JUST-IN-TIME SCALE-OUT OF SHELL PROGRAMS, CORRECTLY

Konstantinos Kallas

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2024

Supervisor of Dissertation

Rajeev Alur, Zisman Family Professor of Computer and Information Science

Graduate Group Chairperson

Mayur Naik, Professor of Computer and Information Science

Dissertation Committee

Sebastian Angel, Raj and Neera Singh Assistant Professor of Computer and Information Science
Vincent Liu, Assistant Professor of Computer and Information Science
Benjamin C. Pierce, Henry Salvatori Professor of Computer and Information Science
Keith Winstein, Associate Professor of Computer Science, Stanford University

JUST-IN-TIME SCALE-OUT OF SHELL PROGRAMS, CORRECTLY

COPYRIGHT

2024

Konstantinos Kallas

Dedicated to Giannis, Katerina, and Nikos.

ACKNOWLEDGEMENT

All the achievements and successes in my life, with no exception, have been achieved through collaborations and discussions with others. This dissertation would not be complete without acknowledging all the people that have contributed to it in direct or indirect ways.

First and foremost, I would like to thank my PhD advisor, Rajeev Alur. From the start of my PhD, Rajeev has managed to grant me independence while at the same time being there to support me in any way I needed—two advisor traits that I previously thought were mutually exclusive and not possible together. My growth as a researcher, academic, and individual has been shaped by Rajeev, and I aspire to be even a fraction as good an advisor as he has been. My gratitude to Rajeev is immense and whatever I write in this paragraph will not be enough to capture how much I appreciate him.

Another person who has been very important for my PhD is Nikos Vasilakis, a very close collaborator, my informal second advisor, and an extremely good friend. It was one of the happiest accidents that I met Nikos and started collaborating with him; as is often the case in our relationship what I consider an accident and take for granted might have been a carefully planned event from his side, but let's run with this for now. All of the work in this dissertation and my development as a researcher could not have been done without him.

A lot of my research has been done in close collaboration and with the support of several great professors at Penn and elsewhere. Michael Greenberg has brought his ingenuity and PL expertise in all our collaborations; in addition to that, his presence makes every research meeting and encounter significantly more fun. Sebastian Angel and Vincent Liu have been incredibly constructive and supportive, helping me develop as a researcher and significantly improving the quality of my work. Benjamin Lee taught one of the most important courses for my development as a systems researcher. Benjamin Pierce has provided a lot of crucial feedback on talks and this dissertation. Stephanie Weirich and Steve Zdancewic have given me very useful feedback and have made me feel embraced in PLClub through their presence and board game nights. Keith Winstein provided crucial feedback on my job talk and supported me a lot as a member of this dissertation's committee. I also want to thank Konstantinos Mamouras and Martin Rinard for providing useful feedback for various aspects of my work. Jonathan Smith has received an obscene number of emails from me request-

ing reference letters, and I thank him greatly for that. Finally, I want to thank Leonidas Lampropoulos, who almost single-handedly convinced me to come to Penn for my PhD.

In addition to the professors, I owe a big thanks to the administrators and other staff at Penn, most importantly Britton, Cheryl, and Maggie, whose daily hard work ensures that everything runs smoothly without us even noticing.

A significant part of my development is owed to my internships at Microsoft Research and AWS. My mentors in these internships, Sebastian Burckhardt and Daniel Schwartz-Narbonne, guided me to do exciting work in areas that I had no experience previously. Big part of the output and enjoyment of my internships is also owed to other collaborators and interns, including Badrish, David, Mark, Kareem, Malte, Debashmita, and Adrian. Last but not least, through my internship at AWS I met Felipe, through whom I met Elena, two of my closest friends. My life would be seriously lacking without them and their love.

I have collaborated with several people throughout my PhD, spending so much time with them that it is hard to distinguish between friends and collaborators at this point. Caleb Stanford was my first mentor in the program, taking me under his wing and guiding me through the maze that is a PhD. Filip Niksic was also a mentor and the person that I most often called for an impromptu night out while he lived in Philadelphia. Both Caleb and Filip quickly became very dear friends, and working and hanging out with them kept me going throughout the first few years of my PhD. Doing research with them was extremely stimulating, and spending time with them was a lot of fun! Haoran Zhang has been a very close colleague and friend in the last few years of my PhD. Haoran is one of the smartest and most productive people I know and working with him has been very enjoyable. Tammam Mustafa was one of my first mentees, and spoiled me with how quickly he grew and became independent. Tammam's spark and excitement are incomparable and working with him and listening to his ideas is always a great pleasure. Jiali Xing has been a great colleague and friend that I met in a course and immediately stuck with due to his immense kindness and warmth. Akis Giannoukos and Spyros Pavlatos have been so much fun to work with in the last few years of my PhD. Finally, Clara Schneidewind accompanied me in a year-long fight against the Coq theorem prover during my first PhD year. In addition to the above, I have collaborated and spent a lot of time with the following fantastic set of people: Lazar Cvetković, Shivam Handa, Felix Stutz, Grigoris Ntousakis, Georgios Liargkovas, Nikos Pagonas, Yizheng

Xie, Ezri Zhu, Evangelos Lamprou, Mayank Keoliya, Dimitris Karnikis, Dimitra Leventi, Thurston Dang, Stephen Mell, Anirudh Narsipur, Siddhartha Prasad, Eric Wang, Seong-Heon Jung, Oğuzhan Çölkesen, Seth Sabar, Zhicheng Huang, Ramiz Dundar, and Max Henri Demoulin.

Surviving and being happy during my PhD has depended a lot on all the friends that I made in Philadelphia. It turns out that it is very hard to precisely remember all parts of one's life going back six years so don't take it personally if your name is missing from this list. Thanks Juan, Giorgo, Paul, Anton, Kishor, Anne Marie, Dimitri, Nicolo, Joey, Max, Santi, Luiz, Miguel, Taso, Melissa, Nacho, Juan, Yannick, Elena, Ingrid, Estella, Mariona, Ariana, Giorgo, Charilae, Marilena, Vasiliki, Aggele, Mariliza, Niko, Paulo, Artemis, Charis, Nektarie, Vageli, Stefane, Lef, Maria, Lefteri, Alex, Rado, Irene, Yao, Li-yao, Robbie, Stelio, Alex, Louka, and Marko. Two of the friends that I made in Philadelphia, Giorgos and Eliza, deserve a special mention because they consistently made my days more colorful and bright. I have spent so much time with them that one wonders how they are still not bored of me.

The daily life during my PhD was significantly more pleasant because of all my officemates (broadly defined). Some of them include: Neeraj, Omar, Gautam, Chris, Nick, Phillip, Gerald, Liangcheng, Yiyun, Jessica, Aalok, Suguman, Bhavana, Edo, Haoxian, Elizabeth, Pardis, Alaia, Cassia, Jonathan, Lucas, Halley, Lawrence, Mukund, Aaditya, Pritam, Alex, Antal, Hengchu, Antuan, and Solomon.

In addition to all the connections in the US, I need to thank my friends from Greece (and Martin from Bulgaria), that have kept me afloat in difficult times, and have made my winters and summers beautiful. A special thanks goes to Kostis, who was not only part of my life in Greece, but ended up in New York City and made me appreciate Astoria.

I want to acknowledge that my decision to pursue a PhD and come to Penn was largely influenced by my experience at NTUA in Greece, both with professors and with other students. My undergraduate advisor, Kostis Sagonas, has been an amazing mentor and inspired my love for compilers and PL. Petros Maragkos was the first to truly inspire me to be precise, concise, and scientific with language; I owe a lot to his lectures. Achilles, Manos, and Thymios were also instrumental. We influenced each other greatly during our last years at NTUA, and our relationships have remained strong since then.

Before closing, I want to acknowledge that big part of my intellectual and emotional growth is owed to my closest companions: Z, D, L, and S. Each one contributed a piece of me, making me who I am.

Finally, I want to thank my family, and most importantly my father Giannis, my mother Katerina, and my brother Nikos. I owe a different part of myself to each of them and without their continuous support I would not be where I am today. Last but not least, part of why I have chosen academia as my life path is due to my grandmother Pepi, who somehow always knew I would follow this path and planted this seed in me at a very young age.

ABSTRACT

JUST-IN-TIME SCALE-OUT OF SHELL PROGRAMS, CORRECTLY

Konstantinos Kallas

Rajeev Alur

Shell programs are critical infrastructure for developers, administrators, and scientists. They are used for all kinds of complex tasks, often as "glue" for succinctly composing existing computational components. Unfortunately, they do not enjoy access to automated performance optimizations typically found in other language environments—including parallelization for scaling out on multicore CPUs and distribution to support processing of data that does not fit on a single machine. This unfortunate state of affairs is due to three fundamental challenges inherent to the shell: (1) shell programs compose arbitrary black-box software components (commands) that are developed in multiple programming languages and cannot be analyzed in a unified way; (2) the language of the shell offers primitives that are highly dynamic, making static analysis intractable; and (3) the shell specification is complex and different implementations vary significantly, making it extremely hard for optimizations to achieve compliance with existing shells, jeopardizing backwards compatibility.

In this dissertation, I propose a novel compilation architecture that addresses the three aforementioned challenges using: (1) a command specification framework to capture command behavior; (2) a just-in-time architecture that applies optimizations at runtime after dynamic information has been resolved; and (3) a shell-to-shell compiler shim whose generated optimized shell programs can be executed by the original shell interpreter. I develop a concrete instantiation of this architecture in a high-performance open-source system called PaSh. I demonstrate the benefits of this compilation architecture on real-world programs using three concrete optimizations: automatically parallelizing, distributing, and reordering the execution of shell programs—achieving significant speedups without jeopardizing compliance with the underlying shell.

TABLE OF CONTENTS

ACKNOWLEDGEMENT iv

ABSTRACT viii

LIST OF TABLES xii

LIST OF ILLUSTRATIONS xiii

CHAPTER 1 : Introduction 1

1.1 Motivation 1

1.2 Approach 3

1.3 Contributions 5

1.4 Outline 6

1.5 Software and Community 7

1.6 Attribution 7

CHAPTER 2 : Background 9

2.1 History 9

2.2 Shell characteristics 10

2.3 Shell limitations 11

2.4 Challenges 12

2.5 Applications 14

2.6 Program development 14

CHAPTER 3 : Related work 16

3.1 Ecosystem support and tooling for the shell 16

3.2 Dataflow graph models 20

3.3 Language parallelization support 22

3.4 Distributed systems 23

3.5 Just-in-time and staged compilation . . . 24

CHAPTER 4 : A formal model of a data processing fragment of the shell . . . 25

4.1 Introduction . . . 25

4.2 Background . . . 27

4.3 Example and overview . . . 28

4.4 An order-aware dataflow model . . . 34

4.5 Parallelization transformations . . . 41

4.6 Related work . . . 48

4.7 Discussion . . . 50

CHAPTER 5 : Specification framework . . . 52

5.1 Introduction . . . 52

5.2 Parallelizability of Standard Libraries . . . 54

5.3 Extensibility Framework . . . 56

CHAPTER 6 : PaSh: Automatic parallelization of shell dataflow regions . . . 62

6.1 Introduction . . . 62

6.2 Background and overview . . . 64

6.3 Dataflow graph model . . . 66

6.4 Runtime . . . 72

6.5 Evaluation . . . 75

CHAPTER 7 : PaSh-JIT: Just-in-time automatic parallelization of complete shell programs . . . 83

7.1 Introduction . . . 83

7.2 Example and overview . . . 85

7.3 Interfacing with the shell . . . 89

7.4 The JIT engine . . . 92

7.5 Parallelizing compilation server . . . 94

7.6 Commutativity awareness . . . 99

7.7 Evaluation 102
7.8 Related work 108
7.9 Discussion 110

CHAPTER 8 : DiSh: Scaling out shell programs on a distributed cluster 112
8.1 Introduction 112
8.2 Background, Example, and Overview 115
8.3 Dynamic Shell Orchestrator 120
8.4 Compiler 122
8.5 Distributed Scheduling 126
8.6 Runtime Support 128
8.7 Evaluation 130
8.8 Related Work 136
8.9 Discussion 138

CHAPTER 9 : Out-of-order speculative execution for the shell 140
9.1 Introduction 140
9.2 System overview 144
9.3 Discussion 148
9.4 Related Work 150

CHAPTER 10 : Conclusion 151
10.1 Future Work 151
10.2 Outlook 151

BIBLIOGRAPHY 153

LIST OF TABLES

TABLE 5.1 Parallelizability Classes. Broadly, UNIX commands can be grouped into four classes according to their parallelizability properties. 55

TABLE 6.1 Summary of UNIX one-liners. Structure summarizes the different classes of commands used in the script. Input and seq. time report on the input size fed to the script and the timing of its sequential execution. Nodes and compile time report on PASH's resulting DFG size (which is equal to the number of resulting processes and includes aggregators, `eager`, and `split` nodes) and compilation time for a `--width` value of 16. 75

TABLE 7.1 Benchmark summary. Summary of all the benchmarks used to evaluate PASH-JIT and their characteristics. 101

TABLE 7.2 Correctness results. Running the POSIX test suite on Bash and PASH-JIT. Tests are grouped in rows by theme. Columns contain the group name, total tests, non-applicable tests, and passing tests for PASH-JIT and Bash. 103

TABLE 8.1 Available options for scaling out shell programs. Compatibility: support unmodified shell scripts. Granularity: support fine-grained distribution. Expressiveness: support arbitrary dynamic behaviors. Agnosticism: support components in any programming language. Equivalence: behavior equivalence with existing shells. 113

TABLE 8.2 Benchmark summary. Summary of all the benchmarks used to evaluate DISH, and their characteristics. 131

TABLE 8.3 DISH performance in 20-node cloud deployment. DISH speedup over Hadoop Streaming for scripts that AHS supports. 133

LIST OF ILLUSTRATIONS

FIGURE 1.1 A shell program that downloads a compressed archive of text files (books from Project Gutenberg), extracts them in a directory, and then performs an analysis to find the frequencies of all words of a specific form. 2

FIGURE 1.2 Overview of contributions in this dissertation. Each block corresponds to a contribution. 5

FIGURE 2.1 Shell programs are developed iteratively, with a *debug cycle* (nodes (1), (2), and (3); in gray) and *development cycle* (nodes (1), (2), (3), and (4)). Improving program performance speeds up the bolded transition from node (2) to node (3). 15

FIGURE 4.1 Dataflow Description Language (DDL). A language used to describe dataflow programs that consume a set of inputs and produce a set of outputs using a graph of computation nodes. 35

FIGURE 4.2 Small Step Execution Semantics for DDL. A single step represents the computation of a dataflow node after it has consumed a new input message. 37

FIGURE 4.3 Auxiliary transformations applied by the compiler on dataflow programs to enable the parallelization transformation. 43

FIGURE 4.4 Visualization of auxiliary transformations applied by the compiler on dataflow programs. 44

FIGURE 4.5 Parallelization transformation applied by the compiler on the dataflow program to expose available data parallelism. 46

FIGURE 4.6 Visualization of parallelization transformation applied by the compiler on dataflow programs. 46

FIGURE 5.1 Calculating maximum temperatures per year. The script downloads daily temperatures recorded across the U.S. for the years 2015–2019 and extracts the maximum for every year. 53

FIGURE 6.1 Output of `pash -width=2` for Fig. 5.1 (fragment). PASH orchestrates the parallel execution through named pipes, parallel operators, and custom runtime primitives—*e.g.*, `eager`, `split`, and `get-pids`. 65

FIGURE 6.2 From a script AST to DFGs. The AST on the left has two dataflow regions, each not extending beyond `&&` (*Cf.*§6.3.1). Identifiers `f1`, `f2`, and `f3` sit at the boundary of the DFG. 67

FIGURE 6.3 Stateless parallelization transformation. The `cat` node is commuted with the stateless node to exploit available data parallelism. 69

FIGURE 6.4 Auxiliary transformations. These augment the DFG with `cat`, `split`, and `relay` nodes. 71

FIGURE 6.5 Eager primitive. Addressing intermediary laziness is challenging: (a) FIFOs are blocking; (b) files alone introduce race conditions between producer/consumer; (c) files + `wait` inhibit task parallelism. Eager `relay` nodes (d) address the challenge while remaining within the PASH model. 73

FIGURE 6.6 Runtime setup lattice. Parallel No Eager and Blocking Eager improve over sequential, but are not directly comparable. PASH w/o Split adds PASH's optimized `eager` relay, and PASH uses all primitives in §6.4 (Fig. 6.7). 74
FIGURE 6.7 PASH's speedup for `width=2–64`. Different configurations per benchmark: (i) PaSh: the complete implementation with `eager` and `split` enabled, (ii) PaSh w/o split: `eager` enabled (no `split`), (iii) Blocking Eager: only blocking `eager` enabled (no `split`), (iv) No Eager: both `eager` and `split` disabled. For some pairs of configurations, PASH produces identical parallel scripts and thus only one is shown. 76
FIGURE 6.8 Unix50 scripts. Speedup (left axis) over sequential execution (right axis) for Unix50 scripts. Parallelism is 16× on 10GB of input data (*Cf.*§6.5.2). Pipelines are sorted in descending speedup order. 79

FIGURE 7.1 PASH-JIT overview. PASH-JIT instruments scripts with calls to the JIT engine, which passes program fragments to the compilation server at run-time. 85
FIGURE 7.2 JIT engine overview. The different stages of the engine's execution. 92
FIGURE 7.3 Compilation server algorithm (pseudocode) extended for dependency untangling. . . 97
FIGURE 7.4 JIT engine algorithm (pseudocode) extended for dependency untangling. 98
FIGURE 7.5 Overview of commutativity-aware transformations. 100
FIGURE 7.6 PASH-JIT Performance. PASH-JIT speedup (*vs.* PASH whenever possible) over Bash for Tab. 7.1 rows 2–5 (left, box) and 6–13 (right, bar) (*Cf.*§7.7.2). 104
FIGURE 7.7 PASH-JIT Dynamic Optimizations. PASH-JIT speedup over Bash when toggling profile-driven compiler configuration and dependency untangling for Tab. 7.1 row 5 (left, box) and 6, 8–13 (right, bar) (*Cf.*§7.7.3). 107
FIGURE 7.8 PASH-JIT Commutativity Awareness. PASH-JIT speedup over Bash when toggling commutativity awareness for Tab. 7.1 rows 2–4 (left, box) and 6, 7 (right, bar) (*Cf.*§7.7.3). 108

FIGURE 8.1 Example script: Downloading a temperature dataset, storing on a distributed file system, and running analysis to extract statistics. 117
FIGURE 8.2 DISH architecture overview. Steps: (a) compile script region; (b) schedule compiled dataflow; (c) send dataflow subgraphs to workers; (d) compilation failed, fall back to original region; and (e) execute script region (compiled or original). 118
FIGURE 8.3 DISH dataflow graph stages. (a) HDFS files are expanded to sequences of blocks. (b) the graph is parallelized based on the command specifications. (c) the scheduler splits the graph and assigns subgraphs to workers. 119
FIGURE 8.4 Dynamic orchestration overview. DISH instruments scripts with calls to the orchestration engine, which passes program fragments to the worker manager at run-time. 122
FIGURE 8.5 Example of independent regions. This shell script compresses all files in a directory—but each iteration results in an independent body region that can be executed in parallel. 125
FIGURE 8.6 Overview of commutativity-aware transformations. (Top) Remote writes and reads added during distributed scheduling. (Mid) Worker-first aggregation. (Bot) Named FIFO teleportation. 126

FIGURE 8.7 DISH performance on a 4-node cluster. DISH speedup (*vs.* PASH and Hadoop Streaming whenever possible) over Bash for Tab. 8.2 rows 1–4 (left, box) and 5–8 (right, bar) (*Cf.*§8.7.1). (Log y-axis; higher is better.) . . . 131
FIGURE 8.8 Dynamic dependency untangling. DISH speedup over Bash when toggling DDU (higher is better). . . . 135

FIGURE 9.1 A bioinformatics script slightly adapted from Köster and Rahmann [101] that maps sequence reads to a reference genome. . . . 141
FIGURE 9.2 A high-level overview of `hs`, a speculative out-of-order shell-script executor. The preprocessor and runtime hooks extend PASH-JIT (Chapter 7) and tracing extends Riker [45]. . . . 142
FIGURE 9.3 Step-by-step scheduling and orchestration of Köster and Rahmann's [101] script, simplified (Fig. 9.1). . . . 143
FIGURE 9.4 Transition system for command state in the scheduling algorithm. . . . 146

CHAPTER 1

Introduction

Unix and Linux shell programs are a ubiquitous part of software infrastructure. According to a recent longitudinal study [64] the shell was the 8th most popular language on Github in 2022. Shell programs are widely used by all types of developers, administrators, and scientists, for all kinds of tasks, from data processing to system orchestration. Crucially, shell programs are often used as system "glue", composing different computational components (commands) to perform complex tasks very succinctly. The shell is particularly well-suited for this task compared to any other programming language due to a unique combination of features: (1) it offers language-agnostic primitives that enable seamless command composition; and (2) it offers dynamic primitives, such as command substitution, that enable interactivity based on the current shell and file system state. Due to these features, shell programs are used for a variety of performance critical tasks, including continuous integration and deployment, bioinformatics workloads, and data processing pipelines. In this dissertation I propose novel techniques and build systems to optimize the performance of such critical shell programs, allowing them to harness multicore and distributed computational resources without affecting their behavior or requiring additional effort from developers.

1.1. Motivation

Despite the shell's prevalence and long history (the first shell was developed by Ken Thompson in 1971), it lags behind other programming environments with respect to automated compiler optimizations, significantly impacting applications. Missing optimizations include (1) parallelization for scaling out on multicore CPUs, (2) distribution for processing data that does not fit on a single machine, (3) incremental execution to reduce redundant work when reexecuting a program, and (4) out-of-order speculative execution for better utilizing underlying computational resources. The lack of support for such optimizations forces developers to either reimplement their complete shell programs and their commands in different languages to optimize their performance, requiring significant effort and often jeopardizing the programs' behavior; or else accept that their performance cannot be improved, hindering downstream tasks. As an example, continuous integration and deployment tasks, *e.g.*, software builds, for big applications and organizations can take on the order of

```
1  IN=${IN:-$TOP/pg}
2  mkdir "$IN"
3  cd "$IN"
4  echo "Download will take some time, be patient..."
5  wget "$SOURCE/data/pg.tar.xz"
6  if [ $? -ne 0 ]; then
7    echo "Download failed!"
8    exit 1
9  fi
10 cat pg.tar.xz | tar -xJ
11
12 cd "$TOP"
13 OUT=${OUT:-$TOP/output}
14 mkdir -p "$OUT"
15 for input in $(ls "$IN"); do
16   cat "$IN/$input" | tr -sc '[A-Z][a-z]' '[\012*]' |
17   grep '^....$' | sort | uniq -c > "$OUT/$input.out"
18 done
```

Fig. 1.1: A shell program that downloads a compressed archive of text files (books from Project Gutenberg), extracts them in a directory, and then performs an analysis to find the frequencies of all words of a specific form.

many hours to days, making it necessary that they run at nights or during weekends instead of on-demand, when a software component is updated.

The main reason why the shell lacks compiler support for optimizations is because there is no principled way to analyze and transform shell programs in a sound and precise manner. This is due to three fundamental challenges related to the shell: (1) shell programs compose arbitrary black-box software components (commands) that are developed in multiple programming languages and cannot be analyzed in a unified way; (2) the language of the shell offers primitives that are highly dynamic, making static analysis intractable; and (3) the shell specification is complex and different implementations vary significantly, making it extremely hard for optimizations to preserve behavior.

Figure 1.1 contains a program that showcases these challenges. This program computes the frequencies of specific word patterns in a collection of books. Before explaining what it does in detail, it is useful to observe that shell programs are very similar to programs in other imperative languages—they have variables, control flow, *etc.*—with one key difference; they delegate complex parts of the computation to external binaries, also called commands or utilities. The commands in this program include `mkdir`, `wget`, `gunzip`, and `ls`. Each

command can be configured using its arguments, for example, `mkdir -p "$OUT"`, which is a command that creates a directory, is invoked with arguments `-p` and `"$OUT"`, configuring it (1) to not exit with an error if the directory already exists, and (2) to set the name of the created directory according to the value of the variable `OUT`.

The script can be split into two independent parts: data downloading and preparation (lines 1-10) and processing (12-18). It first sets the value of variable `IN` if it was not already set (line 1), and then creates a directory with that name and sets it to be the current working directory (lines 2 and 3). After printing a message (line 4) it downloads a compressed archive of a book collection (line 5). It then ensures that the download completed successfully (line 6) by checking the value of variable `?`, a special shell variable that contains the exit code of the last executed command. If the download failed it prints an informative message and exits (lines 7 and 8). If the download succeeded, it extracts the data in the archive using the command `tar` (line 10). The pipe operator `|` is used to compose commands in the shell, connecting the standard output of the first to the standard input of the second.

Before processing, the script changes to a new directory, sets the value of `OUT`, and creates a directory for the script outputs (lines 12-14). Then it loops over the books that were extracted from the downloaded archive: `$(ls "$IN")` captures the standard output of the `ls` invocation and uses it to determine the loop iterations (line 15). The main processing pipeline then splits its book into words (using `tr`) and then filters words with 4 characters (using `grep`) and computes their frequencies (using `sort` and `tr`).

This shell script showcases the first two of the aforementioned challenges. First, it composes arbitrary black box commands to achieve its task, *e.g.*, `mkdir`, `tar`, `tr`, `sort`. Second, the exact iterations that the loop will execute depend on the state of the file system, a component of the state that cannot be determined ahead-of-time. The combination of these two challenges together with the complexity of the shell semantics makes it very hard to develop a solution that automatically optimizes the performance of this and other scripts.

1.2. Approach

In this dissertation, I propose a novel compilation architecture that addresses the three aforementioned challenges. It has three main components.

Specification framework: A lightweight command specification framework decouples the task of specifying external command behavior from the task of developing analyses and optimizations for shell scripts. The first task can be delegated to experts or achieved through crowdsourcing, and these specifications can then be harnessed by tools that analyze and optimize shell scripts composing such commands. As an example, the specification for `tr` and `grep` captures the fact that their invocations in Figure 1.1 only read from their standard input, only write to their standard output, and process each of their input lines independently. This allows a parallelization system to shard them into parallel invocations, each of which processes a different partition of their input.

Just-in-time architecture: A just-in-time architecture addresses the challenge of the shell's dynamism by applying optimizations at runtime, after dynamic information has been resolved. The architecture exploits the fact that the shell's execution is bimodal: most of the computation is done by external commands, and gluing (control flow and command preparation) is done in the language of the shell. The just-in-time architecture executes all the gluing in the shell and performs optimizations right before the invocation of external commands. In the example above, the optimization of the loop body would happen after the execution of `$(ls "$IN")`, having access to the values of all loop iterations and the file system state at this point in time.

Shell-to-shell compiler: A shell-to-shell compiler shim generates optimized shell programs that can be executed by the original shell interpreter to maintain compliance. This is in contrast to having a new shell implementation, which would have to exhaustively implement all edge cases of an existing shell's behavior to match its behavior. Using this shim architecture, I can focus on a fragment of the shell, optimizing it carefully and leaving all the rest of the complex edge cases and behavior to be covered by an existing interpreter.

To support this architecture, I develop a formal model of a data processing fragment of the shell that is used to prove the correctness of several optimizing transformations. At the same time, I develop a concrete instantiation of the architecture in a open-source library of high-performance components, called the PASH project. I use this library as infrastructure to build two systems, PASH and DISH, that target two concrete optimizations: automatic parallelization and distribution. I demonstrate the benefits of these optimizations on a wide variety of real-world programs. The evaluation shows that PASH and DISH achieve significant speedups over the state of the art without jeopardizing compliance with the underlying system shell. Finally,

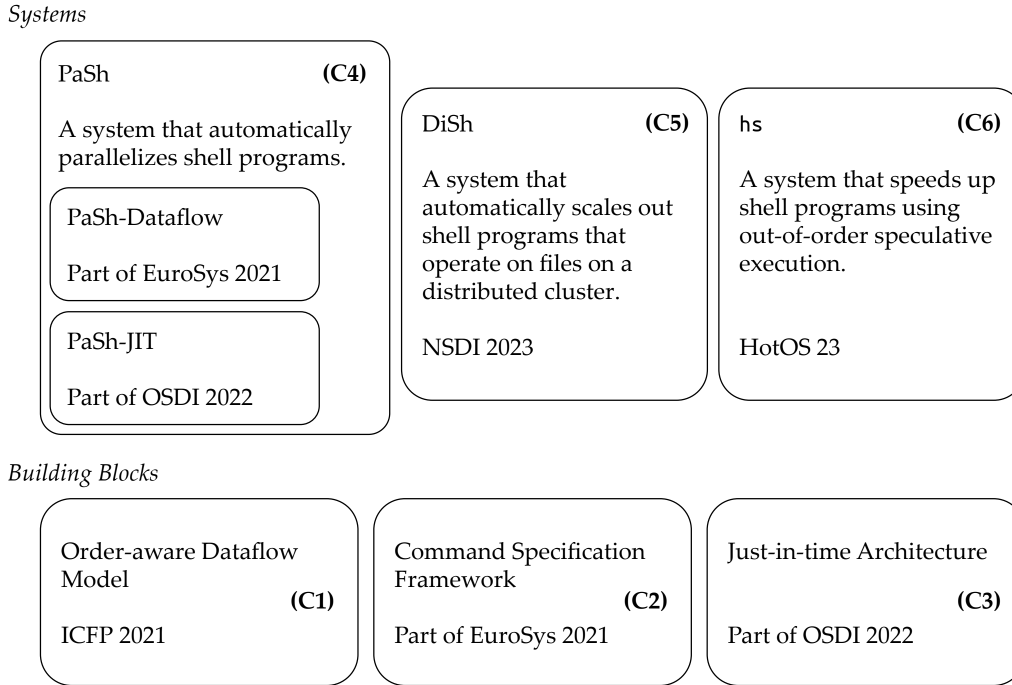


Fig. 1.2: Overview of contributions in this dissertation. Each block corresponds to a contribution.

I develop an extension of the architecture, named `hs`, that can lift the requirement for command specifications while also enabling a new optimization: out-of-order command execution through speculation. This extension builds on two primitives: tracing (to know how a command tries to effect the system) and containment (to isolate its effects and selectively apply them to its environment). Tracing and containment completely lift the requirement for command specifications since a command can simply be executed and its effects discovered at runtime. These two components enable the development of a command scheduler that speculatively executes a sequence of commands out-of-order, improving utilization of the underlying system; similarly to the aformentioned compiler architecture, this optimization can be applied to arbitrary shell programs.

1.3. Contributions

The contributions of this dissertation proposal can be split into two categories, *building blocks* (**C1-3**) that enable a compilation infrastructure for shell programs, and *systems* (**C4-6**) that compose the building blocks and automatically optimize shell programs for real-world use cases. They are displayed visually in Figure 1.2 and summarized below:

C1 A formal model of the dataflow fragment of the shell. This model is order-aware, properly capturing the order in which commands read from their inputs. It is used to develop a shell-to-shell optimizing compiler that is accompanied by proofs that the transformations it performs are correct with respect to this model.

C2 A command specification framework that allows describing aspects of a command's behavior and can be used by others systems to analyze and transform shell programs. The command specification framework is guided by a study on all POSIX and GNU Coreutils commands and is implemented in an open-source library.

C3 A just-in-time compiler architecture that invokes a shell optimizing compiler at runtime after having resolved dynamic information about the state of the system. This enables optimizing highly dynamic shell programs in a sound and effective fashion.

C4 PASH, a concrete instantiation of the compilation architecture in a system that automatically parallelizes arbitrary shell programs. This system combines the building blocks and introduces additional optimizations to achieve speedups for a wide variety of shell programs.

C5 DISH, a system that automatically scales out shell programs that operate on files that reside over a distributed cluster. DISH achieves speedups on a wide range of programs through colocation and parallelization.

C6 A system prototype called `hs` that enables out-of-order command execution through speculation.

1.4. Outline

Chapter 2 provides necessary background on the UNIX shell, its history, benefits, limitations, and target applications. Chapter 3 describes related work and puts my work in the context of the existing literature. Chapter 4 describes contribution **C1** and is based on my ICFP 2021 paper [76]. Chapter 5 describes contribution **C2** and is based on material from my EuroSys 2021 paper [173]. Chapter 6 describes the PASH-Dataflow component of contribution **C4** and is based on material from my EuroSys 2021 paper [173]. Chapter 7 describes contribution **C3** and PASH-JIT, the second part of contribution **C4** and is based on material from my OSDI

2022 paper [95]. Chapter 8 describes contribution **C5** and is based on my NSDI 2023 paper [131]. Chapter 9 describes contribution **C6** and is based on material from my HotOS 2023 paper [110]. Finally, Chapter 10 concludes by describing some future directions and the bigger conclusions and impact of the work described in this dissertation.

1.5. Software and Community

The work described in this dissertation has led to the creation of an open-source library of components called the PASH project [15], which has now grown to an independent project with a much larger scope, *i.e.*, improving the shell ecosystem in a variety of ways, focusing on performance, correctness, and usability among others. It currently contains more than 10 repositories, including tools, libraries, and systems. The development of the components in the PASH project is done by more than 30 contributors in different institutions in academia and industry, and has gathered significant attention from the community, totalling more than 5000 Github stars across all repositories. In 2021, the PASH project started being hosted by the Linux Foundation[1], an organization which hosts a wide variety of open-source projects, including Kubernetes [7] and eBPF [14]. The technical steering committee of the PASH project includes Michael Greenberg, Tammam Mustafa, Nikos Vasilakis, and myself. The two core systems described in this dissertation, PASH[2] and DISH[3], are also available under the PASH project.

1.6. Attribution

Most of the research presented in this dissertation came out of the PASH project, a collaborative project that started by me and Nikos Vasilakis in 2019. In addition to me and Nikos, several others have contributed to the project and the work presented here: particularly Michael Greenberg (for Chapters 7 and 9); Tammam Mustafa (for Chapters 7 and 8); Georgios Liargkovas (for Chapter 9); Konstantinos Mamouras, Achilles Benetopoulos, and Lazar Cvetković (for Chapters 5 and 6); Shivam Handa and Martin Rinard (for Chapter 4); Jan Bielak, Dimitris Karnikis, and Thurston H.Y. Dang (for Chapter 7); and Pratyush Das (for Chapter 8). I wrote all the included material in Chapters 1 and 10 and most of Chapters 2 and 3. Chapters 2 to 9 integrate text from several papers [72, 76, 173, 95, 131, 110] with all the collaborators mentioned above. It is not

[1]https://www.linuxfoundation.org/press/press-release/linux-foundation-to-host-the-pash-project-accelerating-shell-scripting-with-automated-parallelization-for-industrial-use-cases

[2]https://github.com/binpash/pash

[3]https://github.com/binpash/dish

possible to precisely delineate each collaborator's contribution in each of these papers since many ideas and developments became possible through discussions, brainstorming, and pair-programming sessions, but I try to attribute the main contributions of each paper below. For the ICFP 2021 paper [76], Shivam Handa, Nikos Vasilakis, and I developed the formalization of the order-aware dataflow model, and I was the lead developer of the project's implementation. For the EuroSys 2021 paper [173], Nikos Vasilakis and I were the primary authors and we contributed equally to all parts of the work; the rest of the coauthors contributed with the development of some command specifications and runtime components, as well as with the system evaluation. For the OSDI 2022 paper [95], I was the lead developer of the system and its just-in-time architecture; the rest of the coauthors helped with the development of the commutativity-aware optimizations and the parsing library, as well as with the system evaluation. For the NSDI 2023 paper [131], Tammam Mustafa, a master's student at the time, was the first author and the lead developer of the DiSh system; Nikos Vasilakis (Tammam's supervisor) and I worked extensively with Tammam to develop the ideas and the system behind this paper. Finally, for the HotOS 2023 paper [110], Georgios Liargkovas, an undergraduate student at the time, was the first author and the lead developer of the `hs` system prototype; all of the authors contributed to coming up with the idea of speculative execution for shell scripts, and I worked closely with Georgios on the technical development of the system prototype and its evaluation.

CHAPTER 2

Background

In this section I provide some background on the UNIX shell, a brief history, as well as a description of its benefits and drawbacks. First of all, we need to make clear what we mean by the UNIX shell, since no single such thing exists; in fact, what is most commonly refered to as the shell is a combination of (1) a programming language with a set of "shell-like" characteristics, (2) an interactive interpreter that is usually accessed through a terminal, and (3) a "standard library" of commands that can be used to perform more complex data-processing and other tasks. So what is the UNIX shell?

2.1. History

There are two perspectives when defining the shell: the formal, which puts the specification first, and the pragmatic, which focuses on the different shell implementations. In this dissertation I focus on the core of the shell, namely the common characteristics that are both described by the specification and implemented by most shells. First, there exists a POSIX shell specification [73] that attempts to precisely define the shell language and its semantics using natural language. At the same time, there exist multiple shell implementations, most of which diverge from POSIX significantly (on purpose or accidentally). The first UNIX shell implementation was the Thompson shell, introduced in the first version of UNIX in 1971, which was simple but introduced several important features that we now consider synonymous with the shell such as input/output redirection and pipes. These features enable the composition of third-party commands to perform complex tasks, making the external commands a prominent part of the experience of writing shell programs, and go hand in hand with the Unix philosophy (first documented by Doug McIlroy [118]) that advocates modular and compositional commands (with simple interfaces and handling text streams as inputs and outputs). The name "shell" was chosen to separate it from the operating system *kernel*. The kernel is the core of the system and the shell is a user facing program that can access and manage the system by communicating with the kernel. Since then, there have been many shell implementations: bash [145], dash [3], zsh [13], OSH [9], ksh [6], mksh [8], and yash [12] to name a few. Different implementations vary significantly with respect to their popularity, and also with respect to their semantics and language features. Most UNIX systems provide

access to a shell (usually bash or zsh) as the default way to interact with them and configure them.

Due to its ubiquity and power, the shell has been used for a variety of tasks: system configuration, application building and deployment, as well as complex data processing tasks. In the contemporary world, while in principle automated systems replace many shell tasks by taking on various configuration and management jobs, in practice, shell programs still show up everywhere: Docker [123], Vagrant [77], Kubernetes [7], and other cloud deployments are all managed by shell programs. Furthermore attempting to develop a new shell-like language for specific domains leads to different such systems behaving slightly differently from the underlying shell, creating significant confusion to developers, *e.g.*, systemd [10] uses its own variable expansion which is slightly different from the shell's. In their core, all these systems try to provide restricted APIs on top of the shell to improve programmability, but users often end up needing the whole range of the capabilities of the shell.

2.2. Shell characteristics

So what are the characteristics that make the shell the shell?

- **C1 External commands are first class citizens:** The shell supports easy management of external command execution through job control, *e.g.*, sending a command to the background, and input/output redirection.

- **C2 File system is a first class citizen:** The shell supports straightforward access, introspection, and modification of files in the file system.

- **C3 Universal composition:** The shell supports easy composition of external commands using intermediate files or write-once, read-once pipes.

- **C4 Streaming support:** The shell supports stream processing, *i.e.*, incrementally reading input and producing output (both by its available commands and primitives).

The above characteristics make the shell powerful and widely popular for many tasks. Looking back at the program in Figure 1.1, it manages to perform a complex task in a succinct way by leveraging characteristics C1-C3: it uses external commands, such as `wget` and `sort`, to delegate complex functionality, it accesses

and modifies the file system with commands such as `ls` and `tar`, and it composes commands in pipelines to implement the frequency computation using simpler components. Streaming support (C4) also allows the program to be efficient: commands like `grep` in line 16 don't need to block until all of their input is available but can start processing as soon as chunk of the input is produced by `tr`.

For the rest of this dissertation, whenever I refer to the shell, I focus on the set of programming environments (language and libraries) that have the aforementioned characteristics. My research should be applicable to any programming environment that satisfies these characteristics, *e.g.*, scripts in Python that are used to compose external command invocations.

2.3. Shell limitations

While the shell is popular and powerful, it also has significant limitations with respect to performance and correctness: shell programs cannot easily utilize underlying multiprocessor or multinode resources, and they are very hard to debug and test. These limitations (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 [61]), but (iii) are not *essential* to its existence.

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 and their dire consequences. The shell's syntax and its direct access to the user's entire system, both aimed at terse interactive use, lead to a fast-paced high-stakes programming experience where a single typo could erase entire hard drives. Protective mechanisms such as assertions and error handling that are commonly used in critical code written in other languages are not well supported in the shell.

Performance doesn't scale While shell programs have acceptable performance in a single-core setting, they're not tuned for multicore machines and clusters of nodes. Unlike other programming languages, the performance of shell programs is dominated by the performance of the commands that they compose, and unfortunately, most shell commands do not scale. To address this, users turn to restricted parallelism orchestration tools [164, 84, 63, 184] or even worse, replace parts of their programs with programs in parallel frameworks, an error-prone process that requires significant effort.

Redundant recomputation Small changes to the input of a program require a complete re-execution, leading to many hours of wasted redundant computation. This is common in data processing (and preprocessing) workloads, as well as in build, configuration, and setup programs. Domain-specific solutions (such as build systems, e.g., `make`) address this issue for their use cases, but do not generalize or compose.

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, where configuration programs, program and library paths, and environment variables are configured *ad hoc*. The composition primitives do not compose at scale.

In this dissertation, I focus on addressing the scalability limitation, as well as taking some steps towards the support for contemporary deployments.

2.4. Challenges

The aforementioned limitations have been plaguing the shell for many years; why have they not been addressed by subsequent research or newer shell implementations that try to improve on existing ones? I conjecture that this is because of three challenges inherent to the shell that make it very hard to address its limitations in a principled and general manner; some of the challenges depend on its fundamental characteristics, and some are unfortunate results of its history. In particular there are three distinct challenges with addressing the shell's shortcomings that stem from its fundamental characteristics: (1) shell programs compose arbitrary black-box commands that are developed in multiple programming languages and cannot be analyzed in a unified way; (2) the language of the shell offers primitives that are highly dynamic, making static analysis intractable; and (3) the shell specification is complex and different implementations vary significantly, making it extremely hard for optimizations to achieve compliance with existing shells, jeopardizing backwards compatibility.

Composition of arbitrary black-box commands The shell's virtue of limitless composition and having external black-box commands as first class citizens is also its vice: how does one analyze and transform

a shell program if there is no precise model of all the commands that it uses? Any command invoked by a shell program may translate to an `execve` of an arbitrary executable with unknown behavior. 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. This acts as a very high barrier of entry for research and development of tools and analyses for the shell since researchers have to either make a huge effort to hardcode the semantics of each command for their tools to work, or risk the tool not being used widely.

Dynamic language primitives Shell programs are very succinct partly because of the existence of global state, file system and environment variables, that can be accessed and modified at runtime. Accessing the file system state at runtime can be considered as external nondeterministic input for the purposes of analysis, and while this exists in other languages too, *e.g.*, using `eval` in Javascript, shell scripts use these features pervasively to change control flow and program execution. This means that the behavior of a shell program cannot be known statically: an invocation to `grep $X $DIR/*.c` depends primarily on dynamically computed values, including the state of the file system, the current directory, environment variables, and unexpanded strings. This disallows any form of static analysis or transformation for improving the performance of shell programs; a static analysis would either have to be unsound, assuming that the state before execution will be the same as the state at runtime, or ineffective, conservatively assuming the worst for every part of the state that can be modified at runtime.

Backwards compatibility The shell has very complex semantics and multiple diverging implementations, making it very hard to achieve backwards compatibility for new shells and tools that address some of its limitations. The semantics of the shell and common commands are documented in 300pp of standardese [22]. To be able to reason about a program'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 [34, 3] is used for startup programs, while bash [145] is a common interactive choice. Every shell extends POSIX in its own way [78, 71]. The lack of an easy-to-use correctness baseline and the inability to formally reason about the shell's semantics inhibits research on the shell. On the other hand, developing new shells with cleaner semantics is likely to fail without any consideration for backward compatibility. This

challenge does not stem from inherent shell characteristics like the other two, but rather from the current state of the world and past design decisions. Nevertheless, it is not possible to design solutions that address the shell's limitations without providing backwards compatibility to enable adoption.

2.5. Applications

Due to the shell's general nature and the fact that it composes arbitrary external commands, it is used for a wide variety of tasks including software builds, program and text analysis, system log analysis, filesystem analysis, git history analysis, media conversion, natural language processing, web crawling, bioinformatics, scientific computation, software distribution packaging, file encryption and compression, ML workflows, container image build, distributed deployment orchestration, boot sequences, system administration, software testing, and continuous integration workflows. Some of these tasks have been relevant for as long as the shell exists, in fact the shell was created to perform them, while others correspond to contemporary demands. The tasks also vary with respect to how interactive they are; the shell's primitives work with both interactive and non-interactive workloads, from brief 'one-liners' to get a quick job done to system and build programs that consist of thousands lines of code.

2.6. Program development

The shell's interactive nature leads to a program development process that is highly iterative, gradually transitioning to less interactive while the program converges and stabilizes: the programmer gradually explores the data, refining the program in the process, slowly generalizing it to support the complete dataset and satisfy its specification. The shell features facilitate such an exploratory and incremental approach—external commands can be patched together succinctly to implement complex functionality and their effects can be observed by inspecting the file system and performing quick analyses with other commands.

Fig. 2.1 illustrates the development process and contains two iterative cycles here: the smaller cycle (nodes (1), (2), and (3)), is the *debugging cycle*, while the bigger cycle (nodes (1), (2), (3), and (4)) is the *development cycle*. Initial iterations of the development cycle work with a manageable subset of the dataset, as the program grows in capability so will the dataset it's applied to. So iterating starts out fast—doing simple analyses on small amounts of data—and gradually expands to a bigger dataset and more complex programs. Especially on large data sets, the slowest part of the process is rerunning the program (node (2) to node (3)); the bigger

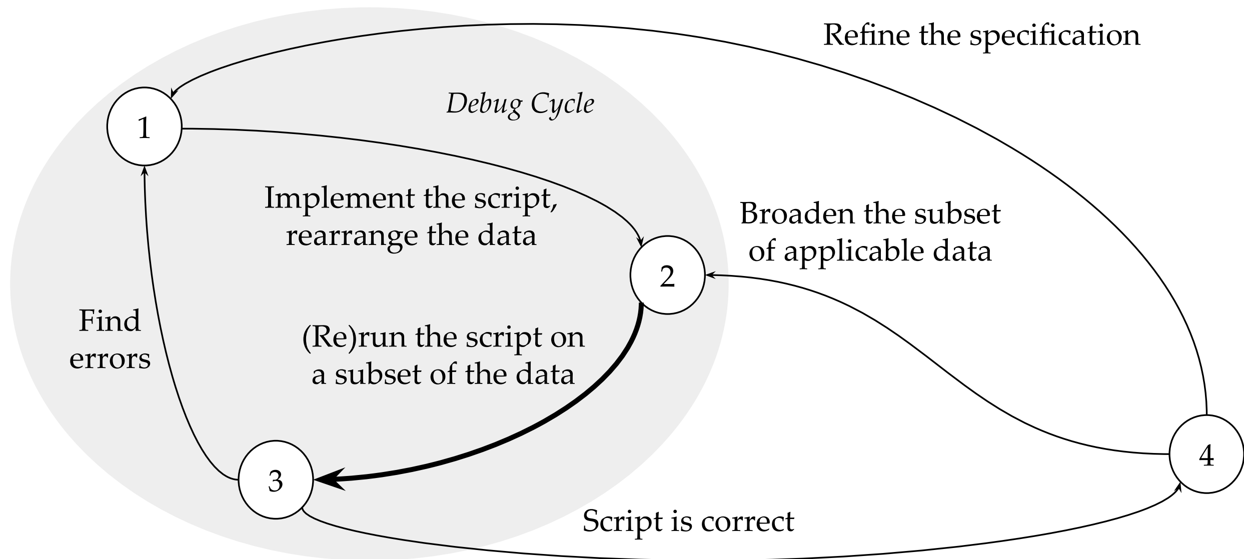


Fig. 2.1: Shell programs are developed iteratively, with a *debug cycle* (nodes (1), (2), and (3); in gray) and *development cycle* (nodes (1), (2), (3), and (4)). Improving program performance speeds up the bolded transition from node (2) to node (3).

the program, the longer it runs—and the slower it is to discover errors.

It is particularly frustrating—and all too common—to run a program for minutes, hours, or even days, to discover that the output is garbage. The wasted work is frustrating and becomes a bottleneck as the data and program complexity grows. This is where addressing the shell's performance limitations can have the biggest impact; optimizing a program to better utilize underlying parallel and distributed computational resources can significantly reduce execution time, improving the development workflow.

CHAPTER 3

Related work

My dissertation has several axes of related work, from tooling specifically designed for the shell to just-in-time compilation.

3.1. Ecosystem support and tooling for the shell

As described in Chapter 2, the shell has significant limitations and shortcomings, a fact that is widely accepted in the community. To address these there has been a big body of prior work that has developed new shell implementations and new abstractions, such as new programming languages for the shell, either replacing or building on top of existing ones. In contrast to most prior work, my research does not attempt to replace existing abstractions, instead aiming to be completely backwards compatible, requiring no modifications to user programs or existing systems. The main reason that I have followed this approach is to enable adoption. It is very hard for someone to replace all their legacy shell code in a new language to get performance or other types of benefits, because the cost (development effort) and risk (bugs) is hard to justify by the potential benefits. Therefore, all of the work described in this dissertation aims for complete backwards compatibility.

This section provides an overview of prior work on the shell, from work trying to achieve performance benefits through parallelism and distribution, to build systems that focus on improving the performance of a specific class of workloads, to work that tries to provide stronger correctness guarantees for shell scripts.

Parallel and distributed shells and tools Several packages expose commands for specifying parallelism and distribution on modern UNIXes—*e.g.*, `qsub` [63], SLURM [184], and GNU `parallel` [164]. These packages allow multiplexing and parallelizing completely independent sets of workloads on a multiprocessor machine or a distributed cluster. The effectiveness of these tools is predicated upon explicit and careful invocation and is limited to embarrassingly parallel (and short) programs. Often, these packages provide options to support an array of special sub-cases—a stark contradiction to the celebrated UNIX philosophy. For example, `parallel` contains flags such as `--skip-first-line`, `-trim`, and `--xargs`, that a UNIX user can achieve using `head`, `sed`, and `xargs`; it also includes other programs with complex semantics, such as the ability to transfer files between computers, separate text files, and parse CSV. Finding the right

combination of flags to achieve performance benefits without affecting the sequential behavior of the program is challenging and puts significant burden on developers. Several shells, such as rc [50], gsh [117], and dgsh [159] introduce primitives for non-linear pipe topologies—some of which target distribution. These shells improve the expressiveness of traditional shells by allowing the creation of shell programs that go beyond linear pipelines, describing streaming directed acyclic graphs that can be more naturally parallelized, *i.e.*, two nodes in the graph that do not depend on each other can be executed completely independently. Here too, however, developers are expected to manually rewrite programs to exploit these new primitives. My work aims to automatically optimize existing shell programs by exploiting available parallelization and distribution potential without any requirements on the program developer.

POSH [144] is a recent shell for programs operating on NFS-stored data. POSH focuses on shell pipelines that process input data that is partitioned on a cluster of nodes. By splitting up the pipeline and running parts of it as close to the data as possible, it reduces network communication and overhead and achieves significant performance benefits. POSH was developed concurrently with the work described in this dissertation and also came up with a notion of command specifications to optimize scripts (similarly to the specification framework described in Chapter 5). However, POSH can only optimize shell pipelines that are fully expanded—*i.e.*, ones that do not use dynamic features like variables and command substitution (see Section 2.4). In contrast to POSH, my work operates on shell programs that use (1) any POSIX composition primitive, and (2) the full set of dynamic features present in the UNIX shell.

Mosh [183] is a remote shell that is designed for intermittent and high-latency connectivity environments. It is meant to be used as the backbone shell for managing a set of distributed computer nodes. It develops a novel state synchronization protocol that improves latency and compared to SSH by speculatively echoing user keystrokes before they have reached the server. This dramatically improves user-experience in low network quality environments because the network latency is hidden from the user's interactive terminal. The goals of my work are orthogonal to Mosh: Mosh optimizes the network and terminal layers of the stack, improving interactive remote shell user experience, while my work optimizes the execution of computationally demanding and long running programs by leveraging parallel and distributed computational resources.

Build systems Build systems target a specific class of workloads, where computation can be expressed as a dependency graph of black-box tasks, allowing independent tasks to be processed independently and in parallel. A quintessential such workload, which is also how these systems took their name, is building software, including (1) configuration, determining relevant characteristics of the underlying system that affect the build; (2) compilation, producing binary code from the program source code; (3) linking, creating a final executable by collecting relevant libraries; (4) testing, running the executable on a representative set of inputs to check that it behaves as expected; and (5) deployment, running the executable on the target system. Even though software builds were the original use case for build systems, they are now used for a variety of workloads that can be expressed using a dependency graph, often called workflows. Build systems and their dependency graph representation also enables *incremental execution*, *i.e.*, avoiding reexecution of parts of the workflow when input has not changed. To achieve that, build systems need to discover the dependencies, *i.e.*, inputs and outputs, of all steps of the target workflow, which they achieve with a combination of user input and dynamic tracing.

On one end of the spectrum, backward build systems like Make [160], OMake [80], Shake [126], Ninja [11], Vesta [79], Buck [55], and Bazel [4], require user input to achieve performance benefits. Developers are expected to explicitly describe the dependencies of each workflow step using some forms of annotations. Using that information, they construct a dependency graph for each workflow and are able to parallelize and execute it incrementally. The correctness and performance of these systems depends on the precision of these annotations: missing a dependency might not trigger the right tasks when changes happen, and adding too many dependencies might lead to redundant recomputation. My work borrows ideas from build systems, *i.e.*, modelling program fragments as graphs, albeit of a different type, but does not require users to describe the dependencies for each program to achieve benefits; in contrast, it leverages specifications that need to be written once per command and can be reused across different programs. The main difference of the graphs used in this dissertation (see Chapter 4) and the graphs in build systems, is that graphs in my work describe continuous dataflow computation, namely tasks that exchange data throughout their execution, instead of dependencies where a task reads the complete output of another task when it is done executing.

On the other end of the spectrum, forward build systems like memoize [116], fabricate [83], Rattle [158]

and more recently Riker [45], achieve incremental execution of shell programs by tracing program execution and constructing the dependency graph at runtime. These systems do not make any assumptions about a program's execution or its commands and therefore do not require any kind of input from users to achieve their benefits. However, being fully dynamic, these systems cannot perform optimizations such as parallelization or distribution: it is too late to parallelize a command after it has started executing and performing side-effects. My work combines command specifications and dynamic tracing to be able to handle all programs, but also to achieve better performance for programs where more structure about each command is known. Furthermore, my work introduces a formal model for an optimizable target fragment of the shell, enabling optimizations that can be proved correct.

Serverless execution of workflows A recent system that has a similar architecture with the work described in this dissertation is gg [59]. It enables distributing traditionally local applications, such as software builds, unit testing, and other data processing tasks, over serverless functions. The key insight behind gg is that it introduces an intermediate representation that describes dependency graphs of computation tasks. Each node of this dependency graph abstraction is a computation thunk that has explicit inputs and outputs; given the inputs and outputs of each thunk, it is possible to infer dependencies among them. Since each thunk is independent, with explicit input requirements, it can easily be deployed in a lightweight container by collecting all of its dependencies. This dependency graph is similar to the ones used in build systems, meaning that in contrast to our dataflow model (Chapter 4) it does not describe continuous dataflow computation: a task can start executing after all of its dependencies have been resolved. Furthermore, gg focuses on task parallelism, *i.e.*, each independent task in the dependency graph can be run in parallel, and does not apply transformations on the graph that can introduce and exploit data-parallelism.

A benefit of gg's intermediate representation is that it decouples programs (frontend) from their deployment (backend). It supports several frontends, which can be broadly separated into two categories: (1) SDKs that allow describing a computation as an explicit dependency graph in a language of choice (such as C++); and (2) a model-based frontend that can automatically generate a dependency graph given a program that uses a restricted set of components. The model-based frontend contains models of several third party commands, mostly focusing on tools that are used in software builds, such as the compiler, assembler, and linker. These

models are akin to the specifications that are described in this dissertation (Chapter 5), the main difference being that they are more expressive, since they can be written in arbitrary code, even sharing code with the command implementation itself. For example, gg's model for the compiler preprocessor can correctly determine that its input dependencies contain the header files that were included in the source file. In contrast, the specifications described in this dissertation are more restricted, *i.e.*, can only describe inputs and outputs that are explicitly referenced in command arguments, but might lead to simpler specifications for simple commands. A comprehensive solution could combine both approaches, providing (1) a specification language for simple commands (to have a low barrier of entry for users) and (2) a fully expressive model subsystem that could describe arbitrary command behavior. Furthermore, gg's model-based approach can only capture dependencies that are fully described by the command models. This means that gg cannot capture the dependencies between commands that are composed using arbitrary shell code. In contrast to gg, PASH addresses this challenge by introducing a just-in-time execution model (see Chapter 7) that avoids reasoning about arbitrary shell code by letting it execute in the shell prior to reasoning and optimizations.

Shell semantics There has been a recent resurgence of research studying the semantics of shell programs. CoLiS [88] is a formally defined and well-behaved alternative shell language that was used as a compilation target to study the behaviors of a large number of Debian package installation programs [26]. This study detected a number of bugs and policy violations in these programs by leveraging a library of specifications for many POSIX utilities [87]. Smoosh [71] develops a formalized, executable reference semantics for the POSIX shell, aiming to address subtleties in the standard [22]. It identifies several divergent behaviors between existing popular shell implementations and develops a new shell implementation that completely follows the POSIX specification. My research builds on insights from both of these lines of work: Smoosh's semantics guides the JIT compiler architecture, its parsing library was the foundation of PASH's parsing library implementation (see Section 7.3), and the design of the command annotation language (see Chapter 5) was guided by the specification of POSIX utilities, while focusing on a different set of properties.

3.2. Dataflow graph models

Directed graph models for parallel and distributed computation have been studied extensively, from the context of synchronous languages (*e.g.*, [104, 147, 29, 115]) to distributed batch and stream processing (*e.g.*, [46, 129, 186, 67, 168, 113]). These models are often called dataflow graph (DFG) models, or simply

dataflow, because edges capture the flow of data between different computation nodes. The main benefit of such models is that they expose both (i) parallelism opportunities, *i.e.*, each different node is a different computational unit that can be executed in parallel, and (ii) parallelism restrictions, *i.e.*, an edge implies that there is some communication/synchronization requirement between the nodes it connects. This makes them an ideal intermediate abstraction between high-level programs and parallel implementations since a compiler can transform a given program to a dataflow graph, which can then be optimized given the available computational resources and requirements.

Since dataflow graphs have been studied in so many different contexts, there are many similar but subtly different models in the literature, each of which has suitable properties for a specific context. In this disseration I am not interested in presenting a complete and thorough exploration of all such models; for that, the reader can turn to one of many extensive surveys on the topic, *e.g.*, the work by Lee [106] or the work by Johnson, Hanna, and Millar [90]. I provide enough context on DFG models to situate the order-aware-dataflow model (ODFM) that is proposed in Chapter 4.

One of the first works that proposed dataflow models as a semantics for parallel computation was the work on Kahn process networks [93, 94]. In Kahn process networks a set of sequential processes communicate through some unbounded first-in, first-out (FIFO) channels Some important properties of Kahn process networks are that they are deterministic, *i.e.*, timing does not affect the results of their execution, and monotonic, which means that the more input is consumed, the more output is produced; put differently, output cannot be retracted once it is produced. Due to these properties, Kahn process networks are suitable for modeling stream processing systems and they are also the model of communication of UNIX pipes. ODFM is a similar model to Kahn process networks but focuses on the particularities of the shell and on enabling parallelization transformations, and therefore differs from KPNs in two key ways: (1) it does not support cycles, and (2) it exposes information about the input consumption order of each node. This order provides enough information at compile time to perform parallelizing transformations while also enabling translation of the dataflow back to a UNIX shell program. This makes ODFM strike a unique balance compared to most other dataflow models. On the one hand we have more abstract DFGs used for data processing (batch or streaming), where nodes do not expose information about the order in which they consume input from their edges;

a choice that enables parallelization transformations but is not restrictive enough to capture the semantics of shell programs. On the other, we have explicit dataflow models like the ones used in synchronous languages (*e.g.*, Lustre [147], Esterel [29], Signal [104]) that focus on efficient low-level implementations and therefore require that each node takes very specific types of inputs.

3.3. Language parallelization support

Prior research on providing parallelization support for other languages and environments usually follows one of two approaches: (1) developing a new higher-level language that is more amenable to parallelization, or (2) developing analyses and tooling to extract parallelism from programs writen in existing mostly sequential languages. The first approach (*e.g.*, [60, 68, 102, 46, 186, 129, 40, 163, 25]) can usually achieve much higher benefits, exploiting finer granularities of parallelization, but comes with all the drawbacks of switching language environments: developers have to learn a new language that might not be as mature as existing ones and have to rewrite their existing programs to achieve any benefit. My work differs in that it operates on completely unmodified shell programs that exercise the whole gamut of POSIX shell behaviors.

The second approach develops tools that provide automatic parallelization for standard sequential code, requiring no program modifications but often posing limitations with respect to the granularity of the parallelism that they can extract. Developments started with explicit `DOALL` and `DOACROSS` annotations [38, 111], continuing with analysis-based compilers [137, 75, 149], profiling-guided speculation [121, 169, 100, 89, 18], and more recently using program synthesis to parallelize fragments of imperative code [152, 56, 57, 157]. These developments have achieved significant performance benefits through parallelization by exploiting available multicore hardware for existing applications. My work draws inspiration from this line of work in that it does not require manual modification to user code and it leverages run-time information to optimize and parallelize user programs. However, these approaches operate at a lower level of abstraction than the work described in this dissertation: they focus on extracting parallelism from single instructions and loops in a single language environment. Instead, my work focuses on a broader, multi-language and whole-program setting: given information about how each single command can be parallelized in a divide-and-conquer fashion, it lifts parallelization across a whole program composing multiple such commands that are written in different languages. This makes such techniques complementary to my work, since they can be used to derive

aggregators and the parallelizability properties of yet unknown shell commands, enabling the parallelization of programs that use them.

3.4. Distributed systems

The work described in this dissertation draws inspiration from several lines of work in the vast literature of distributed systems; from the long line of work on operating systems for the management of distributed clusters of computer nodes, to work on scalable data processing in the context of distributed compute resources, to work on automated optimizations of black-box programs to exploit distribution through the use of user-defined annotations.

Distributed operating systems There is a long history of networked and distributed operating systems [146, 179, 136, 128, 140, 151, 48, 24, 153]. These systems offer abstractions that (1) are similar, but not identical, to the ones offered by UNIX, (2) operate at a lower level of abstraction (*e.g.*, that of system calls, rather than shell primitives), and (3) often aim at simply hiding the network rather than offering scalability benefits. Many of these systems support alternative shells better suited to their purposes, *e.g.*, rc for Plan9, that differ in subtle or non-subtle ways from existing shells. My research takes a different approach: instead of replacing existing abstractions, it develops optimizations that provide performance benefits on top of existing abstractions without requiring any rewriting of user programs.

Annotation-based transformations Recent systems [174, 138, 185] lower the developer effort of scaling out program components by performing program transformations based on user-provided annotations. These systems operate in single-language environments, offering declarative DSLs for tuning the semantics of the resulting distributed program. These systems inspired my work and its use of command annotations to enable the analysis and transformation of programs that use black box components. A difference of the annotations in the context of PASH compared to these other systems is that annotations are more likely to be reused, since many users have access to a same core set of commands, such as the POSIX and GNU Coreutils ones.

Distributed data processing Several systems assist in the development of distributed data processing applications [46, 130, 186, 129, 161, 40] that fall under certain computational classes such as batch and stream processing. The systems deal with many of the challenges of distribution, such as crashes and network faults, but require developers to (re)write their computations manually to reap their benefits. Some of these systems,

e.g., Hadoop Streaming [74] and Dryad Nebula [85], also support third-party language-agnostic components similar to the UNIX shell, atop cluster-computing engines (Hadoop and Dryad, respectively). Both require their users to understand and rewrite their shell programs using the abstractions provided by each framework. The main difference of PASH is that it operates on arbitrary shell programs, without requiring any rewriting of the program from the user.

3.5. Just-in-time and staged compilation

A key component of this dissertation is the just-in-time optimization architecture described in Chapter 7. This component is novel but draws inspiration from the long line of work on just-in-time compilation (for an extended survey see Aycock [23]). Prior work on just-in-time compilation can be split in two categories; it has been studied as (1) a compilation technique for interpreted languages such as JavaScript [65], where critical type information is unavailable prior to execution; and (2) a performance optimization over ahead-of-time compilation, allowing for specialization [167, 86], loop unrolling and function inlining [32, 141], and other profile-guided optimizations [135, 97]. In addition to these two categories, there is also prior work on staged compilation [42] and partial evaluation [91]—techniques that perform some compilation ahead-of-time, waiting for the runtime to specialize and further optimize when there is more information about the environment of the target program and how it is used. The key idea behind all of this prior work is that compilation done at run-time can exploit an abundance of additional information compared to one done statically, ahead-of-time. This information enables significant performance benefits and that is why today many languages have a just-in-time or profile driven compiler.

The work described in this dissertation draws inspiration from work in both contexts—resolving unavailable dynamic information at run-time and performing additional optimizations. It also leverages the optimistic compilation technique employed commonly by just-in-time compilers: when it fails to compile (parallelize), it simply runs the original fragment using the shell interpreter as a fallback. However, compared to most JIT compilers, the just-in-time architecture described in Chapter 7 also deals with a different set of challenges: it operates at a higher level of abstraction (no binary or bytecode, but rather a shell program), with no single unified runtime (multiple runtimes and languages coexist during the execution of a shell program).

CHAPTER 4

A formal model of a data processing fragment of the shell

Material from this chapter was previously published as "Shivam Handa, Konstantinos Kallas, Nikos Vasilakis, and Martin C. Rinard. An Order-Aware Dataflow Model for Parallel Unix Pipelines. Proc. ACM Program. Lang., 5(ICFP), August 2021." [76]. Shivam, Nikos, and I contributed equally to this work, developing the formalization and order-aware dataflow model. I was the lead developer of the project's implementation.

4.1. Introduction

UNIX pipelines are an attractive choice for specifying succinct and simple programs for data processing, system orchestration, and other automation tasks [150]. Consider, for example, the following program based on the original `spell` written by Johnson [27], lightly modified for modern environments:[4]

```
cat f1.md f2.md | tr A-Z a-z | tr -cs A-Za-z '\n' | sort | uniq |                  # (Spell)
  grep -vx -f dict.txt - > out
cat out | wc -l | sed 's/$/ mispelled words!/'
```

The first command streams two markdown files into a pipeline that converts characters in the stream into lower case, removes punctuation, sorts the stream in alphabetical order, removes duplicate words, and filters out words from a dictionary file (lines 1 and 2). A second pipeline (line 3) counts the resulting lines to report the number of misspelled words to the user.

As this example illustrates, the UNIX shell offers a programming model that facilitates the composition of commands using unidirectional communication channels that feed the output of one command as an input to another. These channels are either ephemeral, unnamed pipes expressed using the `|` character and lasting for the duration of the producer and consumer, or persistent, named pipes (UNIX FIFOs) created with `mkfifo` and lasting until explicitly deleted. Each command executes sequentially, with pipelined parallelism available between commands executing in the same pipeline. Unfortunately, this model leaves substantial data parallelism unexploited.

[4] Johnson's program additionally used `troff`, `prepare`, and `col -bx` to clean up now-legacy formatting metadata that does not exist in markdown. Moreover, `comm -13` was replaced with `grep -xvf` to highlight crucial features of the model.

To support the ability to reason about and correctly transform and parallelize UNIX shell pipelines, we present a new *dataflow model*. In contrast to standard dataflow models [93, 94, 105, 107, 98], our dataflow model is *order-aware—i.e.*, the order in which a node in the dataflow graph consumes inputs from different edges plays a central role in the semantics of the computation and therefore in the resulting parallelization. This model is different from models that allow multiplexing different chunks of data in a single channel, such as sharding or tagging, or ones that are oblivious to ordering, such as shuffling—and is a direct byproduct of the ordered semantics of the shell and the opacity of UNIX commands. In the *Spell* script shown earlier, for example, while all commands consume elements from an input stream in order—a property of UNIX streams *e.g.*, pipes and FIFOs—they differ in how they consume across streams: `cat` reads input streams in the order of its arguments, `sort -m` reads input streams in interleaved fashion, and `grep -vx -f` first reads `dict.txt` before reading from its standard input.

We use this order-aware dataflow model (ODFM) to express the semantics of transformations that exploit data parallelism available in UNIX shell computations. These transformations form the basis of the parallelization systems described in later chapters of this dissertation. We also use our model to prove that these transformations are correct, *i.e.*, that they do not affect the program behavior with respect to sequential output.

In summary, this dissertation chapter makes the following contributions:

- **Order-Aware Dataflow Model:** It introduces the *order-aware dataflow model* (ODFM), a dataflow model tailored to the UNIX shell that captures information about the order in which nodes consume inputs from different input edges (§4.4).

- **Transformations and Proofs of Correctness:** It presents a series of ODFM transformations for extracting data parallelism. It also presents proofs of correctness for these transformations (§4.5).

The chapter starts with an informal development building the necessary background (§4.2) and expounding on *Spell* (§4.3). It then presents the main contributions outlined above (§4.4–4.5), compares with prior work (§4.6), and offers a discussion (§4.7).

4.2. Background

This section reviews background on commands and abstractions in the UNIX shell.

4.2.1. UNIX Streams

A key UNIX abstraction is the data *stream*, operated upon by executing commands or *processes*. Streams are sequences of bytes, but most commands process them as higher-level sequences of line elements, with the newline character delimiting each element and the EOF condition representing the end of a stream. Streams are often referenced using a filename, that is an identifier in a global name-space made available by the UNIX file-system such as `/home/user/x`. Some streams can persist as files beyond the execution of the process, whereas other streams are ephemeral in that they only exist to connect the output of one process to the input of another process during their execution.

4.2.2. Commands

Each command is an independent computation unit that reads one or more input streams, performs a computation, and produces one or more output streams. Contrary to programming environments with a closed set of primitives, like Spark [186] and MapReduce [46], there is an unlimited number of UNIX commands, each one of which may have arbitrary behaviors—with the command's side-effects potentially affecting the entire environment on which it is executing. These commands may be written in any language or exist only in binary form, and thus UNIX is not easily amenable to a single parallelizability analysis. UNIX commands are also often configurable, customizing their behavior based on the task at hand. This is usually achieved via environment variables and command options and flags. Prior parallelization tools such as GNU `parallel` leave such analysis to developers that have to ensure that the script behavior will not be affected by parallelization. The work that is presented in later chapters of this dissertation instead introduces specification libraries that identify and describe key properties that hold for entire classes of commands. For example, a property that is useful for parallelization is whether a command is stateless, *i.e.*, whether it maintains state when processing different input items, or whether it processes each input line independently. Commands that satisfy this property can be parallelized by splitting their inputs in lines and then combining their outputs.

4.2.3. Order of input consumption

In UNIX, all streams are ordered and all commands consume elements from their streams in the order they were produced. Additionally, most commands have the ability to operate on multiple files or streams. The order in which commands access these streams is important. In some cases, they read streams in the order of the stream identifiers provided. In other cases, the order is different—for example, an input stream may configure a command, and thus must be read before all the others. Consider for example `grep -f words.txt input.txt`, which first reads `words.txt` to determine the keywords for which it needs to search, and then reads `input.txt` line by line, emitting all lines that contain one of the words in `words.txt`. In other cases, reads from multiple streams are interleaved according to some command-specific semantics.

4.2.4. Composition: UNIX Operators

UNIX provides several primitives for program composition, each of which imposes different scheduling constraints on the program execution. Central among them is the *pipe* (`|`), a primitive that passes the output of one process as input to the next. The two processes form a pipeline, producing output and consuming input concurrently and possibly at different rates. The UNIX kernel facilitates program scheduling, communication, and synchronization behind the scenes. For example, *Spell*'s first `tr` transforms each character in the input stream to lower case, passing the stream to the second `tr`: the two `tr`s form a parallel producer-consumer pair of processes.

Apart from pipes, the language of the UNIX shell provides several other forms of composition—*e.g.*, the sequential composition operator (`;`) for executing one process after another has completed, and control structures such as `if` and `while`. All of these constructs enforce execution ordering between their components.

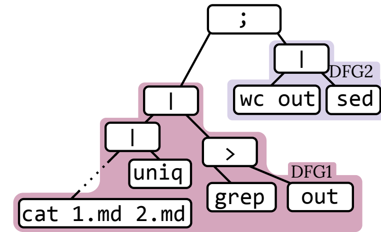
4.3. Example and overview

This section provides intuition of the order-aware dataflow model proposed by following the different phases of a shell-to-shell parallelizing compiler, formalized in the later sections. Given a script such as *Spell* (§4.1), the compiler identifies its regions between synchronization barriers, translates them to DFGs (Shell→ODFM), applies graph transformations that expose data parallelism on these DFGs, and replaces the original dataflow

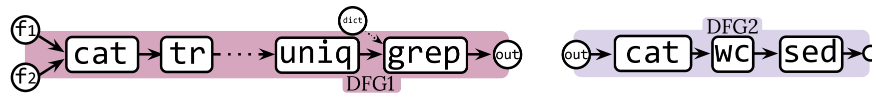
regions with the now-parallel regions (ODFM→Shell).

4.3.1. Shell→ODFM

Provided a shell script, the compiler starts by identifying subexpressions that are potentially parallelizable. The first step is to parse the script, creating an abstract syntax tree like the one presented on the right. Here we omit any non-stream flags and refer to all the stages between (and including) `tr` and `sort` as a dotted edge ending with `cat`.



The compiler then identifies parallelism barriers within the shell script: these are operators that enforce synchronization constraints, such as the sequential composition operator ("`;`"). We call any set of commands that does not include a dataflow barrier a *dataflow region*. Dataflow regions are then transformed to dataflow graphs (DFGs), i.e., instances of our order-aware dataflow model. In our example, there are two dataflow regions corresponding to the following dataflow graphs:

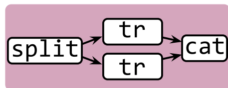


As mentioned earlier (§4.2), the compiler exposes parallelism in each DFG separately to preserve the ordering requirements imposed to ensure correctness. For the rest of this section we focus on the parallelization of DFG1.

4.3.2. Parallelizable Commands

Individual nodes of the dataflow graphs are shell commands. We assume that we have access to relevant information for individual commands, *e.g.*, whether they are amenable to divide-and-conquer data parallelism. Such data parallelism is achieved by splitting the input into pieces (at stream element boundaries), processing partial inputs in parallel, and finally applying an aggregation function to partial outputs to produce the final output. This decomposition breaks a command into two components—a data-parallel function, which is often the command itself, and an aggregation function. The table on the right presents aggregation functions for the shell commands in our example (all of which are parallelizable).

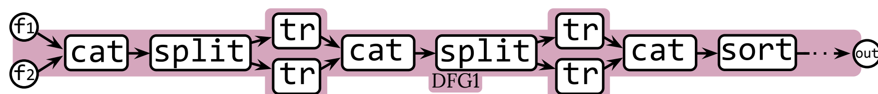
Command	Aggreg. Function
`cat`	`cat $*`
`tr A-Z a-z`	`cat $*`
`tr -d a`	`cat $*`
`sort`	`sort -m $*`
`uniq`	`uniq $*`
`grep -f a -`	`cat $*`
`wc -l`	`paste -d+ $*|bc`
`sed 's/a/b/'`	`cat $*`



For example, consider the decomposition of the `tr` command. Applying `tr` over the entire input produces the same result as `split`ting the input into two, applying `tr` to the two partial inputs, and then merging the partial results with a `cat` aggregation function. Note that both `split` and `cat` are order-aware, *i.e.*, `split` sends the first half of its input to the first `tr` and the rest to the second, while `cat` concatenates its inputs in order. This guarantees that the output of the DFG is the same as the one before the transformation.

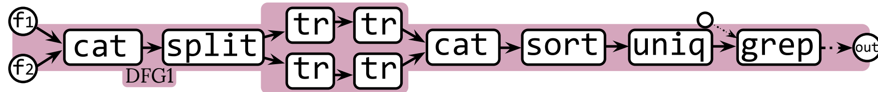
4.3.3. Parallelization Transformations

Given the decomposition of individual commands, the compiler's next step is to apply graph transformations to exploit parallelism present in the computation represented by the DFG. As each parallelizable UNIX command comes with a corresponding aggregation function, the compiler's transformations first convert the DFG into one that exploits parallelism at each stage. After applying the transformation to the two `tr` stages, the DFG looks as follows:

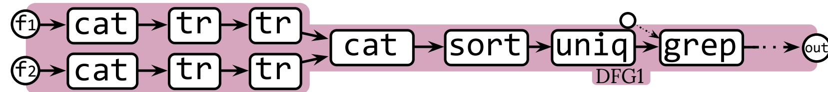


As long as `tr` takes more time to execute than scanning the input with `split` and concatenating the output with `cat`, this transformation can improve performance compared to the original, given adequate parallelism.

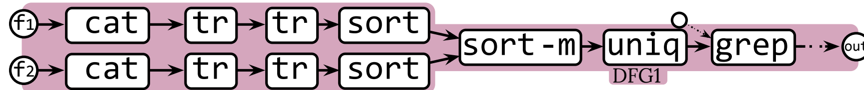
However we can do better! After these transformations are applied to all DFG nodes the next transformation pass is applied to pairs of `cat` and `split` nodes: whenever a `cat` is followed by a `split` of the same width, the transformation removes the pair and connects the parallel streams directly to each other. The goal is to *push* data parallelism transformations as far down the pipeline as possible to expose the maximal amount of parallelism. Here is the resulting DFG for the transformation applied to the two `tr` stages:



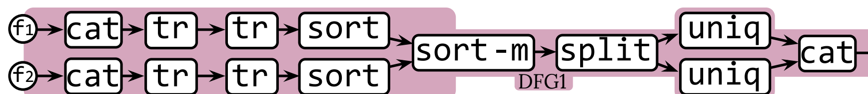
Applying this transformation to the first three stages—*i.e.*, `cat`, `tr`, and `tr`—of DFG1 produces the following transformed DFG.



The next node to parallelize is `sort`. To merge the partial output of parallel `sort`s, we need to apply a sorted merge. (In GNU systems, this is available as `sort -m` so we use this as the label of the merging node.) The transformation then removes `cat`, replicates `sort`, and merges their outputs with `sort -m`:



It then continues pushing parallelism down the pipeline, after applying a `split` function to split `sort -m`'s outputs.

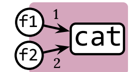


As mentioned earlier, a similar pass of iterative transformations is applied to DFG2, but the two DFGs are not merged to preserve the synchronization constraint of the dataflow barrier "`;`".

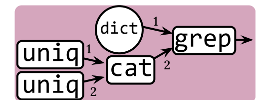
4.3.4. Order Awareness

Data-parallel systems [46, 186] often achieve parallelism using sharding, *i.e.*, partitioning input based on some key, or using shuffling, *i.e.*, arbitrary partitioning of inputs to parallel instances of an operator.

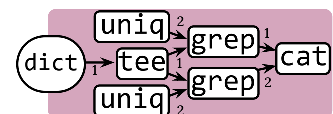
However, these techniques cannot be directly applied to the context of the shell, since (1) UNIX commands and pipelines assume strict ordering of their input elements; (2) most commands do not support sharding, because their inputs cannot be processed independently if grouped by some key; and (3) many commands are not commutative (*e.g.*, `uniq`, `cat -n`). Since our goal is to define a model that applies directly to existing shell scripts, we cannot simply introduce new primitives that support sharding or shuffling, as is done in the case of systems that design an abstraction that fits their needs (*e.g.*, MapReduce, Spark). Thus, data parallelism in the shell requires a careful treatment of input and output ordering.



To further explain the need for order-awareness in a model for data parallel UNIX pipelines, let's look at the following examples. Consider *Spell*'s `cat f1.md f2.md` command that starts reading from `f2.md` only after it has completed reading `f1.md`; note that any or both input streams may be pipes waiting for results from other processes. This order can be visualized as a label over each input edge. Correctly parallelizing this command requires ensuring that parallel `cat` (and possibly followup stages) maintains this order.



As a more interesting example, consider *Spell*'s `grep`, whose DFG is shown on the right. Parallelizing `grep` without taking order into account is not trivial, because `grep -vx -f`'s inputs are not all equal: the `dict` list of patterns should not be split into two partial inputs fed into two copies of `grep`. Taking input ordering into account, however, highlights an important dependency between `grep`'s inputs. The `dict` stream can be viewed as configuring `grep`, and thus `grep` can be modeled as consuming the entire `dict` stream before consuming partial inputs.



Armed with this insight, the compiler parallelizes `grep` by passing the same `dict.txt` stream to both `grep` copies. This requires an intermediary `tee` for duplicating the `dict.txt` stream to both copies of `grep`, each of which consumes the stream in its entirety before consuming the results of the preceeding `uniq`.

Order-awareness is also important to translate the DFG back to a shell script. In the specific example we need to know how to instantiate the arguments and options of each `grep`—*e.g.*, `grep -vx -f p1 p2`. Aggregators are also UNIX commands with their own ordering characteristics that need to be accounted for.

The order of input consumption in the examples of this section is statically known and can be represented for each node as a set of configuration inputs, plus a sequence of the rest of its inputs. To accurately capture the behavior of shell programs, however, ODFM is more expressive, allowing any order of input consumption. The correctness of our parallelization transformations is predicated upon static but configurable orderings: a command reads a set of *configuration* streams to set up the consumption order of its input streams which are then consumed in-order, one after the other.

4.3.5. ODFM→Shell

The transformed graph is finally compiled back to a script that uses POSIX shell primitives to drive parallelism explicitly. A benefit of the dataflow model is that it can be directly implemented on top of the shell, simply translating each node to a command, and each edge to a stream. The generated parallel script for *Spell* can be seen below.

```
mkfifo t{0..14}          # DFG1: start
tr A-Z a-z < f1.md > t0 &
tr A-Z a-z < f2.md > t1 &
tr -d[:punct:] < t0 > t2 &
tr -d[:punct:] < t1 > t3 &
sort < t2 > t4 &
sort < t3 > t5 &
sort -m t4 t5 > t6 &
split t7 t8 < t6 &
# ...
tee t9 > t10 < dict.txt &
grep -vx -f t9 - < t11 > t13 &
grep -vx -f t10 - < t12 > t14 &
cat t13 t14 > out &
wait
rm t{0..14}                # DFG1: end

mkfifo t{0..8}           # DFG2: start
split t0 t1 < out &
wc -l < t0 > t2 &
```

```
wc -l < t1 > t3 &
paste -d+ t2 t3 | bc > t4 &
split t5 t6 < t4 &
sed 's/$/ mispelled words!/' < t5 > t7 &
sed 's/$/ mispelled words!/' < t6 > t8 &
cat t7 t8 &
wait
rm t{0..8}                # DFG2: end
```

The two DFGs are compiled into the two fragments that start with `mkfifo` and end with `rm`. Each fragment uses a series of named pipes (FIFOs) to explicitely manipulate the input and output streams of each data-parallel instance, effectively laying out the structure of the DFG using explicit channel naming, UNIX FIFOs are named in the filesystem similar to normal files. Aggregation functions are used to merge partial outputs from previous commands coming in through multiple FIFOs—for example, `sort -m t4 t6` and `cat t11 t12` for the first fragment, and `paste -d+ t2 t3 | bc` and `cat t7 t8` for the second. A `wait` blocks until all commands executing in parallel complete.

The parallel script is simplified for clarity of exposition: it does not show the details of input splitting, handling of `SIGPIPE` deadlocks, and other technical details that are handled by the current implementation.

Readers might be wondering about the correctness of having two `sed` commands in the parallel script: won't the string "mispelled words" appear twice in the output? Note, however, that the behavior of this script is the same as the original: it appends the string "mispelled words" on all output lines. Since the output of the `wc` stage (fifo `t4`) only contains a single line, the second `sed` will not be given any input and thus will not produce any output.

4.4. An order-aware dataflow model

In this section we describe the order-aware dataflow model (ODFM) and its semantics.

4.4.1. Preliminaries

As discussed earlier (§4.2), the two main shell abstractions are (i) data streams, and (ii) commands communicating via streams. We represent streams as named variables and commands as functions that read from

$$
\begin{array}{rcl}
P & := & \mathcal{I}; \mathcal{O}; \overline{\mathcal{E}} \\
\mathcal{I} & := & \mathsf{input}\ \overline{x} \\
\mathcal{O} & := & \mathsf{output}\ \overline{x} \\
\mathcal{E} & := & \mathsf{Node}(f; \overline{x}_i; \overline{x}_o)
\end{array}
$$

Fig. 4.1: Dataflow Description Language (DDL). A language used to describe dataflow programs that consume a set of inputs and produce a set of outputs using a graph of computation nodes.

and write to streams.

We first introduce some basic notation formalizing data streams on which our dataflow description language works. For a set D, we write D^* to denote the set of all finite words over D. For words $x, y \in D^*$, we write $x \cdot y$ or xy to denote their concatenation. We write ϵ for the empty word and $\perp$ for the End-of-File condition. We say that x is a *prefix* of y, and we write $x \leq y$ if there is a word z such that $y = xz$. The $\leq$ order is reflexive, antisymmetric, and transitive (*i.e.*, it is a partial order), and is often called the *prefix order*. We use the notation $D^* \cdot \perp$ to denote a *closed stream*, abstractly representing a file/pipe stream that has been closed, *i.e.*, one which no process will open for writing. The notation D^* is used to denote an *open stream*, abstractly representing an open pipe. In the rest of out formalization we focus on terminating streams, and therefore terminating programs, since all of the data processing scripts that we have encountered are terminating.

4.4.2. Dataflow Description Language

Figure 4.1 presents the Dataflow Description Language (DDL) for defining dataflow graphs (DFG). A program P in DDL is of the form $\mathcal{I}; \mathcal{O}; \overline{\mathcal{E}}$. $\mathcal{I}$ and $\mathcal{O}$ represent sets of inputs and outputs, which are vectors of the form $\overline{x} = \langle x_1, x_2, \ldots, x_n \rangle$. Variables $x_1, x_2, \ldots$ represent DFG edges, *i.e.*, streams used as a communication channel between DFG nodes and as the input and output of the entire DFG.

$\mathcal{I}$ is of the form $\mathsf{input}\ \overline{x}$, where $\overline{x}$ is the set of the input variables. Each variable $x \in \mathcal{I}$ represents a file `file`(x) that is read from the UNIX filesystem. Note that multiple input variables can refer to the same file.

$\mathcal{O}$ is of the form $\mathsf{output}\ \overline{x}$, where $\overline{x}$ is the set of output variables. Each variable $x \in \mathcal{O}$ represents a file `file`(x) that is written to the UNIX filesystem.

$\mathcal{E}$ represents the nodes of the DFG. A node $\mathsf{Node}(f; \overline{x}_i; \overline{x}_o)$ represents a function from list of input variables

(edges) $\overline{x}_i$ to output variables (edges) $\overline{x}_o$.

A variable in DDL is assigned *only once* and consumed by *only one* node. Variables in $\mathcal{I}$ are never assigned in $\mathcal{E}$ and can only be consumed, and variables in $\mathcal{O}$ can not be read by any node in $\mathcal{E}$. Also the sets $\mathcal{I}$ and $\mathcal{O}$ must be disjoint. All variables which are not included in $\mathcal{I}$ and $\mathcal{O}$ abstractly represent temporary files/pipes which are created during the execution of a shell script. We assume that within a dataflow program, all variables are reachable from some input variables. Finally, DDL does not allow the dataflow graph to contain *any cycles*.

DDL Nodes: Each DDL node corresponds to a command, and therefore needs to satisfy the following properties.

First of all, we assume that commands do not produce output if they have not consumed any input, *i.e.*, the following is true:

$$\langle \epsilon, \dots, \epsilon \rangle = f(\epsilon, \dots, \epsilon).$$

This is not restrictive because we can model most commands as producing their first output either when they receive their first line of input, or when their input is closed.

We also require that f be monotone with respect to a lifting of the prefix order for a sequence of inputs; that is, $\forall, v, v', \overline{v}_i$, if $v \leq v'$, $\langle v_1, \dots, v_n \rangle = f(v, \overline{v}_i)$ and $\langle v'_1, \dots, v'_n \rangle = f(v', \overline{v}_i)$, then $\forall\ k \in [1, n].\ v_k \leq v'_k$. This captures the idea that a node cannot retract output that it has already produced.

We assume that all functions f produce well-formed output, namely that they never produce more values after they close an output stream with $\perp$. Additionally, we assume that a function f closes all of its outputs if all of its inputs are closed, namely:

$$f(v_1 \cdot \perp, v_2 \cdot \perp, \dots, v_n \cdot \perp) = \langle v'_1 \cdot \perp, v'_2 \cdot \perp, \dots, v'_k \cdot \perp \rangle.$$

At any point in time, a command f is waiting on a new message from a subset of its inputs, meaning that if a message arrives in any of these inputs, f can take a processing step. We introduce a function choice_f that returns a set of input indexes to represent this input consumption order. For example, the choice_{cat} function

$$\frac{\begin{array}{c}\mathsf{Node}(f; x_1, \ldots, x_k \ldots, x_n; x'_1, \ldots, x'_p) \in \mathcal{E} \qquad v_k \cdot v_x \leq \Gamma(x_k) \\ |v_x| = 1 \lor v_x = \bot \qquad k \in \mathsf{choice}_f(v_1, \ldots, v_n) \\ \langle v^m_1, \ldots, v^m_p \rangle = [\![f(v_1, \ldots, v_k \circ v_x \ldots, v_n)]\!]_s\end{array}}{\begin{array}{c}\mathcal{I}; \mathcal{O}; \mathcal{E} \vdash \Gamma[x'_1 \to v'_1, \ldots, x'_p \to v'_p], \sigma[x_1 \to v_1, \ldots, x_k \to v_k, \ldots, x_n \to v_n] \rightsquigarrow \\ \Gamma[x'_1 \to v'_1 \cdot v^m_1, \ldots, x'_p \to v'_p \cdot v^m_p], \sigma[x_1 \to v_1, \ldots, x_k \to v_k \cdot v_x, \ldots, x_n \to v_n]\end{array}} \text{STEP}$$

Fig. 4.2: Small Step Execution Semantics for DDL. A single step represents the computation of a dataflow node after it has consumed a new input message.

for the command `cat` always returns the next non-closed index—as `cat` reads its inputs in sequence, each one until depletion.

$$\{k+1\} = \mathsf{choice}_{cat}(v_1 \cdot \bot, \ldots, v_k \cdot \bot, v_{k+1}, \ldots, v_n).$$

For a choice_f function to be valid, it has to return an input index that has not been closed yet. Formally,

$$\mathcal{S} = \mathsf{choice}_f(v_1, \ldots, v_k \cdot \bot, \ldots, v_n) \implies k \notin \mathcal{S}.$$

We assume that the set returned by choice_f cannot be *empty* unless all input indexes are closed, meaning that all nodes consume all of their inputs until depletion even if they do not need the rest of it for processing.

Finally, we introduce an execution wrapper $[\![\cdot]\!]_s$ that describes the latest output of a command given its newly consumed input. We write $v_k \circ v_x$ only in the context of $[\![]\!]_s$ to identify the current input message that was consumed for this step. This is in contrast to $v_k \cdot v_x$, which simply represents the concatenation of two values. For any f, the output of $[\![\cdot]\!]_s$ is defined as follows:

$$\langle v^m_1, \ldots, v^m_p \rangle = [\![f(v_1, \ldots, v_k \circ v_x, \ldots, v_n)]\!]_s$$

if and only if

$$\langle v'_1, v'_2, \ldots, v'_p \rangle = f(v_1, \ldots, v_k, \ldots, v_n) \land \langle v'_1 \cdot v^m_1, \ldots, v'_p \cdot v^m_p \rangle = f(v_1 \ldots, v_k \cdot v_x, \ldots v_n).$$

Processing inputs $v_1, \ldots, v_k, \ldots, v_n$ and then message v_x is equivalent to processing all input $v_1, \ldots, v_k \cdot v_x, \ldots, v_n$ at once.

4.4.3. Semantics

Figure 4.2 presents the small step execution semantics for DDL. A map Γ associates variable names to the data contained in the stream it represents. A second map σ associates the same variable names to the data in the stream that has already been consumed—capturing the read-once semantics of UNIX pipes. Both Γ and σ have the same domain, containing mappings for all the variables in the program. We write $P \vdash \Gamma, \sigma \rightsquigarrow \Gamma', \sigma'$ when a DDL program $P = \mathcal{I}; \mathcal{O}; \mathcal{E}$ steps from state Γ, σ to state Γ', σ'. We use $\rightsquigarrow^*$ to denote 0 or more such program steps.

The small step semantics nondeterministically picks a variable x_k, such that $k \in \mathsf{choice}_f(v_1, \ldots, v_n)$, *i.e.*, f is waiting to read some input from x_k, and $\sigma(x_k) < \Gamma(x_k)$, *i.e.*, there is data on the stream represented by variable x_k that has to be processed. The execution then retrieves the next message v_x to process, and computes new messages $v_1^m, \ldots, v_p^m$ to pass on to the output streams $x_1', \ldots, x_p'$. Note that any of these messages (input or output) might be $\perp$.

The messages $v_1^m, \ldots, v_p^m$ are passed on to their respective output streams (by updating Γ). Note that the size of the output messages could vary, and they could even be empty. Finally, σ is updated to denote that v_x has been processed.

4.4.4. Execution Properties

Let $P = \mathcal{I}; \mathcal{O}; \mathcal{E}$ be a dataflow program, where $\mathcal{I} = \mathsf{input}\ \overline{x}_i$ are the input variables, and $\mathsf{output}\ \overline{x}_o$ are the output variables. We denote the initial mapping σ for any such program as σ_i, where all variables are mapped to the empty string ϵ, *i.e.*, no data has been consumed by any node yet. Let Γ_i be the initial mapping Γ for any dataflow program. We assume that all non-input variables $x \notin \overline{x}_i$, map to the empty string $\Gamma_i(x) = \epsilon$. In contrast, all input variables $x \in \overline{x}_i$, *i.e.*, files already present in the file system, are mapped to the contents of the respective input file $\Gamma_i(x) = v \cdot \perp$. We say that a mapping Γ or σ is closed if and only if all of its variables are closed, *i.e.*, $\forall x, \Gamma(x) = v \cdot \perp$. When no more small step transitions can take place (*i.e.*, all commands have finished processing), the dataflow execution terminates and the contents of output variables in $\mathcal{O}$ can be written to their respective output files.

Theorem 1. *Given a program $P = \mathcal{I}; \mathcal{O}; \mathcal{E}$ and starting maps Γ_i and σ_i, the following statement is true for*

any $\mathsf{Node}(f; x_1, \ldots, x_n; x'_1, x'_2, \ldots, x'_p) \in \mathcal{E}$ *and for any* Γ, σ *such that* $P \vdash \Gamma_i, \sigma_i \rightsquigarrow^* \Gamma, \sigma$*:*

$$\langle \Gamma(x'_1), \ldots, \Gamma(x'_p) \rangle = f(\sigma(x_1), \ldots, \sigma(x_n)).$$

Proof. Proof by induction on the number of execution steps.

Base Case: Let Γ_i and σ_i be the initial mappings. For any x_j in $\langle x_1, \ldots, x_n \rangle$ we know that $\sigma_i(x) = \epsilon$, which means no input has been consumed. Similarly for all x'_j in $\langle x'_1, \ldots, x'_p \rangle$ we know that $\Gamma_i(x) = \epsilon$, since $x'_1, \ldots, x'_p$ are not input variables to the DFG so they will be initialized to ϵ. Given that all functions do not produce output if they have not consumed input, *i.e.*, $\langle \epsilon, \ldots, \epsilon \rangle = f(\epsilon, \ldots, \epsilon)$, the following holds:

$$\langle \Gamma_i(x'_1), \ldots, \Gamma_i(x'_p) \rangle = f(\sigma_i(x_1), \ldots, \sigma_i(x_n)).$$

Induction Case: Let Γ and σ be the mappings after n execution steps such that the following statement is true:

$$\langle \Gamma(x'_1), \ldots, \Gamma(x'_p) \rangle = f(\sigma(x_1), \ldots, \sigma(x_n)).$$

We need to show that after a single execution step $P \vdash \Gamma, \sigma \rightsquigarrow \Gamma', \sigma'$, the following holds:

$$\langle \Gamma'(x'_1), \ldots, \Gamma'(x'_p) \rangle = f(\sigma'(x_1), \ldots, \sigma'(x_n)).$$

Case 1 (a different node of the DFG took a step): If the σ mappings $\sigma(x_i) = \sigma'(x_i)$ are the same for all inputs $i \in [1, n]$, then no message was consumed by node f and therefore the Γ mappings for $x'_1, \ldots, x'_p$ were not updated (since only a single node, in this case f, writes to each variable). More precisely for all $k \in [1, p]$ we know that $\Gamma(x'_k) = \Gamma'(x'_k)$. Then, assuming the induction hypothesis, the following statement is true:

$$\langle \Gamma'(x'_1), \ldots, \Gamma'(x'_p) \rangle = f(\sigma'(x_1), \ldots, \sigma'(x_n)).$$

Case 2 (node f took a step): If there exists an $i \in [1, n]$ such that $\sigma'(x_i) = \sigma(x_i) \cdot v_x$, where $v_x \neq \epsilon$, then a message v_x was processed. Note that the above statement can only be true for a single i, so for all $j \neq i$ we

know that $\sigma(x_j) = \sigma'(x_j)$.

The following statement is true from our induction hypothesis:

$$\langle \Gamma(x'_1), \ldots, \Gamma(x'_p) \rangle = f(\sigma(x_1), \ldots, \sigma(x_i), \ldots, \sigma(x_n)).$$

and from small step semantics, for all $k \in [1, p]$ we know that $\Gamma'(x'_k) = \Gamma(x'_k) \cdot v^m_k$, where:

$$\langle v^m_1, \ldots, v^m_p \rangle = [\![f(\sigma(x_i), \ldots, \sigma(x_i) \circ v_x, \ldots, \sigma(x_n))]\!]_s.$$

Using the definition of $[\![\cdot]\!]_s$, the following statement is true:

$$\langle \Gamma(x'_1) \cdot v^m_1, \ldots, \Gamma(x'_p) \cdot v^m_p \rangle = f(\sigma(x_1), \ldots, \sigma(x_i) \cdot v_x, \ldots, \sigma(x_n)).$$

Therefore, the following is true:

$$\langle \Gamma'(x'_1), \ldots, \Gamma'(x'_p) \rangle = f(\sigma'(x_1), \ldots, \sigma'(x_i), \ldots, \sigma'(x_n)).$$

This concludes the inductive case and the proof. □

We now define a lemma that relates the data that is contained in a stream with the data that is read by its consumer.

Lemma 1. *Given $P = \mathcal{I}; \mathcal{O}; \mathcal{E}$ and $\Gamma_i, \sigma_i, \Gamma, \sigma$ such that $P \vdash \Gamma_i, \sigma_i \rightsquigarrow^* \Gamma, \sigma$, then for all variables x in a program P, the consumed data $\sigma(x)$ is a prefix of the data contained in $\Gamma(x)$, more precisely $\sigma(x) \leq \Gamma(x)$.*

Proof. By induction on the steps $\rightsquigarrow^*$. □

We are now ready to state the main property of the execution of a DDL program, namely, when a DDL program terminates, the data in its streams can be precisely determined by the outputs of its nodes f when given all their closed input, without needing the small step execution $[\![\cdot]\!]_s$.

Theorem 2. *Given $P = \mathcal{I}; \mathcal{O}; \mathcal{E}$ and $\Gamma_i, \sigma_i, \Gamma, \sigma$ such that $P \vdash \Gamma_i, \sigma_i \rightsquigarrow^* \Gamma, \sigma$ and both Γ and σ are closed, then for all $\mathsf{Node}(f; x_1, \dots, x_n; x'_1, \dots, x'_p) \in \mathcal{E}$, the following holds:*

$$f(\Gamma(x_1), \dots, \Gamma(x_n)) = \langle \Gamma(x'_1), \dots, \Gamma(x'_p) \rangle.$$

Proof. Given Theorem 1, we know that

$$f(\sigma(x_1), \dots, \sigma(x_n)) = \langle \Gamma(x'_1), \dots, \Gamma(x'_p) \rangle.$$

Using Lemma 1 and the fact that σ is closed, we know that $\sigma(x) \leq \Gamma(x)$ for any x, and so

$$f(\Gamma(x_1), \dots, \Gamma(x_n)) = \langle \Gamma(x'_1), \dots, \Gamma(x'_p) \rangle,$$

which concludes the proof. □

4.5. Parallelization transformations

In this section we define a set of transformations that expose data parallelism on a dataflow graph. We start by defining a set of helper DFG nodes and a set of auxiliary transformations to simplify the graph and enable the parallelization transformations. Then we identify a property on dataflow nodes that indicates whether the node can be executed in a data parallel fashion. We then define the parallelization transformations and we conclude with a proof that applying all of the transformations preserves the semantics of the original DFG.

4.5.1. Helper Nodes and Auxiliary Transformations

Before we define the parallelization transformations, we introduce several helper functions that can be used as dataflow nodes. We assume that all nodes satisfy the assumptions described in Section 4.4.2. Note that we do not describe the streaming behavior of helper functions, i.e., their outputs on open inputs, allowing for multiple streaming implementations as long as they are monotone (Section 4.4.2) and they agree with the helper function characterizations below.

The first function is $\mathsf{Node}(\mathsf{cat}; x_i; \overline{x})$, which behaves the same as the UNIX command `cat`. Given a list of

input variables cat combines their values into a single output variable:

$$\mathsf{cat}(v_1 \cdot \bot, v_2 \cdot \bot, \ldots, v_m \cdot \bot) = v_1 \cdot v_2 \ldots \cdot \bot.$$

The second function is $\mathsf{Node}(\mathsf{tee}; x_i; \overline{x})$, the behavior of which corresponds to the UNIX command `tee`, i.e. copying its input variable to several output variables.

$$\mathsf{tee}(v \cdot \bot) = \langle v \cdot \bot, \ldots, v \cdot \bot \rangle.$$

The third function is $\mathsf{Node}(\mathsf{relay}; x; x')$, which works works as an identity function.

$$\mathsf{relay}(v \cdot \bot) = v \cdot \bot.$$

Finally, we have $\mathsf{Node}(\mathsf{split}; x_i; \overline{x})$, which takes a single input variable (file or pipe) and sequentially splits it into multiple output variables. Its behavior is characterized as follows:

$$\mathsf{split}(v_1 \cdot v_2 \ldots \cdot v_k \cdot \bot) = \langle v_1 \cdot \bot, v_2 \cdot \bot, \ldots, v_k \cdot \bot, \bot, \bot, \ldots \rangle.$$

In contrast to the aforementioned nodes, this characterization does not uniquely define a single split function since there are many different ways to split its input. This allows the implementation to choose one of many different instantiations of split to achieve different performance characteristics without affecting correctness; the only requirement is that the instantiation satisfies the split property.

Using these helper nodes, our compiler performs a set of auxiliary transformations that are described in Figure 4.3 and depicted in Figure 4.4. Since relay acts as an identity function any edge can be split in two edges composed through a relay. Spliting in multiple stages to get *n* edges is the same as splitting in one step into *n* edges. Similarly, combining *n* edges in multiple stages is the same as combining *n* edges in a single stage. If we split an edge into *n* edges and then combine the *n* edges back, this behaves as an identity. A cat can be pushed following a tee by creating *n* copies of the tee function. If a cat has single incoming edge, we can convert it into a relay. If a split has a single outgoing edge, we can convert it into a relay. A split after a

$$\frac{x' \notin \mathcal{I};\mathcal{O};\mathcal{E} \qquad \mathcal{E}' = \mathcal{E}[x'/x]}{\mathcal{I};\mathcal{O};\mathcal{E} \Longleftrightarrow \mathcal{I};\mathcal{O};\mathcal{E}' \cup \{\mathsf{Node}(\mathsf{relay}; x; x')\}} \ \textsc{Relay}$$

$$\frac{x_s, x_s' \notin \mathcal{I};\mathcal{O};\mathcal{E} \qquad E = \left\{\mathsf{Node}(\mathsf{split}; x; x_s, x_s'), \mathsf{Node}(\mathsf{split}; x_s; x_1, \ldots, x_k), \mathsf{Node}(\mathsf{split}; x_s'; x_{k+1}, \ldots, x_m)\right\}}{\mathcal{I};\mathcal{O};\mathcal{E} \cup \left\{\mathsf{Node}(\mathsf{split}; x; x_1, \ldots, x_m)\right\} \Longleftrightarrow \mathcal{I};\mathcal{O};\mathcal{E} \cup E} \ \textsc{Split-Split}$$

$$\frac{x_c, x_c' \notin \mathcal{I};\mathcal{O};\mathcal{E} \qquad E = \left\{\mathsf{Node}(\mathsf{cat}; x_1, \ldots, x_k; x_c), \mathsf{Node}(\mathsf{cat}; x_{k+1}, \ldots, x_m; x_c'), \mathsf{Node}(\mathsf{cat}; x_c, x_c'; x)\right\}}{\mathcal{I};\mathcal{O};\mathcal{E} \cup \left\{\mathsf{Node}(\mathsf{cat}; x_1, \ldots, x_m; x)\right\} \Longleftrightarrow \mathcal{I};\mathcal{O};\mathcal{E} \cup E} \ \textsc{Concat-Concat}$$

$$\frac{x_1, \ldots, x_n \notin (\mathcal{I};\mathcal{O};\mathcal{E}) \qquad E = \left\{\mathsf{Node}(\mathsf{split}; x; x_1, \ldots, x_n), \mathsf{Node}(\mathsf{cat}; x_1, \ldots, x_n; x'), \right\}}{\mathcal{I};\mathcal{O};\mathcal{E} \cup \left\{\mathsf{Node}(\mathsf{relay}; x; x')\right\} \Longleftrightarrow \mathcal{I};\mathcal{O};\mathcal{E} \cup E} \ \textsc{Split-Concat}$$

$$\frac{\begin{array}{c} x_1^u, x_1^d, x_2^u, x_2^d, \ldots, x_n^u, x_n^d \notin \mathcal{I};\mathcal{O};\mathcal{E} \\ E = \left\{\mathsf{Node}(\mathsf{tee}; x_1; x_1^u, x_1^d), \mathsf{Node}(\mathsf{tee}; x_2; x_2^u, x_2^d), \ldots, \mathsf{Node}(\mathsf{tee}; x_n; x_n^u, x_n^d), \right. \\ \left. \mathsf{Node}(\mathsf{cat}; x_1^u, x_2^u, \ldots, x_n^u; x_o), \mathsf{Node}(\mathsf{cat}; x_1^d, x_2^d, \ldots, x_n^d; x_o')\right\} \end{array}}{\mathcal{I};\mathcal{O};\mathcal{E} \cup \left\{\mathsf{Node}(\mathsf{cat}; x_1, x_2, \ldots, x_n; x), \mathsf{Node}(\mathsf{tee}; x; x_o, x_o')\right\} \Longleftrightarrow \mathcal{I};\mathcal{O};\mathcal{E} \cup E} \ \textsc{Tee-Concat}$$

$$\frac{}{\mathcal{I};\mathcal{O};\mathcal{E} \cup \{\mathsf{Node}(\mathsf{cat}; x; x')\} \Longleftrightarrow \mathcal{I};\mathcal{O};\mathcal{E} \cup \{\mathsf{Node}(\mathsf{relay}; x; x')\}} \ \textsc{One-Concat}$$

$$\frac{}{\mathcal{I};\mathcal{O};\mathcal{E} \cup \{\mathsf{Node}(\mathsf{split}; x; x')\} \Longleftrightarrow \mathcal{I};\mathcal{O};\mathcal{E} \cup \{\mathsf{Node}(\mathsf{relay}; x; x')\}} \ \textsc{One-Split}$$

$$\frac{E = \left\{\mathsf{Node}(\mathsf{relay}; x_1; x_1'), \mathsf{Node}(\mathsf{relay}; x_2; x_2'), \ldots, \mathsf{Node}(\mathsf{relay}; x_n; x_n')\right\}}{\mathcal{I};\mathcal{O};\mathcal{E} \cup \left\{\mathsf{Node}(\mathsf{cat}; x_1, x_2, \ldots, x_n; x), \mathsf{Node}(\mathsf{split}; x; x_1', x_2', \ldots, x_n')\right\} \Longrightarrow \mathcal{I};\mathcal{O};\mathcal{E} \cup E} \ \textsc{Concat-Split}$$

Fig. 4.3: Auxiliary transformations applied by the compiler on dataflow programs to enable the parallelization transformation.

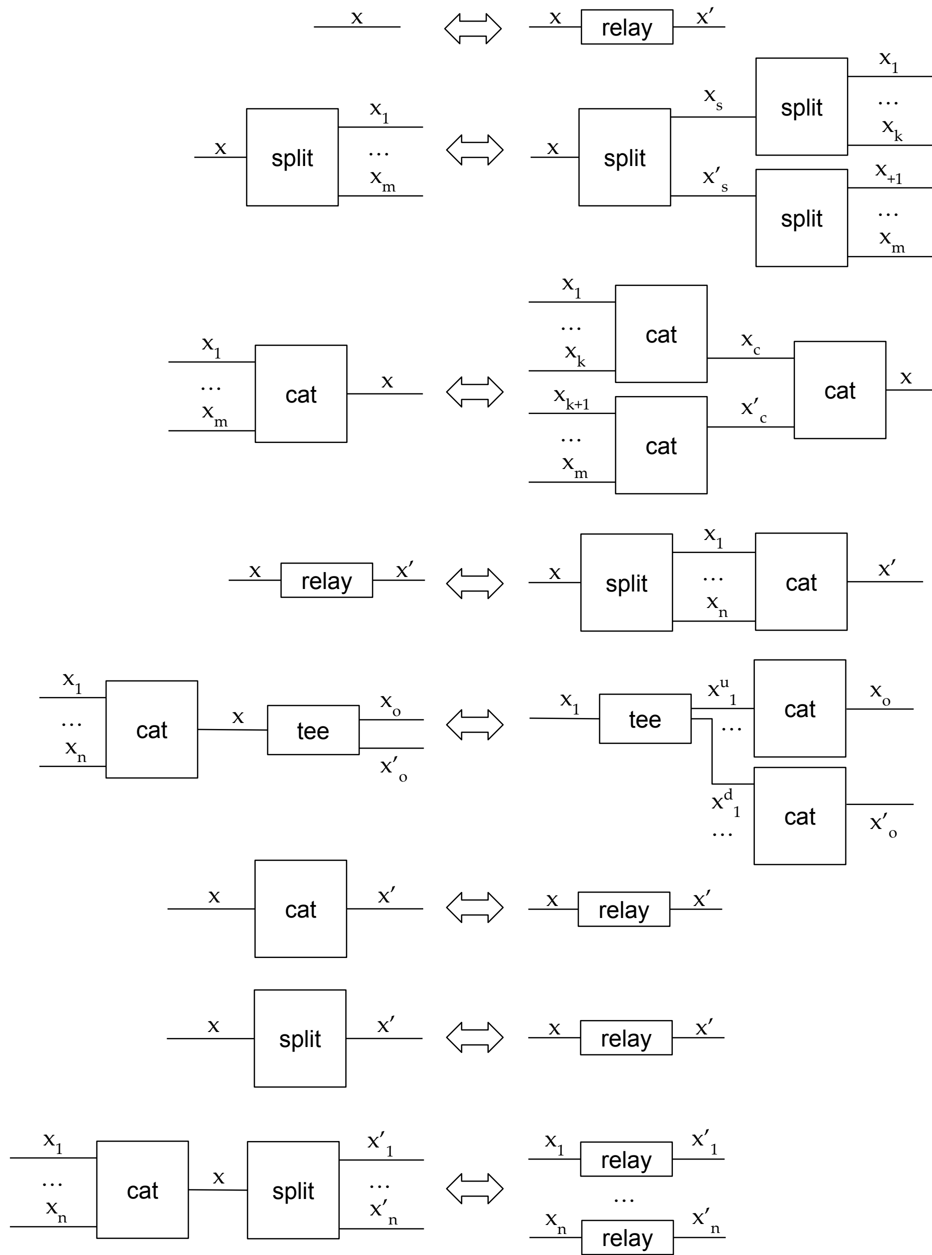


Fig. 4.4: Visualization of auxiliary transformations applied by the compiler on dataflow programs.

cat can be converted into relays, if the input arity of cat is the same as output arity of split. The first seven transformations can be performed both ways, while the last transformations can only be applied from left to right; applying it from right to left would require carefully selecting which edges to apply it to, to not create invalid and circular programs.

4.5.2. Data Parallelism and Transformations

The dataflow model exposes task parallelism as each different node can execute independently—only communicating with the other nodes through their communication channels. In addition to that, it is possible to achieve data parallelism by executing some nodes in parallel by partitioning part of their input.

Sequential Consumption Nodes: We are interested in nodes that produce a single output and consume their main inputs in sequence (one after the other when they are depleted), after an initialization phase of consumption of configuration inputs. There are several examples of shell commands that correspond to such nodes, e.g. `grep`, `sort`, `grep -f`, and `sha1sum`. Let f be such a node, where $x' = f(x_1, \ldots, x_{n+m})$ and $x_1, x_2, \ldots, x_n$ represent the configuration inputs and $x_{n+1}, \ldots, x_{n+m}$ represent the sequential consumption inputs. Without loss of generality we assume that the configuration inputs are the first inputs of a node and the rest are sequentially consumed.

The consumption order of such a command is shown below:

$$\mathsf{choice}_f(\overline{v}) = \begin{cases} \{i \,:\, i \leq n \wedge \neg\,\mathrm{closed}(v_i)\} \text{ if } \exists i \leq n, \neg\mathrm{closed}(v_i) \\ \{i \,:\, \forall j < i, \mathrm{closed}(v_j)\} \text{ otherwise} \end{cases}$$

If we know that a command f satisfies the above property we can represent it as $f(x_1, \ldots, x_n, x_{n+1} \cdot \ldots)$, essentially concatenating its sequential inputs into a single input.

Data Parallel Nodes: We now shift our focus to a subset of the sequential consumption nodes, namely those that can be executed in a data parallel fashion by splitting their inputs. These are nodes that can be broken down in a parallel *map* f_m and an associative *aggregate* f_r. Formally, a data parallel node has to satisfy the following (for simplicity we show the requirement for a single configuration input v_c):

$$f(v_c, v_1 \cdot \ldots \cdot v_n) = f_r(v_c, f_m(v_c, v_1), f_m(v_c, v_2), \ldots, f_m(v_c, v_n)).$$

$$\frac{\begin{array}{c} x_1^c, \ldots, x_n^c, x_{n+1}^c, x_1^m, \ldots, x_n^m \notin \mathcal{I}; \mathcal{O}; \mathcal{E} \\ \mathrm{dp}(f, f_m, f_r) \\ E = \left\{\mathsf{Node}(\mathsf{tee}; x_c; x_1^c, \ldots, x_n^c, x_{n+1}^c)\right\} \cup \\ \left\{\mathsf{Node}(f_m; x_1^c, x_i; x_i^m) : \forall i \in \{1 \ldots n\}\right\} \cup \left\{\mathsf{Node}(f_r; x_{n+1}^c, x_1^m, \ldots, x_n^m; x')\right\} \end{array}}{\mathcal{I}; \mathcal{O}; \mathcal{E} \cup \left\{\mathsf{Node}(\mathsf{cat}; x_1, \ldots, x_n; x), \mathsf{Node}(f; x_c, x; x')\right\} \Longrightarrow \mathcal{I}; \mathcal{O}; \mathcal{E} \cup E} \text{ PARALLEL}$$

Fig. 4.5: Parallelization transformation applied by the compiler on the dataflow program to expose available data parallelism.

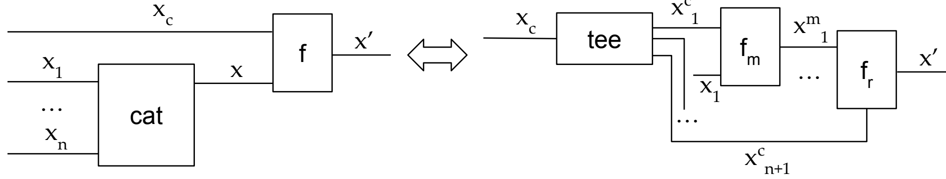


Fig. 4.6: Visualization of parallelization transformation applied by the compiler on dataflow programs.

Note that the above property does not depend on a specific input "split", and can support splitting the input in arbitrary points, *e.g.*, setting v_1 to contain all input and all the rest ($v_2, \ldots$) to contain ϵ. This is why the transformations described later are correct even though the characterizatin of the split auxiliary function is nondeterministic. We denote data parallel nodes as $\mathrm{dp}(f, f_m, f_r)$ Example of such a node that satisfies this property is the `sort` command, where $f_m =$ `sort` and $f_r =$ `sort -m`.

An important observation is that a subset of all data parallel nodes are trivially parallelizable, meaning that $f_m = f$ and $f_r = \mathsf{cat}$.

We can now define a transformation on any data parallel node f, that replaces it with a map followed by an aggregate. This transformation is described in Figure 4.5 and depicted in Figure 4.6. Essentially, all the sequential consumption inputs (that are concatenated using cat) are given to different f_m nodes the outputs of which are then aggregated using f_r while preserving the input order. Note that the configuration inputs have to be duplicated using `tee` to ensure that all parallel f_m and f_r instances will be able to read them in case they are pipes and not files on disk.

Using the auxiliary transformations—by adding a `split` followed by `cat` before a data parallel node, we can

always parallelize them using the parallelization transformation.

Correctness of Transformations: We now prove a series of statements regarding the correctness of our transformations after describing what it means for two programs to be equivalent. Our notion of equivalence only refers to executions that terminate, akin to partial correctness, because out transformations focus on improving the performance of batch computation scripts, assuming that the user only observes the output of a script once it is done executing. Intuitively, dataflow programs are guaranteed to terminate because (1) they do not have cycles; (2) all variables are reachable from the input variables; and (3) all of their nodes are assumed to terminate, meaning that if their inputs are closed with $\perp$, they will eventually stop producing output and close their outputs with $\perp$.

Program Equivalence: Let $P = \mathcal{I}; \mathcal{O}; \mathcal{E}$ and $P' = \mathcal{I}'; \mathcal{O}'; \mathcal{E}'$ be two dataflow programs, where $\mathcal{I} = \langle x^i_1, \ldots, x^i_n \rangle$, $\mathcal{I}' = \langle y^i_1, \ldots, y^i_n \rangle$, $\mathcal{O} = \langle x^o_1, \ldots, x^o_m \rangle$, and $\mathcal{O}' = \langle y^o_1, \ldots, y^o_m \rangle$. We say that two programs are equivalent if and only if given any executions $P \vdash \Gamma_i, \sigma_i \rightsquigarrow^* \Gamma, \sigma$ and $P' \vdash \Gamma'_i, \sigma'_i \rightsquigarrow^* \Gamma', \sigma'$ that terminate, *i.e.*, $\Gamma, \sigma, \Gamma', \sigma'$ are all closed, and given that all initial input values are equal, *i.e.*, for all $k \in [1, n]$. $\Gamma_i(x^i_k) = \Gamma'_i(y^i_k)$, then the values of output variables are the same, *i.e.*, for all $k \in [1, m]$. $\Gamma(x^o_k) = \Gamma'(y^o_k)$.

Theorem 3. *Let $P = \mathcal{I}; \mathcal{O}; \mathcal{E} \cup E$ and $p' = \langle \mathcal{I}; \mathcal{O}; \mathcal{E} \cup E' \rangle$ be two dataflow programs. Let $\mathcal{S}_i$ be the set of input variables in node set E (variables read in E but not assigned inside E). Let $\mathcal{S}_o$ be the set of output variables in the node set E (variables assigned in E but not read inside E). Let $\mathcal{S}'_i$, $\mathcal{S}'_o$ be the input variables and output variables of E'. We assume $\mathcal{S}_i = \mathcal{S}'_i$ and $\mathcal{S}_o = \mathcal{S}'_o$. If $\mathcal{S}_i; \mathcal{S}_o; E$ is equivalent to $\mathcal{S}'_i; \mathcal{S}'_o; E'$, then program P is equivalent to P'.*

Proof. We want to show that for any executions $P \vdash \Gamma_i, \sigma_i \rightsquigarrow^* \Gamma, \sigma$ and $P' \vdash \Gamma'_i, \sigma'_i \rightsquigarrow^* \Gamma', \sigma'$ that terminate, $\Gamma(x^o_k) = \Gamma'(y^o_k)$ holds for all $k \in [1, m]$ as long as all initial input values are equal, *i.e.*, for all $k \in [1, n]$. $\Gamma_i(x^i_k) = \Gamma'_i(y^i_k)$.

First, we know that $\Gamma(x) = \Gamma'(x)$ for any x in $\mathcal{S}_i$, as the dataflow graphs are acyclic and the subgraphs which compute variables in $\mathcal{S}_i$ are the same in both P and P', and the outputs in terminated programs are only determined by the nodes and inputs (according to Theorem 2).

Given that, and since $\mathcal{S}_i; \mathcal{S}_o; E$ is equivalent to $\mathcal{S}_i; \mathcal{S}_o; E'$, we know that $\Gamma(x) = \Gamma(x')$ for all x in $\mathcal{S}_o$.

Using Theorem 2 we can now show that all the rest of the variables (and therefore also output variables x in $\mathcal{O}$) have equal mappings, *i.e.*, $\Gamma(x) = \Gamma'(x)$.

Therefore, both P and P' are equivalent. □

Theorem 4. *The transformations presented in Figure 4.3 and Figure 4.5 preserve program equivalence.*

Proof. We use Theorem 3 to only show equivalences for the transformed subgraphs. The equivalence for all transformations then follows directly from the semantics of the special nodes cat, relay, split, tee, and the properties of f_m and f_r for data parallel commands f. □

4.6. Related work

Dataflow Graph Models: Graph models of computation where nodes represent units of computation and edges represent FIFO communication channels have been studied extensively [47, 98, 107, 105, 93, 94]. ODFM sits somewhere between Kahn Process Networks [93, 94] (KPN), the model of computation adopted by UNIX pipes, and Synchronous Dataflow [107, 105] (SDF). A key difference between ODFM and SDF is that ODFM does not assume fixed item rates—a property used by SDF for efficient scheduling determined at compile-time. Two differences between ODFM from KPNs is that (i) ODFM does not allow cycles, and (ii) ODFM exposes information about the input consumption order of each node. This order provides enough information at compile time to perform parallelizing transformations while also enabling translation of the dataflow back to a UNIX shell script.

Systems for batch [46, 129, 186], stream [67, 168, 113], and signal processing [107, 33] provide dataflow-based abstractions. These abstractions are different from ODFM which operates on the UNIX shell, an existing language with its own peculiarities that have guided the design of the model.

One technique for retrofitting order over unordered streaming primitives such as sharding and shuffling is to extend the types of elements using tagging [21, 181, 20]. This technique would not work in the UNIX shell, because (1) commands are black boxes operating on stream elements in unconstrained ways (but in known

order), and (2) because data streams exchanged between commands contain flat strings, without support for additional metadata extensions, and thus no obvious way to augment elements with tags. ODFM instead captures ordering on the edges of the dataflow graph, and leverages the consumption order of nodes (the choice function) in the graph to orchestrate execution appropriately.

Synchronous languages [104, 147, 29, 115] model stream graphs as circuits where nodes are state machines and edges are wires that carry a single value. Lustre [147] is based on a dataflow model that is similar to ours, but its focus is different as it is not intended for exploiting data-parallelism.

Semantics and Transformations: Prior work proposes semantics for streaming extensions to relational query languages based on dataflow [109, 19]. In contrast to our work, it focuses on transformations of time-varying relations.

More recently, there has been significant work on the correct parallelization of distributed streaming applications by proposing sound optimizations and compilation techniques [81, 154], type systems [114], and differential testing [96]. These efforts aim at producing a parallel implementation of a dataflow streaming computation using techniques that do not take into account the order of consumption of each node—preventing them from being applicable in our setting.

Recent work proposes a semantic framework for stream processing that uses monoids to capture the type of data streams [112]. That work mostly focuses on generality of expression, showing that several already proposed programming models can be expressed on top of it. It also touches upon soundness proofs of optimizations using algebraic reasoning, which is similar to our approach.

Divide and Conquer Decomposition: Prior work has shown the possibility of decomposing programs or program fragments using divide-and-conquer techniques [152, 56, 57, 157]. The majority of that work focuses on parallelizing special constructs—*e.g.*, loops, matrices, and arrays—rather than stream-oriented primitives. Techniques for automated synthesis of MapReduce-style distributed programs [157] can be of significant aid for individual commands. In some cases [56, 57], the map phase is augmented to maintain additional metadata used by the reducer phase. These techniques complement our work, since they can be used to derive aggregators and the parallelizability properties of yet unknown shell commands, making them

possible to capture in our model.

Parallel Shell Scripting: Tools exposing parallelism on modern UNIXes such as `qsub` [63], SLURM [184], AMFS [187] and GNU `parallel` [164] are predicated upon explicit and careful orchestration from their users. Similarly, several shells [50, 117, 180, 159] add primitives for non-linear pipe topologies—some of which target parallelism. Here too, however, users are expected to manually rewrite scripts to exploit these new primitives without jeopardizing correctness.

POSIX Shell Semantics: Our work depends on Smoosh, an effort focused on formalizing the semantics of the POSIX shell [71]. Smoosh focuses on POSIX semantics, whereas our work introduces a novel dataflow model in order to transform programs and prove the correctness of parallelization transformations on them. One of the Smoosh authors has also argued for making concurrency explicit via shell constructs [69]. This is different from our work, since it focuses on the capabilities of the shell language as an orchestration language, and does not deal with the data parallelism of pipelines.

Parallel Userspace Environments: By focusing on simplifying the development of distributed programs, a plethora of environments inadvertently assist in the construction of parallel software. Such systems [136, 128, 24], languages [177, 155, 99], or system-language hybrids [140, 175, 52] hide many of the challenges of dealing with concurrency as long as developers leverage the provided abstractions—which are strongly coupled to the underlying operating or runtime system. Even when these efforts are shell-oriented, such as Plan9's `rc`, they are backward-incompatible with the UNIX shell, and often focus primarily on hiding the existence of a network rather than on modelling data parallelism.

4.7. Discussion

Directly accessing the IR in the implementation: Our implementation currently allows manually developing programs in the ODFM intermediate representation. However, this interface is not that convenient to use as an end-user since it requires manually instantiating each node of the graph with the necessary command metadata, *e.g.*, inputs and outputs. It would be interesting future work to design different frontends that interface with this IR. For example, a frontend compiler from the language proposed by `dgsh` [159]; a shell that supports extended syntax for creating DAG pipelines. The IR could also act as an interface for different backends, for example one that implements ODFM in a distributed setting.

Parallel Script Debugging: Debugging standard shell pipelines can be hard and it usually requires several iterations of trial and error until the user gets the script right. Our approach does not make the debugging experience any worse, as the system produces as output a parallel shell script, which can be inspected and modified like any standard shell script (as seen in §4.3). For example, a user could debug a script by removing a few stages of the parallel pipeline, or redirecting some intermediate outputs to permanent files for inspection. This is possible because of the expressiveness of ODFM and the existence of a bidirectional transformation from dataflow programs to shell scripts, which allows the compiler to simply use a standard shell such as `bash` as its backend.

An approach that is particularly helpful, and which we have used ourselves, is to ask the compiler to add a relay node between every two nodes of the graph and then instantiate this relay node with an identity command that duplicates its input to its output and also a log file.

```
tee $LOG < $IN > $OUT
```

This allows for stream introspection without affecting the behavior of the pipeline, facilitating debugging since the user can inspect all intermediate outputs at once.

Stream Finiteness and Extensions: In our current model, parallelism is achieved by partitioning the finite stream, processing the partitions in parallel, and merging the results. Therefore, our model cannot support nonterminating computations over infinite data streams. All of the data processing scripts that we have encountered conform to this model and are terminating.

CHAPTER 5

Specification framework

Material from this chapter was previously published as "Nikos Vasilakis, Konstantinos Kallas, Konstantinos Mamouras, Achilles Benetopoulos, and Lazar Cvetković. PaSh: Light-Touch Data-Parallel Shell Processing. In Proceedings of the Sixteenth European Conference on Computer Systems, EuroSys '21, page 49-66, New York, NY, USA, 2021. Association for Computing Machinery." [173]. Nikos Vasilakis and I were the primary authors of this paper and contributed equally to all parts of the work; the rest of the coauthors contributed with the development of some command specifications and runtime components, as well as with the system evaluation.

5.1. Introduction

A significant challenge when trying to analyze, optimize, and transform shell scripts, is that they compose arbitrary commands, written in arbitrary languages, for which we might not even have access to their source code. This makes it extremely hard to reason about the script behavior, since it depends on these black-box arbitrary commands.

To address this challenge, we propose a command specification framework that can be used by analysis and optimization systems for shell scripts to reason about commands. The specification for each command can be written by an expert and can be independently verified, thus allowing system designers to focus on their task without having to implement cross language analyses to reason about each individual command.

Running Example: Weather Analysis

Suppose an environmental scientist wants to get a quick sense of trends in the maximum temperature across the U.S. over the past five years. As the National Oceanic and Atmospheric Administration (NOAA) has made historic temperature data publicly available [132], answering this question is only a matter of a simple data-processing pipeline.

Fig. 5.1's script starts by pulling the yearly index files and filtering out URLs that are not part of the compressed dataset. It then downloads and decompresses each file in the remaining set, extracts the values that

```
base="ftp://ftp.ncdc.noaa.gov/pub/data/noaa";
for y in {2015..2019}; do
 curl $base/$y | grep gz | tr -s" " | cut -d" " -f9 |
 sed "s;^;$base/$y/;" | xargs -n 1 curl -s | gunzip |
 cut -c 89-92 | grep -iv 999 | sort -rn | head -n 1 |
 sed "s/^/Maximum temperature for $y is: /"
done
```

Fig. 5.1: Calculating maximum temperatures per year. The script downloads daily temperatures recorded across the U.S. for the years 2015–2019 and extracts the maximum for every year.

indicate the temperature, and filters out bogus inputs marked as 999. It then calculates the maximum yearly temperature by sorting the values and picking the top element. Finally, it matches each maximum value with the appropriate year in order to print the result. The effort expended writing this script is low: its data-processing core amounts to 12 stages and, when expressed as a single line, is only 165 characters long. This program is no toy: a Java program implementing only the last four stages takes 137 LoC [182, §2.1]. To enable such a succinct program composition, UNIX incorporates several features.

UNIX provides an environment for composing commands written in any language. Many of these commands come with the system—*e.g.*, ones defined by the POSIX standard or ones part of the GNU Coreutils—whereas others are available as add-ons. The fact that commands are developed in a variety of languages—including shell scripts—provides users with significant flexibility. For example, one could replace `sort` and `head` with `./avg.py` to get the average rather than the maximum—the pipeline still works, as long as `./avg.py` conforms to the interface outlined earlier.

Command options and flags, used pervasively in UNIX, are configuration options that the command's developer has decided to expose to its users to improve the command's general applicability. For example, by omitting `sort`'s `-r` flag that enables reverse sorting, the user can easily get the minimum temperature. The shell does not have any visibility into these flags; after it expands special characters such as `~` and `*`, it leaves parsing and evaluation entirely up to individual commands.

While these features aid development-effort economy through powerful program composition, they complicate shell script reasoning and parallelization.

Challenge: In contrast to restricted programming frameworks that enable parallelization by supporting a few carefully-designed primitives [168, 37, 46, 186], the UNIX shell provides a wide variety of composable

commands. To be parallelized, each command may require special analysis and treatment—*e.g.*, exposing data parallelism in Fig. 5.1's `tr` or `sort` would require splitting their inputs, running them on each partial input, and then merging the partial results.[5] Automating such an analysis is infeasible, as individual commands are black boxes written in a variety of programming languages and models. Manual analysis is also challenging, due to the sheer number of commands and the many flags that affect their behavior—*e.g.*, Fig. 5.1's program invokes `cut` with two separate sets of flags.

Solution: We address this challenge as follows. To understand standard commands available in any shell, we groups POSIX and GNU commands into a small but well-defined set of *parallelizability classes* (§5.2). Rather than describing a command's full observable behavior, these classes focus on information that is important for data parallelism. To allow other commands to use its transformations, we define a light *specification framework* (also called annotation language in this dissertation) for describing a command's parallelizability class (§5.3). Specifications are expressed once per command rather than once per script and are aimed towards command developers rather than its users, so that they can quickly and easily capture the characteristics of the commands they develop. The specification framework described in this chapter is used to determine the command predicate and relations described in Chapter 4.

In summary, this dissertation chapter makes the following contributions:

- **UNIX Command Parallelizability:** It studies the parallelizability of a wide variety of shell commands in the POSIX and GNU Coreutils command sets (§5.2)
- **Command Specification Framework:** Based on this study, it introduces a lightweight specification framework for commands that are executable in a data-parallel manner (§5.3)

5.2. Parallelizability of Standard Libraries

Broadly speaking, shell commands can be split into four major classes with respect to their parallelization characteristics, depending on what kind of state they mutate and access when processing their input (Tab.5.1). These classes are ordered in ascending difficulty (or impossibility) of parallelization. In this order, classes can be thought of as subsets of the next—*e.g.*, all stateless commands are pure—meaning that the synchroniza-

[5] Commands such as `sort` may have *ad hoc* flags such as `--parallel`, which do not compose across commands and may risk breaking correctness or not exploiting performance potential (§6.5.5).

Tab. 5.1: Parallelizability Classes. Broadly, UNIX commands can be grouped into four classes according to their parallelizability properties.

Class	Key	Examples	Coreutils	POSIX
Stateless	Ⓢ	`tr`, `cat`, `grep`	13 (12.5%)	19 (12.7%)
Parallelizable Pure	Ⓟ	`sort`, `wc`, `head`	17 (16.3%)	13 (8.7%)
Non-parallelizable Pure	Ⓝ	`sha1sum`	13 (12.5%)	11 (7.3%)
Side-effectful	Ⓔ	`env`, `cp`, `whoami`	61 (58.6%)	105 (70.4%)

tion mechanisms required for any superclass would work with its subclass. Commands can change classes depending on their flags, which are discussed later (§5.3).

Stateless Commands: The first class, Ⓢ, contains commands that operate on individual line elements of their input, without maintaining state across lines. These are commands that can be expressed as a purely functional *map* or *filter*—*e.g.*, `grep` filters out individual lines and `basename` removes a path prefix from a string. They may produce multiple elements—*e.g.*, `tr` may insert NL tokens—but always return empty output for empty input. Workloads that use only stateless commands are trivial to parallelize: they do not require any synchronization to maintain correctness, nor caution about where to split inputs.

The choice of line as the data element strikes a convenient balance between coarse-grained (files) and fine-grained (characters) separation while staying aligned with UNIX's core abstractions. This choice can affect the allocation of commands in Ⓢ, as many of its commands (about 1/3) are stateless *within* a stream element—*e.g.*, `tr` transliterates characters within a line, one at a time—enabling further parallelization by splitting individual lines. This feature may seem of limited use, as these commands are computationally inexpensive, precisely due to their narrow focus. However, it may be useful for cases with very long lines such as the `.fastq` format used in bioinformatics.

Parallelizable Pure Commands: The second class, Ⓟ, contains commands that respect functional purity—*i.e.*, same outputs for same inputs—but maintain internal state across their entire pass. The details of this state and its propagation during element processing affect their parallelizability characteristics. Some commands are easy to parallelize, because they maintain trivial state and are commutative—*e.g.*, `wc` simply maintains a counter. Other commands, such as `sort`, maintain more complex invariants that have to be taken into account when merging partial results.

Often these commands do not operate in an online fashion, but need to block until the end of a stream. A

typical example of this is `sort`, which cannot start emitting results before the last input element has been consumed. Such constraints affect task parallelism, but not data parallelism: `sort` can be parallelized using divide-and-conquer techniques—*i.e.*, by encoding it as a group of (parallel) *map* functions followed by an *aggregate* that merges the results.

Non-parallelizable Pure Commands: The third class, Ⓝ, contains commands that, while purely functional, cannot be parallelized within a single data stream. This is because their internal state depends on prior state in non-trivial ways over the same pass. For example, hashing commands such as `sha1sum` maintain complex state that has to be updated sequentially. If parallelized on a single input, each stage would need to wait on the results of all previous stages, foregoing any parallelism benefits.

It is worth noting that while these commands are not parallelizable at the granularity of a single input, they are still parallelizable across different inputs. For example, a web crawler involving hashing to compare individual pages would allow `sha1sum` to proceed in parallel for different pages.

Side-effectful Commands: The last class, Ⓔ, contains commands that have side-effects that cannot be captured precisely—for example, updating environment variables, interacting with the filesystem in unpredictable or difficult to describe ways, and accessing the network. Such commands are not parallelizable without finer-grained concurrency control mechanisms and therefore are not the main focus of this work. This is the largest class, for two main reasons. First, it includes commands related to the file-system—a central abstraction of the UNIX design and philosophy [150]. In fact, UNIX uses the file-system as a proxy to several file-unrelated operations such as access control and device driving. Second, this class contains commands that do not consume input or do not produce output—and thus are not amenable to data parallelism. For example, `date`, `uname`, and `finger` are all commands interfacing with kernel- or hardware-generated information and do not consume any input from user programs or files.

5.3. Extensibility Framework

To address the challenge of a language-agnostic environment, we develop a specification framework that can describe key details about commands and their parallelizability. The framework is extensible and contains two components: an annotation language, and an interface for developing parallel command aggregators. The framework can be used by developers of new commands and maintainers of existing commands. The latter

group can express additions or changes to the command's implementation or interface, which is important as commands are maintained or extended over long periods of time.

The extensibility framework is expected to be used by individuals who understand the commands and their parallelizability properties. The framework could be used as a foundation for crowdsourcing the annotation effort, for testing annotation records, and for generating command aggregators.

Key Concerns: The annotations of our framework focus on three crucial concerns about a command: (C1) its parallelizability class, (C2) its inputs, outputs, and the characteristics of its input consumption, and (C3) how flags affect its class, inputs, and outputs. The first concern was discussed extensively in the previous section; we now turn to the latter two.

Manipulating a shell script in its original form to expose parallelism is challenging as each command has a different interface. Some commands read from standard input, while others read from input files. Ordering here is important, as a command may read several inputs in a predefined input order. For example, `grep "foo" f1 - f2` first reads from `f1`, then shifts to its standard input, and finally reads `f2`.

Additionally, commands expose flags or options for allowing users to control their execution. Such flags may directly affect a command's parallelizability classification as well as the order in which it reads its inputs. For example, `cat` defaults to Ⓢ, but with `-n` it jumps into Ⓟ because it has to keep track of a counter and print it along with each line.

To address all these concerns, we introduce an annotation language encoding first-order logic predicates. The language allows specifying the aforementioned information, *i.e.*, correspondence of arguments to inputs and outputs and the effects of flags. Annotations assign one of the four parallelizability class as a default class, subsequently refined by the set of flags the command exposes. Additionally, for commands in Ⓢ and Ⓟ, the language captures how a command's arguments, standard input, and standard output correspond to its inputs and outputs. Annotations in these classes can also express ordering information about these inputs—effectively lifting commands into a more convenient representation where they only communicate with their environment through a list of input and output files.

The complete annotation language currently contains 8 operators, one of which supports regular expressions. It was used to annotate 47 commands, totaling 708 lines of JSON—an effort that took about 3–4 hours. Annotation records are by default conservative so as to not jeopardize correctness, but can be incrementally refined to capture parallelizability when using increasingly complex combinations of flags. The language is extensible with more operators (as long as the developer defines their semantics); it also supports writing arbitrary Python code for commands whose properties are difficult to capture—*e.g.*, `xargs`, whose parallelizability class depends on the class of the command that it invokes.

Example Annotations: Two commands whose annotations sit at opposing ends of the complexity spectrum are `chmod` and `cut`. The fragment below shows the annotation for `chmod`.

```
{ "command": "chmod",
  "cases": [ { "predicate": "default",
               "class": "side-effectful" } ] }
```

Each annotation is a JSON record that contains the command name and a sequence of cases. Each case contains a predicate that matches on the arguments of the command invocation. It assigns a parallelizability class (C1) to a specific command instance, *i.e.*, the combination of its inputs-output consumption (C2) and its invocation arguments (C3). In this case, `chmod` is side-effectful, and thus the `"default"` predicate of its single `cases` value always matches—indicating the presence of side-effects.

The annotation for the command `cut`, which can be configured to project parts of its input, *e.g.*, a field, is significantly more complex and has two cases. Each of the cases consists of a predicate on `cut`'s arguments and an assignment of its parallelizability class, inputs, and outputs as described above.

```
{ "command": "cut",
  "cases": [
    { "predicate": {
        "operator": "or",
        "operands": [
          { "operator": "val_opt_eq",
            "operands": [ "-d", "\n" ] },
          { "operator": "exists",
```

```
          "operands": [ "-z" ] }
        ]
      },
      "class": "pure",
      "inputs": [ "args[:]" ],
      "outputs": [ "stdout" ]
    },
    { "predicate": "default",
      "class": "stateless",
      "inputs": [ "args[:]" ],
      "outputs": [ "stdout" ]
    }
  ],
  "options": [ "stdin-hyphen", "empty-args-stdin" ],
  "short-long": [
    { "short": "-d", "long": "--delimiter" },
    { "short": "-z", "long": "--zero-terminated" }
  ]
}
```

This predicate indicates that if `cut` is called with `-z` as an argument, then it is in Ⓝ, *i.e.*, it only interacts with the environment by writing to a file (its `stdout`) but cannot be parallelized. This is because `-z` forces `cut` to delimit lines with `NUL` instead of newline, meaning that we cannot parallelize it by splitting its input in the line boundaries. The case also indicates that `cut` reads its inputs from its non-option arguments.

Experienced readers will notice that `cut` reads its input from its `stdin` if no file argument is present. This is expressed in the `"options"` part of `cut`'s annotation, shown below:

```
{ "command": "cut",
  "cases": [ ... ],
  "options": [ "empty-args-stdin",
               "stdin-hyphen" ] }
```

Option `"empty-args-stdin"` indicates that if non-option arguments are empty, then the command reads

from its `stdin`. Furthermore, option `"stdin-hyphen"` indicates that a non-option argument that is just a dash - represents the stdin.

Custom Aggregators: For commands in Ⓢ, the annotations are enough to enable parallelization: commands are applied to parts of their input in parallel, and their outputs are simply concatenated.

To support the parallelization of arbitrary commands in Ⓟ, our framework allows supplying custom *map* and *aggregate* functions. In line with the UNIX philosophy, these functions can be written in any language as long as they conform to a few invariants: (i) *map* is in Ⓢ and *aggregate* is in Ⓟ, (ii) *map* can consume (or extend) the output of the original command and *aggregate* can consume (and combine) the results of multiple *map* invocations, and (iii) their composition produces the same output as the original command.

Most commands only need an *aggregate* function, as the *map* function for many commands is the sequential command itself. We define a set of aggregators for many POSIX and GNU commands in Ⓟ. This set doubles as both an "aggregator standard library" and an exemplar for community efforts tackling other commands. Below is the Python code for one of the simplest *aggregate* functions, the one for `wc`:

```python
#!/usr/bin/python
import sys, os, functools, utils

def parseLine(s):
  return map(int, s.split())

def emitLine(t):
  f = lambda e: str(e).rjust(utils.PAD_LEN, ' ')
  return [" ".join(map(f, t))]

def agg(a, b):
  # print(a, b)
  if not a:
    return b
  az = parseLine(a[0])
  bz = parseLine(b[0])
  return emitLine([ (i+j) for (i,j) in zip(az, bz) ])
```

```
utils.help()
res = functools.reduce(agg, utils.read_all(), [])
utils.out("".join(res))
```

The core of the aggregator, function `agg`, takes two input streams as its arguments. The `reduce` function lifts the aggregator to arity *n* to support an arbitrary number of parallel *map* commands. This lifting allows developers to think of aggregators in terms of two inputs, but generalize them to operate on many inputs. Utility functions such as `read` and `help`, common across our aggregator library, deal with error handling when reading multiple file descriptors, and offer a `-h` invocation flag that demonstrates the use of each aggregator.

The library currently contains over 20 aggregators, many of which are usable by more than one command or flag. For example, the aggregator shown above is shared among `wc`, `wc -lw`, `wc -lm`, *etc.*

CHAPTER 6

PaSh: Automatic parallelization of shell dataflow regions

Material from this chapter was previously published as "Nikos Vasilakis, Konstantinos Kallas, Konstantinos Mamouras, Achilles Benetopoulos, and Lazar Cvetković. PaSh: Light-Touch Data-Parallel Shell Processing. In Proceedings of the Sixteenth European Conference on Computer Systems, EuroSys '21, page 49-66, New York, NY, USA, 2021. Association for Computing Machinery." [173]. Nikos Vasilakis and I were the primary authors of this paper and contributed equally to all parts of the work; the rest of the coauthors contributed with the development of some command specifications and runtime components, as well as with the system evaluation.

6.1. Introduction

A key issue with shell scripts is their performance, and in particular the fact that they cannot adequately exploit multicore computing resources. To parallelize a script requires significant effort shared between two different programmer groups:

- *Command developers*, responsible for implementing individual commands such as `sort`, `uniq`, and `jq`. These developers usually work in a single programming language, leveraging its abstractions to provide parallelism whenever possible. As they have no visibility into the command's uses, they expose a plethora of ad-hoc command-specific flags such as `-t`, `--parallel`, `-p`, and `-j` [139, 119].

- *Shell users*, who use POSIX shell constructs to combine multiple such commands from many languages into their scripts and are thus left with only a few options for incorporating parallelism. One option is to use manual tools such as GNU `parallel` [164], `ts` [84], `qsub` [63], SLURM [184]; these tools are either command-unaware, and thus at risk of breaking program semantics, or too coarse-grained, and thus only capable of exploiting parallelism at the level of entire scripts rather than individual components. Another option is to use shell primitives (such as `&`, `wait`) to explicitly induce parallelism, at a cost of manual effort to split inputs, rewrite scripts, and orchestrate execution—an expensive and error-prone process. To top it off, all these options assume a good understanding of

parallelism; users with domain of expertise outside computing—from hobbyists to data analysts—are left without options.

This dissertation chapter presents PaSh, a system for parallelizing POSIX shell scripts that benefits both programmer groups, with emphasis on shell users. Command developers are given a set of abstractions, akin to lightweight type annotations, for expressing the parallelizability properties of their commands: rather than expressing a command's full observable behavior, these annotations focus primarily on its interaction with state. Shell users, on the other hand, are provided with full automation: PaSh analyzes their scripts and extracts latent parallelism. PaSh's transformations are conservative, in that they do not attempt to parallelize fragments that lack sufficient information—*i.e.*, at worst, PaSh will choose to not improve performance rather than risking breakage.

PaSh's transformations build on the dataflow model and the transformations described in Chapter 4 and its specification framework is the one described in Chapter 5. These components are tied together with PaSh's runtime component. Aware of the Unix philosophy and abstractions, it packs a small library of highly-optimized data aggregators as well as high-performance primitives for eager data splitting and merging. These address many practical challenges and were developed by uncovering several pathological situations, on a few of which we report.

We evaluate PaSh on 44 unmodified scripts including (i) a series of small scripts, ranging from classic Unix one-liners to modern data-processing pipelines, and (ii) two large and complex use cases for temperature analysis and web indexing. Speedups range between 0.89–61.1× (avg: 6.7×), with 39 out of 44 scripts seeing non-trivial speedups. PaSh's runtime primitives add to the base speedup extracted by PaSh's program transformations—*e.g.*, 8.83× over a base 5.93× average for 10 representative Unix one-liners. PaSh accelerates a large program for temperature analysis by 2.52×, parallelizing both the computation (12.31×) and the preprocessing (2.04×) fragment (*i.e.*, data download, extraction, and cleanup), the latter traditionally falling outside of the focus of conventional parallelization systems—even though it takes 75% of the total execution time.

The chapter is structured as follows. It starts by introducing the necessary background on shell scripting and

presenting an overview of PaSh (§6.2). Section 6.3 ties PaSh together with the dataflow model described in Chapter 4. Section 6.4 highlights PaSh's runtime component, discussing the correctness and performance challenges it addresses. Finally, PaSh is thoroughly evaluated on a variety of workloads (§6.5).

6.2. Background and overview

Let's look back at the weather analysis script in Figure 5.1, but with an extended focus on the whole script (instead of looking at individual commands). In addition to the features described in Section 5.1, the Unix shell also supports seamless composition, which is primarily achieved with pipes (`|`), a construct that allows for task-parallel execution of two commands by connecting them through a character stream. This stream contains contiguous character lines separated by newline characters (NL) delineating individual stream elements. For example, Fig 5.1's first `grep` outputs (file-name) elements containing `gz`, which are then consumed by `tr`. A special end-of-file (EOF) condition marks the end of a stream.

Different pipeline stages process data concurrently and possibly at different rates—*e.g.*, the second `curl` produces output at a significantly slower pace than the `grep` commands before and after it. The Unix kernel facilitates scheduling, communication, and synchronization behind the scenes.

While the shell's features aid development-effort economy through powerful program composition, they complicate parallelization, which even for simple scripts such as the one in Fig. 5.1 create several challenges.

Challenge: Composition: Another challenge is due to the language of the POSIX shell. First, the language contains constructs that enforce sequential execution: The sequential composition operator (`;`) in Fig. 5.1 indicates that the assignment to `base` must be completed before everything else. Moreover, the language semantics only exposes limited task-based parallelism in the form of constructs such as `&`. Even though Fig. 5.1's `for` focuses only on five years of data, `curl` still outputs thousands of lines per year; naive parallelization of each loop iteration will miss such opportunities. Any attempt to automate parallelization should be aware of the POSIX shell language, exposing latent data parallelism without modifying execution semantics.

Challenge: Implementation: On top of command and shell semantics, the broader Unix environment has its own set of quirks. Any attempt to orchestrate parallel execution will hit challenges related to task

```
mkfifo $t{0,1...}
curl $base/$y > $t0 & cat $t0 | split $t1 $t2 &
cat $t1 | grep gz > $t3 &
cat $t2 | grep gz > $t4 &
...
cat $t9 | sort -rn > $t11 & cat $t10 | sort -rn > $t12 &
cat $t11 | eager > $t13 & cat $t12 | eager > $t14 &
sort -mrn $t13 $t14 > $t15 &
cat $t15 | head -n1 > $out1 &
wait $! && get-pids | xargs -n 1 kill -SIGPIPE
```

Fig. 6.1: Output of `pash -width=2` for Fig. 5.1 (fragment). PASH orchestrates the parallel execution through named pipes, parallel operators, and custom runtime primitives—*e.g.*, `eager`, `split`, and `get-pids`.

parallelism, deadlock prevention, and runtime performance. For example, forked processes piping their combined results to Fig. 5.1's `head` may not receive a `PIPE` signal if `head` exits prior to opening all pipes. Moreover, several commands such as `sort` and `uniq` require specialized data aggregators in order to be correctly parallelized.

6.2.1. PASH Design Overview

At a high level, PASH takes as input a POSIX shell script like the one in Fig. 5.1 and outputs a new POSIX script that incorporates data parallelism. The degree of data parallelism sought by PASH is configurable using a `--width` parameter, whose default value is system-specific. Fig. 6.1 highlights a few fragments of the parallel script resulting from applying PASH with `--width=2` to the script of Fig. 5.1—resulting in 2 copies of {`grep`, `tr`, `cut`, *etc.*}.

PASH first identifies sections of the script that are potentially parallelizable, *i.e.*, lack synchronization and scheduling constraints, and converts them to dataflow graphs (DFGs). It then performs a series of DFG transformations that expose parallelism without breaking semantics, by expanding the DFG to the desired `width`. Finally, PASH converts these DFGs back to a shell script augmented with PASH-provided commands. The script is handed off to the user's original shell interpreter for execution. PASH addresses the aforementioned challenges as described below.

Composition: To maintain sequential semantics, PASH first analyzes a script to identify *dataflow regions* containing commands that are candidates for parallelization (§6.3.1). This analysis is guided by the script structure: some constructs expose parallelism (*e.g.*, `&`, `|`); others enforce synchronization (*e.g.*, `;`, `||`). PASH then converts each dataflow region to a *dataflow graph* (DFG) (§6.3.2), a flexible representation that enables

a series of local transformations to expose data parallelism, converting the graph into its parallel equivalent (§6.3.3). Further transformations compile the DFG back to a shell script that uses POSIX constructs to guide parallelism explicitly while aiming at preserving the semantics of the sequential program (§6.3.4).

Implementation: PaSh addresses several practical challenges through a set of constructs it provides—*i.e.*, modular components for augmenting command composition (§6.4). It also provides a small and efficient *aggregator library* targeting a large set of parallelizable commands. All these commands live in the `PATH` and are addressable by name, which means they can be used like (and by) any other commands.

6.3. Dataflow graph model

PaSh's core is an abstract dataflow graph (DFG) model (§6.3.2) used as the intermediate representation on which PaSh performs parallelism-exposing transformations. This core dataflow model is based on ODFM (introduced in Chapter 4), but included here for completeness. PaSh first lifts sections of the input script to the DFG representation (§6.3.1), then performs transformations to expose parallelism (up to the desired `--width`) (§6.3.3), and finally instantiates each DFG back to a parallel shell script (§6.3.4). A fundamental characteristic of PaSh's DFG is that it encodes the order in which a node reads its input streams (not just the order of input elements per stream), which in turn enables a set of graph transformations that can be iteratively applied to expose parallelization opportunities for Ⓢ and Ⓟ commands.

To the extent possible, this section is kept informal and intuitive. The full formalization of the dataflow model, the shell↔DFG bidirectional translations, and the parallelizing transformations, as well as their proof of correctness with respect to the script's sequential output, are all presented in Chapter 4.

6.3.1. Frontend: From a Sequential Script to DFGs

Dataflow Regions: In order to apply the graph transformations that expose data parallelism, PaSh first has to identify program sub-expressions that can be safely transformed to a dataflow graph, *i.e.*, sub-expressions that (i) do not impose any scheduling or synchronization constraints (*e.g.*, by using `;`), and (ii) take a set of files as inputs and produce a set of files as outputs. The search for these regions is guided by the shell language and the structure of a particular program. These contain information about (i) fragments that can be executed independently, and (ii) barriers that are natural synchronization points. Consider this code fragment (Fig. 6.2):

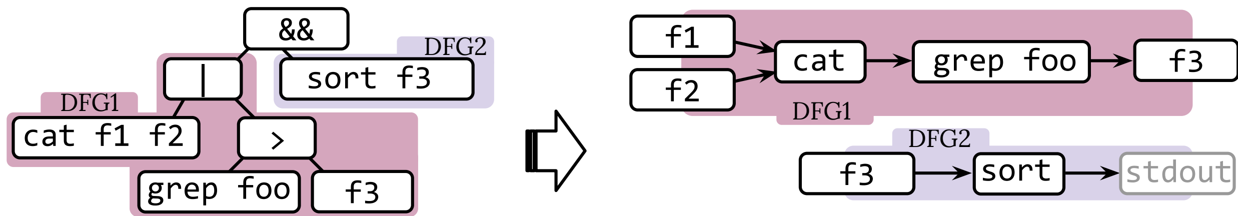


Fig. 6.2: From a script AST to DFGs. The AST on the left has two dataflow regions, each not extending beyond `&&` (*Cf.*§6.3.1). Identifiers `f1`, `f2`, and `f3` sit at the boundary of the DFG.

```
cat f1 f2 | grep "foo" > f3 && sort f3
```

The `cat` and `grep` commands execute independently (and concurrently) in the standard shell, but `sort` waits for their completion prior to start. Intuitively, dataflow regions correspond to sub-expressions of the program that would be allowed to execute independently by different processes in the POSIX standard [73]. Larger dataflow regions can be composed from smaller ones using the pipe operator (`|`) and the parallel-composition operator (`&`). Conversely, all other operators, including sequential composition (`;`) and logical operators (`&&`, `||`), represent barrier constructs that do not allow dataflow regions to expand beyond them.

Translation Pass: PASH's front-end performs a depth-first search on the AST of the given shell program. During this pass, it extends the dataflow regions bottom-up, translating their independent components to DFG nodes until a barrier construct is reached. All AST subtrees not translatable to DFGs are kept as they are. The output of the translation pass is the original AST where dataflow regions have been replaced with DFGs.

To identify opportunities for parallelization, the translation pass extracts each command's parallelizability class together with its inputs and outputs. To achieve this for each command, it searches all its available annotations (§5.3) and resorts to conservative defaults if none is found. If the command is in Ⓢ, Ⓟ, or Ⓝ, the translation pass initiates a dataflow region that is propagated up the tree.

Due to the highly dynamic nature of the shell, some information is not known to PASH at translation time. Examples of such information include the values of environment variables, unexpanded strings, and sub-shell constructs. For the sake of correctness, PASH takes a conservative approach and avoids parallelizing

nodes for which it has incomplete information. It will not attempt to parallelize sub-expressions for which the translation pass cannot infer that, *e.g.*, an environment variable passed as an argument to a command does not change its parallelizability class.

6.3.2. Dataflow Model Definitions

The two main shell abstractions are (i) data streams, *i.e.*, files or pipes, and (ii) commands, communicating through these streams.

Edges—Streams: Edges in the DFG represent streams, the basic data abstraction of the shell. They are used as communication channels between nodes in the graph, and as the input or output of the entire graph. For example, the edges in DFG1 of Figure 6.2 are the files `f1`, `f2`, and `f3`, as well as the unnamed pipe that connects `cat` and `grep`. We fix the data quantum to be character lines, *i.e.*, sequences of characters followed by the newline character,[6] so edges represent possibly unbounded sequences of lines. As seen above, an edge can either refer to a named file, an ephemeral pipe, or a UNIX FIFO used for interprocess communication. Edges that do not start from a node in the graph represent the graph inputs; edges that do not point to a node in the graph represent its outputs.

Nodes—Commands: A node of the graph represents a relation (to capture nondeterminism) from a possibly empty list of input streams to a list of output streams. This representation captures all the commands in the classes Ⓢ, Ⓟ, and Ⓝ, since they only interact with the environment by reading and writing to streams. We require that nodes are monotone, meaning that they cannot retract output once they have produced it. As an example, `cat`, `grep`, and `sort` are the nodes in the DFGs of Figure 6.2.

Streaming Commands: A large subset of the parallelizable Ⓢ and Ⓟ classes falls into the special category of streaming commands. These commands have two execution phases. First, they consume a (possibly empty) set of input streams that act as configuration. Then, they transition to the second phase where they consume the rest of their inputs sequentially, one element at a time, in the order dictated by the configuration phase, and produce a single output stream. The simplest example of a streaming command is `cat`, which has an empty first phase and then consumes its inputs in order, producing their concatenation as output. A more interesting example is `grep` invoked with `-f patterns.txt` as arguments; it first reads `patterns.txt` as

[6] This is a choice that is not baked into PASH's DFG model, which supports arbitrary data elements such as characters and words, but was made to simplify alignment with many UNIX commands.

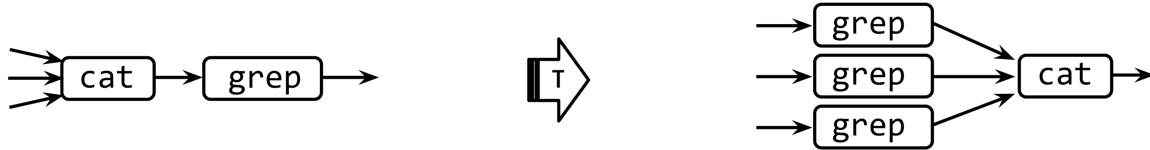


Fig. 6.3: Stateless parallelization transformation. The `cat` node is commuted with the stateless node to exploit available data parallelism.

its configuration input, identifying the patterns for which to search on its input, and then reads a line at a time from its standard input, stopping when it reaches EOF.

6.3.3. Graph Transformations

PASH defines a set of semantics-preserving graph transformations that act as parallelism-exposing optimizations. Both the domain and range of these transformations are graphs in PASH's DFG model; transformations can be composed arbitrarily and in any order. Before describing the different types of transformations, we formalize the intuition behind classes Ⓢ and Ⓟ described informally earlier (§5.2).

Stateless and Parallelizable Pure Commands: Stateless commands such as `tr` operate independently on individual lines of their input stream without maintaining any state (§5.2). To avoid referring to the internal command state, we can instead determine that a command is stateless if its output is the same if we "restart" it after it has read an arbitrary prefix of its input. If a command was stateful, then it would not produce the same output after the restart. Formally, a streaming command f is stateless if it commutes with the operation of concatenation on its streaming input, *i.e.*, it is a semigroup homomorphism:

$$\forall x, x', c, f(x \cdot x', c) = f(x, c) \cdot f(x', c)$$

In the above $x \cdot x'$ is the concatenation of the two parts of f's streaming input and c is the configuration input (which needs to be passed to both instances of f). The above equation means that applying the command f to a concatenation of two inputs x, x' produces the same output as applying f to each input x, x' separately, and concatenating the outputs. Note that we only focus on deterministic stateless commands and that is why

f is a function and not a relation in the above.

Pure commands such as `sort` and `wc` can also be parallelized, using divide-and-conquer parallelism. These commands can be applied independently on different segments of their inputs, aggregating their outputs to produce the final result. More formally, these pure commands f can be implemented as a combination of a function map and an associative function aggregate that satisfy the following equation:

$$\forall x, x', c, f(x \cdot x', c) = \text{aggregate}(\text{map}(x, c), \text{map}(x', c), c)$$

Parallelization Transformations: Based on these equations, we can define a parallelization transformation on a node $f \in Ⓢ$ whose streaming input is a concatenation, *i.e.*, produced using the command `cat`, of n input streams and is followed by a node f' (Fig. 6.3). The transformation replaces f with n new nodes, routing each of the n input streams to one of them, and commutes the `cat` node after them to concatenate their outputs and transfer them to f'. Formally:

$$f(x_1 \cdot x_2 \cdots x_n, s) \Rightarrow f(x_1, s) \cdot f(x_2, s) \cdots f(x_n, s)$$

The transformation can be extended straightforwardly to nodes $v \in Ⓟ$, implemented by a (map, aggregate) pair:

$$v(x_1 \cdot x_2 \cdots x_n, s) \Rightarrow$$
$$\text{aggregate}(\text{map}(x_1, s), \text{map}(x_2, s), \ldots \text{map}(x_n, s), s)$$

Both transformations can be shown to preserve the behavior of the original graph assuming that the pair (map, aggregate) meets the three invariants outlined earlier (§5.3) and the aforementioned equations hold.

Auxiliary Transformations: PASH also performs a set of auxiliary transformations t_{1-3} that are depicted in Fig. 6.4. If a node has many inputs, t_1 concatenates these inputs by inserting a `cat` node to enable the parallelization transformations. In cases where a parallelizable node has one input and is not preceded by a concatenation, t_2 inserts a `cat` node that is preceded by its inverse `split`, so that the concatenation can be

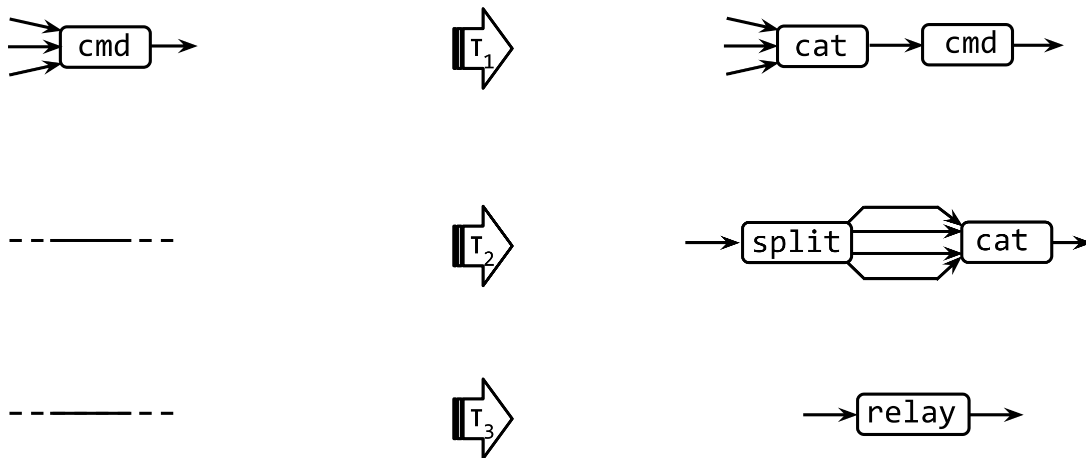


Fig. 6.4: Auxiliary transformations. These augment the DFG with `cat`, `split`, and `relay` nodes.

commuted with the node. Transformation t_3 inserts a relay node that performs the identity transformation. Relay nodes can be useful for performance improvements (§6.4), as well as for monitoring and debugging.

Degree of Parallelism: The degree of parallelism achieved by PASH is affected by the width of the final dataflow graph. The dataflow width corresponds, intuitively, to the number of data-parallel copies of each node of the sequential graph and thus the fanout of the `split` nodes that PASH introduces. The dataflow width is configured using the `--width` parameter, which can be chosen by the user depending on their script characteristics, input data, and target execution environment. By default, PASH assigns width to 2 if it is executing on a machine with 2-16 processors, and `floor(cpu_cores/8)` if it is executing on a machine with more than 16 processors. This is a conservative limit that achieves benefits due to parallelism but does not consume all system resources. It is not meant to be optimal, and as shown in our evaluation, different scripts achieve optimal speedup with different `--width` values, which indicates an interesting direction for future work.

6.3.4. Backend: From DFGs to a Parallel Shell Script

After applying transformations (§6.3.3), PASH translates all DFGs back into a shell script. Nodes of the graph are instantiated with the commands and flags they represent, and edges are instantiated as named pipes. A prologue in the script creates the necessary intermediate pipes, and a `trap` call takes care of cleaning up when the script aborts.

6.4. Runtime

This section describes technical challenges related to the execution of the resulting script and how they are addressed by PASH's custom runtime primitives.

Overcoming Laziness: The shell's evaluation strategy is unusually lazy, in that most commands and shell constructs consume their inputs only when they are ready to process more. Such laziness leads to CPU underutilization, as commands are often blocked when their consumers are not requesting any input. Consider the following fragment:

```
mkfifo t1 t2
grep "foo" f1 > t1 & grep "foo" f2 > t2 & cat t1 t2
```

The `cat` command will consume input from `t2` only after it completes reading from `t1`. As a result, the second `grep` will remain blocked until the first `grep` completes (Fig. 6.5a).

To solve this, one might be tempted to replace FIFOs with files, a central UNIX abstraction, simulating pipes of arbitrary buffering (Fig. 6.5b). Aside from severe performance implications, naive replacement can lead to subtle race conditions, as a consumer may reach EOF before a producer. Alternatively, consumers could wait for producers to complete before opening the file for reading (Fig. 6.5c); however, this would insert artificial barriers impeding task-based parallelism and wasting disk resources—that is, this approach allows for data parallelism to the detriment of task parallelism.

To address this challenge, PASH inserts and instantiates eager `relay` nodes at these points (Fig. 6.5d). These nodes feature tight multi-threaded loops that consume input eagerly while attempting to push data to the output stream, forcing upstream nodes to produce output when possible while also preserving task-based parallelism. In PASH's evaluation (§6.5), these primitives have the names presented in Fig. 6.5.

Splitting Challenges: To offer data parallelism, PASH needs to split an input data stream to multiple chunks operated upon in parallel. Such splitting is needed at least once at the beginning of a parallel fragment, and possibly every time within the parallel program when an *aggregate* function of a stage merges data into a single stream.

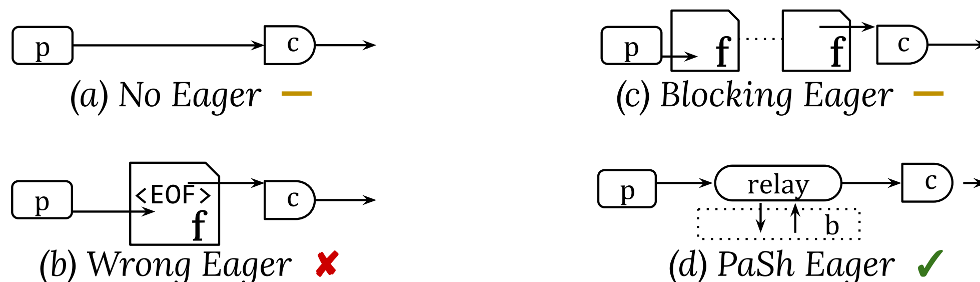


Fig. 6.5: Eager primitive. Addressing intermediary laziness is challenging: (a) FIFOs are blocking; (b) files alone introduce race conditions between producer/consumer; (c) files + `wait` inhibit task parallelism. Eager `relay` nodes (d) address the challenge while remaining within the PASH model.

To achieve this, PASH's transformations insert split nodes that correspond to a custom `split` command. For `split` to be effective, it needs to disperse its input uniformly. PASH does not do this in a round-robin fashion, as that would require augmenting the data stream with additional metadata to maintain FIFO ordering—a challenge for both performance and correctness. PASH instead splits chunks in-order, which necessitates knowledge of the input size beforehand and which is not always available. To address this challenge, PASH provides a `split` implementation that first consumes its complete input, counts its lines, and then splits it uniformly across the desired number of outputs. PASH also inserts eager `relay` nodes after all `split` outputs (except for the last one) to address laziness as described above.

Dangling FIFOs and Zombie Producers: Under normal operation, a command exits after it has produced and sent all its results to its output channel. If the channel is a pipe and its reader exits early, the command is notified to stop writing early. In UNIX, this notification is achieved by an out-of-band error mechanism: the operating system delivers a `PIPE` signal to the producer, notifying it that the pipe's consumer has exited. This handling is different from the error handling for other system calls and unusual compared to non-UNIX systems[7] primarily because pipes and pipelines are at the heart of UNIX. Unfortunately though, if a pipe has not been opened for writing yet, UNIX cannot signal this condition. Consider the following script:

```
mkfifo fifo1 fifo2
cat in1 > fifo1 & cat in2 > fifo2 &
cat fifo1 fifo2 | head -n 1 & wait
```

[7]For example, Windows indicates errors for `WriteFile` using its return code—similar to `DeleteFile` and other Win32 functions.

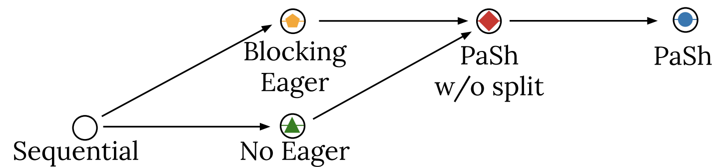


Fig. 6.6: Runtime setup lattice. Parallel No Eager and Blocking Eager improve over sequential, but are not directly comparable. PASH w/o Split adds PASH's optimized `eager` relay, and PASH uses all primitives in §6.4 (Fig. 6.7).

In the code above, `head` exits early, causing the last `cat` to exit before opening `fifo2`. As a result, the second `cat` never receives a `PIPE` signal that its consumer exited—after all, `fifo2` never even had a consumer! This, in turn, leaves the second `cat` unable to make progress, as it is both blocked and unaware of its consumer exiting. Coupled with `wait` at the end, the entire snippet reaches a deadlock.

This problem is not unique to PASH; it occurs even when manually parallelizing scripts using FIFOs (but not when using *e.g.*, intermediary files, *Cf.* §6.4, Laziness). It is exacerbated, however, by PASH's use of the `cat fifo1 fifo2` construct, used pervasively when parallelizing commands in Ⓢ.

To solve this problem, PASH emits cleanup logic that operates from the end of the pipeline towards its start. The emitted code first gathers the IDs of the output processes and passes them as parameters to `wait`; this causes `wait` to block only on the output producers of the dataflow graph. Right after `wait`, PASH inserts a routine that delivers `PIPE` signals to any remaining processes upstream.

Aggregator Implementations: As discussed earlier, commands in Ⓟ can be parallelized using a *map* and an *aggregate* stage (§5.1). PASH implements *aggregate* for several commands in Ⓟ to enable parallelization. A few interesting examples are *aggregate* functions for (i) `sort`, which amounts to the merge phase of a merge-sort (and on GNU systems is implemented as `sort -m`), (ii) `uniq` and `uniq -c`, which need to check conditions at the boundary of their input streams, (iii) `tac`, which reverses its input and consumes stream descriptors in reverse order, and (iv) `wc`, which adds inputs with an arbitrary number of elements (*e.g.*, `wc -lw` or `wc -lwc` *etc.*). The *aggregate* functions iterate over the provided stream descriptors, *i.e.*, they work with more than two inputs, and apply pure functions at the boundaries of input streams (with the exception of `sort` that has to interleave inputs).

Tab. 6.1: Summary of UNIX one-liners. Structure summarizes the different classes of commands used in the script. Input and seq. time report on the input size fed to the script and the timing of its sequential execution. Nodes and compile time report on PASH's resulting DFG size (which is equal to the number of resulting processes and includes `aggregators`, `eager`, and `split` nodes) and compilation time for a `--width` value of 16.

Script	Structure	Input	Seq. Time	#Nodes	Compile Time	Highlights
nfa-regex	3 × Ⓢ	1 GB	79m35s	49	0.05s	complex NFA regex
sort	Ⓢ, Ⓟ	10 GB	21m46s	77	0.09s	`sorting`
top-n	2 × Ⓢ, 4 × Ⓟ	10 GB	78m45s	96	0.14s	double `sort`, `uniq` reduction
wf	3 × Ⓢ, 3 × Ⓟ	10 GB	22m30s	96	0.15s	double `sort`, `uniq` reduction
spell	4 × Ⓢ, 3 × Ⓟ	3 GB	25m7s	193	0.34s	comparisons (`comm`)
difference	2 × Ⓢ, 2 × Ⓟ, Ⓝ	10 GB	25m49s	125	0.19s	non-parallelizable `diffing`
bi-grams	3 × Ⓢ, 3 × Ⓟ	3 GB	38m9s	185	0.31s	stream shifting and merging
set-difference	5 × Ⓢ, 2 × Ⓟ, Ⓝ	10 GB	51m32s	155	0.32s	two pipelines merging to a `comm`
sort-sort	Ⓢ, 2 × Ⓟ	10 GB	31m26s	154	0.29s	parallelizable Ⓟ after Ⓟ
shortest-scripts	5 × Ⓢ, 2 × Ⓟ	85 MB	28m45s	142	0.33s	long Ⓢ pipeline ending with Ⓟ

6.5. Evaluation

This section reports on whether PASH can indeed offer performance benefits automatically and correctly using several scripts collected out from the wild along with a few micro-benchmarks for targeted comparisons.

Highlights: This paragraph highlights results for `width=16`, but PASH's evaluation reports on varying `widths` (2–64). Overall, applying PASH to all 44 unmodified scripts accelerates 39 of them by 1.92–17.42×; for the rest, the parallel performance is comparable to the sequential (0.89, 0.91, 0.94, 0.99, 1.01×). The total average speedup over all 44 benchmarks is 6.7×. PASH's runtime primitives offer significant benefits—for the 10 scripts that we measured with and without the runtime primitives they bump the average speedup from 5.9× to 8.6×. PASH significantly outperforms `sort --parallel`, a hand-tuned parallel implementation, and performs better than GNU `parallel`, which returns incorrect results if used without care.

Using PASH's standard library of annotations for POSIX and GNU commands (§5.1), the vast majority of programs (> 40, with > 200 commands) require no effort to parallelize other than invoking PASH; only 6 (< 3%) commands, outside this library, needed a single-record annotation (§6.5.4).

In terms of correctness, PASH's results on multi-GB inputs are identical to the sequential ones. Scripts feature ample opportunities for breaking semantics (§6.5.5), which PASH avoids.

Setup: PASH was run on 512GB of memory and 64 physical × 2.1GHz Intel Xeon E5-2683 cores, Debian 4.9.144-3.1, GNU Coreutils 8.30-3, GNU Bash 5.0.3(1), and Python 3.7.3—without any special configura-

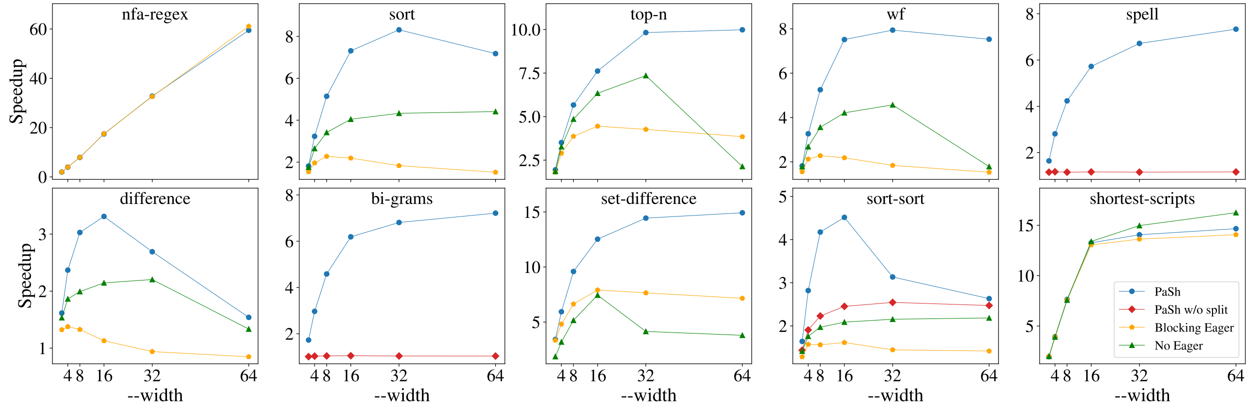


Fig. 6.7: PASH's speedup for `width=2–64`. Different configurations per benchmark: (i) PaSh: the complete implementation with `eager` and `split` enabled, (ii) PaSh w/o split: `eager` enabled (no `split`), (iii) Blocking Eager: only blocking `eager` enabled (no `split`), (iv) No Eager: both `eager` and `split` disabled. For some pairs of configurations, PASH produces identical parallel scripts and thus only one is shown.

tion in hardware or software. Except as otherwise noted, (i) all execution time measurements are averaged over 3 runs, (ii) all pipelines are set to (initially) read from and (finally) write to the file-system, (iii) `curl` fetches data from a different physical host on the same network connected by 1Gbps links.

We note a few characteristics of our setup that minimize measurement variability: (1) our evaluation experiments take several hours to complete (about 23 hours for the full set), (2) our experimental infrastructure is hosted on premises, not shared with other groups or researchers, (3) the runtime does not include any managed runtimes, virtualization, or containment,[8] (4) many commands are repeated many times—for example, there are more than 40 instances of `grep` in our benchmark set. The set of benchmarks also executes with smaller inputs multiple times a week (using continuous integration), reporting minimal statistical differences between runs.

Parallelism: PASH's degree of parallelism is configured by the `--width` flag (§6.3.3). PASH does not control a script's initial parallelism (*e.g.*, a command could spawn 10 processes), and thus the resulting scripts often reach maximum parallelization benefits with a value of `width` smaller than the physical cores available in our setup (in our case 64).

[8]While PASH is available via Docker too, all results reported in this paper are from non-containerized executions.

6.5.1. Common UNIX One-liners

We first evaluate PASH on a set of popular, common, and classic UNIX pipeline patterns [27, 28, 165]. The goal is to evaluate performance benefits due to PASH's (i) DFG transformations alone, including how `--width` affects speedup, and (ii) runtime primitives, showing results for all points on the runtime configuration lattice (Fig. 6.6).

Programs: Tab. 6.1 summarizes the first collection of programs. NFA-Regex is centered around an expensive NFA-based backtracking expression and all of its commands are in Ⓢ. Sort is a short script centered around a Ⓟ command. Wf and Top-n are based on McIlroy's classic word-counting program [28]; they use sorting, rather than tabulation, to identify high-frequency terms in a corpus. Spell, based on the original `spell` developed by Johnson [27], is another UNIX classic: after some preprocessing, it makes clever use of `comm` to report words not in a dictionary. Shortest-scripts extracts the 15 shortest scripts in the user's `PATH`, using the `file` utility and a higher-order `wc` via `xargs` [165, pg. 7]. Diff and Set-diff compare streams via a `diff` (in Ⓝ, non-parallelizable) and `comm` (in Ⓟ), respectively. Sort-sort uses consecutive Ⓟ commands without interleaving them with commands that condense their input size (*e.g.*, `uniq`). Finally, Bi-grams replicates and shifts a stream by one entry to calculate bigrams.

Results: Fig. 6.7 presents PASH's speedup as a function of `width`=2–64. Average speedups of the optimized PASH, *i.e.*, with `eager` and `split` enabled, for `width`={2, 4, 8, 16, 32, 64} are {1.97, 3.5, 5.78, 8.83, 10.96, 13.47}×, respectively. For No Eager, *i.e.*, PASH's transformations without its runtime support, speedups drop to 1.63, 2.54, 3.86, 5.93, 7.46, 9.35×.

Plots do not include lines for configurations that lead to identical parallel programs. There are two types of such cases. In the first, the PaSh (blue) and PaSh w/o Split (red, hidden) lines are identical for scripts where PASH does not add `split`, as the width of the DFG is constant; conversely, when both lines are shown (*e.g.*, Spell, Bi-grams, and Sort), PASH has added `splits` due to changes in the DFG width (e.g. due to a Ⓝ command). In the second type, *Pash w/o Split* (red) is identical to No Eager (green, hidden) and Blocking Eager (orange, hidden) because the input script features a command in Ⓟ or Ⓝ relatively early. This command requires an aggregator, whose output is of width 1, beyond which PaSh w/o Split configurations are sequential and thus see no speedup. Finally, Tab. 6.1 shows that PASH's transformation

time is negligible, and its COST [120], *i.e.*, the degree of parallelism threshold over which PASH starts providing absolute execution time benefits, is 2.

Discussion: As expected, scripts with commands only in Ⓢ see linear speedup. PASH's `split` benefits scripts with Ⓟ or Ⓝ commands, without negative effects on the rest. PASH's `eager` primitive improves over No Eager and Blocking Eager for all scripts. No Eager is usually faster than Blocking Eager since it allows its producer and consumer to execute in parallel. Sort-sort illustrates the full spectrum of primitives: (i) PaSh w/o Split offers benefits despite the lack of `split` because it fully parallelizes the first `sort`, and (ii) PaSh gets full benefits because `splitting` allows parallelizing the second `sort` too.

As described earlier, PASH often achieves the maximum possible speedup for a `width` that is lower than the number of available cores—*i.e.*, `width`=16–32 for a 64-core system. This is also because PASH's runtime primitives spawn new processes—*e.g.*, Sort with `width`=8 spawns 37 processes: 8 `tr`, 8 `sort`, 7 *aggregate*, and 14 `eager` processes.

Take-aways: PASH accelerates scripts by up to 60×, depending on the characteristics of the commands involved in a script. Its runtime constructs improve over the baseline speedup achieved by its parallelization transformations.

6.5.2. Unix50 from Bell Labs

We now turn to a set of UNIX pipelines found out in the wild.

Programs: In a recent celebration of UNIX's 50-year legacy, Bell Labs created 37 challenges [103] solvable by UNIX pipelines. The problems were designed to highlight UNIX's modular philosophy [118]. We found unofficial solutions to all-but-three problems on GitHub [30], expressed as pipelines with 2–12 stages (avg.: 5.58). They make extensive use of standard commands under a variety of flags, and appear to be written by non-experts (contrary to §6.5.1, they often use sub-optimal or non-UNIX-y constructs). PASH executes each pipeline as-is, without any modification.

Results: Fig. 6.8 shows the speedup (left) over the sequential runtime (right) for 31 pipelines, with `width`=16 and 10GB inputs. It does not include 3 pipelines that use `head` fairly early thereby finishing execution in under 0.1 seconds. We refer to each pipeline using its x-axis index (#0–30) in Fig. 6.8. Average speedup is

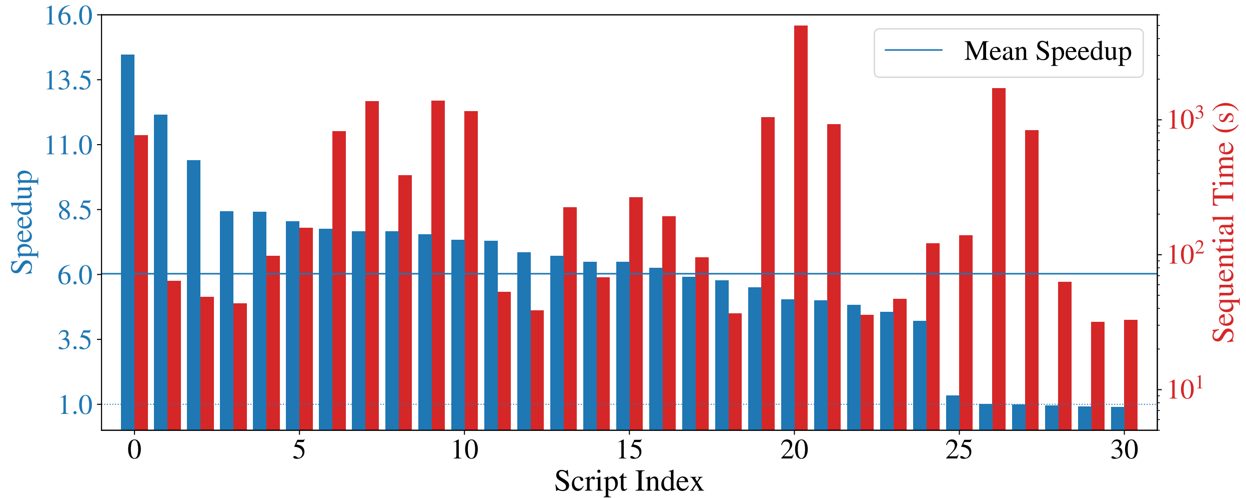


Fig. 6.8: Unix50 scripts. Speedup (left axis) over sequential execution (right axis) for Unix50 scripts. Parallelism is 16× on 10GB of input data (*Cf.*§6.5.2). Pipelines are sorted in descending speedup order.

6.02×, and weighted average (with the absolute times as weights) is 5.75×.

Discussion: Most pipelines see significant speedup, except #25-30 that see no speedup because they contain general commands that PASH cannot parallelize without risking breakage—*e.g.*, `awk` and `sed -d`. A UNIX expert would notice that some of them can be replaced with UNIX-specific commands—for example `awk "{print \$2, \$0}" | sort -nr`, used to sort on the second field can be replaced with a single `sort -nr -k 2` (#26). The targeted expressiveness of the replacement commands can be exploited by PASH—in this specific case, achieving 8.1× speedup (*vs.* the original 1.01×).

For all other scripts (#0–24), PASH's speedup is capped due to a combination of reasons: (i) scripts contain pure commands that are parallelizable but don't scale linearly, such as `sort` (#5, 6, 7, 8, 9, 19, 20, 21, 23, 24), (ii) scripts are deep pipelines that already exploit task parallelism (#4, 10, 11, 13, 15, 17, 19, 21, 22), or (iii) scripts are not CPU-intensive, resulting in pronounced I/O and constant costs (#3, 4, 11, 12, 14, 16, 17, 18, 22).

Take-aways: PASH accelerates unmodified pipelines found in the wild; small tweaks can yield further improvements, showing that PASH-awareness and scripting expertise can improve performance. Furthermore, PASH does not significantly decelerate non-parallelizable scripts.

6.5.3. Use Case: NOAA Weather Analysis

We now turn our attention to Fig. 5.1's script (§6.2).

Program: This program is inspired by the central example in "Hadoop: The Definitive Guide" [182, §2], where it exemplifies a realistic analytics pipeline comprising 3 stages: fetch NOAA data (shell), convert them to a Hadoop-friendly format (shell), and calculate the maximum temperature (Hadoop). While the book focuses only on the last stage, PASH parallelizes the entire pipeline.

Results: The complete pipeline executes in 44m2s for five years (82GB) of data. PASH with `width`=16 leads to 2.52× speedup, with different phases seeing different benefits: 2.04× speedup (vs. 33m58s) for all the pre-processing (75% of the total running time) and 12.31× speedup (vs. 10m4s) for computing the maximum.

Discussion: The speedup of the preprocessing phase of the pipeline is bounded by the network and I/O costs since `curl` downloads 82GB of data. However, the speedup for the processing phase (CPU-bound) is 12.31×, much higher than what would be achieved by parallelizing per year (for a total of five years). Similar to Unix50 (§6.5.2), we found that large pipelines enable significant freedom in terms of expressiveness.

Take-aways: PASH can be applied to programs of notable size and complexity to offer significant acceleration. PASH is also able to extract parallelism from fragments that are not purely compute-intensive, *i.e.*, the usual focus of conventional parallelization systems.

6.5.4. Use Case: Wikipedia Web Indexing

We now apply PASH to a large web-indexing script.

Program: This script reads a file containing Wikipedia URLs, downloads the pages, extracts the text from HTML, and applies natural-language processing—*e.g.*, trigrams, character conversion, term frequencies—to index it. It totals 34 commands written in multiple programming languages.

Results: The original script takes 191min to execute on 1% of Wikipedia (1.3GB). With `width`=16, PASH brings it down to 15min (12.7×), with the majority of the speedup coming from the HTML-to-text conversion.

Discussion: The original script contains 34 pipeline stages, thus the sequential version already benefits from task-based parallelism. It also uses several utilities not part of the standard POSIX/GNU set—*e.g.*, its

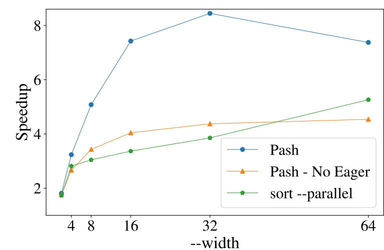
url-extraction is written in JavaScript and its word-stemming is in Python. PASH can still operate on them as their parallelizability properties—Ⓢ for url-extract and word-stem—can be trivially described by annotations. Several other stages are in Ⓢ allowing PASH to achieve benefits by exposing data parallelism.

Take-aways: PASH operates on programs with (annotated) commands outside the POSIX/GNU subsets and leads to notable speedups, even when the original program features significant task-based parallelism.

6.5.5. Further Micro-benchmarks

As there are no prior systems directly comparable to PASH, we now draw comparisons with two specialized cases that excel within smaller fragments of PASH's proposed domain.

Parallel Sort: First, we compare a sort parallelized by PASH (S_p) against the same sort invoked using the --parallel flag set (S_g).[9] While the --parallel flag is not a general solution, since it is not offered by all commands, but the comparison serves to establish a baseline for PASH. S_g's parallelism is configured to 2× that of S_p's --width (*i.e.*, the rightmost plot point for S_g is for --parallelism=128) to account for PASH's additional runtime processes.



A few points are worth noting. S_p without eager performs comparably to S_g, and with eager it outperforms S_g (∼ 2×); this is because eager adds intermediate buffers that ensure CPU utilization is high. S_g indicates that sort's scalability is inherently limited (*i.e.*, due to sort, not PASH); this is why all scripts that contain sort (*e.g.*, §6.5.1–6.5.4) are capped at 8× speedup. The comparison also shows PASH's benefits to command developers: a low-effort parallelizability annotation achieves better scalability than a custom flag (and underlying parallel implementation) manually added by developers.

GNU Parallel: We compare PASH to parallel (v.20160422), a GNU utility for running other commands in parallel [164], on a small bio-informatics script. Sequential execution takes 554.8s *vs.* PASH's 128.5s (4.3×), with most of the overhead coming from a single command—cutadapt.

There are a few possible ways users might attempt to use GNU parallel on this program. They could use

[9] Both sorts use the same buffer size internally [142].

it on the bottleneck stage, assuming they can deduce it, bringing execution down to 304.4s (1.8× speedup). Alternatively, they could (incorrectly) sprinkle `parallel` across the entire program. This would lead to 3.2× performance improvements but incorrect results with respect to the sequential execution—with 92% of the output showing a difference between sequential and parallel execution. PASH's conservative program transformations are not applied in program fragments with unclear parallelizability properties.

CHAPTER 7

PaSh-JIT: Just-in-time automatic parallelization of complete shell programs

Material from this chapter was previously published as "Konstantinos Kallas, Tammam Mustafa, Jan Bielak, Dimiris 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 769–785, Carlsbad, CA, July 2022. USENIX Association." [95]. I was the primary author of this paper, being the lead developer of the system and its just-in-time architecture; the rest of the coauthors helped with the development of the commutativity-aware optimizations and the parsing library, as well as with the system evaluation.

7.1. Introduction

PASH, the system described in Chapter 6 accelerates shell pipelines by exploiting data parallelism: using *ahead-of-time* (AOT) analysis and transformation, these systems parse, analyze, and transform shell scripts into new scripts that execute in parallel.

Unfortunately, AOT parallelization quickly becomes intractable due to the dynamic nature of the shell: dynamic features such as variable expansion and command substitution, pervasive in shell scripts, generate and consume values at run-time while depending on and interacting with the broader environment—*i.e.*, the filesystem, the environment variables, and the shell interpreter itself. Additionally, modern shells offer several different configurations and execution modes, leading to complex behaviors described in hundreds of pages of POSIX standardese [22]. The complexity of these interactions and their side-effects lead existing parallelization tools to an unavoidable trade-off between (1) being conservative, aborting on scripts that use dynamic features, or (2) being unsound, possibly breaking scripts during parallelization. Recent systems [144, 173, 156] tend to be conservative—operating only on fully expanded shell pipelines and having a hard time even on simple uses of variables (see §7.2).

This dissertation chapter presents PASH-JIT, a production-grade just-in-time (JIT) shell-script compiler aimed at non-interactive parallelization: PASH-JIT focuses on three practical (but conflicting) goals: (G1)

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

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

We apply PASH-JIT to a variety of benchmarks, ranging from scripts collected from the wild to the POSIX test suite. PASH-JIT behaves identically to Bash 4.4.20(1) on 406 out of 408 applicable POSIX tests; matching Bash is a significant achievement even for a non-parallelizing shell—shells in widespread use differ on much larger subsets of tests. PASH-JIT offers speedups up to 33.7× over Bash on a 64-core machine (improving the state of the art [173] by 2× on average), notably parallelizing scripts that prior work failed to parallelize due to dynamic behaviors.

The chapter begins by exemplifying dynamic shell features and the application of PASH-JIT's techniques (§7.2). Sections 7.3–7.6 describe PASH-JIT's main contributions:

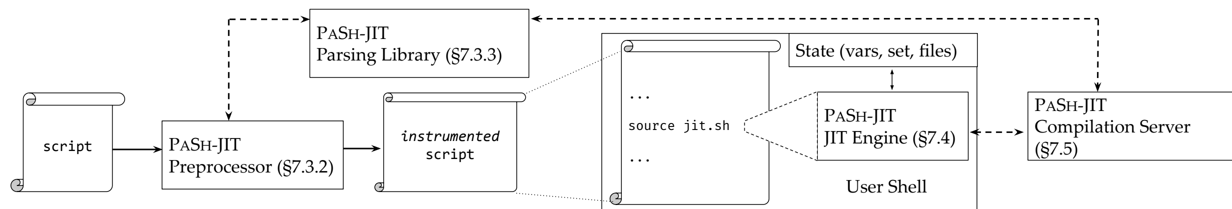


Fig. 7.1: PASH-JIT overview. PASH-JIT instruments scripts with calls to the JIT engine, which passes program fragments to the compilation server at run-time.

- *A dynamic interposition framework for the shell*: A just-in-time analysis and optimization subsystem enables safe and effective parallelization during the execution of a script, dealing with the challenges of dynamic shell-script behavior. A first pass determines where to insert calls to a parallelizing optimizer in a given input script (§7.3), which is then invoked on-the-fly while the script is executing (§7.4).

- *A stateful, parallelizing compilation server*: PASH-JIT queries a long-lived parallelization server at run-time to compile script fragments. This model improves run-time efficiency by avoiding startup costs on every JIT invocation, and enables additional run-time optimizations for (1) executing independent regions in parallel, and (2) pipelining compilation and execution. The core of the server has been modelled and formally verified using SPIN [82] (§7.5).

- *Commutativity-aware optimization*: Additional compilation optimizations target commands that are commutative with respect to their input, along with parallelizing transformations and run-time primitives that improve the run-time performance of scripts that contain such commands (§7.6).

The chapter then presents PASH-JIT's evaluation (§7.7) and related work (§7.8), before concluding (§7.9). PASH-JIT is MIT-licensed open-source software supported by the Linux Foundation at https://github.com/binpash/.

7.2. Example and overview

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

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

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

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

None of the values of these variables can be known ahead of time just by analyzing the program's source code. They become known only at run-time, when the shell interpreter reaches these points in the program's execution. A sound AOT compiler such as PaSh [173] or POSH [144] would fail to parallelize—foregoing all the performance benefits of data-parallel execution spread across many files in `$IN`.

PaSh-JIT instead takes a JIT approach that interjects parallelization opportunities during and throughout the script's execution (Fig. 7.1).

Dynamic interposition (§7.3): PaSh-JIT first uses a preprocessing step to instrument all potentially optimizable program regions with calls to the JIT engine. PaSh-JIT chooses regions to maximize the potential

benefits of parallelizing them: intuitively, commands and pipelines can yield significant benefits, whereas word expansion, control flow, and variable assignments are operations that do not perform heavy computation and can therefore be left as they are. PaSh-JIT's preprocesor and compiler both make extensive use of parsing/unparsing of shell source code, implemented as a new parsing library. After PaSh-JIT has inserted calls to the JIT engine, it invokes the user's shell interpreter to execute this transformed script. During this execution, the JIT engine calls the parallelizing compiler at run-time—right before the execution of each fragment, when the state of the shell and the file system have already been resolved. The transformed program maps original commands to *regions*—for example, `region8` corresponds to the `cd` call and `region10` corresponds to the pipeline in the `for` loop.

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

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

JIT engine (§7.4): Internally, the JIT engine first saves the state of the shell at that point in the script's execution to isolate it from compilation—protecting the shell from the JIT engine and protecting the JIT engine from obscure shell configurations. PaSh-JIT then invokes the compiler to attempt to parallelize the fragment. If the compiler succeeds, PaSh-JIT runs the resulting parallel fragment; if not, it runs the original, unmodified region. In both cases, PaSh-JIT will first restore the state of the shell before executing the fragment. Whether the compiler succeeds or not depends on the properties of the fragment's code—*e.g.*, PaSh-JIT will reject `region8` due to the side-effectful `cd` command, but will accept `region10` compiling `grep` and `sort` into the parallel fragment below:

```
c_split /tmp/fifo8 /tmp/fifo9 /tmp/fifo10 &
c_wrap 'grep "^....$"' </tmp/fifo9 >/tmp/fifo11 &
```

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

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

Dependency untangling (§7.5): While the JIT engine operates as if invoked on every region, PASH-JIT is engineered to spawn a long-running stateful compilation server just once, feeding it compilation requests until the execution of the script completes. This design has two benefits: (1) it reduces run-time overhead by avoiding reinitializing the compiler for each compilation request; and (2) it allows maintaining and querying past compilation results when compiling a new fragment. The latter allows PASH-JIT to untangle dependencies *across* regions, finding and exploiting opportunities for cross-region parallel execution. For example, the server's first invocation on `region10` (the body of the loop) determines that all prior successfully compiled regions have finished executing. PASH-JIT can thus simply run the loop in the background and continue with the second iteration in a task-parallel fashion, without waiting for the first iteration to complete executing. During the second invocation on `region10`, PASH-JIT will use the dependency state to determine that while the previously compiled fragment is still running, the input and output files of the two regions are completely independent and can thus be executed in parallel: our loop is now pipelined! PASH-JIT goes beyond intra-region data parallelism: the JIT enables inter-region task parallelism by resolving dependencies and confirming they are independent.

Commutativity analysis & compilation (§7.6): The first goal when compiling fragments such as `region10` is to identify command sequencies that are parallelizable using a divide-and-conquer strategy. Due to the shell's order-aware nature [76], naive divide-and-conquer would need to (1) read the entire input before splitting it, to determine the exact size of each batch, leading to stalled pipeline parallelism; and (2) wait until

all of its predecessors have consumed their batch, storing data after split on disk, to ensure that all parallel nodes will not wait for their input.

While these overheads are unavoidable in the general case, and are indeed incurred by prior systems [173, 156], they can fortunately be alleviated for subsets of parallelizable commands. Two such subsets include (1) stateless commands such as `grep -c '^....$'` that operate in a line-oriented fashion, meaning that data-parallel copies of these commands can combine their partial output using a reordering operation, and (2) commutative commands such as `sort -u` that produce equivalent output regardless of the order of the input lines. PASH-JIT leverages this insight to achieve more effective parallelization by splitting into streaming micro-batches (using `c_split`) in a round-robin fashion—avoiding the overheads of reading all the input before splitting and of unnecessary storage to disk. It also wraps stateless commands to strip and re-add the microbatch headers (using `c_wrap`) and removes these headers completely before commutative commands (using `c_strip`).

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

7.3. Interfacing with the shell

PASH-JIT works by interposing on the shell, effectively rewriting invocations to external commands. Challenges arise due to the shell's complex semantics and its intricate internal state, both of which complicate side-effect-free interposition. The shell uses a string-based, bi-modal semantics: commands undergo *expansion*, a string rewriting phase where variables, tildes, and globs are processed before the commands undergo *evaluation*. Both modes have complex semantics heavily involved with the shell's state [71]; any rewriting must be careful to leave the shell's state unaltered.

7.3.1. Dynamic Interposition

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

AOT parallelization centers around the identification of *parallelizable regions*—script fragments that may be safely parallelized to yield performance gains. Semantically, parallelizable regions only contain a set of command invocations that satisfy the following conditions: (1) they have no file dependencies (*interference-free*), *i.e.*, all commands can execute concurrently without affecting each other, (2) they communicate with each other using explicit UNIX channels (fifos/pipes); (3) they are *pure*, only affecting the environment by reading and writing to files, *i.e.*, they do not modify environment variables; and, (4) they are fully expanded. An AOT compiler parses and transforms these regions to an intermediate representation such as directed-acyclic [144] or dataflow [173] graphs, abstracted as functions that take a set of input files and produce a set of output files [76]. It then applies transformations on these graphs to perform the original computation in parallel.

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

7.3.2. Preprocessor

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

The intuition behind PASH-JIT's preprocessor is that a syntactic analysis of a shell script is enough to suggest potential parallelizable regions. This analysis is imprecise: there is no way to determine whether a command

invocation will be pure ahead of time. Its goal however, is not to find parallelizable regions exactly, but rather to find potential compilation sites—PASH-JIT sorts out the details at run-time, using up-to-date information about the system's state.

There is a trade-off when choosing the right size for these regions: the larger the region, the more opportunities exist for analysis and optimization but the less likely it is for the entire region to be parallelizable. PASH-JIT targets a middle-ground: maximal syntactic schedule-free regions—*i.e.*, command sequences composed using shell primitives that do not impose scheduling restrictions. By focusing on maximal schedule-free regions, PASH-JIT minimizes the number of compiler invocations and maximizes the cross-command parallelization opportunities for the compiler. Note that schedule-free regions underapproximate interference-free regions (§7.3.1), *e.g.*, two commands composed in sequence, using `;`, that write to different files do not interfere but are not syntactically schedule-free.

The preprocessor finds these maximal regions by searching the AST bottom-up, combining schedule-free subtrees when they are composed using constructs that do not introduce scheduling constraints (*e.g.*, `&`, `|`). When a region cannot outgrow a certain subtree, it is replaced with a call to the JIT engine. If successfully compiled, a region is transformed to a dataflow graph—a convenient and well-studied computation model amenable to transformation-based optimizations [76]. The instrumented AST resulting from the compilation is finally translated (unparsed) back to shell code and sent over to the underlying shell for execution.

7.3.3. Parsing Library

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

PASH-JIT parses lines of shell script as they come in, and unparses lines in order to execute them in the user's shell; it also uses parsing and unparsing during compilation, when the compilation server emits an optimized string or passes strings to the shell for expansion. PASH-JIT initially used `libdash`—an OCaml library built using the dash parser and part of Smoosh [70, 71]—but this caused two main issues. First, `libdash`'s unparsing introduced several bugs, as at the time it was used by the libdash project primarily for testing and diagnostics and had much of its was functionality untested. Second, `libdash` parsing introduced significant run-time overhead due to (1) the cost of forking and executing the OCaml binary, (2) overheads

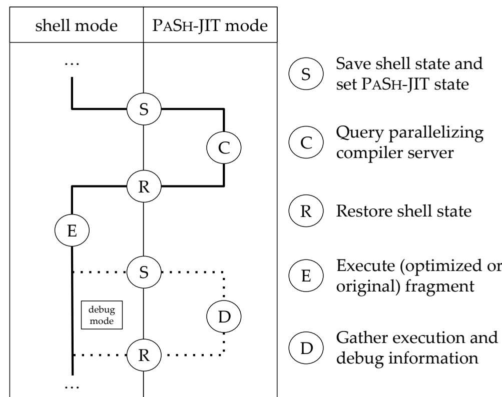


Fig. 7.2: JIT engine overview. The different stages of the engine's execution.

due to serialization and deserialization during communication, and (3) suboptimal implementation. Runtime overheads were a significant concern due to PASH-JIT's online JIT parallelization, which intermixes calls to the compiler during the program's execution—bringing parsing and unparsing into the critical path of program execution.

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

7.4. The JIT engine

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

The JIT engine is a reflective shell script: by inspecting the state of the shell and that of the broader system, it can transparently work with the compiler to determine whether or not to parallelize a script (Fig. 7.2). When running scripts with PASH-JIT, it is helpful to think of the shell as having two modes: (1) conventional shell

mode, where scripts execute in the original shell context, and (2) PaSh-JIT mode, where the runtime reflects on shell state and invokes a compiler to determine whether to execute the original or an optimized version of the target region. To switch from shell mode to PaSh-JIT mode, the JIT engine must carefully save the state of the user's shell; to switch back, it must carefully put things back just the way they were. A shell's state is quite complex: beyond saving and restoring variables, the runtime must account for various shell flags along with other internal shell state (*e.g.*, the previous exit status, working directory).

7.4.1. JIT Stages

When running normally, the JIT engine transitions into and out of PaSh-JIT mode once per possible parallelizable region (Fig. 7.2): the JIT engine saves the shell state and switches into PaSh-JIT mode (S); then it tries to compile the current fragment (C); whether successful or not, the JIT engine restores the state and switches back to shell mode (R); and, finally, either the original fragment or the optimized parallel version is executed (E). With debugging enabled, the JIT engine switches back into PaSh-JIT mode (S) to collect debugging information (D), restoring again afterwards (R).

Saving (S): When entering a possible parallelizable region, the first step is to save the shell state—recording the previous command's exit status, the values of environment variables, and the configuration of the shell—essentially, a continuation that can later be restored to execute the target fragment. Once the state is saved, PaSh-JIT mode reconfigures the user's shell to avoid changing script behavior. For example, if the user's shell has the `-e` "exit on error" flag set, the shell should exit immediately when a command (or a pipeline) returns a non-zero exit status, unless that command is in a checked position (*e.g.*, after `!`, or in the condition of an `if` or `while`) [22]. However, failing commands should not stop the JIT itself, so `-e` is unset (and will be restored later in (R)).

Compilation (C): With the state saved and shell reconfigured, PaSh-JIT tries to compile the script fragment: the JIT engine queries the compilation server (§7.5) with the script fragment (already parsed during preprocessing) along with the saved shell state, so that the compilation server can try to expand all of the words in the fragment. The server responds to indicate whether it managed to optimize the fragment.

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

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

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

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

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

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

7.5. Parallelizing compilation server

For each possible parallelizable region, the JIT engine queries the compiler: can this region actually be optimized? To answer this question, PaSh-JIT builds on the foundation of PaSh (see Chapter 6) dataflow compiler (§7.5.1). As ever, it focuses on preserving behavior and minimizing overhead.

To preserve correct behavior in the face of the shell's dynamism, PaSh-JIT expands each script region prior to compilation (§7.5.2). To minimize overhead due to fixed startup costs—*e.g.*, initialization, dependency loading, logging setup, and output file arrangement—PaSh-JIT packages the new compiler as a stateful

compilation server communicating via UNIX domain sockets.[10]

The compilation server is also augmented to support a larger set of optimization opportunities, by storing and using information from one compilation to help another. PASH-JIT's long-lived compilation server achieves these additional optimizations by allowing parallelizable regions that work on independent inputs and outputs to be run in parallel (§7.5.3) and by learning to improve its parallelism configuration from past compilations (§7.5.4).

7.5.1. Command Annotations

PASH-JIT uses the command annotation and specification framework introduced in Chapter 5, extended to also indicate whether a command invocation is commutative (§7.6.1). This framework provides information about a command invocation's parallelizability class, inputs, and outputs. A command annotation can be used to extract high-level information about a specific command invocation, *i.e.*, a precise instantiation of its flags, options, and arguments. For example, annotations determine whether a given command invocation is pure and what its inputs and outputs are.

PASH-JIT uses this annotation framework to extract information for commands that are not shell builtins—that is, commands like `sort` and `grep`. Annotations enable analyses and transformations over command invocations by lifting them to pure dataflow nodes in a dataflow intermediate representation (IR) [76]. For example, `grep -f dict.txt src.txt > out.txt` is a dataflow node with two input files (`dict.txt` and `src.txt`) and one output file (`out.txt`), which are all extracted from the annotation of the `grep` command. Annotations also describe parallelization opportunities, *e.g.*, `grep "pattern" src.txt` processes each line of `src.txt` independently, and so it can be parallelized.

7.5.2. Early, Pure Expansion

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

One way to achieve expansion would be for PASH-JIT to maintain a "mirror" Bash process when initializing,

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

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

PaSh-JIT avoids the overhead of a mirror shell by performing its own expansion, relying on the optimistic nature of the JIT engine (§7.4): if most common forms can be expanded in the compiler itself, the compiler will succeed often without incurring interprocess communication overheads; if expansion fails, PaSh-JIT will just run the original fragment. Armed with this insight, PaSh-JIT implements a subset of expansion in the compilation server itself. PaSh-JIT's custom expansion is *purely* functional, in that it does not affect shell state by setting variables or running command substitutions. The expansion routine is implemented in less than 300 LOC of Python, and reduces the compilation overhead significantly (§6.5). Expansion takes the host shell's configuration and expands common, safe expansions in as many positions as possible—in simple commands, pipelines, and other parallelizable regions.

PaSh-JIT's expansion routine implements most parameter formats, plain tildes, and appropriate quoting. Currently, it does not cover impure expansion (*e.g.*, parameter formats that have side-effects like `${x=foo}`, which will set `x` to `foo` if `x` is unset), since impurity violates the parallelizable region requirements. It also does not implement a few expansion cases—*e.g.*, arithmetic expansions of the form `$((x + 1))`—that were not seen in the corpus of parallelizable scripts used to evaluate PaSh-JIT (§6.5). Adding support for unimplemented forms would require engineering effort, but not a fundamental change to PaSh-JIT's expansion. If the expansion encounters a term it cannot expand—because it is unimplemented or because it would be impure—the compilation process aborts and PaSh-JIT runs the original fragment.

7.5.3. Dependency Untangling

PaSh-JIT's compilation server makes it easy to detect when parallelizable regions are independent—including, for example, independent program fragments that are sequentially composed with `;` or different iterations of a `for` loop. A key insight here is the semantics of PaSh-JIT's successful compilation: if the PaSh-JIT

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

Fig. 7.3: Compilation server algorithm (pseudocode) extended for dependency untangling.

compiler succeeds on a given region, that region's original script fragment must only affect its input and output streams (files). That is, successful fragment compilation means that the fragment is *pure*, reading from and writing to a well-defined set of streams without modifying any other global system state such as non-temporary streams or environment variables.

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

To discover independent fragments, the compilation server (Fig. 7.3) and JIT engine (Fig. 7.4) are extended to communicate about successfully compiled fragments. Coordinating using `exit` requests, the compilation server maintains a map of running fragments. When it receives a compilation request that succeeds, the server waits for all prior fragments with dependencies to finish executing; only then does it send the compiled fragment to the JIT engine for execution in the background. While the compiled fragment executes in the

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

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

Fig. 7.4: JIT engine algorithm (pseudocode) extended for dependency untangling.

background, the JIT engine can exit PASH-JIT mode, and execution proceeds with the rest of the input script. When execution reaches another fragment and the JIT engine returns to PASH-JIT mode, the JIT engine will block again until the compilation server responds. Even if the compilation server encounters a fragment that fails to compile, the server blocks on dependencies: the uncompilable fragment might have arbitrary side-effects.

To ensure that our algorithm is correct, we modeled it using the SPIN Model Checker [82] and we verified (i) that it does not lead to deadlocks, (ii) that no failed compiled region is running simultaneously with any other region, and (iii) that two regions with dependencies never run at the same time.

7.5.4. Profile-driven Compiler Configuration

The long-lived PASH-JIT compilation server can additionally use dynamic information to improve compilation. One particularly effective optimization is to dynamically determine maximum parallelism degree. As scripts might already feature task-based parallelism, spawning too many data-parallel processes can overload the system—leading to higher overheads that cut into the speedup or even result in a slowdown. These slowdowns tend to occur when there are many computationally light commands with small inputs, *i.e.*, when the overhead of managing parallelism is higher relative to the actual work to be done. The PASH-JIT compiler can reflect on prior fragments to determine an appropriate parallelism degree.

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

overall execution time.

7.6. Commutativity awareness

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

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

7.6.1. Compilation: Dataflow Model

The PASH-JIT compiler operates on a dataflow IR that builds on PASH, where commands correspond to nodes and communication channels correspond to edges between nodes. To enable commutativity-aware transformations, PASH-JIT extends PASH's annotation framework (§7.5.1) to indicate whether a command invocation is commutative (in addition to its parallelizability characteristics).

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

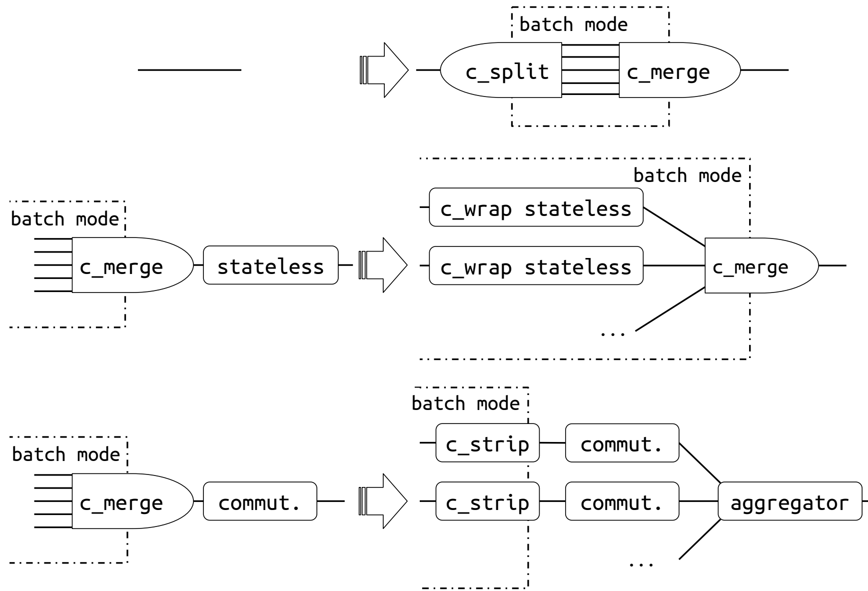


Fig. 7.5: Overview of commutativity-aware transformations.

header-removal node that often precedes commutative commands.

Transformations: To expose commutativity-aware parallelism, PASH-JIT transforms the dataflow graph; see §7.2 for an example. The transformations are visualized in Figure 7.5. The first transformation introduces a pair of `c_split` and `c_merge` before any commutative (*e.g.*, `sort`) or stateless (*e.g.*, `grep`) command. Another transformation then tries to eliminate unnecessary splits and merges, delaying `c_merge` as late as possible (*i.e.*, enclosing the biggest possible part of the graph). If a stateless command follows a `c_merge`, the command is wrapped with `c_wrap` and the `c_merge` is commuted after it. If a commutative command follows a `c_merge`, the command is parallelized and `c_merge` is transformed to a set of `c_strip` commands. Finally, if a `c_split` follows a `c_merge`, then the two are fused together to the identity function, connecting the inputs of `c_merge` with the outputs of `c_split`.

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

Tab. 7.1: Benchmark summary. Summary of all the benchmarks used to evaluate PASH-JIT and their characteristics.

	Benchmark Set	Short Label	Sections	Scripts	LOC	Input	Source
1	POSIX Test Suite	PosixTests	§7.7.1	7	29k	—	[73]
2	Common & Classic One-liners	Classics	§7.7.1–7.7.3	10	123	14G	[28, 27, 165, 92, 118]
3	Bell Labs Unix50	Unix50	§7.7.1–7.7.3	36	142	21G	[103, 30]
4	COVID-19 Transit Analytics	COVID-mts	§7.7.1–7.7.3	4	79	3.4G	[171]
5	Natural-Language Processing	NLP	§7.7.1–7.7.3	21	306	1060 books	[44]
6	NOAA Weather Analysis	AvgTemp	§7.7.1–7.7.3	1	31	36.2G	[182]
7	Wikipedia Web Indexing	WebIndex	§7.7.1–7.7.3	1	116	1000 files	[173]
8	Video/Audio Processing	MediaConv	§7.7.1–7.7.3	2	35	2.2+2.2G	[144, 159]
9	Program Inference	ProgInf	§7.7.1–7.7.3	1	18	2330 libraries	[176]
10	Traffic/PCAP Log Analysis	LogAnalysis	§7.7.1–7.7.3	2	63	10–20G	[144, 159]
11	Genomics Computation	Genomics	§7.7.1–7.7.3	1	34	100G	[143, 39]
12	AUR Package Compilation	AurPkg	§7.7.1–7.7.3	1	27	150 packages	[41]
13	Encryption/Compression	FileEnc	§7.7.1–7.7.3	2	44	20G	[125]
14	Microbenchmarks	MicroBench	§7.7.3	1	6	—	custom (ours)

7.6.2. Runtime: Commutativity Implementation

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

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

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

Handling inputs with long lines: An input may contain lines that are longer than the `c_split` block size. Such an event leads to non-uniform block sizes and high memory consumption, because each block must be read and sized completely before splitting and adding to the header. PASH-JIT addresses this issue by introducing the `is_last` header field in `c_split`: if a block exceeds the specified size (due to containing large lines) the block is split into multiple blocks; all blocks share the same `block_id` but only the last

sets `is_last` to true. Sub-blocks with the same `block_id` are sent downstream in-order, and therefore downstream commands can use the `is_last` information to correctly reconstruct the output and know when a block ends. Block splitting reduces memory requirements and improves performance, as it allows for higher utilization regardless of the frequency of newlines. And blocks maintain a constant size throughout the flow, despite the presence of commands with high output-to-input ratio such as `curl`.

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

7.7. Evaluation

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

To evaluate PASH-JIT, we use three experiments on benchmarks (Tab. 7.1). The first experiment focuses on PASH-JIT's compatibility and uses the entire POSIX test suite as well as additional scripts (§7.7.1). The second experiment focuses on the performance gains achieved by PASH-JIT's parallelization, evaluated using a variety of benchmarks and workloads (§7.7.2). The last experiment zooms into PASH-JIT-internal overheads and associated optimizations (§7.7.3).

Hardware & software setup: PASH-JIT was run on 64 physical × 2.1GHz Intel Xeon E5-2683 cores with 512GB of RAM, Debian 4.9.144-3.1, GNU Coreutils 8.30-3, GNU Bash 4.4.20(1), and Python 3.7.3. There is no special configuration in hardware or software. We use Dash v.0.5.8-2.10 and Ksh v.93u+ 2012-08-01. All scripts were executed completely unmodified, using environment variables, loops, and other shell constructs. To minimize statistical non-determinism, we host our experimental infrastructure on our own premises, avoid sharing with other research groups, and repeat the experiments several times noting

Tab. 7.2: Correctness results. Running the POSIX test suite on Bash and PaSh-JIT. Tests are grouped in rows by theme. Columns contain the group name, total tests, non-applicable tests, and passing tests for PaSh-JIT and Bash.

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

imperceptible variance.

7.7.1. Correctness

We evaluate the correctness of PaSh-JIT across all benchmarks from Tab. 7.1 by checking that PaSh-JIT's stdout and exit status are equivalent to the ones produced from Bash. The output is over 650 million lines (18GB), taken from 82 scripts, in all of which PaSh-JIT's output and exit status are correct. To increase our confidence on correctness, we use the POSIX shell test suite with both Bash and PaSh-JIT.

Benchmarks: The POSIX test suite is a thorough evaluation of shell behavior, comprising 1007 'assertions' evaluated using 494 distinct, assertion-numbered test cases over 29k LOC of shell scripts (plus library support). We exclude (a) 78 test cases because they test the platform (*e.g.*, locales) rather than the shell, and (b) 8 cases because they test interactivity, which is out of scope for PaSh-JIT (§7.1). These leave a total of 408 runnable test cases. The test cases use a mix of shell language features (*e.g.*, redirection, pipes), builtin commands (*e.g.*, `set`, `echo`), and standard UNIX utilities (*e.g.*, `printf`, `grep`). The POSIX suite tests many corner cases of shell behavior—*e.g.*, that aliases ending in space continue alias expansion (Assertion no. 284), that pipelines take precedence over redirections in their constituent commands (no. 454), or that `return` in a `trap` action restores the previous command's exit status (no. 651)—totaling several thousand behaviors. The exact number of 'tests' is hard to quantify: some test cases check a single behavior (*e.g.*, expanding an unset variable under `set -u`); others check hundreds (*e.g.*, many different characters escape properly; many different arithmetic expressions evaluate correctly).

Results: PaSh-JIT overwhelmingly agrees with Bash (Tab. 7.2). PaSh-JIT passes 374 and fails 34 POSIX tests, while Bash passes 376 and fails 32 POSIX tests. PaSh-JIT diverges from Bash on the test cases for a mere 2 tests (no. 430 and 691) where Bash passes but PaSh-JIT fails. These two failures concern the ranges

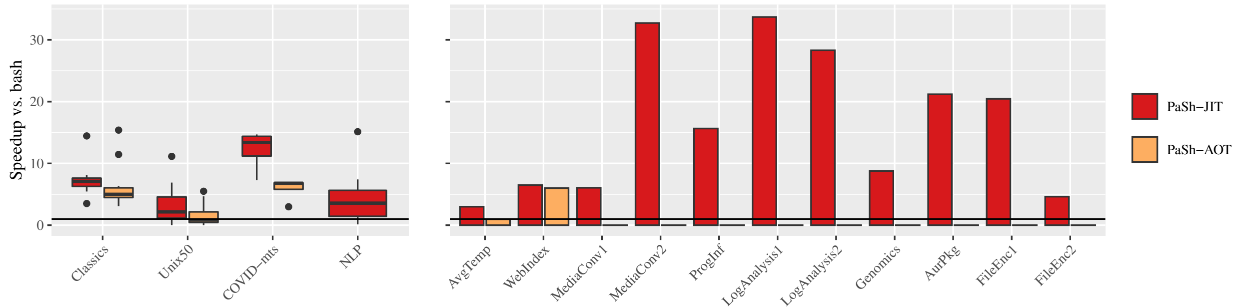


Fig. 7.6: PASH-JIT Performance. PASH-JIT speedup (*vs.* PASH whenever possible) over Bash for Tab. 7.1 rows 2–5 (left, box) and 6–13 (right, bar) (*Cf.*§7.7.2).

of non-zero exit status and are in fact due to an unusual inconsistency in Bash itself (see "Discussion", below).

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

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

Bash is inconsistent when called with the `-c` flag. Contrary to most other shells (*i.e.*, dash, ksh, mksh, posh, sash, Smoosh, yash, zsh), Bash is the only shell that, when failing during `-c` invocations, exits with 127—*i.e.*, outside the POSIX-mandated range. When PASH-JIT invokes the underlying Bash interpreter using `-c` in order to set `$0`, it receives and propagates an exit status that does not comply with POSIX. The rest of the Bash failing tests are caused by various subtleties; it is not clear which failures are 'true bugs' and which are considered desirable divergences from the spec. Greenberg and Blatt [71] discuss how implementations diverge from the POSIX spec. PASH-JIT mirrors the behavior of Bash in all those cases.

To put the number of diverging tests of PASH-JIT and Bash into perspective, we note that other production

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

7.7.2. Performance

We evaluate PASH-JIT's performance on 12 sets of real-world shell scripts taken from a variety of sources (Tab. 7.1, rows 2–13), totalling 82 shell scripts and 1015 LOC.

Benchmarks: Classics and Unix50 contain classic and recent (c. 2019) scripts making heavy use of UNIX and Linux built-in commands. COVID-mts contains four scripts used to analyze real telemetry data from mass-transit schedules during a large metropolitan area's COVID-19 response. NLP contains several scripts from UNIX-for-poets, a tutorial for developing programs for natural-language processing out of UNIX and Linux utilities. AvgTemp contains a large script downloading and processing multi-year temperature data across the US. WebIndex is a large multi-stage script for web crawling and indexing, using a variety of third-party and built-in utilities. MediaConv contains two scripts that process, transform, and compress video and audio files. ProgInf contains a script that downloads JavaScript packages from the npm registry and applies a security-oriented static program analysis. LogAnalysis contains two scripts that apply typical system-administration and network-traffic analyses over log files. Genomics contains a script that processes next-generation sequencing data for the purposes of diagnostic virology. AurPkg contains the main script that compiles, builds, and packages software for the AUR Linux distribution. Finally, FileEnc contains long aliases that encrypt and compress files.

Results: PASH-JIT surpasses PASH's speedups (*vs.* Bash) on existing benchmarks and extends speedups to new ones (Fig. 7.6). Box-plots show results for multi-benchmark suites (Tab. 7.1, rows 2–5) and bars for individual scripts (Tab. 7.1, rows 5–13). PASH-JIT can run several more scripts than PASH (for which performance bars are set to 0). Across all benchmarks, PASH-JIT achieves an average speedup of 5.86× (*vs.* 2.9× for PASH) and a maximum speedup of 33.7× (*vs.* 15.38× for PASH).

A few scripts exhibit slowdowns when compiler startup, runtime, and parallelization overheads (splitting, merging) start dominating. PASH-JIT decelerates 14 scripts; PASH decelerates 20 scripts—and cannot run

30 additional scripts that PASH-JIT parallelizes. The scripts that PASH-JIT decelerates either have short sequential running times (8ms–10s) or have very short-running fragments in tight loops (*e.g.*, 1K iterations, 14ms per iteration). For example, PASH-JIT decelerates Unix50's `20.sh` (Bash: 8ms; PASH-JIT: 1.3s) and NLP's `no-vowel.sh` (Bash: 14s; PASH-JIT: 0.24×), on which PASH cannot operate.

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

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

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

7.7.3. Further Microbenchmarks

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

Dynamic optimizations: To better understand the benefits of dependency untangling and profile-driven compiler configuration (CC), we use benchmarks that have sequences of statements—*e.g.*, some form of sequential composition or `for`-loops: rows 5, 6, 8–13 from Tab. 7.1. One-line scripts such as Unix50 and WebIndex feature single pipelines and thus cannot benefit from any inter-region optimizations.

Across all scripts and compared to Bash, PASH-JIT achieves a speedup of 8.17×. PASH-JIT without profile-driven CC achieves 7.58×, and additionally without dependency untangling 0.55× (Fig. 7.7). The 0.55× slowdown is due to limited intra-region parallelization in these benchmarks. Profile-driven CC may slightly reduce speedup in highly parallelizable scripts, because it explores lower parallelization degrees.

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

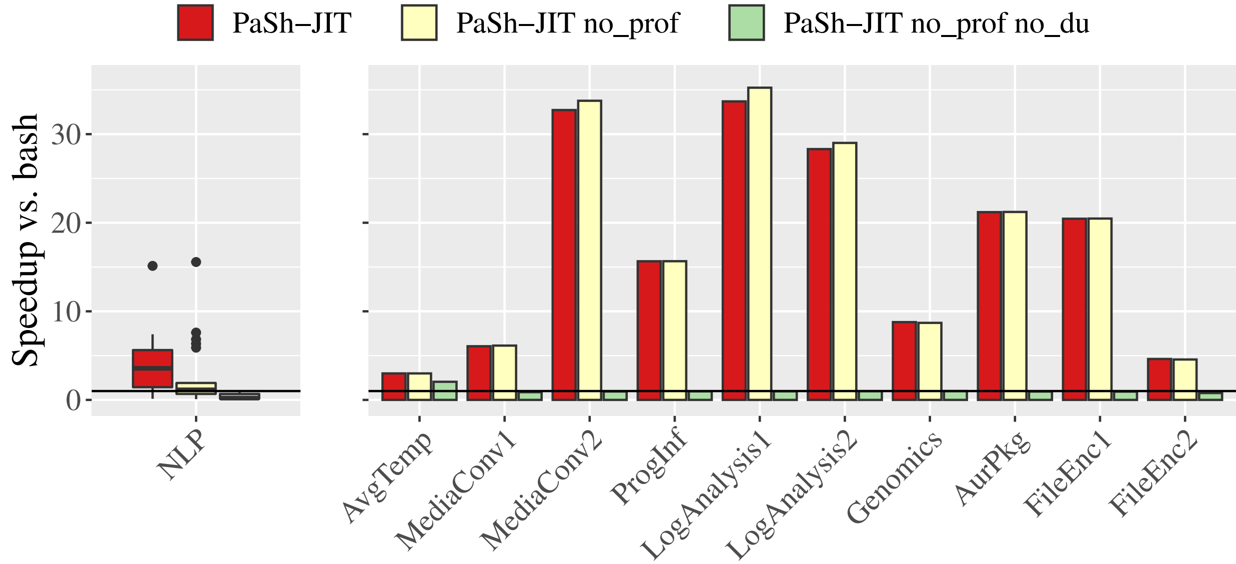


Fig. 7.7: PaSh-JIT Dynamic Optimizations. PaSh-JIT speedup over Bash when toggling profile-driven compiler configuration and dependency untangling for Tab. 7.1 row 5 (left, box) and 6, 8–13 (right, bar) (*Cf.*§7.7.3).

disable all dynamic optimizations to isolate the benefits of commutativity, and compare with the sequential Bash baseline.

Commutativity-aware PaSh-JIT achieves an average speedup of 4.52× and a maximum of 14.68× (Fig. 7.8). Without commutativity-related optimizations, PaSh-JIT achieves an average speedup of 3.72× and a maximum of 15.38×. Commutativity improves the average case but not cases that already see high speedups, as these (1) have negligible overheads coming from input reading—most overheads come due to line processing—and (2) commutativity extensions add some overhead due to the `c_wrap` primitive.

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

JIT engine overhead: To evaluate the benefits of PaSh-JIT's runtime optimizations, we design a worst-case parallelization benchmark: a script that contains a `for` loop that performs 100 iterations of `echo hi`. A tight loop with a minimal-overhead body emphasizes the JIT engine overheads by allowing no parallelization gains. The table on the right shows the run-time performance of four PaSh-JIT configurations compared to Bash: (1) PaSh-JIT without custom expansion, compilation server, and dynamic optimizations, (2) PaSh-JIT without compilation server,

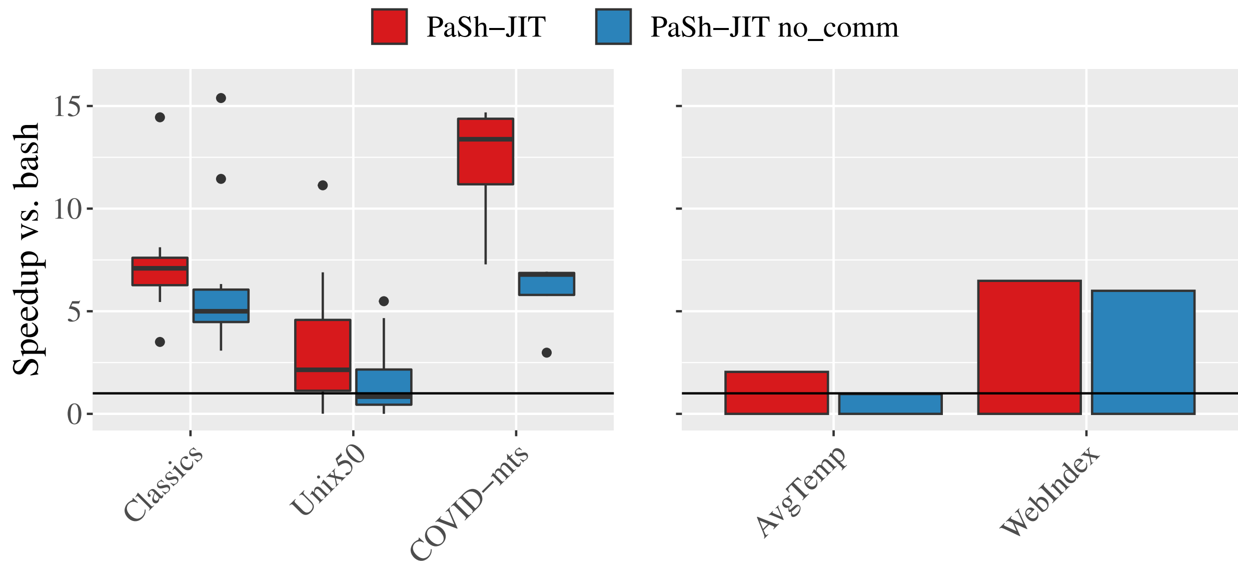


Fig. 7.8: PASH-JIT Commutativity Awareness. PASH-JIT speedup over Bash when toggling commutativity awareness for Tab. 7.1 rows 2–4 (left, box) and 6, 7 (right, bar) (*Cf.*§7.7.3).

and dynamic optimizations, (3) PASH-JIT without the dynamic optimizations, and (4) the complete PASH-JIT. PASH-JIT's runtime optimizations (custom expansion, compilation server, and dependence untangling) improve performance by 12× (over the `-esd` configuration without them). As `echo hi` writes to stdout, dependence untangling does not manage to run it in parallel, and thus its benefit is only due to pipelining. Even then, PASH-JIT's JIT engine overhead is not negligible (about 47ms per JIT invocation), as it needs to save the state and invoke the compiler for every iteration of the loop body.

7.8. Related work

Parallel shell scripting: Recent work addresses significant challenges related to automatic shell script parallelization. POSH [144] and PASH [173] are mostly-automated ahead-of-time shell-script parallelization systems; as described earlier, these systems focus on fully expanded shell pipelines that do not make use of dynamic features. Recent work explored an order-aware dataflow model as a foundation for modeling the transformations these systems perform and proving them correct [76]. To enable divide-and-conquer parallelism, KumQuat [156] proposes a program-synthesis technique for generating aggregators for black-box commands.

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

Other work on shell script parallelization either requires manual effort or is applicable to a smaller subset of scripts than our work. Such work includes: utilities like `qsub` [63], SLURM [184], and `parallel` [164]; shells with non-linear pipe topologies [50, 117, 159]; and using the shell itself as a DSL for concurrency [69].

Unix-related parallelization: There has been a significant body of work on parallel (and distributed) Unix and Unix-like environments [136, 128, 24], including shell-oriented efforts such as Plan9's `rc` [140]. Contrary to PaSh-JIT, these systems did not (aim to) offer full compatibility with the sequential Unix shell. They also focused on systems-level and program-runtime support, rather than automated program analyses and transformations.

Just-in-time compilation: Just-in-time compilation has been studied for long time [23], mainly in two contexts: (1) as a compilation technique for interpreted languages such as JavaScript [65], where critical type information is unavailable prior to execution; and (2) as a performance optimization over ahead-of-time compilation, allowing for specialization [167, 86], loop unrolling and function inlining [32, 141], and other profile-guided optimizations [135, 97]. PaSh-JIT draws inspiration from work in both contexts—resolving unavailable dynamic information at run-time and performing additional optimizations. It also leverages the optimistic compilation technique employed commonly by just-in-time compilers: when it fails to compile (parallelize), it simply runs the original fragment using the shell interpreter as a fallback option. PaSh-JIT differs from most JITs, dealing with different challenges: it operates at a higher level of abstraction, in a unique programming environment with no single unified runtime.

PaSh-JIT also draws inspiration from staged compilation [42] and partial evaluation [91]. These techniques perform some compilation ahead-of-time, waiting for the runtime to specialize and further optimize when there is more information about the environment of the target program and how it is used.

Parallelization in other contexts: More general parallelization support can be grouped into two categories: languages and tools. One approach to parallelization support is to use tools that requires writing in a

new higher-level programming language [60, 168, 102] or a dataflow-based model embedded in an existing language [46, 186, 129, 40, 163, 25]. These tools usually offer automation, but require re-expressing existing computations in domain-specific programming models; PASH-JIT operates on completely unmodified POSIX shell scripts that use unusual features and obscure corner cases.

Another approach to parallelization support uses tools that provide automatic parallelization for standard sequential code, requiring no program modifications but often posing limitations with respect to the granularity of the parallelism that they can extract. The general approach started with explicit `DOALL` and `DOACROSS` annotations [38, 111], continuing with analysis-based compilers [137, 75, 149], and more recent work using profiling-guided speculation [121, 169, 100, 89, 18]. PASH-JIT draws inspiration from this line of work: it does not require manual modification to user code, and it leverages run-time information to optimize and parallelize user scripts. Existing tools work on imperative code with memory accesses, but PASH-JIT works at a higher level of abstraction: commands that affect the file system and the broader executing environment.

Shell correctness and POSIX compliance: Smoosh [71] offers a formalized, executable reference semantics for the POSIX shell, aiming to address subtleties in the standard [22]. PASH-JIT leverages Smoosh to identify and resolve issues in its JIT engine (§7.4) and to guide its early expansion routine (§7.5.2). It also builds on Smoosh's analysis to leverage the POSIX test suite for characterizing shell behavior.

PASH-JIT reimplements Smoosh's `libdash` [70], which presents dash's parser as a library (§7.3.3). We chose `libdash` over Morbig [148] because (1) `libdash` reuses dash's production-grade parser, and (2) `libdash` supports line-oriented input, but Morbig is strictly ahead-of-time.

7.9. Discussion

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

Interactivity: PASH-JIT's design goals (§7.1) do not include interactivity; an interactive shell switches between consuming its input (shell commands) and redirecting it to its executing commands—challenging for PASH-JIT's loose coupling. Furthermore, avoiding shell modifications leads to additional runtime overhead (since the state of the shell has to be reflected upon and is not accessible with a single dereference). Adding robust support for interactivity and improving runtime overhead would likely require a more intrusive design, *e.g.*, altering Bash's source and interposing directly. However, such a design would make PASH-JIT Bash-specific, requiring users to install a new shell, and would significantly complicate the engineering and maintenance effort involved.

Expansion: Some of PASH-JIT's expansion behaves in a way not exactly as specified by POSIX, although we conjecture (and our evaluation confirms, §6.5) it is safe. For example, pipelines are supposed to expand each component in its own subshell (though the last component may run in the outer shell, depending on a shell's implementation choices). PASH-JIT's expansion operates on each component of the pipeline early; each component uses its own copy of the shell environment, to simulate the subshells. We haven not proved these early expansions sound, and it would be interesting future work to pursue that, *e.g.*, by using Smoosh's semantics.

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

CHAPTER 8

DiSh: Scaling out shell programs on a distributed cluster

Material from this chapter was previously published as "Tammam Mustafa, Konstantinos Kallas, Pratyush Das, and Nikos Vasilakis. DiSh: Dynamic Shell-Script Distribution. In 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23), pages 341–356, Boston, MA, April 2023. USENIX Association." [131]. Tammam Mustafa, a master's student at the time, was the first author of this paper and the lead developer of the DiSh system; Nikos Vasilakis (Tammam's supervisor) and I worked extensively with Tammam to develop the ideas and the system behind this paper.

8.1. Introduction

Chapters 6 and 7 describe how shell scripts can be parallelized to effectively harness resources of a single multiprocessor. However, today's workloads have grown to such an extent that the resources of multiple computers are required to process them effectively. Such scale-out is often necessary not only to accelerate computations, but also to compute over data that either do not fit on a single computer or are naturally distributed across multiple computers.

State of the art: Shell users dealing with large datasets that do not fit on a single computer are left with only a few options (Tab. 8.1). One option is to use a distributed shell [172, 62, 50]. Distributed shells require rewriting scripts manually and only support a small subset of Unix features—often with limited, if any, dynamic features and varying support for composition constructs. A recent distributed shell named POSH [144] can handle a subset of shell scripts without rewriting—although that subset is limited to dataflow-only computations and also does not include arbitrary dynamic shell behaviors. In addition, since POSH is a shell reimplementation, it is not behaviorally equivalent with existing shells and thus risks breaking ported scripts. A second option is to rewrite (parts of) the script in a cluster-computing framework [46, 186, 129, 182]. These only support pure computations (*e.g.*, batch, stream), require manual rewriting, and only rarely [74, 85] support language-agnostic components. Another option is job scheduling tools [164, 84, 63, 184], but these operate at a coarse granularity and do not leverage parallelism available in individual commands. Yet another option is to rewrite scripts in languages that support distribution [177, 17, 122, 127], foregoing the

Tab. 8.1: Available options for scaling out shell programs. Compatibility: support unmodified shell scripts. Granularity: support fine-grained distribution. Expressiveness: support arbitrary dynamic behaviors. Agnosticism: support components in any programming language. Equivalence: behavior equivalence with existing shells.

Approach	Compatibility	Granularity	Expressiveness	Agnosticism	Equivalence	Examples
Distributed Shells	□	■	■	■	□	[50, 172, 62]
POSH	■	■	□	■	□	[144]
Cluster Comp. Frameworks (CCF)	□	◪	□	□	□	[186, 129, 161, 182]
Language-agnostic CCFs	□	◪	□	■	■	[74, 85]
Job Scheduling Tools	■	□	□	■	■	[164, 84, 63, 184]
Other languages	□	■	■	□	□	[177, 166, 52]
DiSh	■	■	■	■	■	

shell's succinctness and language agnosticism. To summarize, these options operate on a subset of the shell, require significant manual effort, risk breaking correctness, or—most often—suffer from a combination of these limitations (see §8.8 for more details).

Dynamic shell-script distribution: This chapter presents DiSh, a system designed to scale out shell scripts operating on distributed filesystems while maintaining full POSIX compatibility. DiSh satisfies all requirements in Table 8.1: it operates on existing shell scripts; it distributes scripts at the granularity of individual commands; it handles arbitrary dynamic shell features such as substitution and expansion; it allows the use of commands and utilities of any language; and, most importantly, it is behaviorally equivalent to Bash.

DiSh first instruments the execution of a script to identify regions that may benefit from distribution. At runtime, it compiles these regions to an intermediate representation which it then optimizes to introduce appropriate parallelism, buffering, communication, and coordination. DiSh then executes each compiled region in a distributed fashion using the same shell interpreter, components, and data as the original script.

Implementation and results: DiSh is implemented as a shim layer (following the architecture introduced in Chapter 7) that wraps and orchestrates the (completely unmodified) user shell, delegating all execution to the underlying shell available on each computing node. This design hides distribution from the user and avoids modifying the underlying shell interpreter: the user thinks that their original script is being executed

(but faster); each underlying shell is given a part of the distributed script to execute. As a result, DISH achieves a new milestone in automated shell-script distribution: it offers significant performance benefits, it avoids modifications to shell scripts, and it maintains full POSIX compatibility. Additionally, this modular design allows further research and improvements without modifications in the underlying shell.

We characterize DISH's performance on a 4-node on-premise cluster and a 20-node cloud deployment using 76 scripts—including ones not trivially expressible in modern distributed computing frameworks, such as scripts with `for` loops, side-effects, and complex third-party components. DISH surpasses the speedups achieved by production-grade systems on existing benchmarks and extends speedups to new ones: it achieves significant speedups over (1) Bash (avg: 13.6×; max: 136.3×), a single-node shell-interpreter baseline; (2) PASH (avg: 8.9×; max: 108.8×), a shell-script parallelization system; and (3) Hadoop Streaming (avg: 7.2×; max: 32.3×), a cluster computing framework that supports language-agnostic components and shell scripts. Moreover, whereas Hadoop Streaming does not support 27/76 scripts and requires rewriting 7/76 scripts, DISH runs all scripts without any modifications; in fact, DISH is able to execute the entire POSIX shell test suite, only diverging in one error code out of thousands of assertions.

Chapter outline: The chapter begins with an example and overview (§8.2) of DISH's use and techniques. Sections 8.3–8.6 present DISH's key components:

- Dynamic orchestration (§8.3): DISH parses, pre-processes, expands, and orchestrates its input script to enable dynamic distribution at runtime. This architecture heavily builds on PASH-JIT that was described in Chapter 7.

- Compilation (§8.4): During script execution, DISH compiles certain regions to an intermediate representation and applies a series of optimizations. The compiler core is the same as the one described in Chapter 6.

- Distribution (§8.5): DISH distributes each region to a set of workers in a way that promotes co-location of processing primitives and the data blocks these operate on.

- Runtime support (§8.6): DISH bundles additional runtime primitives supporting correct and efficient

communication in the context of distributed shell script execution.

We then present DISH's evaluation (§8.7) and related work (§8.8).

DISH limitations: DISH currently does not tolerate failures such as worker aborts or network partitions. In such occasions, users are expected to rerun their scripts similar to how they do in non-distributed executions: due to the shell's dynamic features and its support for third-party components, users often re-run failing scripts from the start. The current DISH prototype does not implement support for security features such as encryption and containment.

Availability: All the work described in this chapter has been implemented and incorporated into PASH, and can be found at https://github.com/binpash/dish.

8.2. Background, Example, and Overview

DISH allows everyday shell scripts to reap the benefits of distributed computing: processing datasets that do not fit on a single machine, often also speeding up expensive computations.

Intended use: DISH is designed to support a variety of use cases, depending on the details of the distributed environment on which the system is executing. The most common case is one where input data are downloaded and stored in a distributed file system such as HDFS[11] and then processed using various analyses. This is useful for datasets that do not fit on a single computer, that are naturally distributed across multiple computers, or that can be processed faster in a data-parallel fashion. DISH will distribute the computation appropriately, often running data-parallel instances on multiple machines and multiple processors per machine. DISH also supports hybrid operation where data resides on both distributed and local file systems; this is useful for computations that contain CPU-intensive stages over datasets that do not necessarily reside on distributed file systems.

Example script: Fig. 8.1 shows a shell script that calculates maximum and average temperatures across the US, on datasets hosted on the National Oceanic and Atmospheric Administration (NOAA). The script is split into three parts: (p. 1) an 11-stage pre-processing pipeline to download data from NOAA and store them on HDFS, with the data range controlled upon invocation via dynamic arguments `$1` and `$2`; (p. 2, 3)

[11] The choice of HDFS is not binding. DISH could work on top of any distributed file system (*e.g.*, NFS or Alluxio [108]) that exposes the locations of file blocks. To achieve performance benefits due to co-location, there also needs to be available compute on the nodes that host that file system.

two 5-stage pipelines calculating and storing maximum and average temperatures to the local file system.

HDFS is a distributed file system for handling large data sets on commodity hardware. Scripts like the one in Fig. 8.1 that process files stored in distributed file systems spend most of their execution time moving files across the network. On a 4-node cluster (§8.7) and 3.6GB of input, running just `hdfs dfs -cat` takes 346s; computing pipeline 2 (maximum temperature) only adds 6s. This phenomenon is due to pipeline parallelism: the execution time of all concurrently executing commands is mostly shadowed by `hdfs dfs -cat`.

Opportunities for scale-out: There are ample opportunities for improving the performance of this script. Since all parts contain stages that operate on large datasets, we should be able to execute (at least some of) their stages in a data-parallel fashion. For example, we should parallelize commands that process their input independently, such as `cut` and `grep`, by having them operate in parallel over partial inputs.

Additionally, carefully colocating computation and data should also improve performance. For example, we should schedule the data-parallel execution of the aforementioned `cut` and `grep` instances on machines that store the respective data segments. Directly operating on distributed file segments, rather than gathering and processing data on a subset of the machines, eliminates most data-movement overheads.

Finally, the execution of program fragments that do not depend on each other could become concurrent: since parts 2 and 3 are independent on each other, we should be able to overlap their execution in a task-parallel fashion.

Key challenges: Unfortunately, exploiting these opportunities to scale out execution automatically is particularly challenging in the context of the shell. First, exposing opportunities at the level of individual commands such as `cut` and `grep` is challenging—and this is why prior systems often focused on coarser, script-level or job-level granularity [63, 184].

Second, pervasive dynamic features, file-system introspection, and other side-effects impede traditional distribution approaches based on static transformation—this is why prior shell-script distribution work [144, 74] focuses on side-effect-free dataflow subsets. These challenges are compounded by the presence of more elaborate control flow such as `for` loops, `break`, and `trap` statements present in ordinary shell scripts.

```
NOAA=${NOAA:-http://ndr.md/data/noaa/}
TEMPS=${TEMPS:-/noaa/temps.txt}
hdfs dfs -mkdir /noaa

## Pipeline 1: Download temperature data
##             and store to HDFS
seq $1 $2 | sed "s;^;$NOAA;" |
 sed 's;$;/;' | xargs -r -n 1 curl -s | grep gz |
 tr -s ' \n' | cut -d ' ' -f9 |
 sed 's;^\(.*\)\(20[0-9][0-9]\).gz;\2/\1\2\.gz;' |
 sed "s;^;$NOAA;" | xargs -n1 curl -s |
 gunzip | hdfs dfs -put - $TEMPS

## Pipeline 2: Compute maximum temperature
##             over all data
hdfs dfs -cat $TEMPS | cut -c 89-92 | grep -v 999 |
 sort -rn | head -n1 > max.txt

## Pipeline 3: Compute average temperature
##             over all data
hdfs dfs -cat $TEMPS | cut -c 89-92 | grep -v 999 |
 awk "{ t += \$1; i++ } END { print t/i }" > avg.txt
```

Fig. 8.1: Example script: Downloading a temperature dataset, storing on a distributed file system, and running analysis to extract statistics.

Third, behavioral equivalence with existing shells is practically unattainable, especially with new shell implementations; after all, even production-grade shells such as Bash and zsh diverge subtly in their POSIX behavior [71]. A new distributed shell [144, 50] has little hope of *not* breaking some scripts.

DiSh overview: To overcome these challenges DiSh (1) extracts details about the behavior of commands through command annotations, (2) deals with dynamic features and side-effects by analyzing scripts at runtime using dynamic orchestration, and (3) achieves behavioral equivalence with Bash by only performing script transformations and delegating execution to the underlying interpreter. DiSh is designed to dynamically orchestrate, compile, schedule, and support the execution of shell scripts (Fig. 8.2). DiSh's orchestration (§8.3) kicks in when a potentially distributable script region is identified, saves a snapshot of the user's shell environment (variables, configuration) and invokes the DiSh compiler with the candidate region (Fig. 8.2a). The compiler analyzes this region and if possible, translates it to a dataflow graph—which it then optimizes to introduce parallelism, buffering, *etc.* (§8.4), finally passing it off to the scheduler (Fig. 8.2b); or aborts compilation (Fig. 8.2d) because it cannot guarantee that the region is pure, *i.e.*, side-effect-free. The

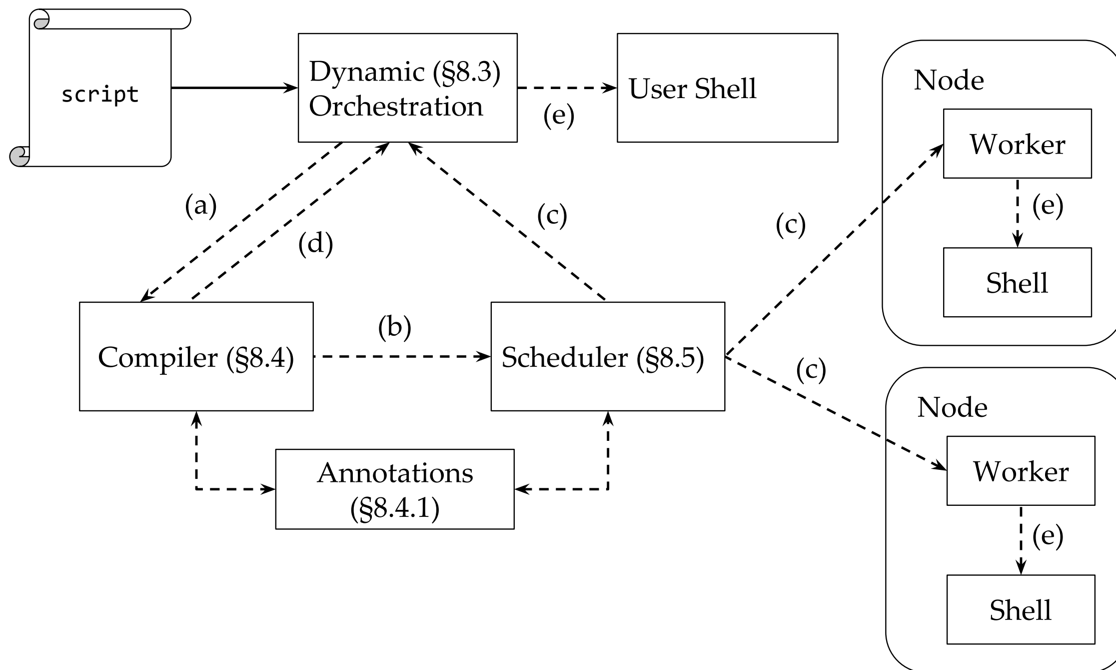


Fig. 8.2: DISH architecture overview. Steps: (a) compile script region; (b) schedule compiled dataflow; (c) send dataflow subgraphs to workers; (d) compilation failed, fall back to original region; and (e) execute script region (compiled or original).

scheduler (§8.5) divides the compiled dataflow graph into different subgraphs which it sends to available cluster workers (Fig. 8.2c). In response to these execution requests, workers apply a second pass of optimizations to better utilize available resources, translate the dataflow graph back to a shell script (Fig. 8.2e), load the snapshot of the shell environment stored by the orchestrator, and execute the script using the local, unmodified shell interpreter (§8.6).

Applying DISH: DISH preprocesses the script in Fig. 8.1 to identify script regions that could benefit from distribution—in this case, all three pipelines. It then replaces each of these regions with calls to the dynamic orchestrator and attempts to distribute them at runtime. During execution, the orchestrator queries the DISH compiler to determine whether a region is pure and thus distributable: if the compiler succeeds, it translates the region to a dataflow graph. Since regions contain arbitrary black-box commands, DISH cannot analyze them directly. Instead, it employs a command specification framework that contains partial specifications of command invocations such as their inputs and outputs. For example, DISH's compiler uses these specifications to determine that `hdfs dfs -cat /noaa/temps.txt` reads from the HDFS file `/noaa/temps.txt`

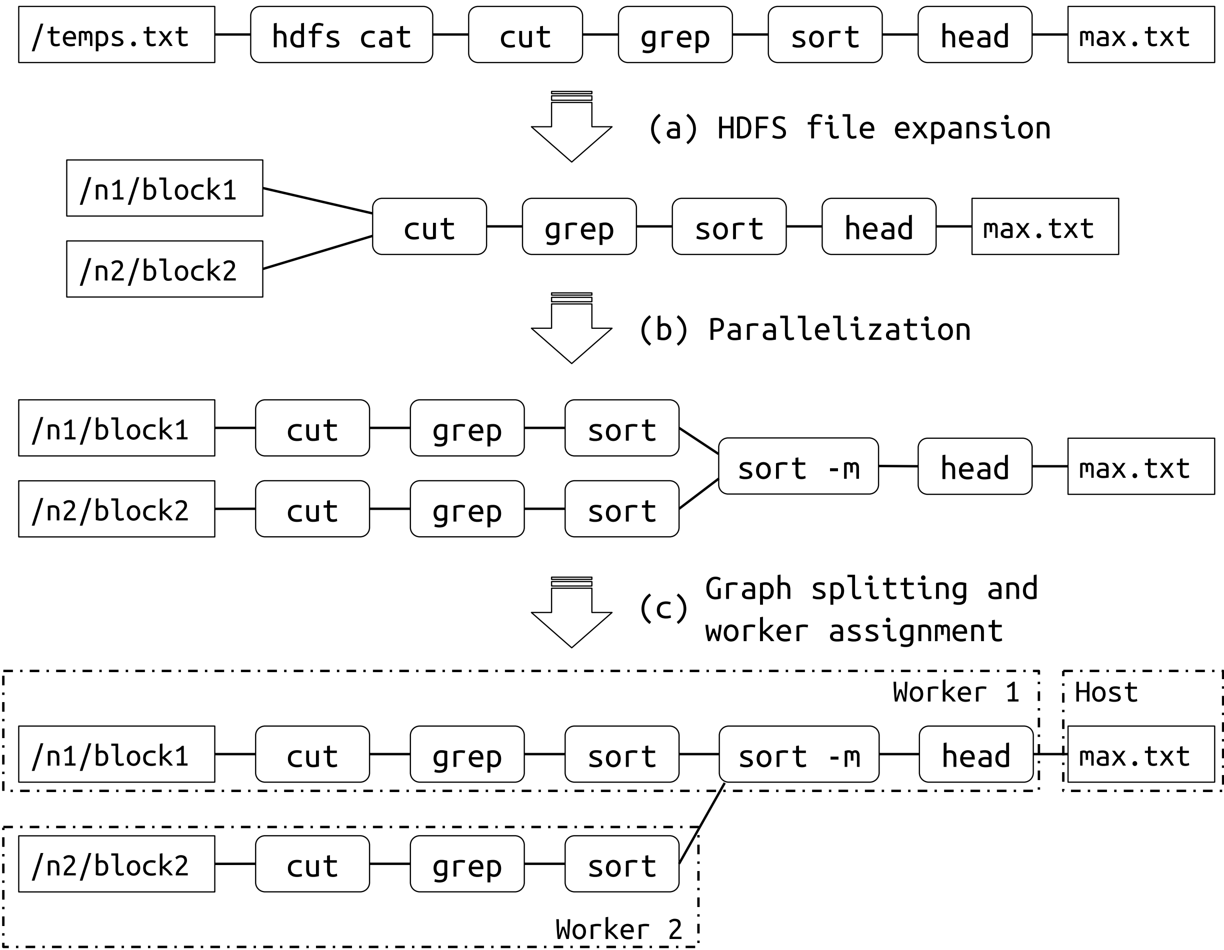


Fig. 8.3: DISH dataflow graph stages. (a) HDFS files are expanded to sequences of blocks. (b) the graph is parallelized based on the command specifications. (c) the scheduler splits the graph and assigns subgraphs to workers.

and writes to `stdout`. Once a region is in dataflow form, DISH applies transformations to distribute it.

Fig. 8.3 shows the distribution stages for pipeline 2 (maximum temperature). DISH first detects operations on HDFS files (*i.e.*, HDFS `cat`) and expands each distributed file to its segments (datablocks), often stored on different physical machines. Informed by command annotations, DISH applies parallelization transformations: commands like `cut` and `grep` are parallelizable directly and can be executed on the machine with the raw input datablock. The scheduler then splits the compiled graph into subgraphs and maps them to workers in a data-aware fashion. Finally, each worker translates the graph back to a shell script, adds additional runtime primitives (commands), and executes it locally.

The result? DISH drops the execution of pipeline 2 from 352s to 6s while maintaining full behavioral equivalence and requiring no modifications to the user shell.

8.3. Dynamic Shell Orchestrator

To facilitate adoption, an important desideratum in the design of DISH is to achieve behavioral equivalence with the underlying shell interpreter. To achieve this, DISH is not designed to operate as another shell, but rather wraps the user's existing shell interpreter and the shell interpreters on the worker machines. As a result, DISH hides parallelization and distribution from both the user and the underlying shells: the user thinks that their original script is being executed—just faster—and each underlying shell simply executes a standard non-distributed shell script. This allows DISH to achieve exceedingly high compatibility with the underlying shell implementation (§8.7.3), while also minimizing maintenance costs since updates and modifications on the underlying shell are reflected in DISH without any change. DISH's orchestrator architecture heavily builds on the PASH-JIT that was described in Chapter 7, extending it with environment sharing.

Fig. 8.4 shows an overview of the structure of DISH's dynamic orchestration. To achieve dynamic shell script orchestration without any shell-interpreter modification, DISH opts for a light-weight script instrumentation pre-processing step: it instruments *potentially* distributable regions with invocations to the orchestration engine. It chooses regions with the goal of maximizing distribution benefits: intuitively, it focuses on commands and pipelines rather than control-flow statements and variable assignments. However, the choice of these region boundaries is not binding—the preprocessor just needs to be precise enough to determine potential regions, but DISH will eventually decide whether or not (and if yes, how) to distribute a candidate region at runtime. The preprocessor first parses the original script, it then replaces the relevant program regions with orchestration prefixes, and then un-parses (emits) it back as an instrumented script that is given for execution to the user's shell interpreter.

The instrumented script then makes calls to the orchestration engine. The orchestration engine is itself a shell script coordinating with the compiler and worker manager and attempting to distribute the upcoming region (see §8.4 and 8.5 for details). If it succeeds, it runs the distributed version of the region. If it aborts, it just falls back to the original region, executing it normally. Reasons for aborting include the region being side-effectful, *e.g.*, modifying some environment variable, or lacking relevant command annotations.

Preprocessor: The preprocessor searches for maximal potentially distributable regions by processing the AST bottom-up, combining distributable subtrees when they are composed using constructs that do not introduce scheduling constraints (*e.g.*, `&`, `|`). When a region cannot outgrow a certain subtree, DISH replaces it with a call to the orchestration engine. If the region is successfully compiled (at runtime), DISH translates it to a dataflow representation—a convenient and well-studied model amenable to transformation-based optimizations [76]. At a later point, DISH running on each node translates the instrumented AST resulting from the compilation back to shell code and passes it to the underlying shell for execution.

Parsing library: DISH invokes parsing and unparsing routines frequently, and therefore needs them to be very efficient. To that end, it uses an internal Python implementation [95] of POSIX-shell-script parsing and unparsing based on `libdash` [70, 71]. The DISH parser contains several optimizations such as caching, inlining, and careful array appending to achieve improved performance.

Orchestration engine: DISH's orchestration engine is designed to maintain the original script behavior and minimize runtime overhead—as it is invoked multiple times per script. The engine is a reflective shell script: it coordinates transparently with the compiler to determine whether or not to parallelize a script by inspecting the state of the shell and that of the broader system. DISH constantly switches between two execution modes when executing scripts: (1) conventional shell mode, where scripts execute in the original shell context, and (2) DISH mode, where the runtime reflects on shell state and invokes the compiler to determine whether to execute the original or an optimized version of the target region. To switch from shell mode to DISH mode, the engine saves the state of the user's shell; to switch back, it restores the state of the user's shell. The state of a shell is quite complex: apart from saving and restoring variables, DISH must account for various shell flags along with other internal shell state (*e.g.*, the previous exit status, working directory). During an invocation, the engine first switches to DISH mode, communicates with the compiler and scheduler to determine whether a region can be safely distributed, and it then switches back to shell mode to execute the original or distributed version of the script.

Environment sharing: The distributed version of the script region might execute on a different shell (or even machine). Therefore, a challenge that DISH needs to address is to make sure that all regions execute in the correct environment—including access to the latest variable values and function definitions. To achieve that the engine takes a snapshot of the environment right before execution. It then transfers the snapshot to

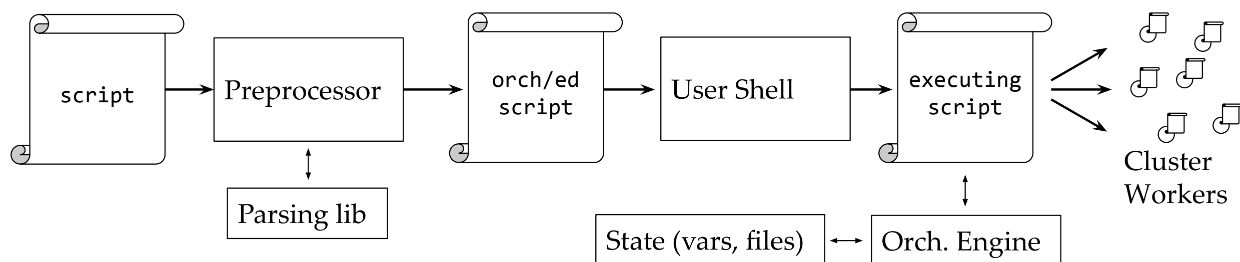


Fig. 8.4: Dynamic orchestration overview. DISH instruments scripts with calls to the orchestration engine, which passes program fragments to the worker manager at run-time.

the distributed workers, which they load before executing the incoming script fragment. This is safe to do since successful distribution of a region implies that it is pure (and therefore does not affect the environment), and thus the snapshot will be valid until the region finishes execution.

String expansion: To correctly determine if a script region is safe to distribute, the compiler needs to expand all strings in that region. Since DISH performs compilation and distribution of each script region at runtime, right before execution, it has access to all the latest variables and system state to fully expand all strings in the region. DISH only implements a common and safe subset of all available expansions, and avoids implementing side-effectful expansions that have the risk of affecting the environment (*e.g.*, `${x=foo}`: set `x` to `foo` if `x` is unset). Note that DISH keeps expansion local: it does not expand regions succeeding the target region, as these might depend on the execution of the target region.

8.4. Compiler

This section describes the compiler of DISH, which builds on the PASH parallelizing compiler (see Chapter 6). The compiler is given the AST of an input script fragment and information about the commands in that fragment (§8.4.1). It then attempts to transform it to a dataflow graph (§8.4.2), an intermediate representation amenable to parallelization transformations. If the compiler succeeds in transforming a script region to a parallel dataflow graph, that graph is then passed to the scheduler which then decides how to map subgraph components to the available worker nodes. As the compiler operates at runtime in a just-in-time fashion, it exploits ample opportunities for parallelization even across subgraphs (§8.4.3).

8.4.1. Command Annotations

DiSh needs to support analyses and transformations over third-party commands, without access to their source code. To achieve this, DiSh uses the specification framework that was described in Chapter 5, which we briefly overview here for completeness. Command annotations act as an intermediate layer that provides restricted but sufficient information about the behavior of a command to analysis and transformation systems like DiSh. They also enable reuse, as they are not tied to a particular analysis and can thus be reused by different tools. For this work, DiSh reuses the set of annotations developed for PaSh-JIT (Chapter 7) extended with annotations for commands that appear in the evaluation of DiSh (§8.7).

A command annotation in DiSh encodes information at the level of individual command invocations, *i.e.*, precise instantiations of a command's flags, options, and arguments. Among other information, annotations determine how a command invocation affects its environment, and specifically whether it is pure, *i.e.*, whether it only affects its environment by writing and reading to and from a well-defined set of files—information which DiSh uses when translating commands to and from dataflow nodes (§8.4.2). For example, the annotation for `grep` can be used to extract that the script fragment `grep -f dict.txt src.txt > out.txt` contains two input files `dict.txt` and `src.txt` and one output file `out.txt`. This knowledge of input and output files is used by DiSh to enable location-aware distribution, by scheduling the computation on nodes that contain relevant data blocks. Additionally, annotations describe parallelization opportunities—*e.g.*, that `grep "pattern" src.txt` processes each line of `src.txt` independently and thus can be parallelized at a line boundary.

8.4.2. Dataflow Model

The core of DiSh's compiler is the order-aware dataflow model introduced in Chapter 4 that captures pure shell script regions that read from a well-defined set of input files and write to a well-defined set of output files—*i.e.*, they do not modify their environment in any other way. This model is expressive enough to capture a shell subset used pervasively in data processing scripts.

In this model, nodes represent commands and edges represent files, pipes, named FIFOs, and file descriptors. The model is order-aware in the sense that it keeps information about the order in which nodes read from their

inputs, which is important for the script's semantics. For example, `grep "pattern" in1.txt - in2.txt` first reads from `in1.txt`, then from its standard input, and then from `in2.txt`. This order awareness allows DISH to perform transformations that optimize execution of a script—*e.g.*, by exposing parallelism—but preserve its original behavior.

Translation workflow: Given an AST representation of an input script region, the compiler uses annotations to deduce whether commands are pure *i.e.*, they only affect their environment through a well-defined set of output files, and attempts to transform them to dataflow nodes. If all commands in the region are pure the compiler transforms the region to a dataflow graph. It then applies transformations (described below), optimizing the graph to expose parallelism and improve the script's performance. Finally, it serializes the graph back to a (now optimized) shell script, by translating every node back to a command and connecting them all together with appropriate channels (*e.g.*, FIFOs, RFIFOs, redirections).

Transformations: DISH's transformations enable data-parallel execution by replicating nodes in the graph and adding appropriate split and merge nodes around them. They apply a pass over the graph to remove pairs of inverse nodes—*i.e.*, pairs of nodes whose semantic effects cancel out but whose performance effects are additive—for example, a concatenation-style merge followed by a linear split. For commutative commands, *i.e.*, commands that produce the same output irregardless of their input-line order, DISH applies transformations that pack and unpack metadata across the graph—achieving better performance by avoiding unnecessary blocking and buffering. Finally, to improve the flow of data across the graph, DISH applies additional transformations that inject hybrid memory-disk buffer nodes in points in the graph that are likely to become bottlenecks.

Remote file resources and HDFS files: To support scripts that perform data analysis on a combination of HDFS and local files, DISH extends the dataflow model with remote-file resources (RFRs) that encode file blocks in different nodes. RFRs usually represent blocks of files that are partitioned and replicated in HDFS, and contain information about the location of the data in the distributed environment. This information could contain multiple locations to support replication, and is used by the scheduler to assign script fragments to different workers. When the DISH compiler comes across an HDFS file path, it queries HDFS to determine the locations of its file blocks and then expands that file to a sequence of RFRs, each of which represents a block.

```
for item in $(hdfs dfs -ls -C ${IN});
do
    output_name=$(basename $item).zip
    hdfs dfs -cat $item |
        gzip -c > $OUT/$output_name
done
```

Fig. 8.5: Example of independent regions. This shell script compresses all files in a directory—but each iteration results in an independent body region that can be executed in parallel.

8.4.3. Dynamic Dependency Untangling (DDU)

Scripts often contain regions that are independent, *i.e.*, they have different (file) working sets. Independent regions could potentially run in parallel, better utilizing computational resources and improving the execution times of the scripts in which they belong. However, inferring independence statically and ahead of time is challenging as shell scripts make extensive use of dynamic features. Figure 8.5 shows an example script that contains independent fragments but also features dynamic behavior. This script iterates over all files in an HDFS directory, compresses them using `gzip`, and finally stores them as independent files.

Determining independence statically in this script would require inferring values of environment variables (like `IN` and `OUT`) and the state of the file system, *e.g.*, `hdfs dfs -ls`. DISH's dynamic orchestration (§8.3) circumvents this challenge by making distribution decisions during the execution of the script when environment variables and the file system state are known. DISH further exploits this by discovering independent dataflow regions at runtime and executing them in parallel—even if they were not parallel in the original script.

When DISH successfully compiles a dataflow region (at runtime), it knows that the region is pure and therefore can determine the region's inputs and outputs—and it does so for free, without additional analysis or inference stages. DISH then uses this information to check for read-write or write-write dependency conflicts with regions that are running concurrently. If none is found, DISH passes the region to the scheduler, which orchestrates distributed execution, and then immediately continues the execution of the script until it reaches the next dataflow region. Whenever the compilation of a dataflow region fails, DISH cannot safely detect the input and output information of this region—and thus it needs to wait until every previous region is done executing to ensure that no dependency will be violated.

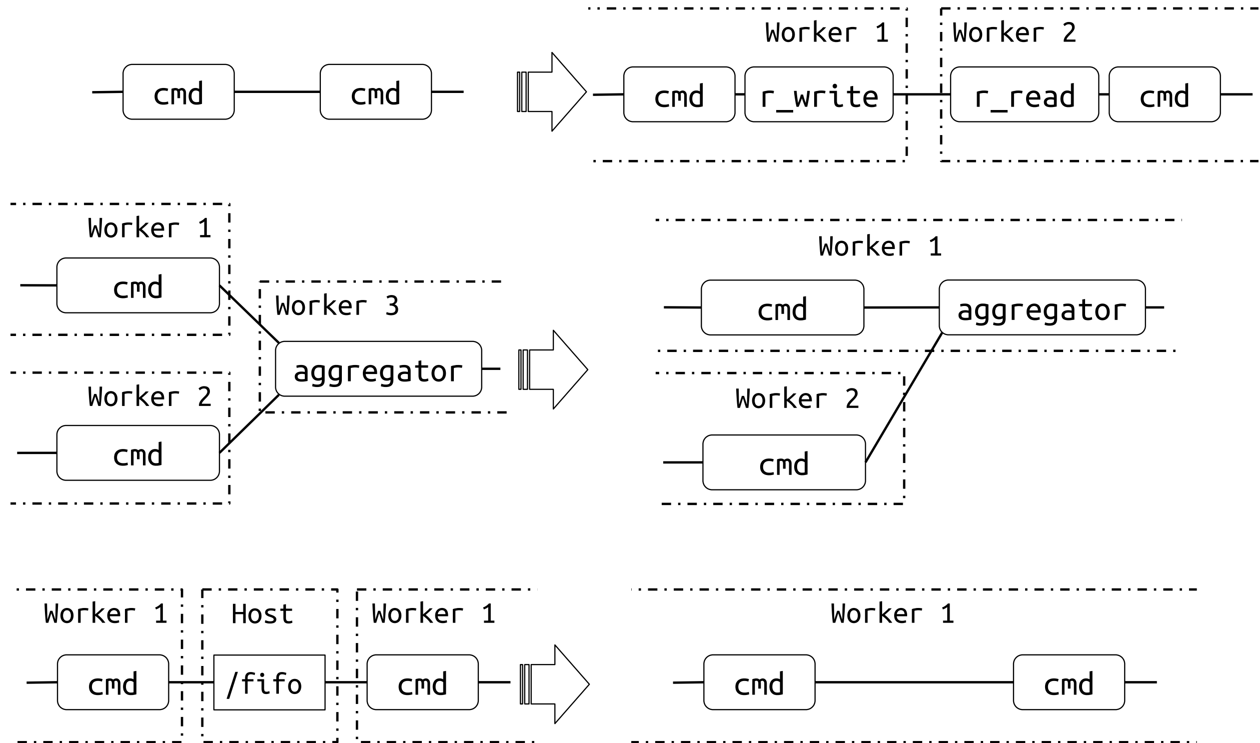


Fig. 8.6: Overview of commutativity-aware transformations. (Top) Remote writes and reads added during distributed scheduling. (Mid) Worker-first aggregation. (Bot) Named FIFO teleportation.

Since DDU is done at runtime it is both sound, *i.e.*, it does not execute dependent fragments concurrently, and precise, *i.e.*, it offers significant benefits due to improved parallelism and resource utilization—especially for scripts that do not contain highly data-parallelizable commands, such as the commands in the aforementioned compression script (Fig. 8.5). Compared to analyses over static languages, DDU cannot identify global optimizations such as reordering the final command in the script to run first. This lack of optimality is not specific to DDU, but applies to any shell script analysis; in fact, as far as we know there is no sound and precise static analysis for shell scripts.

8.5. Distributed Scheduling

This section describes how DISH's scheduler distributes a compiled script to a set of workers. The scheduler is given a dataflow graph that is already parallelized and has HDFS files expanded to sequences of remote file resources (RFRs) representing their blocks. The task of the scheduler is then to distribute this graph with the goal of optimizing performance by both utilizing available resources and moving computation close to

the data. Currently the scheduler knows about the workers in the cluster ahead of time using a configuration file.

The scheduler makes a decision on how to split the graph based on a policy that optimizes performance through co-location of data blocks and the commands that execution over them. The scheduler processes the top-level dataflow graph to generate a set of subgraphs, one for each worker and one for the host machine executing the script. It then replaces edges corresponding to communication channels (*e.g.*, FIFOs, pipes) at the boundaries of each subgraph with remote channels—adding a remote write node on the sender side and a remote read node on the receiver side (see Fig. 8.6, Top). It also inserts remote reads for subgraph nodes that access files stored on remote workers. The final generated subgraph represents the script fragment that is passed for execution to the user shell running on each worker: the compiled script handles all the redirection to and from local files and the standard input, output, and error streams to and from the worker.

Data-aware scheduling policy: The highest performance overhead when executing distributed shell scripts is networked data movement across workers. DISH addresses this overhead by introducing a greedy scheduling policy that allocates subgraphs in a way that attempts to minimize data movement across workers. If a data file (or block) is available on a worker, then DISH maps the maximal dataflow subgraph that starts from that file to that worker—*i.e.*, scheduling as much of the processing as possible on the worker. The scheduler also tracks the amount of work that each worker currently has scheduled, which can vary due to dynamic dependency untangling (§8.4.3): if a data file is replicated across multiple workers, DISH chooses the worker with the least amount of pending work to execute that subgraph.

Worker-first aggregation: The distributed dataflow graphs that DISH executes often contain aggregation (*i.e.*, merge) nodes, similarly to the reduce stages in Hadoop Streaming. Regardless of the worker on which the aggregation is performed, data from different workers will need to be combined onto a single worker and thus these dataflow nodes will necessarily result in data movement. DISH prioritizes performing aggregation on one of the participating workers, because workers already contain a subset of the data used in the aggregation (see Fig. 8.6, Mid). This optimization is particularly beneficial for scripts that filter and aggregate data, often containing commands such as `grep` and `uniq`, because any filtering stages prior to aggregation result in reduced data transfer.

It is worth noting that, absent additional information about commands [144], the location of aggregators involves challenging trade-offs not addressable with a single optimization policy. For scripts that include aggregators that do not reduce data sizes, DISH's worker-first aggregation optimization risks transferring more data. As DISH's evaluation confirms (§8.7), however, worker-first aggregation results in performance benefits for most scripts.

Delegated script concretization: DISH's scheduler sends workers dataflow subgraphs, encoded in DISH's intermediate representation, instead of concrete shell scripts ready for execution. Each dataflow subgraph contains holes that workers are expected to fill in, based on the specifics of their local environment. This choice simplifies DISH's distributed execution, as the scheduler does not need to have up-to-date information about several worker details such as the temporary directories they use. Additionally, this choice enables better resource utilization in a heterogeneous environments with different worker capabilities: a worker can apply another optimization pass to the dataflow subgraph it receives to better manage and utilize its resources.

Named FIFO teleportation: Scripts often use named FIFOs to share data between concurrently executing processes. Named FIFOs introduce a performance challenge, because they are local files that reside on the host machine where the script was executed. Therefore, by default, all data that would normally go through named FIFOs in the original execution would now have to go back and forth between workers and the machine for which the script was developed. DISH addresses this challenge by observing that named FIFOs are ephemeral, *i.e.*, they maintain no data after the execution of a dataflow region. Based on this observation, DISH migrates named FIFOs to workers closer to the data, eventually deleting the migrated versions after the dataflow region has finished executing (see Fig. 8.6, Bot). This transformation, termed FIFO teleportation, improves performance by avoiding unnecessary data movement in scripts that use FIFOs.

8.6. Runtime Support

DISH has to address several runtime challenges: communication among workers, identification of HDFS data block locations, and correctness in view of HDFS blocks split independently of newlines—an assumption necessary for several dataflow transformations. This section describes several components of DISH's runtime that address the above challenges.

Remote FIFO channel: As described earlier (§8.5), connections between dataflow nodes are instantiated using UNIX FIFOs in a single-machine setting. Unfortunately, FIFOs do not support networked operation and thus cannot cross worker boundaries. To address this challenge, DISH introduces a remote FIFO primitive (RFIFO) that is implemented in Go and uses socket-based communication. RFIFOs are intended to operate identically to FIFOs, *i.e.*, implement the semantics of dataflow graph edges, but with support for operation over the network. They have a unique identifier and two ends—a read end and a write end.

Since shell streams are lazy, *i.e.*, a producer blocks until its consumer requests input, the network link is often not fully utilized, lowering throughput and risking introducing significant latency. To avoid these throughput and latency challenges, DISH adds two buffer nodes to the dataflow graph: one before the write end of the RFIFO, to allow uninterrupted access to data, and one after the read end of the RFIFO, to force the read to request data. This lazy-to-strict optimization maintains correctness and improves performance in most cases; in rare cases, it may lead to unnecessary data transfer between nodes—*e.g.*, when there is a `head` command right after the read end of an RFIFO.

Port discovery service: As transformations and optimizations are applied during the execution of a script—contrary to most other distributed environments—DISH's scheduler cannot statically predict which ports will be available at runtime for RFIFOs at each worker: different scripts and script fragments running concurrently during a single execution may collide on port usage. To address that, each DISH worker implements a port discovery service (PDS) that can be accessed by remote FIFOs to (1) advertise their port, and (2) discover the port that their other end uses. The discovery service is implemented in Go with gRPC [2] and supports a few remote procedure calls (RPCs), central among which are a `put` call for advertisement and a `get` call for discovering the port of a remote end. RFIFOs are extended with gRPC clients to advertise ports among local PDS or identify the ports corresponding to their other end by querying the PDS of the respective worker. By deferring port selection until runtime execution, DISH's port discovery service facilitates loose subgraph coupling and simplifies remote subgraph execution on multiple workers.

HDFS data retrieval: During transformations, the DISH compiler (§8.4) needs to retrieve information about HDFS paths to expand them into block sequences. This expansion happens on a critical runtime path and thus needs to be efficient. A prior implementation of DISH invoked this expansion on every HDFS path

using a shell command—by wrapping `fsck`, a command offered by HDFS API for querying the health of the disk in the cluster, returning information about a file and its partitioning into blocks. This implementation ended up incurring significant latency (> 1s), and thus DISH switched to the web API reducing expansion to sub-10ms latency.

Enforcing logical block boundaries: A key challenge when processing separate file blocks in HDFS is the mismatch between compiler assumptions about the block shape and how blocks are actually stored in HDFS: HDFS blocks might not be split on newline boundaries, but the parallelizing transformations performed by the DISH compiler (§8.4) assume that all blocks are logically separated by newlines. This assumption is crucial and depends on the way commands process their input, *e.g.*, `sort` processes its input line by line, and therefore would require a significantly more complex parallelization transformation if its input could be split at arbitrary points. Developing complex custom parallelization transformations for each command would be infeasible in practice due to the sheer number of available commands and would not allow DISH to reuse the parallelization transformations developed for PASH [173].

Instead of relaxing the compiler assumption, DISH addresses the mismatch by ensuring it holds during script execution using additional runtime support. DISH implements a distributed file reader (DFR) primitive that runs as a service on every worker. The DFR service ensures that parallel dataflow nodes only process batches that are split in newline boundaries, independent of how the actual physical blocks are split—providing the illusion of a logical block that ends at a newline to its consumer. Given a distributed file path, DFR reads the local file or block from the worker's disk going beyond the first newline character in its block. If the block is not terminated with a new line, then the DFR communicates with the reader of the next block (and potentially any readers after that), returning a complete logical block to its consumer. When a compiled dataflow graph is translated back to a script, DISH prefixes file paths with a command invoking a DFR client that communicates with the relevant DFR service to retrieve the relevant logical block. Both service and client are implemented in Go, communicating using gRPC and protobufs [66].

8.7. Evaluation

We are interested in evaluating two aspects of DISH: (1) its performance, and (2) its compatibility with Bash.

Tab. 8.2: Benchmark summary. Summary of all the benchmarks used to evaluate DISH, and their characteristics.

	Benchmark	Scripts	Pure HS	LOC	Input	Source
1	Classics	10	7/10	123	3G	[28, 27, 165, 92, 118]
2	Unix50	34	30/34	142	21G	[103, 30]
3	COVID-mts	4	4/4	79	3.4G	[171]
4	NLP	21	-	306	120 books	[44]
5	AvgTemp	1	1/1	31	3.6G	[182]
6	MediaConv	2	-	35	0.8 & 0.4G	[144, 159]
7	LogAnalysis	2	-	63	0.7 & 1.3G	[144, 159]
8	FileEnc	2	-	44	1.3G	[125]

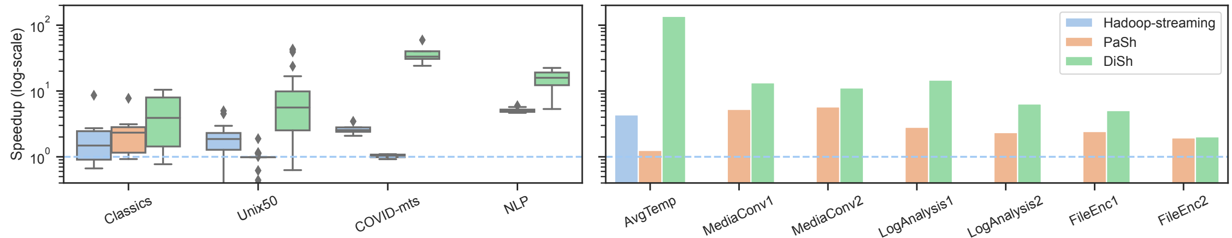


Fig. 8.7: DISH performance on a 4-node cluster. DISH speedup (*vs.* PASH and Hadoop Streaming whenever possible) over Bash for Tab. 8.2 rows 1–4 (left, box) and 5–8 (right, bar) (*Cf.*§8.7.1). (Log y-axis; higher is better.)

Experiments: We perform four experiments using several real-world shell scripts taken from a variety of sources (Tab. 8.2). The first two experiments focus on the performance gains (§8.7.1) achieved by DISH's distribution on (1) a 4-node on-premise cluster, and (2) a 20-node cloud deployment—both over a variety of benchmarks and workloads. We compare DISH's performance against (1) GNU Bash [145], the *de facto* sequential shell-script execution environment; (2) Apache Hadoop Streaming [74] (AHS), a production-grade distributed data-processing framework that supports language-agnostic executables; and (3) in the case of the 4-node setup, PASH-JIT [173, 95], a shell-script parallelization system from the Linux Foundation. For the rest of this section we refer to PASH-JIT as PASH for brevity. PASH's parallelism benefits make it a likely alternative to DISH for smaller clusters, where DISH's anticipated benefits of distribution might be smaller, but this likelihood diminishes as the size of the cluster grows.

The last two experiments evaluate DISH's dynamic dependency untangling (§8.7.2) and DISH's correctness (§8.7.3), *i.e.*, its compatibility with respect to Bash across all scripts and the POSIX shell test suite.

Benchmarks: We use 8 sets of real-world benchmarks, totaling 76 shell scripts and 823 LoC. Classics and Unix50 contain classic and recent (c. 2019) scripts that make heavy use of UNIX and Linux built-in

commands. COVID-mts contains four scripts used to analyze real telemetry data from mass-transit schedules during a large metropolitan area's COVID-19 response. NLP contains several scripts from UNIX-for-poets, a tutorial for developing programs for natural-language processing out of UNIX and Linux utilities. AvgTemp contains a large script downloading and processing multi-year temperature data across the US. MediaConv contains two scripts that process, transform, and compress video and audio files. LogAnalysis contains two scripts that apply typical system-administration and network-traffic analyses over log files. Finally, FileEnc contains aliases that encrypt and compress files.

Baselines and implementations: Bash, PASH, and DISH executed every shell script completely unmodified. Apache Hadoop Streaming (AHS) posed significant expressiveness limitations. Only 42 scripts in Classics, Unix50, COVID-mts, and AvgTemp out of 76 scripts can be implemented natively (Tab. 8.2, col. Pure HS). Another 7 scripts required manual porting by splitting them into mappers, reducers, and additional components: These components were not available natively by AHS—for example, components for reading from two pipelines for `diff.sh` and for sorting after the reducer for `bigrams.sh` (both in Classics). During porting, we put significant care to avoid limiting AHS's parallelism: we modified 3 AHS scripts in Classics to help HS introduce additional parallelism—for example, we manually expanded `tr -cs` into `tr -c | grep -v` (both stateless). None of the scripts in NLP, MediaConv, or LogAnalysis can be implemented in AHS as they perform processing in loops, the iterations of which depend on the files in a statically indeterminable directory (see Fig. 8.5) and are therefore not expressible in AHS. We attempted to replace the body of the loop with an AHS invocation but the startup overhead ended up dwarfing the execution time by a factor of ten on average.

Hardware & software setup: The 4-node cluster consists of four 6-core Intel(R) Core(TM) i7-10710U CPU nodes each with 64GBs of RAM, located in the same room and connected with an average bandwidth of 90.8 Mbits/sec. The 20-node deployment consists of xl170 Cloudlab [51] nodes, each equipped with 10 $\times$ Intel Core E5-2640 2.4 GHz CPUs and 8GB of memory. Single-machine shells (Bash & PASH) were evaluated on a machine with 20 $\times$ 2.80GHz Intel(R) Core(TM) i9-10900 CPUs and 32GB of memory.

For ease of deployment and reproducibility, we used Docker `swarm` to deploy (1) HDFS, and (2) the DISH runtime. The containers were created using the standard Ubuntu 18.04 image. We use Bash v.5.0.3, PASH

Tab. 8.3: DISH performance in 20-node cloud deployment. DISH speedup over Hadoop Streaming for scripts that AHS supports.

DISH speedup over AHS						
Benchmark	Avg	Min	25th	50th	75th	Max
Classics	2.74×	0.92×	2.41×	2.60×	2.85×	6.55×
Unix50	6.64×	0.91×	2.85×	5.38×	10.4×	16.9×
COVID-mts	10.4×	6.64×	8.91×	9.27×	-	16.8×
AvgTemp	7.85×	-	-	-	-	-

v.6e2ecba, and HDFS/Hadoop streaming version 3.2.2. We explicitly disabled checksum verification from HDFS in all configurations, scripts, and measurements. All scripts were executed completely unmodified, using environment variables, loops, and other shell constructs. To minimize statistical non-determinism we repeated the experiments 3 times noticing imperceptible variance (< 1%).

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

8.7.1. Performance

How does DISH's distributed perform on small on-premise clusters and multi-node cloud deployments, and how does it compare to state-of-the-art systems?

Results: Fig. 8.7 (note the log y-axis) shows the performance of DISH, PASH, and AHS on a 4-node on-premise cluster across all benchmarks of Tab. 8.2. Box plots (left) show result quartiles for multi-benchmark suites (Tab. 8.2, rows 1–4) and bars (right) show results for individual scripts (Tab. 8.2, rows 5–8). Across all benchmarks, DISH achieves an average speedup of 13.6× (*vs.* 2.55× for PASH and 2.1× for AHS) and a maximum speedup of 136.3× (*vs.* 7.8× for PASH and 8.6× for Hadoop Streaming). The average execution time of all scripts on Bash is 299s, ranging from 1s for `34.sh` in Unix50 to 2840s for `nfa-regex.sh` in Classics. DISH is only slower than Bash (737s vs 568s) in the case of `diff.sh` from Classics, for which AHS is even slower (766s). DISH achieves a performance comparable to Bash (1-2s) in `4.sh` and `34.sh` from Unix50, because both perform a short-running `head`.

Tab. 8.3 shows the speedup of DiSh *over AHS* on a 20-node Cloudlab deployment across all scripts implementable with AHS (Classics, Unix50, COVID-mts, AvgTemp). Across all benchmarks, DiSh achieves an average speedup of 6.17× and a maximum speedup of 16.95× over AHS. DiSh is slower than AHS only for three scripts: `nfa-regex.sh` from Classics (0.92×), `29.sh` and `30.sh` from Unix50 (0.91× and 0.94×).

Across all scripts in both deployments, DiSh's overheads (startup cost, dynamic orchestration, preprocessing, compilation, scheduling) take less than 1 second.

Discussion: DiSh is faster than Bash, PaSh, and AHS across Tab. 8.2's suites (rows 1–4) with respect to average, and across all of Tab. 8.2 individual benchmarks (rows 5–8)—often by a significant margin (*e.g.*, 134× for AvgTemp against PaSh). DiSh's (and PaSh's) speedup over Bash is due to parallelism. DiSh's speedup over PaSh is due to DiSh' co-location of data and computation: PaSh cannot offload computation and thus first gathers all data onto a single machine—a time-consuming stage—and then starts processing in parallel. DiSh is slower than Bash only for `diff.sh`, because (1) it is not highly parallelizable and (2) it performs no filtering, *i.e.*, its output is the same size as its input. In contrast to Bash, which simply fetches all data and processes it locally, DiSh tries to allocate most commands on the workers, but this leads to increased data movement since moving data between workers does not avoid sending the whole output to the client.

DiSh's speedup over AHS is due to a few different reasons. One reason is the increased expressiveness of DiSh's dataflow model: DiSh accepts and parallelizes complete scripts, discovering more opportunities for parallelism. Many of the AHS scripts are broken into multiple map and reduce stages, often leaving pipeline parallelism and data parallelism unexploited. Another reason is DiSh's dynamic independence discovery, which allows for additional parallelism and better utilization of resources—in ways that AHS does not support; we zoom into these benefits below (§8.7.2). In the Cloudlab deployment, DiSh is (marginally) slower than AHS in only two cases: (1) a script that is embarrassingly parallel and thus implementable in AHS using only a single mapper (`nfa-regex.sh`), and (2) two scripts in Unix50 that see slightly more benefits from our manual, hand-optimized AHS rewrite than they do from DiSh's automated distribution.

We found porting scripts to AHS a serious challenge. Many scripts required significant manual effort, re-

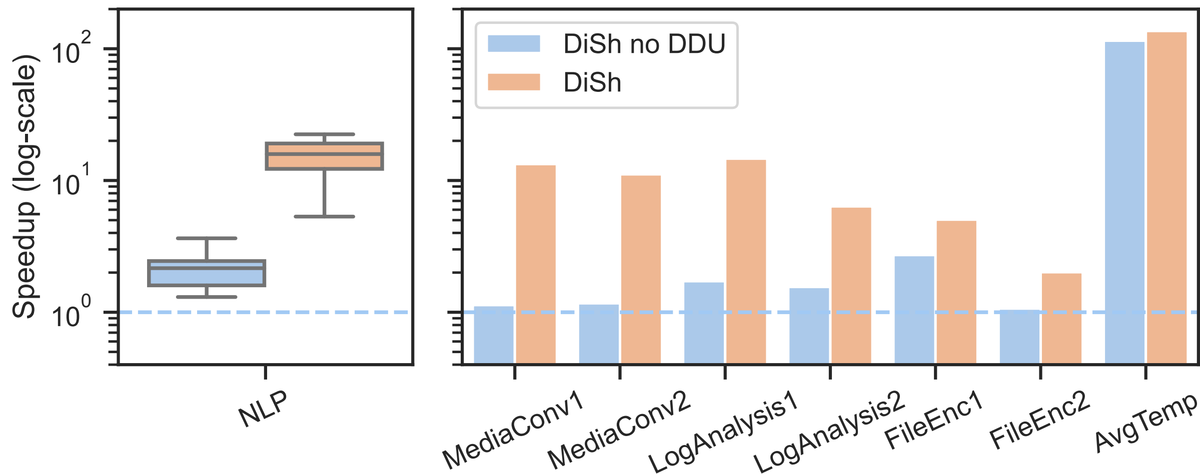


Fig. 8.8: Dynamic dependency untangling. DISH speedup over Bash when toggling DDU (higher is better).

sulted in multiple error-and-fix cycles, and led to script size increases. To overcome AHS's expressiveness limitations, we had to modify a few scripts in unintuitive ways—often combining plain Bash scripts with AHS mappers and reducers. These modifications made scripts significantly more complex and compounded the effort to test and maintain them. Instead, DISH distributed scripts successfully without any such challenges.

8.7.2. Dynamic Dependency Untangling

What is the speedup due to dynamic dependency untangling?

Results: Figure 8.8 shows DISH's speedup over Bash with and without dynamic dependency untangling (DDU, § 8.4.3). It excludes scripts that contain a single dataflow, for which DDU is not applicable. DISH's average speedup over DISH-w/o-DDU is 6.9×, ranging between 1.2–13.9×.

Discussion: Enabling DDU improves performance significantly across all relevant scripts, by running independent dataflow regions in parallel. This allows DISH to expose parallelism not just within data pipelines but across them, improving utilization. DDU also improves the distributed execution of scripts that operate on many files, many or most of which are small enough to fit on a single HDFS block.

DDU is the main reason why DISH gets an edge over Bash on scripts that (1) have implicit independences that are not highly parallelizable, and (2) operate on small data that incur imperceptible data-movement costs.

Examples of such scripts include MediaConv1 and FileEnc2.

8.7.3. Correctness

What is DISH's output compatibility with respect to Bash?

Results: To check the correctness of DISH across all benchmarks, we check that its stdout and exit status are equivalent to the ones produced by Bash. Across all benchmarks, totaling over 650 millions lines (18GB) of output, DISH produces the same output and exit status as Bash.

We additionally execute the complete POSIX shell-test suite to evaluate DISH's compatibility with Bash. Out of all relevant tests, DISH diverges from Bash in two cases and only with respect to the exit status it returns: both exit with an error, but Bash returns 1 whereas DISH returns 127, which is outside of the POSIX mandated exit status range between 1–125. The reason is that DISH always invokes the underlying Bash interpreter using the `-c` flag to set the `$0` variable, and Bash (contrary to most other shells, *e.g.*, dash, ksh, mksh, sash, Smoosh, yash, zsh) exits with 127 in particular failing cases when called with `-c`.

Discussion: All benchmarks in Tab. 8.2 were executed with DISH repeatedly. After hundreds of runs over several weeks, we observed dozens of different execution orders. Comparing the output on every run provides significant confidence about the correctness of the resulting distributed execution. The POSIX test suite mostly evaluates the correctness of dynamic orchestration (§8.3), as it does not feature many opportunities for parallelization and features no opportunities for distribution.

8.8. Related Work

DISH is related to a large body of prior work.

Distributed data processing: Several environments assist in the development of distributed software systems: distributed computing frameworks [46, 130, 186, 129, 161] and domain-specific languages [17, 31, 122, 49, 127] simplify the development of distributed systems that fall under certain computational classes such as batch processing, stream processing, *etc.* These systems deal with many of the challenges of distribution, but require developers to (re)write their computations manually in models that differ significantly from UNIX shell programming.

Hadoop Streaming and Dryad Nebula are abstractions that allow using third-party language-agnostic com-

ponents similar to the UNIX shell, atop cluster-computing engines (Hadoop and Dryad, respectively). Both require their users to understand and rewrite their shell scripts using the abstractions provided by each framework. DISH can operate on arbitrary shell scripts automatically, without requiring any manual effort from its users.

Distributed shells and tools: Several packages expose commands for specifying parallelism and distribution on modern UNIXes—*e.g.*, `qsub` [63], SLURM [184], calls to GNU `parallel` [164]. Different from DISH, their effectiveness is predicated upon explicit and careful invocation and is limited to embarrassingly parallel (and short) programs. Often, these commands provide options to support an array of special subcases—a stark contradiction to the celebrated UNIX philosophy. For example, `parallel` contains flags such as `--skip-first-line`, `-trim`, and `--xargs`, that a UNIX user can achieve using `head`, `sed`, and `xargs`; it also includes other programs with complex semantics, such as the ability transfer files between computers, separate text files, and parse CSV. DISH embraces the UNIX philosophy, attempting to rewrite shell programs to leverage distributed infrastructure.

Several shells [50, 117, 159] add primitives for non-linear pipe topologies—some of which target distribution. Here too, however, developers are expected to manually rewrite scripts to exploit these new primitives.

POSH [144] is a recent shell for scripts operating on NFS-stored data. It brings pipeline components closer to the data on which they operate, but operates only on shell pipelines that are fully expanded—*i.e.*, ones that do not use dynamic features. DISH operates on shell scripts that use (1) any POSIX composition primitive, and (2) the full set of dynamic features present in the UNIX shell.

Distributed operating systems: There is a long history of networked and distributed operating systems [146, 179, 136, 128, 140, 151, 48, 24, 153]. These systems offer abstractions that (1) are similar, but not identical, to the ones offered by UNIX, (2) operate at a lower level of abstraction (*e.g.*, that of system calls, rather than shell primitives), and (3) often aim at simply hiding the network rather than offering scalability benefits. Instead of implementing full-fledged distributed operating system, DISH shows that a thin but sophisticated rewriting-based shim can operate on completely unmodified programs, avoid requiring any user input, and achieve significant speedups by executing fragments in parallel across nodes.

Cloud build systems: Several cloud build systems [4, 80, 53, 5] distribute and parallelize the execution of large builds by constructing dependency graphs using dependency information explicitely specified by their users. Contrary to these systems, DISH operates on general shell programs without exploiting domain-specific information—*e.g.*, build dependencies—and by taking a just-in-time approach that resolves dependencies during the execution of the script.

Correct distribution of dataflow graphs: The DFG is a prevalent model in several areas of data processing, including batch- and stream-processing. Systems implementing DFGs often perform optimizations that are correct given subtle assumptions on the dataflow nodes that do not always hold, introducing erroneous behaviors. Recent work [81, 154, 114, 96] attempts to address this issue by performing optimizations only in cases where correctness is preserved, or by testing that applied optimizations preserve the original behavior. DISH uses its dynamic orchestration to achieve compatibility with the underlying shell and then achieves correct distribution on a per-region level by building on prior work on provably correct transformations for order-aware dataflow graphs (see Chapter 4).

8.9. Discussion

Programmability: An important consideration with any automated system is how it affects programmability, and specifically the ability to debug a misbehaving program or to test a program for correctness. DISH does not negatively affect the developer experience compared to a shell: a developer can use a combination of the many existing tools and commands—*e.g.*, `head` and `grep`—as they would normally do to inspect their script's output and determine what is wrong. When it comes to shell scripts intended for distributed environments, DISH in fact improves developer experience: a developer may use the same set of commands for local or distributed interactions—*e.g.*, to inspect and project parts of a file, regardless of whether that file is stored in HDFS or the local system. Furthermore, developers using DISH can reap the scalability benefits of distribution in analyzing or testing scripts by automatically scaling out load to multiple computers.

Fault tolerance: DISH does not tolerate failures such as worker aborts or network partitions (§8.1). In such cases users are expected to rerun their scripts as shell users normally do in the non-distributed case. Achieving fault tolerance in the context of general shell scripts is in fact particularly challenging due to the prevalence of black-box components that may perform arbitrary side-effects. A fault-tolerant version of DISH should be

able to track all these side-effects and re-execute them appropriately when a script fails. This is in contrast to constrained cluster computing frameworks such as MapReduce and Spark that have precise information about the inputs and outputs of purely functional program components enabling simplified re-execution of dependency graphs (lineage) in the presence of failures. DISH's design however combined with incremental script execution [45] creates an opportunity for addressing this challenge with a hybrid approach: employ conventional fault tolerance approaches for script fragments with annotation information, and instrument the rest of the script to capture and replay its side-effects appropriately in cases of failure.

CHAPTER 9

Out-of-order speculative execution for the shell

Material from this chapter was previously published as "Georgios Liargkovas, Konstantinos Kallas, Michael Greenberg, and Nikos Vasilakis. Executing Shell Scripts in the Wrong Order, Correctly. In Proceedings of the 19th Workshop on Hot Topics in Operating Systems, HOTOS '23, page 103–109, New York, NY, USA, 2023. Association for Computing Machinery." [110]. Georgios Liargkovas, an undergraduate student at the time, was the first author of this paper and the lead developer of the `hs` system prototype; all of the authors contributed to coming up with the idea of speculative execution for shell scripts, and I worked closely with Georgios on the technical development of the system prototype and its evaluation.

9.1. Introduction

This chapter focuses on enabling a powerful optimization for shell programs: *out-of-order program execution* [16]. 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. We identify dynamic interposition, tracing, and containment as key ingredients for this kind of optimization support. Tracing and containment yield another advantage over prior work described in this dissertation: we can appropriately trace, contain, and selectively merge a command's effects without any foreknowledge of its semantics—that is, without need for command specifications!

To explain the potential of out-of-order execution for shell scripts, we ground the discussion with a concrete instance of a shell script that suffers from a common disorder: overly sequential execution.

A patient script: Consider the core of a real bioinformatics script for mapping genomic sequence data to a reference genome (Fig. 9.1), a typical task in, *e.g.*, cancer genomics [124]. 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

```
1  SAMPLES="100 101 102 103"
2  REF="hg19.fa"
3  GROUPS="1 2"
4  # (a) Index
5  bwa index "$REF"
6  for sm in $SAMPLES
7  do
8    # (b) Align sample
9    for gr in $GROUPS
10   do
11     bwa aln "$REF" "$sm.$gr.fastq" > "$sm.$gr.sai"
12   done
13   # (c) Combine sample pairs
14   bwa sampe "$sm.1.sai" "$sm.2.sai" |
15     samtools view -Shu - > "$sm.bam"
16   # (d) Remove polymerase chain reaction-induced dups
17   samtools rmdup "$sm.bam" "$sm.nodup.bam"
18   # (e) Plot coverage histogram
19   samtools mpileup "$sm.nodup.bam" |
20     cut -f4 | python plot.py "$sm.coverage.pdf"
21   # Delete temporary files
22   rm -f "$sm.1.sai" "$sm.2.sai"
23 done
```

Fig. 9.1: A bioinformatics script slightly adapted from Köster and Rahmann [101] that maps sequence reads to a reference genome.

genome and 3.3GB input samples takes about 30 minutes on a 3GHz 16-core machine on Cloudlab [51]. 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 whose execution depends on each other.

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:

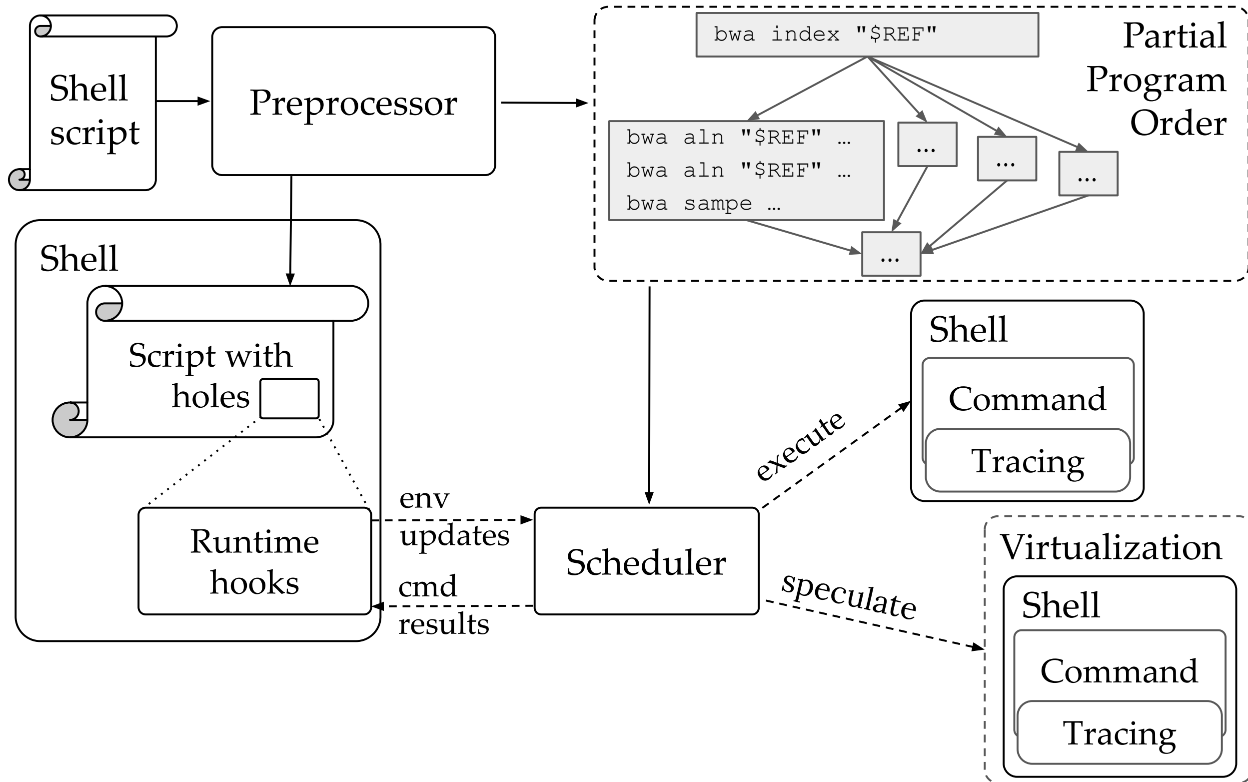


Fig. 9.2: A high-level overview of `hs`, a speculative out-of-order shell-script executor. The preprocessor and runtime hooks extend PASH-JIT (Chapter 7) and tracing extends Riker [45].

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, while intermediate values can usually be inferred in object code, it is impossible to infer the state of the file system at different points in the execution of a shell script. 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 spec-

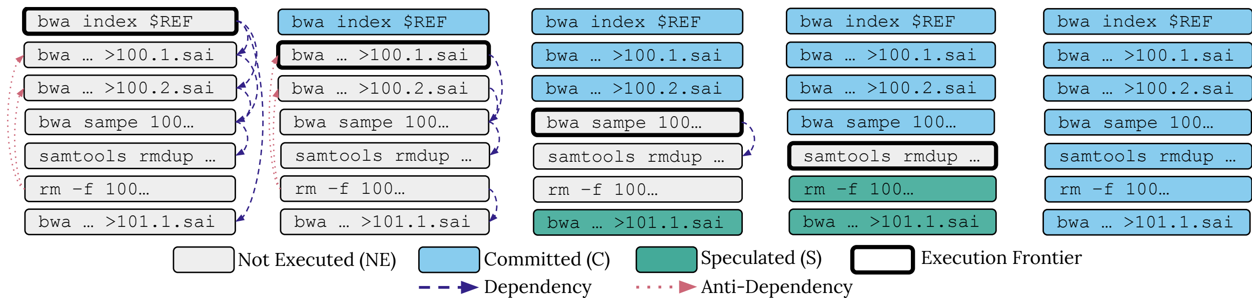


Fig. 9.3: Step-by-step scheduling and orchestration of Köster and Rahmann's [101] script, simplified (Fig. 9.1).

ify or even understand command behavior: our techniques optimistically execute commands in an isolated environment—identifying and rolling back conflicting side-effects as they arise.[12]

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. 9.2): 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 [51], 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).

[12] 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.

We now apply `hs` on the bioinformatics script (Fig. 9.1), sketching its design as we go (Fig. 9.2). `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 [95]. 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. 9.1) would all be replaced with holes, with each hole corresponding to a command in the execution set (Fig. 9.3). 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. 9.1'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. 9.4). 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 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. 9.3, 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.

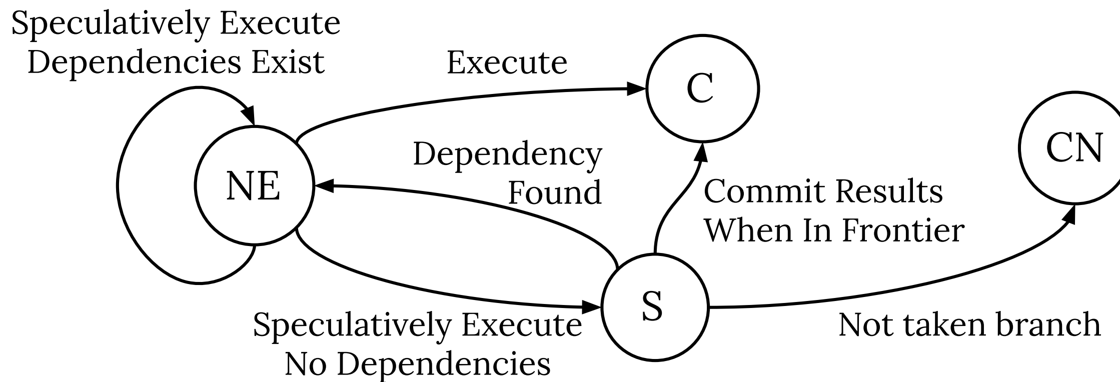


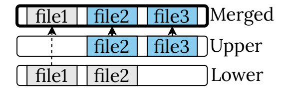
Fig. 9.4: Transition system for command state in the scheduling algorithm.

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 [45] 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 [1] and OverlayFS [36]: we can execute 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. 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.*, `send`). 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 not speculated but rather executed normally, *i.e.*, with minimal tracing and without virtualization. Even if all speculation of next commands fails, the frontier command will always execute normally and therefore 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. 9.1), artificially introducing failures into all speculation yields a 38 minutes execution time (26% slowdown).

Applicability: The techniques used in `hs` are not limited to data processing scripts (Fig. 9.1); 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.

9.3. 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 a 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 buggy (but not malicious) third-party scripts before committing their results. Today's containerization systems, like Docker [123], set up a *different* environment, making it hard to merge

changes to the underlying system—but `try` virtualizes the *existing* system.[13]

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 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

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

possibility is *fusion* [43], 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 single-binary suites of software utilities, 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 [178, 58].

9.4. Related Work

Automated parallelization for the shell: Recent work on shell-script parallelization and distribution [173, 95, 131, 144] (some of which is described in the previous chapters of this dissertation) has delivered significant performance benefits by exploiting lightweight command specifications. The approach proposed by `hs`, however, does not require *any* command specifications—we infer the necessary command-execution information at runtime.

Explicit dependency encoding: Workflow and build systems [160, 101, 4, 59] 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 [162] or security checks [134]. In some of these proposals speculation is enabled by modifying the application (*e.g.*, Undo [35]), while others support arbitrary black-box applications [162, 133, 134]. 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 [170, 54]. 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.

CHAPTER 10

Conclusion

This dissertation describes the development of the first systems to automatically optimize shell scrips. In addition to the systems, this dissertation also introduces three key components that underlie the systems and enable their development: (1) a command specification framework that allows reasoning about scripts with black-box commands, (2) an order-aware dataflow model that is the foundation of a shell-to-shell compiler, and (3) a just-in-time architecture that enables optimizing scripts with arbitrarily dynamic behavior. Two key characteristics of the work described in this dissertation is that it does not require any modifications to the user scripts or the underlying interpreter, and that it is accompanied by strong practical and formal correctness guarantees, *i.e.*, that the optimized scripts have the same beahvior as the originals. Finally, all of the work described in this dissertation is open-source, hosted under the Linux Foundation as the PaSh project, and can be found at https://github.com/binpash.

10.1. Future Work

A lot of the work described in this dissertation can act as the foundation for additional systems, tooling, and support for the shell. Two interesting directions for future work include: supporting the deployment of shell scripts to cloud resources for additional scalability and for processing cloud-native data; and developing tooling that checks that a shell script behaves as expected before it is executed. The second direction is particularly important both to protect from benign bugs, but also from malicious ones, since shell command injections are ranked as the 5th most dangerous type of weakness in 2023 according to the CWE database[14]. The contributions described in this dissertation can kickstart the development of such tools, *e.g.*, a cloud-native shell can use the just-in-time architecture as its foundation to be able to support arbitrary scripts with correctness guarantees.

10.2. Outlook

We often tend to think that higher level and more declarative languages are easier targets for optimizations and for building systems to achieve performance benefits. One bigger conclusion of this dissertation is that

[14] https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html.

even if a language has characteristics that are hostile to optimizations, *e.g.*, high dynamism and black-box subcomponents, it still has a beautiful and elegant core that cleanly describes relevant workloads and is amenable to optimizations. The work described in this dissertation identifies one such fragment for the shell, one that describes dataflow computations, that captures the bulk of the computation that happens in the shell. By clearly delineating this fragment and optimizing it, while carefully maintaining behavioral equivalence for the rest of the language, this work manages to automatically get significant performance benefits for arbitrary shell scripts.

Furthermore, the scope of the work described in this dissertation is not limited to the language of the shell. There exist several other component composition workloads, such as scripts written Python and Perl, or applications written using workflow languages, that have similar characteristics and constraints, where my work could be applicable. The main common characteristic of these workloads is that they are all bimodal; the core language, *e.g.*, shell, Python, *etc.*, is used to compose or glue together several components that can be written in arbitrary languages and can thus be considered black-boxes. Additionally, most of the computation happens using the black-box components so there is significant benefit to be acquired by optimizing them instead of the core language. Finally, for these workloads it is very hard to reimplement the language interpreter from scratch, because that would require significant effort and jeopardize compliance with the legacy behavior. The three main building blocks introduced in this dissertation, *i.e.*, the specification framework, the just-in-time architecture, and the order-aware dataflow model, can be lifted to arbitrary workloads for which these constraints hold, enabling performance and other types of benefits there too.

BIBLIOGRAPHY

[1] namespaces(7) – linux manual page. https://man7.org/linux/man-pages/man7/namespaces.7.html.

[2] gRPC. https://grpc.io/, 2018. Accessed: 2019-04-16.

[3] Dash (Debian Almquist shell). https://git.kernel.org/pub/scm/utils/dash/dash.git/, 2021.

[4] Bazel Dynamic Execution. https://bazel.build/remote/dynamic, 2022. [Online; accessed Feb 1, 2022].

[5] Google Cloud Build. https://cloud.google.com/build/docs/overview, 2022. [Online; accessed Feb 1, 2022].

[6] ksh (korn shell). http://www.kornshell.com/, 2023.

[7] Kubernetes. https://kubernetes.io/, 2023.

[8] mksh (MirBSD Korn Shell). http://www.mirbsd.org/mksh.htm, 2023.

[9] Oil Shell. https://www.oilshell.org/, 2023.

[10] Systemd: System and Service Manager. https://systemd.io/, 2023.

[11] The Ninja Build System. https://ninja-build.org/, 2023.

[12] Yash (Yet Another Shell). https://magicant.github.io/yash/, 2023.

[13] zsh (Z shell). https://www.zsh.org/, 2023.

[14] eBPF (extended Berkeley Packet Filter). https://ebpf.io/, 2024.

[15] PaSh Project. https://github.com/binpash, 2024.

[16] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers: Principles, Techniques, and Tools*, volume 2. Addison-Wesley Reading, 2007.

[17] Peter Alvaro, Neil Conway, Joseph M Hellerstein, and William R Marczak. Consistency Analysis in Bloom: a CALM and Collected Approach. In *CIDR*, pages 249–260, 2011.

[18] Sotiris Apostolakis, Ziyang Xu, Greg Chan, Simone Campanoni, and David I August. Perspective: A Sensible Approach to Speculative Automatic Parallelization. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 351–367, 2020.

[19] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal*, 15(2):121–142, 2006.

[20] K. Arvind, E. David Culler, Robert Iannucci, Vinod Kathail, Keshav Pingali, and Robert Thomas. The Tagged Token Dataflow Architecture. Technical report, MIT Laboratory for Computer Science, 1984.

[21] K. Arvind and Rishiyur S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Trans. Comput.*, 39(3):300–318, March 1990.

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

[23] John Aycock. A Brief History of Just-In-Time. *ACM Computing Surveys (CSUR)*, 35(2):97–113, 2003.

[24] Amnon Barak and Oren La'adan. The MOSIX Multicomputer Operating System for High Performance Cluster Computing. *Future Generation Computer Systems*, 13(4):361–372, 1998.

[25] Jonathan C Beard, Peng Li, and Roger D Chamberlain. RaftLib: a C++ Template Library for High Performance Stream Parallel Processing. *The International Journal of High Performance Computing Applications*, 31(5):391–404, 2017.

[26] Benedikt Becker, Nicolas Jeannerod, Claude Marché, Yann Régis-Gianas, Mihaela Sighireanu, and Ralf Treinen. Analysing installation scenarios of debian packages. In *Tools and Algorithms for the Construction and Analysis of Systems: 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25–30, 2020, Proceedings, Part II*, pages 235–253. Springer, 2020.

[27] Jon Bentley. Programming Pearls: A Spelling Checker. *Commun. ACM*, 28(5):456–462, May 1985.

[28] Jon Bentley, Don Knuth, and Doug McIlroy. Programming Pearls: A Literate Program. *Commun. ACM*, 29(6):471–483, June 1986.

[29] Gérard Berry and Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, November 1992.

[30] Pawan Bhandari. Solutions to unixgame.io. https://git.io/Jf2dn, 2020. Accessed: 2020-04-14.

[31] Martin Biely, Pamela Delgado, Zarko Milosevic, and Andre Schiper. Distal: A Framework for Implementing Fault-tolerant Distributed Algorithms. In *Proceedings of the 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, DSN '13, pages 1–8, Washington, DC, USA, 2013. IEEE Computer Society.

[32] Carl Friedrich Bolz. *Meta-tracing just-in-time compilation for RPython*. PhD thesis, Universitäts-und

Landesbibliothek der Heinrich-Heine-Universität Düsseldorf, 2014.

[33] Timothy Bourke and Marc Pouzet. Zélus: A synchronous language with odes. In *Proceedings of the 16th International Conference on Hybrid Systems: Computation and Control*, HSCC '13, pages 113–118, New York, NY, USA, 2013. Association for Computing Machinery.

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

[35] Aaron B Brown and David A Patterson. Undo for operators: Building an undoable e-mail store. In *USENIX Annual Technical Conference, General Track*, pages 1–14, 2003.

[36] Neil Brown, Miklos Szeredi, Amir Goldstein, Vivek Goyal, Randy Dunlap, Linus Torvalds, Pavel Tikhomirov, Kevin Locke, Sargun Dhillon, Chengguang Xu, and Deming Wang. The overlay filesystem. *The Linux Kernel documentation*, 2022. Started in 2014.

[37] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. *ACM Trans. Graph.*, 23(3):777–786, August 2004.

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

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

[40] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.*, 38:28–38, 2015.

[41] Armando Cerna. Pacaur building Script. https://github.com/armandocerna/dotfiles/blob/master/scripts/pacaur.sh.

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

[43] Wei-Ngan Chin. Safe fusion of functional expressions ii: Further improvements. *Journal of Functional Programming*, 4(4):515–555, 1994.

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

[45] 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.

[46] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[47] Jack B. Dennis. First version of a data flow procedure language. In B. Robinet, editor, *Programming Symposium*, pages 362–376, Berlin, Heidelberg, 1974. Springer Berlin Heidelberg.

[48] Sean Dorward, Rob Pike, David L Presotto, Dennis Ritchie, Howard Trickey, and Phil Winterbottom. Inferno. In *Proceedings IEEE COMPCON 97. Digest of Papers*, pages 241–244. IEEE, 1997.

[49] Cezara Drăgoi, Thomas A Henzinger, and Damien Zufferey. PSync: a partially synchronous language for fault-tolerant distributed algorithms. *ACM SIGPLAN Notices*, 51(1):400–415, 2016.

[50] Tom Duff. Rc—A shell for Plan 9 and Unix systems. *AUUGN*, 12(1):75, 1990.

[51] 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.

[52] Jeff Epstein, Andrew P. Black, and Simon Peyton-Jones. Towards Haskell in the Cloud. In *Proceedings of the 4th ACM Symposium on Haskell*, Haskell '11, pages 118–129, New York, NY, USA, 2011. ACM.

[53] Hamed Esfahani, Jonas Fietz, Qi Ke, Alexei Kolomiets, Erica Lan, Erik Mavrinac, Wolfram Schulte, Newton Sanches, and Srikanth Kandula. CloudBuild: Microsoft's Distributed and Caching Build Service. In *SEIP*. IEEE - Institute of Electrical and Electronics Engineers, June 2016.

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

[55] Facebook. Buck: A high-performance build tool. https://buck.build/, 2023.

[56] Azadeh Farzan and Victor Nicolet. Synthesis of divide and conquer parallelism for loops. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 540–555, New York, NY, USA, 2017. Association for Computing Machinery.

[57] Azadeh Farzan and Victor Nicolet. Modular divide-and-conquer parallelization of nested loops. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 610–624, New York, NY, USA, 2019. Association for Computing Machinery.

[58] Andrew Ferguson and Philip Wadler. When will deforestation stop. In *Glasgow Workshop on Func-*

tional Programming, 1988.

[59] 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.

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

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

[62] Jim Garlick. pdsh. https://github.com/chaos/pdsh, 2022. [Online; accessed September 15, 2022].

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

[64] Inc. GitHub. The 2022 State of the Octoverse: Top languages over the years. https://octoverse.github.com/2022/top-programming-languages, 2022. [Online; accessed July 3, 2023].

[65] Google. V8 JavaScript Engine. https://developers.google.com/v8/.

[66] Google. Protocol Buffers. https://developers.google.com/protocol-buffers, 2022. Accessed: 2022-06-01.

[67] Michael I Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. *ACM SIGPLAN Notices*, 41(11):151–162, 2006.

[68] Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. A Stream Compiler for Communication-Exposed Architectures. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, page 291–303, New York, NY, USA, 2002. Association for Computing Machinery.

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

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

[71] 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.

[72] Michael Greenberg, Konstantinos Kallas, and Nikos Vasilakis. Unix Shell Programming: The next

50 Years. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 104–111, New York, NY, USA, 2021. Association for Computing Machinery.

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

[74] Hadoop. Hadoop Streaming. https://hadoop.apache.org/docs/r1.2.1/streaming.html, 2022. [Online; accessed September 15, 2022].

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

[76] Shivam Handa, Konstantinos Kallas, Nikos Vasilakis, and Martin C. Rinard. An Order-Aware Dataflow Model for Parallel Unix Pipelines. *Proc. ACM Program. Lang.*, 5(ICFP), August 2021.

[77] Mitchell Hashimoto. *Vagrant: up and running: create and manage virtualized development environments*. " O'Reilly Media, Inc.", 2013.

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

[79] Allan Heydon, Roy Levin, Timothy Mann, and Yuan Yu. *The Vesta approach to software configuration management*. Compaq. Systems Research Center [SRC], 2001.

[80] Jason Hickey and Aleksey Nogin. OMake: Designing a Scalable Build Process. In Luciano Baresi and Reiko Heckel, editors, *Fundamental Approaches to Software Engineering*, pages 63–78, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[81] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. A Catalog of Stream Processing Optimizations. *ACM Computing Surveys (CSUR)*, 46(4):46:1–46:34, March 2014.

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

[83] Berwyn Hoyt, Bryan Hoyt, and Ben Hoyt. Fabricate: The better build tool. https://github.com/SimonAlfie/fabricate, 2009.

[84] Lluis Batlle i Rossell. *tsp(1) Linux User's Manual*. https://vicerveza.homeunix.net/ viric/soft/ts/, 2016.

[85] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72, 2007.

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

[87] Nicolas Jeannerod, Claude Marché, Yann Régis-Gianas, Mihaela Sighireanu, and Ralf Treinen. *Specification of UNIX utilities*. PhD thesis, ANR, 2019.

[88] Nicolas Jeannerod, Claude Marché, and Ralf Treinen. A formally verified interpreter for a shell-like programming language. In *Verified Software. Theories, Tools, and Experiments: 9th International Conference, VSTTE 2017, Heidelberg, Germany, July 22-23, 2017, Revised Selected Papers 9*, pages 1–18. Springer, 2017.

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

[90] Wesley M Johnston, JR Paul Hanna, and Richard J Millar. Advances in dataflow programming languages. *ACM computing surveys (CSUR)*, 36(1):1–34, 2004.

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

[92] Dan Jurafsky. Unix for Poets. https://web.stanford.edu/class/cs124/lec/124-2018-UnixForPoets.pdf, 2017.

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

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

[95] 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 769–785, Carlsbad, CA, July 2022. USENIX Association.

[96] Konstantinos Kallas, Filip Niksic, Caleb Stanford, and Rajeev Alur. DiffStream: Differential Output Testing for Stream Processing Programs. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–29, 2020.

[97] Konstantinos Kallas and Konstantinos Sagonas. HiPErJiT: A Profile-Driven Just-in-Time Compiler for Erlang. In *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages*, pages 25–36, 2018.

[98] Richard M. Karp and Raymond E. Miller. Properties of a model for parallel computations: Determi-

nacy, termination, queueing. *SIAM Journal on Applied Mathematics*, 14(6):1390–1411, 1966.

[99] Charles Edwin Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin M. Vahdat. Mace: Language Support for Building Distributed Systems. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 179–188, New York, NY, USA, 2007. ACM.

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

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

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

[103] Nokia Bell Labs. The Unix Game—Solve puzzles using Unix pipes. https://unixgame.io/unix50, 2019. Accessed: 2020-03-05.

[104] Paul Le Guernic, Albert Benveniste, Patricia Bournai, and Thierry Gautier. Signal–a data flow-oriented language for signal processing. *IEEE transactions on acoustics, speech, and signal processing*, 34(2):362–374, 1986.

[105] Edward A Lee and David G Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[106] Edward A Lee and Thomas M Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, 1995.

[107] Edward Ashford Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, 36(1):24–35, January 1987.

[108] Haoyuan Li. *Alluxio: A virtual distributed file system*. University of California, Berkeley, 2018.

[109] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 311–322, New York, NY, USA, 2005. Association for Computing Machinery.

[110] Georgios Liargkovas, Konstantinos Kallas, Michael Greenberg, and Nikos Vasilakis. Executing Shell Scripts in the Wrong Order, Correctly. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, HOTOS '23, page 103–109, New York, NY, USA, 2023. Association for Computing Machinery.

[111] Amy W. Lim and Monica S. Lam. Maximizing Parallelism and Minimizing Synchronization with Affine Transforms. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 201–214, New York, NY, USA, 1997. ACM.

[112] Konstantinos Mamouras. Semantic foundations for deterministic dataflow and stream processing. In *European Symposium on Programming*, pages 394–427. Springer, Cham, 2020.

[113] Konstantinos Mamouras, Mukund Raghothaman, Rajeev Alur, Zachary G. Ives, and Sanjeev Khanna. StreamQRE: Modular Specification and Efficient Evaluation of Quantitative Queries over Streaming Data. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '17, pages 693–708, New York, NY, USA, 2017. ACM.

[114] Konstantinos Mamouras, Caleb Stanford, Rajeev Alur, Zachary G. Ives, and Val Tannen. Data-Trace Types for Distributed Stream Processing Systems. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 670–685, New York, NY, USA, 2019. ACM.

[115] Florence Maraninchi and Yann RéMond. Argos: An automaton-based synchronous language. *Comput. Lang.*, 27(1–3):61–92, April 2001.

[116] Bill McCloskey. Memoize. https://github.com/kgaughan/memoize.py, 2008.

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

[118] Malcolm D McIlroy, Elliot N Pinson, and Berkley A Tague. UNIX Time-Sharing System: Foreword. *Bell System Technical Journal*, 57(6):1899–1904, 1978.

[119] Peter M McIlroy, Keith Bostic, and M Douglas McIlroy. Engineering radix sort. *Computing systems*, 6(1):5–27, 1993.

[120] Frank McSherry, Michael Isard, and Derek G Murray. Scalability! But at what COST? 15:241–299, 2015.

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

[122] Christopher Meiklejohn and Peter Van Roy. Lasp: a language for distributed, eventually consistent computations with CRDTs. In *Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data*, page 7. ACM, 2015.

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

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

[125] Jürgen Cito Michael Schröder. An Empirical Investigation of Command-Line Customization. *arXiv preprint arXiv:2012.10206*, 2020.

[126] Neil Mitchell. Shake before building: Replacing make with haskell. *ACM SIGPLAN Notices*, 47(9):55–66, 2012.

[127] Adrian Mizzi, Joshua Ellul, and Gordon Pace. D'artagnan: An embedded dsl framework for distributed embedded systems. In *Proceedings of the Real World Domain Specific Languages Workshop 2018*, pages 1–9, 2018.

[128] 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.

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

[130] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. CIEL: A Universal Execution Engine for Distributed Data-flow Computing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 113–126, Berkeley, CA, USA, 2011. USENIX Association.

[131] Tammam Mustafa, Konstantinos Kallas, Pratyush Das, and Nikos Vasilakis. DiSh: Dynamic Shell-Script distribution. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 341–356, Boston, MA, April 2023. USENIX Association.

[132] National Oceanic and Atmospheric Administration. National Climatic Data Center. https://www.ncdc.noaa.gov/, 2017.

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

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

[135] Guilherme Ottoni. HHVM JIT: A Profile-guided, Region-based Compiler for PHP and Hack. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 151–165, 2018.

[136] 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.

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

[138] Shoumik Palkar and Matei Zaharia. Optimizing Data-intensive Computations in Existing Libraries with Split Annotations. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 291–305, New York, NY, USA, 2019. ACM.

[139] Davide Pasetto and Albert Akhriev. A comparative study of parallel sort algorithms. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 203–204, 2011.

[140] 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.

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

[142] Pixelbeat. Answer to: Sort –parallel isn't parallelizing. https://superuser.com/a/938634, 2015. Accessed: 2020-04-14.

[143] Jon Puritz. BIO594: Using genomic techniques to examine the evolution of populations. https://git.io/JY6J7, 2019. Accessed: 2020-10-05.

[144] 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.

[145] Chet Ramey. Bash reference manual. *Network Theory Limited*, 15, 1998.

[146] Richard F Rashid and George G Robertson. *Accent: A communication oriented network operating system kernel*, volume 15. ACM, 1981.

[147] Christophe Ratel, Nicolas Halbwachs, and Pascal Raymond. Programming and verifying critical systems by means of the synchronous data-flow language lustre. pages 112–119, 1991.

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

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

[150] Dennis M. Ritchie and Ken Thompson. The UNIX Time-sharing System. *SIGOPS Oper. Syst. Rev.*, 7(4):27–, January 1973.

[151] Marc Rozier, Vadim Abrossimov, François Armand, Ivan Boule, Michel Gien, Marc Guillemont, Frédéric Herrmann, Claude Kaiser, Sylvain Langlois, Pierre Léonard, et al. Overview of the Chorus distributed operating system. In *Workshop on Micro-Kernels and Other Kernel Architectures*, pages 39–70. Seattle WA (USA), 1992.

[152] Radu Rugina and Martin Rinard. Automatic parallelization of divide and conquer algorithms. In *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '99, pages 72–83, New York, NY, USA, 1999. Association for Computing Machinery.

[153] Jan Sacha, Jeff Napper, Sape Mullender, and Jim McKie. Osprey: Operating system for predictable clouds. In *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN 2012)*, pages 1–6. IEEE, 2012.

[154] Scott Schneider, Martin Hirzel, Buğra Gedik, and Kun-Lung Wu. Safe Data Parallelism for General Streaming. *IEEE Transactions on Computers*, 64(2):504–517, Feb 2015.

[155] Peter Sewell, James J. Leifer, Keith Wansbrough, Francesco Zappa Nardelli, Mair Allen-Williams, Pierre Habouzit, and Viktor Vafeiadis. Acute: High-level Programming Language Design for Distributed Computation. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, ICFP '05, pages 15–26, New York, NY, USA, 2005. ACM.

[156] Jiasi Shen, Martin Rinard, and Nikos Vasilakis. Automatic Synthesis of Parallel Unix Commands and Pipelines with KumQuat. CoRR abs/2012.15443 (2021). *arXiv preprint arXiv:2012.15443*, 2021.

[157] Calvin Smith and Aws Albarghouthi. Mapreduce program synthesis. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 326–340, New York, NY, USA, 2016. Association for Computing Machinery.

[158] Sarah Spall, Neil Mitchell, and Sam Tobin-Hochstadt. Build scripts with perfect dependencies. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020.

[159] Diomidis Spinellis and Marios Fragkoulis. Extending Unix Pipelines to DAGs. *IEEE Transactions on Computers*, 66(9):1547–1561, 2017.

[160] Richard M Stallman and Roland McGrath. GNU Make—A Program for Directing Recompilation. https://www.gnu.org/software/make/manual/make.pdf, 1991.

[161] Craig A Stewart, Timothy M Cockerill, Ian Foster, David Hancock, Nirav Merchant, Edwin Skidmore, Daniel Stanzione, James Taylor, Steven Tuecke, George Turner, et al. Jetstream: a self-provisioned, scalable science and engineering cloud environment. In *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*, pages 1–8, 2015.

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

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

[164] Ole Tange. GNU Parallel—The Command-Line Power Tool. *;login: The USENIX Magazine*, 36(1):42–47, Feb 2011.

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

[166] Elixir Core Team. Elixir. https://elixir-lang.org/, 2023.

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

[168] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. Streamit: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 179–196, Berlin, Heidelberg, 2002. Springer-Verlag.

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

[170] Josep Torrellas. Speculation, thread-level. In David Padua, editor, *Encyclopedia of Parallel Computing*, pages 1894–1900. Springer US, Boston, MA, 2011.

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

[172] Junichi Uekawa. dsh. https://www.netfort.gr.jp/~dancer/software/dsh.html.en, 2022. [Online; accessed September 15, 2022].

[173] Nikos Vasilakis, Konstantinos Kallas, Konstantinos Mamouras, Achilles Benetopoulos, and Lazar Cvetković. PaSh: Light-Touch Data-Parallel Shell Processing. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 49–66, New York, NY, USA, 2021. Association for Computing Machinery.

[174] Nikos Vasilakis, Ben Karel, Yash Palkhiwala, John Sonchack, André DeHon, and Jonathan M. Smith. Ignis: Scaling Distribution-oblivious Systems with Light-touch Distribution. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 1010–1026, New York, NY, USA, 2019. ACM.

[175] Nikos Vasilakis, Ben Karel, and Jonathan M. Smith. From Lone Dwarfs to Giant Superclusters: Re-

thinking 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.

[176] Nikos Vasilakis, Cristian-Alexandru Staicu, Grigoris Ntousakis, Konstantinos Kallas, Ben Karel, André DeHon, and Michael Pradel. Preventing Dynamic Library Compromise on Node.js via RWX-Based Privilege Reduction. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 1821–1838, New York, NY, USA, 2021. Association for Computing Machinery.

[177] Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in ERLANG (2Nd Ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1996.

[178] Philip Wadler. Deforestation: Transforming programs to eliminate trees. In H. Ganzinger, editor, *ESOP '88*, pages 344–358, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg.

[179] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The LOCUS distributed operating system. *ACM SIGOPS Operating Systems Review*, 17(5):49–70, 1983.

[180] Edward Walker, Weijia Xu, and Vinoth Chandar. Composing and executing parallel data-flow graphs with shell pipes. In *Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science*, WORKS '09, New York, NY, USA, 2009. Association for Computing Machinery.

[181] Ian Watson and John Gurd. A prototype data flow computer with token labelling. In *1979 International Workshop on Managing Requirements Knowledge (MARK)*, pages 623–628. IEEE, 1979.

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

[183] Keith Winstein and Hari Balakrishnan. Mosh: An interactive remote shell for mobile clients. In *USENIX Annual Technical Conference*, volume 4, 2012.

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

[185] Gina Yuan, Shoumik Palkar, Deepak Narayanan, and Matei Zaharia. Offload annotations: Bringing heterogeneous computing to existing libraries and workloads. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 293–306, 2020.

[186] 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.

[187] Zhao Zhang, Daniel S. Katz, Timothy G. Armstrong, Justin M. Wozniak, and Ian Foster. Parallelizing the execution of sequential scripts. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, New York, NY, USA, 2013. Association for Computing Machinery.