Compiling the uncompilable: A case for shell script compilation

Daniel L. Wang *,1,

Department of Electrical Engineering and Computer Science

Charles S. Zender,

Department of Earth System Science

Stephen F. Jenks

Department of Electrical Engineering and Computer Science, University of California, Irvine, CA 92697 USA

Preprint submitted to Elsevier *7 August 2008*

Abstract

Shells, as command interpreters, are the classical way for humans to interact with computing systems, and modern shell features have extended this basic functionality with higher-level programming language constructs. Although implementing compilation in these shell languages is generally unprofitable and intractable, many advantages, such as isolation, filesystem abstraction, security, portability, parallelization and locality optimization are possible, using standard compilation techniques. While compilation is not possible for all scripts, there exist shell scripts of a class that are, in practice, both profitable and tractable to compile and execute. This class of scripts is prevalent in the scientific computing community, where scripts are commonly used to automate data processing sequences. We describe a prototype shell compilation and implementation for these scripts, noting advantages and challenges, and illustrating the significant performance potential available. Our results show that shell compilation is a viable means of automatically identifying and exploiting high-level program parallelism using existing sequential script specification and without requiring reimplementation.

Key words: programming languages, shell languages, domain-specific language, command interpreters, optimizing compiler, locality optimization, parallel programming, dataflow, performance, workflow

* Corresponding author.
Email addresses: wangd@uci.edu (Daniel L. Wang), zender@uci.edu (Charles S. Zender), sjenks@uci.edu (Stephen F. Jenks).

[1] Supported by the National Science Foundation under grant IIS-0431203.

1 Introduction

Since the development of "RUNCOM", a shell for the Compatible Time-Sharing System (CTSS)[1] in the mid 1960s, shell-scripting has enjoyed continued popularity among users, programmers, administrators, and all those who interact with UNIX or BSD systems. Shells, i.e. command interpreters, are the classical way for humans to interact with these systems, and shell scripting *languages* extend the basic task of selecting and running programs with higher level programming language constructs. However, their focus on efficiency in interactively running and connecting programs makes them unmatched for short tasks like system initialization, administrative tasks, and compositions of simple tools, and cumbersome for larger tasks, where its compactness and specialization limits readability and generality.

The commands executed by most shell scripts are simple and short-lived, but in the scientific computing community, shell scripts form the backbone of significant data reduction and analysis tasks that can take hours or days to run serially. While parallel execution of many of the commands is feasible, it is unreasonable to expect scientists from non-computing disciplines to define dependences in a safe and effective manner.

Given their widespread use in the scientific community, we explore the performance potential of shell scripts, beginning with a description of shell language. We then propose an alternative execution model, where scripts are compiled into workflows and executed in parallel. We describe our prototype implementation, illustrating the performance improvements. Our experiences are generalized as we then discuss the important issues to be considered in any

implementation of shell compilation. Finally, we describe other work related to shells compilation and parallel program execution and summarize our case for script compilation.

2 Shell Language

2.1 Definition

Shell command languages exist as a means to direct the loading and execution of programs within a hosting operating system. For the purposes of this paper, we define shell language to mean the language accepted by a command language interpreter, the *sh* utility, as specified by POSIX [2]. Most shells implement the POSIX feature set, and share the same core ideas and purpose, although their syntaxes may differ.

2.2 Purpose

The purpose of a shell language is to provide a convenient and flexible means of expressing what program to execute and how it should be executed. Shells date back to the mid-1960s, when Louis Pouzin coined the term to describe a command language for driving the execution of command scripts (sequences), so that multiple commands could be executed on a computer in the the absence of an operator[3]. While graphical user interfaces now provide alternate means for humans to specify command execution, shell language remains the most popular choice to automate the execution of command sequences. For example, every version of Microsoft Windows has provided shell-scripting via

a batch file facility, a feature included since the earliest versions of the Windows predecessor DOS. Shell scripts are "programs of programs," and it is this usage that motivates our study.

While it is possible to invoke programs in other languages, only shell language is as purpose-built. Program execution is possible in C, for example, but using C instead of shell language would be unquestionably painful, even in interpreted form (e.g. Cint[4] and CINT[5]). [6] compares the "hello, world" program in `sh` versus C, finding their program sizes to be 29 bytes and 16 kilobytes, respectively.

2.3 Features

Shell language has a simple syntax, designed to minimize syntactical clutter in the most common cases. Execution, for example, is specified by simply naming the desired program or command followed by its arguments in a command-line. Identically-named programs are distinguished by their locations on the filesystem and are indicated by prepending this location (the *path* name) to the program name. General-purpose programming features developed out of early macro and variable capabilities, and more interactive features like command and filename completion developed for user convenience.

The shell features conditional branching through *case-esac* and *if-elif-else-fi* constructs, which are evaluated sequentially. Looping is expressed using *for-do-done* constructs, which executes a command sequence for each item in a list, and *while-do-done* or *until-do-done*, which repeat sequences as long as a particular expression evaluates as true or false, respectively. These control flow

structures let programmers specify flexible and iterative behavior analogous to control flow structures in other imperative languages.

The shell recognizes a single primitive type, string, for variables, which may exist in two namespaces. *Shell* variables are defined and exist within a particular running shell context. They exist for command and parameter substitution, as well as for control flow structures, but are not otherwise visible to other programs unless they are *exported* to the environment. *Environment* variables are managed by the operating system for the shell instance. The environment variable namespace context is inherited by programs launched by the shell, but environment variables are otherwised used similarly as shell variables. Both shell and environment variables are syntactically referenced identically, with precedence given to shell variables.

Functions or subroutines can be defined in the shell, but do not occupy separate contexts, and do not have local variables except for their positional arguments.. Variable references are always dynamically-scoped—each line is interpreted and executed one at a time.

The shell provides an interface for I/O-redirection which facilitates directing a program's standard input or output to or from a file or another program.

Job control is achieved by a syntax for background launching of programs, coupled with built-in shell commands to list jobs and to switch them between foreground and background. These facilities allow explicit specification of parallel programs (or program sequences), albeit without built-in support for synchronization.

2.4 Execution Model

Shell language exports an immediate, serial, and local execution model. Commands, whether programs or shell built-ins, are expected to be executed immediately, one after another, without any sort of batch queuing delay, on the same machine as the running shell instance. Commands are executed in the context of the running shell (including shell variables) as well as the "environment," which includes operating system-managed per-process context such as user privileges, environment variables, and filesystem context. Each running shell instance maintains separate context, and children shell instances, e.g. those used for executing shell scripts, inherit a copies of their parent contexts. While programs can be run in parallel, the shell commands which launch and toggle jobs between foreground and background are executed immediately and serially.

2.5 Limitations

While shell language has many features in common with more general-purpose languages, its limitations make it unsuitable for programming long, complex programs and applications. Only scalar variables are supported, and while similar type-less approaches exist in other languages, shell arithmetic is integer-only and limited . Objects, records, containers, or array data structures are not supported (arrays are supported in other popular shells). Variables exist in a single global scope, making recursion difficult at best. Unsurprisingly, shells also lack synchronization primitives for concurrent programming.

Additional evidence of shell language's limitations is seen in the popular use

of shell "one-liners" that rely on tool languages like `AWK` or `sed`, or more full-featured languages like Perl. These languages, freed need for interactivity and a terse program launching syntax, let a shell programmer perform the text processing often needed to match one program's output to another's input. While such common use of external languages reflects shell language's strength as a glue language, it also reflects an expectation that computation will be done through external programs rather than within the shell.

3 Shell Compilation

3.1 Applicability

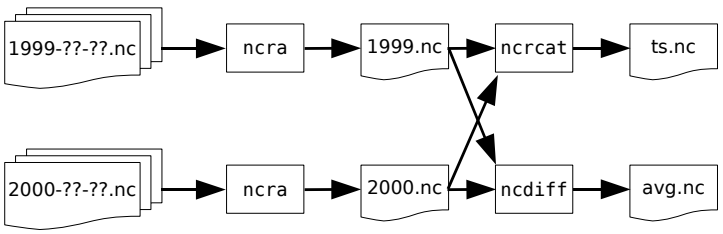
Our initial motivation was based on the widespread use of shell scripting to automate data processing and analysis. No other language provides as easy or as familiar a mechanism to reuse existing scientific tools. These scientific scripts form a large class which share three characteristics. First, these scripts are composed of programs which are well-behaved, i.e. their behavior is highly predictable, or their unpredictable characteristics are ignorable. This means that behavioral models can be written which allow a compiler to reason about program behavior, such as file access patterns. Second, these scripts are written to process data, so their intrinsic operation consists of data flowing through a chain (or web) of steps. Dataflow systems possess natural concurrency, so these scripts, given that they specify dataflow behavior, contain significant potential for parallelism. Third, they make use of a limited set of programs—common tools for data processing usually exist in each scientific community. Implementing program behavior specifications for a compiler to reason about such

scripts is therefore tractable—not all possible programs need be considered.

For concreteness, consider the following shell script, which is an example of a scientific processing script.

```
ncra 1999-??-??.nc 1999.nc     # avg over 1999
ncra 2000-??-??.nc 2000.nc     # avg over 2000
ncrcat 1999.nc 2000.nc ts.nc   # concat into time-series
ncdiff 2000.nc 1999.nc diff.nc # compute yearly difference
```

Given the availability of data stored in files according to a "YYYY-MM-DD.nc" naming convention, this script creates a yearly time-series of years 1999 and 2000, along with the average difference between the two years, using the netCDF Operators (NCO), a suite for data manipulation and analysis common in geosciences[7]. Its side-effects are insignificant or ignorable, it clearly represents a sequence of processing steps on data, and it easily accomplishes its job using a limited set of programs. We illustrate its dataflow graphically below.



While these program set limitations may seem to constrain shell compilation to irrelevance, for many application domains, these limitations are small and easily handled. The benefits we describe below, especially in performance, cannot be easily obtained otherwise. Shell compilation leverages the largest advantage of shells, namely its ease in running and connecting together pro-

grams regardless of their implementation languages.

3.2 Definition

We define shell *compilation* as the transformation from shell script representation to a dataflow program, or workflow specification. Traditional script execution, in contrast, involves no intermediate analysis of the script, consisting purely of line-by-line parsing and execution. We illustrate a possible script compilation sequence in figure 1. The process begins with lexical and syntax analyses like most compilers, but the remaining steps are different. At the end, where another compiler would produce a program that can be loaded and executed, the resultant workflow can then be optionally optimized and executed on a workflow engine.

Like other compilers, the Syntax Parser builds a parse-tree representation reflecting shell language control flow constructs. The Partial Script Evaluator uses this representation to evaluate the shell language portion of the script. In this stage, the compiler executes variable assignments, evaluates control flow constructs such as loops and conditional statements, and performs command and parameter substitution. Partial Script Evaluation is not intrinsic to shell compilation, but we believe its presence provides significant performance benefits (see section 5.2)

The Dependency Analyzer applies programmed knowledge of program behavior to determine inputs and outputs of each program invocation and to build a graph representing all data dependencies between the program commands in the script. The resulting graph is free of loops and conditionals and may

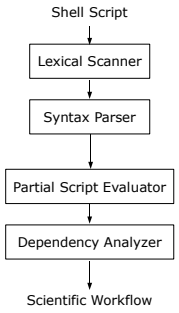


Fig. 1. Steps in shell compilation

be used for execution on a workflow engine. At this stage, the compiler may verify that the script makes no references to invalid programs and no accesses to unpermitted or invalid files. Because this step relies on models of program behavior that cannot be generated automatically, it relies on the manual effort to support each program, preventing shell compilation from being possible for all scripts.

This definition of shell compilation leverages a shell language property shared only with other wrapper languages or glue languages, that they function only to provide a high-level organization for the more interesting behavior implemented externally. Because script interpretation speed is rarely a bottleneck, optimization of only shell constructs is of limited utility. Performance is rarely affected by slow variable assignment or shell arithmetic. The scope of compilation must extend, therefore, beyond the semantics of the shell language to include program behavior. Compilation leverages models of individual program behavior to model and optimize behavior of entire scripts.

3.3 Semantic differences

In order to achieve performance benefits, some differences in execution semantics are necessary. Normal script execution with a standard shell interpreter implies in-order, sequential execution of each command line. Within the limits of this model, compilation could at best, convert a script into an executable consisting of the proper sequence of *exec* system calls. Again, no performance improvement would be seen unless interpretation were a bottleneck. We propose loosening execution correctness to require merely the correct final state rather without defining how execution should progress.

Having a correct final state means execution should terminate with a state which is indistinguishable or equivalent from the perspective of the user. For simplicity, we restrict the output state to consider only filesystem contents—correct contents written to output files. This excludes file metadata such as timestamps or file access order, because those do not affect data correctness in most applications. Intermediate files, i.e. files written during execution but are not part of the user's output, are also not considered. While intermediate files may be deterministic, their value to the user is similar to the value of files created by the the standard C *tmpfile* function, or files residing in "/tmp" or "C:\TEMP"—useful for forensic analysis, but generally disposable. Correct output files have bit-for-bit identical contents, excepting parts which are non-deterministic relative to input files (such as timestamps or process identifiers). With intermediate state now unspecified, transient disk or memory footprint or disk access patterns are no longer under direct programmer control—responsibility lies with the compiler whose resource management can be automatic if not better.

With this looser definition of correctness, programs may be launched out-of-order and in parallel. Program execution may even be migrated to other physical machines, or avoided entirely, as long as correct results are obtained, perhaps through caching. To account for this flexibility, an execution model should not guarantee system state, but rather leave it undefined, between execution start and completion. Interruption of execution under this looser definition may leave the system in an undefined state, although practical implementations should allow the filesystem to be rolled back to equivalent initial conditions. The freedom gained by loosening correctness is similar to that gained by not requiring precise interrupts in a particular microprocessor design.

3.4 Advantages

Altering the definition of correctness to be based on results rather than implementation, and using information related to program semantics confer a few key advantages.

Parallel execution is the most obvious benefit. Shell scripts may be parallelized by simply instructing the interpreter to start all programs as background processes (for example, by appending "&" to all program invocations), but such a method is unlikely to produce correct results without utilizing knowledge of program semantics and inserting additional programs or sophisticated shell constructs to function as barriers or other forms of synchronization. With knowledge of program semantics, parallelism can be exploited while still achieving correctness. This allows shell-scripted tasks to leverage parallel hardware without being ported to shared-memory or message-passing libraries

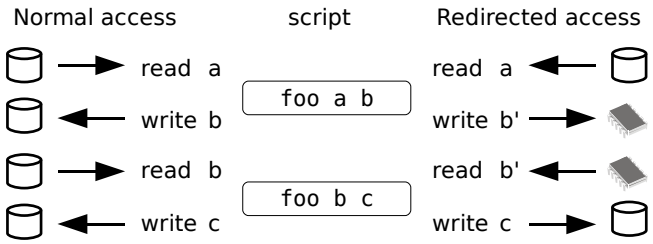


Fig. 2. Filenames can be rewritten to lessen disk access.

or being rewritten in parallel languages.

With knowledge of program semantics, a compilation system can detect the inputs and outputs of programs from their command-lines. Knowing each command-line's inputs and outputs, it can construct dependence relations between commands, and thus detect where commands can be executed in parallel while maintaining correct results. Coupled with the loose correctness guarantee, program semantic knowledge enables the system to alter filenames in program arguments and to effectively abstract filesystem access. The resulting layer of indirection provides locality and interference optimization, isolation, and portability. Optimizing locality and interference in script execution has significant benefits to overall execution performance, especially in I/O bound scripts. Some scripts show little benefit from parallelization without such optimization. Scripts, especially those whose sequential execution alone saturates disk bandwidth, can benefit from a system which can rewrite commands so that they access different disks, thus preventing disk contention from degrading performance. Freedom to redirect file access also allows the system to convert disk accesses to in-memory filesystem accesses . Given sufficient memory capacity, intermediate files need never be written to disk (see figure 2). In contrast, operating system caching, while effective, is unlikely to eliminate disk access to the same degree due to metadata consistency requirements. Files which are detected to reside remotely, e.g. via NFS, can be

prefetched into an in-memory filesystem while other parts the script are run. Similarly, files to be written remotely may be written first to memory, deferring remote transmission, and decoupling program performance from network performance problems.

Abstracting file access allows a system to isolate script execution to prevent scripts from accessing inappropriate files. Because the compilation system understands program behavior, it can flexibly allow accesses by some programs but not by others, giving each program different privileges, perhaps even depending on context (such as whether its inputs are *tainted*). This differs from the type of isolation provided by the chroot system call which provides a sanitized view of the filesystem. Executing scripts may also be prevented from interfering with each other, even if they reference identical filenames, provided that their output files are destined for different locations, e.g. in environments where servers execute scripts on behalf of users. Script programmers are also freed from manually choosing or stochastically generating non-conflicting names for temporary files.

Portability is also enhanced by abstracting file access. A compilation system can detect when scripts reference platform- or system-specific pathnames and substitute equivalent pathnames where possible. This facilitates automatically distributing execution among multiple machines, provided the machines can be given, or already have access to the files needed.

Finally, because correctness only depends on final state, a compilation system can remap, substitute, add, or eliminate commands in execution. Program names can be remapped for portability reasons. Unsafe program references may be replaced with references to safe or hardened equivalents. A single

command-line can be replaced with multiple command-lines. Suppose a script specifies a command which aggregates and processes data from many files and whose execution exceeds the available memory resources of the running system. If the compiler knows of an equivalent command sequence which has lower memory requirements, e.g. by processing the files in smaller batches and aggregating the results, it may choose to substitute the new sequence. Conversely, the compiler could detect where multiple command-lines could be combined into a single command. Each time the compiler finds a match in its table of equivalence productions, it can choose to substitute based on some measure of goodness (perhaps using a utility function). Caching can also be used to reuse results from previous executions, or to eliminate redundant command-lines within scripts.

To better understand the potentials of these command transformations, consider this following example.

```
wc /export/server1/home/user/paper.tex
```

Assuming that /export/server1 were a remote NFS mount, this command performs a word-count of /home/user/paper.tex using `wc`. Consider this alternate command which appears functionally equivalent to the user (assuming the appropriate permissions in place).

```
rsh server1 "wc /home/user/paper.tex"
```

We can express this equivalence as:

`wc /export/<machine>/<path>` $\leftrightarrow$ `rsh <machine> "wc /<path>"`

although a compiler should note other information about `wc` and `rsh`, such as

how to compute the input and output filenames and model its behavior. The goodness of the transformation can be evaluated using a corresponding utility function either statically at compile-time, or dynamically at run-time, since some factors which affect the goodness, e.g. available network bandwidth, `rsh` transaction latency, may vary significantly over time.

3.5 Challenges

There are several difficulties in implementing shell compilation as described so far. The first difficulty relates to the implementation of shell language syntax itself. Shell languages are typically documented assuming an implicit knowledge of the shell interpreter basics, and are sometimes described by comparison to other shell languages. Documentation usually focuses on usage and features rather than specification, although the POSIX specifications for command shells[2] is an important exception. Achieving full functionality in a new implementation is a significant task, even without the complex features mentioned above.

Another difficulty is the difficulty of expressing and utilizing knowledge of program behavior. Script compilation is most easily understood in the context of constitutent programs which can be modeled as performing transformations of a given number of input files and writing outputs to a given set of output files. Clearly not all programs can be easily characterized this way. Examples of programs which cause difficulty are interactive programs such as terminal programs (e.g. *telnet*) or text editors (e.g. *vi*), or shell programs themselves. In these cases, program behavior file access is difficult to predict by analyzing the command-line alone, or the desired "result" behavior of the program is not

adequately captured by a model of transforming inputs and writing outputs. Automatically determining which programs follow this model is similar to the halting problem and is therefore intractable. To support a program therefore requires hand-coded specifications or other manual assistance. Just as compilers in other programming languages consider function calls *opaque*, a shell compiler must otherwise consider program invocations to have unknown side effects that prevent re-ordering.

Fault or exception handling in out-of-order shell execution can be problematic. To parallelize, the system must dispatch a given command-line before its script-order predecessor command-lines have completed. For example, consider a script whose first command line hangs the script in an infinite loop. Parallel execution would speculate ahead, according to its model of expected behavior, and thus produce results distinguishable from the non-parallel, or serial interpreted version. In out-of-order microprocessors, similar issues from faulting instructions are handled by delaying register or memory writes by speculated instructions until their predecessors have completed (committed) normally. Upon failure, speculated instructions are squashed and their results are not persisted in register or memory state. Parallel script execution can perform similarly, by virtualizing file writes of speculated program calls, and discarding results upon parent failures. Program side effects (behavior not predicted or modeled by the shell compiler), however, would still persist, and must therefore be assumed to be negligible or effectively nonexistent.

Finally, compilation is less effective with scripts including programs that require user interaction. Because their behaviors are not predictable, their presence in scripts may make the script unsuitable for compilation, or at best require a significantly more complex specification of behavior, and be sched-

uled with constraints that may prevent execution parallelization. In multicomputer execution, it is unclear whether the scheduler should dispatch interactive programs locally, or arrange to forward user interaction between the invoking workstation and a remote node. User interaction during script execution presents a significant complication to parallel or distributed execution (and to batch execution, similarly), and should not be supported in those cases.

3.6 Execution Model

A compiled execution model implies transformation of original source code into another format which is more directly executable. Because many benefits of shell compilation depend on an execution engine which leverages the additional semantic knowledge, we describe both compilation and execution.

3.6.1 Compilation Phase

Instead of line-by-line interpretation, compilation begins by scanning for shell language constructs and generating a parse tree. After this first pass, a second pass evaluates conditional branches and unrolls loops and other built-in constructs (those typically evaluated within the shell) in order to present a simpler, cycle-free, hazard-free workflow to the execution engine. Once the script is flattened into a simpler sequence, its command-lines are matched against the set of allowed programs and their arguments are parsed to determine input and output filenames. Inputs are matched against a hash table which stores output file mappings of previous lines, and against the catalog of available source files. Outputs are inserted into the hash table, creating new mappings if entries already exist. These mappings replace the original script-

specified filenames. The hash table facilitates wildcard globbing to match files that should exist at that point in execution. We discuss file abstraction further in section 5.3. At this point, the compiler can check input and output to enforce file access privileges. Once each command is matched, remapped, and validated, the compiler builds a corresponding object and links it with parent commands according to file inputs. The resulting set of commands form a workflow.

3.6.2 Execution Phase

Once the workflow graph is complete, the system can begin executing commands. In a simple implementation, the workflow may be executed by simply invoking the commands in the order that they were parsed and expanded during compilation. Parallel execution, in contrast, requires more careful bookkeeping of commands. A basic method is to maintain four classifications of commands—*unready*, *ready*, *running*, and *complete*. Most commands enter the execution phase as unready. Once their dependencies are satisfied, they become ready. The roots of the workflow, i.e. commands with no parents, begin ready. Ready commands can be invoked whenever there are free resources, and, once invoked, become running. The running classification exists to ensure commands are not called twice (alternatively, a more sophisticated scheduler could allow replicated executions for increased data locality in multicomputer environments). Finally, once a running command successfully finishes, it becomes complete. At this point, its children (dependent commands) become ready if they have no other non-complete parents. Once all commands are complete, the script itself is considered complete. The resulting execution is topologically ordered, and approximates a dataflow computation. Output files

can be published to the owning user in one batch, or individually as they are written.

The prototype described in the following section combines both compilation execution phases, presenting an interface that mimics interpreted execution.

4 Prototype

We have implemented a prototype compilation and execution system for shell scripts in the domain of geoscience data analysis. It directly addresses the need for larger scale analysis tools by scaling existing tools and parallelizing their execution. Data movement challenges are addressed by filename virtualization and portability, which allows scripts to be safely and efficiently executed at remote data centers, all but eliminating the need to download input data. Its use of shell scripts to specify custom analyses allows use by individual scientists with little or no retraining.

4.1 Application domain

The geosciences community faces a growing imbalance between the rate of data production and the rate of data analysis and utilization. Although data are expected to be viewed, processed and analyzed many times more frequently than they are produced, the lack of scalability in these tools places severe limits on their usage. Scientists often limit resolution in simulations and observation because of data manipulation and processing challenges, rather than supercomputing limits (in simulations) or hardware technology (in observations). Because they typically apply data analysis using shell-scripted sequences of

these tools, they can directly benefit from the performance advantages of script compilation.

This domain represents an ideal opportunity to demonstrate benefits of script compilation for the following reasons. First, only a limited number of programs need support. NCO [7] is a suite of orthogonal processing primitives that are frequently used in scripted sequences that perform a variety of both simple and complex operations on geoscience data. Because of the challenges (see section 3.5) of supporting large numbers of programs, restricting support to NCO simplifies implementation.

Second, geoscience data analysis makes use of programs whose behavior is generally well-behaved, well-understood, and well-documented—programs typically read in data from files and output derived data or statistics. At the same time, NCO operators employ advanced file selection syntax, which requires a compilation system to recognize argument semantics when virtualizing file access.

Third, analysis scripts are usually non-interactive and represent implicit workflows whose dataflow and parallel execution potential can be exposed automatically using compilation. While consitutent programs may be parallelized via MPI or OpenMP, script compilation can expose higher-level parallelism that does not require reimplementation.

4.2 Design goals

The prototype is designed to meet the data analysis and reduction needs of the geoscience community while illustrating the potential benefits offered by

shell compilation. It supplements existing data service software by providing computational analysis service. As a public-facing service, it must isolate tasks for security. It must support enough shell syntax to be useful; however, implementing the full standard is unnecessary since most analysis scripts use only a small subset of available syntax. End user benefit is the goal, rather than elegance or complete syntax coverage. Scientists should be able to *interchangably* run shell scripts via either compiled execution or plain POSIX interpretation.

4.3 Features and Limitations

The implementation limits support for shell constructs to those most often encountered in geoscience scripts: for-loops, if-branching, and shell variables. While the syntax for handling environment variables is supported, the system does not maintain a distinction from shell variables and does not apply them in the environment during execution—they are not necessary for the supported program set. Shell scripts that execute using the Bourne shell are able to run unmodified in our system, but can detect the system (and handle portability issues) by testing a predefined shell variable. The system is intended for remote access, allowing clients to submit scripts for execution and to download results afterwards. By user request, we have applied a heuristic that defines *results* files as those not referenced as inputs after being written (i.e. they are *leaf* files). All other files written by script execution are marked as intermediate and can be optimized differently. In the event that non-leaf files are desired as results, they can be marked in the script via special command lines (i.e. dummy pass-through commands)—a more elegant scheme is currently unnecessary.

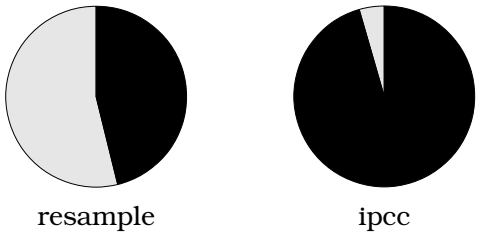


Fig. 3. Ratio of computation time(gray) to input fetch time(black) for two scripts

4.4 Experiences

Experiences with the prototype from a systems perspective are detailed in [8]; we present here the highlights that relate to shell compilation issues.

4.4.1 Performance

The original intent of the prototype system was to scale shell script performance by eliminating unnecessary data download and by exploiting latent parallelism in geoscience data reduction scripts. Our tests have shown that the transfer costs alone account for a significant fractions (≈50% in one case, ≈95% in another, assuming 30Mbit/s bandwidth) of overall end-to-end time, and are practically eliminated by sending the computation to the data source (see figure 3). Script compilation provides the sandboxing and portability that allows a data server to accept and safely execute existing desktop shell scripts. Script compliation was also shown effective at exposing parallelism without re-implementing existing analysis tools or porting analysis shell scripts to explicitly parallel formats such as workflow languages such as [9]. While a script's potential parallelism width varies over the course of its execution (sometimes numbering in the hundreds), the average width was between 10 and 19.

Figure 4 illustrates the overall performance achieved on a single SMP machine

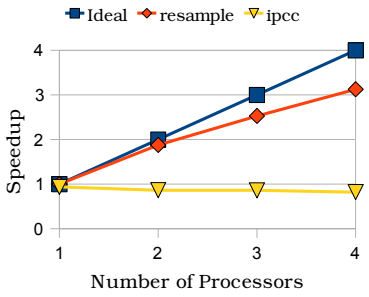


Fig. 4. Speedup vs. processor count

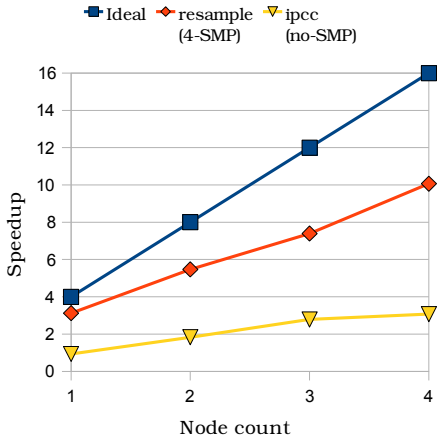


Fig. 5. Speedup vs. node count

by compiler-exposed parallelism, relative to ideal speedup (zero overhead, zero non-parallelizable components). The *resample* script was easily parallelized, giving a speedup close to ideal, but suffered from bookkeeping overhead in the 4-way case where the command completion rate was about one every 70ms. The *ipcc* script did not benefit from SMP parallelism, since its sequential execution because of its data intensity which saturates the disk manufacturer's maximum transfer rate at 65MB/s. The disk bottleneck is verified in multicomputer execution, which is plotted in figure 5. Overall, speedup was substantial, but limited by increased overhead due to the relatively large latency of network versus shared memory for synchronization. However, the *ipcc* script was able to leverage the increased disk bandwidth available from replicating files

across all machines. A more comprehensive performance analysis is available in [8].

Although disk-intensive tasks often bottleneck with increasing CPU parallelism, the system can reduce contention by transparently remapping files to an in-memory filesystem. This explicit caching behavior provided performance benefits even without parallelism (14%), but enabled a computational speedup of 3.6 compared to 2.4 when commands were allocated to all four cores of a 4-way SMP workstation.

Finally, we should note that compilation time was excessive on one benchmark, whose lenghty script of 14,640 unique command-lines exposed the inefficiencies of our parser implementation. However, we expect that most scripts can be compiled in far less time. Another benchmark of similar data intensity was compiled in roughly 0.5 seconds, owing to its simpler and more compact script about two pages long.

4.4.2 Usability

One of shell compilation's biggest advantages is its ability to bring parallel processing to non-computer scientists. Because shells are nearly universal in scientific workstations, no specialized knowledge is needed. Many users already maintain their own data analysis scripts, making porting to a shell compiling system simple. Usually, only minor modifications to path specifications are necessary. The loosened correctness guarantee and unspecified execution order have little importance in data analysis scripts, since it is more important to have correct, useful results quickly. System administrators appreciate the restricted program set in open-access systems, following the principle of least-

privilege.

5 Discussion

5.1 Syntax subset

One implementation difficulty was the choice of exactly what subset of shell syntax to support. The initial implementation supported neither variables nor control flow constructs such as branching or looping, though both were demanded and added as the potential and realized speedups became evident. Some other features that are worth supporting include variables, wildcard expansion, and I/O redirection. Variables are required in order to support looping, and their presence is essential to writing portable scripts. Wildcard expansion significantly simplifies working with multiple files (see section 5.3). I/O redirection allows programs that work with standard input and standard output to be parallelized within the more generic file-based model.

Some features of shell programming should not be supported. In a compiled system, where parts of the script may be executing on different machines, job control command semantics are not easily defined or implemented. This is a consequence of the shift in a script's meaning from a specification of execution order towards a specification of desired output. Compiled scripts should also not support all programs—exclusion of interactive programs and other programs whose behavior cannot be determined *a priori* from their command-lines is natural and discourages use of compilation for scripts which will not benefit. Although scripts containing these less deterministic programs can be supported by splitting compilation and execution at these occurrences,

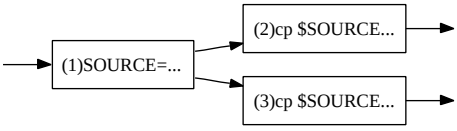
the unclear benefits do not justify the additional complexity. In open-access environments, restricting the program set is doubly important to reduce the possibilities for unknown security holes.

5.2 Partial evaluation

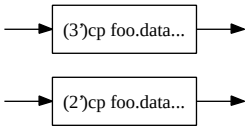
One important issue was deciding which script elements should be evaluated during compilation and which should be used to develop the workflow. Consider the following script lines, which copy a file:

```
SOURCE=foo.data  # (1) set SOURCE
cp $SOURCE bar.data  # (2) copy the file
cp $SOURCE foobar.data  # (3) copy the file again
```

Dataflow analysis easily detects that the second line depends on the first line, and could produce a workflow graph looking like:



However, we can produce an optimized graph with fewer nodes by evaluating the variable assignment in (1) during compilation and substituting in (2) and (3).



The latter graph has fewer nodes and has no non-parallel components.

Similarly, evaluating control-flow structures eliminates control hazards and enables dependencies to be evaluated over the whole script. The resulting dependency graph lacks unused script portions (dead code elimination), expands iterative portions (loop unrolling and parallelism), and can be scheduled for execution with greater freedom. Specifically, if-then-else and for-do-done constructs are prime candidates for pre-evaluation. These branch and loop evaluations rely on the assumption that their conditional or looping expressions do not depend on outputs of previous lines. In our application domain, this assumption generally holds. Eliminating control hazards and providing the entire execution graph aids a system's admission control as well. Scripts that execute too many commands, or reference invalid or unauthorized files can be rejected as a whole instead of aborting in the midst of execution.

Partial evaluation processes the "meta" portion of the script, that is, the portion that supports the main purpose of the script. A standard shell interpreter distinguishes between built-in and external commands, evaluating the former internally and executing programs specified by the latter. Compilation should process these built-ins (or a subset), but some external programs should be processed as well. These include `seq` and `printf` which are often used to generate loop ranges and format filename strings in command lines. In general, referentially transparent portions of the script are good candidates for partial evaluation, but should be restricted to extremely lightweight programs because their partial evaluation would occur during compilation and without optimization. [2]

[2] Overall, partial evaluation serves to evaluate the parts of the program which are referentially transparent, that is, they can be replaced by their values without changing the program.

While standard shell interpretation does not provide abstraction from filesystem names, our experience has identified at least two additional useful layers. First, we separate the script file layer from the physical filesystem. This provides the benefits introduced in section 3.4: locality and interference optimization, isolation, and portability. This abstraction is achieved by remapping all references to filenames in the script. This includes not only filenames in program arguments, but other arguments from which the program may compute filenames (as in the NCO tools [7]), and wildcards originally expanded against the filesystem.

The second layer of abstraction is inserted between script filenames and physical filenames in order to eliminate write-after-read and write-after-write data hazards (anti, and output dependencies). Read-after-write hazards, or true, dependencies are resolved by command ordering. The new layer, consists of *logical* names and is identical to the script layer, except that logical names must refer to files that are immutable, once written. This converts commandlines into static single assignment (SSA)[10] form, and prevents the aliasing that occurs when scripts write to the same filename more than once. The table below summarizes these levels.

File Namespace	Multiplicity	Description
Script	[1]	user-visible
Logical	[1,n]	SSA form
Physical	[0,1*]/logical	local copy

*Note that logical files may have multiple physical copies.

5.4 Files as Variables

To evaluate the language characteristics of shells, it is useful to consider files as the analog of variables in other languages. A particular filesystem is like a memory system, and the ability to mount other filesystems like mapping in other segments in "virtual memory." Like memory pages and segments, parts of the filesystem can be marked with a combination of read, write, and execute privileges. Symbolic links are like reference variables. Files do not need to be declared or fixed in size when they are created, and offer the same flexibility as variables in loosely-typed languages. Files have global scope, but are organized hierarchically in directories, just as variables in nested data structures. Although shells do not have local variables(files) with scope-limited lifetime as other languages, the concept of a "working directory" simplifies naming similarly. Unlike other languages, however, shells have a separate search path for referencing executable variables(files) whereas other languages unify the namespace of data and code.

Considering files in the same vein as variables, may they be garbage collected?

Shells do not implement garbage collection—although the reclaiming of unlinked inodes is similar, it occurs in repair contexts. In considering shell scripts as programs, garbage collection would mean automatic deletion of unreachable (dead) files, but since files are always reachable unless overwritten or explicitly deleted, it is unclear how to identify which files are effectively dead. Standard garbage collection techniques like mark-and-sweep do not seem effective, since all files are reachable through the filesystem hierarchy. We can infer that files only written-to but not read-from in the script are "live," and desired as useful output, and the contrapositive—that files written-to and read-from are dead—is true often enough to be a good heuristic, though not always the case. Our prototype uses this heuristic since it holds true for most scripts of our domain.

Considering platform portability in light of this file-based model, shell language experiences similar portability problems as other languages. Just as standard system libraries may vary in location and cause portability problems in C programs, filesystem paths and layout vary and may cause similar problems for shell programmers.

One feature in shell language which has no equivalent is wildcard expansion. For example, `cp * ..` copies all files in the current directory to the parent directory. We are not aware of the equivalent concept, copying all variables in local scope to another (e.g. parent) scope in a non-shell language, although some languages can achieve similar behavior using introspection or a variable representing the current context.

6 Related Work

Work in shell compilation has been relatively scarce, favoring instead development of new, more programmable shells, such as `rc` for Plan 9 [11], or other languages that provide shell-like functionality within a more general-purpose syntax [12]. Scsh[6] provides a good description of the weakness of the Bourne shell as a programming language, and proposes the addition of program-running and program-connecting capabilities to a general-purpose programming language. The `make` system [13] can accomplish a similar performance capability in more modern parallel [14] or distributed [15] implementations, but at the cost of a vastly different syntax that forces user specification of command inter-dependencies.

Grid workflow research has explored similar issues in parallel and distributed execution. GridAnt [16] provides similar abstraction in its workflow specification and offers similar flexibility in scheduling and distributing work. GRID superscalar[17] and its successor COMP superscalar similarly convert sequential applications to dependency-aware workflows, but require use of a C-like syntax uses function signatures to signify dependence relations. All grid workflow systems, including GridAnt, require explicit user specification of dependencies, which script compilation avoids.

7 Conclusion

Shell languages are special programming languages which contain many difficulties and lack many features in comparison to other languages. Their specialized purpose of facilitating and automating execution of other programs

has led to design choices that make their compilation useless or practically impossible in the general case. However, by restricting the program choice and loosening the correctness model, compilation becomes possible, and the resulting execution flexibility enables significant advantages to security, portability, and most of all, performance. Compilation exploits a characteristic shared by a significant class of shell scripts—that they define data flows in terms of ordered command-line sequences processing files. Our prototype, implemented for this class of scripts in the domain of geoscience data reduction, illustrates the significant performance potential as well as the issues that prevent shell compilation from being universally applicable. Results show that shell compilation may be used as an alternative method of leveraging parallel hardware—by parallelizing sequences of applications rather than rewriting the applications themselves. With the increasing importance of parallelism as a means for achieving high performance, shell compilation merits further study since it provides performance benefits to applications without requiring deep study of their internal algorithms.

References

[1] A. Scherr, AN ANALYSIS OF TIME-SHARED COMPUTER SYSTEMS, Tech. Rep. MIT-LCS-TR-018, Massachusetts Institute of Technology (June 1965).

[2] IEEE, 1003.1-2004 INT/2004 Edition, IEEE Standard Interpretations of IEEE Standard Portable Operating System Interface for Computer Environments (IEEE Std 1003.1-2004), IEEE, New York, NY, USA, 2004.

[3] L. Pouzin, The origin of the shell (2000).

URL http://multicians.org/shell.html

[4] J. W. Davidson, J. V. Gresh, Cint: a risc interpreter for the c programming language, in: SIGPLAN '87: Papers of the Symposium on Interpreters and interpretive techniques, ACM, New York, NY, USA, 1987, pp. 189–198.

[5] R. Brun, F. Rademakers, et al., ROOT-An Object Oriented Data Analysis Framework, Proceedings AIHENP 96 (1997) 81–86.

[6] O. Shivers, A scheme shell, Tech. rep., Massachusetts Institute of Technology, Cambridge, MA, USA (1994).

[7] C. S. Zender, Analysis of self-describing gridded geoscience data with netCDF Operators (NCO), Environ. Modell. Softw. 23 (10) (2008) 1338–1342.

[8] D. L. Wang, C. S. Zender, S. F. Jenks, Efficient clustered remote data analysis workflows using SWAMP (2008).

[9] T. Oinn, M. Greenwood, M. Addis, M. N. Alpdemir, J. Ferris, K. Glover, C. Goble, A. Goderis, D. Hull, D. Marvin, P. L. P. Lord, M. R. Pocock, M. Senger, R. Stevens, A. Wipat, C. Wroe, Taverna: lessons in creating a workflow environment for the life sciences, Concurrency and Computation: Practice and Experience 18 (10) (2006) 1067–1100.
URL http://dx.doi.org/10.1002/cpe.993

[10] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, F. K. Zadeck, Efficiently computing static single assignment form and the control dependence graph, ACM Transactions on Programming Languages and Systems 13 (4) (1991) 451–490.
URL http://doi.acm.org/10.1145/115372.115320

[11] T. Duff, RC—a shell for plan 9 and UNIX, W. B. Saunders Company, Philadelphia, PA, USA, 1990, pp. 283–296.

[12] C. W. Fraser, D. R. Hanson, A high-level programming and command language, in: SIGPLAN '83: Proceedings of the 1983 ACM SIGPLAN symposium on Programming language issues in software systems, ACM, New York, NY, USA, 1983, pp. 212–219.

[13] S. C. Johnson, Yacc: Yet another compiler compiler, in: UNIX Programmer's Manual, Vol. 2, Holt, Rinehart, and Winston, New York, NY, USA, 1979, pp. 353–387.
URL citeseer.ist.psu.edu/johnson79yacc.html

[14] E. H. Baalbergen, Design and implementation of parallel make, Computing Systems 1 (2) (1988) 135–158.

[15] A. Lih, E. Zadok, PGMAKE: A portable distributed make system (1994).

[16] K. Amin, G. von Laszewski, M. Hategan, N. J. Zaluzec, S. Hampton, A. Rossi, Gridant: A client-controllable grid workflow system, in: HICSS '04: Proceedings of the Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) - Track 7, IEEE Computer Society, Washington, DC, USA, 2004, p. 70210.3.

[17] R. Badia, J. Labarta, R. Sirvent, J. Pérez, J. Cela, R. Grima, Programming Grid Applications with GRID Superscalar, Journal of Grid Computing 1 (2) (2003) 151–170.