Executing Shell Scripts in the Wrong Order, Correctly

Georgios Liargkovas
Brown University & AUEB
Providence, RI, USA
gliargovas@aueb.gr

Konstantinos Kallas
University of Pennsylvania
Philadelphia, USA
kallas@seas.upenn.edu

Michael Greenberg
Stevens Institute of Technology
Hoboken, USA
mgreenbe@stevens.edu

Nikos Vasilakis
Brown University
Providence, USA
nikos@vasilak.is

ABSTRACT

Shell scripts are critical infrastructure, for developers, administrators, and scientists, and therefore deserve the full suite of advances in compiler optimizations. We propose executing scripts out-of-order to better utilize the underlying 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. We address these challenges by 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.

1 INTRODUCTION

Shell programming remains prevalent in today's computing landscape. GitHub steadily rank the shell among the top 10 programming languages for the past decade, and with increasing popularity [11]: in 2020 it jumped to eighth—growing faster than languages such as C and Ruby—and in 2021 it ranked sixth in terms of popularity increase—above languages with highly 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 [4, 7, 8, 10, 13, 15, 18–21, 23].

Despite its popularity, shell tooling has not kept up: weak linters, no debugging, and—our focus—no compilation or optimization. Even though the shell's performance is not what it could be, the shell remains convenient for many long running tasks—*e.g.*, builds, orchestration, continuous integration, and data-processing. The lack of support is not surprising, though: the shell is a glue language, piecing together a polyglot patchwork of individual 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, together enabling a powerful and important optimization to the shell: *out-of-order program execution* [2]. A program's execution order need not be determined by *syntax*, *i.e.*, the order in which blocks or instructions are located in the program, but rather by *semantics*, *i.e.*, 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. Combined, these ingredients provide another advantage over prior work [13, 19, 23] on the shell: we can appropriately trace, contain, and selectively merge a command's effects without any foreknowledge of the command—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: Let us 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 [17]. The script first (a) indexes the reference genome; it then (b) aligns each set of samples based on the genome, (c) combines the results, (d) removes duplicates, and (e) plots a coverage histogram. Running this script for a 152-MB reference genome and 3.3-GB input samples takes about 30 minutes on a 3GHz 16-core machine on Cloudlab [5]. The script invokes a variety of commands: specialized genomics executables (`bwa`, `samtools`), core utilities (`cut`) and custom scripts in interpreted languages (`python plot.py`). 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 big loop iteration is independent of the others and each group's alignment can be done independently. Sadly, the

```
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 [14] that maps sequence reads to a reference genome.

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 substitution, and parameters determined by previous commands. This is in contrast to traditional compiler optimizations working on object code, simply see instruction 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. This, again, is in contrast to the finite and well-defined set of instructions in object code, with generally clear dependencies and effects.

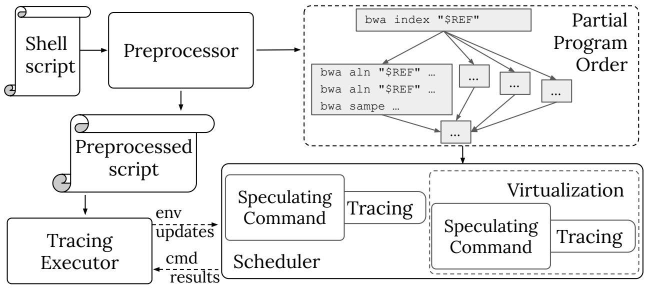


Figure 2: A high-level overview of a speculative out-of-order shell-script executor.

A prescription: While compiler reordering optimizations are traditionally static and pessimistic, our approach for the shell is—must be—*dynamic* and *opportunistic*. A *dynamic* approach circumvents the intractability of ahead-of-time order extraction: our proposed techniques extract information about the execution order incrementally, building a gradual understanding of this order during the execution of the script. An *opportunistic* approach avoids the need for a full understanding of command behavior: our techniques can optimistically execute commands in an isolated environment—dealing with conflicting side-effects as they later arise.[1]

Our approach has three parts (Fig. 2): a script preprocessor, a scheduler, and a tracing executor. The preprocessor translates a script into a partial order on commands or, rather, lines of shell script. It hands this partial order off to be scheduled and executed *speculatively*: the scheduler executes commands opportunistically out-of-order and takes care of rolling back only when dependencies have been violated. It uses (1) tracing to discover command dependencies and detect dependency violations, and (2) containment to shielf against interference and allow rollbacks. The tracing executor looks at the preprocessed version of the original script and communicates with the scheduler; its job is to hide out-of-order execution so that our reorderings are semantically transparent, *i.e.*, the script runs the same.

A relief of symptoms: On a 3GHz 16-core machine on Cloudlab [5], the syntax-guided execution order executes the script in about 30 minutes; the speculative out-of-order execution guided by its semantics completes in 7 minutes and 35 seconds (3.9× speedup).

2 THE TREATMENT, APPLIED

We now apply our approach on the script in Fig. 1, in the process sketching the design of an out-of-order shell script

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

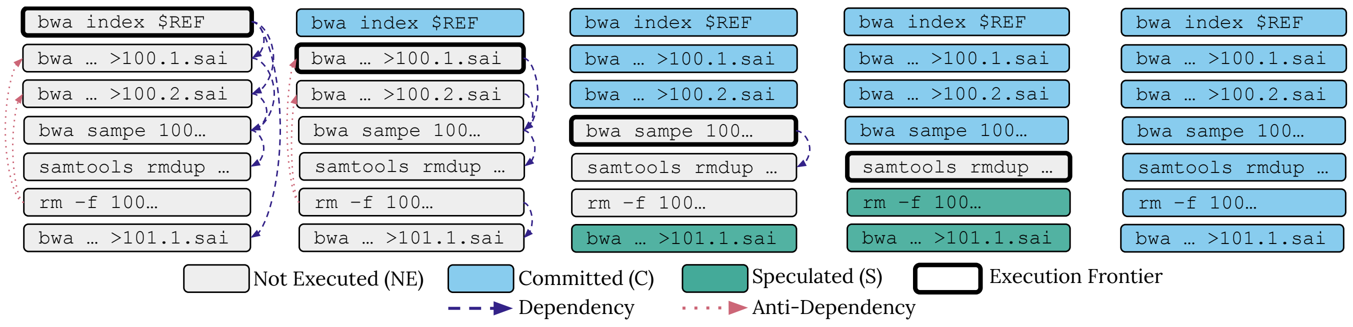


Figure 3: Step-by-step orchestration of a simplified version of the example in Fig. 1.

executor (see Fig. 2). Our proposed executor combines preprocessing, tracing, speculation, and containment.

Preprocessing: First, the shell script is sent through a preprocessor that extracts all commands in the script. 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 stubs that will interact with a tracing executor when executed at runtime. These command nodes are then added to an *execution set* that stores all the commands that need to be executed and is provided to a 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.

For the script in Fig. 1, lines 5, 11, 14-15, 17, and 19-20 would all be replaced with stubs, and their commands would be added to the execution set. A subset of the execution set is shown in 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 they might change during the execution of the script.

Tracing executor: The tracing executor executes the preprocessed script containing the stubs: it blocks whenever it reaches a stub and waits until the scheduler has completed the execution of the particular command, to receive its exit status and observe its effects on the file-system. It also keeps track of extra-command dependencies—*e.g.*, standard variable assignment, special variables (*e.g.*, `$?`), shell state reconfigurations (*e.g.*, `set -e`)—and propagates these to the scheduler. In Fig. 1's script, the executor propagates assignments to variables such as `REF`, `sm`, and `gr` to the scheduler; other commands in the execution set observe the latest state.

Scheduler: The scheduler is responsible for running commands in the execution set in the program (partial) order. Commands can be in one of four states: (NE) not executed, (S) speculated, (C) committed taken, and (CN) committed not taken. An invariant of the execution set is that committed commands form a closed prefix: if a command has committed all its previous commands have committed too.

Fig. 4 shows the possible transitions a command can take depending on its state. At each step, the scheduler takes the first (NE) command with respect to the partial order and executes it while tracing the files that it reads from and writes to; the tracing executor can provide the latest relevant environment updates (such as variable assignments) to run it correctly. The scheduler speculatively executes a number of upcoming commands, assuming that the state of the shell environment and the file system will not change until their actual execution. To speculatively execute commands, the scheduler must be able to decide whether to merge their changes or roll them back—and to achieve this, it execute commands in a virtualized environment (see later).

Once the first command finishes executing, it is marked as (C) committed. Its results are passed to the tracing executor, and its write-set—*i.e.*, the files that it wrote to—is kept to check for any dependencies with the read- and write-sets of speculated commands. The read-set of each speculated command is checked against the write-set of all non committed commands that precede it in the partial order. For example, for the fifth command `samtools rmdup` of Fig. 3's second step, the executor checks the write-sets of both invocations of `bwa aln` and the invocation of `bwa sampe`. If there is no dependency (the read-set of the command is independent from all write-set of preceding commands), the command is marked as (S) speculated; alternatively it is marked as non executed (NE), which means that it will be considered for execution in the next cycle.

If the first command is speculated (S), then instead of executing it, the executor makes sure that its speculation is valid—*i.e.*, that no extra-command dependency changes were observed after its speculation. In this case, its changes are committed to the file system, marking the command with

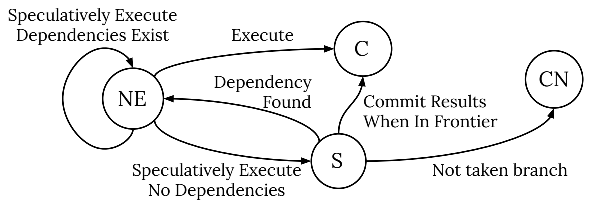


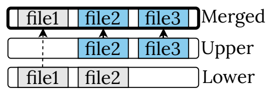
Figure 4: Transition system for command state in the scheduling algorithm.

(C). If a command that was speculated does not end up being executed (a branch that was not taken) it is marked as (CN) to preserve the committed prefix invariant.

Tracing: In order to discover the read- and write-sets of executed commands, we propose tracing their file-accessing system calls. Whenever a command performs a read (or write) call, the tracer stores the file in the command's read (or write) set. Optimizations seem possible: tracing only relevant system calls, and intercepting calls to frequently used `libc` wrappers such as `open` to avoid `ptrace` overhead.

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. To achieve this, the scheduler leverages a combination of custom namespaces [1], `chroot`, and OverlayFS [3]. This combination allows executing commands speculatively in a restricted environment that isolates side-effects between executions.

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. We use OverlayFS to capture modifications to the underlying system to a separate copy for each speculated command, deciding later whether we will merge this copy to the base file system. OverlayFS provides a layered representation of the filesystem, allowing operation on one workspace copy while keeping another copy clean. In its simplest form, OverlayFS is organized in three different layers: lower, upper, and merged. The lower layer contains the filesystem that is shared as-is, along with every different overlay merged instance. The upper layer, unique to each instance, contains every change introduced in the merged layer by a specific instance. Files are lazily copied from the lower to the upper layer when a command attempts to modify one of them.



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 files when necessary. Finding a dependency results in discarding the upper layer of the speculated command and a new OverlayFS.

Finally, we use `chroot` to change the root of the speculated command so that it considers the merged directory of OverlayFS as its root. We also capture the stdout/err of speculated commands and release it once they become committed.

Fail-fast speculative execution: Executing commands using the speculation and containment techniques outlined earlier avoids affecting the environment with arbitrary side effects. But some effects are necessary for a command to execute successfully. For example, a speculated (and thus contained) `curl` instance would return a failed response, as the `read` operation over the network would be contained—no actual network communication would be taking place. Therefore, we should be able to determine that a command attempted to perform an effect that failed, and thus that its speculative execution is invalid and should be terminated.

To address this, we propose runtime interception when such side effects happen (*e.g.*, signals, network accesses). This interception then (1) kills the speculated command, tearing down relevant containment setup and reclaiming its computational resources, and (2) informs the scheduler to not speculate this command again, since its success depends on non-virtualizable side-effects. The command is then marked to avoid re-speculating it in the future.

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 straighline syntactic-order execution. The worst-case performance in our setting corresponds to all speculations having failed, always discovering dependencies and discarding their results. The scheduler design satisfies this requirement since in each cycle 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 Fig. 1's script, failing all speculation *hypothetically* (*i.e.*, it does not happen) would result in 38 minutes (26% slowdown).

Limitations: The approach outlined above assumes that commands are not malicious—and thus the speculation and virtualization support are not intended to provide isolation against security threats present in these commands. Additionally, it assumes 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. For example, if a command accesses the PID of the previously executed command with `$!`, our speculation engine will fail to provide the right value.

3 DISCUSSION

We have proposed concrete tooling to improve shell script performance—out-of-order speculative execution. But our work is also foundation on which to build.

Optimal scheduling and performance tradeoffs: Out-of-order speculative execution trades CPU utilization to improve latency; speculating more commands means lower latency but also more CPU cycles through failed speculations. Any fixed choice of tradeoff will be wrong some of the time. We have not explored this tradeoff adequately; it would be interesting to investigate a configurable and gradual scheduling algorithm that makes bets commensurate with its budget: at low system load, make bigger bets and speculate further out; at high 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 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 hybrid file system synchronization mechanism, part eager part lazy, to make sure that changes to file system state are efficiently exchanged between the speculated and main environment. Some relevant files could be transferred up front (*e.g.*, binaries) while the rest could be lazily transferred when a command tries to read or write them.

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 analogue to function inlining: fuse commands 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 them.

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 all the details of a script: execution information, dependencies between commands—everything. Given such detailed information, we could rewrite the input script to expose the true command dependencies. The rewritten script would be significantly more maintainable and would facilitate debugging, since explicit dependencies provide documentation and can be used by the developer to localize an error. At the same time, the rewritten script should better utilize the underlying resources with no overhead from speculation, tracing, or virtualization. Or, rather than yielding a script, we could produce a `Makefile` or some other explicit representation of dependencies.

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. In contrast to today's containerization systems like Docker [16], which set up a different environment making it hard to merge changes to the underlying system, `try` would virtualize the existing system.

4 RELATED WORK

Automated parallelization for shell scripts: Recent work on shell-script parallelization and distribution [13, 19, 23] has delivered significant performance benefits by exploiting light (but non-zero) command specifications. Contrary to these approaches, our approach does not require *a priori* command specifications—instead it infers the necessary command-execution information at runtime.

Explicit dependency encoding: Several workflow systems [6, 12, 14, 22] allow expressing program dependency graphs, by manually encoding all input and output dependencies of each program step. Such ahead-of-time and static encoding achieves improved program scheduling, but (1) requires users to painstakingly provide all dependencies, and (2) cannot express the high dynamism prevalent in shell scripts. The approach described in this paper alleviates both of these challenges.

Resurgence of shell research: Broader recent work on the shell [4, 7–10, 13, 15, 18–21, 23] highlights renewed interest on and around the shell. We view our work as building on and extending this work: not only can we expand the reach and range of optimizations for the shell, but we can 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 is lacking such infrastructure—partly due to many of its unusual characteristics, and partly due to historical coincidence. As a result, we are missing opportunities to leverage and investigate promising optimizations for the shell but also opportunities to extract reusable tools and techniques beyond these optimizations. In-progress tackling of out-of-order execution in a new system is a first step towards this direction.

REFERENCES

[1] namespaces(7) – linux manual page.

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

[3] Neil Brown. The overlay filesystem. *The Linux Kernel documentation*.

[4] Charlie Curtsinger and Daniel W Barowy. Riker: Always-Correct and fast incremental builds from simple specifications. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 885–898, 2022.

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

[6] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *USENIX Annual Technical Conference*, pages 475–488, 2019.

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

[8] Michael Greenberg, Konstantinos Kallas, and Nikos Vasilakis. The future of the shell: Unix and beyond. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, pages 240–241, New York, NY, USA, 2021. Association for Computing Machinery.

[9] 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, pages 104–111, New York, NY, USA, 2021. Association for Computing Machinery.

[10] Shivam Handa, Konstantinos Kallas, Nikos Vasilakis, and Martin Rinard. An order-aware dataflow model for extracting shell script parallelism. *Proc. ACM Program. Lang.*, 4(ICFP), August 2021.

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

[12] Google Inc. *Bazel*, 2015.

[13] Konstantinos Kallas, Tammam Mustafa, Jan Bielak, Dimitris Karnikis, Thurston H.Y. Dang, Michael Greenberg, and Nikos Vasilakis. Practically correct, just-in-time shell script parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 1–18. USENIX Association, July 2022.

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

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

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

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

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

[19] 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.

[20] Jiasi Shen, Martin Rinard, and Nikos Vasilakis. 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*, PPoPP '22, pages 431–432, New York, NY, USA, 2022. Association for Computing Machinery.

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

[22] Richard M Stallman, Roland McGrath, and Paul Smith. Gnu make. *Free Software Foundation, Boston*, 1988.

[23] 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*, pages 49–66, New York, NY, USA, 2021. Association for Computing Machinery.