Practically Correct, Just-in-Time Shell Script Parallelization

Konstantinos Kallas
University of Pennsylvania

Tammam Mustafa
MIT CSAIL

Jan Bielak
XIV Staszic High School

Dimitris Karnikis
Aarno Labs

Thurston H.Y. Dang*
MIT CSAIL

Michael Greenberg
Stevens Institute of Technology

Nikos Vasilakis
MIT CSAIL

Abstract

Recent shell-script parallelization systems enjoy mostly automated speedups by parallelizing scripts ahead-of-time. Unfortunately, such static parallelization is hampered by dynamic behavior pervasive in shell scripts—*e.g.*, variable expansion and command substitution—which often requires reasoning about the current state of the shell and filesystem.

We present a just-in-time (JIT) shell-script compiler, PASH-JIT, that intermixes evaluation and parallelization during a script's run-time execution. JIT parallelization collects run-time information about the system's state, but must not alter the behavior of the original script and must maintain minimal overhead. PASH-JIT addresses these challenges by (1) using a dynamic interposition framework, guided by a static preprocessing pass, (2) developing runtime support for transparently pausing and resuming shell execution; and (3) operating as a stateful server, communicating with the current shell by passing messages—all without requiring modifications to the system's underlying shell interpreter.

When run on a wide variety of benchmarks, including the POSIX shell test suite, PASH-JIT (1) does not break scripts, even in cases that are likely to break shells in widespread use; and (2) offers significant speedups, whenever parallelization is possible. These results show that PASH-JIT can be used as a drop-in replacement for any non-interactive shell use, providing significant speedups without any risk of breakage.

1 Introduction

The UNIX shell is an environment for composing programs from components written in a variety of programming languages. Coupled with UNIX's toolbox philosophy [41], this language agnosticism makes the shell a popular choice for succinctly expressing tasks that involve data processing, system orchestration, and other automation. Recent systems [52, 55, 63] accelerate such tasks by exploiting data parallelism: using *ahead-of-time* (AOT) analysis and transformation, these systems parse, analyze, and transform shell scripts into new scripts that execute in parallel.

Unfortunately, AOT parallelization quickly becomes intractable due to the dynamic nature of the shell: dynamic features such as variable expansion and command substitution, pervasive in shell scripts, generate and consume values at run-time while depending on and interacting with the broader environment—*i.e.*, the filesystem, the environment variables, and the shell interpreter itself. Additionally, modern shells offer several different configurations and execution modes, leading to complex behaviors described in hundreds of pages of POSIX standardese [2]. The complexity of these interactions and their side-effects lead existing parallelization tools to an unavoidable trade-off between (1) being conservative, aborting on scripts that use dynamic features, or (2) being unsound, possibly breaking scripts during parallelization. Recent systems [52, 55, 63] tend to be conservative—operating only on fully expanded shell pipelines and having a hard time even on simple uses of variables (see §2).

This paper presents PASH-JIT, a production-grade just-in-time (JIT) shell-script compiler aimed at non-interactive parallelization: PASH-JIT focuses on three practical (but conflicting) goals: (G1) run-time-informed parallelization: PASH-JIT leverages run-time information to parallelize script fragments that depend on state that is statically indeterminable; (G2) full behavioral equivalence: PASH-JIT is aware of the full set of dynamic behaviors present in POSIX shells, producing results that are indistinguishable from the sequential execution on the system's shell interpreter; (G3) loose shell coupling: PASH-JIT avoids modifications to the system's underlying shell interpreter, eschewing practical problems (*e.g.*, maintaining two Bash implementations). PASH-JIT behaves as a drop-in shell shim enhancing any non-interactive shell use, providing significant speedups without any risk of breakage.

PASH-JIT's key insight is to parallelize scripts just-in-time: by intermixing evaluation and parallelization during a script's execution, PASH-JIT collects and uses the latest possible run-time information about the state of an expression's vari-

*The author is now at Google but the work was done while he was at MIT.

ables, the shell, and the filesystem. PASH-JIT parallelizes script fragments when it is safe to do so, resolving indeterminacies in the broader environment on the fly. Unfortunately, low-overhead run-time-informed parallelization (G1) is particularly challenging to implement in view of full behavioral equivalence (G2) and loose shell coupling (G3). PASH-JIT addresses this conundrum using: (1) a dynamic interposition framework, guided by an instrumentation preprocessing pass; (2) support for reentrance, transparently pausing and resuming the execution of the underlying shell interpreter at run-time; and (3) a stateful, long-lived compilation server that communicates with the current shell by exchanging messages. A 9K-LOC implementation and several run-time optimizations—*e.g.*, dynamic independence discovery, commutative-aware parallelization—complete the picture.

We apply PASH-JIT to a variety of benchmarks, ranging from scripts collected from the wild to the POSIX test suite. PASH-JIT behaves identically to Bash 4.4.20(1) on 406 out of 408 applicable POSIX tests; matching Bash is a significant achievement even for a non-parallelizing shell—shells in widespread use differ on much larger subsets of tests. PASH-JIT offers speedups up to 33.7× over Bash on a 64-core machine (improving the state of the art [63] by 2× on average), notably parallelizing scripts that prior work failed to parallelize due to dynamic behaviors.

The paper begins by exemplifying dynamic shell features and the application of PASH-JIT's techniques (§2). Sections 3–6 describe PASH-JIT's main contributions:

- *A dynamic interposition framework for the shell*: A just-in-time analysis and optimization subsystem enables safe and effective parallelization during the execution of a script, dealing with the challenges of dynamic shell-script behavior. A first pass determines where to insert calls to a parallelizing optimizer in a given input script (§3), which is then invoked on-the-fly while the script is executing (§4).
- *A stateful, parallelizing compilation server*: PASH-JIT queries a long-lived parallelization server at run-time to compile script fragments. This model improves run-time efficiency by avoiding startup costs on every JIT invocation, and enables additional run-time optimizations for (1) executing independent regions in parallel, and (2) pipelining compilation and execution. The core of the server has been modelled and formally verified using SPIN [29] (§5).
- *Commutativity-aware optimization*: Additional compilation optimizations target commands that are commutative with respect to their input, along with parallelizing transformations and run-time primitives that improve the run-time performance of scripts that contain such commands (§6).

The paper then presents PASH-JIT's evaluation (§7) and related work (§8), before concluding (§9). PASH-JIT is MIT-licensed open-source software supported by the Linux Foundation at https://github.com/binpash/.

2 Example & Overview

Below is a shell program that downloads a compressed archive of text files (books from Project Gutenberg), extracts them in a directory, and then performs an analysis to find the frequencies of all words of a specific form.

```
IN=${IN:-$TOP/pg}
mkdir "$IN"
cd "$IN"
echo "Download will take some time, be patient..."
wget "$SOURCE/data/pg.tar.xz"
if [ $? -ne 0 ]; then
  echo "Download failed!"
  exit 1
fi
cat pg.tar.xz | tar -xJ

cd "$TOP"
OUT=${OUT:-$TOP/output}
mkdir -p "$OUT"
for input in $(ls "$IN"); do
  cat "$IN/$input" | tr -sc '[A-Z][a-z]' '[\012*]' |
    grep '^....$' | sort | uniq -c > "$OUT/$input.out"
done
```

The program makes pervasive use of the shell's dynamic features. For example, it uses environment variables such as `$TOP`, variable expansion like `${OUT:-$TOP/output}` to assign default values, command substitution `$(...)` as part of the loop condition, and state reflection on the file system by running `ls` on `$IN` (itself resolved dynamically).

None of the values of these variables can be known ahead of time just by analyzing the program's source code. They become known only at run-time, when the shell interpreter reaches these points in the program's execution. A sound AOT compiler such as PASH-AOT [63] or POSH [52] would fail to parallelize—foregoing all the performance benefits of data-parallel execution spread across many files in `$IN`.

PASH-JIT instead takes a JIT approach that interjects parallelization opportunities during and throughout the script's execution (Fig. 1).

Dynamic interposition (§3): PASH-JIT first uses a preprocessing step to instrument all potentially optimizable program regions with calls to the JIT engine. PASH-JIT chooses regions to maximize the potential benefits of parallelizing them: intuitively, commands and pipelines can yield significant benefits, whereas word expansion, control flow, and variable assignments are operations that do not perform heavy computation and can therefore be left as they are. PASH-JIT's preprocesor and compiler both make extensive use of parsing/unparsing of shell source code, implemented as a new parsing library. After PASH-JIT has inserted calls to the JIT engine, it invokes the user's shell interpreter to execute this transformed script. During this execution, the JIT engine calls the parallelizing compiler at run-time—right before the execution of each fragment, when the state of the shell and the file system have already been resolved. The transformed program

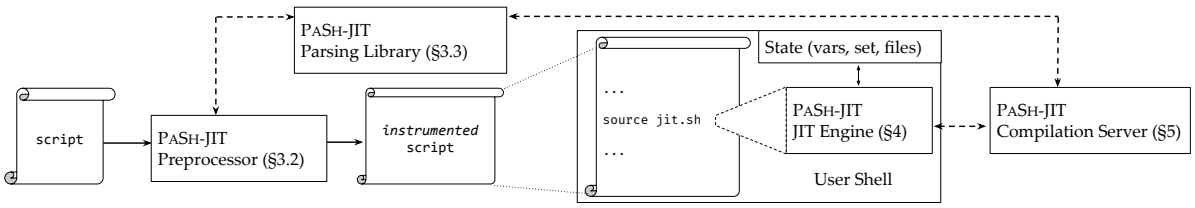

Fig. 1: PaSh-JIT overview. PaSh-JIT instruments scripts with calls to the JIT engine, which passes program fragments to the compilation server at run-time.

maps original commands to *regions*—for example, `region8` corresponds to the `cd` call and `region10` corresponds to the pipeline in the **`for`** loop.

```
source jit.sh "$region8"                    # cd $TOP
OUT=${OUT:-$TOP/output}
source jit.sh "$region9"              # mkdir -p "$OUT"
for input in $(ls "$IN"); do
  source jit.sh "$region10" # cat "$IN/$input" | ...
done
```

The command `source jit.sh "$regionN"` invokes the JIT engine passing as argument the corresponding fragment. The `source` built-in retains the same shell environment, reflecting any effects directly into the current environment.

JIT engine (§4): Internally, the JIT engine first saves the state of the shell at that point in the script's execution to isolate it from compilation—protecting the shell from the JIT engine and protecting the JIT engine from obscure shell configurations. PaSh-JIT then invokes the compiler to attempt to parallelize the fragment. If the compiler succeeds, PaSh-JIT runs the resulting parallel fragment; if not, it runs the original, unmodified region. In both cases, PaSh-JIT will first restore the state of the shell before executing the fragment. Whether the compiler succeeds or not depends on the properties of the fragment's code—*e.g.*, PaSh-JIT will reject `region8` due to the side-effectful `cd` command, but will accept `region10` compiling `grep` and `sort` into the parallel fragment below:

```
c_split /tmp/fifo8 /tmp/fifo9 /tmp/fifo10 &
c_wrap 'grep "^....$"' </tmp/fifo9 >/tmp/fifo11 &
c_wrap 'grep "^....$"' </tmp/fifo10 >/tmp/fifo12 &
c_strip </tmp/fifo11 >/tmp/fifo13 &
c_strip </tmp/fifo12 >/tmp/fifo14 &
sort </tmp/fifo13 >/tmp/fifo15 &
sort </tmp/fifo14 >/tmp/fifo16 &
eager.sh </tmp/fifo15 >/tmp/fifo17 &
eager.sh </tmp/fifo16 >/tmp/fifo18 &
sort -m /tmp/fifo17 /tmp/fifo18 >/tmp/fifo19 &
```

The resulting compiled fragment executes in a data-parallel fashion: data is split by PaSh-JIT primitives, then fed to multiple instances of `grep` and `sort` runnning in parallel, and finally merged at the end of the parallel execution.

Dependency untangling (§5): While the JIT engine operates as if invoked on every region, PaSh-JIT is engineered to spawn a long-running stateful compilation server just once, feeding it compilation requests until the execution of the script completes. This design has two benefits: (1) it reduces run-time overhead by avoiding reinitializing the compiler for each compilation request; and (2) it allows maintaining and querying past compilation results when compiling a new fragment. The latter allows PaSh-JIT to untangle dependencies *across* regions, finding and exploiting opportunities for cross-region parallel execution. For example, the server's first invocation on `region10` (the body of the loop) determines that all prior successfully compiled regions have finished executing. PaSh-JIT can thus simply run the loop in the background and continue with the second iteration in a task-parallel fashion, without waiting for the first iteration to complete executing. During the second invocation on `region10`, PaSh-JIT will use the dependency state to determine that while the previously compiled fragment is still running, the input and output files of the two regions are completely independent and can thus be executed in parallel: our loop is now pipelined! PaSh-JIT goes beyond intra-region data parallelism: the JIT enables inter-region task parallelism by resolving dependencies and confirming they are independent.

Commutativity analysis & compilation (§6): The first goal when compiling fragments such as `region10` is to identify command sequencies that are parallelizable using a divide-and-conquer strategy. Due to the shell's order-aware nature [28], naive divide-and-conquer would need to (1) read the entire input before splitting it, to determine the exact size of each batch, leading to stalled pipeline parallelism; and (2) wait until all of its predecessors have consumed their batch, storing data after split on disk, to ensure that all parallel nodes will not wait for their input.

While these overheads are unavoidable in the general case, and are indeed incurred by prior systems [55, 63], they can fortunately be alleviated for subsets of parallelizable commands. Two such subsets include (1) stateless commands such as `grep -c '^....$'` that operate in a line-oriented fashion, meaning that data-parallel copies of these commands can combine their partial output using a reordering operation, and (2) commutative commands such as `sort -u` that produce equivalent output regardless of the order of the input lines. PaSh-JIT leverages this insight to achieve more effective parallelization by splitting into streaming micro-batches (using `c_split`) in a round-robin fashion—avoiding the overheads of reading all the input before splitting and of unnecessary storage to disk. It also wraps stateless commands to strip and re-add the microbatch headers (using `c_wrap`) and removes these headers completely before commutative commands (using `c_strip`).

Zooming back out: Fundamentally, PASH-JIT is neither a shell nor requires modifications to a user's shell. Rather, it is an interposition shim located between a user and their shell, deciding whether to optimize parts of the user script on the fly, using information about the execution state of the shell interpreter. PASH-JIT combines several techniques that allow harnessing speedups not attainable by ahead-of-time parallelization on both dataflow-only scripts and larger scripts with dynamic components and complex control flow; all of this, without modifying the behavior of the original script.

3 Interfacing With the Shell

PASH-JIT works by interposing on the shell, effectively rewriting invocations to external commands. Challenges arise due to the shell's complex semantics and its intricate internal state, both of which complicate side-effect-free interposition. The shell uses a string-based, bi-modal semantics: commands undergo *expansion*, a string rewriting phase where variables, tildes, and globs are processed before the commands undergo *evaluation*. Both modes have complex semantics heavily involved with the shell's state [24]; any rewriting must be careful to leave the shell's state unaltered.

3.1 Dynamic Interposition

To understand PASH-JIT's interposition, we must first understand the simpler structure of ahead-of-time (AOT) parallelization. While preserving a script's original behavior, AOT parallelization rewrites calls to external commands to exploit parallelism. External commands consume substantially more time and resources than shell language features (like expansion or loops) during the execution of typical shell scripts.

AOT parallelization centers around the identification of *parallelizable regions*—script fragments that may be safely parallelized to yield performance gains. Semantically, parallelizable regions only contain a set of command invocations that satisfy the following conditions: (1) they have no file dependencies (*interference-free*), *i.e.*, all commands can execute concurrently without affecting each other, (2) they communicate with each other using explicit UNIX channels (fifos/pipes); (3) they are *pure*, only affecting the environment by reading and writing to files, *i.e.*, they do not modify environment variables; and, (4) they are fully expanded. An AOT compiler parses and transforms these regions to an intermediate representation such as directed-acyclic [52] or dataflow [63] graphs, abstracted as functions that take a set of input files and produce a set of output files [28]. It then applies transformations on these graphs to perform the original computation in parallel.

PASH-JIT works similarly, but applies these steps at a much finer granularity and in a dynamic, online fashion. PASH-JIT's dynamic interposition mechanism pauses execution right before each parallelizable region, compiling it to an efficient and equivalent parallel script fragment, and executing that instead. Working dynamically means PASH-JIT has up-to-date information and can achieve increased parallelism.

3.2 Preprocessor

Dynamic script interposition without any shell-interpreter modifications is hard. To achieve this, PASH-JIT opts for a light-weight script instrumentation pre-processing step: it marks *possible* parallelizable regions with code that dynamically determines whether or not to invoke the compiler.

The intuition behind PASH-JIT's preprocessor is that a syntactic analysis of a shell script is enough to suggest potential parallelizable regions. This analysis is imprecise: there is no way to determine whether a command invocation will be pure ahead of time. Its goal however, is not to find parallelizable regions exactly, but rather to find potential compilation sites—PASH-JIT sorts out the details at run-time, using up-to-date information about the system's state.

There is a trade-off when choosing the right size for these regions: the larger the region, the more opportunities exist for analysis and optimization but the less likely it is for the entire region to be parallelizable. PASH-JIT targets a middle-ground: maximal syntactic schedule-free regions—*i.e.*, command sequences composed using shell primitives that do not impose scheduling restrictions. By focusing on maximal schedule-free regions, PASH-JIT minimizes the number of compiler invocations and maximizes the cross-command parallelization opportunities for the compiler. Note that schedule-free regions underapproximate interference-free regions (§3.1), *e.g.*, two commands composed in sequence `;` that write to different files do not interfere but are not syntactically schedule-free.

The preprocessor finds these maximal regions by searching the AST bottom-up, combining schedule-free subtrees when they are composed using constructs that do not introduce scheduling constraints (*e.g.*, `&`, `|`). When a region cannot outgrow a certain subtree, it is replaced with a call to the JIT engine. If successfully compiled, a region is transformed to a dataflow graph—a convenient and well-studied computation model amenable to transformation-based optimizations [28]. The instrumented AST resulting from the compilation is finally translated (unparsed) back to shell code and sent over to the underlying shell for execution.

3.3 Parsing Library

Parsing and unparsing are key operations in PASH-JIT and must address several challenges.

PASH-JIT parses lines of shell script as they come in, and unparses lines in order to execute them in the user's shell; it also uses parsing and unparsing during compilation, when the compilation server emits an optimized string or passes

strings to the shell for expansion. PASH-JIT initially used `libdash`—an OCaml library built using the dash parser and part of Smoosh [23, 24]—that caused two main issues. First, `libdash`'s unparsing introduced several bugs, as at the time it was used by the libdash project primiarly for testing and diagnostics—had much of its was functionality untested. Second, `libdash` parsing introduced significant run-time overhead due to (1) the cost of forking and executing the OCaml binary, (2) overheads due to serialization and deserialization during communication, and (3) suboptimal implementation. Run-time overheads were a significant concern due to PASH-JIT's online JIT parallelization, which intermixes calls to the compiler during the program's execution—bringing parsing and unparsing into the critical path of program execution.

To address these issues, PASH-JIT reimplements its own version of `libdash` in Python called Pylibdash. The Pylibdash implementation develops Python bindings for the dash parser and completely reimplements unparsing—adding 0.9k LOC of Python over libdash, structured as a separate library usable by other projects. The Pylibdash implementation contains several optimizations such as caching, inlining, and careful array appending to avoid some accidentally quadratic costs in the original implementation. As a side benefit, using a custom implementation reduces the number of dependencies required by PASH-JIT's installation.

4 The JIT Engine

The PASH-JIT preprocessor identifies possible parallelizable regions and instruments the shell script to dynamically determine whether they can be optimized by invoking the JIT engine. The JIT engine faces two key challenges: it must not change the original script behavior, and it must run with low overhead as it is invoked multiple times per script.

The JIT engine is a reflective shell script: by inspecting the state of the shell and that of the broader system, it can transparently work with the compiler to determine whether or not to parallelize a script (Fig. 2). When running scripts with PASH-JIT, it is helpful to think of the shell as having two modes: (1) conventional shell mode, where scripts execute in the original shell context, and (2) PASH-JIT mode, where the runtime reflects on shell state and invokes a compiler to determine whether to execute the original or an optimized version of the target region. To switch from shell mode to PASH-JIT mode, the JIT engine must carefully save the state of the user's shell; to switch back, it must carefully put things back just the way they were. A shell's state is quite complex: beyond saving and restoring variables, the runtime must account for various shell flags along with other internal shell state (*e.g.*, the previous exit status, working directory).

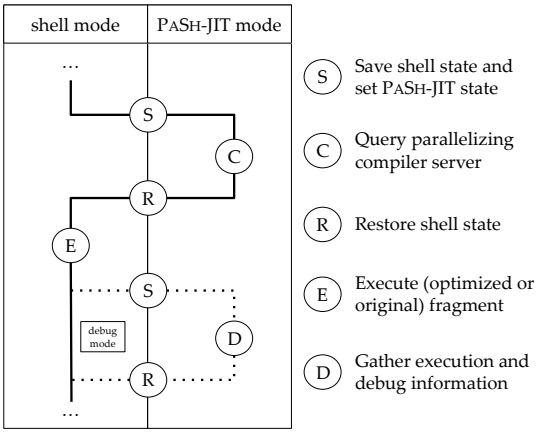

Fig. 2: Overview of JIT engine stages.

4.1 JIT Stages

When running normally, the JIT engine transitions into and out of PASH-JIT mode once per possible parallelizable region (Fig. 2): the JIT engine saves the shell state and switches into PASH-JIT mode (S); then it tries to compile the current fragment (C); whether successful or not, the JIT engine restores the state and switches back to shell mode (R); and, finally, either the original fragment or the optimized parallel version is executed (E). With debugging enabled, the JIT engine switches back into PASH-JIT mode (S) to collect debugging information (D), restoring again afterwards (R).

Saving (S): When entering a possible parallelizable region, the first step is to save the shell state—recording the previous command's exit status, the values of environment variables, and the configuration of the shell—essentially, a continuation that can later be restored to execute the target fragment. Once the state is saved, PASH-JIT mode reconfigures the user's shell to avoid changing script behavior. For example, if the user's shell has the `-e` "exit on error" flag set, the shell should exit immediately when a command (or a pipeline) returns a non-zero exit status, unless that command is in a checked position (*e.g.*, after `!`, or in the condition of an **`if`** or **`while`**) [2]. However, failing commands should not stop the JIT itself, so `-e` is unset (and will be restored later in (R)).

Compilation (C): With the state saved and shell reconfigured, PASH-JIT tries to compile the script fragment: the JIT engine queries the compilation server (§5) with the script fragment (already parsed during preprocessing) along with the saved shell state, so that the compilation server can try to expand all of the words in the fragment. The server responds to indicate whether it managed to optimize the fragment.

Restoring (R): Whether or not compilation was successful, the JIT engine exits PASH-JIT mode, restoring the continuation saved earlier (S) to prepare to execute the fragment. One particular challenge in this mode is to restore state while

accommodating different shell modes. Suppose PASH-JIT is in `-e` mode, trying to run some possible parallelizable region, and the command *before* this region exited with status 47 in a checked position, *i.e.*, without forcing the shell to exit. The JIT engine saves the exit status so as to not overwrite it. The fragment may depend on the exit status, so PASH-JIT needs to restore it before running the fragment. But it must be careful—simply running `(exit 47)` would force the shell to exit. Thus PASH-JIT runs the subshell in a checked position:

```
if (exit "$pash_previous_exit_code"); then
    source "$fragment"; ...
else
    source "$fragment"; ...
fi
```

This odd code ensures that the fragment (in identical branches) has access to the previous exit status (in the checked, conditional position of the **`if`**) without exiting when `-e` is set.

Execution (E): Back in shell mode, the JIT engine executes the fragment. If the compiler was successful, then the JIT engine selects the optimized script fragment. If the compiler failed, the JIT engine falls back to the original fragment. Either way, control flows back to the original shell.

Debug mode (S) (D) (R): When PASH-JIT is in debugging mode, the JIT engine will re-enter PASH-JIT mode after execution (E) in order to log information about the script, such as execution time and exit status. Standard execution skips this extra save/restore cycle.

5 Parallelizing Compilation Server

For each possible parallelizable region, the JIT engine queries the compiler: can this region actually be optimized? To answer this question, PASH-JIT builds on ideas from the PASH-AOT [63] dataflow compiler (§5.1). As ever, it focuses on preserving behavior and minimizing overhead.

To preserve correct behavior in the face of the shell's dynamism, PASH-JIT expands each script region prior to compilation (§5.2). To minimize overhead due to fixed startup costs—*e.g.*, initialization, dependency loading, logging setup, and output file arrangement—PASH-JIT packages the new compiler as a stateful compilation server communicating via UNIX domain sockets.[1]

The compilation server is also augmented to support a larger set of optimization opportunities, by storing and using information from one compilation to help another. PASH-JIT's long-lived compilation server achieves these additional optimizations by allowing parallelizable regions that work on independent inputs and outputs to be run in parallel (§5.3) and by learning to improve its parallelism configuration from past compilations (§5.4).

[1] We experimented with both socket and FIFO-based communication, but we saw no significant performance differences.

5.1 Command Annotations

PASH-JIT uses the command annotation and specification framework introduced by PASH-AOT [28, 63], extended to also indicate whether a command invocation is commutative (§6.1). This framework provides information about a command invocation's parallelizability class, inputs, and outputs. A command annotation can be used to extract high-level information about a specific command invocation, *i.e.*, a precise instantiation of its flags, options, and arguments. For example, annotations determine whether a given command invocation is pure and what its inputs and outputs are.

PASH-JIT uses this annotation framework to extract information for commands that are not shell builtins—that is, commands like `sort` and `grep`. Annotations enable analyses and transformations over command invocations by lifting them to pure dataflow nodes in a dataflow intermediate representation (IR) [28]. For example, `grep -f dict.txt src.txt > out.txt` is a dataflow node with two input files (`dict.txt` and `src.txt`) and one output file (`out.txt`), which are all extracted from the annotation of the `grep` command. Annotations also describe parallelization opportunities, *e.g.*, `grep "pattern" src.txt` processes each line of `src.txt` independently, and so it can be parallelized.

5.2 Early, Pure Expansion

PASH-AOT can only attempt to compile script fragments where all words are completely expanded. Running dynamically, PASH-JIT goes beyond PASH-AOT by expanding words according to the current state of the system (shell, file system, *etc.*).

One way to achieve expansion would be for PASH-JIT to maintain a "mirror" Bash process when initializing, which it could then query with any word to expand using `echo`. Every time PASH-JIT would query the compilation server with a fragment, it would also provide the latest state of the shell, which would in turn be passed to the mirror process to ensure it reflects the latest state. This expansion method would be correct, as it would leverage the underlying shell. It would, however, be expensive, since each fragment contains many unexpanded words and each unexpanded word would have to be expanded using its own `echo` command—leading to unnecessary run-time costs.

PASH-JIT avoids the overhead of a mirror shell by performing its own expansion, relying on the optimistic nature of the JIT engine (§4): if most common forms can be expanded in the compiler itself, the compiler will succeed often without incurring interprocess communication overheads; if expansion fails, PASH-JIT will just run the original fragment. Armed with this insight, PASH-JIT implements a subset of expansion in the compilation server itself. PASH-JIT's custom expansion is *purely* functional, in that it does not affect shell state by setting variables or running command substi-

tutions. The expansion routine is implemented in less than 300 LOC of Python, and reduces the compilation overhead significantly (§7). Expansion takes the host shell's configuration and expands common, safe expansions in as many positions as possible—in simple commands, pipelines, and other parallelizable regions.

PASH-JIT's expansion routine implements most parameter formats, plain tildes, and appropriate quoting. Currently, it does not cover impure expansion (*e.g.*, parameter formats that have side-effects like `${x=foo}`, which will set `x` to `foo` if `x` is unset), since impurity violates the parallelizable region requirements. It also does not implement a few expansion cases—*e.g.*, arithmetic expansions of the form `$((x + 1))`—that were not seen in the corpus of parallelizable scripts used to evaluate PASH-JIT (§7). Adding support for unimplemented forms would require engineering effort, but not a fundamental change to PASH-JIT's expansion. If the expansion encounters a term it cannot expand—because it is unimplemented or because it would be impure—the compilation process aborts and PASH-JIT runs the original fragment.

5.3 Dependency Untangling

PASH-JIT's compilation server makes it easy to detect when parallelizable regions are independent—including, for example, independent program fragments that are sequentially composed with `;` or different iterations of a `for` loop. A key insight here is the semantics of PASH-JIT's successful compilation: if the PASH-JIT compiler succeeds on a given region, that region's original script fragment must only affect its input and output streams (files). That is, successful fragment compilation means that the fragment is *pure*, reading from and writing to a well-defined set of streams without modifying any other global system state such as non-temporary streams or environment variables.

The PASH-JIT compiler thus tracks each parallelizable region in terms of its read and write sets, which suffice to detect read-write and write-write dependencies between fragments. If two fragments (a) compile successfully and (b) have no dependencies, they can be executed in parallel. This optimization improves performance not only because of the parallel speedup, but also because it overlaps (*i.e.*, pipelines) compilation and execution, reducing net run-time overhead.

To discover independent fragments, the compilation server (Fig. 3) and JIT engine (Fig. 4) are extended to communicate about successfully compiled fragments. Coordinating using `exit` requests, the compilation server maintains a map of running fragments. When it receives a compilation request that succeeds, the server waits for all prior fragments with dependencies to finish executing; only then does it send the compiled fragment to the JIT engine for execution in the background. While the compiled fragment executes in the background, the JIT engine can exit PASH-JIT mode, and execution proceeds with the rest of the input script. When

```
# State contains a map from ids to
#   inputs and outputs.
while True:
  req = receive_request()
  if reached_script_end(req):
    wait_all()
    exit()
  else if is_exit_request(req):
    state.remove_id(req.id)
  else if is_compile_request(req):
    compile_res = compile(region)
    if not compile_res.success:
      wait_all()
      respond(compile_res)
    else if compile_res.success:
      # Wait until all ids with dependencies
      #   finish executing.
      wait_for_dependencies(compile_res.inputs,
                            compile_res.outputs)
      request_id = fresh_id()
      state.add_request(request_id, compile_res)
      respond(compile_res, request_id)
```

Fig. 3: Compilation server algorithm (pseudocode) extended for dependency untangling (*Cf.*§5.3).

```
# Blocking query
res = query_server(compile_request(region))

if res.success:
  # Run the compiled code in parallel
  fork({
    run(compiled)
    send_exit(res.id)
  })
else:
  run(original)
...
```

Fig. 4: JIT engine algorithm (pseudocode) extended for dependency untangling (*Cf.*§5.3).

execution reaches another fragment and the JIT engine returns to PASH-JIT mode, the JIT engine will block again until the compilation server responds. Even if the compilation server encounters a fragment that fails to compile, the server blocks on dependencies: the uncompilable fragment might have arbitrary side-effects.

To ensure that our algorithm is correct, we modeled it using the SPIN Model Checker [29] and we verified (i) that it does not lead to deadlocks, (ii) that no failed compiled region is running simultaneously with any other region, and (iii) that two regions with dependencies never run at the same time.

5.4 Profile-driven Compiler Configuration

The long-lived PASH-JIT compilation server can additionally use dynamic information to improve compilation. One particularly effective optimization is to dynamically determine maximum parallelism degree. As scripts might already fea-

ture task-based parallelism, spawning too many data-parallel processes can overload the system—leading to higher overheads that cut into the speedup or even result in a slowdown. These slowdowns tend to occur when there are many computationally light commands with small inputs, *i.e.*, when the overhead of managing parallelism is higher relative to the actual work to be done. The PASH-JIT compiler can reflect on prior fragments to determine an appropriate parallelism degree.

The compilation server is often queried to compile the *same* fragment many times—*e.g.*, in each iteration of a loop. At run-time, the compiler collects and maintains execution-time information. As program fragments are recompiled, PASH-JIT tries progressively narrower parallelization degrees in an attempt to minimize overall execution time.

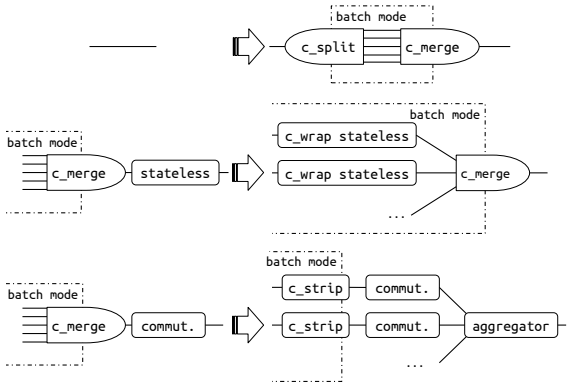

Fig. 5: Overview of commutativity-aware transformations.

6 Commutativity Awareness

Commutative commands can improve parallelization gains by allowing PASH-JIT to split and process data-parallel partial inputs in small and order-independent batches. Splitting input into many small batches improves expected CPU utilization and allows for additional pipeline parallelism. CPU utilization is improved due to an increase in partial input batches: the more work items, the more uniform the work each parallel copy does. Additional pipeline parallelism is achieved by overlapping input splitting and processing: rather than reading the entire input before deciding how to split it into batches, input can be split via small incremental steps that are immediately handed off to data-parallel commands for processing.

The PASH-JIT compiler uses these insights to produce more efficient parallel implementations of scripts that contain commutative commands. It introduces a few auxiliary nodes in its intermediate representation (IR) that orchestrate parallel execution for stateless and commutative commands, and compiler transformations that insert these nodes in a dataflow graph. It also provides efficient primitives implementing these nodes when instantiating in the parallel target script.

6.1 Compilation: Dataflow Model

The PASH-JIT compiler operates on a dataflow IR that builds on PASH-AOT, where commands correspond to nodes and communication channels correspond to edges between nodes. To enable commutativity-aware transformations, PASH-JIT extends PASH-AOT's annotation framework (§5.1) to indicate whether a command invocation is commutative (in addition to its parallelizability characteristics).

Command nodes: PASH-JIT introduces the following four dataflow nodes, which correspond to PASH-JIT-provided binary commands available in the `PATH`: `c_split`, `c_wrap`, `c_strip`, and `c_merge`. The `c_split` node takes a single input stream and N output streams. It splits its input into small batches, prepends a header on each batch identifying its sequence number, and then forwards it to one of the N outputs depending on a load-balancing strategy. Currently, PASH-JIT implements a round-robin strategy. The `c_merge` node performs the inverse operation: it merges N input streams into one and removes any headers. The `c_wrap` command is used to wrap stateless commands. It removes the header, forwards the input to the command, and then adds the header back to the command output. Finally, `c_strip` is a single-input-single-output header-removal node that often precedes commutative commands.

Transformations: To expose commutativity-aware parallelism, PASH-JIT transforms the dataflow graph; see §2 for an example. The transformations are visualized in Figure 5. The first transformation introduces a pair of `c_split` and `c_merge` before any commutative (*e.g.*, `sort`) or stateless (*e.g.*, `grep`) command. Another transformation then tries to eliminate unnecessary splits and merges, delaying `c_merge` as late as possible (*i.e.*, enclosing the biggest possible part of the graph). If a stateless command follows a `c_merge`, the command is wrapped with `c_wrap` and the `c_merge` is commuted after it. If a commutative command follows a `c_merge`, the command is parallelized and `c_merge` is transformed to a set of `c_strip` commands. Finally, if a `c_split` follows a `c_merge`, then the two are fused together to the identity function, connecting the inputs of `c_merge` with the outputs of `c_split`.

An important execution invariant is that `c_split` and `c_merge` (or `c_strip`) satisfy the requirements of well-formed parentheses, *i.e.*, a `c_split` must always be followed by a `c_merge` or a set of `c_strip` commands. PASH-JIT's dataflow graphs are essentially bimodal, since subgraphs that are between a `c_split` and a `c_merge` will execute with batches, requiring all commands in them to be wrapped with `c_wrap`, while the rest of the dataflow graph executes like the original.

Tab. 1: Benchmark summary. Summary of all the benchmarks used to evaluate PASH-JIT and their characteristics.

	Benchmark Set	Short Label	Sections	Scripts	LOC	Input	Source
1	POSIX Test Suite	PosixTests	§7.1	7	29k	—	[26]
2	Common & Classic One-liners	Classics	§7.1–7.3	10	123	14G	[6,7,33,41,59]
3	Bell Labs Unix50	Unix50	§7.1–7.3	36	142	21G	[8,37]
4	COVID-19 Transit Analytics	COVID-mts	§7.1–7.3	4	79	3.4G	[62]
5	Natural-Language Processing	NLP	§7.1–7.3	21	306	1060 books	[15]
6	NOAA Weather Analysis	AvgTemp	§7.1–7.3	1	31	36.2G	[65]
7	Wikipedia Web Indexing	WebIndex	§7.1–7.3	1	116	1000 files	[63]
8	Video/Audio Processing	MediaConv	§7.1–7.3	2	35	2.2+2.2G	[52,56]
9	Program Inference	ProgInf	§7.1–7.3	1	18	2330 libraries	[64]
10	Traffic/PCAP Log Analysis	LogAnalysis	§7.1–7.3	2	63	10–20G	[52,56]
11	Genomics Computation	Genomics	§7.1–7.3	1	34	100G	[11,51]
12	AUR Package Compilation	AurPkg	§7.1–7.3	1	27	150 packages	[13]
13	Encryption/Compression	FileEnc	§7.1–7.3	2	44	20G	[43]
14	Microbenchmarks	MicroBench	§7.3	1	6	—	custom (ours)

6.2 Runtime: Commutativity Implementation

The runtime splits the source in small batches (that contain complete lines) in a round-robin fashion.

Protocol: To reconstruct the order of different outputs while merging, PASH-JIT needs to keep track of ordering as input batches are sent to different command copies for processing and, more generally, as input-output batches flow throughout the parallelized script. To achieve this, PASH-JIT wraps all input batches with a header that contains the three following fields: `block_id`, for ordering blocks; `block_size`, the size of the block in bytes; and `is_last`, a boolean value true only for the last block with a given `block_id`.

Utilization and deadlocks: PASH-JIT must avoid deadlocks during write operations between the wrapper commands and the commands they wrap—*i.e.*, the two should never be blocked trying to write at the same time. Additionally, the wrappers must maximize utilization of the command they wrap, *i.e.*, they should never wait on input unnecessarily. To avoid deadlocks, PASH-JIT wrappers use non-blocking read and write; and to increase utilization and reduce waiting time, they write in small chunks of 32KB.

Handling inputs with long lines: An input may contain lines that are longer than the `c_split` block size. Such an event leads to non-uniform block sizes and high memory consumption, because each block must be read and sized completely before splitting and adding to the header. PASH-JIT addresses this issue by introducing the `is_last` header field in `c_split`: if a block exceeds the specified size (due to containing large lines) the block is split into multiple blocks; all blocks share the same `block_id` but only the last sets `is_last` to true. Sub-blocks with the same `block_id` are sent downstream in-order, and therefore downstream commands can use the `is_last` information to correctly reconstruct the output and know when a block ends. Block splitting reduces memory requirements and improves performance, as it allows for higher utilization regardless of the frequency of newlines. And blocks maintain a constant size throughout the flow, despite the presence of commands with high output-to-input ratio such as `curl`.

Handling small inputs: Inputs that are smaller than `c_split`'s block size lead to a single block and thus sequential execution. PASH-JIT's `c_split` addresses this issue by first attempting to read an input size s equal to `downstream_count * block_size` bytes before forwarding any blocks. If the total input is larger than s, this buffering ensures that all parallel instances will get at least one block; if the total input is smaller than s, then the input read is re-split into blocks fairly and forwarded downstream. The size s is configurable and defaults to 1MB, which we empirically determined avoids both high overhead and low utilization.

7 Evaluation

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

To evaluate PASH-JIT, we use three experiments on benchmarks (Tab. 1). The first experiment focuses on PASH-JIT's compatibility and uses the entire POSIX test suite as well as additional scripts (§7.1). The second experiment focuses on the performance gains achieved by PASH-JIT's parallelization, evaluated using a variety of benchmarks and workloads (§7.2). The last experiment zooms into PASH-JIT-internal overheads and associated optimizations (§7.3).

Hardware & software setup: PASH-JIT was run on 64 physical × 2.1GHz Intel Xeon E5-2683 cores with 512GB of RAM, Debian 4.9.144-3.1, GNU Coreutils 8.30-3, GNU Bash 4.4.20(1), and Python 3.7.3. There is no special configuration in hardware or software. We use Dash v.0.5.8-2.10 and Ksh v.93u+ 2012-08-01. All scripts were executed completely unmodified, using environment variables, loops, and other shell

Tab. 2: Correctness results. Running the POSIX test suite on Bash and PASH-JIT. Tests are grouped in rows by theme. Columns contain the group name, total tests, non-applicable tests, and passing tests for PASH-JIT and Bash.

	Test Suite	Tests	Untested	PASH-JIT	Bash
1	Parsing	38	5	33/33	33/33
2	Expansion	83	8	71/75	71/75
3	Errors	38	3	26/35	27/35
4	Commands and redirects	99	2	96/97	96/97
5	Subshells and pipelines	56	7	46/49	46/49
6	Builtins	113	40	60/73	61/73
7	Special cases	67	21	42/46	42/46

constructs. To minimize statistical non-determinism, we host our experimental infrastructure on our own premises, avoid sharing with other research groups, and repeat the experiments several times noting imperceptible variance.

7.1 Correctness

We evaluate the correctness of PASH-JIT across all benchmarks from Tab. 1 by checking that PASH-JIT's stdout and exit status are equivalent to the ones produced from Bash. The output is over 650 million lines (18GB), taken from 82 scripts, in all of which PASH-JIT's output and exit status are correct. To increase our confidence on correctness, we use the POSIX shell test suite with both Bash and PASH-JIT.

Benchmarks: The POSIX test suite is a thorough evaluation of shell behavior, comprising 1007 'assertions' evaluated using 494 distinct, assertion-numbered test cases over 29k LOC of shell scripts (plus library support). We exclude (a) 78 test cases because they test the platform (*e.g.*, locales) rather than the shell, and (b) 8 cases because they test interactivity, which is out of scope for PASH-JIT (§1). These leave a total of 408 runnable test cases. The test cases use a mix of shell language features (*e.g.*, redirection, pipes), builtin commands (*e.g.*, `set`, `echo`), and standard UNIX utilities (*e.g.*, `printf`, `grep`). The POSIX suite tests many corner cases of shell behavior—*e.g.*, that aliases ending in space continue alias expansion (Assertion no. 284), that pipelines take precedence over redirections in their constituent commands (no. 454), or that **`return`** in a `trap` action restores the previous command's exit status (no. 651)—totaling several thousand behaviors. The exact number of 'tests' is hard to quantify: some test cases check a single behavior (*e.g.*, expanding an unset variable under `set -u`); others check hundreds (*e.g.*, many different characters escape properly; many different arithmetic expressions evaluate correctly).

Results: PASH-JIT overwhelmingly agrees with Bash (Tab. 2). PASH-JIT passes 374 and fails 34 POSIX tests, while Bash passes 376 and fails 32 POSIX tests. PASH-JIT diverges from Bash on the test cases for a mere 2 tests (no. 430 and 691) where Bash passes but PASH-JIT fails. These two failures concern the ranges of non-zero exit status and are in fact due to an unusual inconsistency in Bash itself (see "Discussion", below).

When running the test suite, PASH-JIT invokes the compiler a total of 3304 times, each for a different potentially optimizable fragment; 713 (20%) of those invocations successfully compile, *i.e.*, PASH-JIT generates and runs parallel code. Successful compilation does not necessarily translate to a speedup on individual tests, though: the POSIX suite tends to test with small scripts, so the compiled fragments contain very little computation—not much for PASH-JIT to optimize.

Discussion: PASH-JIT diverges from Bash in two cases only in the exit status returned. Both PASH-JIT and Bash exit with an error: Bash returns 1, and PASH-JIT returns 127. For the two failing cases, POSIX mandates (since 2008) that the exit status be between 1–125, making PASH-JIT's behavior incorrect. Why does PASH-JIT produce a different status?

Bash is inconsistent when called with the `-c` flag. Contrary to most other shells (*i.e.*, dash, ksh, mksh, posh, sash, Smoosh, yash, zsh), Bash is the only shell that, when failing during `-c` invocations, exits with 127—*i.e.*, outside the POSIX-mandated range. When PASH-JIT invokes the underlying Bash interpreter using `-c` in order to set `$0`, it receives and propagates an exit status that does not comply with POSIX. The rest of the Bash failing tests are caused by various subtleties; it is not clear which failures are 'true bugs' and which are considered desirable divergences from the spec. Greenberg and Blatt [24] discuss how implementations diverge from the POSIX spec. PASH-JIT mirrors the behavior of Bash in all those cases.

To put the number of diverging tests of PASH-JIT and Bash into perspective, we note that other production shells fail in significantly greater numbers: dash passes 3 tests that Bash fails and fails 20 that Bash passes; ksh passes 2 tests that Bash fails and fails 20 that Bash passes; and zsh cannot run the test suite at all. These results combined show that, in practice, PASH-JIT is virtually indistinguishable from its underlying shell interpreter on POSIX features.

7.2 Performance

We evaluate PASH-JIT's performance on 12 sets of real-world shell scripts taken from a variety of sources (Tab. 1, rows 2–13), totalling 82 shell scripts and 1015 LOC.

Benchmarks: Classics and Unix50 contain classic and recent (c. 2019) scripts making heavy use of UNIX and Linux built-in commands. COVID-mts contains four scripts used to analyze real telemetry data from mass-transit schedules during a large metropolitan area's COVID-19 response. NLP contains several scripts from UNIX-for-poets, a tutorial for developing programs for natural-language processing out of UNIX and Linux utilities. AvgTemp contains a large script downloading and processing multi-year temperature data across the US. WebIndex is a large multi-stage script for web crawling and

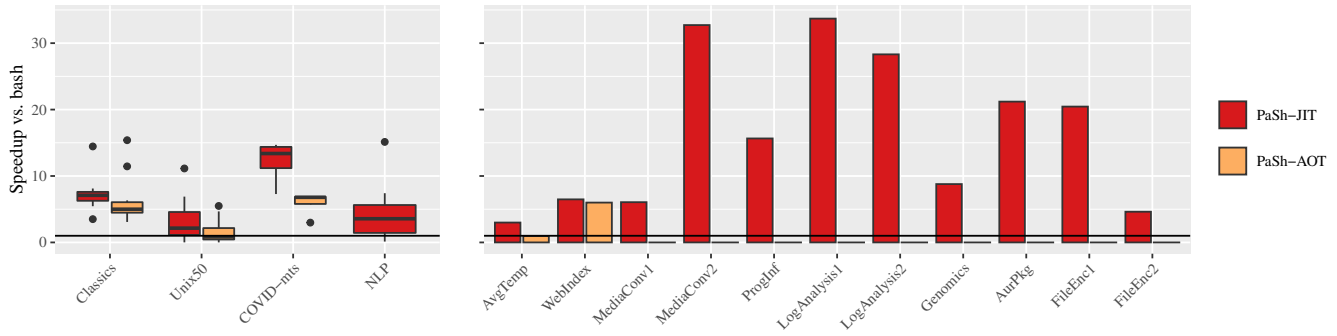

Fig. 6: PASH-JIT Performance. PASH-JIT speedup (*vs.* PASH-AOT whenever possible) over Bash for Tab. 1 rows 2–5 (left, box) and 6–13 (right, bar) (*Cf.*§7.2).

indexing, using a variety of third-party and built-in utilities. MediaConv contains two scripts that process, transform, and compress video and audio files. ProgInf contains a script that downloads JavaScript packages from the npm registry and applies a security-oriented static program analysis. LogAnalysis contains two scripts that apply typical system-administration and network-traffic analyses over log files. Genomics contains a script that processes next-generation sequencing data for the purposes of diagnostic virology. AurPkg contains the main script that compiles, builds, and packages software for the AUR Linux distribution. Finally, FileEnc contains long aliases that encrypt and compress files.

Results: PASH-JIT surpasses PASH-AOT's speedups (*vs.* Bash) on existing benchmarks and extends speedups to new ones (Fig. 6). Box-plots show results for multi-benchmark suites (Tab. 1, rows 2–5) and bars for individual scripts (Tab. 1, rows 5–13). PASH-JIT can run several more scripts than PASH-AOT (for which performance bars are set to 0). Across all benchmarks, PASH-JIT achieves an average speedup of 5.86× (*vs.* 2.9× for PASH-AOT) and a maximum speedup of 33.7× (*vs.* 15.38× for PASH-AOT).

A few scripts exhibit slowdowns when compiler startup, runtime, and parallelization overheads (splitting, merging) start dominating. PASH-JIT decelerates 14 scripts; PASH-AOT decelerates 20 scripts—and cannot run 30 additional scripts that PASH-JIT parallelizes. The scripts that PASH-JIT decelerates either have short sequential running times (8ms–10s) or have very short-running fragments in tight loops (*e.g.*, 1K iterations, 14ms per iteration). For example, PASH-JIT decelerates Unix50's `20.sh` (Bash: 8ms; PASH-JIT: 1.3s) and NLP's `no-vowel.sh` (Bash: 14s; PASH-JIT: 0.24×), on which PASH-AOT cannot operate.

Discussion: PASH-JIT is faster than PASH-AOT on all suites 2–5 (w.r.t. average) and on all individual benchmarks 5–13, often by a significant margin (3.1×).

PASH-JIT speeds up many scripts PASH-AOT cannot, as PASH-AOT's ahead-of-time parallelization cannot reason about the shell's dynamic features. PASH-AOT offers no speedup on the NLP suite, nor on any individual scripts except for AvgTemp and WebIndex.

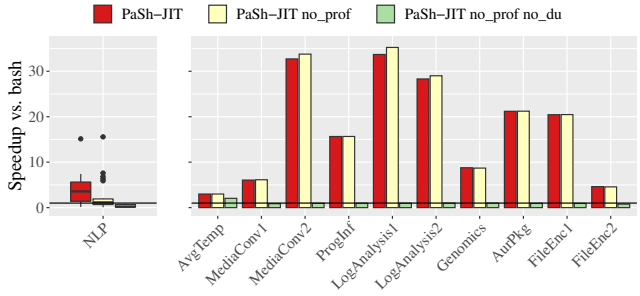

Fig. 7: PASH-JIT Dynamic Optimizations. PASH-JIT speedup over Bash when toggling profile-driven compiler configuration and dependency untangling for Tab. 1 row 5 (left, box) and 6, 8–13 (right, bar) (*Cf.*§7.3).

Compared to Bash, PASH-JIT is faster (or at least as good) in all cases, except when the given script is very short-running (*e.g.*, `unix50-20.sh`), or with a tight loop with a very short-running body (*e.g.*, `nlp-no-vowel.sh`).

7.3 Further Microbenchmarks

This section zooms into the benefits of PASH-JIT's optimizations targeting dependency untangling, profile-driven compiler configuration, commutativity analysis, and JIT engine overheads.

Dynamic optimizations: To better understand the benefits of dependency untangling and profile-driven compiler configuration (CC), we use benchmarks that have sequences of statements—*e.g.*, some form of sequential composition or `for`-loops: rows 5, 6, 8–13 from Tab. 1. One-line scripts such as Unix50 and WebIndex feature single pipelines and thus cannot benefit from any inter-region optimizations.

Across all scripts and compared to Bash, PASH-JIT achieves a speedup of 8.17×. PASH-JIT without profile-driven CC achieves 7.58×, and additionally without dependency untangling 0.55× (Fig. 7). The 0.55× slowdown is due to limited intra-region parallelization in these benchmarks.

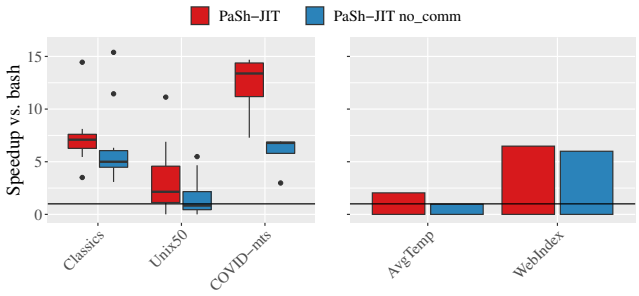

Fig. 8: PASH-JIT Commutativity Awareness. PASH-JIT speedup over Bash when toggling commutativity awareness for Tab. 1 rows 2–4 (left, box) and 6, 7 (right, bar) (*Cf.*§7.3).

Profile-driven CC may slightly reduce speedup in highly parallelizable scripts, because it explores lower parallelization degrees.

Commutativity awareness: To evaluate the benefits of commutativity-related optimizations, we focus on all scripts with intra-region parallelization potential: Classics, Unix50, COVID-mts, AvgTemp, and WebIndex; the performance of the rest is affected negligibly by changes to single-region transformations. We disable all dynamic optimizations to isolate the benefits of commutativity, and compare with the sequential Bash baseline.

Commutativity-aware PASH-JIT achieves an average speedup of 4.52× and a maximum of 14.68× (Fig. 8). Without commutativity-related optimizations, PASH-JIT achieves an average speedup of 3.72× and a maximum of 15.38×. Commutativity improves the average case but not cases that already see high speedups, as these (1) have negligible overheads coming from input reading—most overheads come due to line processing—and (2) commutativity extensions add some overhead due to the `c_wrap` primitive.

JIT engine overhead: To evaluate the benefits of PASH-JIT's runtime optimizations, we design a worst-case parallelization benchmark: a script that contains a **`for`** loop that performs 100 iterations of `echo hi`. A tight loop with a minimal-overhead body emphasizes the JIT engine overheads by allowing no parallelization gains. The table on the right shows the run-time performance of four PASH-JIT configurations compared to Bash: (1) PASH-JIT without custom expansion, compilation server, and dynamic optimizations, (2) PASH-JIT without compilation server, and dynamic optimizations, (3) PASH-JIT without the dynamic optimizations, and (4) the complete PASH-JIT. PASH-JIT's runtime optimizations (custom expansion, compilation server, and dependence untangling) improve performance by 12× (over the `-esd` configuration without them). As `echo hi` writes to stdout, dependence untangling does not manage to run it in parallel, and thus its benefit is only due to pipelining. Even then, PASH-JIT's JIT engine overhead is not negligible (about 47ms per JIT invocation), as it needs to save the state and invoke the compiler for every iteration of the loop body.

Config.	Time (s)
Bash	0.008
PASH-JIT -esd	59.334
PASH-JIT -sd	15.376
PASH-JIT -d	6.124
PASH-JIT	4.708

8 Related Work

Parallel shell scripting: Recent work addresses significant challenges related to automatic shell script parallelization. POSH [52] and PASH-AOT [63] are mostly-automated ahead-of-time shell-script parallelization systems; as described earlier, these systems focus on fully expanded shell pipelines that do not make use of dynamic features. Recent work explored an order-aware dataflow model as a foundation for modeling the transformations these systems perform and proving them correct [28]. To enable divide-and-conquer parallelism, KumQuat [55] proposes a program-synthesis technique for generating aggregators for black-box commands.

PASH-JIT builds on all this prior work, addressing fundamental limitations in static, ahead-of-time parallelization: AOT approaches apply to a very small subset of real shell scripts. By opting for just-in-time parallelization, PASH-JIT achieves parallel script behavior that is practically indistinguishable from the sequential execution—and ample opportunities for additional acceleration.

Other work on shell script parallelization either requires manual effort or is applicable to a smaller subset of scripts than our work. Such work includes: utilities like `qsub` [19], SLURM [66], and `parallel` [58]; shells with non-linear pipe topologies [17, 40, 56]; and using the shell itself as a DSL for concurrency [22].

Unix-related parallelization: There has been a significant body of work on parallel (and distributed) UNIX and UNIX-like environments [4, 44, 47], including shell-oriented efforts such as Plan9's `rc` [49]. Contrary to PASH-JIT, these systems did not (aim to) offer full compatibility with the sequential UNIX shell. They also focused on systems-level and program-runtime support, rather than automated program analyses and transformations.

Just-in-time compilation: Just-in-time compilation has been studied for long time [3], mainly in two contexts: (1) as a compilation technique for interpreted languages such as JavaScript [20], where critical type information is unavailable prior to execution; and (2) as a performance optimization over ahead-of-time compilation, allowing for specialization [30, 60], loop unrolling and function inlining [9, 50], and other profile-guided optimizations [34, 46]. PASH-JIT draws inspiration from work in both contexts—resolving unavailable dynamic information at run-time and performing additional optimizations. It also leverages the optimistic compilation technique employed commonly by just-in-time compilers: when it fails to compile (parallelize), it simply runs the original fragment using the shell interpreter as a fallback option.

PASH-JIT differs from most JITs, dealing with different challenges: it operates at a higher level of abstraction, in a unique programming environment with no single unified runtime.

PASH-JIT also draws inspiration from staged compilation [14] and partial evaluation [32]. These techniques perform some compilation ahead-of-time, waiting for the runtime to specialize and further optimize when there is more information about the environment of the target program and how it is used.

Parallelization in other contexts: More general parallelization support can be grouped into two categories: languages and tools. One approach to parallelization support is to use tools that requires writing in a new higher-level programming language [18, 21, 36] or a dataflow-based model embedded in an existing language [5, 12, 16, 45, 57, 67]. These tools usually offer automation, but require re-expressing existing computations in domain-specific programming models; PASH-JIT operates on completely unmodified POSIX shell scripts that use unusual features and obscure corner cases.

Another approach to parallelization support uses tools that provide automatic parallelization for standard sequential code, requiring no program modifications but often posing limitations with respect to the granularity of the parallelism that they can extract. The general approach started with explicit `DOALL` and `DOACROSS` annotations [10, 38], continuing with analysis-based compilers [27, 48, 54], and more recent work using profiling-guided speculation [1, 31, 35, 42, 61]. PASH-JIT draws inspiration from this line of work: it does not require manual modification to user code, and it leverages run-time information to optimize and parallelize user scripts. Existing tools work on imperative code with memory accesses, but PASH-JIT works at a higher level of abstraction: commands that affect the file system and the broader executing environment.

Shell correctness and POSIX compliance: Smoosh [24] offers a formalized, executable reference semantics for the POSIX shell, aiming to address subtleties in the standard [2]. PASH-JIT leverages Smoosh to identify and resolve issues in its JIT engine (§4) and to guide its early expansion routine (§5.2). It also builds on Smoosh's analysis to leverage the POSIX test suite for characterizing shell behavior.

PASH-JIT reimplements Smoosh's `libdash` [23], which presents dash's parser as a library (§3.3). We chose `libdash` over Morbig [53] because (1) `libdash` reuses dash's production-grade parser, and (2) `libdash` supports line-oriented input, but Morbig is strictly ahead-of-time.

Resurgence of shell research: Recent shell research [24, 25, 39, 43, 52, 55, 56, 63] highlights renewed interest in shell scripting both as a vehicle for impactful research and as a target worthy of scientific attention. We see PASH-JIT as a natural continuation of the insights and research behind recent shell-script parallelization systems [25, 28, 52, 63], allowing other researchers to leverage PASH-JIT's POSIX-compliant high-performance just-in-time compilation in their future work.

9 Discussion & Conclusion

The shell provides a dynamic programming language with complex evaluation-and-expansion semantics and ubiquitous side-effects—effects that interact with the entire UNIX system similar to how a conventional programming language interacts with its runtime environment. The benefits of just-in-time compilation for dynamic languages are clear, and PASH-JIT is the first JIT compiler that targets challenges unique in the UNIX shell ecosystem. PASH-JIT forms a promising drop-in shebang replacement: its POSIX compliance rivals shells in widespread use; and its performance benefits go well beyond the state of the art.

Interactivity: PASH-JIT's design goals (§1) do not include interactivity; an interactive shell switches between consuming its input (shell commands) and redirecting it to its executing commands—challenging for PASH-JIT's loose coupling. Furthermore, avoiding shell modifications leads to additional runtime overhead (since the state of the shell has to be reflected upon and is not accessible with a single dereference). Adding robust support for interactivity and improving runtime overhead would likely require a more intrusive design, *e.g.*, altering Bash's source and interposing directly. However, such a design would make PASH-JIT Bash-specific, requiring users to install a new shell, and would significantly complicate the engineering and maintenance effort involved.

Expansion: Some of PASH-JIT's expansion behaves in a way not exactly as specified by POSIX, although we conjecture (and our evaluation confirms, §7) it is safe. For example, pipelines are supposed to expand each component in its own subshell (though the last component may run in the outer shell, depending on a shell's implementation choices). PASH-JIT's expansion operates on each component of the pipeline early; each component uses its own copy of the shell environment, to simulate the subshells. We haven not proved these early expansions sound, and it would be interesting future work to pursue that, *e.g.*, by using Smoosh's semantics.

Command annotations: PASH-JIT's performance benefits depend on the existence of command parallelizability annotations. The annotations used by PASH-JIT depend on the PASH-AOT annotation library [63], which includes many commands in the POSIX and GNU Coreutils sets. Apart from commands in these sets, a script may contain other commands—for which PASH-JIT will lack annotations and thus will not attempt to parallelize to maintain soundness (§1). To better harness PASH-JIT parallelization in their scripts, users can: (1) opt for more restricted, rather than more general, utilities with more constrained and thus parallelizable behaviors (*e.g.*, use `cut` rather than `awk` when projecting columns, as `awk` programs are not parallelizable in general); or (2) add

their own annotations for custom commands to inform PASH-JIT on how to parallelize them.

Enabling other analyses: Even though PASH-JIT is mainly focused on parallelization, its just-in-time structure is not limited to it. By slightly modifying the preprocessor and by replacing the compilation server logic, PASH-JIT can be made to perform different types of analyses and transformations, while maintaining its benefits—compliance with the underlying shell, loose coupling, and low runtime overheads. This enables exciting avenues of future tooling and support for the shell, like incremental execution, automatic distribution, and safety monitoring.

Conclusion: Fundamentally, PASH-JIT shows that it is possible to build a just-in-time shell-script parallelization infrastructure that is substantially faster and more applicable than prior work, is loosely coupled, and addresses critical challenges associated with the shell ecosystem's polyglot runtime environment. But also, PASH-JIT is not a toy: it enables other researchers to use a production-grade POSIX-compliant shell compiler for impactful future work.

Acknowledgements: We would like to thank Achilles Benetopoulos, Ben Karel, Caleb Stanford, the OSDI 2022 reviewers, and our shepherd, Robert Soulé, for discussions and feedback that helped improve the presentation of the paper; the participants of UCSC's LSD seminar for early discussions on dependency untangling; and the open-source developers who have contributed to PASH. This material is based upon work supported by DARPA contract no. HR00112020013 and no. HR001120C0191, and NSF awards CCF 1763514 and 2008096.

References

[1] Sotiris Apostolakis, Ziyang Xu, Greg Chan, Simone Campanoni, and David I August. Perspective: A sensible approach to speculative automatic parallelization. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 351–367, 2020.

[2] The Austin Group. POSIX.1 2017: The Open Group Base Specifications Issue 7 (IEEE Std 1003.1-2008), 2018.

[3] John Aycock. A brief history of just-in-time. *ACM Computing Surveys (CSUR)*, 35(2):97–113, 2003.

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

[5] Jonathan C Beard, Peng Li, and Roger D Chamberlain. Raftlib: a C++ template library for high performance stream parallel processing. *The International Journal of High Performance Computing Applications*, 31(5):391–404, 2017.

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

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

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

[9] Carl Friedrich Bolz. *Meta-tracing just-in-time compilation for RPython*. PhD thesis, Universitäts-und Landesbibliothek der Heinrich-Heine-Universität Düsseldorf, 2014.

[10] Michael Burke and Ron Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '86, pages 162–175, New York, NY, USA, 1986. ACM.

[11] Enrico Cappellini, Frido Welker, Luca Pandolfi, Jazmín Ramos-Madrigal, Diana Samodova, Patrick L Rüther, Anna K Fotakis, David Lyon, J Víctor Moreno-Mayar, Maia Bukhsianidze, et al. Early pleistocene enamel proteome from dmanisi resolves stephanorhinus phylogeny. *Nature*, 574(7776):103–107, 2019.

[12] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38:28–38, 2015.

[13] Armando Cerna. Pacaur building script.

[14] Craig Chambers. Staged compilation. *ACM SIGPLAN Notices*, 37(3):1–8, 2002.

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

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

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

[18] Matteo Frigo, Charles E Leiserson, and Keith H Randall. The implementation of the cilk-5 multithreaded language. *ACM Sigplan Notices*, 33(5):212–223, 1998.

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

[20] Google. V8 javascript engine. https://developers.google.com/v8/.

[21] Michael I Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S Meli, Andrew A Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, et al. A stream compiler for communication-exposed architectures. In *ACM SIGOPS Operating Systems Review*, volume 36, pages 291–303. ACM, 2002.

[22] Michael Greenberg. The posix shell is an interactive dsl for concurrency. https://cs.pomona.edu/~michael/papers/dsldi2018.pdf, 2018.

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

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

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

[26] The Open Group. Posix. https://pubs.opengroup.org/onlinepubs/9699919799/, 2018. [Online; accessed November 22, 2019].

[27] Mary W Hall, Jennifer M Anderson, Saman P. Amarasinghe, Brian R Murphy, Shih-Wei Liao, Edouard Bugnion, and Monica S Lam. Maximizing multiprocessor performance with the suif compiler. *Computer*, 29(12):84–89, 1996.

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

[29] Gerard J. Holzmann. The model checker spin. *IEEE Transactions on software engineering*, 23(5):279–295, 1997.

[30] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Hideaki Komatsu, and Toshio Nakatani. A study of devirtualization techniques for a java just-in-time compiler. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 294–310, 2000.

[31] Nick P Johnson, Hanjun Kim, Prakash Prabhu, Ayal Zaks, and David I August. Speculative separation for privatization and reductions. *ACM SIGPLAN Notices*, 47(6):359–370, 2012.

[32] Neil D Jones. An introduction to partial evaluation. *ACM Computing Surveys (CSUR)*, 28(3):480–503, 1996.

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

[34] Konstantinos Kallas and Konstantinos Sagonas. Hiperjit: A profile-driven just-in-time compiler for erlang. In *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages*, pages 25–36, 2018.

[35] Hanjun Kim, Nick P Johnson, Jae W Lee, Scott A Mahlke, and David I August. Automatic speculative doall for clusters. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 94–103, 2012.

[36] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L Paul Chew. Optimistic parallelism requires abstractions. *ACM SIGPLAN Notices*, 42(6):211–222, 2007.

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

[38] Amy W. Lim and Monica S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 201–214, New York, NY, USA, 1997. ACM.

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

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

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

[42] Mojtaba Mehrara, Jeff Hao, Po-Chun Hsu, and Scott Mahlke. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. *ACM Sigplan Notices*, 44(6):166–176, 2009.

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

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

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

[46] Guilherme Ottoni. Hhvm jit: A profile-guided, region-based compiler for php and hack. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 151–165, 2018.

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

[48] David A Padua, Rudolf Eigenmann, Jay Hoeflinger, Paul Petersen, Peng Tu, Stephen Weatherford, and Keith Faigin. Polaris: A new-generation parallelizing compiler for mpps. In *In CSRD Rept. No. 1306. Univ. of Illinois at Urbana-Champaign*, 1993.

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

[50] Ian Piumarta and Fabio Riccardi. Optimizing direct threaded code by selective inlining. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 291–300, 1998.

[51] Jon Puritz. Bio594: Using genomic techniques to examine the evolution of populations, 2019. Accessed: 2020-10-05.

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

[53] Yann Régis-Gianas, Nicolas Jeannerod, and Ralf Treinen. Morbig: A Static Parser for POSIX Shell. In *Software Language Engineering (SLE)*, Boston, United States, November 2018.

[54] Martin C Rinard and Pedro C Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(6):942–991, 1997.

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

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

[57] Justin Talbot, Richard M. Yoo, and Christos Kozyrakis. Phoenix++: Modular mapreduce for shared-memory systems. In *Proceedings of the Second International Workshop on MapReduce and Its Applications*, MapReduce '11, page 9–16, New York, NY, USA, 2011. Association for Computing Machinery.

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

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

[60] Scott Thibault, Charles Consel, Julia L Lawall, Renaud Marlet, and Gilles Muller. Static and dynamic program compilation by interpreter specialization. *Higher-Order and Symbolic Computation*, 13(3):161–178, 2000.

[61] Chen Tian, Min Feng, and Rajiv Gupta. Supporting speculative parallelization in the presence of dynamic data structures. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 62–73, 2010.

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

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

[64] Nikos Vasilakis, Cristian-Alexandru Staicu, Grigoris Ntousakis, Konstantinos Kallas, Ben Karel, André DeHon, and Michael Pradel. Preventing dynamic library compromise on node.js via RWX-based privilege reduction. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 1821–1838, New York, NY, USA, 2021. Association for Computing Machinery.

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

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

[67] 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 *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.