Executing Shell Scripts in the Wrong Order, Correctly

Georgios Liargkovas
gliargovas@aueb.gr
Brown University & AUEB

Konstantinos Kallas
kallas@seas.upenn.edu
University of Pennsylvania

Michael Greenberg
michael@greenberg.science
Stevens Instisute of Technology

Nikos Vasilakis
nikos@vasilak.is
Brown University & MIT

ABSTRACT

Shell scripts are critical infrastructure for developers, administrators, and scientists; and ought to enjoy the performance benefits of the full suite of advances in compiler optimizations. But between the shell's inherent challenges and neglect from the community, shell tooling and performance lags far behind the state of the art. We propose executing scripts out-of-order to better use modern computational resources. Optimizing any part of an arbitrary shell script is very challenging: the shell language's complex, late-bound semantics makes extensive use of opaque external commands with arbitrary side effects.

We work with the grain of the shell's challenges, meeting dynamism with dynamism: we optimize at runtime, speculatively executing commands in an isolated and monitored environment to determine and contain their behavior. Our proposed approach can yield serious performance benefits (up to 3.9× for a bioinformatics script on a 16-core machine) for arbitrarily complex scripts without modifying their behavior. Contained out-of-order execution obviates the need for command specifications, operates on external commands, and yields a much more general framework for the shell. Script writers need not change a thing and observe no differences: they get improved performance with the interpretability of sequential output.

CCS CONCEPTS

• **Software and its engineering → Scripting languages**; *Compilers*; *Operating systems*.

HotOS '23, June 22–24, 2023, Providence, RI, USA

ACM ISBN 979-8-4007-0195-5/23/06.
https://doi.org/10.1145/3593856.3595891

ACM Reference Format:
Georgios Liargkovas, Konstantinos Kallas, Michael Greenberg, and Nikos Vasilakis. 2023. Executing Shell Scripts in the Wrong Order, Correctly . In *Workshop on Hot Topics in Operating Systems (HotOS '23), June 22–24, 2023, Providence, RI, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3593856.3595891

1 INTRODUCTION

Shell programming is as prevalent as ever. GitHub has steadily ranked the shell among the top ten programming languages for the past decade, and with increasing popularity [15]: in 2020 it jumped to eighth—growing faster than languages such as C and Ruby; in 2021 it ranked sixth for popularity increase—above languages with active communities, such as Python and Kotlin. These return-to-the-shell trends in industry are mirrored by a resurgence of academic research on the shell [5, 10, 11, 13, 16, 18, 22, 25–27, 31].

Despite its popularity, shell tooling has not kept up: weak linters, no debugging, and—our focus—no compilation or optimization. Even with its sub-par performance, the shell is the go-to choice for many long running tasks—*e.g.*, builds, orchestration, continuous integration, and data-processing. Poor support for the shell is hardly a surprise, though: it is a 'mere' glue language, a polyglot patchwork of external commands in one of the most dynamic, latest-bound languages in common use—optimization is quite a challenge!

We identify dynamic interposition, tracing, and containment as key ingredients for this kind of optimization support. Combined, these enable a powerful optimization: *out-of-order program execution* [1]. A program's execution order need not be determined by *syntax*, *i.e.*, the order in which blocks or instructions are written, but rather by *semantics*, *i.e.*, the true dependencies between different blocks or instructions. It is only safe to rearrange a program in ways that respect these dependencies; to be worthwhile, a rearrangement must also (1) accelerate execution, *e.g.*, by executing fragments for which input data is already available, and (2) better utilize the underlying resources available to the program. Tracing and containment yield another advantage over prior work [16, 25, 31] on the shell: we can appropriately trace, contain, and selectively merge a command's effects without

```
SAMPLES="100 101 102 103"
REF="hg19.fa"
GROUPS="1 2"
# (a) Index
bwa index "$REF"
for sm in $SAMPLES
do
  # (b) Align sample
  for gr in $GROUPS
  do
    bwa aln "$REF" "$sm.$gr.fastq" > "$sm.$gr.sai"
  done
  # (c) Combine sample pairs
  bwa sampe "$sm.1.sai" "$sm.2.sai" |
    samtools view -Shu - > "$sm.bam"
  # (d) Remove polymerase chain reaction-induced dups
  samtools rmdup "$sm.bam" "$sm.nodup.bam"
  # (e) Plot coverage histogram
  samtools mpileup "$sm.nodup.bam" |
    cut -f4 | python plot.py "$sm.coverage.pdf"
  # Delete temporary files
  rm -f "$sm.1.sai" "$sm.2.sai"
done
```

Figure 1: A bioinformatics script slightly adapted from Köster and Rahmann [17] that maps sequence reads to a reference genome.

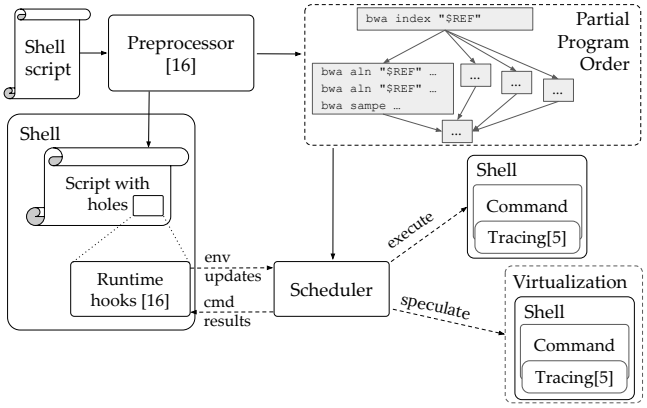


Figure 2: A high-level overview of hs, a speculative out-of-order shell-script executor. The preprocessor, runtime hooks, and tracing extend components from prior work [5, 16].

any foreknowledge of its semantics—that is, without need for command annotations!

We explain our proposal with a concrete instance of a common disorder of shell scripts: overly sequential execution.

A patient: Consider the core of a real bioinformatics script for mapping sequence reads to a reference genome (Fig. 1), a typical task in, *e.g.*, cancer genomics [21]. The script first indexes the reference genome (a); it then aligns each set of samples based on the genome (b), combines the results (c), removes duplicates (d), and plots a coverage histogram (e). Running this script for a 152MB reference genome and 3.3GB input samples takes about 30 minutes on a 3GHz 16-core machine on Cloudlab [6]. The script invokes a variety of commands: specialized genomics executables (`bwa`, `samtools`), core utilities (`cut`, `rm`), and custom scripts in interpreted languages (`python plot.py`). It combines these commands using various shell features (parameters, `for`, `>`, `|`). Several of those invocations are completely independent, and could be safely executed in any order. Every command depends on the initial indexing (a), but each outer loop iteration works on a different sample and is independent of the others. Within each sample, each group's alignment can be done independently. Sadly, the execution order of these invocations *on any modern shell interpreter* will depend entirely on the script's syntax—*i.e.*, the order in which the developer wrote the commands—leaving significant opportunities for optimization unexploited.

A treatment: We will optimize shell scripts by reordering and interleaving their commands, letting the semantic dependencies guide execution instead of syntactic ordering. We will execute independent commands out of order and in parallel, enforcing order only between commands that depend on each other (*true* dependencies).

Easier said than done! Decoupling execution order and syntax order poses daunting challenges. First, the shell is hostile to analysis, so it is hard to predict which commands will run at all, never mind their order: commands are interleaved with complex and highly dynamic control flow—*e.g.*, `if` statements, command substitutions, and parameters determined by previous commands. The shell's dynamism contrasts sharply with traditional compiler optimizations working on object code, *i.e.*, instructions sequences with occasional control flow. Second, an invoked command's semantics is coarse, complex, and unbounded—if not completely opaque. It is impossible to statically determine their interdependencies. The shell, again, presents serious challenges compared to the finite and well-defined set of instructions in object code, with generally clear dependencies and effects.

A prescription: While compiler reordering optimizations are traditionally static and pessimistic, our approach for the shell must be *dynamic* and *opportunistic*. A *dynamic* approach circumvents the intractability of ahead-of-time order extraction: our techniques learn about the execution order dependencies incrementally, building up understanding as the script runs. An *opportunistic* approach means we need not specify or even understand command behavior:

our techniques optimistically execute commands in an isolated environment—identifying and rolling back conflicting side-effects as they arise.[1]

We implement our approach in a prototype we call hs—so called because it's the shell (sh), but out-of-order. hs has three parts (Fig. 2): a script preprocessor, a scheduler, and runtime hooks. The *preprocessor* extracts commands and their partial program order, leaving holes in the preprocessed script where these commands were originally. Each of these holes is instrumented with runtime hooks that communicate with the scheduler; the partial order captures the syntactically determined execution order of different commands and is then handed off to the scheduler for execution. The *scheduler* executes commands opportunistically out-of-order, rolling back when dependencies have been violated. It uses (1) tracing to discover command dependencies and detect dependency violations, and (2) containment to shield against interference and allow rollbacks. The *runtime hooks* are invoked while executing the preprocessed script and communicate with the scheduler; their job is to hide out-of-order execution so that our reorderings are semantically transparent, *i.e.*, the script runs the same. They achieve that by propagating environment updates to the scheduler so that it has a fresh and correct view of the execution environment, potentially triggering some reexecution, and modify the shell state according to the effects of each command that was executed by the scheduler.

A relief of symptoms: On a 3GHz 16-core machine on Cloudlab [6], the ordinary syntax-guided execution order executes the script in about 30 minutes; the speculative out-of-order execution guided by the script's semantics completes in 7 minutes and 35 seconds (3.9× speedup).

2 THE TREATMENT, APPLIED

We now apply hs on the bioinformatics script (Fig. 1), sketching its design as we go (Fig. 2). hs combines preprocessing, tracing, speculation, and containment.

Preprocessing: First, the shell script is sent through a preprocessor that extracts all commands in the script. The prototype preprocessor of hs builds on and extends the just-in-time component of PaSh [16]. The preprocessor is the only syntax-driven component of our approach, parsing the shell script and replacing all command nodes in the abstract syntax tree (AST) with holes managed by the runtime hooks during execution. These command nodes are then added to the *execution set*—all of the commands that need to be executed—and sent to the scheduler. The execution set also encodes the syntactic program order, *i.e.*, the order in which commands were originally (syntactically) written. This is a partial (rather than total) order, as some commands are not syntactically ordered—*e.g.*, two different branches of an `if` statement. The partial order is an under-approximation of the control flow graph, as it doesn't model control builtins like `break`, reflection builtins like `source`, or function calls.

For the bionformatics script lines 5, 11, 14-15, 17, and 19-20 (Fig. 1) would all be replaced with holes, with each hole corresponding to a command in the execution set (Fig. 3). After preprocessing, commands in the execution set may contain all sorts of unresolved fragments—*e.g.*, unexpanded strings, unresolved variables, and unevaluated command substitutions—similar to `$REF` (line 11). These are script fragments that cannot be evaluated statically, as their values might change during execution.

Runtime hooks: The runtime hooks are invoked during the execution of the preprocessed script. When execution reaches a hole, the hooks block and wait until the scheduler has completed the execution of the command for that hole. The hooks receive the command's exit status and observe its effects on the file-system and the shell state (like variable updates or shell state reconfigurations (*e.g.*, `cd` or `set -e`). The hooks must propagate all of this update information to the scheduler, as it will affect commands downstream in the partial order. In Fig. 1's script, the hooks propagate assignments to variables such as `$REF`, `$sm`, and `$gr` to the scheduler; other commands observe the latest state.

Scheduler: The scheduler is responsible for running commands in the execution set according to the program's partial order. Commands can be in one of four states: not executed (NE), speculated (S), committed/taken (C), and committed/not taken (CN). Committed commands form a closed prefix in the partial order: if a command is committed, all prior commands are committed, too.

A command's state determines how the scheduler treats it (Fig. 4). At each step, the scheduler selects a minimal (NE) command in the partial order and executes it, tracing its reads and writes; the runtime hooks ensure the command runs in the latest configuration (filesystem, environment variables, etc.). The scheduler speculatively executes a *number* of upcoming commands, optimistically assuming that the configuration will not change in ways that affect their execution. To speculatively execute commands safely, the scheduler must be able to trace, contain, and merge (or rollback) their results—to achieve this, we execute commands in a virtualized environment (see below).

Once the first command in the partial order program finishes executing, it is directly marked as committed/taken (C). Its results are passed to the runtime hooks, which record its write-set—*i.e.*, the files that it wrote to—to later check for any dependencies with the read- and write-sets of speculated

[1] We say 'opportunistic' rather than 'optimistic', as we modulate our optimism: we will only speculate commands which we can see have *some* hope of succeeding.

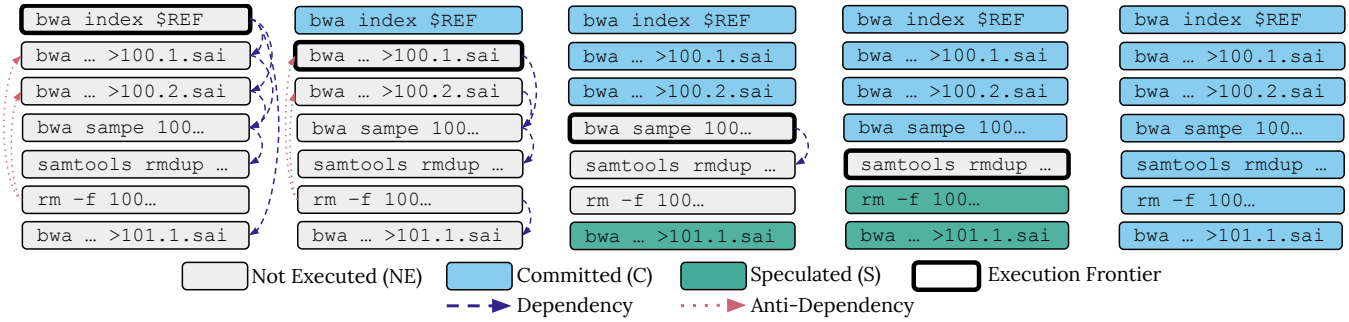


Figure 3: Step-by-step scheduling and orchestration of Köster and Rahmann's [17] script, simplified (Fig. 1).

commands. When a command c that isn't first in the partial order terminates, we check its read-set against the write-set of all preceding commands that were not yet committed when we started speculating c. For example, when speculatively executing the fifth command `samtools rmdup` in the second step of scheduling (Fig. 3, second column), the scheduler checks the write-sets of both invocations of `bwa aln` and the invocation of `bwa sampe` (dashed arrows). If there is no dependency (the read-set of the command is independent from all preceding write-sets), we mark the command speculated (S); if there is a dependency, we leave the command not executed (NE), considering it for execution in the next round of scheduling.

If the scheduler selects a command that is already speculated (S), then we can try to commit: the scheduler makes sure that the results of speculation are valid—*i.e.*, that no extra-command dependency changes were observed since speculation. If no dependencies emerged, the scheduler commits the changes, updating the file system and shell state, and marking the command committed/taken (C). If a speculateed command ends up not being executed (*e.g.*, a branch that was not taken), we mark the command as commited/not taken (CN) to preserve the closed prefix invariant.

Tracing: In order to discover the read- and write-sets of executed commands, we trace filesystem-affecting system calls. Whenever a command performs a read (or write) call, the tracer records it in the command's read (or write) set. We build on Riker's [5] system call tracing, which already has some optimizations to lower overhead: tracing only relevant system calls, and intercepting calls to `libc` via `LD_PRELOAD`.

Virtualization: Our approach requires that the scheduler can control whether (and when) to apply the effects of speculatively executed commands, making them persist in the broader operating environment. We use a combination of custom namespaces [19] and OverlayFS [3]: we can execute commands speculatively in a restricted environment that isolates side-effects between executions.

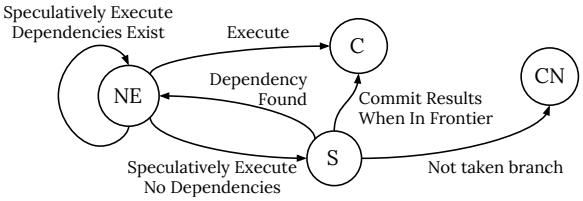
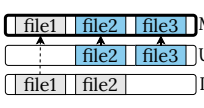


Figure 4: Transition system for command state in the scheduling algorithm.

We use `unshare` to create new namespaces for speculatively executing commands, disallowing any types of side effects—*e.g.*, accessing the network or sending signals—except from writing to a file or reading from a file in the file system. The IPC, mount, network, PID, and user namespaces are unshared. We use OverlayFS to capture any modifications to the underlying system in a separate copy for each speculated command, deciding later whether to merge or drop these changes. OverlayFS provides a layered representation of the filesystem, allowing operation on one workspace copy while keeping another copy clean. OverlayFS has three different layers: merged, lower, and upper. The merged layer presents the union of the lower and upper layers: it is the lower layer with the upper layer's changes applied. The lower layer is the 'base' filesystem—for us, it's the original filesystem, and every overlay shares the same lower layer. The upper layer, unique to each overlay instance, holds the updates to the lower layer. When we speculate commands, they can *only* see the merged layer: they seem to be affecting the whole system, but their changes are caught and stored in the upper layer; files are lazily copied from the lower to the upper layer as writes occur.



If a file exists in both layers, the merged layer can only access the instance of the upper layer, concealing the lower layer—*e.g.*, if `file1` and `file2` pre-exist, running `echo "foo" > file2` and `echo "foo" > file3` results in the merged layer shown on the right. Committing

a speculated command copies the contents of the upper layer to the base file system, overwriting and deleting files when necessary. When we detect a dependency, we discard the upper layer of the speculated command and will re-run it in a fresh overlay. We also capture the stdout/err of speculated commands and release it once they become committed. When a command reads or writes to pipes, we must be careful to capture and replay the pipes appropriately should we need to re-speculate the command.

Fail-fast speculative execution: Containment allows `hs` to speculatively execute commands while collecting their effects and selectively applying them to the underlying system. But some effects must actually happen for a command to execute successfully. For example, a speculated (and thus contained) `curl` would return a failed response, as the `recv` operation over the network would be contained—no actual network communication would be taking place. It's not enough to merely contain effects: we must detect when containment changes behavior and treat the command differently.

Runtime interception can detect side effects in IPC namespaces (*e.g.*, signals) and network namespaces (*e.g.*, `recv`). When the runtime hooks detect such an effect, they (1) kill the speculated command, tearing down relevant containment setup and reclaiming its computational resources, and (2) inform the scheduler to not speculate this command again, since its success depends on non-virtualizable side-effects.

Worst-case performance: A critical requirement for any out-of-order execution optimization is that its worst-case performance does not significantly diverge from the original straightline syntactic-order execution. The worst-case performance in our setting corresponds to all speculations having failed, always discovering dependencies and discarding speculation results. The scheduler design satisfies this requirement since in each round the first non-committed command (the frontier) is executed normally, *i.e.*, with minimal tracing and without virtualization: even if all speculation fails, the execution time will correspond to the baseline execution time with the minimal overhead (from tracing and the communication between the executor and scheduler). For the bioinformatics script (Fig. 1), artificially introducing failures into all speculation yields a 38 minutes execution time (26% slowdown).

Applicability: `hs` is not limited to data processing scripts (Fig. 1); it can be applied to any script that (1) spends significant execution time and resources on external commands, and (2) contains non-trivial dependencies between these commands. Many shell application domains that satisfy these requirements: data processing, build scripts, continuous integration and deployment (CI/CD), scientific computation, orchestration, maintenance, and configuration.

Limitations: Our approach assumes that commands are not malicious. While `unshare` offers more protection than `chroot`, our speculation and virtualization support are not intended to defend against security threats present in scripts. Additionally, we assume that commands do not change their behavior based on their relative execution times or absolute PIDs—as these values will not be the same as in the original executions for speculated commands (due to unsharing of the process namespace). For example, if a command accesses the PID of the previously executed command with `$!`, our speculation engine will not provide the exact same value as in the sequential execution. Our virtualization barrier is only as good as the OS makes it: if, say, reading from a filesystem is observable (*e.g.*, it causes reads to an S3 bucket, which causes billing), then our virtualization will be observable.

3 DISCUSSION

Our proposed system for out-of-order speculative execution promises to improve shell script performance. But beyond these immediate dividends, our work is also foundation on which to build.

Virtualization as a primitive: We use containment and virtualization to optimize the execution of compositions of arbitrary black-box commands that could perform any side-effect on their surrounding system; instead of knowing what a command does *a priori*, we simply run it and observe what it did. Easy and frictionless virtualization could have many other uses for developers—it ought to be a primitive in their toolkit. We envision a higher-order command—call it `try`—where `try cmd` contains `cmd` and records its effect, letting users decide whether to merge its effects onto the underlying system. A motivating example: virtualize complex and potentially risky third-party scripts before committing their results. Today's containerization systems, like Docker [20], set up a *different* environment, making it hard to merge changes to the underlying system—but `try` virtualizes the *existing* system.[2]

Optimal scheduling and performance tradeoffs: Out-of-order speculative execution trades compute for latency; speculating more commands means lower latency but also more CPU and memory usage through failed speculations. Any fixed tradeoff will be wrong some of the time. A better tradeoff would use a configurable, gradual scheduling algorithm that makes bets commensurate with its budget: at low system load, make bigger bets and speculate further out; at high system load, make more conservative bets and speculate less—or not at all.

Harnessing heterogeneous resources: Our simple scheduler speculatively executes all of a script's commands on

[2]https://github.com/binpash/try is a prototype.

the same machine, betting that it has unutilized computational resources (*e.g.*, additional cores) that could be used to speed up the computation. To ensure correct execution, speculated commands are already virtualized and isolated from the main execution environment. With our commands so neatly contained... why stay on the same machine? We could run commands in a variety of 'modern' environments: serverless functions, cloud compute, a distributed cluster. Keeping the local and remote compute synchronized demands a sometimes-eager sometimes-lazy file system synchronization mechanism: the bottleneck becomes synchronizing changes to file system state. Some relevant files could be transferred up front (*e.g.*, binaries, obvious inputs) while the rest could be lazily transferred on demand.

Script maintainability and debuggability: The succinctness of shell scripts facilitates quick prototyping and experimentation, but makes it hard to maintain scripts for longer periods of time. Our proposed approach records several details of a script: execution information and dependencies between commands. Given such detailed information, we could rewrite the input script to expose the true command dependencies. If done with care, rewritten scripts could be more maintainable and debuggable: explicit dependencies provide documentation and can be used by the developer to localize an error. At the same time, a rewritten script should better utilize the underlying resources with less overhead from speculation, tracing, or virtualization. Or, rather than yielding a script, we could produce a `Makefile` or some other explicit representation of dependencies.

More shell optimization: Given the feasibility of our command scheduling and out-of-order execution and the past success of parallelization and distribution... what other optimizations can we apply to the shell? One possibility is *fusion* [4], an optimization from functional programming analogous to loop fusion: we can combine whole command invocations to reduce redundant parsing/unparsing communication overheads between them, enabling whole program optimizations across different commands. Such an approach might be particularly effective on multi-call binaries, like busybox. The space of compiler optimizations is vast, and we suspect that our work could help support a variety of other impactful optimizations, like constant folding, common subexpression elimination, or deforestation [8, 32].

4 RELATED WORK

Automated parallelization for the shell: Recent work on shell-script parallelization and distribution [16, 25, 31] has delivered significant performance benefits by exploiting lightweight command specifications. Our approach, however, does not require *any* command specifications—we infer the necessary command-execution information at runtime.

Explicit dependency encoding: Workflow and build systems [9, 14, 17, 28] explicitly express dependency graphs by manually encoding all input and output dependencies of each step. Encoding dependencies statically and ahead-of-time yields better program schedules, but (1) requires users to provide *all* dependencies or suffer from stale or incorrect results, and (2) cannot express the high dynamism prevalent in shell scripts. Our approach addresses both these challenges.

Speculative execution: Speculation and rollback are not new ideas, with an extensive history not just in architecture but also at the application level, *e.g.*, for system configuration [29] or security checks [24]. In some of these proposals speculation is enabled by modifying the application (*e.g.*, Undo [2]), while others support arbitrary black-box applications [23, 24, 29]. Thread-level speculation is a widely studied technique for extracting parallelism from applications at runtime by speculatively executing parts of them in different threads and rolling them back if dependencies are violated [7, 30]. We build on these ideas, coupling more tightly with the shell's language and semantics: instead of considering the whole script as a black-box application, separating it into very fine-grained tasks, and tracking all interprocess boundaries and low-level application state; we use the script semantics to separate it into logical application components (command invocations). Our approach lets us track less information (file system modifications and shell state), reducing overhead and simplifying our implementation.

Resurgence of shell research: The shell has enjoyed renewed academic interest [5, 10–13, 16, 18, 22, 25–27, 31]. We build on and extend this work: not only do we expand the reach and range of optimizations for the shell, but we extract reusable tools and techniques for others.

5 CONCLUSION

Modern programming languages come with state-of-the-art compilation and optimization machinery readily available to everyday developers. Despite its prominence, the shell lacks such support—partly due to its unusual characteristics, and partly due to historical accident. Neglecting the shell is a mistake, leaving performance and usability gains on the table. We investigate a promising optimization for the shell—out-of-order execution—which offers reusable tools and techniques beyond the optimization itself.

ACKNOWLEDGMENTS

We want to thank the HotOS reviewers, as well as Anirudh Narsipur, Ayush Bhardwaj, Diomidis Spinellis, Felix Stutz, and Siddhartha Prasad. This material is based upon work supported by NSF award CCF 2124184 and DARPA contract HR001120C0191.

REFERENCES

[1] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. 2007. *Compilers: principles, techniques, and tools*. Vol. 2. Addison-wesley Reading.

[2] Aaron B Brown and David A Patterson. 2003. Undo for Operators: Building an Undoable E-mail Store.. In *USENIX Annual Technical Conference, General Track*. 1–14.

[3] Neil Brown, Miklos Szeredi, Amir Goldstein, Vivek Goyal, Randy Dunlap, Linus Torvalds, Pavel Tikhomirov, Kevin Locke, Sargun Dhillon, Chengguang Xu, and Deming Wang. 2022. The Overlay filesystem. *The Linux Kernel documentation* (2022). https://docs.kernel.org/filesystems/overlayfs.html Started in 2014..

[4] Wei-Ngan Chin. 1994. Safe fusion of functional expressions II: Further improvements. *Journal of Functional Programming* 4, 4 (1994), 515âĂŞ555. https://doi.org/10.1017/S0956796800001179

[5] Charlie Curtsinger and Daniel W Barowy. 2022. Riker: Always-Correct and Fast Incremental Builds from Simple Specifications. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 885–898.

[6] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 1–14. https://www.flux.utah.edu/paper/duplyakin-atc19

[7] Alvaro Estebanez, Diego R Llanos, and Arturo Gonzalez-Escribano. 2016. A survey on thread-level speculation techniques. *ACM Computing Surveys (CSUR)* 49, 2 (2016), 1–39.

[8] Andrew Ferguson and Philip Wadler. 1988. When Will Deforestation Stop. In *Glasgow Workshop on Functional Programming*.

[9] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers.. In *USENIX Annual Technical Conference*. 475–488.

[10] Michael Greenberg and Austin J. Blatt. 2020. Executable formal semantics for the POSIX shell. *Proc. ACM Program. Lang.* 4, POPL (2020), 43:1–43:30. https://doi.org/10.1145/3371111

[11] Michael Greenberg, Konstantinos Kallas, and Nikos Vasilakis. 2021. The Future of the Shell: Unix and Beyond. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Ann Arbor, Michigan) *(HotOS '21)*. Association for Computing Machinery, New York, NY, USA, 240–241. https://doi.org/10.1145/3458336.3465296

[12] Michael Greenberg, Konstantinos Kallas, and Nikos Vasilakis. 2021. Unix Shell Programming: The Next 50 Years. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Ann Arbor, Michigan) *(HotOS '21)*. Association for Computing Machinery, New York, NY, USA, 104–111. https://doi.org/10.1145/3458336.3465294

[13] Shivam Handa, Konstantinos Kallas, Nikos Vasilakis, and Martin Rinard. 2021. An Order-aware Dataflow Model for Extracting Shell Script Parallelism. *Proc. ACM Program. Lang.* 4, ICFP, Article 88 (Aug. 2021), 32 pages.

[14] Google Inc. 2015. Bazel. https://bazel.build/

[15] Github Inc. 2022. The top programming languages. https://octoverse.github.com/2022/top-programming-languages.

[16] Konstantinos Kallas, Tammam Mustafa, Jan Bielak, Dimitris Karnikis, Thurston H.Y. Dang, Michael Greenberg, and Nikos Vasilakis. 2022. Practically Correct, Just-in-Time Shell Script Parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, 1–18. https://www.usenix.org/conference/osdi22/presentation/kallas

[17] Johannes Köster and Sven Rahmann. 2012. Snakemake—a scalable bioinformatics workflow engine. *Bioinformatics* 28, 19 (2012), 2520–2522.

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

[19] Linux man-pages project. [n. d.]. namespaces(7) – Linux manual page. https://man7.org/linux/man-pages/man7/namespaces.7.html

[20] Dirk Merkel et al. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux j* 239, 2 (2014), 2.

[21] Matthew Meyerson, Stacey Gabriel, and Gad Getz. 2010. Advances in understanding cancer genomes through second-generation sequencing. *Nature Reviews Genetics* 11, 10 (2010), 685–696.

[22] Jürgen Cito Michael Schröder. 2020. An Empirical Investigation of Command-Line Customization. *arXiv preprint arXiv:2012.10206* (2020). https://arxiv.org/abs/2012.10206

[23] Edmund B Nightingale, Peter M Chen, and Jason Flinn. 2005. Speculative execution in a distributed file system. *ACM SIGOPS operating systems review* 39, 5 (2005), 191–205.

[24] Edmund B Nightingale, Daniel Peek, Peter M Chen, and Jason Flinn. 2008. Parallelizing security checks on commodity hardware. *ACM SIGARCH Computer Architecture News* 36, 1 (2008), 308–318.

[25] Deepti Raghavan, Sadjad Fouladi, Philip Levis, and Matei Zaharia. 2020. POSH: A Data-Aware Shell. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 617–631.

[26] Jiasi Shen, Martin Rinard, and Nikos Vasilakis. 2022. Automatic Synthesis of Parallel Unix Commands and Pipelines with KumQuat. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Seoul, Republic of Korea) *(PPoPP '22)*. Association for Computing Machinery, New York, NY, USA, 431–432. https://doi.org/10.1145/3503221.3508400

[27] Diomidis Spinellis and Marios Fragkoulis. 2017. Extending Unix Pipelines to DAGs. *IEEE Trans. Comput.* 66, 9 (2017), 1547–1561.

[28] Richard M Stallman, Roland McGrath, and Paul Smith. 1988. GNU make. *Free Software Foundation, Boston* (1988).

[29] Ya-Yunn Su, Mona Attariyan, and Jason Flinn. 2007. AutoBash: Improving configuration management with operating system causality analysis. *ACM SIGOPS Operating Systems Review* 41, 6 (2007), 237–250.

[30] Josep Torrellas. 2011. Speculation, Thread-Level. In *Encyclopedia of Parallel Computing*, David Padua (Ed.). Springer US, Boston, MA, 1894–1900. https://doi.org/10.1007/978-0-387-09766-4_170

[31] Nikos Vasilakis, Konstantinos Kallas, Konstantinos Mamouras, Achilles Benetopoulos, and Lazar Cvetković. 2021. PaSh: Light-Touch Data-Parallel Shell Processing. In *Proceedings of the Sixteenth European Conference on Computer Systems*. Association for Computing Machinery, New York, NY, USA, 49–66. https://doi.org/10.1145/3447786.3456228

[32] Philip Wadler. 1988. Deforestation: Transforming programs to eliminate trees. In *ESOP '88*, H. Ganzinger (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 344–358.