Posh: A Data-Aware Shell

DEEPTI RAGHAVAN, SADJAD FOULADI, PHILIP LEVIS, AND MATEI ZAHARIA

Deepti Raghavan is a PhD candidate in computer science at Stanford University, advised by Phil Levis and Matei Zaharia. She focuses on topics in networking and distributed systems. She is interested in optimizing data processing for networked applications. deeptir@cs.stanford.edu

Sadjad Fouladi is a PhD candidate in computer science at Stanford University, working with Keith Winstein on topics in networking, video systems, and distributed computing. His current projects include general-purpose lambda computing and massively parallel ray-tracing systems. sadjad@cs.stanford.edu

Philip Alexander Levis is an Associate Professor of Computer Science and Electrical Engineering at Stanford University, where he heads the Stanford Information Networks Group (SING) and co-directs Lab64, Stanford's electrical engineering maker space. He has a self-destructive aversion to low-hanging fruit and a deep appreciation for excellent engineering. pal@cs.stanford.edu

Running I/O-intensive shell pipelines over the network requires transferring huge amounts of data but relatively little computation. We present Posh, a shell framework that accelerates unmodified shell workflows over networked storage by offloading computation to proxy servers closer to the data. Posh provides speedups ranging from 1.6× to 15× compared to bash over NFS for a wide range of applications.

The UNIX shell is a linchpin in computing systems and workflows. Developers use shell tools not only for data processing with core utilities such as `sort`, `head`, `cat`, and `grep`, but also for programs such as Git, ImageMagick, and FFmpeg. The UNIX shell was designed in a time dominated by local and then LAN storage when file access was limited by disk access times, so networked storage was an acceptable tradeoff. Today, solid-state disks have reduced access times by orders of magnitudes, while networked attached storage remains popular.

Running I/O-intensive shell pipelines over networked storage incurs high overheads. Consider generating a `tar` archive on NFS. The `tar` utility copies the source files and adds a small amount of metadata: the server reads blocks and sends them over a network to a client, which shifts their offsets and sends them back. NFS mitigates this problem by offering compound operations and server-side support for primitive commands such as `cp`, but even something as simple as `tar` requires large network transfers.

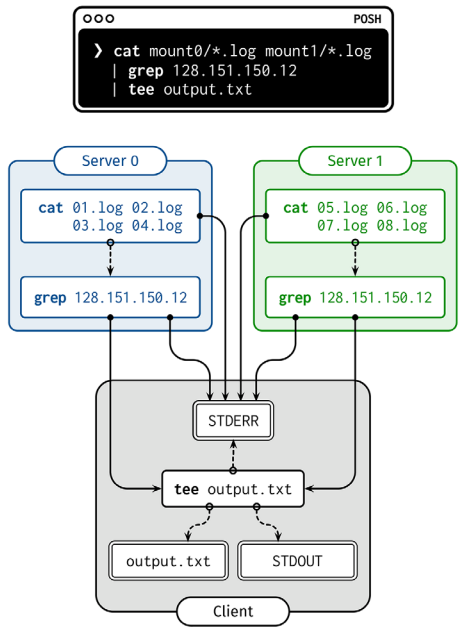
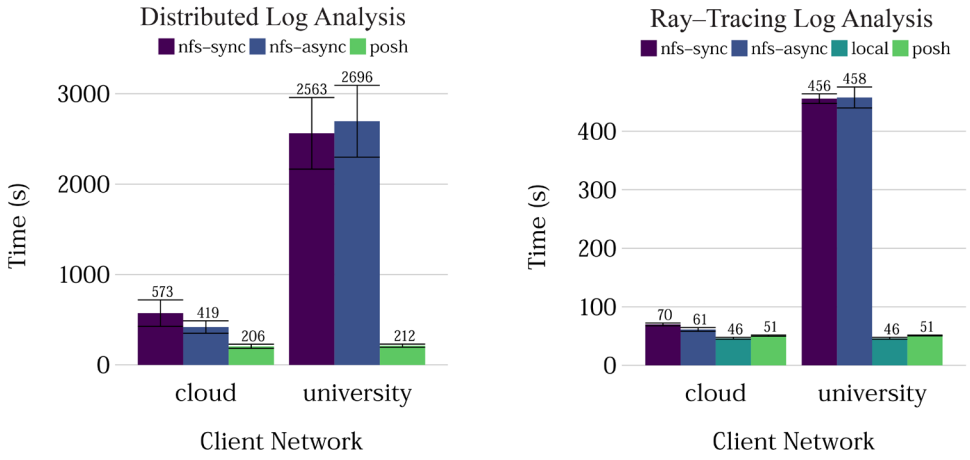


Figure 1: Users can type in unmodified shell workflows to Posh's shell prompt. Posh will transparently schedule and execute individual commands on remote proxy servers closer to the remote data but ensure the entire workflow retains local execution semantics.

Matei Zaharia is an Assistant Professor of Computer Science at Stanford and Chief Technologist at Databricks. He started the Apache Spark project during his PhD at UC Berkeley and has worked on other widely used data analytics and AI software, including MLflow and Delta Lake. At Stanford, he is co-PI of the DAWN lab working on infrastructure for machine learning. Matei's research was recognized through the 2014 ACM Doctoral Dissertation Award, an NSF CAREER Award, and the US PECASE award. matei@cs.stanford.edu



Figures 2 and 3: End-to-end latency of Posh on two 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.

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] is not a new paradigm: systems such as Spark [2], Active-Disks [3], and stored procedures in databases all move computation closer to the data. However, 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 article presents Posh, the "Process Offload Shell," a system that offloads portions of *unmodified* shell workflows to proxy servers closer to the data. A proxy server can run on the actual remote file server 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 identifies parts of shell pipelines that can be safely offloaded to a proxy server and selects which candidates run on a proxy in order to minimize data movement. It then distributes computation across an underlying runtime while maintaining the exact output semantics expected by a local program. Figure 1 shows running a workflow via Posh. The user enters the unmodified workflow at the shell prompt and the output appears at the client's shell as normal, but Posh offloads some of the commands.

Posh is available at https://github.com/deeptir18/posh. This article will cover examples of shell workflows where Posh can be useful, a brief overview of the core ideas behind Posh, and how to get started with the system. For a detailed discussion of the research ideas behind Posh, we refer the reader to our USENIX ATC '20 paper [4].

Examples of Posh

Posh is useful for shell workflows that are I/O bound, have smaller output than input size, are metadata heavy (make many file-system `stat()` requests), or are parallelizable. In this section, we will discuss examples of shell workflows that incur large overheads over networked storage and show that Posh accelerates them to achieve near-local execution time. Figures 2–4 illustrate the execution time of running each of these applications with an NFS mount configured with either `sync` and `async`, and with Posh, over 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"). Posh can offload computation

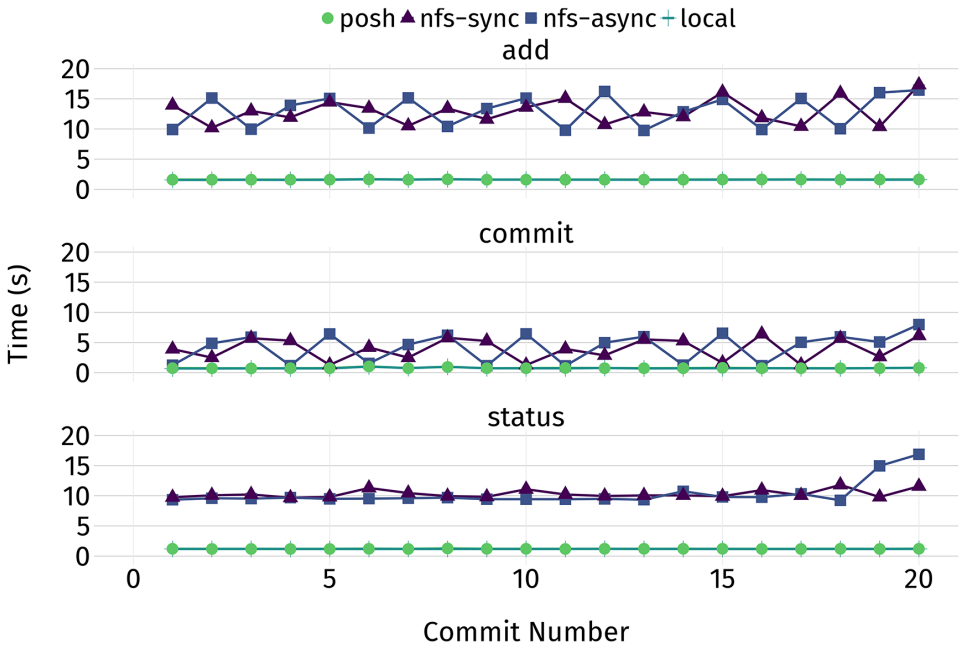


Figure 4: Average latency of 20 `git status`, `git add`, and `git commit` commands run on Chromium repo, of Posh compared to NFS and local execution, for a client in the same cloud datacenter as the storage server. Posh provides up to 10–15× speedups by preventing round trips for file system metadata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

to a proxy server directly running at the NFS servers. Figures 3 and 4 additionally include a baseline that demonstrates local execution time, where the data is stored on a local SSD. Compared to bash over NFS, Posh sees a 1.6–12.7× speedup in the execution time of these applications.

For each of these applications, the shell workflow (the bash script) itself is *completely unmodified;* the workload is just run within a Posh shell environment. Posh can accelerate these workflows because the shell knows metadata about the commonly used shell commands within these workflows, which we will discuss in the next section. We describe each workflow in turn.

Distributed Log Analysis (Figure 2)

This application is based on a workflow where system administrators run analysis on 80 GB of input logs split across five *different storage servers*, to search for an IP address within these logs. The workflow runs `cat` over all of the files and filters for a particular IP with `grep` and then writes the final results, only about 0.8 KB of data, back to a file stored locally at the client. Posh splits the computation across the five machines and aggregates the output in the correct order. By offloading and parallelizing, Posh improves the runtime by 12.7× in the university-to-cloud setting and by 2× in the cloud-to-cloud setting.

Ray-Tracing Log Analysis (Figure 3)

This workflow analyzes the logs of a massively distributed research ray-tracing (computer graphics) system [5] to track a ray (a simulated ray of light) through the workers it traversed.

The analysis first cleans and aggregates each worker's log, 6 GB in total, into one 4 GB file. It then runs `sed` to search for the path of a single ray (e.g., a straggler) across all the workers and stores the output on a file at the client:

```
cat logs/1.INFO | grep "\[RAY\]" | head -n1 | cut -c 7- > \
   logs/rays.csv
cat logs/*.INFO | grep "\[RAY\]" | grep -v pathID | \
   cut -c 7- >> logs/rays.csv
cat logs/rays.csv | sed -n '/^590432,/p' > local/output.log
```

The output of `sed` is much smaller than the 10 GB of data processed. This application is a best-case workload for Posh: it is I/O bound and can be parallelized, and the output is a tiny fraction of the data it reads. Posh achieves an 8× improvement on the university-to-cloud network and no improvement on the cloud-to-cloud network: Posh offloads all the computation and only needs to stream the output of `sed` back to the client. However, the data movement overhead only matters in the university-to-cloud setting, where the network connection is slower.

Git Workflow (Figure 4)

This application imitates a developer's `git` workflow over the Chromium repository. 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`. Figure 4 shows the latency of each command for each of the 20 commits. These commands are extremely metadata-heavy: commands like `status` and `add` check the status of every file in the repository to see if it has been

cat A B C D | grep "foo"

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

grep [splittable_across_input,
      filters_input]:
  PARAMS:
    - type:string,
      size:1
    - type:input_file,
      size:list
```

tar -czf file.tar input_dir

```
tar:
  FLAGS:
    - short:c
    - short:z
  OPTPARAMS:
    - short:f,
      long:file,
      type:output_file,
      size:1
  PARAMS:
    - type:input_file,
      size:list
```

tar -xzf file.tar

```
tar [needs_current_dir]:
  FLAGS:
    - short:x
    - short:z
  OPTPARAMS:
    - short:f,
      long:file,
      input_file,
      size:1
```

Figure 5: Example annotations for `cat`, `grep`, and `tar`. Most of the information in the annotations tells Posh information about the possible arguments for each command and their syntax. They contain type assignments for each argument, which tell Posh how the argument will be used as well as other information used for scheduling and automatic parallelization. `tar` requires more than one annotation because `tar -x` and `tar -c` invocations have conflicting type semantics: `-f` is an `input_file` in one case and an `output_file` in the other.

modified. When run over a networked file system, this incurs many round trips. In the cloud-to-cloud setting, this causes Posh to achieve 10–15× improvement over NFS. Running `git status` took up to two hours in the university-to-cloud setting, so we omitted this network for this application.

To enable Posh's acceleration of a shell workload, the user must provide metadata about the individual shell commands the workflow uses. This metadata, called *annotations*, allows Posh to determine which files these commands access, so it can further schedule the workflow across the underlying runtime. The next section will discuss annotations in more detail.

Transparently Offloading Shell Computation: Annotations

Annotations summarize information to Posh about individual shell commands, such as `tar`, `cat`, or `grep`. Posh's key insight is that many shell workflows only read and write to files specified in their command-line invocation, so Posh can deduce which files a workflow accesses by understanding which arguments correspond to files. Annotations contain a list of possible arguments and whether they correspond to files, so Posh can understand which files an arbitrary invocation of a command would access. Additionally, annotations contain information relevant to scheduling the workflow.

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`. Posh must understand which files (if any) each command accesses and where these files live, so 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 preceded by `-f`, followed by an input file argument not preceded by a short option. The third command, also `tar`, takes an input file argument preceded by `-f` and implicitly takes its output argument as the current directory. Finally, `git` also implicitly relies on the current directory as a dependency.

Secondly, in order to produce an execution schedule that reduces data movement, Posh must understand the relationship between the inputs and outputs of a command. In the `cat | grep` example, if the argument to `cat` is a remote file, to minimize data movement, Posh can offload both `cat` and `grep` since `grep` filters its input. Finally, for applications like the distributed log analysis application discussed previously, where the input files for a command live on different mounts, Posh needs to know how to safely parallelize the command in order to be able to offload it at all. However, parallelization is not safe for all commands: `wc`, for example, "reduces" the input, as opposed to commands like `cat` or `grep`, which merely map over the input. Posh's annotations summarize file dependencies, data movement semantics, and parallelization semantics for commonly used commands.

Figure 5 shows examples of annotations, for `cat`, `grep`, and `tar`. Most of the information in the annotations summarize the semantics for the arguments for each command, or information

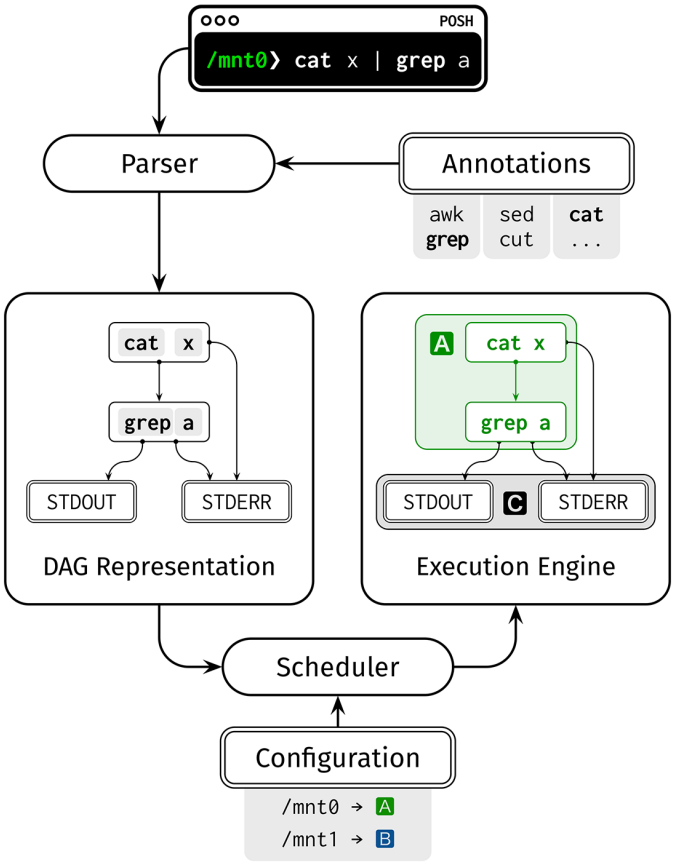


Figure 6: In Posh's main workflow, a shell command is passed to the parser, which uses the annotations to generate and schedule a DAG representation of the command. The DAG includes which machine—A, B, or C (client) here—to run each command on. The execution engine finally runs the resulting DAG.

that is summarized in the documentation pages for these commands. Moreover, they contain a type assignment for each argument: `input_file`, `output_file`, or `string`. For `cat`, the `splittable` keyword indicates to Posh that `cat` can be split in a data parallel way across its arguments, as long as the outputs are stitched together in the correct order. For `grep`, the `splittable_across_input` keyword indicates that `grep` can be parallelized across its standard input. As mentioned before, the `-f` argument indicates an `input_file` for a `tar -x` invocation but an `output_file` for a `tar -c` invocation. To resolve this, Posh allows multiple annotations per command, per type of invocation, and tries each until it finds an annotation that matches the current command invocation.

We envision that developers can share annotations for popular commands, so users do not necessarily need to write their own annotations. These annotations are inspired by recent proposals to annotate library function calls for automatic pipelining and parallelization [6]. Please see our research paper [4] for a more detailed overview of the Posh annotation interface.

Distributed Scheduling and Execution

This section briefly explains how Posh uses the annotations to schedule and execute shell workflows, summarized in Figure 6. The Posh parser turns each pipeline (each line of a shell workflow, 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. 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 greedy 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. Our research paper [4] contains more details on the scheduling algorithm.

Getting Started with Posh

This section details the steps to running and using Posh.

0. Running the Posh servers

The administrator who controls the proxy server must run the Posh server binary, which allows it to receive requests to offload computation on behalf of a single remote file-system mount. The proxy server just needs read and write access to this folder; it need not run at the storage server itself. Invoking the server, shown below, requires specifying the absolute path for the mount being accessed and a temporary directory for writing the output of intermediate computation.

```
admin@~$ $POSH_SRC/target/release/server --folder /mnt/logs \
  --tmpfile /tmp/posh
```

1. Posh client configuration

The client needs to provide a file that contains annotations for any commands the client wants to accelerate. It must also have a list of proxy servers associated with client file-system mounts. The configuration file, shown below, maps IP addresses to the corresponding mount, written as an absolute path.

```
mounts:
      "255.255.255.0": "/home/user/remote_mount1"
      "255.255.255.1": "/home/user/remote_mount2"
```

2. Running the client shell

Posh provides two client binaries: one that provides a shell prompt and one that runs scripts by running each line in the script. To run the binary that provides a shell prompt, the client can run:

```
deeptir@~$ $POSH_SRC/target/release/shell-client \
  --annotations_file <annotations_file> --mount_file \
  <config_file>
posh>>>$ <ENTER COMMANDS>
```

3. Running applications

After running the shell, users can run unmodified shell workflows as normal. For example, the user could type in the following workflow from the distributed log analysis example discussed previously:

```
posh>>> $ cat mount0/logs/*.csv mount1/logs/*.csv \
   mount2/logs/*.csv mount3/logs/*.csv mount4/logs/*.csv \
   | grep '128.151.150' > $LOCAL_FILE
```

Conclusion and Next Steps

We have described Posh, a framework that transparently distributes I/O-heavy shell computation that operates on remote data, by pushing computation to run closer to the data. Posh uses annotations, a model of shell programs, to automatically infer what files an arbitrary command line will read and write to in order to schedule computation across proxy servers. Posh and its annotations provide a model of commands that enable rewiring their dependencies to direct output over the network rather than to a UNIX pipe while retaining local execution semantics. While Posh currently uses this model to transparently schedule and offload commands across proxy servers to push code closer to the data, it could in the future provide more optimal scheduling or even failure recovery. Consider programs that access files from two different locations that cannot be parallelized, such as `comm`. Instead of running them at the client, Posh could run them on one of the servers but stream or transfer the necessary inputs beforehand. To provide failure recovery semantics, Posh could rewrite workflows to write to temporary locations and only write to the final location when the entire operation is successful. For more information on this project, including our research paper, the code, and quick-start guides, please visit our GitHub page, https://github.com/deeptir18/posh.

Acknowledgments

We thank our ATC 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] A. Barbalace, A. Iliopoulos, H. Rauchfuss, and G. Brasche, "It's Time to Think about an Operating System for Near Data Processing Architectures" in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS '17)*, pp. 56–61.

[2] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing," in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI '12)*, pp. 15–28.

[3] A. Acharya, M. Uysal, and J. Saltz, "Active Disks: Programming Model, Algorithms, and Evaluation," in *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '98)*, pp. 81–91.

[4] D. Raghavan, S. Fouladi, P. Levis, and M. Zaharia, "POSH: A Data-Aware Shell," in *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*, pp. 617–631.

[5] S. Fouladi, F. Romero, D. Iter, Q. Li, S. Chatterjee, C. Kozyrakis, M. Zaharia, and K. Winstein, "Outsourcing Everyday Jobs to Thousands of Cloud Functions with gg," *;login:*, vol. 44, no. 3 (Fall 2019), pp. 5–11.

[6] S. Palkar and M. Zaharia, "Optimizing Data-Intensive Computations in Existing Libraries with Split Annotations," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*, pp. 291–305.