Search Should Be a System Call

Brenden Kokoszka and Patrick Donnelly and Douglas Thain
Department of Computer Science and Engineering, University of Notre Dame
Technical Report TR-2013-03
April 2013

Abstract

Conventional operating systems are designed with the assumption that applications are relatively close to their data. However, as virtualization of systems, networks, and storage becomes more widespread, the typical latency between applications and data has increased significantly. This increase in latency can have a dramatic effect on application performance, particularly on common system-level tools that perform frequent metadata searches. To address this problem, we propose that metadata search be elevated to a first-class system call within the kernel. We present three implementations of the concept: in an operating system kernel, in a ptrace sandbox, and in a user-level file system. We demonstrate that there are many opportunities to easily exploit the search capability in standard system-level tools with modest coding effort and no change in user behavior. We evaluate the performance of common applications using the search system call with varying levels of virtualization, showing reductions in system call traffic ranging from 5 to 95 percent.

1 Introduction

Traditional operating system interfaces were designed in a time when the software was relatively "close" to the hardware. As such, most system calls are relatively simple requests that correspond closely to capabilities in the underlying hardware, and are expected to have relatively low cost. However, as an increasing number of virtualization technologies are put into wide use, the distance between software and hardware increases. Virtual machines increase the cost of communication between applications and the operating system, and between the OS and its drivers. Distributed file systems, storage networks, and storage virtualization all increase the distance and cost between device drivers and the underlying storage devices. As a result, operations that were once considered simple and inexpensive can increase in cost by several orders of magnitude.

There is a long history of addressing this type of problem by increasing the level of abstraction used by the application. By allowing an application to express a high-level intent to perform multiple operations, the underlying software and hardware can do a better job of carrying out the user's intent. A well known example of this evolution took place in disk drives. An OS no longer needs to direct the mechanical motion of a spinning disk; instead, the OS issues multiple operations concurrently via NCQ [5] and the disk drive (or disk array) takes care of scheduling, buffering, and fault tolerance. Likewise, it is rare for modern applications to draw graphics pixel-by-pixel; instead, complex operations are specified in a language like Postscript [15] or OpenGL [24], enabling fast, parallel rendering on specialized architectures found in devices like printers and GPUs.

We argue that the distance between applications and storage has increased to the point where it is time to re-examine the interface between the application and the file system.

In this paper, we focus on the performance of *metadata search* within the file system. A large fraction of the system call interactions between conventional applications and the OS is due to programs searching through the file system: the shell searches for executables to run, the linker searches for libraries to load, interpreted languages search for classes and extension modules, the list goes on and on. Even the most trivial application like `ls` can invoke thousands of system calls before it begins its real work. When these searches occur on a conventional operating system running on native hardware, the cost can be overlooked. But, when multiple virtualization layers are introduced and

system scale grows, these searches become a significant fraction of execution time.

To address this problem, we introduce a new system call named **search()**. Search simply takes a list of paths to be searched, a target pattern to find, and a few options to control the behavior of the search. By elevating this operation into a single system call, we introduce several new opportunities:

- **Fewer Kernel Interactions.** Every interaction an application has with the operating system kernel is an interruption in task scheduling, cache locality, and hardware properties. Especially as layers of virtualization are introduced, the cost of each interaction increases. By reducing the interaction to a single system call, the penalty of virtualization is reduced.
- **Remote Delegation.** In the case where the metadata is stored on a remote service, typically a distributed file system, the entire search operation can be delegated to the remote device, where it can be performed close to the data itself. Modern file systems that implement a distinct metadata server (or a cluster of metadata servers) are particularly well-suited for this.
- **Database Techniques.** Once the entire search operation is expressed as a single operation, a whole fleet of database techniques become relevant to serving the operation efficiently – directories may be indexed, queries may be reordered for locality, and results cached or combined for scalability.
- **Parallelization.** In cases where the application wishes to search across multiple file systems – or the file system itself has internal parallelization – parts of the search may be performed concurrently for significant speedup.

We evaluate the concept of the search system call by implementing it within the native Linux kernel. We modify a selection of standard GNU system tools to use the system call internally without any change in user behavior. We evaluate a series of system call intensive operations in a number of configurations including native hardware, a KVM [12] virtual machine, a ptrace-based sandbox, and a user-level remote file system. We demonstrate that `search` dramatically reduces the number of system call interactions between user and kernel. This has a modest performance benefit when running on native hardware, and an increasing benefits as the cost of virtualization is increased.

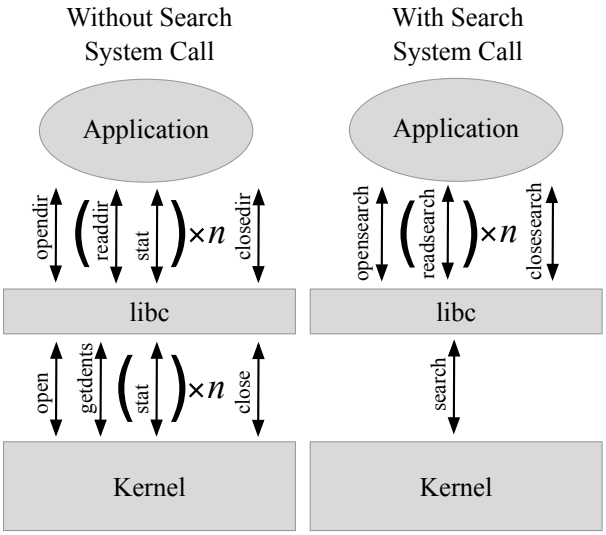

Figure 1: *A single* `search` *system call replaces the many system calls used to search through an n-file directory.*

2 Search Opportunities

In a conventional operating system, search is employed ubiquitously by many different system tools in order to locate the components that they need to operate correctly. In all cases, a large number of metadata operations are issued without interruption in order to reach a decision.

Following are examples of search operations common to many operating systems, with explanations of how they appear in current versions of the GNU tools running on Linux.

Path Search. Every time a new program is invoked, the file system must be searched for the desired executable to run, taking the list from the `PATH` environment variable. In a large organization with a shared file system (or on a personal computer with a lot of software) there may be a very long list of directories to search before finding the desired entry.

On GNU-Linux, path searching is typically done by the standard library in the `execvp()` system call, or in some cases is handled by the user's shell. A typical system call trace would produce a series of `exec` system calls to test for the existence of the executable for each entry in the `PATH`.

Library Search. Similar to `PATH` searching, programs search for a number of libraries on startup. Modular setups also affect the time to search for a library due to the setting of the `LD_LIBRARY_PATH` environment variable. The number of permutations increases with architectural variants supported. In the case of an application which links to dozens of

```
open("/opt/lib/tls/x86_64/libselinux.so.1", O_RDONLY) = -1 ENOENT (No such file or directory)
stat("/opt/lib/tls/x86_64", 0x7fff941a9720) = -1 ENOENT (No such file or directory)
open("/opt/lib/tls/libselinux.so.1", O_RDONLY) = -1 ENOENT (No such file or directory)
stat("/opt/lib/tls", 0x7fff941a9720)    = -1 ENOENT (No such file or directory)
open("/opt/lib/x86_64/libselinux.so.1", O_RDONLY) = -1 ENOENT (No such file or directory)
stat("/opt/lib/x86_64", 0x7fff941a9720) = -1 ENOENT (No such file or directory)
open("/opt/lib/libselinux.so.1", O_RDONLY) = -1 ENOENT (No such file or directory)
stat("/opt/lib", 0x7fff941a9720)        = -1 ENOENT (No such file or directory)
open("/usr/local/lib/tls/x86_64/libselinux.so.1", O_RDONLY) = -1 ENOENT (No such file or directory)
stat("/usr/local/lib/tls/x86_64", 0x7fff941a9720) = -1 ENOENT (No such file or directory)
open("/usr/local/lib/tls/libselinux.so.1", O_RDONLY) = -1 ENOENT (No such file or directory)
stat("/usr/local/lib/tls", 0x7fff941a9720) = -1 ENOENT (No such file or directory)
open("/usr/local/lib/x86_64/libselinux.so.1", O_RDONLY) = -1 ENOENT (No such file or directory)
stat("/usr/local/lib/x86_64", 0x7fff941a9720) = -1 ENOENT (No such file or directory)
open("/usr/local/lib/libselinux.so.1", O_RDONLY) = -1 ENOENT (No such file or directory)
stat("/usr/local/lib", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
...
```

Figure 2: Example of Dynamic Library Searching
The first lines few output by `strace ls`, showing the dynamic linker searching for a single standard library `libselinux.so.1` when the library path contains three entries: `/opt/lib:/usr/local/lib:/lib64`.

libraries, it is possible for thousands of system calls to fly by before the program has an opportunity to run. Figure 2 shows an all-too-common example of `ls` searching for its libraries.

Filename Globbing. The shell often undertakes search to complete filename patterns, which is known as *globbing*. These may be simple completions like `*.c` or recursive directory completions like `*/*`. The user may also do interactive completions by typing a file prefix and pressing `TAB`. In GNU-Linux, the `bash` shell handles these by opening the directory appropriate to the context, reading all of the entries with `getdents` and then applying `stat` to get the metadata of the desired matches.

Directory Listing. Listing the contents of a directory is another common search pattern. While a program such as `ls` is the most obvious example of directory listing, there are many other applications that could benefit from the search system call. For example, when changing to another directory, many shells will list child directories of the parent, as part of tab-completion, to offer suggestions on a directory to change to. In this case, the shell is searching a single directory for directories or links pointing to directories. In GNU-Linux, `ls` uses `getdents` to retrieve all the names in a directory, then calls `stat` for every entry when detailed metadata is required.

Tree Search. The GNU `find` utility provides a command interface for searching for files in the file system. It provides advanced mechanisms for searching a directory tree such as pruning sub-trees, ignoring files, and selecting files based on metadata. Internally, this follows the same pattern as those tools above.

These are only a few examples of search. Many others come to mind: compilers search for include files, web browsers search for extensions, and dynamic languages search for modules. Each of these examples can be collected into a common code pattern: for each directory in a list, search for the presence of files matching a pattern, and return that list of files and (optionally) their metadata. We propose to elevate this common pattern into a single system call – **search** – which carries out the entire set of metadata operations and returns a bulk result to the caller.

3 Search Interface

We define the search system call as follows:

```
int search(const char *paths,
           const char *pattern,
           int flags,
           void *buffer,
           size_t len);
```

Where **paths** is a colon-separated list of directories to be searched, **pattern** is a filename pattern to search for, **flags** control the search behavior, **buffer** is where the results will be placed, and **len** is the size of the buffer.

The `pattern` argument is a glob-like expression against which file pathnames are matched. We selected pattern semantics that best served the needs of the system tools described below. The following characters have a special meaning in `pattern`:

- *** (asterisk)** The asterisk is a wildcard matching zero or more characters, but not /, the directory separator. For example, `f*/b*` would match "foo/bar" but not "foo/xyz/bar".
- **? (interrogation point)** The interrogation point is a wildcard matching zero or one characters, but not /.
- **| (pipe)** The pipe is an alternation operator allowing multiple patterns to match against a pathname (logical OR).
- **/ (path separator)** The forward slash matches literally against a file's canonical pathname. Matching the forward slash follows the semantics of the UNIX `fnmatch` function to match a directory heirarchy, for example, `linux/Makefile` will only match file "Makefile" if contained in a parent directory "linux".

  All patterns are anchored to match against the end of the pathname. Simply put, this means the pattern must always match the final pathname component, the filename. The pattern may match the pathname *partially*. As a special case, a leading forward slash anchors a pattern to the top-level directory being searched. This causes the pattern to be anchored on both the front and back of the pathname and affects the recursive behavior of `search`. A pattern with a leading forward slash is considered *non-recursive* while any other pattern is *recursive*, able to match a file anywhere in the directory tree.

The `flags` argument contains a number of bit fields including:

- **STOPATFIRST** Halt `search` once the first match is encountered. This flag is particularly useful when searching for libraries.
- **METADATA** Include the the `stat` information for each matched file as though a `stat` system call were executed on each result.
- **INCLUDEROOT** Include the top-level directory the matched filename was found in. This option is necessary if more than one directory is in the `paths` argument to `search`.
- **R_OK, W_OK, X_OK** Only match files that the user can read, write, or execute, respectively.

The results of `search` are returned in a binary-packed format that is space-efficient but somewhat inconvenient to parse. Instead, we provide a user-level interface that is analagous to `opendir`:

```
SEARCH * opensearch( char *paths,
                     char *pattern,
                     int flags);
struct searchent *
         readsearch( SEARCH *search );
int      closesearch( SEARCH *search );

struct searchent {
    char *path;
    struct stat *info;
    int errsource;
    int err;
};
```

When `opensearch` is called, it calls the `search` system call and stores the results. Each call to `readsearch` parses another `searchent` structure out of the buffer, which contains a matching path, the metadata, and (possibly) an error condition. `closesearch` releases the buffer when done.

3.1 Examples

A few examples will clarify how the interface is used in practice. Suppose that a user at the shell enters `ch` and presses `TAB` to complete the list of executables with that prefix. The shell code (ignoring error handling) would look like this:

```
SEARCH *s;
struct searchent *e;

s = opensearch(
        "/bin:/usr/bin:/home/fred/bin",
        "/ch*",INCLUDEROOT|X_OK);

while( e=readsearch(s) ) {
    printf("%s\n",e->path);
}

closesearch(s);
```

After reviewing the list, the user selects `chmod` and enters the command. The shell would then locate the executable like this:

```
opensearch(
       "/bin:/usr/bin:/home/fred/bin",
       "/chmod", INCLUDEROOT|STOPATFIRST|X_OK);
```

A common directory list, `ls -l`, is simply:

```
opensearch(".", "/*", METADATA);
```

Now suppose the user enters `cat */*.c` to concatenate all C source files in all directories. The shell would implement the pattern completion:

```
opensearch(".", "/*/*.c", INCLUDEROOT);
```

The `search` interface can also be used to perform directory-related operations which may not commonly be considered searches. For example, this concatenates the contents of two directories:

```
opensearch("/foo:/bar", "/*", INCLUDEROOT);
```

While the following gives a flattened representation of the tree at `foo`, using a *recursive* pattern:

```
opensearch("/foo", "*", INCLUDEROOT);
```

Now if the user wants to recursively locate all C source files below the current working directory via `find -name \*.c`:

```
opensearch(".", "*.c", INCLUDEROOT);
```

4 Other Considerations

Error Reporting. There are a number of file operation primitives which may fail during the course of a search. Some failures may be expected and should not cause the entire search to fail. For example, a file or directory may be unlinked before `stat` or `open`, or the permissions on a directory may not allow traversal. When handling these common errors, the operating system tries to gracefully ignore the error but still passes the failure up to the user within the `result` buffer.

A failed action is reported in the `result` buffer in three parts: the type of action which failed, the pathname which the action failed on, and the `errno` returned by the action. For example, if while performing a recursive search, a sub-directory named "Private" for which the user lacks list permissions is encountered, an `open` operation would fail on path "Private" with error `EPERM`. This information is reported in the `errsource`, `path`, and `err` fields of the `searchent` struct, respectively.

Crossing File System Boundaries. During the course of `search`, a mount point for another file system might be encountered. If the mounted file system natively supports the search system call, it can be most efficiently searched by invoking a new child `search` system call directly on the mounted file system's root. The results from the child `search` can then be appended to the results of the parent `search` operation.

Because a new `search` call may be launched at a file system boundary, it is necessary to modify the pattern for the new call. This is due to partial pathname matches where the parent directory exists on another mount point. For example, "/var/tmp" is commonly mounted on a separate file system to prevent "/var" or "/" from running out of space. If a user searches "/var" with the pattern `tmp/foo`, the second `search` passed to the file system driver for "/var/tmp" should receive two patterns `/foo|tmp/foo`. The first pattern `/foo` will only match a top-level file "/foo" in the "/var/tmp" file system. The second pattern will allow the recursive search for `tmp/foo`. This *pattern decomposition* always results in non-recursive patterns being added to the new search.

Interruptibility. A potentially long-running system call should be interruptible so an application, such as a shell, can be stopped by a user or external force. At most system layers, the `search` system call is decomposed into a series of `stat`, `access`, `getdents`, and `open` system calls. So if these file system operation primitives are interruptible, then the search can also be interrupted.

If `search` is implemented by the file system — probably a network file system — then interruptibility is more difficult. In the case of a network file system, interrupting the remote procedure call (RPC) may not be possible so the system is forced to disconnect from the network volume temporarily. Current implementations of NFS provide an `intr` mount option to allow signals to interrupt the process using NFS. Otherwise, a process would be listed as *uninterruptible* and cannot be destroyed.

5 Implementations

To validate this concept, we have implemented the `search` system in the Linux kernel, in a ptrace-based sandbox, and in a user-level distributed file system. In each case, the necessary code modifications were modest, only a few hundred lines of code.

5.1 Native Linux

Our implementation of `search` within Linux comprises about 500 total lines of code. The system call is implemented entirely within the portable system layer and invokes abstract VFS operations to traverse directories, obtain metadata, and so forth. A more complete implementation would add a `search` operation to the VFS interface, and delegate the operation to drivers that provide an optimized implementation.

It uses a buffer-oriented interface for returning results to the user. This is a necessity due to the kernel

being unable to return results via a callback (without heavy cost). This type of interface is mirrored in the Linux **getdents** system call which returns a number of directory entries in a buffer for an open directory. For the library interface, we are still able to offer a streaming interface through `readsearch`.

Linux uses a virtual file system (VFS) abstraction to enable the various file I/O system calls to easily interact with a directory tree composed of multiple file systems. These file systems can be very different but offer a common interface to the VFS adapter. Like other I/O system calls such as `open`, we use this interface to search through the directory trees for matching files. The basic operations we will use are reading directory entries (`readdir`) and getting the attributes of matched files (`stat`).

One of the complications of creating a `search` system call on Linux is dealing with the recursive aspect of a depth-first search. Each directory searched results in a new procedure call to match the contents of the directory with the given pattern. Additionally, matching *globbing* patterns–patterns which can match a range of characters–is naturally recursive. Linux strongly discourages recursive algorithms because of very limited stack space available. By default, this stack space is 8192 bytes on a RedHat Linux 6 x86-64 machine. Because of this, we initially allocate large blocks on the heap to store directory entries and various state. This allows the stack to be used almost exclusively for the procedure call frames, allowing deep recursion. By default, we limit a search to 16 levels of recursion before raising an `EOVERFLOW` error. So, searching `/usr` for a file stored in `/usr/1/2/3/4/.../16/17` would cause an abort in the system call.

Our tests for Linux will focus on achieving at least competitive performance for search-enabled utilities with the equivalent non-search-enabled utilities. **stat** and **open** are aggressively optimized system calls which, when data is colocated with the kernel, will be difficult to match or beat. The main performance benefit comes from avoiding system call transitions.

5.2 Ptrace Sandbox (Parrot)

A growing amount of software uses the `ptrace` interface as a basic virtualization mechanism, allowing a controlling process to run a captive process and to observe or implement system calls on its behalf. Common applications include full virtualization [6], local file system development [26] or process isolation.

In this case, we have used Parrot [28], a `ptrace`-based virtual file system typically used to attach programs running in a batch system to remote storage devices over the wide area network. Like any similar software, programs running in Parrot pay a performance penalty due to the order-of-magnitude increase in system call latency through `ptrace`.

From the user perspective, Parrot hooks other file systems into the system namespace as if they were mounted normally via the kernel. For example, a GNU FTP server could be accessed as a file via "/ftp/ftp.gnu.org/gnu/tar/tar-1.26.tar.gz" directly from the ptraced application. We will utilize this capability in our tests of the `search` system call against custom remote file systems.

To Parrot, we have added an implementation of the `search` system call, which is also about 500 lines of code. In the simplest case, the application invokes `search` and Parrot decomposes it into basic system calls, thus avoiding multiple `ptrace` transactions. When the underlying file system driver supports it, the search operation is passed over the network in order to avoid multiple network transactions.

5.3 User Level Filesystem (Chirp)

To evaluate the use of `search` in a distributed context, we make use of the Chirp [29] distributed file system, which is a user-level system originally designed to interoperate easily with Parrot. A Chirp file server is a user-level process that accepts incoming TCP connections, authenticates the client, and then processes I/O requests in a simple ASCII protocol that corresponds closely to the POSIX interface. (e.g. the client sends the text "open /etc/hosts r" and the server responds with a file descriptor.)

To Chirp, we added a `search` RPC which corresponds closely to the `search` system call. The client sends a path list, a pattern, and flags, and the server responds with the list of results. Because the client and server communicate via TCP, the `search` RPC can stream back an arbitrary number of results without problem. Further, if the file server is running on a kernel that supports the system call natively, the operation can be passed down yet again.

From the user's perspective, a Chirp server can be accessed by starting the application through Parrot, and then accessing files through a path like `/chirp/hostname/path` as if they were local file names. To evaluate storage systems of varying latency, we added to Chirp an option to add an artificial delay to each RPC.

One complication that results is when the Chirp server is acting as an access control enforcement point. The calling user may not have access to all

of the files that the server itself can access. In this case, when the client requests a search, the file server must filter the results from either the native kernel search system call or implement the search itself.

6 Modified Applications

To demonstrate the benefit of using the `search` system call, we modified several common applications and libraries to use `search`. We then compared the search-enabled applications with their unmodified counterparts, as will be discussed in Section 7.

bash v4.2: There are two obvious locations for search optimization in the `bash` shell: PATH searching and glob expansion. PATH searching is implemented much like the examples in Section 3.1. glob expansion is not complete because bash uses its own globbing language that is an extension of the `glob` and `fnmatch` interfaces in the standard library.

For simplicity, we did not handle these extensions. In a complete implementation, we would expect that an application with patterns that cannot be expressed in the `search` system call would pass a simplified search down to the kernel, then perform a second level of filtering at the user level.

ls v8.13: We extended `ls` to use the `search` system call to obtain the listing for a directory. By using the `METADATA` flag, `ls` is able to obtain the `stat` information for each entry without `stat`ting results individually (e.g. `ls -l`). By using recursive patterns, `ls` can return results from an arbitrarily deep directory tree with a single call to `search`.

ld.so v2.12.2: The loader must search `LD_LIBRARY_PATH` for dynamic libraries each time an application is executed. The optimization which presents itself here is essentially the same as the one made to bash's `PATH`-searching routine. A single `search` system call is used to check each entry in `LD_LIBRARY_PATH` for the needed library.

glibc v2.12.2: The one glibc function we modified is `execvpe`. Normally, `exevpe` appends the executable name to each `PATH` element and attempts to execute the resulting path, generating at most one `exec` system call for each `PATH` element. The `search` system call performs this `PATH`-searching in a single system call.

find v4.4.2: Like `ls`, `find` traverses a directory structure and outputs its contents. However, unlike `ls`, `find` provides an extensive set of options for pruning the search tree and filtering search output. It is not possible to entirely replicate the functionality of `find` using solely the `search` system call. However, it is still possible to intelligently utilize the `search` system call at certain places within `find` to improve performance.

Our modification of `find` uses the `METADATA` flag to avoid making `stat` system calls. The directory on which `find` is applied gets recursively searched and the `stat` struct of each directory entry is saved to a lookup table. `find` checks this table whenever it would otherwise make a `stat` system call, thus greatly reducing the application's system call overhead.

7 Evaluation

To evalute the impact of the search system call, we focus on two metrics: system call count and wall clock execution time. Though execution time is what we seek to reduce in practice, it is highly dependent on the environment in which the test is performed. System call count, on the other hand, is a portable result that explains the reason behind the speedup.

The tests are run on a workstation machine with a Intel Core 2 Duo x86-64 CPU (2 cores), 4 GB of RAM, 8GB of swap space, and a 1 gigabit network link. The system runs Red Hat Enterprise Linux 6.3 using Linux kernel 3.4.0. An identical machine on the same switch is used as a remote fileserver when needed. File system caches are dropped at the beginning of each measurement.

The tests are run in six different environments:

- **Native** Red Hat Enterprise Linux 6.3 (Linux kernel 3.4.0) modified to natively support the `search` system call.
- **KVM** The previous environment running as a KVM virtual machine.
- **ptrace Sandbox** Parrot running on the native environment, passing through all file system operations to the underlying file system.
- **Remote {1ms,5ms,}** Parrot running on Linux with a remote Chirp [29] file system. In the Remote 1ms and Remote 5ms environments, we have imposed an artifical 1ms and 5ms RPC latency on top of the latency of our local network.

The environments differ in terms of the time cost of performing a system call, as summarized in Table 1.

Across these six environments, we ran eight applications. The results of the tests are in Tables 2 and 3.

Environment	stat μs	open μs
Native	6.8	9.0
KVM	10.8	14.9
ptrace	101.5	152.0
Remote	221.0	335.8
Remote 1ms	1365.3	2671.5
Remote 5ms	5367.0	10684.5

Table 1

- **find** - Executes the command `find testbench/10 -name xyz`, where the directory "testbench/xyz" contains 1024 child directories.
- **ls -l** - Executes the command `ls -l` on a directory containing 500 files. The reduction in `open` system calls is mostly due `ld.so`'s use of `search` to locate libraries. The reduction in `stat` system calls is due to the use of the `search` system call to retrieve `stat` information for each of the listed items.
- **ls -lR** - Executes the command `ls -lR` on a directory tree containing 1024 child directories. The `search` system call has a much larger impact on `ls` when it is used recursively as in this test. Because `search` can match filenames recursively, the contents of an arbitrarily large directory tree can be retrieved with a single `search` call.
- **exec** - Executes a simple C program that calls `execvpe` to execute `cat` and then terminates. Three `search` system calls made by search-enabled glibc replace over half of the system calls that are made by the program when run with unmodified glibc, resulting in a sizable reduction in execution time.
- **configure** - Runs the `configure` script for `vim` version 7.3. Present in the test are four different uses of `search`: PATH-searching in `bash` and `execvpe`, glob expansion, and library-searching. This test highlights the benefits of the `search` system call in a non-trivial situation.
- **bash** - Opens `bash` and runs the command `ls *` on a directory containing 500 files. In this test, it is `bash`, not `ls`, that has been modified to use the `search` system call.
- **firefox** - Opens the address `http://google.com` with Firefox, renders it, and then exits. As with `make`, we did not modify Firefox, and so this test shows only the impact of the glibc optimizations. Unlike `make`, Firefox only uses `ld.so` and not `execvpe`. Though benefiting from only `ld.so` optimizations, Firefox still exhibits modest reductions in system call count and execution time.
- **make** - Builds `vim`, i.e. runs `vim`'s makefile. We did not modify `make` itself, so the system call reduction in the `search`-enabled run of the test is due solely to search optimizations in `bash` and glibc.

System Call Count. Table 2 shows the result of running each of the benchmark applications on the native machine using the `strace` tool to count the number of system calls with and without the search system call enabled. The right side of the table breaks down the `search` system call counts by the calling code, indicating the nature of the optimization. For applications whose key functionality is addressed by `search` (`find` and `ls`), the reduction in system calls is an order of magnitude or more. For others (`firefox` and `make`) the improvement is more modest (4-10 percent.)

Execution Time. Table 3 shows the wall clock time to run each of the benchmark applications on each of the six environments, with and without the `search` system call optimization. (In the remote cases, `search` is executed at the remote file server.) In the native case, the time benefits are modest, but increase as the virtualization levels increase.

Directory Scalability. Figure 3 shows the execution time and system calls, respectively, made by `find` as it searched a complete balanced binary directory tree of varying depths. Even as the number of directories increases exponentially, the search-enabled version of `find` exhibits only a slight linear increase in its execution time and system call count. In this case, the cost of increasing the directory tree size is dominated by user-kernel switching.

Remote Delegation. Figure 4 shows how search can be implemented in different layers of the software stack, ranging from the application itself (the conventional) way, in the kernel, in a remote fileserver, or in the remote fileserver's kernel. In this case, we re-run `find` in each configuration, demonstrating how moving search down each layer reduces the total number of interactions and the wall clock time.

Application	Total Count	% Reduc.	Totals by System Call Type					Search Callsites					
			open	{f,l,}stat	execve	getdents	other	bash glob	bash PATH	ls	find	execvp	ld.so
find -name xyz	129	99.5%	18	11	2	0	98				1		4
	26789		6202	4130	2	4094	12361						
ls -lR	2231	91.0%	19	15	2	0	2195			1			8
	24857		2156	6192	2	4094	12413						
exec	102	53.4%	15	9	4	0	74					1	2
	219		69	63	20	0	67						
ls -l	1201	33.6%	20	516	2	0	663			1			8
	1809		110	1051	2	2	644						
configure	630494	21.8%	38415	90345	5077	52	496605	221	318			183	5197
	807159		132011	177403	11411	494	485840						
bash -c 'ls *'	1382	12.3%	36	1040	4	2	300	1	1				11
	1576		172	1128	4	2	270						
firefox	4764	10.1%	105	112	2	0	4545						63
	5302		646	145	2	0	4509						
make	985684	4.6%	288628	142257	1138	30	553631	3	110			152	1603
	1033077		313852	164357	4691	36	550141						

Table 2: This table profiles the types of system calls generated in each of the tests we ran (columns 4-8). It further organizes search system calls by the site from which they were invoked (columns 9-14). Gray cells denote the versions of programs extended to use the search system call.

Application	Native		KVM		ptrace Sandbox		Remote		Remote 1ms		Remote 5ms	
	Time	% Reduc.	Time	% Reduc.	Time	% Reduc.	Time	% Reduc.	Time	% Reduc.	Time	% Reduc.
find -name xyz	2951	16.3%	6002	9.6%	1176	86.1%	7479	63.8%	7695	85.9%	8357	95.1%
	4054		11024		8519		20662		54712		172022	
ls -lR	3085	23.9%	9400	14.7%	2291	69.5%	6882	54.9%	7304	77.5%	8327	90.9%
	4407		11024		7528		15293		32591		91778	
exec	20	25.9%	106	24.2%	491	22.6%	5707	1.1%	5892	2.9%	6692	8.3%
	27		140		635		5769		6068		7294	
ls -l	89	23.2%	946	13.2%	1095	4.7%	6880	2.3%	8963	8.6%	16408	15.3%
	116		1091		1149		7043		9804		19372	
configure	15043	0.4%	25894	2.12%	118442	19.5%	167109	18.7%	338480	33.8%	804196	40.7%
	15109		26457		147166		205644		511484		1357043	
bash -c 'ls *'	269	18.2%	599	3.2%	1197	8.1%	7433	-5.5%	6514	1.1%	6964	7.0%
	329		619		1303		7040		6589		7487	
firefox	759	27.0%	1418	26.6%	65426	2.3%	13915	-1.5%	14400	-1.1%	88024	9.5%
	1040		1932		66959		13707		14240		97246	
make	72957	0.6%	91124	0.5%	252230	6.5%	389438	3.9%	829179	15.0%	2340774	18.3%
	73413		91625		269812		405377		976440		2866695	

Table 3: This table shows the wall time in milliseconds required to run each test program both with and without the search system call in various environments. The displayed times are in milliseconds and are averages from ten runs (all but two tests had standard deviations below 22% of their mean). Gray cells denote the versions of programs extended to use the search system call.

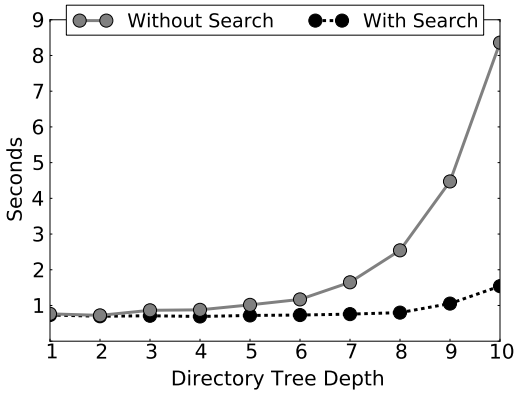

Figure 3: *Execution time of the* `find` *tool as it searches through a complete binary directory tree of increasing depth. Note that each tree contains* $2^{depth}-1$ *directories so the x-axis is logarithmic with respect to directory count.*

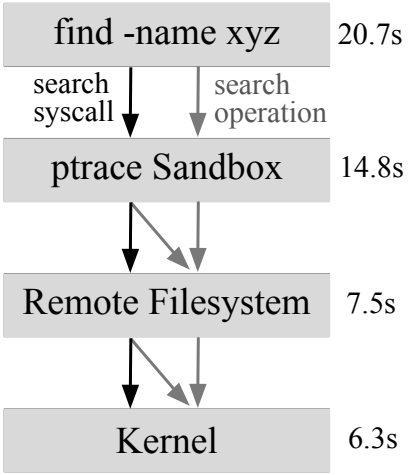

Figure 4: *Execution time of* `find` *script with the search operations progressively moved into deeper layers of the storage stack. The black arrows represent the* `search` *system call; the gray arrows represent the search operation itself (consisting of many* `stat()`*s* `opens()`*s, etc.)*

8 Observations

- *Empircally, search is a very common activity in standard system tools.* Although we have only instrumented a handful of code points, we have already captured hundreds of search operations per second, particularly in highly scripted tasks like `configure`. Many of them are redundant in that they search for the same executables and libraries multiple times from different programs, and could benefit from some form of query caching in the kernel.
- *The* `search` *system call significantly reduces the number of user-kernel actions.* Even in small-scale user level tools like `ls`, there is only one call to `search`, but it eliminates thousands of interactions with the kernel and (potentially) the underlying file system. In larger scale tests like `configure` and `make`, the overall effect is reduced, but given the larger number of calls like `open` and `stat` remaining, the are further opportunities to locate and elevate `search`
- *The higher level abstraction of* `search` *has major benefits in high virtualization environments.* When running on native hardware, `search` offers modest benefits in execution time (with no penalty), but as virtualization is added, the ability to "punch through" the layers pays increasing dividends. In one case (`find`), the time improvement is an order of magnitude.

9 Related Work

File system studies [21, 13] have found that as many as 42% of file system system calls were `stat` and `fstat`. The common usage pattern identified by Roselli et al. is a `stat` for each file in a directory listing and a `stat` before each open, which is consistent with our observations. The problem is exacerbated in large multi-user distributed file system installations, where it is common to have many independent directories for each software item installed.

A common design pattern of scalable file systems is a centralized metadata service describing a large number of storage devices that contain file data. This pattern is used in AFS [11], GFS [8], Hadoop [9], and NFSv4 [23]. While a central metadata server simplifies consistency management, it presents a secondary scalability limit. This has been addressed in recent years by file systems that put the metadata alongside the data storage [31], in a small cluster of metadata servers (Ceph [32], GPFS [22]), or in a distinct scalable name service (GIGA [19]). In each of these cases, the cost of performing metadata operations is quite high because the client must perform multiple operations against several servers, and cache coherency issues prevent caching of data. These are all good candidates for implementing a native search operation within the metadata server itself.

A number of additions to the POSIX interface and semantics have been proposed to address common metadata operations of applications, typically

by optimizing one particular system call at at time. The most widely used optimization (as in NFS) is to permit the caching of metadata queries for a short time, even when this conflicts with the desired consistency semantics. POSIX extensions have been proposed [33, 30] which returns the results of `lstat` for each directory entry in `getdents` This extension has been implemented in HPC file systems such as PVFS [3, 4]. Still, even with improved directory listing support, directories must be opened and closed which can result in performance bottlenecks [2]. Patil et al.[18] have proposed the evolution of the file system in this direction by adding higher level capabilities that address indexing, search, and file locality, demonstrating a prototype based on PVFS.

Deeper in the storage stack, a number of efforts have improved the organization of the underlying devices to support search operations. In-memory directory hashing and on-disk B-trees have been implemented in FreeBSD FFS [7] and XFS [27], respectively, to provide better support for file searching and indexing. The object storage model [16] raises the level of abstraction of individual disks from blocks to inode-like extents. Going farther, intelligent storage [20] proposes that search and indexing capabilities be native to individual storage devices. Our proposal for a search system call is consistent with this line of development, and could be pushed as far as implementing a search capability within individual storage devices.

From the top down, there have been many attempts to combine the capabilities of file systems and databases, so as to facilitate more efficient data query, often for rich media applications. Inversion [17] is an example of implementing a file system on a relational database (rather than vice versa) so as to accomodate deep queries into the file system tree. Spyglass [14] is developed on top of file system infrastructure to harness access to this information to allow for efficient metadata search queries. Most desktop operating systems now incorporate an asynchronous indexing service to permit deep content search across the file system. Our work is complementary to these efforts in that we show how search is *already ubiquitous* in standard system tools and not just a value-added operation for the end user. By elevating search to a first-class operation, we accelerate existing tools without any change in user behavior, enabling future connections to these types of operations.

A long-standing design question is whether new operations in an existing interface should be achieved by carefully selecting new fundamental operations or by adding a limited programming interface. Over the years, the latter approach has seen prototypes in operating systems [1], networks [10], and file servers [25], but limited deployment in practice. In this work, we have made a limited expansion of programmability in a way that closely matches application needs without seeking complete generality of computation.

10 Conclusion and Future Work

We have argued that search should be a first class system call in the file system interface. We have demonstrated empirically that searches are common in standard system tools, and that by elevating the interface, we can significantly reduce the total number of interactions by moving the implementation into the kernel. The benefit of this change is amplified as the cost of virtualization increases. In future work, we envision identifying other high level operations that are candidates for elevation in the file system interface, finding opportunitites to implement search within scalable distributed file systems, and exploiting opportunitites for parallelism and other optimizations.

References

[1] B , B., S , S., P , P., S , E., F , M., B , D., C , C., E , S. Extensibility safety and performance in the spin operating system. In *ACM SIGOPS Operating Systems Review* (1995), vol. 29, ACM, pp. 267–283.

[2] B , P., N , P. Removing bottlenecks in distributed filesystems: Coda & InterMezzo as examples. In *Proceedings of Linux Expo 1999* (1999).

[3] C , P., L , S., R , R., V , M., K , J., L , T. Small-file access in parallel file systems. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on* (2009), IEEE, pp. 1–11.

[4] C , P., L III, W., R , R., T , R. PVFS: A parallel file system for linux clusters. In *Proceedings of the 4th annual Linux Showcase & Conference-Volume 4* (2000), USENIX Association, pp. 28–28.

[5] D , B. Native command queuing-advanced performance in desktop storage. *Potentials, IEEE 24*, 4 (2005), 4–7.

[6] D , J. A user-mode port of the Linux kernel. In *USENIX Annual Linux Showcase and Conference* (Atlanta, GA, October 2000).

[7] D , I., M , D. Recent filesystem optimisations on freebsd. In *Proceedings of the USENIX Annual Technical Conference (FREENIX Track)* (2002), pp. 245–258.

[8] G , S., G , H., L , S. The Google filesystem. In *ACM Symposium on Operating Systems Principles* (2003).

[9] H . http://hadoop.apache.org/, 2007.

[10] H , M., K , P., M , J., G , C., N , S. Plan: A packet language for active networks. In *ACM SIGPLAN Notices* (1998), vol. 34, ACM, pp. 86–93.

[11] H , J., K , M., M , S., N , D., S - , M., S , R., W , M. Scale and performance in a distributed file system. *ACM Trans. on Comp. Sys. 6*, 1 (February 1988), 51–81.

[12] K , A., K , Y., L , D., L , U., L , A. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium* (2007), vol. 1, pp. 225–230.

[13] L , A., P , S., G , G., M , E. Measurement and analysis of large-scale network file system workloads. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference* (2008), pp. 213–226.

[14] L , A., S , M., B , T., P , S., M , E. Spyglass: Fast, scalable metadata search for large-scale storage systems. In *Proccedings of the 7th conference on File and storage technologies* (2009), USENIX Association, pp. 153–166.

[15] M G , H. *PostScript by Example*. Addison-Wesley Longman Publishing Co., Inc., 1993.

[16] M , M., G , G., R , E. Object based storage. *IEEE Communications 41*, 8 (August 2003).

[17] O , M. The design and implementation of the Inversion file system. In *Proceedings of the Winter 1993 USENIX Technical Conference* (1993), pp. 205–217.

[18] P , S., G , G., G , G., L , J., P , M., T - , W., X , L. *In search of an API for scalable file systems: Under the table or above it?* Defense Technical Information Center, 2009.

[19] P , S., G , G., L , S., P , M. Giga+: scalable directories for shared file systems. In *Proceedings of the 2nd international workshop on Petascale data storage: held in conjunction with Supercomputing'07* (2007), ACM, pp. 26–29.

[20] R , A., J , M., M , M., D , B., D , D. Towards efficient search on unstructured data: an intelligent-storage approach. In *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management* (2007), ACM, pp. 951–954.

[21] R , D., L , J., A , T. A comparison of file system workloads. In *Proceedings of the annual conference on USENIX Annual Technical Conference* (2000), USENIX Association, pp. 4–4.

[22] S , F., H , R. GPFS: A shared-disk file system for large computing clusters. In *USENIX Conference on File and Storage Technologies (FAST)* (Jan 2002).

[23] S , S., N , D., E , M., R , D., C , B., T , R., B , C. NFS version 4 protocol.

[24] S , D. *OpenGL reference manual: The official reference document to OpenGL, version 1.2*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[25] S , M., A -D , A., A -D , R. Evolving rpc for active storage. In *ACM SIGPLAN Notices* (2002), vol. 37, ACM, pp. 264–276.

[26] S , R., W , C., S , G., Z , E. Rapid file system development using ptrace. In *Proceedings of the 2007 workshop on Experimental computer science* (2007), ACM, p. 22.

[27] S , A., D , D., H , W., A , C., N - , M., P , G. Scalability in the XFS file system. In *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference* (1996), USENIX Association, pp. 1–1.

[28] T , D., L , M. Parrot: An Application Environment for Data-Intensive Computing. *Scalable Computing: Practice and Experience 6*, 3 (2005), 9–18.

[29] T , D., M , C., H , J. Chirp: A Practical Global Filesystem for Cluster and Grid Computing. *Journal of Grid Computing 7*, 1 (2009), 51–72.

[30] V , M., L , S., R , R., K , R., W , L., . Extending the POSIX I/O interface: A parallel file system perspective. Tech. rep., Argonne National Laboratory (ANL), 2008.

[31] W , S., P , K., B , S., M , E. Dynamic metadata management for petabyte-scale file systems. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing* (2004), IEEE Computer Society, p. 4.

[32] W , S. A., B , S. A., M , E. L., L , D. D. E., M , C. Ceph: A scalable, high-performance distributed file system. In *USENIX Operating Systems Design and Implementation* (2006).

[33] W , B. POSIX IO extensions for HPC. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST)* (2005).