Extending Unix Pipelines to DAGs

Diomidis Spinellis, *Senior Member, IEEE,* and Marios Fragkoulis

Abstract—The Unix shell *dgsh* provides an expressive way to construct sophisticated and efficient non-linear pipelines. Such pipelines can use standard Unix tools, as well as third-party and custom-built components. *Dgsh* allows the specification of pipelines that perform non-uniform non-linear processing. These form a directed acyclic process graph, which is typically executed by multiple processor cores, thus increasing the processing task's throughput. A number of existing Unix tools have been adapted to take advantage of the new shell's multiple pipe input/output capabilities. The shell supports visualization of the process graphs, which can also aid debugging. *Dgsh* was evaluated through a number of common data processing and domain-specific examples, and was found to offer an expressive way to specify processing topologies, while also generally increasing processing throughput.

Index Terms—Process-level parallelism, Unix, pipeline, pipes and filters architecture

1 Introduction

A pipeline is a set of processes that are linearly connected so that the output of one is delivered as input to the next one in the series. While the concept's origin in the form used today was popularized by the Unix operating system, pipelines are nowadays supported by many modern systems. The availability of pipelines has allowed the development of toolkits that build on them and the emergence of the "pipes and filters" architecture. Although pipelines are extremely powerful, the standard Unix shell's restriction of the supported topology to a linear sequence, limits the structure or performance of many useful applications that require a more general way to interconnect processes. This paper describes the design and implementation of a shell that addresses this problem by supporting a syntax and an execution environment for creating arbitrary networks of communicating processes.

On Unix systems and those systems influenced by them, the establishment of process pipelines is supported through the *pipe* system call, which creates a unidirectional serial communication channel that two processes can use to send data from one to the other. The *pipe* call is in turn utilized to support the shell's *pipeline* syntax, which allows the creation of a linearly connected set of processes, where each process receives input from its preceding one and passes its output to the next one. For example, the following shell command

ls | wc –l

will create two processes: *ls*, which will list on its standard output the names of files in the current directory, and *wc*, which will count the number of lines appearing in its standard input. The pipeline symbol | joining the two processes, will result in the creation of a pipe, and the joining of the standard output of the *ls* process with the standard input of the *wc* process. Thus, the command will display the *number* of files in the current directory.

Pipes in the form they are most commonly used today were added to the Unix operating system in 1972 [1] after strong advocacy by M. D. McIlroy, who wrote the following in a 1964 list of four action items [2].

> "1. We should have some ways of coupling programs like garden hose—screw in another segment when it becomes necessary to massage data in another way. This is the way of I/O also."

Initially the *pipe* system call was implemented, then the shell's redirection operators were extended to support the expression of pipelines. A notation for expressing pipelines with a unique operator was devised a few months afterwards. A similar abstraction, called "Communication Files" [3, pp. 175–197], is reported [1] to have pre-existed in the Dartmouth Time Sharing System without the Unix developers knowing it at the time.

Unix pipelines can be modelled as a restricted form of Hoare's communicating sequential processes (CSP) [4], [5]. Under the CSP model, processes communicate by sending or receiving values through named unbuffered channels. Pipes differ from the CSP model in that they offer a (typically small) buffer. In practice, this increases the throughput but also the latency of jobs employing pipelines, making Unix pipelines more suitable for batch-oriented processing, and less fitting for implementing, say, interactive applications.

Pipelines offer many advantages. They provide a natural syntax for setting up processes to work together without onerous prearrangements, realizing what was later termed the *pipes and filters* architecture [6] [7, pp. 21–22]. Two processes linked by a pipe can cooperate simply by handling data in a compatible format; newline-terminated text streams are a particularly popular one. Pipelines promote abstraction, encapsulation, and loose coupling by allowing the design of programs that do a single thing well, and leave additional processing to be performed by other specialized programs.

• *The authors are with the Department of Managment Science and Technology, Athens University of Economics and Business, Athens GR-104 34, Greece. E-mail: {dds, mfg}@aueb.gr.*

Manuscript received 5 Dec. 2016; revised 5 Apr. 2017; accepted 12 Apr. 2017. Date of publication 17 Apr. 2017; date of current version 15 Aug. 2017. Recommended for acceptance by P. Laplante.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2017.2695447

For example, most of the Unix tools lack formatting and pagination facilities, because these are supposed to be handled by other tools, such as *pr*, *fmt*, *troff*, and *more*. In addition, if a pipeline's programs process data on a record by record basis, without requiring the buffering of all their input into memory, the pipeline can process an arbitrary (in theory infinite) amount of data. Finally, pipelines exploit the processing parallelism offered by modern multi-core processors, because the processes that comprise them can often run in parallel.

Unsurprisingly, the pipeline abstraction has been extremely successful. As an indication of their widespread use, consider that the 697 shell scripts appearing in the source code of a modern Unix derivative (FreeBSD version 11.0), contain about 6,400 pairs of pipe-connected processes with more than 40 percent of the scripts containing at least one. In addition, the existence of pipelines has spearheaded the development of widely-used domain-specific program families and toolkits whose components have been designed to be combined through pipelines. Examples include the Unix document preparation tools [8], *netpbm*, used for image processing [9], the Generic Mapping Tools (GMT) [10], NMRPipe [11], used for processing nuclear magnetic resonance data, the Madagascar shared research environment for computational data analysis in geophysics and related fields [12], and the LLVM compiler infrastructure [13].

The linear way in which pipelines are expressed in the popular Unix shells typically restricts pipelines to the expression of sequences where a single producer is linked to a single consumer. This is problematic when the output of an expensive process must be piped for processing by more than one downstream processes, or when one process requires input from more than one upstream process.

As an example for multiple downstream producers consider the task of using the *curl* command to fetch a large archive from the web, calculating its cryptographic checksum for verification with the *md5sum* command, and unpacking it into a local file system with the *tar* file archiver. This can be accomplished by the following two pipelines, which however waste network bandwidth and processing power, by fetching the file twice.

```
curl http://example.com/archive | md5sum
curl http://example.com/archive | tar xzf -
```

Alternatively, the file could be temporarily stored as a whole onto a local file system wasting storage space and bandwidth.

```
curl http://example.com/archive >/tmp/filename
md5sum /tmp/filename
tar xzf /tmp/filename
```

The problem of handling multiple upstream producers is often evident when using Unix programs that expect input from more than one file, such as *paste* which pastes together records, *comm* which finds common records among two files, and *join* which performs a relational join between two files. As an example, the following command sequence uses two temporary files to list the names of files that exist only in one of the two directory hierarchies given as an argument to the *find* command. It works by creating a sorted list of the file names in each directory by means of the *find* and *sort* commands, and then by using the *comm* command to list the records that do not appear in both of its inputs.

```
find /dir1 -printf '%P\n' | sort >/tmp/dir1
find /dir2 -printf '%P\n' | sort >/tmp/dir2
comm -3 /tmp/dir1 /tmp/dir2
```

There are two main workarounds for sidestepping the restriction of pipelines to a linear sequence, but both leave something to be desired. The use of temporary files wastes space and can impact performance. In addition, temporary files require some delicate programming to guarantee their deletion when a shell script is interrupted, and necessitate inventing additional names to describe data that simply flows between processes. Another workaround works for programs following a convention under which a command-line argument of "-" denotes the program's standard input, rather than a file name. This makes it possible to replace one of the temporary files with input from a pipeline. However, this trick only handles the case of a single input.

Some modern Unix shells, such as *bash*, offer a feature named *process substitution* [14, p. 219]. Through the feature's syntax, a program argument of the form <(*producer*) will be substituted by an internally-generated pipe endpoint name that will receive its input from the *producer* process, while an argument of the form >(*consumer*) will be substituted by a similar name that will feed the input it receives to *consumer*. Following this scheme, the file fetching example could be re-written with the use of the *tee* program, which copies its standard input both to its standard output and to its specified arguments, as follows,

```
curl http://example.com/archive |
tee >(md5sum) |
tar xzf -
```

Similarly, the directory comparison example would be written as follows.

```
comm -3 <(find /dir1 ... | sort) <(find /dir2 ... | sort)
```

The main problem with the process substitution method is that piping output from a single producer to multiple consumers (*one-to-many*) cannot be combined with piping output from multiple producers to a single consumer (*many-to-one*). In addition, in the latter case, the processing must be awkwardly specified outside-in from the right to the left, which is the opposite order from the one in which pipelines are conventionally expressed.

Modern Unix systems offer *named pipes*, also known as FIFOs, which can be used to hand-craft arbitrary process communication topologies. However, if combined one-to-many and many-to-one piping are setup by using named pipes, another problem will occur. Due to the limited buffering offered by typical programs, deadlocks can easily occur when a process consuming data from many producers with more than one input, blocks waiting for input from one of the processes feeding it. This can cause a second feeding process to block, waiting to send its output to another one of the consumer process's inputs, and, thereby, blocking the upstream process feeding both processes that provide data to the consumer one.

The directed acyclic graph (DAG) shell (*dgsh*) described in this paper extends the linear pipeline syntax of common Unix shells with syntax and an underlying implementation that allows interprocess data flows to follow a DAG topology.

Specifically, processes form the DAG's nodes and pipeline connections form the DAG's edges, with data flowing in the direction of the graph's edges. As an example, the file fetching case would be expressed in a *dgsh* script presented in Listing 1. There, the fetched file is fed into the *tee* command, which sends one copy of its input to *md5sum* and another to *tar*.

Listing 1. Fetch file once; compute checksum and unpack in parallel

```
curl http://example.com/archive |
tee |
{{
  md5sum
  tar xzf -
}}
```

Through its syntax, *dgsh* allows the expressive and efficient specification of complex sophisticated process graphs consisting of standard Unix tools, as well as third-party and custom-built components.

Such graphs can be used for processing data sets and streams, with gains in throughput achieved by eliminating the use of temporary files and by having processes executing on multiple processor cores.

In the following sections we will overview the syntax and semantics of *dgsh* (Section 2), detail its design and implementation (Section 3), present its use through representative motivating examples (Section 4), and evaluate its performance and expressiveness (Section 5). Section 6 outlines related work in the area. Finally, Section 7 concludes this paper with a short discussion and an overview of directions for future work.

2 Syntax and Semantics

Dgsh extends the syntax of the Unix Bourne shell [15] and the affordances of some key Unit tools. The extensions involve a) the provision of an inter-process communication (IPC) mechanism through a syntax that allows the specification of pipelines communicating in a directed acyclic process graph, and, b) the ability of the specified commands to receive or produce multiple input/output (I/O) streams. These changes allow communicating processes to perform non-uniform non-linear processing.

In abstract terms, a DAG is understood as a recursive composition of boxes b with $\mathrm{In}(b)$ ordered inputs and $\mathrm{Out}(b)$ ordered outputs. A box's number of inputs or outputs may be designated "flexible", which can take values $[0, \infty]$.

A composition step (symbolized by `|`) matches in order the outputs L of one box b_1 to the inputs R of another b_2. A pairwise composition may involve at most one box with flexible input and one box with flexible output. In this case flexible inputs or outputs may expand to cover the other box's cardinality of outputs or inputs, with the cardinality of a flexible element on the other side counting as one. The cardinalities of L and R must be equal, either directly or by expanding through flexibility.

Boxes can be recursively grouped by bracketing an ordered set of (asynchronously executing) boxes using a pair of double curly braces (`{{ }}`). The inputs and outputs of the group are those of the grouped boxes, ordered according to the order of the boxes in the group. At most one box in a group may have flexible input or output; the corresponding flexibility is inherited by the group.

```
⟨dgsh_block⟩      ::= '{{' ⟨dgsh_list⟩ '}}'

⟨dgsh_list⟩       ::= ⟨dgsh_list_item⟩ '&'
                    | ⟨dgsh_list_item⟩ '\n'
                    | ⟨dgsh_list_item⟩ ';'
                    | ⟨dgsh_list_item⟩ ⟨dgsh_list⟩

⟨dgsh_list_item⟩  ::= ⟨simple_command⟩
                    | ⟨dgsh_block⟩
                    | ⟨dgsh_list_item⟩ '|' ⟨dgsh_list_item⟩
```

Fig. 1. The *dgsh* syntax.

Concretely, the syntax of *dgsh* is formally defined through the BNF grammar appearing in Fig. 1. A *dgsh* script follows the syntax of a *bash* shell script with the addition of *multipipe blocks*. A *multipipe block* starts with the symbols "`{{`" and ends with the symbols "`}}`". Data may be piped into and out of the block through multiple pipes.

A multipipe block can contain elements to execute *simple commands* (following the Bourne shell terminology), other multipipe blocks, or pipelines composed of the previous two types of elements. Commands not connected with a pipe execute asynchronously when connected with the corresponding connector offered by the shell (&), the semicolon connector (;), or the newline (\n) character. In this way *dgsh* enforces parallel asynchronous processing of the commands given within a multipipe block.

Listing 2. A *dgsh* script that identifies spelling errors

```
tee |
{{
  {{
    # Obtain text's words; one at each line
    tr -cs A-Za-z \\ n |
    # Convert capital letters to lower case
    tr A-Z a-z |
    sort -u
    # Ensure dictionary is sorted consistently
    sort /usr/share/dict/words
  }} |
  # Identify words that are not in the dictionary
  comm -23
  # Format the text's words into lines
  fmt
}} |
# Color words that are not in the dictionary
grep -F -f - -i --color -w -C 2
```

To provide the reader with a feeling of the syntax, an example *dgsh* script is shown in Listing 2. The script finds misspelled words in its standard input based on a scheme used by the first draft of the Unix spell program [16], and, as a modern extension, it also highlights them in the original text. The script works as follows. First, the *tee* command copies its text input to two destinations: the inner multipipe block and the *fmt* command which formats the text into neat lines. Within the inner multipipe block two command sequences generate two sorted streams of words: one with all words in the text, and one with all the words in the

system dictionary. The text's words are generated by first having *tr* map all the non-alphabetic characters into newlines, second by having another *tr* instance map uppercase characters to lowercase, and third by sorting the output into a list of unique words. The two streams are then passed from the inner multipipe block to the *comm* command, which compares the two sorted sequences and outputs only those records that appear in the first one (i.e., the misspelled words). Finally, the *grep* command receives from the two input channels of the outer multipipe block a) the misspelled word patterns to match, and b) the stream of original text in which to find them and highlight them.

3 Design and Implementation

Dgsh is made available an open source software system comprising

- a modified implementation of the *bash* [14] Unix shell,
- a set of Unix tools adapted to utilize the availability of multiple I/O channels,
- a wrapper process and a suite of Unix tools wrapped with it to become compatible with *dgsh*,
- a few newly-developed tools to aid the construction of DAG processing networks,
- a library implementing the *dgsh* interprocess communication protocol, which involves a negotiation procedure for connecting the communication channels of the DAG's processes, and
- a communication channel concentrator, which is used to connect the DAG's processes into a ring network topology during the negotiation phase.

In brief, these elements work together as follows. *Dgsh* scripts invoke the modified *bash* shell with the name *dgsh* (or pass to *bash* the `--dgsh` command-line flag) in order to use the *dgsh* constructs detailed in Section 2. The scripts can use some Unix tools, such as *tee*, *cat*, or *join*, which we adapted to take advantage of the *dgsh*'s support for multiple I/O channels. These tools can read input (*cat*, *join*) from multiple upstream process, or produce output (*tee*) for multiple downstream ones. *Dgsh* scripts can also use any other Unix tool, which is automatically or manually wrapped in order to become compatible with the *dgsh* interprocess communication protocol.

When the *dgsh* shell parses a multipipe block graph, it initially connects all processes together into a ring topology (Section 3.1). It connects nodes that fan-in from multiple inputs or fan-out to multiple outputs through the communication channel concentrator. The ring setup allows the processes to allocate and obtain the required communication channels (pipes) by means of the *dgsh* interprocess communication protocol and the corresponding negotiation procedure (Section 3.3). All *dgsh*-compatible programs are linked with the *dgsh* library, which implements the negotiation procedure and hands to the participating processes correspondingly allocated pipe endpoints.

Dgsh uses internally two powerful underutilized IPC mechanisms: socket pairs and Unix domain sockets. It builds on them to provide higher-level abstractions in the same way as high-level programming languages are implemented using low-level machine instructions. Socket pairs are unnamed pairs of connected sockets. Compared to traditional Unix pipes, socket pairs offer bidirectional communication and the ability to transmit file descriptors between processes. *Dgsh* uses them during the negotiation phase, which allocates and distributes the pipes that will be used to transfer the processing data (Section 3.1). Unix domain sockets are another low-level mechanism. They are difficult to use, because few standard Unix utilities provide a shell-level interface to them. *Dgsh* uses them for transferring data between arbitrary processes through the stored value inter-process communication commands described in Section 3.2.

3.1 Shell Extensions and Ring Setup

Dgsh extends the *bash* shell [14] with multipipe blocks that allow the construction of directed acyclic process graphs.

Bash's modification involves:

- the parsing of multipipe blocks,
- the connection of all processes into a ring communication topology, using concentrator processes where required,
- the use of socket pairs, rather than pipes, to connect the ring's processes together,
- the setup of the DGSH_IN and DGSH_OUT environment variables, which *dgsh*-compatible processes use in order to negotiate on the corresponding I/O side(s) to obtain multipipe I/O communication channels,
- the definition of the *call* alias,
- the setting of a path to the installation location of the adapted programs and executable scripts for wrapped commands,
- the adjustment of signal handling to ensure all processes on the graph are correctly terminated, and
- the support of visualization.

In order to communicate the required and available connections between processes, a negotiation message block has to pass (multiple times) through all the graph's processes. For this to happen, the standard I/O channels of connected processes should support both read and write access, so that when the message block reaches the end of a graph's leaf part it can travel back to its origin. The Unix pipes that shells use to connect processes, only allow communication in one direction. To address this problem we modified *bash* to initially connect processes in *dgsh* graphs, instead of pipes, with unnamed pairs of connected sockets (*socket pairs*), which support bidirectional communication.

In addition, at the input and output side of multipipe blocks, a helper program, the *dgsh* concentrator (*dgsh-conc*), allows multiple processes to be connected for passing around the message block. The modified *bash* dynamically creates and executes each concentrator process with arguments the number of processes that will be connected to it and its type, that is input or output. Then *bash* connects the concentrator with the I/O channels of the processes attached to it.

Thus, the concentrator, acts as a network switch between producer and consumer processes that will communicate in a one-to-many or many-to-one fashion. It passes the negotiation message block among the processes connected to it according to routing rules illustrated in Fig. 2. The concentrator merely facilitates the negotiation procedure by passing around the message block, it does not participate in it. It is required a) to allow processes with a single input or output to communicate with many, and b) to establish a circular

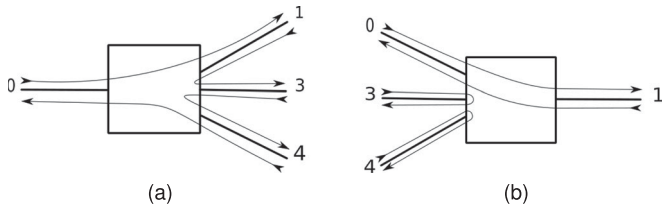


Fig. 2. The *dgsh* concentrator: Connecting (a) the standard output of a single producer process to the standard input of many consumer processes, and, (b) the standard output of many producer processes to the standard input of a single consumer process.

path traversing all the graph's nodes. When the negotiation procedure finishes, concentrators are disconnected from the graph of communicating processes and terminate.

Fig. 2 details how processes within a multipipe block are connected through the concentrator. The numbers represent file descriptors available for input and output (0 is the standard input; 1 is the standard output) and the arrows show the message routing enforced by an input or output concentrator. A many-to-many connection is implemented through the pairing of an input concentrator to an output one. A concentrator process can also be connected to another one when multipipe blocks are directly connected together or nested within one another.

The communication flow that results by following the routing rules of concentrators corresponds to a depth first search algorithm. Fig. 3 presents this flow on the process graph of the short example script given in Listing 2 and illustrated in Fig. 6. The key elements of the ring communication setup algorithm are the following.

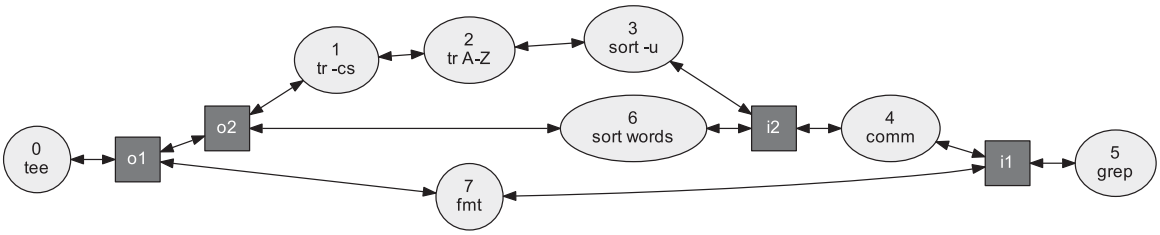
- Processes on a linear path send the message block to the next one; at the end of a path (*dgsh* processes that receive *dgsh*-compatible input, but are not connected for *dgsh*-compatible output), the message block is sent back toward its origin.
- Output concentrators communicate the block to their output channels in order; in this way they enforce traversing of unexplored paths.
- Input concentrators allow the message block to continue its course deeper into the graph only when it comes from their first input (the standard input channel); when they receive it from other input channels they send the message block back, thus pruning the multiple traversal of the explored path.

Listing 3. Compare C source files in two directory hierarchies

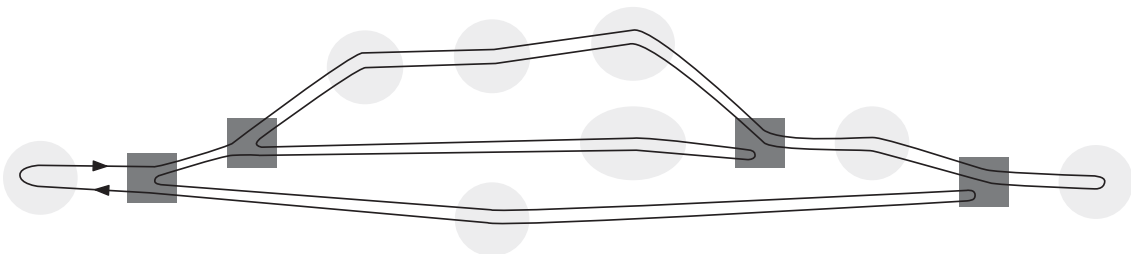
```
{{
  find $1 −type f −name '∗.[ch]' −print | sort
  find $2 −type f −name '∗.[ch]' −print | sort
}} |
join
```

In graphs where there are more than one paths to start from, there is no way to switch between unexplored paths. As an example, consider Listing 3, which results in the graph depicted in Fig. 4. The problem arises, because an input concentrator is put before *join* to prune explored paths, but an output concentrator, which would communicate the message block from path to path, is absent. In these cases we create a special output concentrator that takes no input and merely connects the candidate initiating processes. There the concentrator plays the role of the initiating process. It creates the message block and sends it to the first unexplored path according to the routing rules. When the concentrator receives the message block back, that means that a starting path is exhausted. Then the concentrator routes the message block to the next path until all paths are explored.

The ability to construct sophisticated processing topologies brings with it the need for visualizing them. The generated processing graph can be visualized by running *dgsh* with the environment variable DGSH_DOT_DRAW set to the name of a file where the graph's representation will be written. Two graphs are output for each script, the fully connected graph used during the negotiation and the one that



(a) The socketpipe connections between processes, including the output (oX) and input (iX) concentrators



(b) The message block travel path through the above processes

Fig. 3. Process connections and message block flow during the negotiation procedure for the example script in Listing 2.

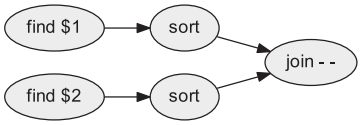


Fig. 4. Compare C source files in two directory hierarchies.

is used during the actual processing. The graphs are output in a format directly processable by *GraphViz* [17] *dot* [18], which can easily render the graph's specification into a variety of output formats. The images depicted in Fig. 7 were generated in this way.

Finally, to aid the troubleshooting of complex processing topologies, a debugging command-line option makes *dgsh* output additional information, such as the indexes of the processes on the graph.

3.2 New and Modified Unix Tools

Dgsh requires tools to be aware of it in order to participate in the negotiation procedure and retrieve the connections required for their operation. In addition, the multipipe capabilities of *dgsh* allow tools to offer, as part of their interface, multiple stream inputs or outputs. This is done when tools that naturally process multiple input streams (e.g., *join*, *comm*, *sort –merge*) or generate multiple output streams (e.g., *grep*, *comm*) expose their input and output channels to *dgsh*.

Based on this insight, we decided[1] to adapt some select existing Unix tools to take advantage of *dgsh*'s capabilities. Note that this also happened when the original Unix pipes were implemented: a frantic effort converted existing Unix tools into filters [1]. With *dgsh* the terms *standard input* and *standard output* that trace back to 1963 and the SNOBOL programming language [19] often lose their special meaning.

The tool *comm*, for instance, which is used to find common lines between two input streams, can currently accept one of the streams from its standard input. A modified version of it for use with *dgsh* can take two stream inputs. In addition, *comm* provides a single output configurable through command-line flags, which can contain lines occurring only in the first stream, only in the second stream, or in both. The modified version can provide all types of differences between the two inputs in three different output streams.

Table 1 presents eight Unix tools adapted to support *dgsh* pipelines, as well as the *dgsh* helper tools with their input and output affordances. Of the Unix tools, *grep* can take both the pattern to match and the text to search as streams and can provide matching and non-matching lines and files in four different output streams. The *paste* tool can horizontally stitch together an arbitrary number of input streams, while *sort* can merge-sort multiple input streams. The modified *diff* and *join* tools can receive both of their inputs through pipe streams.

Dgsh-tee (available through file system links as *tee*, *cat*, and *perm*) will read data from an arbitrary number of input sources and send it to an arbitrary number of output sinks. All sources and sinks can be pipe endpoints obtained through the *dgsh* negotiation procedure. It resembles in its operation the Unix *tee* and *cat* programs, but offers additional capabilities required for the operation of *dgsh*. In contrast to the standard implementation of *tee*, *dgsh-tee* will buffer the data it handles, so it will prevent a deadlock or starvation when one or more sinks are unable to read data. To accommodate buffering of large data sets, *dgsh-tee* allows the specification of the main memory buffer size and a directory to use for temporary files when the main memory buffer overflows. Through a command-line option, *dgsh-tee* can be setup to scatter its input fairly across the sinks, rather than copying it to all. When this option is in effect, the input data are divided into chunks of one or more lines or blocks, and each chunk is written only to a single sink. This is useful for dividing the work among multiple processes operating in parallel. The work allocation can be performed by *dgsh-parallel*, a *dgsh* tool modelled after *GNU parallel* [22], which implements the *split-apply-combine* data processing strategy [23]. The *dgsh-tee* command also provides a command-line option to implement the generalization of the transposition Flownomial connector ${}^{a}X^{b}$ [20], [21]. This permutes the command's N inputs into N outputs in a specified order.

Dgsh-wrap is a helper tool that participates in a negotiation procedure on behalf of a specified command. This extends the programs that can work with *dgsh* to cover all command-line tools by wrapping them on the fly. Wrapping is preferable to source code adaptation for two reasons. First, wrapping tools rather than adapting their source code is a trivial operation, which doesn't require access to the source code and a build environment. Second, there are Unix tools, such as *uniq* and *tr*, whose adaptation would not open new opportunities for use, because they are filters with just a single input and output. On the other hand, the wrapping process is not used on all commands, because, as outlined at the beginning of this section, some can become more flexible, responsive, and expressive through specialized adaptation. As an example the *comm*, *cut*, and *grep* commands can distribute their output to multiple output streams, while the provided versions of the *cat* and *tee* commands reduce blocking along the *dgsh* graphs through buffering and synchronous I/O multiplexing.

Dgsh-wrap receives as arguments a command and any particular I/O requirements it has. These are specified through two command-line flags, *–mute, -m* and *–deaf, -d*, which mean that the program does not require output or input respectively. The *dgsh-wrap* tool also supports in

TABLE 1
Tools Adapted or Developed for Multipipe Use with *dgsh*

Tool	Input channels	Output channels	Flownomial connector [20], [21]
cat (*dgsh-tee*)	0–N	1	$\vee_a^k, I_a$
comm	0–2	0–3	
diff	0–2	0–1	
grep	0–2	0–4	
join	0–2	0–1	
paste	0–N	0–1	
perm (*dgsh-tee*)	1–N	1–N	${}^{a}X^{b}$
sort	0–N	0–1	
tee (*dgsh-tee*)	1	0–N	$\wedge_a^k$
dgsh-readval	0	1	
dgsh-wrap	0–N	0–1	
dgsh-parallel	0–N	0–N	
dgsh-enumerate	0	0–N	
dgsh-writeval	1	0	
dgsh-merge-sum	1	0	

1. Following a suggestion by Doug McIlroy.

command line arguments the placeholder "<|", which signifies that a command will accept additional input streams. This placeholder is then passed to the command as the file descriptor N by using the file path `/proc/self/fd/`N. The *-s* command-line flag can be used to signal that the command's standard input should also be passed to the command as a file descriptor argument.

After the negotiation's end, when all connections are ready, *dgsh-wrap* executes the command. In this way any Unix command-line program can be adapted to participate in *dgsh* pipelines with minimal effort.

To avoid the hassle of prepending *dgsh-wrap* to existing tools, *dgsh* automatically prepends *dgsh-wrap* to all non-*dgsh* compatible commands participating in a *dgsh* graph when it prepares their execution. In addition, for tools with special I/O requirements, e.g., *echo*, which takes no input, the *dgsh* distribution places into a path location looked up first, executable scripts with the same name as each tool and appropriate command-line arguments. For instance, an executable script for *echo* is placed in */usr/local/libexec/dgsh/echo* and contains the following code.

```
#!/usr/local/libexec/dgsh/dgsh-wrap -d /bin/echo
```

Dgsh-writeval and *dgsh-readval* support asynchronous data transfer between arbitrary processes through the storage of a data stream's last record into a named buffer. This record can be later retrieved asynchronously by one or more readers. Data in a stored value can be piped into a process or out of it, or it can be read using the shell's command output substitution syntax. Stored values are implemented internally through Unix-domain sockets, a background-running store program (*dgsh-writeval*) and a reader program (*dgsh-readval*).

Dgsh-writeval will read values from its standard input and make them available to other processes for reading through a specified Unix-domain socket. Thus, a *dgsh-writeval* process acts as a data store: it reads a series of values (think of them as assignments), and provides a way to read the store's current value (from the socket). By default *dgsh-writeval* will retain the last value it reads (a newline-terminated line or a data block of a specified length). The default behavior can be modified through options so that it stores a specified data window (e.g., the last 100 records) of a stream it processes.

Dgsh-readval is the data store client, which complements *dgsh-writeval*. By default it will communicate with the store specified through the path to a Unix domain socket, ask to read the last (final) record written to that store, and write the value on its standard output. Its behavior on failures is designed to circumvent various race conditions that can arise when the stores and the read clients start concurrently. An option can be used for *dgsh-readval* to ask *dgsh-writeval* to terminate.

3.3 Interprocess Communication Setup

Dgsh supports the realization of non-linear pipelines through the already described *multipipe blocks* ({{ ...}}) and an interprocess communication protocol implementing a *negotiation procedure* among the participating *dgsh* processes.

The runtime negotiation procedure is required in order to accommodate the combination of two types of commands. First, commands that change the number of required input or provided output channels based on command-line arguments, and second, commands that can dynamically adjust the number of I/O channels they can process based on upstream or downstream requirements. As a (contrived) motivating example consider the following *dgsh* pipeline.

comm −3 | **paste**

The *comm* command compares two sorted input streams and outputs records that exist only in the first one, only in the second one, or in both. Its command-line arguments allow the suppression of any of the three outputs. Therefore, a modified *dgsh*-compatible version of *comm* that sends each of the three result sets to a separate output stream, can have zero to three outputs (two in the preceding example, which suppresses the common records). Furthermore, a *dgsh*-compatible version of the *paste* command, which combines into a single line sequentially corresponding lines from multiple inputs, can support an arbitrary number of input channels. Consequently, in our example *comm* must inform *paste* that it will supply two output channels, and *paste* must adjust its behavior to read from exactly two input channels.

Before the negotiation procedure is run, *dgsh* creates socket pairs and concentrators to connect the graph's processes into a circular ring, as described in Section 3.1. Then, the *dgsh* processes connected in the ring run the negotiation procedure and also create Unix domain sockets corresponding to any stored values. The resources supporting the negotiation procedure are freed or terminated at the end of it or in case of an error.

The negotiation procedure involves collecting processes' I/O requirements to allocate dynamically the input and output connections the processes require. The phases of the negotiation procedure are:

1) communicate input and output constraints,
2) solve the I/O constraint problem,
3) communicate the solution, and,
4) create and allocate pipes.

In *phase 1*, processes communicate their input and output requirements. The procedure is initiated by a process with no *dgsh* pipe connected to its input channel or the specially connected output concentrator, if more than one such processes exist. The initiating process creates a message block to store the topology of the graph, that is, the processes and the connections between them. Each process through which the message block passes, fills in the block its identification and the number of I/O channels it requires and provides. It then dispatches the block to the next process on the graph through the ring's socket pairs. Phase 1 terminates when the initiating process receives back the message block.

In *phase 2* the initiating process, having received the message block with the I/O requirements of all processes in the graph, executes the algorithm for solving the I/O constraint problem. The algorithm makes an initial allocation of each process's fixed I/O channels to edges so that all available channels are reserved. Then it traverses the graph edge by edge and cross matches the allocations of the connected graph. We distinguish the following cases when cross matching each edge's pair of constraints.

- *Flexible – flexible*: Assign exactly one connection to the pair.

From	Output Provision	To	Input Requirement	Phase 1 Matched	Phase 1 Connected	Phase 2 Matched	Phase 2 Connected
tee	flexible	**tr** −cs	1	✗	–	✓	1
		sort words	0	✗	–	✓	0
		fmt	1	✗	–	✓	1
tr −cs	1	**tr** A−Z	1	✓	1	✓	1
tr A−Z	1	**sort** −u	flexible	✗	–	✓	1
sort −u	1	**comm**	2	✓	1	✓	1
sort words	1			✓	1	✓	1
comm	1	**grep**	2	✓	1	✓	1
fmt	1			✓	1	✓	1
Edges matched					5		9
Total edges					9		9

Fig. 5. The phase 2 I/O constraint satisfaction algorithm applied on Listing 2's script.

- *Fixed – flexible*: Assign connections equal to the fixed constraint.
- *Fixed – fixed*: If the two constraints are equal, assign the same number of connections. Otherwise, assign the minimum of the two numbers to both processes, and move the unmatched connections to another process along the appropriate graph branch, prioritizing moves to processes with flexible requirements.

Fig. 5 presents a simple application of the algorithm on Listing 2's script. The column titled "Phase 1" shows the results of assigning weight to edges with fixed constraints, while the one titled "Phase 2" shows the results after applying cross match constraints node-wise and edge-wise. Edges connecting a process with flexible requirements are resolved at the cross matching phase, because the pair's process with the fixed constraint decides the outcome.

The algorithm's output is the number of I/O connections allocated to each process. As a last step of phase 2, the result is incorporated into an updated message block, which is again sent to traverse the graph.

In *phase 3*, the initiating process communicates to all processes the message block containing the computed solution. The phase terminates when the initiating process receives back the message block.

Finally, in *phase 4*, processes with outgoing connections create the allocated number of pipes and send the output file descriptor via the *sendmsg()* system call through the socket pair connected to their standard output. Processes with incoming connections await to get the input file descriptors for their pipes from the socket pair connected to their standard input. The concentrator receives the file descriptors from producer process(es) and communicates them to consumer process(es) according to the solution found in the message block. A complication arises when multipipe blocks are nested within one another, which results in multiple concentrators connected to each other. In this case the communicated pipes have to go through a number of intermediate concentrators. After the successful completion of this phase, the concentrators terminate and processes start their normal execution.

Fig. 6 depicts the graph after the negotiation procedure, when the connections have been setup and the programs are running. Note the highlighted *sort* process in the middle of Fig. 6, which is only connected to the terminal *tee* process. This *sort* process sorts a file; therefore it neither requires nor receives an input stream.

Dgsh-negotiate is the library that implements the negotiation procedure. All processes in the graph use it—directly or through *dgsh-wrap*—in order to participate in the negotiation procedure. The library exposes a single API function.

```
#include <dgsh.h>
int dgsh_negotiate(int flags, const char *prog_name,
                   int *n_input_fds, int *n_output_fds,
                   int **input_fds, int **output_fds);
```

When the function is called, the *n_input_fds* and *n_output_fds* arguments point to the number of input and output channels respectively that the program can accommodate. When the function returns successfully, the pointed integers are set to the number of channels that the program should use. In that case, the *input_fds* and *output_fds* arguments are set by the library to point to dynamically allocated memory buffers containing the file descriptors to use for input and output. The *flags* argument is used for adjusting the function's behavior and for extending the API in the future. Currently, the only available flag is DGSH_HANDLE_ERROR. This modifies the function so that if an error occurs during the negotiation, it will print an

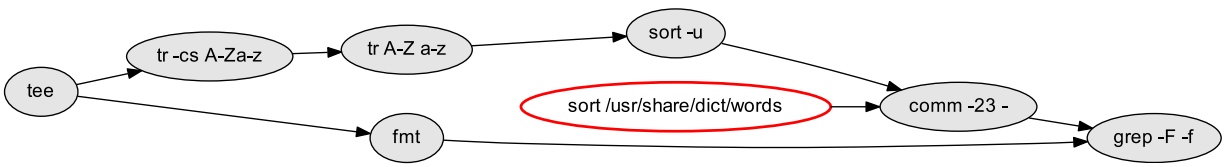


Fig. 6. Visualisation of data flow along the process graph expressed in Listing 2 during execution.

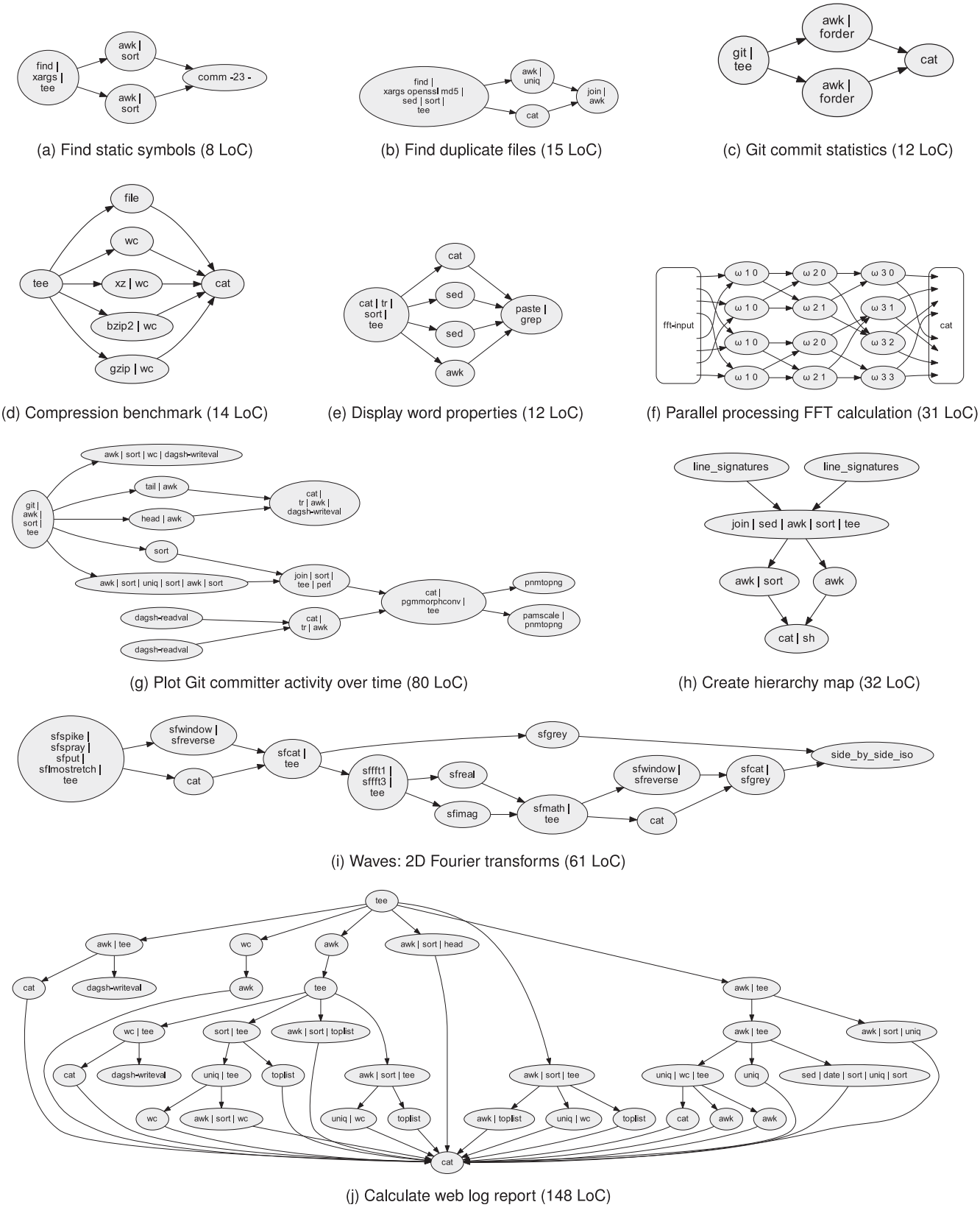


Fig. 7. Toplogies of some examplar *dgsh* applications, and commented lines of code required for their implementation.

error message to the standard error and exit the program, rather than return with an error code. The program's name, given as the *prog_name* argument, is used for providing diagnostic output and for naming nodes in the *GraphViz* visualization files.

4 Representative Examples

We have devised numerous examples demonstrating the expressiveness, versatility, and performance of *dgsh*. Their processing topologies can be seen in Fig. 7. In the Figure,

processes are depicted as ellipses. The command arguments are not included in the diagrams in order to conserve space by eliminating non-essential details. Each sub-figure's caption ends with the lines of code required for the corresponding implementation. The source code for all examples is available online;[2] what follows here is a brief description of the processing performed by each depicted example, and the *dgsh* facilities demonstrated by it. Other examples included in the *dgsh* distribution compute properties of an incoming text stream, calculate code metrics, and process nuclear magnetic resonance in-phase/anti-phase channels obtained from heteronuclear single quantum coherence spectroscopy.

To identify **C/C++ symbols that should be declared with file-local visibility** (*static*) the example in Fig. 7a starts with a *find* command combined with *nm* to produce all defined symbols. The symbols are then provided to two separate pipelines that filter them to produce all defined or exported symbols and all undefined or imported symbols respectively. The two streams feed the input of *comm* that only prints the symbols found in the first stream, that is, the exported ones that are unused in other files, which should be declared *static*.

The task of *finding duplicate files* in a directory hierarchy can be easily and efficiently implemented by calculating a hash for each file, ordering those hashes, and finding duplicate ones. The use of standard Unix commands for this task is difficult, because *uniq*, which can find duplicate lines, works at a line rather than field level, and therefore requires the isolation of the hashes from the path names. Using *dgsh* streams, the hashes of duplicate files can be isolated and then joined again with their names after the processing, by using the Unix relational join command, *join*, with the files' hash as the key. The corresponding sequence is depicted in Fig. 7b.

Listing 4. Calculate commit statistics

```
# Order input records by their frequency
forder() { sort | uniq −c | sort −rn }
git log −−format="%an:%ad" −date=default "$@" |
tee |
{{
  echo "Authors ordered by number of commits"
  awk −F: '{print $1}' |
  forder
  echo "Days ordered by number of commits"
  awk −F: '{print substr(2, 1, 3)}' |
  forder
}} |
cat
```

The simple sequence in Fig. 7c **processes the output of a** git log *command*, and lists the authors and days of the week ordered by the number of associated commits. A single (often expensive) pass through *Git*'s revision graph is used to generate both data streams. The script demonstrates the use of standard shell functions whose contained commands take part in the negotiation procedure. The complete source code of the script is included in Listing 4.

The sequence in Fig. 7d reads data from its standard input, and **reports file type, length, and compression performance** for that data. The data never touch the disk. The sequence demonstrates the use of multipipe blocks to efficiently parallelise computations and gather their output in a deterministic order.

In the example shown in Fig. 7f, four input streams are combined through the Unix *paste* command, to **list words with specified properties** in the script's standard input: those containing a two-letter palindrome, those containing four consonants, and those longer than 12 characters.

The sequence in Fig. 7f implements a **fast Fourier transform** (FFT) via matrix factorization through the parallel processing of the signal flow [24]. The ω nodes are defined as follows:

$$\omega\ s\ n \equiv \omega^n_{m=2^s} \equiv \left(e^{\frac{2\pi i}{m}}\right)^n$$

Each stage of the transform is connected to the next one through the permutation command *perm* (${}^aX^b$), which reorders the flow as required by the signal flow graph's so-called butterfly operations. (To save space, the *perm* commands are not shown in the Figure.)

Processing with *dgsh* is not restricted to text streams. The more complex example illustrated in Fig. 7g will process a project's Git history, filter the intermediate results, and create through morphological convolution two PNG **diagrams depicting committer activity** over time, with the most active committers appearing at the diagram's center vertical. The generated image is drawn in parallel by two processes: one that converts it directly into PNG format and one that scales it before the conversion.

Consider the (real world) task of **mapping files and directories that have moved** during a project's evolution from their old to their new location. This task is performed by the script shown in Fig. 7h. Given two directory hierarchies A and B passed as input arguments (where these represent a project at different parts of its lifetime) the script will copy the files of hierarchy A to a new directory, passed as a third argument, corresponding to the structure of directories in B. This is done by calculating signatures for all lines in each hierarchy, joining them by file name and content, generating commands for copying files and creating directories, and then ordering those commands so that directories are created before files are copied to them. The script demonstrates the use of *join* to gather results from streams in an output multipipe block, and the use of *tee* to order asynchronously produced results from it. It also demonstrates the use of a new *bash* alias, *call*, which allows exported *bash* functions to participate in *dgsh* pipelines. The function *line_signatures* calculates signatures for all lines in a hierarchy. Behind the scenes, *line_signatures* is executed by another *bash* instance that is wrapped with *dgsh-wrap* in order to participate in the negotiation procedure. The *call* alias command resolves to the following.

```
dgsh−wrap bash −c 'line_signatures'
```

The example shown in Fig. 7i, demonstrates **using the tools of the Madagascar research environment** [12], and also having two independent *dgsh* pipelines in the same script. (Only one is shown in the Figure.) The script creates two graphs: a broadened pulse and the real part of its 2D Fourier transform, and a simulated air wave and the amplitude of its 2D Fourier transform. Data are sent to Madagascar commands, such as as *sffft1*, which performs a fast Fourier

2. http://www.spinellis.gr/sw/dgsh/#examples

transform along the first axis, *sfwindow*, which extracts a data set's window portion, *sfgrey*, which generates a raster plot, and *sfread*/*sfimag*, which extract a complex data set's real or imaginary part. The generated data streams have a mathematical function applied to them through *sfmath* as follows.

```
}} |
sfmath nostdin=y re=<| im=<| \
  output="sqrt(re*re+im*im)" |
```

Sfmath can receive input either from its standard input or from files. In our example, *sfmath* takes two input streams mapped to arguments "re" and "im" respectively. The output is computed according to the formula specified in the "output" argument given the two input streams. Because the single standard input channel is insufficient due to the presence of two input streams, we feed input to *sfmath* through two "<|" placeholder file arguments and force it to ignore its standard input channel.

Finally, the example shown in Fig. 7j demonstrates the use of *dgsh* for the **analysis of web log data**. When the corresponding stream is given as input a web server's log data, it will produce a report indicating the total number of accesses, web pages, hosts, domains, area requests, and bytes served and accesses per hour, date, and day of week. Computations are efficiently parallelised through a hierarchy of multipipe blocks up to four levels deep.

5 Evaluation

We analyzed the performance and expressiveness of *dgsh* by comparing it with currently used alternatives.

5.1 Performance

The performance of *dgsh* was evaluated by running the diverse example programs that were written to illustrate its expressiveness on large data sets. Many of the benchmarked programs were presented in Section 4; all are included together with the measurement scripts in the *dgsh* distribution.

The key numbers associated with the benchmarks are listed in Table 2. A row for each run lists the example's name, the benchmarked implementation (*sh* is used to denote the use of linear pipelines and temporary files to distribute data to multiple sinks), the duration of the negotiation procedure for *dgsh* scripts (Neg.), the real (elapsed), user, and system time associated with the run, and the *dgsh* speedup compared to the particular alternative implementation ($T_{\text{imp.}}/T_{\text{dgsh}}$). A more detailed description of the table's columns follows.

The second column of Table 2 states how each program was run. The alternatives are a) a pipe network constructed by *dgsh* and, b) having *sgsh*, the ancestor of *dgsh*, transform the program into an equivalent one that uses only linear pipelines and temporary files. For a few examples other alternative implementations were also benchmarked. The *web-log-report* example was written to reimplement a more than a decade old Perl script, which was benchmarked as well. After seeing the large speedup obtained by the *dgsh* implementation, a compatible implementation was also coded in Java, which, in turn, prompted the implementation in Java of the *text-properties* example. In addition, the *ft2d* example was also benchmarked using the *SCons* (a name derived from Software Construction) front-end, which is distributed with the Madagascar research environment as a superior alternative to the classic *make* utility.

TABLE 2
Performance and Resource Utilization of *dgsh* Compared to Alternative Implementations

Task	Imp.	Time (s)				Speed.
		Neg.	Real	User	Sys.	
code-metrics	dgsh	0.06	100.48	190.44	10.86	
	sh		102.42	198.76	4.53	1.0
commit-stats	dgsh	0.02	5.01	9.02	3.59	
	sh		6.06	6.11	1.42	1.2
compress-compare	dgsh	0.08	29.74	38.13	0.32	
	sh		37.85	37.65	0.11	1.3
duplicate-files	dgsh	0.11	21.81	20.36	1.94	
	sh		24.78	20.69	1.99	1.1
ft2d	dgsh	0.09	19.38	22.54	5.98	
	sh		23.16	20.79	3.32	1.2
	SCons		25.13	22.06	3.39	1.3
map-hierarchy	dgsh	0.06	102.29	57.96	15.29	
	sh		136.17	51.55	11.01	1.3
spell-highlight	dgsh	0.11	7.63	7.61	0.27	
	sh		9.38	9.36	0.20	1.2
static-functions	dgsh	0.09	5.55	3.96	0.47	
	sh		7.13	6.80	0.59	1.3
text-properties	dgsh	0.01	17.18	80.29	0.58	
	sh		30.96	51.88	0.20	1.8
	Java		13.02	13.32	0.18	0.8
web-log-report	dgsh	0.07	6.75	22.56	5.54	
	sh		15.57	15.13	2.95	2.3
	Java		11.13	12.08	0.19	1.6
	Perl		54.01	48.61	5.29	8.0
word-properties	dgsh	0.02	3.97	3.88	0.1	
	sh		3.85	3.88	0.06	0.9

The same programs were also run with *dgsh* and *sgsh* on an empty data set, to examine the overhead imposed by *dgsh*'s negotiation procedure. The third column of Table 2 presents this cost. We observe that it amounts to a fraction of the script's execution time ranging approximately between 10-100 milliseconds irrespective of the size of the graph.

Regarding the speedup shown in the last column, $T_{\text{imp.}}$ is the execution time of a program in terms of real (elapsed) time using the specified alternative implementation to *dgsh* while T_{dgsh} is the execution time of a program using *dgsh*, again in terms of real (elapsed) time.

The following datasets were used:

- The text of three Project Gutenberg books repeated ten times: (*History of the United States* by Charles A. Beard and Mary Ritter Beard, *The Adventures of Sherlock Holmes* by Arthur Conan Doyle, and *Les Misérables* by Victor Hugo). This was used for benchmarking the programs *spell-highlight*, *word-properties*, *compress-compare*, and *text-properties*.
- The web server log clarknet_access_log_Aug28 retrieved from the Internet Traffic Archive [25]. (Used by *web-log-report*.)
- The Linux Git repository[3] between commit points 1da177e4c and 899552d6e. (Used by *commit-stats*, *code-metrics*, and *map-hierarchy*.)

3. https://github.com/torvalds/linux.git

The tests were run on an eight core (two four-core Intel Xeon E5-1410 processors operating at 2.80 GHz) machine with 12 GB of RAM, and five 1.8 TB disks attached through a 6 Gb/s PERC H 710 PCI Express RAID controller, running the Debian GNU/Linux 8.6 (jessie) distribution. To minimize the effects of caching, all scripts were run twice in an otherwise idle system, and only the results of the second execution were retained.

As can be seen, in nine out of eleven cases *dgsh* offers a speedup (the job executes faster in real time) compared to the plain *sh* performance, in one case the performance is the same, and in one (*word-properties*) its performance is slightly lower. Also evident is the fact that most *dgsh* implementations are less efficient that the plain *sh* ones, consuming more CPU resources. Apparently, this happens, because they utilize more CPU cores, trading CPU utilization to gain performance. In one case (*web-log-report*) the *dgsh* implementation outperformed significantly not only an equivalent Perl script, but also a Java program running the same task.

5.2 Expressiveness

We evaluate the expressiveness for *dgsh* by comparing programs that can be written in it against some generally available alternatives: the process substitution offered by the *bash* shell and the explicit construction of named pipes. Although formal ways to evaluate a language's expressiveness have been proposed [26], our goal here is to demonstrate the practical merits of using *dgsh* rather than claim expressiveness superiority in a formal sense.

Consider as an example, the following short *dgsh* script, in which two processes, *b* and *c*, receive their input from process *a* and send their output to *d*.

```
a | {{
   b
   c
}} | d
```

The two halves of the process connection graph can be expressed using the process substitution mechanism of *bash*: *a* can send its output to *b* and *c*,

```
a >(b) >(c)
```

and *d* can derive its input from *b* and *c*.

```
d <(b) <(c)
```

Unfortunately, the process substitution mechanism does not offer a way to express both halves together, because the syntax that allows a process to substitute a named file can only be used as an argument to a single process (specifying an input or an output file, but not both).

Connecting all the example's processes together can be achieved by creating named pipes and redirecting through them, as shown in Listing 5. However, this alternative is more verbose, requires the meticulous specification of all input and output connections through the named pipe endpoints, and entails the explicit construction and removal of the named pipes.

Some Unix shells also offer a co-process mechanism that automatically associates a process with pipes connected on its input and output. Yet this facility is typically restricted to a single asynchronously running process, thus prohibiting the creation of more complex process graphs.

Listing 5. Connecting processes through named pipes

```
mkfifo b.inp b.out c.inp c.out
d b.out c.out &
b < b.inp > b.out &
c < c.inp > c.out &
a b.inp c.inp
rm b.inp b.out c.inp c.out
```

6 Related Work

The ideas behind *dgsh* and related systems trace their origins to the theory [27], methods [28], and languages [29] associated with *data flow programming* [20], [21] and *process networks* [30], [31]. A number of shells developed in the past have addressed the problem of specifying non-linear pipe topologies in diverse ways. The approach adopted by *dgsh* improves over those designs in terms of syntax, generality, performance, and compatibility with a widely used shell.

State of the art frameworks for expressing and executing complex data flows and process networks are of two types: a) workflow management and task scheduling frameworks designed primarily for batch processing, such as *Luigi*,[4] Airflow,[5] and Dask[6] and, b) stream processing engines, such as Apache Spark Streaming [32], Apache Storm,[7] and Apache Flink [33].

The frameworks for batch processing offer scalable parallelized workflow management of DAG topologies in a fault-tolerant manner, but do not allow I/O communication between the processes in a streaming fashion. Only Dask offers stream processing, which is limited to standard Python queue objects. In addition, all three frameworks run on top of the Python platform and require explicit configuration of a process graph through Python code for defining aspects such as dependencies and sinks.

The stream processing engines are fault-tolerant, scalable, and support stream processing algorithms, such as windowing. All three support DAG data flows in specific programming languages, for the most part Java, Python, and Scala. In Storm, support for more programming languages is possible through an adapter library that implements the communication protocol for sharing data in JSON format. Adapter libraries are currently available for Python, Ruby, and Fancy.

One issue with stream processing regards unbounded memory requirements. Storm escapes unbounded memory requirements by allowing users to tweak the data production and consumption ratio, increase the buffers' size for all or a specific topology, and set the maximum number of pending tuples (*max spout pending*), which configures the rate of data emission from external sources. Flink adjusts to the pressure of data production at higher rates than data consumption by slowing down the producers. Spark Streaming divides the data into small batches that are

4. https://github.com/spotify/luigi
5. https://airflow.incubator.apache.org/
6. http://dask.pydata.org/en/latest/
7. https://storm.apache.org

allocated to workers and uses dynamic load balancing to avoid struggle effects.

Compared to the data flow and process network frameworks, *dgsh* supports stream processing of DAGs of commands written in any programming language through a *pipes and filters* architecture using the syntax and semantics of the *bash* shell. Regarding unbounded memory requirements, *dgsh-tee*, which copies, gathers, or scatters data from or to multiple sources or sinks, provides command-line arguments for configuring the buffering of data. *Dgsh-tee* can use all of a machine's available memory and also temporary files if the main memory space is not sufficient. On the other hand, *dgsh-tee* cannot fine-tune the I/O rate.

Rochkind's *2dsh* (reference [34] as cited by Korn [35]) reportedly supported a network of processes, but has not seen further development. Tom Duff's *rc* shell [36] supports trees through *pipeline branching*. The branches are created by writing a command's file argument in a form that denotes the standard output of a specified command's execution. For example, the command `cmp <{old} <{new}` will compare the output of command *old* with the output of command *new* while `tee <{proc1} <{proc2}` will have both *proc1* and *proc2* process the standard input of *tee*. A similar syntax, with the use of brackets instead of curly braces, is also supported by the *bash* shell under the term *process substitution* [14, p. 219]. The *rc* and *bash* approach provides similar affordances to *dgsh*'s in terms of multipipe blocks, but, unlike *dgsh*, the approach does not allow the combination of output multipipe blocks with input ones.

The *gsh* shell [37] allows the creation of arbitrary graphs by specifying named channels as input and output for commands. Similarly to *dgsh*, *gsh* uses a specially crafted command, TEE, for distributing the output of one process to many others and relies on custom-modified programs, with uppercase names such as DIFF, GREP, SED, and SORT. In contrast to *dgsh*, *gsh* supports interactive use through a prompt that indicates when the specified graph is fully connected. In addition, *gsh* does not allow channels to be passed to processes as arguments, while with *dgsh-wrap* any tool can participate in a *dgsh* script. The *gsh* author hints that file descriptor drivers (`/dev/fd`*N*) and FIFOs could be used to address this deficiency.

Another system that allows the creation of arbitrary process communication graphs is *expect*. This differs significantly from the other shells described so far, because inter-process communication is performed explicitly through a high-level procedural language based on *Tcl* [38]. The *expect* tool, targeting the need to automate the operation of interactive applications, bases inter-process communication on pseudo-terminals rather than pipes. Consequently, its users pay the price of interfacing with the system's pseudo-terminal driver.

Two systems, MTX [39] and VUFC [40], solve the mismatch between a complex graph topology and what can be specified in a linear textual language, by using a graphical notation and a corresponding editor. From the two systems MTX is more powerful, because it allows the specification of DAGs, whereas VUFC only allows the specification of processing trees. In addition, MTX supports the dynamic reconfiguration of the graph's topology. A significant limitation of MTX is that it merges internally the results of multiple streams into one, with the option of adding a marker to indicate the stream from which particular data arrive. This puts the onus of demultiplexing data to applications, which must be crafted to deal with such markers. The MTX shell performs the dynamic reconfiguration by tying processes to pseudo-terminals and, therefore, as is also the case for *expect*, imposes the corresponding performance penalty to communicating applications.

Given the stagnation of CPU clock frequencies and the widespread availability of multi-core processors and multiprocessor clusters in cloud data centers, the efficient use of these resources, especially for homogeneous data processing, has attracted considerable interest. Two representative systems that address the exploitation of computing clusters are MapReduce [41] and Hadoop [42]. In contrast to *dgsh*, these systems offer native support for allocating and scheduling tasks among the resources of a computing cluster. To do that they require the custom implementation of the map and reduce jobs, whereas *dgsh* can use existing tools for this purpose. However, the *Hadoop streaming* utility that comes with the Hadoop distribution,[8] allows the specification of Map/Reduce jobs with any executable program or script as the mapper or the reducer.

Microsoft Dryad [43] is a general purpose distributed execution system and cluster manager for data parallel tasks. It introduces a graph specification language for forming directed acyclic data flow graphs. Dryad schedules programs on the graph for execution on available workers in the underlying cluster and allows programs to communicate through temporary files, TCP-level pipes, and shared memory FIFOs. The API for specifying data flow graphs is available in C++.

Walker et al. [44] present an extension of *bash*'s shell pipes to form DAGs but also graphs with cycles. The authors implement the concepts of shell fork, join, and cycle, where fork and join resemble *dgsh*'s output and input multipipe blocks respectively. Despite their common points, two main differences between this work and *dgsh* are that it supports non-linear uniform processing through the parallel and distributed processing of a single program and the use of temporary files to feed input from a producer process to consumer processes.

At the multicore utilization front, GNU Parallel is a Unix command-line tool that helps executing multiple jobs in parallel [22]. It allocates the appropriate number of processes to each CPU core, splits the input among the processes, and gathers the output in a deterministic manner. GNU Parallel by default groups the output of the specified jobs, printing the results only once a command is finished. This requires storing the output in temporary files with an associated overhead in space and CPU resources, and thus prevents the pipeline from streaming data from start to end. In contrast, *dgsh* encourages the use of deterministic $O(N)$ gather operators, where N is the number of input processes. This approach allows the efficient processing of infinitely long data streams.

Another entrant in this category is PUSH, a Unix shell explicitly targeting data intensive supercomputing (DISC) [45]. PUSH allows the homogeneous splitting of work among processes residing on multiple hosts through an abstraction termed a *multipipe*. An interesting feature of PUSH is that, rather than being data agnostic, it allows the specification of

8. http://hadoop.apache.org/docs/stable/streaming.html

the input and output record separators, thus allowing the automatic multiplexing and demultiplexing of data.

7 Conclusion

In the preceding sections we showed that *dgsh* supports the concise and readable implementation of many tasks that call for a non-linear arrangement of data flow through pipes. Most examined cases are associated with a speedup compared to an implementation involving temporary files.

The facilities provided by *dgsh* address two trends in modern computing environments: sophisticated data processing pipelines and multi-core or distributed executions. The non-linear pipe configurations that *dgsh* supports, eliminate the storage and performance overhead of temporary files, and provide additional opportunities for processes to operate in parallel. Increased opportunities for parallelism can be readily exploited by multi-core processors. In turn, increased parallelism and reduced temporary file overheads, help achieving higher throughput in data processing applications.

Given the number of past attacks to the problem of constructing non-linear data flow configurations through a Unix shell, the widespread adoption of *dgsh* is not yet a given. However, the match of its provided features with modern needs, in conjunction with its ability to be used with an existing widely-used shell, allows for a certain amount of optimism.

Two directions for future work are the following. The naming of data streams can add more expressiveness in forming process graphs. An example concerns processes that need to process data that are available on a sibling processing stream rather than directly upstream. Second, built-in support for executing *dgsh* processes on multiple hosts, can extend the use of *dgsh* on computing clusters and distributed systems.

The improved throughput observed in data processing performed with *dgsh* hints that there may be further room for significant performance improvements by attacking the costs of context switching and data copying. One such avenue might be through the use of the fast I/O library [46], and a facility that would allow multiple cooperating processes to run as threads within a single process.

Availability

The source code and documentation of *dgsh* are made available as open source software under the Apache license at http://www.spinellis.gr/sw/dgsh. Enhancements, fixes, and issues can be submitted through the GitHub URL http://github.com/dspinellis/dgsh.

To compile and run *dgsh* a C compiler and GNU make must be available. An installation of the *GraphViz* suite is required for the visualization of *dgsh* graphs. The dgsh suite has been tested under Debian and Ubuntu Linux, FreeBSD, and macOS operating systems.

Acknowledgments

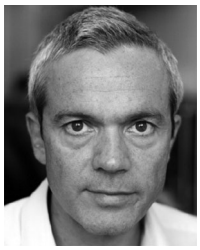
We thank Vaptistis Anogeianakis, Earl Barr, Georgios Gousios, Vassilios Karakoidas, Panos Louridas, and Audris Mockus for their help in the evaluation of *dgsh* and their constructive comments. We are especially grateful for the support, encouragement, and insightful feedback we received from Doug McIlroy over successive design iterations.

The research described has been partially carried out as part of the CROSSMINER Project, which has received funding from the European Union's Horizon 2020 Research and Innovation Programme under grant agreement No. 732223.

References

[1] D. M. Ritchie, "The evolution of the UNIX time-sharing system," *AT&T Bell Laboratories Tech. J.*, vol. 63, no. 8, pp. 1577–1593, Oct. 1984.
[2] M. McIlroy, "Summary—what's most important," Oct. 1964. [Online]. Available: http://www.thocp.net/biographies/papers/pipeline_mcIllroy.htm
[3] S. Garland, A. Dwyer, E. Pepper, A. Colvin, and N. Bourbaki, *Syst. Program. Reference Manual*, DTSS Incorporated, Oct. 1979, cSS Doc. #1-TM059. Publication number 1059.
[4] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, pp. 666–677, Aug. 1978.
[5] C. A. R. Hoare, *Communicating Sequential Processes*. Englewood Cliffs, NJ, USA: Prentice Hall, 1985.
[6] R. Meunier, "The pipes and filters architecture," in *Pattern Languages of Program Design*, J. O. Coplien and D. C. Schmidt, Eds. Reading, MA, USA: Addison-Wesley, 1995, ch. 22, pp. 427–440.
[7] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*. Englewood Cliffs, NJ: Prentice Hall, 1996.
[8] B. W. Kernighan, "The UNIX system document preparation tools: A retrospective," *AT&T Tech. J.*, vol. 68, no. 4, pp. 5–20, Jul./Aug. 1989.
[9] J. Poskanzer, et al., "Overview of Netpbm," Jan. 2014, current Aug. 2014, [Online]. Available: http://netpbm.sourceforge.net/doc/
[10] P. Wessel and W. H. F. Smith, "Free software helps map and display data," *EOS Trans. Amer. Geophysical Union*, vol. 72, no. 41, pp. 441, 445–446, 1991.
[11] F. Delaglio, S. Grzesiek, G. W. Vuister, G. Zhu, J. Pfeifer, and A. Bax, "NMRPipe: A multidimensional spectral processing system based on UNIX pipes," *J. Biomolecular NMR*, vol. 6, no. 3, pp. 277–293, 1995.
[12] S. Fomel and G. Hennenfent, "Reproducible computational experiments using SCons," in *Proc. 32nd Int. Conf. Acoust. Speech Signal Proc.*, 2007, pp. 1257–1260.
[13] C. Lattner, *LLVM*. Raleigh, NC, USA: lulu.com, 2012, ch. 11.
[14] C. Newham, *Learning Bash Shell: Unix Shell Program*. Newton, MA, USA: O'Reilly Media, Inc., 2009.
[15] S. R. Bourne, "The UNIX shell," *Bell Syst. Tech. J.*, vol. 56, no. 6, pp. 1971–1990, Jul./Aug. 1978.
[16] B. W. Kernighan, "UNIX for beginners," Bell Laboratories, Murray Hill, NJ, USA, Comput. Sci. Tech. Rep. 75, Feb. 1979.
[17] E. R. Gansner and S. C. North, "An open graph visualization system and its applications to software engineering," *Softw.: Practice Experience*, vol. 30, no. 11, pp. 1203–1233, 2000.
[18] E. R. Gasner, E. Koutsofios, S. C. North, and K.-P. Vo, "A technique for drawing directed graphs," *IEEE Trans. Softw. Eng.*, vol. 19, no. 3, pp. 124–230, May 1993.
[19] R. E. Griswold, J. F. Poage, and I. P. Polonsky, *The SNOBOL4 Programming Language*, 1st ed. Englewood Cliffs, NJ, USA: Prentice-Hall, 1968.
[20] M. Broy and G. Stefănescu, "The algebra of stream processing functions," *Theoretical Comput. Sci.*, vol. 258, no. 1/2, pp. 99–129, May 2001.
[21] G. Stefanescu, "Algebra of flownomials," Technische Univerität München, Munchen, Germany, Tech. Rep. TUM-I9437, 1994.
[22] O. Tange, "GNU parallel: The command-line power tool," *;login:*, vol. 36, no. 1, pp. 42–47, Feb. 2011.
[23] H. Wickham, "The split-apply-combine strategy for data analysis," *J. Statist. Softw.*, vol. 40, no. 1, pp. 1–29, Apr. 2011.
[24] M. C. Pease, "An adaptation of the fast Fourier transform for parallel processing," *J. ACM*, vol. 15, no. 2, pp. 252–264, Apr. 1968.
[25] P. Danzig, J. Mogul, V. Paxson, and M. Schwartz, "The Internet traffic archive," 2000, [Online]. Available: http://ita.ee.lbl.gov, current Apr. 2017.
[26] M. Felleisen, "On the expressive power of programming languages," in *Proc. 3rd Eur. Symp. Program.*, May 1990, pp. 134–151.
[27] G. Kahn, "The semantics of a simple language for parallel programming," in *Proc. Int. Federation Inf. Proc.*, Aug. 1974, pp. 471–475.

[28] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Comput.*, vol. C-36, no. 1, pp. 24–35, Jan. 1987.
[29] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language LUSTRE," *Proc. IEEE*, vol. 79, no. 9, pp. 1305–1320, Sep. 1991.
[30] W. B. Ackerman, "Data flow languages," *IEEE Comput.*, vol. 15, no. 2, pp. 15–25, Feb. 1982.
[31] E. A. Lee and T. M. Parks, "Dataflow process networks," *Proc. IEEE*, vol. 83, no. 5, pp. 773–801, May 1995.
[32] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proc. 24th ACM Symp. Operating Syst. Principles*, 2013, pp. 423–438.
[33] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache Flink: Stream and batch processing in a single engine," *IEEE Data Eng. Bull.*, vol. 38, no. 4, pp. 28–38, 2015.
[34] M. J. Rochkind, "2DSH—An experimental shell for connecting processes with multiple data streams," Bell Laboratories, Murray Hill, NJ, USA, Technical Memorandum TM-80–9323-3, 1980.
[35] D. D. Korn, "Introduction to KSH-I (issue 3)," Bell Laboratories, Murray Hill, NJ, Technical Memorandum TM-59554–860602-04. Charge Case 311531-0101. File Case 49059-6, 1987.
[36] T. Duff, "Rc – a shell for Plan 9 and Unix systems," in *Proc. Summer 1990 UKUUG Conf.: UNIX—The Legend Evolves*, Jul. 1990, pp. 21–33.
[37] C. McDonald and T. I. Dix, "Support for graphs of processes in a command interpreter," *Softw. Practice Experience*, vol. 18, no. 10, pp. 1011–1016, Oct. 1988.
[38] J. K. Ousterhout, *Tcl and the Tk Toolkit*. Boston, MA, USA: Addison-Wesley, 1994.
[39] S. a. Uhler, "MTX—a shell that permits dynamic rearrangement of process connections and windows," in *Proc. Conf. Proc.*, Jan. 1990, pp. 275–285.
[40] D. Spinellis, "Unix tools as visual programming components in a GUI-builder environment," *Softw. Practice Experience*, vol. 32, no. 1, pp. 57–71, Jan. 2002.
[41] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. 6th Symp. Operating Syst. Des. Implementation*, 2004, pp. 137–149.
[42] T. White, *Hadoop: The Definitive Guide*. Sebastopol, CA, USA: O'Reilly, 2009.
[43] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proc. 2nd ACM SIGOPS/EuroSys Eur. Conf. Comput. Syst. 2007*, 2007, pp. 59–72.
[44] E. Walker, W. Xu, and V. Chandar, "Composing and executing parallel data-flow graphs with shell pipes," in *Proc. 4th Workshop Workflows Support Large-Scale Sci.*, 2009, pp. 11:1–11:10.
[45] N. P. Evans and E. Van Hensbergen, "Brief announcement: PUSH, a DISC shell," in *Proc. 28th ACM Symp. Principles Distrib. Comput.*, 2009, pp. 306–307.
[46] A. Hume, "A tale of two greps," *Softw. Practice Experience*, vol. 18, no. 11, pp. 1063–1072, 1988.



Diomidis Spinellis is a professor in the Department of Management Science and Technology, Athens University of Economics and Business, Greece. He is the author of two award-winning books, *Code Reading* and *Code Quality: The Open Source Perspective*, as well as more than 250 widely-cited scientific papers. In 2016, he published the book *Effective Debugging: 66 Specific Ways to Debug Software and Systems*. He has served for a decade as a member of the *IEEE Software* editorial board, authoring the regular "Tools of the Trade" column. He has written the *UMLGraph* and *CScout* tools as well as code that ships with Apple's macOS and BSD Unix. From January 2015 he is serving as the editor-in-chief of the *IEEE Software*. He is a senior member of the ACM and the IEEE.

Marios Fragkoulis received the MSc degree with distinction in computer science from the Department of Computing, Imperial College London and the PhD degree on main memory data analytics from the Department of Management Science and Technology, Athens University of Economics and Business. He has published in the European Conference of Computer Systems (EuroSys) and Springer's journal of Computing. He is the developer of the PiCO QL online data analytics system and co-developer of the directed acyclic graph shell (dgsh). He is a member of the ACM and the EuroSys Society.