special cases where the command can be parallelized.

Step 2: Minimizing data transfer. In Algorithm 1, after assigning locations to nodes with constraints (lines 1-4), POSH assigns locations for the remaining command nodes. POSH first iterates through all the source nodes of the graph, traces the path from each source to a sink node with the `GetPathToSink` helper function defined in lines 18-19 of Algorithm 2, and considers each path individually. Each path is guaranteed to be linear because each node in the graph effectively has one child. POSH assumes most data transfer occurs along `stdout` streams: even though command nodes have two children (one edge to `stdout` and one to `stderr`), POSH only pays attention to `stdout` connections. Read nodes can have one child by definition, and write nodes are sinks, so have no children.

Within a path, when the source and sink nodes have the same location, scheduling is simple: all nodes along the path from the source to the sink are assigned that location (lines 8-11). However, when the source and sink have different locations, the scheduler must find the edge along which cross-location data transfer should occur: to minimize data transfer, this should be the edge where the *least* data flows. POSH first iterates along each edge in a path and assigns relative weights according to heuristics (lines 13-18). POSH assumes nodes produce the same amount of output as input, or filters input by half (lines 16-18). The helper function `IsFilterNode` called on line 16 returns true for commands whose annotations indicate that they filter their input from the **`filters_input`** keyword. This heuristic, that filter commands halve their input, obviously does not fit all cases. Some filters, such as `wc`, usually produce much less than half the input. To find the minimum cut, only the relative ordering of edges matters, rather than the absolute weight values.

POSH then iterates along each path and schedules each unassigned node's location to be either the source location or the sink location, depending on if the node is before or after the minimum cut edge. For example, if a path contains `cat`, `grep`, and `cut`, and writes to `stdout` on the client, POSH determines the minimum weight edge is in between `cut` and the `stdout` write node. When there are conflicting assignments from nodes appearing in two different paths, POSH schedules the node on the client (lines 25-26).

5.3 Parallelization

The second challenge POSH resolves is determining when commands are safe to parallelize and then guaranteeing correct execution while parallelizing nodes. After determining a command is safe to split, correctly executing the command in a data-parallel way requires: (1) splitting the command only across the argument that can be split while preserving the other arguments and (2) stitching the outputs of the command back together in the correct order.

Determining which nodes to parallelize. POSH can automatically parallelize nodes that are safe to parallelize across the files they access, or across their input edges. In the first case, to parallelize the command `cat -n A.txt B.txt C.txt`, POSH needs to know that the file arguments are `A.txt, B.txt` and `C.txt`, and that the "`-n`" flag must be preserved. POSH uses the per-argument **`splittable`** keyword in an annotation, which indicates that the command can be split across a particular argument (here, the file argument). In the above example, POSH would replace the single node for `cat` with three nodes, each with the `-n` flag and one of `A.txt, B.txt` or `C.txt`.

In the second case, POSH allows nodes to also be parallelized *across their input streams*. Concretely, if the `cat` above piped its output to `grep`, there would not be much performance benefit to parallelizing `cat`, unless POSH also split `grep` across the three input `cat` nodes. POSH determines which commands this parallelization is safe for via the `splittable_across_input` keyword. The annotation for `grep` would include this keyword, so POSH would replace the `grep` node with three `grep` nodes for each `cat` node. This is not safe for commands that reduce or merge the input such as word count (`wc`): since the annotation for `wc` would not include this keyword, POSH would ensure that the output of the prior commands are merged before executing `wc`.

Splitting across mounts. When POSH splits nodes that can be parallelized, it will inherently split commands that read and write to *different mounts*. This allows the scheduler to bypass the restriction in Algorithm 2, lines 12-13. If a single command accesses files in different mounts, but the command can be split, instead of assigning this node to run at the client, POSH will split it into multiple workers that run in parallel on different proxies. POSH by default parallelizes commands across machines, but within a single machine, the *maximum splitting factor* parameter determines the degree to which to split further. The default value is 1, but increasing the splitting factor to a value $s > 1$ causes POSH to split a command into s commands that each operate on $\frac{n}{s}$ sized chunks of the input files, where n is the number of files that the node accesses on a single machine.

Correctly preserving output order. In order to ensure that the output of the entire pipeline is correct, when POSH splits a command in parallel, it must ensure that any node that reads the output of this node now reads the output of the replacement nodes *in sequence*. When nodes execute, they process their inputs in sequential order, guaranteeing correct output order.

6 POSH CONFIGURATION AND EXECUTION

In this section, we discuss how to configure POSH and how POSH executes programs.

6.1 POSH Configuration

To understand how to setup and configure POSH, consider a client that has access to folders on two NFS servers, within a nearby datacenter. Each NFS administrator has agreed to allow a separate proxy server, that resides within the same datacenter, to access the same mounts on behalf of the client (via NFS).[1] To use POSH, each of the two proxy servers must run the POSH *server* program, which is configured with a list of client credentials mapped to the folder paths each client has access to (which the proxy servers themselves access via NFS). The client must run the POSH *client* program, which takes in a file containing the shell annotations, and a configuration file that specifies a mapping from locally mounted folders, to the addresses for proxy servers for those mounts. In the previous example, the client's configuration file would map each of the two mounted folders to the corresponding proxy server. Together, the client and the two proxy servers make up POSH*'s execution engine*, which can be used to execute any schedules POSH's parser creates.

6.2 Execution Engine

After the client schedules a shell pipeline, the execution engine can execute the DAG.

Setup Phase. First, the client divides the DAG into subgraphs that need to execute on different machines (including a subgraph that will execute on the client itself). The client handles setting up persistent connections with proxy servers for any pipes between DAG nodes assigned to two different machines. Finally, when all pipes are setup, the client sends a request to each proxy server to start executing their portion of the DAG.

Execution Phase. Once each proxy server receives a request to execute a subprogram DAG, it will first spawn all the nodes in the DAG corresponding to processes that need to be executed (e.g., `cat` or `grep`). To redirect I/O between processes, POSH spawns a thread for each redirection that needs to occur and copies the output from each node to the correct pipe, TCP stream or file. For nodes that have multiple inputs, nodes process these inputs sequentially. Nodes that send output to nodes with multiple inputs buffer the content until the receiver starts processing that node's output.

6.3 Implementation

POSH is implemented in about 12,400 lines of Rust code. POSH uses Rust's CLAP library [38] to build custom parsers for each command, based on the command's annotation, to find out which arguments are actually present in an invocation.[2]

7 METHODOLOGY

We evaluate POSH by measuring its performance impact over five unmodified I/O-intensive shell applications. This section describes our evaluation methodology: the applications, why they cover a broad range of I/O-intensive shell applications, and our experimental setups.

7.1 Applications

For each application, in our evaluation setting, we assume that all input data files and intermediate files live on a remote NFS mount, so POSH accelerates these applications by preventing unnecessary data movement. Some applications require writing the final output to `stdout` or a file on the client; we specify this on a per-application basis.

Ray-tracing log analysis. The first application represents a best-case workload for POSH: it is computationally light, can be parallelized, and its output is a tiny fraction of the data it reads. The application analyzes logs of a massively distributed research ray-tracing (computer graphics) system [18], to track a

[1]The remote fileserver itself can also run the POSH server program and act as a proxy server.

[2]The implementation of POSH is available at https://github.com/deeptir18/posh.

task (a simulated ray of light) through the path of workers it traversed. The analysis first cleans and aggregates each worker's log into one file with `cat`, `grep`, `head` and `cut`. It then runs `sed` to search for the path of a single ray (e.g., a straggler) across all the workers and stores the final output on a file at the client.

Thumbnail generation. The next application is CPU-intensive, but still produces output that is a tiny fraction of its input, and is highly parallelizable. The application uses ImageMagick [30] to generate thumbnails of size 10 KB for each image in a folder of about 1090 images, each about 4 MB large; the output thumbnails are also written to the remote mount.

Port scan analysis. The third application is computationally heavy, but not parallelizable, and involves data transfers of large files. ZMap [16] is a network scanning tool that performs Internet-wide scans of the public IPv4 address space. Network security researchers run the following shell application to analyze 40 GB subset of a full Internet scan of port `80`; the final output file is stored in the remote mount.

1. Clean the raw data with a program called `zannotate`.
2. Use a JSON processing tool, `jq`, to isolate the *IP* and *AS ID#* (Autonomous System ID) columns.
3. Use `pr` to merge the columns together.
4. Use `awk` to count the number of IPs per AS.

Distributed log analysis. The next application is a synthetic benchmark that models system administrators running analysis on logs across *different storage servers*, to search for an IP address within the access logs stored across different machines. It is computationally light, highly parallelizable, and the output is a tiny fraction of the input. This workload runs `cat` over all of the files and then filters for a particular IP with `grep` and writes the results back to the file stored locally at the client. Each of five storage servers contains approximately 15 GB of logs from the SEC's EDGAR Log File Data Set [53].

Git workflow. The final application, a `git` pipeline on the Chromium [24] repository, attempts to imitate a developer's `git` workflow and is extremely *metadata-heavy*. After rolling back the repository by 20 commits and saving each commit's patch, the workload successively applies each patch and runs three `git` commands: `git status`, `git add .`, and `git commit -m`.

7.2 Setup and Baselines

Baselines. For all workloads, we compare POSH to two NFS configurations, one with with synchronous operations (`rw,sync,no_subtree_check`) and the other with asynchronous operations (`rw,async,no_subtree_check`).

POSH configuration. For all experiments, unless otherwise specified, the POSH proxy runs on the same machine as the NFS server. The POSH client also mounts the NFS folder locally in case some computations must run on the client. Section 8.2.1 evaluates the impact of placing the proxy directly at the storage server versus on another machine.

Network settings. We focus on two network scenarios:

1. Client and server in the same Google Compute Platform (GCP) region (`us-west2`). The RTT and throughput between the two machines are 0.5 ms and 5-10 Gbps, as measured by `ping` and `iperf`.
2. Client in a university network (at Stanford) and server in a nearby GCP region (`us-west2`). The RTT and throughput between the client and server are approximately 20 ms and 600 Mbps.

Setup. GCP client, proxy, and storage machines are configured with 4 vCPUs, 15 GB of memory, and 10 Gbps egress network bandwidth (`n1-standard-4`); they run Ubuntu 19.04. The client at Stanford runs Ubuntu 18.04. All storage servers store data on regional persistent SSDs.

8 EVALUATION

Our evaluation seeks to answer several questions: *(1)* Can POSH accelerate end-to-end pipelines that use many different command-line tools to access remote data over NFS (§8.1)? *(2)* What is the best way to configure POSH (§8.2)? *(3)* Where do performance benefits come from (§8.3)?

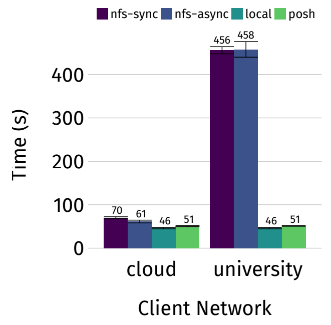
8.1 End-to-End Application Performance

For each workload described in §7.1, Figures 3 and 4 show the performance of POSH compared to `bash` over NFS for two network settings: one where the client is in the same GCP region as the storage server ("cloud") and one where the client is in a university network outside the datacenter ("university"). For `git`, Figure 4 only shows results for a client in the same datacenter.

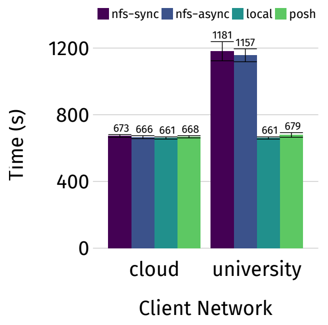
Summary of all results. On the university-to-cloud network, POSH performs 8× better than `bash` over NFS on the ray-tracing workload, 1-2× better on the thumbnail generation and port scan analysis workloads, and 12.7× better on the distributed log analysis workload. On the cloud-to-cloud network, POSH outperforms `bash` over NFS for the `git` workload and distributed log analysis workload; however, POSH does not outperform `bash` over NFS on the other three applications, partially because these applications are more bandwidth than latency sensitive. We discuss each set of applications below.

Ray-tracing log analysis. This workload sees an 8× improvement on the university-to-cloud network and no improvement on the cloud-to-cloud network. The workload reads 6 GB of input from about 2000 files over NFS, and aggregates them into one 4 GB file, which is written back to NFS. The final output of `sed` on the aggregated file is much smaller (20 lines). POSH prevents 10 GB of data from being copied across the network unnecessarily. On the cloud-to-cloud network, both the overheads of separate filesystem requests (to open and read 2000 files) and the overheads of transferring data to the client are not as large.

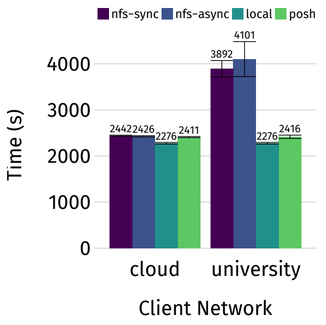
Thumbnail generation. This workload sees a 1.7× improvement in the university-to-cloud setting and no improve-



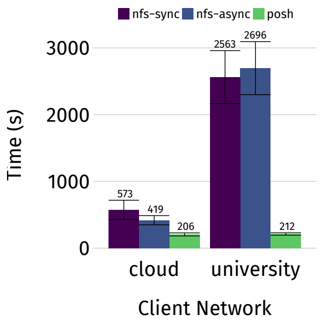
(a) Ray-Tracing Log Analysis



(b) Thumbnail Generation



(c) Port Scan Analysis



(d) Distributed Log Analysis

Figure 3: End to end latency of POSH on four applications, compared to NFS sync, NFS async and local execution time for two networks: one where the client is in a university network and one where the client is in the same GCP region as the storage server. The POSH proxy runs directly on the NFS server. POSH provides between 1.6-12.7× speedups in the university-to-cloud network compared to NFS. Using POSH from a client outside the datacenter results in about same latency as a client inside the datacenter using NFS, with barely any overhead over local execution.

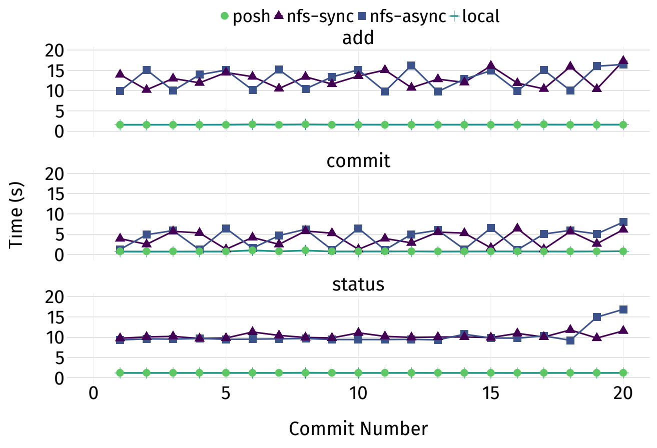


Figure 4: Average latency of 20 `git status`, `git add`, and `git commit` commands run on `Chromium` repo, of POSH compared to NFS and local execution, for a client in the same cloud datacenter as the storage server. POSH provides up to 10-15× speedups by preventing round trips for filesystem metadata calls.

ment in the cloud-to-cloud setting. Generating thumbnails with ImageMagick is computationally heavy: it takes 12 minutes to finish when running locally. While generating thumbnails produces a smaller input (12 MB of thumbnails vs. 15 GB of input images), in the cloud network, transferring 15 GB of data to the client will take about 12 seconds on a 10 Gbps connection. However, in the university-to-cloud setting, POSH attenuates the added delay from transferring data over a slower network.

Port scan analysis. The scanning analysis workload sees a 1.6× benefit with POSH in the university-to-cloud setting, but no benefit in the cloud-to-cloud setting. The scanning workload starts by processing one large 40 GB file with `zannotate` and writing the result back to NFS. This is more bandwidth than latency sensitive, as the application makes fewer filesystem requests across the network than the ray-tracing or thumbnail generation workloads. In addition, `zannotate` is CPU-intensive as it must parse each line of JSON in the input file.

Distributed log analysis. This workload sees a 12.7× improvement in the university-to-cloud setting, because POSH is able to *parallelize* the computation across the five different mounts and only aggregate the result locally. Both offloading the computation in order to prevent data movement as well as running the work on each machine in parallel, instead of sequentially, reduces latency. Even in the cloud-to-cloud setting, this results in a 2× speedup.

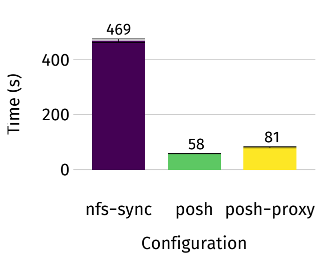
Git workflow. POSH sees the greatest performance benefit, a 10-15× latency improvement, when running `git` commands over NFS. Figure 4 shows the time for each `git status`, `git add`, and `git commit` commands for 20 commits, in the cloud-to-cloud network. We did not perform this full analysis for the university-to-cloud network, because the time to run a single `add` was up to two hours. `git` repositories typically contain many small files; commands like `status` and `add` check the status of every file in the folders to see if it has been modified. This results in NFS making filesystem requests such as `stat` for every file. As a comparison, the ray-tracing log analysis workload makes around 2,500 `open()` calls and 2,500 `stat()` class. For a `git add` in this workload, these numbers are 34,000 and 340,000, respectively, measured by `strace`. By offloading these commands to the server, POSH avoids many unnecessary roundtrips.

8.2 POSH Configuration

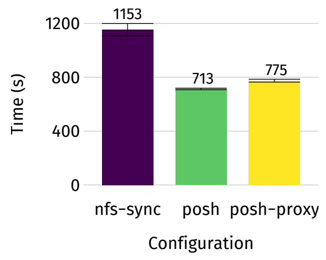
We evaluate two aspects of POSH's configuration: placement of proxy servers and maximum parallelization factor within a single machine.

8.2.1 Proxy Placement

Figures 5 and 6 shows the cost of placing the proxy server on a different machine from the storage server, within the same datacenter, for the client at Stanford, for three applications: ray-tracing, thumbnail generation, and `git`. The proxy server is closer to the data than the client; it has both a higher bandwidth



(a) Ray-Tracing Log Analysis



(b) Thumbnail Generation

Figure 5: Cost of running the POSH proxy on a separate server from the storage server, for a client outside the datacenter ("POSH-proxy", versus POSH where the proxy is at the storage server ("POSH"), and the baseline NFS sync execution time. POSH-proxy has a low overhead over POSH because there is not much overhead to running NFS between two machines in the datacenter.

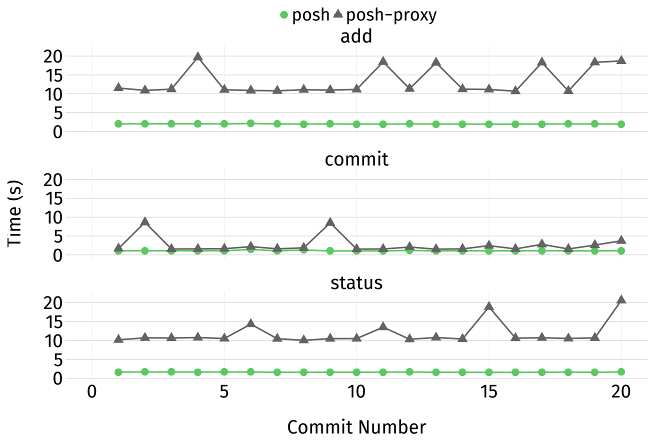
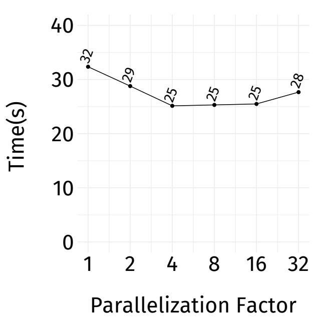


Figure 6: Cost of running the POSH proxy on a separate server from the storage server for a client outside the datacenter for the `git` workload. There is a 10-15x slowdown over POSH running directly at the storage server because running bash over NFS for this workload results in a 10-15x slowdown, due to filesystem metadata calls.

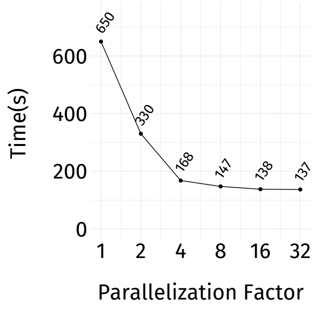
(10 Gbps vs. 600 Mbps) as well as lower latency (0.5 ms vs. 20 ms) connection to the storage server. However, in this setting, the proxy server will have at least the same latency as `bash` over NFS for the cloud-to-cloud setting. The proxy still must perform remote filesystem requests and transfer all the necessary data within the datacenter, because it mounts the data over NFS as well. `bash` over NFS does not have much overhead for the ray-tracing and thumbnail generation workloads, but has a 10-15× overhead for the `git` workload, due to many filesystem metadata calls. However, running `git` at the client outside the datacenter over NFS would have taken on the order of *hours*; POSH merely takes seconds.

8.2.2 Parallelization on a Single Machine

For a 16-core machine, Figure 7 shows the effects of varying the maximum splitting factor for the ray-tracing and thumbnail generation workload. The latency of the ray-tracing workload decreases until $s = 4$ processes and the latency of the thumbnail generation workload decreases until $s = 16$ processes. The limit at 16 is expected for a machine that only has 16 cores. In addition, with higher splitting factors, there is a higher overhead to spawning more threads and context switching. The ray-tracing workload does not benefit from parallelism past 4 threads because it already consists of multiple processes running in parallel, as the workload has many commands (`cat`, `grep`, `cut`, `head`): splitting this further does not provide further benefit.



(a) Ray-Tracing Parallelization



(b) Thumbnail Generation Parallelization

Figure 7: Latency improvements from using POSH to parallelize pipelines within a single machine, for the ray tracing and thumbnail generation workloads.

Application	NFS	POSH
Ray-Tracing Log Analysis	10 GB	3 KB
Thumbnail Generation	15 GB	0
Port Scan Analysis	175 GB	0
Distributed Log Analysis	80 GB	76.3 KB

Table 1: Bytes transferred over the network. Data movements are significantly reduced by POSH.

Application	Setup Time
Ray-Tracing Log Analysis	50 ms
Thumbnail Generation	30 ms
Port Scan Analysis	10 ms
Distributed Log Analysis	10 ms
`git status`	0 ms

Table 2: Median setup time.

8.3 Performance Improvements Analysis

Data movement reductions. For each of the log analysis applications and the thumbnail generation application, Table 1 reports the number of bytes of data transferred over the network that this application would generate, as well as the number transferred with POSH and its scheduling mechanism.

POSH overheads. Table 2 reports the latency of POSH's parsing and scheduling steps before execution for a single pipeline (single line of the script) from each application. The overheads are on the order of 10s of milliseconds and barely

Scenario	Latency	Data Movement
NFS Sync	225.8s ± 36.1s	6.38 GB
POSH	221.6s ± 22.1s	6.38 GB
POSH-OPT	33.27s	3.11 GB

Table 3: Expected latency of running POSH with a scheduler that can handle commands that need files from different mounts, for the command `comm A B`, for the university to cloud network.

affect total end-to-end latency. This includes time to parse the configuration file, parse all annotations (which is done once on shell startup), and parse and schedule each command in the pipeline. The ray-tracing and thumbnail generation workloads require resolving more filepaths, causing a larger overhead than the other applications.

9 LIMITATIONS AND FUTURE WORK

Algorithm limitations. POSH's scheduling algorithm handles pipelines that access data on different mounts in a map-reduce style pattern [13], but cannot handle commands which access data on different mounts that cannot be split. Consider a command such as `comm`, that finds the common lines between two files on different mounts: POSH would schedule this command to run at the client, causing no performance benefits over `bash` over NFS. However, if POSH scheduled this command on one of the proxy servers, rather than the client, and transferred the necessary files beforehand, POSH could provide a benefit over NFS. To produce such a schedule, POSH could consider the input files for each command, the data transfer speeds between the proxy machines and the dependencies of the DAG to construct an optimization problem and use standard graph partitioning techniques to solve for the optimal execution location of all nodes. Table 3 shows the expected benefits of a such a scheduler ("POSH-OPT"), for a `comm` command that correlates two 3 GB files stored at two different proxies. It estimates execution time by summing the local execution time of `comm` with the time to transfer one of the files (measured by the time to `scp` the file between the two machines).

Security. POSH allows users to offload parts of shell commands to proxy servers, which could be running directly at the storage layer. POSH currently does not address the security implications of this system design. POSH might allow users to access files such as `/proc/sys` on the storage server which should be restricted. To mitigate this, POSH could ensure that offloaded programs run with limited file access permissions, so they do not access restricted files, and only access files that are specified by the input and output arguments parsed from the annotations. POSH could use sandboxing [20] mechanisms, for example, to restrict users from running offloaded programs that access the network.

Resource management. Since POSH proxy servers could run directly at the storage server, a shared storage server could result in an overloaded CPU and longer latencies for users who expected their commands to take less time since they were offloaded, as well as slowdowns to regular remote filesystem requests. POSH does not currently have policies for load balancing and multitenancy, but could explore policies suggested by prior work [6]. However, initial experiments show that POSH could use a simple policy on the storage server such as monitoring how many cores are in use, and refusing to run programs at the storage server when it is overloaded.

Failure recovery. Currently, POSH does not recover from server-side failures; it does not have mechanisms to migrate or restart jobs if single commands within pipelines fail. Since POSH aims to provide *shell semantics*, which involves streaming data without providing fault tolerance for failed commands, POSH currently does not provide the fault tolerance mechanisms present in standard cluster computing frameworks [17, 31, 61]. However, this is an interesting area of future work: POSH knows exactly what files are being modified or created from the annotations, so POSH could modify programs to write to temporary locations, and only write to the final location when the entire operation is successful.

10 CONCLUSION

I/O-intensive shell pipelines that run over networked storage incur a significant cost from moving data over the network. We present POSH, a framework that accelerates unmodified shell pipelines with I/O heavy components that access networked storage such as NFS. POSH intercepts shell pipelines and moves individual commands closer to the data by offloading them to run on proxy servers closer to the data than the client. POSH uses metadata about shell commands, written once per command, called annotations, that specify information relevant to safely offloading these commands to proxy servers as well as scheduling them to minimize data movement. POSH uses annotations to schedule and automatically parallelize shell pipelines across the client and proxy servers, while maintaining local execution semantics. We showed that POSH can accelerate a wide range of unmodified shell applications running over NFS and allow them to achieve local execution times.

ACKNOWLEDGMENTS

We thank our shepherd, Mahadev Satyanarayanan, and the anonymous ATC reviewers for their invaluable feedback. We are grateful to Shoumik Palkar, Deepak Narayanan, Riad Wahby, Keith Winstein, Liz Izhikevich, Akshay Narayan and members of the Stanford Future Data and SING Research groups for their comments on various versions of this work. This research was supported in part by affiliate members and other supporters of the Stanford DAWN project—Ant Financial, Facebook, Google, Infosys, NEC, and VMware, as well as the NSF under CAREER grant CNS-1651570 and Graduate Research Fellowship grant DGE-1656518. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active disks: Programming model, algorithms and evaluation. In *ASPLOS*, 1998.

[2] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards type inference for JavaScript. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, 2005.

[3] BSD Authors. rsh. https://linux.die.net/man/1/rsh.

[4] FFmpeg Authors. FFmpeg. https://ffmpeg.org/.

[5] Antonio Barbalace, Anthony Iliopoulos, Holm Rauchfuss, and Goetz Brasche. It's time to think about an operating system for near data processing architectures. In *HotOS*, 2017.

[6] Alexandros Batsakis, Randal Burns, Arkady Kanevsky, James Lentini, and Thomas Talpey. CA-NFS: A congestion-aware network file system. *ACM Transactions on Storage (TOS)*, 2009.

[7] Alysson Bessani, Ricardo Mendes, Tiago Oliveira, Nuno Neves, Miguel Correia, Marcelo Pasin, and Paulo Verissimo. SCFS: A shared cloud-backed file system. In *Usenix ATC*, 2014.

[8] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. In *SOSP*, 1993.

[9] Ming Chen, Dean Hildebrand, Geoff Kuenning, Soujanya Shankaranarayana, Bharat Singh, and Erez Zadok. Newer is sometimes better: An evaluation of NFSv4.1. In *SIGMETRICS*, 2015.

[10] Ming Chen, Dean Hildebrand, Henry Nelson, Jasmit Saluja, Ashok Sankar Harihara Subramony, and Erez Zadok. vNFS: Maximizing NFS performance with compounds and vectorized i/o. In *FAST*, 2017.

[11] Alvin Cheung, Owen Arden, Samuel Madden, and Andrew C. Myers. Speeding up database applications with Pyxis. In *SIGMOD*, 2013.

[12] Brent Chun and Andrew McNabb. pssh. https://code.google.com/archive/p/parallel-ssh/.

[13] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[14] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. Query processing on smart SSDs: Opportunities and challenges. In *SIGMOD*, 2013.

[15] John R Douceur, Jeremy Elson, Jon Howell, and Jacob R Lorch. The utility coprocessor: Massively parallel computation from the coffee shop. In *Usenix ATC*, 2010.

[16] Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Zmap: Fast internet-wide scanning and its security applications. In *Usenix Security*, 2013.

[17] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *Usenix ATC*, 2019.

[18] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. Outsourcing everyday jobs to thousands of cloud functions with gg. *Usenix Login*, 2020.

[19] Apache Software Foundation. Hadoop. http://hadoop.apache.org/.

[20] Tal Garfinkel et al. Traps and pitfalls: Practical problems in system call interposition based security tools. In *NDSS*, 2003.

[21] Roxana Geambasu, Amit A Levy, Tadayoshi Kohno, Arvind Krishnamurthy, and Henry M Levy. Comet: An active distributed key-value store. In *OSDI*, 2010.

[22] Git SCM. https://git-scm.com/.

[23] GNU. Program argument syntax conventions. https://www.gnu.org/software/libc/manual/html_node/Argument-Syntax.html, 2020 (Accessed March 28,2020.).

[24] Google. Chromium. https://chromium.googlesource.com/chromium/src, 2020 (Accessed January 4, 2020).

[25] Google. Cloud storage. https://cloud.google.com/storage, 2020 (Accessed May 29, 2020).

[26] Mark S. Gordon, D. Anoushe Jamshidi, Scott Mahlke, Z. Morley Mao, and Xu Chen. COMET: Code offload by migrating execution transparently. In *OSDI*, 2012.

[27] Robert S Gray. Agent Tcl: A flexible and secure mobile-agent system. In *Tcl/Tk Workshop*, 1996.

[28] Mary Hall, Peter Kogge, Jeff Koller, Pedro Diniz, Jacqueline Chame, Jeff Draper, Jeff LaCoss, John Granacki, Jay Brockman, Apoorv Srivastava, William Athas, Vincent Freeh, Jaewook Shin, and Joonseok Park. Mapping irregular applications to DIVA, a PIM-based data-intensive architecture. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, 1999.

[29] T. Haynes. NFS Version 4 Minor Version 2. https://tools.ietf.org/html/draft-ietf-nfsv4-minorversion2-41, 2016.

[30] ImageMagick – convert, edit, or compose bitmap images. https://imagemagick.org/.

[31] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

[32] Zsolt István, David Sidler, and Gustavo Alonso. Caribou: intelligent distributed storage. *VLDB*, 2017.

[33] Jeroen Janssens. *Data Science at the Command Line: Facing the Future with Time-Tested Tools*. O'Reilly Media, Inc., 1st edition, 2014.

[34] Rakesh Jha, Dennis T. Cornhill, and J. Michael Kamrad. Ada program partitioning language: A notion for distributing ada programs. *IEEE Trans. Softw. Eng.*, 1989.

[35] A. D. Joseph, A. F. de Lespinasse, J. A. Tauber, D. K. Gifford, and M. F. Kaashoek. Rover: A toolkit for mobile information access. In *SOSP*, 1995.

[36] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the emerald system. In *SOSP*, 1987.

[37] Chet Juszczak et al. Improving the write performance of an NFS server. In *USENIX Winter*, 1994.

[38] Kevin K. Command line argument parser. https://crates.io/crates/clap, 2019.

[39] Chinmay Kulkarni, Sara Moore, Mazhar Naqvi, Tian Zhang, Robert Ricci, and Ryan Stutsman. Splinter: Bare-metal extensions for multi-tenant low-latency storage. In *OSDI*, 2018.

[40] Redis Labs. Redis. https://redis.io/.

[41] James Lentine, Anshul Madan, and Trond Myklebust. Accelerating nfs with server-side copy. In *FAST*, 2011.

[42] Microsoft. TypeScript. https://www.typescriptlang.org/.

[43] Microsoft. CREATE PROCEDURE (Transact-SQL). https://docs.microsoft.com/en-us/sql/t-sql/statements/create-procedure-transact-sql?view=sql-server-ver15, 2017.

[44] Oracle. Developing and using stored procedures. https://docs.oracle.com/cd/B28359_01/appdev.111/b28843/tdddg_procedures.htm, 2020 (accessed May 27, 2020).

[45] Mark Oskin, Frederic T. Chong, and Timothy Sherwood. Active pages: A computation model for intelligent memory. In *ISCA*, 1998.

[46] Shoumik Palkar and Matei Zaharia. Optimizing data-intensive computations in existing libraries with split annotations. In *SOSP*, 2019.

[47] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A case for intelligent RAM. *IEEE Micro*, 1997.

[48] Google Cloud Platform. A user-space file system for interacting with Google Cloud Storage. https://github.com/GoogleCloudPlatform/gcsfuse, 2020 (Accessed May 29, 2020).

[49] Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. Safe & efficient gradual typing for TypeScript. In *POPL*, 2015.

[50] s3fs fuse. FUSE-based file system backed by amazon S3. https://github.com/s3fs-fuse/s3fs-fuse, 2020 (Accessed May 29, 2020).

[51] s3ql. A full featured file system for online data storage. https://github.com/s3ql/s3ql, 2020 (Accessed May 29, 2020).

[52] Robert Schmid, Max Plauth, Lukas Wenzel, Felix Eberhardt, and Andreas Polze. Accessible near-storage computing with fpgas. In *EuroSys*, 2020.

[53] U.S. Securities, Division of Economic Exchange Commission, and Risk Analysis. Edgar log file data set.

[54] Amazon Web Services. Amazon S3. https://aws.amazon.com/s3/, 2020 (Accessed May 29, 2020).

[55] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A user-programmable SSD. In *OSDI*, 2014.

[56] Ole Tange. GNU Parallel 2018. https://doi.org/10.5281/zenodo.1146014, March 2018.

[57] Vasily Tarasov, Dean Hildebrand, Geoff Kuenning, and Erez Zadok. Virtual machine workloads: The case for new NAS benchmarks. In *FAST*, 2013.

[58] Michael Vrable, Stefan Savage, and Geoffrey M Voelker. BlueSky: A cloud-backed file system for the enterprise. In *FAST*, 2012.

[59] Louis Woods, Jens Teubner, and Gustavo Alonso. Less watts, more performance: An intelligent storage engine for data appliances. In *SIGMOD*, 2013.

[60] N Yezhkova, L Conner, R Villars, and B Woo. Worldwide enterprise storage systems 2010–2014 forecast: recovery, efficiency, and digitization shaping customer requirements for storage systems. *IDC, May*, 2010.

[61] 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 in-memory cluster computing. In *NSDI*, 2012.