Design and Implementation of WFS - A Web File System For Linux 2.2.1

Vishal Kathuria, Bruce Jackson
{brucej, vishal}@cs.wisc,edu

Abstract: *WFS is a new Linux filesystem which provides a filesystem like interface for the World Wide Web. It was developed as a kernel module for the Linux Kernel 2.2.1, and utilizes a user level process (web daemon) to service HTTP document fetch requests. The filesystem provides for caching of remote documents and can process multiple outstanding requests concurrently. As remote documents are fetched, content contained in hyperlinks within those documents is extracted and mapped into the local filesystem. This remote directory content information is maintained for each directory managed by WFS in a special file '...'. The utility* `lsw` *is used to list and manage the remote directory content. The partition managed by WFS is a read-only partition for WFS clients. However, clients are able to flush entries from the WFS partition using a special utility* `rwm`*. Our performance study shows that WFS is about 30% slower than AFS for file access traces composed of 100% cache misses. The lower WFS performance is probably due in part to the larger amount of processing pushed to the user level process in WFS, and the use of a general HTTP library for document fetches.*

1. Introduction

Web browsers provide an interface to public portions of remote file systems. In this paper we describe the implementation of a new Linux filesystem that provides a filesystem like interface to the web. This new filesystem, WFS, allows users to 'browse' the web in a manner similar to browsing a directory structure. WFS is mounted as a disk partition that acts as a cache for portions of the web that have been recently visited. Throughout this paper we refer to this partition as `/wfs/`. The remainder of this paper is organized as follows. In Section 2 we describe the overall architecture of WFS and discuss factors which influenced our design decisions. Section 3 describes in detail our implementation of WFS. The implementation of the user level process used by the kernel to service web lookups is described in Section 4. Some small additional tools (such as `lsw`, a command to list web content) are used in conjunction with WFS. The implementation and function of these tools is described in Section 5. We compare the performance of WFS to AFS in Section 6. Finally, we discuss our conclusions in Section 7.

2. Design

2.1 Semantics

There are semantic differences between a web file system and a traditional filesystem. These semantic distinctions influenced our design decisions. For instance, in a normal filesystem when a user issues the command '`cd foo`' and the directory `foo` does not exist, the command terminates and an error is returned. However, in WFS, the same command may result in the creation of a new directory `foo`. In general, the `/wfs/` partition represents the root of the entire internet, with each subdirectory representing a host that has been discovered and visited. The directory structure below these host directories represents the portion of the directory structure at that host that has been visited or is known to be publicly accessible (e.g. `/wfs/www.cnn.com/markets/`). This structure is dynamically created during normal use of

the filesystem. For example, when a user first encounters a directory (say `/wfs/`) and issues the command `ls`, there is no listing (a static listing of all known web sites that exist on the internet cannot be kept). If that user attempts the command '`cd www.cs.wisc.edu`', the directory `/wfs/www.cs.wisc.edu/` will be automatically created by the filesystem if (1) it does not exist, and (2) `www.cs.wisc.edu` is a reachable site on the internet.

Another semantic difference for web filesystems is that the `/wfs/` partition is a read-only public cache. Users are not allowed to directly create or modify files and directories within this partition. File creation is transparent to the user as remote files are accessed. The regular '`ls`' command shows the files and directories that are currently in the WFS cache. The WFS tool command '`lsw`' will show the directory entries that are known to exist on the remote sites but have not yet been brought into the cache. Although users do not have write permission in `/wfs/`, they are allowed to remove files from the `/wfs/` partition. A utility `rmw` is provided for this purpose.

2.2 Design decisions

2.2.1 Architecture

Figure 1 shows the high level architecture of WFS. The shaded boxes are components implemented as part of WFS, and the non shaded components show how the WFS fits in and interacts with the Linux kernel. The Linux kernel provides a Virtual File System (VFS) which is an abstract interface to all filesystems. Each filesystem provides its own implementation of the standard UNIX filesystem and inode operations (`create(), open(), lookup()`, etc..) and registers these with the VFS [3]. The Linux kernel resolves which function to call for a given system call based on which filesystem is being accessed.

The fundamental idea behind our design is to attempt a web lookup when a normal `lookup()` fails. This web lookup is handled by the web daemon (a user level process) on behalf of the kernel. We decided to push the web requests and subsequent response processing to a user level process in order to minimize the amount of code introduced into the kernel. Using a user level process to provide file system services is also done in other systems such as AFS [2]. Using this approach we were able to develop WFS as a modified version of the standard Linux filesystem EXT2 with relatively small changes to the kernel and EXT2. We implemented WFS as a kernel module. The required kernel, VFS, and EXT2 filesystem modifications are discussed in Section 3. The user level web daemon process is discussed in Section 4.

2.2.2 Cache Management

There are two issues involved in cache management: validation and replacement. The three popular validation policies are: (1) validate on every access (i.e. whenever a URL is accessed, send a request to remote server asking whether the remote copy is newer than the cached copy. If yes, fetch the new copy otherwise use the cached copy), (2) validate if the cached copy is older than a certain time period, and (3) validate on user request, similar to "reload" request in browsers. We chose to use (3) in WFS for two reasons: (1) performance, and (2) we decided to leave the policy decision regarding when to validate a cached copy to the user/application that is using WFS. A single cache validation policy applied globally is a bad idea. For instance, URLs of news sites might be validated at much shorter intervals than URLs such as the

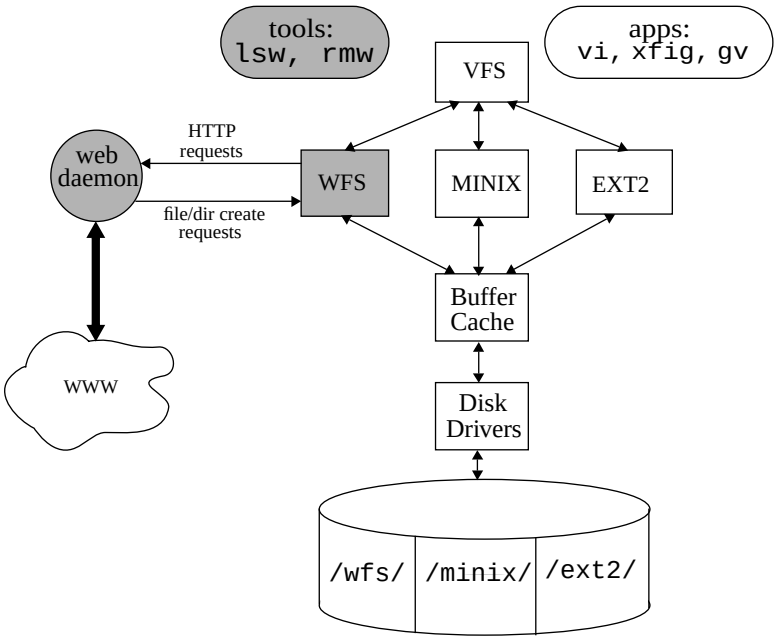

Figure 1. *Architecture of the WFS filesystem. Shaded boxes are components and tools for WFS. WFS is a modified version of the Linux filesystem EXT2. The web daemon is a user level process with root privileges and secure communication channels with the kernel. It services HTTP requests and modifies the partition `/wfs/` through WFS. It is only activated when a call to `lookup()` fails, in which case a web lookup is attempted.*

Linux Kernel Hackers Guide tutorials, or novels which need not be validated for months or longer.

Another cache management decision involves deciding what to do when the `/wfs/` partition is full. One option is to have the kernel periodically scan `/wfs/` and delete all files that have resided in the cache longer than a fixed time. However, this policy will produce an unpredictable filesystem interface to the user. For instance, a user may see a file using '`ls`' one day, only to have it mysteriously disappear by the time they next attempt to access it. Therefore, we decided to push the responsibility of cache space management to the user. When the `/wfs/` partition is full the user must make space just as he/she would if their AFS partition were full. The utility rmw is used to delete content from `/wfs/`. The semantics of '`rm /wfs/www.foo.edu`' will be to remove the directory `www.foo.edu` from the cache `/wfs/`. This design decision presents a problem since we must allow the user to delete files from the read only /wfs/ partition. To accomplish this we provide a utility, `rmw` to use in place of `rm` (Section 2.2.3)

2.2.3 Utilities

Apart from the utilities like `ls`, `cd`, `cat`, `vi`, `xv`, and `gv` for accessing the contents of a `/wfs/` directory, our design required the development of two additional utilities: `lsw` and `rmw`. The utility `lsw` will show the directory entries that WFS knows to exist on the remote sites. The `rmw` utility allows users to remove contents from the read-only `/wfs/` partition. The development of these tools is discussed in Section 5.1 and Section 5.2 respectively.

3. WFS Module

The WFS module is the same as the EXT2 module except for modifications to the `lookup()` function. When `lookup()` is called it does an EXT2 style lookup on disk. If it fails to find the `inode` for the requested path, it sends a message to webd requesting it to fetch that file from the web.

3.1 How lookup() Services a Request

The method `lookup()` is given an `inode` of the directory in which a file is to be looked up and the `dentry` of the file (See Appendix A for a description of struct `dentry`). For example, the function call `fopen("abc/def","r")` will lead to first calling of `lookup(inode_wd,dentry_abc)`. This `dentry` of `abc` contains all the information about `abc` except the inode (which is why the lookup is called).

After `lookup()` supplies the `inode` of `abc`, it is called again to find the `inode` of the next piece of the path, `lookup(inode_abc, dentry_def)`. Thus even if a long path is supplied to `fopen()`, `lookup()` resolves it one piece of the path at a time. Therefore we will concentrate on how a single invocation of `lookup()` works.

Consider a case when a user enters a command cat /wfs/www.cs.wisc.edu/csl/faq/unix/index.html. Assume that the directory `/wfs/www.cs.wisc.edu/csl` is already cached in `/wfs/` so `lookup()` will proceed normally until `faq` is encountered. In that case `lookup(inode_csl, dentry_faq)` is called. Since there is no entry `faq` on disk, a web fetch request is initiated. The `lookup()` function must generate a URL to supply to the web daemon.

3.1.1 Construction of the URL from a dentry

For constructing the URL for `faq`, `lookup()` uses the `dentry` for `faq` provided to it. The struct `dentry` has the following fields of interest to us.

```
struct dentry {
        struct dentry * d_parent;       /* parent directory */
        struct qstr d_name;             /* name of the entity to which this
                                        dentry belongs */
      .
}
```

`lookup()` continues following the parent pointers to walk through the dentries of all the components of the path leading to `faq`. It reads each component name and prepends it to the URL string. When the root is reached, the string "`http://`" is prepended to the URL string and the URL string is sent to webd.

The VFS cache maintains chains of dentries such that if there is a dentry for file `f.x` in the cache then the dentries of all the components of the path starting from the root of the filesystem to the file `f.x` will also be found in cache. This ensures that `lookup()` will never encounter a null `d_parent` and the above scheme will always work.

3.2 WFS Module Implementation Details

3.2.1 Secure Message Passing Channels

Our design requires a means for communication between the kernel and the user level process webd. This channel must be a secure channel which only the kernel and webd are able to access. We implemented this bi-directional channel using 2 message queues. Message queues are created by the kernel using the system call `msgget()`. These channels can be assigned permissions in a similar way to assigning read and write permissions to files. In this way we were able to restrict access to these queues to only the kernel and webd. Communication is accomplished through the calls `msgsnd()` and `msgrcv()`. Message queues are destroyed using the call `msgctl()`. Each of these calls has an internal system call of the form `sys_msgget()`, etc.. We modified the kernel to provide these functions to the WFS module in the following two ways. First, we exported these symbols from the kernel so that modules could access them (by modifying `ksyms.c`). Second, we modified the functions `sys_msgsnd()` and `sys_msgrcv()` so that the kernel could call them to pass messages to webd. By default in Linux 2.2.1, these functions assume a user level process is calling them and they try to copy parameters from the user space to kernel space. When the kernel makes these calls, it does not need to do this copying because these parameters are already in kernel space.

3.2.2 VFS Modifications

We made minor modifications to VFS to keep the VFS directory cache (which contains dentries) consistent. All the HTTP related work of WFS is done inside the `wfs_lookup()` procedure which is called by `real_lookup()` of VFS. `real_lookup()` creates a new `dentry` and links it into the directory cache. This `dentry` is then passed to `wfs_lookup()` which is expected to set the `inode` field of the struct `dentry` to the `inode` of the requested file or to null (in cases where the file does not exist). If `wfs_lookup()` cannot locate the file on disk, it sends a message to webd and blocks, waiting for a response from webd. The webd fetches the file from the internet and creates the corresponding file in the `/wfs/` partition. During the process of creating this file, a `dentry` for this file is created in the VFS cache. There is another `dentry` for this file that was created by `real_lookup()` and given to `wfs_lookup()` (as described earlier). In order to maintain consistency of the cache, only one of these entries should be kept so we decided that the entry created by `real_lookup()` would be purged. `wfs_lookup()` cannot do the purging because `real_lookup()` still has a pointer to this entry. So `wfs_lookup()` returns an error code, informing `real_lookup()` that it should get rid of the `dentry` it has a pointer to. `real_lookup()` (which is part of VFS) was modified so that when `wfs_lookup()` returns error code 44, `real_lookup()` calls the `dput()` method to purge the `dentry` it created and calls `cached_lookup()` to look for the entry created by webd. `real_lookup()` returns that `dentry`.

4. The Web Daemon

The web daemon (webd) is responsible for processing web lookups on behalf of the kernel. On receiving a request from the WFS module does an HTTP request to fetch the file from the internet. If the

file is successfully fetched, it is stored in the cache (the `/wfs/` partition) and success is reported to WFS. Otherwise a failure is reported.

4.1 Action Inside Web Daemon

Continuing the example from the previous section, webd receives a message from `lookup()` to attempt a web lookup for "`http://www.cs.wisc.edu/csl/faq`". The webd does not know whether "`faq`" is a file or a directory on the remote site. It sends out HTTP request for "`http://www.cs.wisc.edu/csl/faq`" to the remote httpd. If "`faq`" is a directory, httpd sends the following message back in response to the HTTP request.

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<HTML><HEAD>
<TITLE>301 Moved Permanently</TITLE>
</HEAD><BODY>
<H1>Moved Permanently</H1>
The document has moved <A HREF="http://www.cs.wisc.edu/csl/faq/">here</A>.<P>
</BODY></HTML>
```

This message informs webd that `faq` is a directory and webd creates a directory corresponding to `faq` in `csl` directory. It also sends another request for the URL "`http://www.cs.wisc.edu/csl/faq/`" (notice the / at the end of "`faq`"). This request will result in a HTTP response that contains an `index.html` file as the response body.

This `index.html` response may be a real file in a remote directory or a generated `index.html` file that contains the remote directory contents. In either case, `index.html` contains information about the remote directory structure. The web daemon parses `index.html`, extracts links, and uses this information to construct a remote content file (...) for each directory. This remote content file will be used by the directory listing utility `lsw`, which is described in Section 5.1.

4.2 Implementation

4.2.1 Secure Communication with the Kernel

Message queues were chosen as the communication channel between the webd and the WFS module (henceforth referred as ‘the module’). Whenever the web daemon is started, it sends a message on the queue reporting its PID (process ID). The module always checks the queue for pending messages from the daemon before sending a message to the webd. If there is a message then the PID is stored in a variable by the module. The module uses this PID to check whether there is a webd running on the system. This is done to report an error if the webd goes down webd. In this case WFS should revert to EXT2 functionality instead of just hanging.

4.2.2 HTTP document fetching

For the fetching of the documents, we decided to use libwww [6] instead of attempting to imple-

ment our own HTTP code, or modifying HTTP capable utilities like wget [5]. Using a general purpose library such as libwww usually has some performance overheads, but also makes the implementation cleaner and more extensible. For example, extending WFS to handle ftp requests would be straightforward with webd using libwww. libwww also has support for parsing html documents and multithreading. Unfortunately the libwww documentation was so poor that we decided not to use the libwww html parser to scan "index.html" documents for links. Instead we implemented our own link scanning code, which is probably much faster than a general purpose html parser. Nevertheless, the HTTP request and response facilities of libwww were very useful in the development of WFS.

4.2.3 HTTP Response Processing

An HTTP request issued by webd will result in some type of an HTTP response. The web daemon processes the response based on the HTTP status code. If a status code other than 301 (re-direct), or 200 (OK) is returned, webd sends the kernel a message that the web lookup failed. The original call to `lookup()` will then fail. A status 301 code indicates redirection and results in webd creating a directory in `/wfs/` corresponding to the URL. A status 200 response indicates that the HTTP request was successful, and the response body contains the requested resource. In this case, webd responds in 1 of 2 ways. If the response does not correspond to an `index.html` file, webd simply creates a file in the `/wfs/` partition and writes the HTTP response body to that file. If the response corresponds to an `index.html` file (the response returned after a 301 re-direction is the one and only time that an `index.html` file is returned), additional processing is done before creating the local file. In particular, the `index.html` is scanned for links using our link scanning parser. For each link that is found, webd determines if that link contains information about the remote site that should be mapped into the local `/wfs/` partition. For each link that contains useful information, and entry is made into the special file '...'. The file... exists for each directory in `/wfs/`. It can be though of as a database of known remote resources that exist for a given directory. This file is used and modified by the wfs tool `lsw` (Section 5.1). Entries into this database are of the following form:

```
resource_type$discoverytime$resource_name
```

For instance, if an `index.html` response contains a link such as `<A HREF = "foo.html">`, the following entry would be created in ... :

```
0$Apr 28 05:14  $foo.html
```

There may be many links in an HTML document that are irrelevant to WFS. For instance, the links <a href = "mailto:bj@aa.bb.com">, <A HREF="#A">, <A HREF="?N=D"> do not point to remote documents or directories. The link scanner ignores these ireelevant links, and makes no entry in ... for them. If an HTML document contains multiple links to the same resource, a single entry is made in ... for that resource.

5. WFS tools

5.1 lsw

The WFS tool `lsw` is used to maintain and display known remote information about directories in `/wfs/`. This information is stored in a special file '...', which is created each time a directory is created anywhere in `/wfs/`. Directories are only created in `/wfs/` as a result of fetching a remote directory. When an `index.html` file is fetched and the resulting directory is created, the `index.html` file is scanned to extract all links it contains. These links are examined, and any links that reveal subdirectory information are mapped into '...' in a fixed, simple to parse format. The command `lsw` scans this file, removes duplicates (and entries that have been brought into the cache), and displays the results. The following example illustrates the use of `lsw`. Assume that the directory `/wfs/www.cs.wisc.edu/~brucej` has been created through normal use of WFS. When the directory `~brucej/` is created, the `index.html` file corresponding to that remote directory is scanned for links. Assume it contains the following links:

```
<A HREF = "http://www.cnn.com/world_headlines.html"> headlines </A>
<A HREF = "http://www.cs.wisc.edu/csl/">useful info</A>
<A HREF = "http://www.cs.wisc.edu/~bart/736.html">advanced os</A>
<A HREF = "http://www.cs.wisc.edu/~brucej/foo.html">foo file</A>
<A HREF = "http://www.cs.wisc.edu/~brucej/bar/">bar dir</A>
<A HREF = "resume.ps">resume</A>
<A HREF = "xml/">xml data files</A>
<A HREF = "http://www.cs.wisc.edu/~brucej/mountain.gif">Mt. St Helens</A>
```

These links would be stored in `/wfs/www.cs.wisc.edu/~brucej/...` as follows:

```
0$Apr 28 05:14  $foo.html
1$Apr 28 05:14  $bar
0$Apr 28 05:14  $resume.ps
1$Apr 28 05:14  $xml
0$Apr 28 05:14  $mountain.gif
```

The date is the `index.html` fetch time (i.e. the remote information discovery time), the first column is either a 0 for a remote file, or a 1 for a remote directory, and the final field is the remote file name. Fields are delimited by dollar signs. Notice that only links that point to files within the remote directory's sub-tree are stored. We also considered creating symbolic links for each link pointing outside this sub-tree, but decided against it because of the large number of symlinks that would be generated. When the command `lsw` is executed while in `~brucej/`, the following screen output is generated:

```
root@crash2:/wfs/www.cs.wisc.edu/~brucej}-> lsw
[web] -r--r--r--   wedb   Apr 28 05:14  foo.html
[web] dr--r--r--   wedb   Apr 28 05:14  bar
[web] -r--r--r--   wedb   Apr 28 05:14  resume.ps
[web] dr--r--r--   wedb   Apr 28 05:14  xml
[web] -r--r--r--   wedb   Apr 28 05:14  mountain.gif
```

This information represents only the remote uncahced files and directories that are thought to exist in `http://www.cs.wisc.edu/~brucej`. Once a file has been fetched from the remote site (for instance through the command '`vi foo.html`'). The remote information entry is removed from '...' and that file will appear only through a normal '`ls`' command (i.e. the output from '`lsw`' will not show `foo.html` but '`ls`' will after is has been brought into the cache).

5.2 `rmw`

The tool `rmw` is a simple command owned by the root with the setuid bit set. This command allows users to delete any and all files in `/wfs/` (they do not have write permission for these files). Users of WFS must be able to do this because each user must manage his/her own space (see Section 2.2.2). The use of this tool is restricted to the `/wfs/` partition in order to prevent users from deleting anything they wish in other filesystem partitions.

6. Performance

For a study of the performance of WFS, AFS is a good comparison. AFS and WFS have many things in common such as accessing remote files, caching of whole files on access, and the use of a user level process for servicing requests (Venus or afsd for AFS [2] and webd for WFS). In addition, the homepages of users in the CS department are accessible from both WFS (via HTTP) and AFS. For example vishal's homepage is accessible as `http://www.cs.wisc.edu/~vishal` through HTTP. The same file can be accessed from AFS as `~vishal/public/html/index.html`. Our aim was to measure the CPU time taken when the same set of file accesses is performed using AFS, and then using WFS.

For this purpose we wrote a benchmark (bm) which flushes the cache and then accesses a set of files. This process is repeated in a loop a fixed number of times and the time taken to execute the filesystem access string is measured. Our first benchmark measures performance when none of the files being accessed are in the cache. By removing this flushing step in the benchmark, we could get performance statistics for the case when the accessed files are found in the cache. We are continuing to collect this data and the results will be reported in our final report.

Because of limitations imposed on us by system administrators, we were forced to try to compare the performance of the two systems using two very different machines. We had to use two machines because crash2 (where WFS is developed and installed) is on the unsup network and is not allowed to run an AFS client. Although we had access to a Linux machine with AFS access (parrolles), we were not allowed root permissions on parrolles, so we could not run WFS on it. Therefore we performed AFS tests on parrolles and WFS tests on crash2. parrolles is a 200MHz Pentium Pro with 64 MB RAM running Red Hat Linux 5.2 Kernel 2.0.36. crash2 is a 90MHz Pentium with 160MB RAM running Red Hat Linux 5.2 Kernel 2.2.1. These are clearly very different machines. We attempted to measure the performance ratio between these two machines by measuring execution time of our benchmarks on a filesystem common to both (EXT2). This ratio was used to estimate the performance of WFS on parrolles after collecting actual performance data on crash2.

6.1 Measuring CPU time

Most time data was collected using the time command and the times system call. We could not do this for afsd because it is a process started by root. We used the `top` command instead for afsd. `top` shows, the amount of CPU time that has been consumed by a process since it started running. We recorded this value for afsd just before running our benchmark and just after we finished running it. The difference was used as the CPU time spent by afsd in serving the requests generated by the benchmark.

6.2 Speed Comparison of Parrolles and Crash2

As discussed above, to be able to compare AFS and WFS performance, the ratio of the speeds of parrolles and crash2 was needed. We accomplished this by running the same program on parrolles and crash2 and taking the ratio of CPU time taken.

Figure 2. *Relative speed of parrolles and crash2 for a file system intensive application*

The problem is that this ratio is not independent of the program which is being run. We decided to modify the benchmark we were using for AFS and WFS to measure EXT2 performance on both machines. For this, the files accessed by the bm were copied to /tmp and EXT2 was used to access those files. Figure 2 shows the time taken by bmext2 (benchmark modified for EXT2) on parrolles and crash2. Each data point is the average total time taken for 5 runs of the same benchmark for a given value of n where n is the number of loop iterations done in the benchmark. The slope of this line gives us an estimate of the ratio of speeds of the two computers (henceforth called R_s = Speed of parrolles/speed of crash2). The value of R_s yielded by this figure is 4.04. The appendix contains the entire data set used to construct Figure 2. In order to see how R_s might differ for different applications, we also collected data running the program top on both machines. The results of that experiment are summarized in Figure 3. In this case, the value of R_s=3.15. This is smaller but in the same range of the value obtained for running the benchmarks on EXT2.

Figure 3. *Relative speed of parrolles and crash2 for an application that is not file system intensive (top)*

We believe that the EXT2 obtained value is more accurate for comparing WFS to AFS because it is based on filesystem intensive application, whereas top makes very few file system accesses.

6.3 Performance Data for AFS and WFS

The performance metric we use is CPU time. The purpose of the benchmark is to continuously attempt to access files that are not in the cache to measure the amount of time taken by each filesystem to service the fetch. We have concentrated on this area for performance since this is likely to be the most costly operation for these two filesystems. The pseudo-code for the benchmark is as follows.

```
repeat n times {
        1. flush cache

        2. for each file in a collection of files: {
              Open the file
              Read first byte of the file
              Close the file
           }
}
```

We ensured that the collection of files had a reasonable mix of document types (such as like html, ppt, ps, gif, jpg, directories etc). The time measurements we recorded as the CPU time is measured as follows.

```
Total CPU time = CPU time for benchmark + CPU time for the daemon
```

Our initial data demonstrated WFS to be far slower than AFS. We used the times system call to

find out the portions of WFS which were taking most of the time. This led us to make some optimizations in WFS, including moving some library initialization and cleanup code out of the main loop in webd. After making these changes, we again collected data. This time WFS appeared to be 50% faster than AFS (when the speed ratio was factored in). We were very happy, but skeptical of our results. The elation soon turned into suspicion and we turned our attention to discovering why AFS appeared to be slower than WFS. We discovered that a considerable amount of CPU time was being spent in flushing the AFS cache. Since flushing the AFS cache is functionally a much more complex operation than flushing the WFS cache, we decided that it was unfair to include cache flushing time in our data and decided to record only the file access times. The raw data we collected using this scheme is presented in Table 1. Figure 4 shows the result graphically. The different data points were obtained by changing the value of the loop counter.

Table 1. *Raw performance data for AFS (measured on parrolles) and WFS (measured on crash2)*

Number of iterations per benchmark run	AFS average afsd time	AFS average benchmark time	AFS average total time	WFS average webd time	WFS average benchmark time	WFS average total time
100	1.2	3.934	5.134	23.316	2.298	25.614
200	2.2	7.516	9.719	43.664	4.454	51.118
400	4.4	15.102	19.502	94.632	8.884	103.516

Using the speed ratio obtained from the EXT2 bm study on parrolles and crash2, we normalized the total time values for the benchmark data obtained running WFS on crash2 to what we estimate they would be if we were able to run WFS on parrolles. The result is shown in Table 2.

Table 2. *AFS benchmark times compared to WFS benchmark times with WFS data normalized using the relative speed ratio of parrolles to crash2*

Number of iterations per benchmark run	AFS average total time	WFS total time normalized using EXT2 data	WFS performance penalty
100	5.134	6.331	23.3%
200	9.719	12.635	30.05%
400	19.502	25.587	31.20%

6.4 Results

On an average, the time taken by WFS was about 30% more than AFS. We also made a few other observations

Figure 4. *Performance of WFS compared to AFS with WFS performance normalized*

- On an AFS access that missed the cache, most of the time is spent inside the AFS module and only a fraction of that time is spent in afsd. This means that AFS designers decided to build most of the functionality of afs in afs module.
- Most of the time in a WFS cache miss is spent in webd.

Every time the daemon is invoked, WFS has to pay a price for communication and task switch. Since a big part of AFS functionality is built into the kernel module itself, our feeling is that AFS is not invoking afsd very often and many of the requests are satisfied by the module itself. This may be part of the reason why AFS performs better than WFS.

7. Conclusions (bruce)

Our implementation of WFS succeeded in producing a filesystem interface to the World Wide Web. The initial performance results indicate that WFS performs significantly worse than AFS (~ 30% slower). We believe this is due to 2 main factors. First, we pushed much of the functionallity out into the user level process, webd. This causes a lot of overhead due to context switching and kernel boundary crossings. We made this design desision in order to limit the amount of code introduced into the kernel, and payed the performance penalty for it. Second, we used a general purpose library (libwww) for servicing our remote document fetches. This decision was again made in order to simplify the implementation. Although the performance is slower than AFS, WFS performs reasonably well considering that AFS is a

very mature system.

8. References

[1] *The Linux Kernel,* version 0.8-3 by David A. Rusling

[2] Satyanarayanan M et al, *The ITC Distributed File System: Principles and Design*

[3] *The Linux Kernel Hackers' Guide*, version 0.7 by Michael K. Johnson

[4] *The Linux Kernel Module Programming Guide*, version 1.0 by Ori Pomerantz

[5] *GNU Wget Manual*, http://www.gnu.org/manual/wget/index.html

[6] *Libwww - The W3C Protocol Library*, http://www.w3.org/Library/

Appendix A

To come in the final document...