The Once and Future Shell

Michael Greenberg
Pomona College

Konstantinos Kallas
University of Pennsylvania

Nikos Vasilakis
MIT

Abstract

The UNIX shell is a powerful, ubiquitous, and reviled tool for managing computer systems. The shell has been largely ignored by academia and industry. While many replacement shells have been proposed, the Unix shell manages to stick around. The shell is a powerful tool that has suffered from inattention for too long. Two recent threads of formal and practical research on the shell enable new approaches. We can help manage the shell's essential shortcomings (dynamism, power, and abstruseness) and address its inessential ones. Improving the shell holds much promise for development, ops, and data processing; we outline several avenues of research building on recent advances.

1 Introduction

The UNIX shell is an attractive choice for specifying succinct and simple scripts for system orchestration, data processing, and other automation tasks. Moreover, it is the first tool that one has access to in every UNIX distribution. Its benefits and its unavoidability make the shell extremely well exercised, if not well loved, in the current status quo. While in principle cloud computing deprecates many shell tasks as automated systems take on various configuration and management tasks, in practice, shell scripts show up everywhere in the cloud regime: Docker, Vagrant, Kubernetes, and other cloud deployments are all managed by shell scripts.

For a long time, the shell was neglected by both academia and industry as irredeemably bad. However, two recent projects—PaSh [37] and POSH [32]—gave us a glimpse of the possibilities, transforming shell scripts to data flow graphs; such graphs can be heavily optimized to produce fast, data-parallel interpretations of what were ordinary shell pipelines. Building on mechanized semantics with convenient parser hooks [15, 16], we can rehabilitate the shell. It is a critical corner of computing; let it enjoy the advantages of nearly fifty years of systems and languages research that has neglected it.

1.1 Shouldn't we just replace the shell?

What if we just gave up on the shell? After all, moving from System V init scripts to systemd has arguably improved the Linux boot sequence. Why install packages manually when you can specify base images in Docker and Vagrant, and Travis CI will let you specify `addons` to provision per-platform. While some simple shell scripts could just as well be Python scripts, complex pipelines are much harder to port over. Using `popen` or even support libraries like Plumbum lack the easy composition enjoyed in the shell [14, 19].

Moves to systemd, Docker, Vagrant, and other 'modern' services have indeed improved things a great deal, but there have been serious regressions. Systemd uses its own variable expansion regime, slightly different from the shell's... encouraging you to run `sh -c` if you have an actual pipeline to run. Dockerfiles and CI config files are *almost* shell scripts, but there's no convenient way to just execute the `RUN` commands or `script` lists they contain. And few are the Vagrantfiles that don't call out to some `provision.sh` to set up dependencies.

Giving up on the shell means that each 'modern' system tool will have its own janky quasi-shell language: a worse situation. In our diagnosis, the desire to give up on the shell stems from three problems that make it hard to think clearly about the shell and make it reach its full potential:

1. The shell contains a language that can be used to compose arbitrary commands, that can be written in arbitrary languages, and have arbitrary behaviors.
2. The shell's execution depends on a variety of dynamic components, such as the state of the file system and the values of environment variables.
3. The shell's semantics is black magic, specified in a 119 page impenetrable document that is the POSIX shell specification (with an extra 160pp on utilities!).

These three characteristics make it very difficult to approach the issues that the shell has using a principled approach, as any principled solution (i) should be applicable on a wide variety of arbitrary commands, (ii) should account for its dynamic nature, and (iii) should handle the intricacies of its semantics.

1.2 The shell is actually good

In the rest of this paper, we argue that the shell is a useful abstraction deserving of our attention despite its imperfections (Section 2). We identify enabling technologies that point to a way forward for the shell (Section 3) and what we see as the next steps for shell and related technologies (Section 4).

2 The Good, the Bad, and the Ugly

The shell is a mixed bag: ubiquity, power, clumsiness, obscurity. Here are some aspects of the shell that are (i) worth keeping (*the good*), (ii) essential but challenging (*the bad*), and (iii) inessential flaws that can be addressed (*the ugly*).

2.1 The Good

The UNIX shell hits a sweet spot between succinctness, expressive power, and performance. McIlroy presents a striking example [5], but the shell can go further: over 100 lines of Java code that perform a temperature analysis [40] can be translated to a four-stage pipeline `cut -c 89-92 | grep -v 999 | sort -rn | head -n1` (1 line, 48 characters) of comparable performance. So what are the ingredients of the shell's success?

G1: Universal composition. The shell is a natural composer of programs. More than any other language, the shell makes it easy to combine a variety of existing tools written themselves in a broad array of languages. Composing existing tools is a giant leap towards productivity, reliability, and simplicity—after all, "the most radical possible solution for constructing software is not to construct it at all." [8]. Unix aids by promoting a component philosophy [27] and by serving as the *de facto* component library [25].

G2: Stream processing. The UNIX shell embeds a small domain-specific language (DSL) for expressing pipelined stream computations—semantically close to Kahn process networks [21] and co-routines [22]. Coupled with task-based parallelism, filesystem-backed naming and indirection (of both code and data), language-agnostic composition, and higher-order primitives like `xargs`, this DSL turns out to be quite powerful.

G3: Unix-native. The features and abstractions of the shell are well suited to the Unix file system and file-based abstractions. UNIX can be viewed as a naming service, mapping strings to longer strings, be it data files or programs. Naming is both convenient and essential to (powerful classes of) computation [35]. Processes, PATH entries, files, streams, environment variables are all names, and the shell is the tool for manipulating and interacting with names.

G4: Interactive. The shell is not just a programming language, but the lived-in environment. By having the entire environment be the program context, the shell lowers the barrier between interactive and non-interactive use. Interactivity is further facilitated by commands that are short and which often take single-letter flags with default options informed by real practical use [5].

2.2 The Bad

Some of the shell's essential characteristics make life difficult (Section 1). It's hard to imagine 'addressing' these characteristics without turning the shell into something it isn't; it's hard to get the good of the shell without these bad qualities [14].

B1: Too arbitrary. The shell's virtue of limitless composition (**G1**) is also its vice: the shell can compose arbitrary commands written in arbitrary languages. Any 'simple' shell command may translate to an `execve` of some arbitrary executable with arbitrary, unknown behaviors. Calls to `execve` make unified analysis very challenging, as different source languages won't share semantics; binary analysis—the lowest common denominator—cannot discover high-level invariants.

B2: Too dynamic. The behavior of a shell program cannot be known statically: a simple `grep $CWD -in ~/.*shrc` depends primarily on dynamically computed values, including the state of the file system, the current directory, environment variables, and unexpanded strings. Harder still, shell scripts change the world: normal programs mutate the heap, but shell programs mutate the entire environment in which they run.

B3: Too obscure. The semantics of the shell and common commands are documented in 300pp of standardese [3]. To be able to reason about a script's behavior, one needs to understand the exact behavior of its composition operators, the role of the environment, and the intricate state of the shell interpreter. Furthermore, there is no *single* shell environment. Multiple shells (with subtle behavior differences) coexist in the same machine: a pared down shell [1,7] is used for startup scripts, while bash [34] is a common interactive choice. Every shell extends POSIX in its own way [16,18].

2.3 The Ugly

The shells good and bad sides are intertwined. The shell also has several flaws that (i) prevent it from being used for a wider variety of tasks, (ii) make the life of shell developers very difficult (leading to frustrated revulsion [11]), but (iii) aren't *essential* to the character of the shell.

U1: Error-proneness. While all dynamic programming languages suffer from bugs that manifest as runtime errors,

the shell is known for its many potential sources of error—with dire consequences! A bug in a normal program might corrupt the heap, produce erroneous results, or lead to a crash. In a shell program, the stakes are higher: the "heap" is the filesystem. Buggy shell programs have wiped hard drives!

U2: Performance doesn't scale. While shell scripts have acceptable performance in a single-core setting, they're not tuned for multicore machines, clusters of nodes, or very large inputs. Unlike other programming languages, the performance of shell scripts is dominated by the performance of the commands that they compose, and unfortunately, most shell commands do not scale. This leads users to replace parts of their scripts with programs in parallel frameworks, an error-prone process that requires significant effort.

U3: Redundant recomputation. Furthermore, small changes to a scripts input require rerunning the whole script, leading to many hours of wasted redundant computation. Domain specific solutions (such as build systems, e.g., `make`) address this issue for their use cases, but do not generalize.

U4: No support for contemporary deployments. The shell's core abstractions were designed to facilitate orchestration, management, and processing on a single machine. However, the overabundance of non-solutions—*e.g.*, `pssh`, GNU `parallel`, web interfaces—for these classes of computation on today's distributed environments indicates an impedance mismatch between what the shell provides and the needs of these environments. This mismatch is caused by shell programs being pervasively side-effectful, and exacerbated by classic single-system image issues: a whole swath of configuration scripts, program and library paths, environment variables are configured *ad hoc*—and a set of composition primitives that do not compose at scale.

3 Enablers

The shell has been around for half a century, so the addressable challenges (**U1-4**) outlined in the previous section have been around, too. What's changed today? Three key enablers—two recent research threads and one proposed here—combine to unlock the shell's missing potential. These enablers have both conceptual merit, allowing others to build on their ideas, as well as practical value, as they are concrete open-source systems that can be used in other research.

3.1 Two Recent Enablers

Two recent enablers are (1) the formalization of the POSIX shell, and (2) light-touch parallelization and distribution transformation systems for the shell.

E1: Libdash & Smoosh Smoosh [16] is a recent effort to formalize the semantics of the POSIX shell, addressing the obscurity and subtleties of the prose standard [3] (**B3**). It has two parts: (1) libdash, a linkable parsing library that supports parsing POSIX shell scripts and pretty printing their ASTs, and (2) a POSIX formalization mechanized in Lem, an ML-like language that can be compiled to Coq for proofs or OCaml for execution. Smoosh has found several bugs in popular shell implementations as well as the POSIX specification and test suite.

Smoosh is a key enabler because it allows others to (1) study the POSIX shell and its nuanced behavior, (2) understand the divergences introduced by different shells, and (3) design and implement (formal) analyses and transformations on top of its symbolic, executable semantics.

E2: PaSh & POSH PaSh [37] and POSH [32] both proposed annotation languages as a high-level specification interface for dealing with the challenges of unknown command behavior (**B1**). Both use specifications to characterize commands' important properties, like how they interact with state and streams. These specifications make shell scripts with arbitrary commands amenable to analysis and transformation. Annotations are written once for each command and can be aggregated in annotation libraries and shared between users. Both PaSh and POSH use these specifications to transform shell pipelines to dataflow graphs with specific properties, that they then optimize to either (1) achieve data parallelism in a multicore setting [37], or (2) offload computation efficiently in a distributed setting [32].

PaSh and POSH are key enablers because (1) they show that the shell too can benefit from automated transformation frameworks, (2) they offer a baseline for other tools to improve performance performance, (3) they propose annotation languages that enable reasoning about arbitrary commands, and (4) they propose a dataflow model for a subset of the shell—a foundation for further analyses and transformations.

3.2 A proposal...

These two threads of recent work are a stepping stone for further research on the shell. To illustrate this, we sketch in detail one possible next step: a POSIX-shell parallelizing system, called Just a shell, or Jash for short; later we sketch in broad strokes a number of research directions involving the shell (Section 4).

PaSh and POSH showed that the performance of shell scripts is not dominated by the performance of the shell interpreter, but rather by the commands that the scripts compose [32, 37]. Before these systems, the only option that users had for improving shell script performance was to rewrite parts of their scripts in other, usually lower-level languages—a process that is error-prone and might not even lead to improvements. PaSh and POSH identified fragments of shell

scripts that can benefit from automated performance improvements and built optimizing compilers that achieve *order-of-magnitude* performance improvements in these fragments. These tools have paved the way for exciting performance optimization on the shell, but face two challenges.

Expansion↔evaluation As discussed earlier (**B2**), the execution of shell scripts depends on several dynamic components such the state of the file system, the current working directory, and the values of environment variables. Therefore, ahead-of-time solutions, such as the ones proposed by PaSh and POSH, choose between being conservative and ineffective or optimistic and unsound. Consider the following script, which is based on the original `spell` program by Johnson [4], lightly modified for modern environments.

```
1 FILES="$@"
2 cat $FILES | tr A-Z a-z |
3     tr -cs A-Za-z '\n' | sort -u |
4     comm -13 $DICT -
```

The above script checks the spelling of a set of input files based on a dictionary file that is referenced by the environment variable `$DICT`. An ahead-of-timecompiler has no knowledge of the input files and thus cannot properly decide if and how to parallelize or distribute the above pipeline—*i.e.*, neither PaSh nor POSH optimize this script.

To support the shell's dynamism (**B2**), Jash's optimization subsystem will run as a just-in-time (JIT) compiler. The JIT subsystem tightly couples with the shell, switching back and forth between interpretation and optimization; interpretation is provided by the user's original shell and deals with dynamic features such as parameter expansion, and optimization is provided by the core analysis and transformation infrastructure and deals with parallelization. This architecture allows Jash to call the compiler at the right time—with as much runtime information available as possible—resulting in a system that can perform sound optimizations.

By running just-in-time, the optimization subsystem has access to crucial information regarding performance optimizations—*e.g.*, file sizes, mappings from filesystems to physical media, and system load. Jash can determine in the moment whether it is even worth attempting parallelization for small inputs. For the above example, the JIT compiler would first determine the input files, expand `$DICT`, and then determine how to parallelize the pipeline in lines 2-4. Expanding the parameters before running the pipeline must be done with care: early expansions shouldn't have side-effects; Smoosh's semantics is critical for this kind of reasoning. To sum up: Jash will not only be sound with respect to shell semantics, it will be more effective.

Restrictive assumptions PaSh and POSH focus on achieving performance improvements given an abundance of underlying computational resources. PaSh assumes a machine with lots of available storage for buffering and high storage throughput. POSH assumes a cluster where computational resources per node are unlimited and thus co-locating the computation with the data never degrades performance. These assumptions are fine to make, but do not represent the entire population of shell users, which ranges from owners of palm-sized computers to administrators of supercomputers

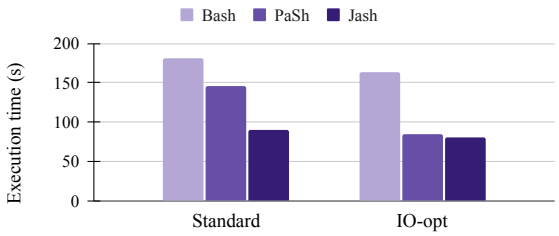

Figure 1: Executing a script that sorts the words of a 3GB input file with bash, PaSh, and the Jash prototype. Both instances are `c5.2xlarge` AWS EC2. The standard instance has a `gp2` disk (100 IOPS that bursts to 3K) while the IO-opt has a `gp3` disk (15K IOPS). PaSh performs worse on 'Standard' because it doesn't take system resources into account.

To relax these assumptions, we are developing a resource-aware optimization procedure that ensures performance improvements on a multitude of underlying platforms. The procedure is built on top of a cost-aware dataflow model, allowing for an extensible graph rewriting system that applies transformations with certain performance objectives within a specified cost budget. The JIT compiler keeps the optimization procedure up-to-date on the currently available resources of the underlying infrastructure as well as the size and characteristics of the input. Figure 1 shows the execution time of a script with bash, PaSh, and the Jash prototype on two EC2 instances with different IO capabilities—Jash exhibits better performance in more settings, because it is resource-aware.

The JIT compiler and resource-aware optimization yield a shell that can be used by *anyone* on *any infrastructure* and still lead to performance benefits for a wider variety of scripts and input workloads.

3.3 ...and Enabler

Jash includes a few other standalone contributions such as high-performance libdash-JIT interactions, but its primary contribution is its architecture, which enables other work.

E3: Jash Jash promises to operate on full-POSIX shell scripts, namely ones exhibiting the full range of dynamic behavior found in modern shell scripts—including but not limited to variable expansion, command substitution, process substitution. Jash's JIT-based approach helps us surmount an obstacle essential to the shell—namely, the limitations of

static insights in the dynamic world of the shell (**B2**). Combined, **E1–3** open exciting new research directions.

4 Where do we go from here?

The aforementioned enablers open exciting new directions in distributed computing, incremental computation, expressiveness support, and tooling for the shell.

Distribution Building a distributed UNIX equivalent, in which UNIX abstractions transcend single-computer boundaries, has been a goal since the 1970s [29–31, 38, 39]. Most attempts implemented full-fledged distributed operating systems, but enablers **E1–3** hint at the option of a language-systems hybrid: a thin but sophisticated rewriting-based shim á la Jash would have simplified the design of these systems, and would have armed them with the semantic foundation necessary for tackling unavoidable distribution tradeoffs [2, 12, 26].

A few other connections can enable worthwhile research directions. UNIX's toolbox philosophy, built around purely-functional primitives, aligns well with recent trends in both micro-services and server-less computing [20]. The foundation of PaSh and POSH annotations (**E2**), paired with Jash's cost-aware late-bound optimizations (**E3**), can help exploit this alignment. On this foundation, the dynamic nature of distributed systems with that of the shell work synergistically; in contrast to existing cluster-computing systems [41], the shell's dynamic nature does not permit a static AoT analysis to exploit distribution. Finally, a deep understanding of the shell language (**E1**) can guide the design of distribution-friendly language subsets—with promising candidates being a streaming DSL [36] or a concurrency DSL [13].

Incremental Computation The annotation systems proposed by PaSh and POSH and the JIT optimization subsystem in Jash (**E2**, **E3**), can also serve as the foundation of an incremental computation framework to address **U3**. Incremental computation is especially useful in the context of the shell: developing a shell script is an iterative cycle of coding, testing, inspecting, and tweaking. Furthermore, shell scripts are often used for data downloading, extracting, cleaning, and other processing tasks. Re-executing such scripts on large datasets can be prohibitively expensive!

Incremental computation [33] has been widely studied for a variety of domains, from restricted ones, such as stream processing [28]; to general ones, such as threading support [6]. However, in almost all cases it requires extending the programming model in order to expose necessary information such as input-output dependencies that can be exploited by the IC compiler and runtime [9, 17]. Simply extending the shell language would break legacy scripts and restrict the possible adoption of such a framework.

PaSh and POSH's annotation languages are the missing link, exposing the necessary information for an incremental computation framework. The shell already encapsulates foreign code as arbitrary commands, irrespective of their implementation details. The JIT framework will have up-to-date information on the inputs to these annotated program. Combined, we have the critical building blocks for a runtime that incrementally reinterprets a script given changes of its input.

Heuristic support Perhaps the hardest part of supporting shell programming is reasoning about the commands users run. Both PaSh and POSH use annotations to specify parallelization-relevant aspects of commands' behavior. These annotations were hand-written after carefully inspecting each command to determine its parallelizability properties. Hand-writing annotations even for common commands is quite a task, but users will need help not just with standard utilities, but their own programs (**B1**). Formal methods techniques such as fuzz testing, program analysis, and active learning could (i) test that a command conforms to its annotation or even (ii) learn a command's annotation by inspecting its behavior. Testing and inferring annotations would enable large annotation libraries (formal, symbolic man pages) concerning a variety of properties (e.g., concurrency/parallelism, expected command line argument syntax, filesystem access, expected runtime/memory usage).

Better annotations on many properties allow for a variety of analyses—substantially more than the syntactic checks of ShellCheck [24], the man-page-directed listings of Explain-Shell [23], or the purely textual model of NoFAQ [10]. Two promising directions: identifying errors in a shell script (i.e., commands that could exit with non-zero status), and a shell tutor (**B3**, **U1**). The tutor could use the library of annotations as a database to either answer queries about particular commands or to guide users while they develop a script. Building on the JIT execution framework, one could develop a sound analysis that detects command misuse at runtime, due to missing dynamic information (**B2**), but as early as possible.

Formal support Smoosh provides formal support for building support systems for the shell (**B1**). Rich command annotations feed back into more robust symbolic execution and program analysis tools. The JIT optimization subsystem can be proved correct with respect to Smoosh's formal semantics and command annotations. Just as formal work on undefined behavior in C has supported both tool authors and standards writers, formal work on the shell promises continued improvements in standards and shared understanding.

5 Rehabilitating the shell

Building on recent advances, the shell is a promising area of research. Neglected for so long, improving the shell will improve the experience for users of many stripes (development, ops, data processing, and novices). The shell warrants a fresh look from the research community.

Acknowledgments

We want to thank the HotOS reviewers for their thoughtful feedback. This material is based upon work supported by DARPA contract no. HR00112020013 and no. HR001120C0191, and NSF awards CCF 1763514 and 2008096. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect those of DARPA or NSF.

References

[1] Dash (debian almquist shell).

[2] Daniel Abadi. Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. *Computer*, 45(2):37–42, 2012.

[3] The Austin Group. Posix.1 2017: The open group base specifications issue 7 (ieee std 1003.1-2008), 2018.

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

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

[6] Pramod Bhatotia, Pedro Fonseca, Umut A Acar, Björn B Brandenburg, and Rodrigo Rodrigues. ithreads: A threading library for parallel incremental computation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 645–659, 2015.

[7] Stephen R Bourne. *An introduction to the UNIX shell*. Bell Laboratories. Computing Science, 1978.

[8] Frederick P. Brooks, Jr. No silver bullet—essence and accidents of software engineering. *Computer*, 20(4):10–19, April 1987.

[9] Sebastian Burckhardt, Daan Leijen, Caitlin Sadowski, Jaeheon Yi, and Thomas Ball. Two for the price of one: A model for parallel and incremental computation. *ACM SIGPLAN Notices*, 46(10):427–444, 2011.

[10] Loris D'Antoni, Rishabh Singh, and Michael Vaughn. Nofaq: Synthesizing command repairs from examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 582–592, New York, NY, USA, 2017. Association for Computing Machinery.

[11] Simson Garfinkle, Daniel Weise, and Steven Strassmann. *UNIX-Hater Handbook*. IDG Books Worldwide, Inc., 1994.

[12] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002.

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

[14] Michael Greenberg. Word expansion supports posix shell interactivity. In *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming*, Programming'18 Companion, page 153–160, New York, NY, USA, 2018. Association for Computing Machinery.

[15] Michael Greenberg. Smoosh: the symbolic, mechanized, observable, operational shell, 2019. Current version: 0.1.

[16] 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.

[17] Matthew A Hammer, Khoo Yit Phang, Michael Hicks, and Jeffrey S Foster. Adapton: Composable, demand-driven incremental computation. *ACM SIGPLAN Notices*, 49(6):156–166, 2014.

[18] Helmut Herold. *Linux-Unix-Shells: Bourne-Shell, Korn-Shell, C-Shell, bash, tcsh*. Pearson Deutschland GmbH, 1999.

[19] https://github.com/tomerfiliba/plumbum/graphs/contributors. Plumbum: shell combinators, January 2018.

[20] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*, 2019.

[21] Gilles Kahn. The semantics of a simple language for parallel programming. *Information Processing*, 74:471–475, 1974.

[22] Gilles Kahn and David B. MacQueen. Coroutines and networks of parallel processes. *Information Processing*, 77:993–998, 1977.

[23] Idan Kamara. explainshell, 2016.

[24] koalaman. Shellcheck, 2016.

[25] Butler W Lampson. Software components: Only the giants survive. In *Computer Systems: Theory, Technology, and Applications*, pages 137–146. Springer, January 2004.

[26] Haonan Lu, Christopher Hodsdon, Khiem Ngo, Shuai Mu, and Wyatt Lloyd. The snow theorem and latency-optimal read-only transactions. In *OSDI*, pages 135–150, 2016.

[27] M McIlroy, EN Pinson, and BA Tague. Unix time-sharing system. *The Bell system technical journal*, 57(6):1899–1904, 1978.

[28] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. Differential dataflow. In *CIDR*, 2013.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] Ganesan Ramalingam and Thomas Reps. A categorized bibliography on incremental computation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 502–510, 1993.

[34] Chet Ramey. Bash, the bourne- again shell. In *Proceedings of The Romanian Open Systems Conference & Exhibition (ROSE 1994), The Romanian UNIX User's Group (GURU)*, pages 3–5, 1994.

[35] Olin Shivers. What's in a name?, 2018. Accessed: 2018-09-27.

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

[37] Nikos Vasilakis, Konstantinos Kallas, Konstantinos Mamouras, Achilleas Benetopoulos, and Lazar Cvetković. Pash: Light-touch data-parallel shell processing. In *Proceedings of EuroSys 2021*, April 2021.

[38] Nikos Vasilakis, Ben Karel, and Jonathan M. Smith. From lone dwarfs to giant superclusters: Rethinking operating system abstractions for the cloud. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*, HOTOS'15, pages 15–15, Berkeley, CA, USA, 2015. USENIX Association.

[39] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The LOCUS distributed operating system. In *ACM SIGOPS Operating Systems Review*, volume 17, pages 49–70. Acm, 1983.

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

[41] 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.