# Optimizing Shell Scripting Languages

Emery D. Berger

*Department of Computer Science*
*University of Massachusetts*
*Amherst, MA 01003*
emery@cs.umass.edu

## Abstract

We present an optimizing compiler that dramatically improves the performance of shell script languages. Our prototype system, called Shark, performs a number of domain-specific optimizations. We show that, subject to some constraints, we can treat file I/O in shell languages like variables in regular programming languages. This insight allows us to leverage traditional compiler analyses to eliminate temporary files, transform file-based I/O to pipes, and extract coarse-grained parallelism. We show how we can further improve performance by eliminating process creation overhead, which can dominate the runtime of shell scripts. By modifying just a few lines of code, we can convert external commands like `grep` into dynamic libraries. Shark loads these libraries just once so that subsequent invocations execute with the speed of function calls, and then unloads them when they are no longer needed. We present results with our prototype demonstrating substantial speedups over the `bash` shell.

## 1 Introduction

Shell script languages like `sh`, `csh`, and `bash` [2, 13] are popular glue languages. Because most UNIX commands allow their input and output to be redirected to files or pipes, programmers can use shell scripts to quickly build programs that use other UNIX commands as components [10, 19]. Shell scripts are in wide use for a variety of tasks, including system administration and source code configuration [5], because of their relative simplicity, familiarity and ubiquity. Unfortunately, shell languages are notoriously slow.

There are many sources for the poor performance of shell languages. First, shell languages typically interpret shell scripts and pay the associated overhead, including repeated conversions to and from strings. Second, shell interpreters do not optimize their input scripts beyond translating them into an intermediate form. Finally, shell languages create a new process to run most external commands. The overhead of repeated process invocation dominates the runtime of most shell scripts [4].

In this paper, we demonstrate a prototype optimizing compiler called Shark that dramatically improves the performance of shell scripts. This paper makes the following contributions:

**Domain-Specific Optimization of Shell Languages:** We demonstrate that, with certain restrictions, we can treat file I/O operations in shell scripts like variable assignment in traditional programming languages. We then perform a modified dependence analysis that yields a number of domain-specific optimizations, including the elimination of temporary files and the conversion of file I/O to pipes. These optimizations reduce the use of the filesystem and improve application locality. We exploit the same dependence analysis to extract coarse-grained parallelism, allowing Shark to spawn processes in parallel in order to take advantage of multiple processors or processors with simultaneous multithreading capabilities [8].

**Effective Use of Dynamic Libraries:** While the above optimizations are effective for many programs, repeated process invocation is the performance bottleneck for some shell scripts. The second contribution of this paper is to show how to eliminate this bottleneck by converting off-the-shelf programs like `grep` into dynamic libraries[1]. This optimization alone can dramatically reduce the runtime of some scripts. The conversion to dynamic libraries is surprisingly straightforward, requiring the modification and addition of just a few lines of code. We use the Reap memory manager to eliminate the potential for memory leaks caused by these external commands [1]. Shark also minimizes the memory footprint caused by incorporating these libraries by loading them only when needed and unloading them when done.

[1]These libraries are also referred to as *DLLs* or *shared objects*.

## 1.1 Outline

The remainder of this paper is organized as follows. We discuss related work on shell languages and optimization in Section 2. We describe the Shark language in detail in Section 3. We present the domain-specific optimizations that Shark implements in Section 4. Section 5 presents the key challenges to optimization posed by shell languages. We describe our optimization algorithms in detail in Section 6. We describe the process of converting ordinary applications to dynamic libraries in Section 7. We describe our experimental methodology and present experimental results in Section 8. We describe future directions and extensions to Shark in Section 9 and conclude in Section 10.

# 2 Related Work

## 2.1 Shells

While the original UNIX shell language was written by Ken Thompson, it was eventually supplanted by the Bourne shell, by Steve Bourne at Bell Labs [2, 18]. Subsequent shell languages include the Korn shell, Bill Joy's C shell, the "Bourne-again" shell (`bash`), and the `Rc` shell from Plan 9 by Duff [6, 13]. POSIX standard 1003.2-1992 defines a standard set of requirements for shell languages, and includes such features as file input-output redirection, pipelines, expansion of wildcard characters, and command substitution. Newer shell languages generally provide more features rather than improved performance. These features include richer iteration constructs, cleaner syntax, and improved command-line editing.

## 2.2 Script Languages

Owing to the inefficiencies and relative linguistic paucity of shell languages, a number of other scripting languages have arisen in recent years. The most prominent of these are Perl, Python and Tcl [15, 17, 21]. Perl and Python in particular provide improved performance over shell script languages. They achieve these improvements primarily by building in common functionality like regular-expression pattern matching and optimizing those patterns, rather than by performing the domain-specific optimizations we describe.

## 2.3 Dynamic Libraries

Most modern operating systems provide support for dynamic libraries. which allow code to be loaded into a running program. Using dynamic libraries has a number of advantages, including beneficial effects on both system memory usage and software engineering. Smaragdakis includes a number of examples as well as a novel use of dynamic libraries to support layered software development [20].

A recent version of the Korn shell, ksh93, allows external C functions to be dynamically loaded into the shell via the `builtin` command [12]. The authors observe dramatic speedups for the execution of short-lived commands (up to 50x), and also note the loss of process isolation that comes with dynamically loading external commands. Unlike Shark, ksh93 allows external commands to access certain shell internals, adding flexibility but increasing the risk that an external command will crash the shell. Ksh93 places the burden on the programmer to decide when it is appropriate to load them, and does not provide a facility to unload them. By contrast, Shark automatically loads libraries at their first use and unloads them after their last use. We discuss other limitations of ksh93's use of libraries in Section 7.

## 2.4 Shell Optimization

We are aware of only one previous approach to automatically improving the performance of shell scripts, a commercial "shell compiler" called `ccsh` from Comeau and Associates [3]. `Ccsh` compiles Bourne shell scripts into C programs that it subsequently compiles by invoking the C compiler. The company claims performance improvements of from 2x to 10x, but these are primarily due to the incorporation of many common utilities (like `grep` and `cat`) as built-in functions[2]. Shark exploits dynamically-loaded versions of these utilities to provide similar performance gains, but differs from `ccsh` by implementing a number of domain-specific optimizations. Shark also does not require a separate compilation phase.

## 2.5 Programming Language Optimizations

By recognizing that filesystem I/O is analogous to variable assignment in traditional programming languages, we are able to take advantage of traditional compiler optimizations. In particular, Shark performs an adapted dependence analysis. The literature on the topic of optimizations is vast; Muchnick's textbook includes extensive discussion and citations [16].

# 3 The Shark Language

Shark implements a subset of the Bourne shell language that includes a number of its key features, including input-output redirection, pipes, conditionals, and command substitution [2]. Any Shark script can be executed by the Bourne shell or any upwardly-compatible shell. Unlike most shell languages, the grammar for the Shark language is context-free, allowing us to use `yacc` and `lex` for its parser and lexer [14].

[2]The `ccsh` compiler was unavailable to us for evaluation at the time of writing.

In order to keep the language context-free and to simplify compilation, we omit a number of features (like `alias`), and restricted certain features, including iteration constructs and the expansion of variables. We also limited the use of other constructs. For instance, Shark requires that the `source` command, which imports other shell scripts, always be unconditionally executed. We deliberately do not include any constructs related to job control, as Shark is designed to extract all available parallelism. We leave the addition of other shell features to Shark as future work. While Shark implements only a subset of standard shell programming languages, we believe that this subset constitutes a useful core.

Shark internally implements only those commands that affect our optimizations, including file concatenation (`cat`), file deletion (`rm`) and symbolic link creation (`ln`). We recognize these primarily in order to track their effects on the file system. We show in Section 4 how these affect our optimizations. All other commands are executed as dynamically-loaded functions if these are available. Otherwise, Shark invokes a process to execute each command. We discuss the transformation of existing UNIX utilities into dynamic libraries in Section 7.

Execution of a Shark script is performed just as with other shell languages by invoking `shark` followed by the script name. Shark loads the entire program, performs a whole program analysis to effect its domain-specific optimizations, and executes the script. Shark maintains the transformed program in memory and does not create object code files.

## 4 Domain-Specific Optimizations

When Shark loads a script, it executes a whole-program analysis to drive a number of optimizations. We divide these optimizations into four categories: filesystem optimization, pipelining, parallelization, and command invocation optimization. Table 1 summarizes these optimizations and their goals. Below we discuss each in detail.

### 4.1 Filesystem Optimization

We implement one key optimization over access to the filesystem, which is the elimination of unnecessary file creation. We also remove all commands that have no side effect except the creation of unnecessary files. Figure 1 depicts an example of the effect of this optimization, eliminating a number of useless commands.

### 4.2 Pipeline Optimization

Whenever possible, we convert file I/O to pipes. We replace files that are used exactly once as input and output to two commands by a pipe connecting the two commands. If the file is not deleted before the termination of the script, we add a call to the `tee` command, which writes its input to both a file and to standard output.

```
cat foo | grep abc > tmpfoo;
cat foo | grep xyz > foo2;
rm tmpfoo;
```

Figure 1: Elimination of unnecessary file creation. All of the commands show in grey can be removed, because their only side effect is a file that the script deletes before termination (`tmpfoo`).

```
cat foo > tmpFile;
grep abc < tmpFile > outFile1;
cat foo > persistent;
grep xyz < persistent > outFile2;
rm tmpFile;
```

(a) The original program.

```
cat foo | grep abc > outFile1;
cat foo | tee persistent
  | grep xyz > outFile2;
```

(b) The transformed program.

Figure 2: Pipeline optimization. We can remove the file `tmpFile` altogether, but must use `tee` to output the file `persistent`.

Using pipes instead of or in addition to file I/O can improve performance in several ways. First, eliminating file I/O is almost always desirable because of the high cost of disk accesses. Second, using pipes improves locality. Because pipes are smaller than the L1 cache on most processors, access to the pipe buffer will incur few L1 cache misses and no L2 misses. By contrast, sequentially writing and then reading a file will result in numerous cache misses in both the L1 and L2 caches. Finally, using pipes can improve the throughput of a script executing on a multiprocessor system because pipes allow producer and consumer processes to operate concurrently.

The example in Figure 2 illustrates both forms of the pipeline optimization. We do not need to create the file `tmpFile` because it is deleted before the script terminates. However, we must send output to the file `persistent`, so we `tee` output to it.

### 4.3 Parallelization

The pipeline optimization described above increases the concurrency of Shark scripts. Shark also parallelizes processes that have no command or data dependencies. Such processes are safe to spawn in parallel. In Figure 3, we show an example of processes that are independent and

| Shark optimizations | | |
|---|---|---|
| *Optimization* | *Definition* | *Goal* |
| **Filesystem** | Eliminate creation of unnecessary files | Reduce I/O, unnecessary computations |
| **Pipelining** | Convert file I/O to pipes | Reduce I/O, improve locality & concurrency |
| **Parallelization** | Schedule independent commands in parallel | Increase concurrency |
| **Command invocation** | Transform program using domain-specific knowledge | Optimize command usage |

Table 1: The optimizations that Shark implements, along with their goals for improving performance.

```
grep abc < foo > outFile1;
grep xyz < foo > outFile2;
grep qwe < outFile2 > outFile3;
```

(a) The original program.

```
(grep abc < foo > outFile1 ||
 grep xyz < foo > outFile2);
grep qwe < outFile2 > outFile3;
```

(b) The transformed program.

Figure 3: Parallelization. We identify independent command chains and execute them in parallel, separating invocations with "||" rather than ";". Here Shark identifies that the first two commands can proceed in parallel. Note that we disable pipeline optimization for this example.

```
cat foo | cat | grep abc > bar;
```

(a) The original program.

```
(grep abc < foo) > bar;
```

(b) The transformed program.

Figure 4: Command invocation optimization. Here we perform `cat`-elimination to change calls to `cat` into input redirection or to eliminate them altogether.

thus can be parallelized. To indicate that two processes execute in parallel, we separate them with "||" rather than ";". This construct is not actually exposed to a Shark programmer but used here for notational convenience. Because `outFile1` and `outFile2` are independent, the two commands creating them can be executed in parallel. However, because the creation of `outFile3` depends on `outFile2`, that command must follow the other two.

### 4.4 Command Invocation

Shark also can exploit the semantics of certain commands to transform them into more efficient forms. We have implemented one straightforward optimization called `cat`-elimination. The `cat` command copies its input to standard output and concatenates all files in its argument list. When `cat` is called with one argument, we replace it by input redirection. When it is called without any arguments, we can eliminate it completely. The example in Figure 4 shows both cases.

## 5 Challenges to Optimization

There are many complications to implementing the optimizations we describe above. Unknown commands may have arbitrary effects on the filesystem, making the reordering or elimination of file I/O unsafe. The creation and removal of symbolic or hard links adds the familiar problem of *aliasing*. UNIX systems also contain special filesystems like `/dev` and `/proc` whose contents do not behave like ordinary files [11]. We handle these problems with a variety of approaches.

First, we provide Shark with a list of *well-behaved* commands. Well-behaved commands have no impact on the filesystem except for reading from or writing to files mentioned in their argument lists. Examples of trusted commands include `grep`, `cat`, and `diff`.

In general, we cannot consider interpreters like `awk` or `perl` to be well-behaved. These programs, by executing their argument scripts, may arbitrarily and unpredictably modify the filesystem. Such behavior represents a pathological case. Because we expect most programs used in shell scripts to be well-behaved, we place both `awk` and `perl` on a *usually well-behaved* list.

As mentioned, certain special files also complicate optimization. Our interim approach is to conservatively treat any absolute pathname as a special file and not optimize accesses to it. In a future version of Shark, we plan to allow programmers to *annotate* both invocations of usually well-behaved commands that are unsafe and special files. We describe these and other proposed extensions in Section 10.

Finally, we restrict the creation and deletion of both files and links to only refer to constants. With this restriction, we can statically examine calls to `rm` and `ln` to maintain the complete alias information required to safely perform optimizations. We can also (but currently do not) gather information before execution about the status of symbolic and hard links to files mentioned in the script.

## 6 Optimization Algorithm

In this section, we present in detail how Shark performs its optimizations. We rely on a dependence analysis to perform three of the optimizations: filesystem optimization, pipelining, and parallelization. This analysis is a whole-program optimization that is analogous to many traditional compiler optimizations. In essence, we consider commands and I/O in the same way that ordinary compilers treat functions, variables, and assignment, but with the restrictions described in Section 5.

We build an abstract syntax tree during the parse phase and use this tree as the only intermediate data representation. Rather than performing "tree surgery", which can be faster but is also error-prone, each step of the algorithm creates a new copy of the tree. Despite this apparent inefficiency and the many passes we use in the algorithm, we have found that analysis time is fast and accounts for a small percentage of Shark's runtime of a shell script.

### 6.1 Cat-elimination

We perform our one command invocation optimization, `cat`-elimination, by walking through the abstract syntax tree looking for calls to `cat`. We replace pipelines (e.g., `cat foo | baz`) with input operations (e.g., `baz < foo`), and remove calls to `cat` with no arguments.

### 6.2 Temporary File Identification

We then walk through the abstract syntax tree to gather a list of temporary files that we can eliminate in later phases. We create a list of these temporary files by iterating backwards through the script, adding every file argument to `rm`, and removing files that are created by output redirection. Recall that in a subsequent pipelining optimization phase, we eliminate these temporary files.

### 6.3 Graph Transformation

Next, we canonicalize the program into a sequence of "IOScripts", which group related commands and their I/O in preparation for converting the program into a graph. We do this first by transforming all pipelines in the original script into commands that write to or read from new temporary files, which we also add to the temporary file list. As an example, this changes `foo | bar` into `foo > tmp1; bar < tmp1`. We then replace all input or output operations by combined I/O commands, or IOScripts.

We now transform this canonicalized abstract syntax tree into a dependency graph [16]. This directed acyclic graph contains only data-dependent edges: in this initial formulation of our optimization algorithm, we do not optimize over control dependencies. Each vertex represents a filename and each directed edge corresponds to dependencies, labeled with a sequence of commands. Each filename also has an associated "version number" that corresponds to the number of writes applied to the file. We allow multiple instances of vertices (files) with the same version number to appear in the graph.

To represent the data dependencies between files and commands, we visit each IOScript node in the tree and add two vertices connected by a directed edge to the graph connecting input files to commands, and commands to output files. We also add directed edges between successive versions of files, that is, from version $n$ to version $n+1$. These edges preserve the serial dependencies between file writes and ensure correct output ordering.

### 6.4 Unnecessary file elimination

Next, we perform the elimination of unnecessary file creation. This optimization pass is essentially the same as the dead-code elimination phase employed by most modern compilers [9]. This pass implements the filesystem optimization described in Section 4.1. Notice, however, that this does not eliminate temporary files that communicate data from one command to another (as in `foo > tmp; bar < tmp`). We convert these into pipes when possible in the next pass.

### 6.5 Pipeline optimization

The transformation of file I/O to pipes is only safe for temporary files that are both created and deleted during script execution. We identify as *intermediate files* those that have an in-degree and out-degree of one. By removing all intermediate file vertices that also appear on the temporary file list, we replace all unnecessary file I/O by pipes. We add a call to `tee`, as described in Section 4.2, to preserve intermediate files that are not temporaries.

### 6.6 Parallelization

Finally, we transform the optimized graph back into an abstract syntax tree corresponding to an executable schedule. We gather all of the roots of the graph (vertices with no incoming edges) and process the graph from each root in breadth-first order. We assign a breadth-first level number to each vertex corresponding to its maximum distance from a root. All vertices with the same level can be safely executed in parallel. We then build a tree consisting of sequences of parallel spawns where all vertices of a given level execute before all those of the next level.

The resulting tree implements the parallelization optimization, completing Shark's optimization algorithm. In the next section, we describe our other approach to reducing script execution time, the replacement of process creation by dynamic library calls.

## 7 Using Dynamic Libraries

Because the overhead of process invocation can be quite high, Shark strives to eliminate this overhead by taking advantage of dynamic libraries. Using dynamic libraries instead of invoking a new process for each command can substantially reduce the runtime of shell scripts. One drawback of dynamic libraries is that they require position-independent code, which used to impose a slight runtime penalty. However, latest versions of gcc (3.2 and above) *always* generate position-independent code for x86 architectures, so using dynamic libraries is never disadvantageous on this important class of systems.

### 7.1 Ksh93's Approach

The only other shell language we are aware of that allows dynamic loading of commands is the most recent version of the Korn shell, ksh93 [12]. However, as we note in Section 2, their approach suffers from numerous limitations.

**Manual loading:** First, ksh93 requires the programmer to invoke a command (`builtin`) to replace an external command with a dynamic library, and does not provide any means to unload the dynamic library. The decision to load a library is therefore irrevocable. The programmer must weigh the potential performance gains against the increased footprint of the shell interpreter.

**Special libraries:** Second, the developer of a dynamically-loadable command for ksh93 must use header files and libraries from the Korn Shell SDK. This means that any library used by the external command must also be recompiled to use the SDK. This process could become quite burdensome.

**No dynamic memory allocation:** Finally, the authors of ksh93 recommend that developers avoid the use of `malloc` to prevent memory leaks. These leaks can arise when a command does not free all of its allocated memory or is interrupted (i.e., via `SIGINT`) before it can do so. We believe that this limitation makes it impractical for most developers to incorporate their code as dynamic libraries.

### 7.2 Shark's Approach

Shark simplifies the incorporation of external commands by eliminating most of these limitations. These commands need just a handful of changes, which we outline below. Shark does not require the use of special header files or libraries. Most importantly, we allow external commands to use `malloc`.

We avoid potential memory leaks by employing our Reap memory manager [1]. Reap allows the creation and destruction of separate memory areas ("reaps") which support both the allocation and deallocation of individual objects. Shark assigns a separate reap to each external command. When Shark unloads the associated dynamic library, it also destroys its associated reap, thus reclaiming any memory allocated by the command. We redefine `malloc` and `free` and the associated memory management functions in terms of the analogous calls provided by the Reap API (`reapalloc`, etc.).

There are two other important changes required in order to adapt existing programs. First, we cannot allow calls to `exit()` in a dynamic library to actually exit the script: they must instead propagate their return value. We first change any calls to `exit()` in the `main` routine to `return`. We then take advantage of the ANSI C non-local goto API (`setjmp()` and `longjmp()`) to cause calls to `exit()` to jump to the end of the `main` routine. We wrap the entire main routine except for variable assignment in a conditional that calls `setjmp()` using a global jump buffer. We then redefine `exit()` to perform a `longjmp()` to that buffer. For C++ programs, we would have to redefine `exit` to throw an exception because C non-local gotos do not respect C++ semantics.

The other important change is that we need to reinitialize static variables. These variables are initialized when the dynamic library is first loaded and retain their values across invocations. We often need to reinitialize these statics back to their original values. The key statics used by GNU programs are from the `getopt` package which handles argument parsing. We reinitialize `optind` and `optarg` to 0 and NULL, respectively.

### 7.3 Limitations of Dynamic Libraries

While dynamic libraries can substantially reduce the cost of process invocation, as we show in Section 8, there are some inherent limitations caused by using them instead of invoking processes.

First, it will not always be straightforward to convert off-the-shelf programs into dynamic libraries. There may be global state that needs to be reinitialized but that is not directly accessible (e.g., file-scoped variables in external libraries). While we have not yet run into this problem, it remains a possibility.

There is also a fundamental tension between increasing concurrency and using dynamic libraries. Most programs are not re-entrant and so it is not safe to execute simultaneous calls to these libraries in different threads. We currently handle this case dynamically: we only allow one instance of a dynamic library to execute at a time. This approach allows multiple dynamic libraries to run in parallel threads, but only if they are all different commands. Otherwise, we pay the price of process invocation in order to exploit parallelism.

Finally, one unavoidable limitation of using dynamic libraries is that we forfeit the protection provided by sepa-

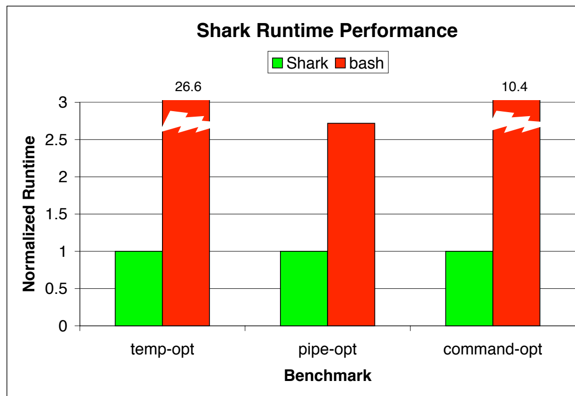


Figure 5: Runtime, normalized to Shark, for microbenchmarks demonstrating the effects of three optimizations (smaller is better). Bash takes from 2 to 26 times longer to execute these scripts.

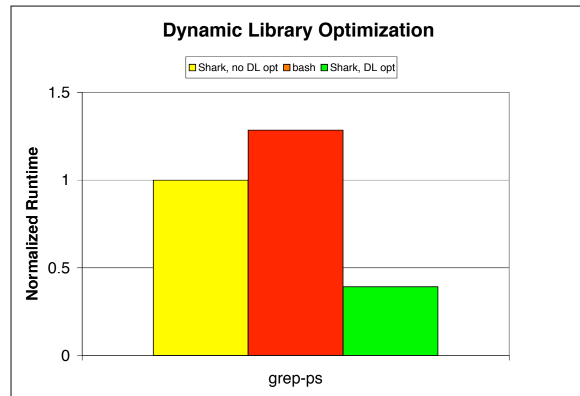


Figure 6: Runtime, normalized to Shark, demonstrating the effect of Shark's dynamic library optimization (smaller is better). Dynamic library optimization yields a three-fold performance improvement.

rate processes. A bug in a dynamic library can corrupt the entire shell. On the other hand, most of the utilities that programmers use, like `grep` and `diff`, are mature pieces of code that have been extensively tested. We feel that the performance tradeoff outweighs the risks for this domain.

# 8 Experimental Results

In this section, we compare the execution times of Shark and the Bourne shell on a number of shell programs. To our knowledge, there does not exist a set of shell language benchmarks. We present preliminary results on a handful of microbenchmarks that we wrote ourselves.

All runtimes are the best of three runs at real-time priority after one warm-up run. We compiled all programs with the GNU C compiler version 3.2, and ran our experimens on a 2.8 GHz Pentium III system with 1GB of RAM under Windows XP. Because process creation in Windows is relatively expensive compared to UNIX systems, we plan to run experiments on Linux and Solaris systems (both uni- and multiprocessor) for the final version of this paper.

## 8.1 Runtime Performance

Because Shark is a subset of the Bourne shell, we directly compare the execution times of Shark to execution of the same scripts by `bash`. We use the latest available version of `bash`, version 2.05b.0(7). We report here the impact of code optimization and the use of dynamic libraries. Because these two are complementary approaches, we measure these separately in order to isolate the effect of the optimizations.

### 8.1.1 Code Optimizations

Figure 5 demonstrates the effectiveness of three optimizations: temporary-file elimination, pipeline-conversion, and command-invocation optimization (cat-elimination). Both temporary-file elimination and cat-elimination achieve dramatic improvements in performance compared to bash (running 10x to 26x faster). Pipeline-conversion achieves gains which are more modest but still substantial, executing 2.5 times faster than bash.

### 8.1.2 Dynamic Libraries

In order to measure the effectiveness of Shark's automatic use of dynamic libraries, we created a simple text-extraction benchmark that executes `grep` repeatedly on a PostScript input file. Figure 6 shows the results normalized to the runtime of bash. Without using the dynamic library optimization, Shark runs around 30% faster than bash. By using a dynamic library version of `grep`, Shark runs more than 3 times faster than bash.

# 9 Future Work

The Shark system is currently a prototype. We plan to incorporate more shell language features, focusing on the most commonly-used ones that do not preclude our optimizations.

We anticipate adding further optimizations to Shark. We plan to incorporate hints to the filesystem via `fcntl` to indicate that files used by shell scripts should be optimized for sequential access. While we currently do not take advantage of the ability to resize pipes on systems that provide this facility (e.g., Windows), we plan to explore this possibility. We expect increasing the size of pipes above 4K

to improve throughput. We also plan to implement some measure of concurrency control to prevent the parallelization optimization from generating too many processes for the system to handle efficiently.

Finally, we need a way to convey information to the shell language about the programs we plan to use. We can exploit information about the kinds of files generated by programs and how programs use their arguments in order to achieve better performance. We are currently considering an annotation language that resembles Broadway [7]. Broadway provides a simple yet rich means of describing the effects of functions on their inputs, outputs, and global state. We believe that adapting this framework would make it simple for developers to provide information to Shark about their application's behavior on their arguments and the filesystem.

## 10 Conclusions

In this paper, we have described Shark, an optimizing compiler for shell languages. We believe we are the first to show how to adapt traditional compiler analysis to this domain, subject to certain restrictions. Using this analysis, we can perform a variety of domain-specific optimizations that enhance performance. These optimizations include the elimination of unnecessary commands and file creation, conversion of file I/O to pipelines, parallelization, and the transformation of certain commands to more efficient forms. We also demonstrate a technique for converting existing code to dynamic libraries, allowing further performance gains.

The traditional focus of recent work on shell languages has been the incorporation of new features. We adopt a different approach, developing a spare shell language that is simultaneously useful and amenable to effective optimization. Our preliminary results show that we can optimize shell languages and achieve substantial improvements in performance.

## 11 Acknowledgments

Thanks to both Sam Guyer and Scott Kaplan for useful discussions about compiler analysis applied to the unusual domain of shell languages.

## References

[1] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Reconsidering custom memory allocation. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA) 2002*, Seattle, Washington, November 2002.

[2] S. R. Bourne. UNIX time-sharing system: The UNIX shell. *Bell System Technical Journal*, 57(6):1971–1990, 1978.

[3] Comeau Computing, Inc. CCsh, the Bourne shell compiler. `www.comeaucomputing.com/faqs/ccshdoc.html`.

[4] Jim Coplien, Alan Robertson, and Gregg Wonderly. UNIX shell patterns. In *Pattern Languages of Programs (PLoP) Conference 1996*, 1996.

[5] David J. MacKenzie and Akim Demaille. Autoconf. `www.gnu.org/software/autoconf/autoconf.html`.

[6] Tom Duff. Rc - a shell for Plan 9 and UNIX systems.

[7] Samuel Z. Guyer and Calvin Lin. An annotation language for optimizing software libraries. In *Domain-Specific Languages*, pages 39–52, 1999.

[8] Intel Corporation. Hyper-Threading Technology. `developer.intel.com/technology/hyperthread/`.

[9] Kenneth W. Kennedy. A survey of data flow analysis techniques. *Program Flow Analysis: Theory and Applications*, pages 5–54, 1981.

[10] Brian W. Kernighan and Rob Pike. *The UNIX Programming Environment*. Prentice Hall, 1984.

[11] T. J. Killian. Processes as files. In *Proceedings of the Summer 1984 USENIX Conference*, pages 203–207, Salt Lake City, Utah, USA, 1984.

[12] David Korn, Charles Northrup, and Jeffrey Korn. The new KornShell – ksh93. *The Linux Journal*, 27, 1996.

[13] David G. Korn. ksh: An extensible high level language. In *Proceedings of the USENIX Very High Level Languages Symposium*, pages 129–146, 1994.

[14] J. Levine, T. Mason, and D. Brown. *lex & yacc*. O'Reilly & Associates, 2nd edition, 1992.

[15] M. Lutz. *Programming Python*. O'Reilly & Associates, 1996.

[16] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, 1st edition, 1997.

[17] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison Wesley, 1994.

[18] D. M. Ritchie. The evolution of the UNIX time-sharing system. *The Bell System Technical Journal*, 63, 8:1577–1594, 1984.

[19] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *The Bell System Technical Journal*, 57(6 (part 2)):1905+, 1978.

[20] Yannis Smaragdakis. Layered development with UNIX dynamic libraries. In *Proceedings of the Seventh International Conference on Software Reuse (ICSR)*, 2002.

[21] Larry Wall and Randall L. Schwartz. *Programming Perl*. O'Reilly & Associates, 1992.