Cookfs

New VFS driver for standalone binaries

Wojciech Kocjan, BitRock S.L.

http://sourceforge.net/projects/cookit/

Abstract

Cookfs is an open source VFS for Tcl written in C and Tcl. It is designed to achieve better compression ratio compared to Mk4 VFS and ZIP archives. It also offers configurable compression algorithm – currently zlib and bz2, with plans for lzma in the future.

It's also designed to be used in Tclkit-like applications. Cookit project shows how cookfs can be used for this purpose.

Introduction

Historical background

Tcl is the first (and probably the only) language that has a solution for providing Tcl runtime and Tcl/Tk based application as single executable file. This requires embedding additional files inside the executable in a way that the Tcl I/O mechanism can access them. Tcl 8.3 supported this functionality internally and version 8.4 started exposing Virtual FileSystem (VFS) API and allowed creating additional VFS drivers that provided means of accessing files – listing them, getting/setting information and opening for reading and/or writing.

This brilliant move allowed a way for multiple solutions such as TclPro's prowrap[1] binary, TclKit[2], freeWrap[3] and eTcl[4] to offer standalone Tcl interpreter binaries. These solutions also provided mechanisms for embedding Tcl/Tk based applications by putting all scripts and resource files into a single binary.

These technologies base on either ZIP archive format (prowrap, freeWrap and eTcl) or Metakit archive (TclKit). ZIP was chosen as it is a standard for storing one or more files in a single archive. Metakit is a complex, but efficient generic database engine that can be easily

---

[1] http://www.tcl.tk/software/tclpro/doc/TclProUsersGuide14.pdf

[2] http://www.equi4.com/tclkit/

[3] http://freewrap.sourceforge.net/

[4] http://www.evolane.com/software/etcl/index.html

adapted to store multiple files in a single archive. Metakit based solution is called Mk4tcl and aside from TclKit it was also adopted by commercial offerings such as Tcl Dev Kit[5] from ActiveState.

Why cookfs?

Cookfs is a similar solution – its main goal is being able to store multiple files in a single archive, focusing on standalone binaries and packaging of large Tcl applications as a single file. The main difference between approaches is that cookfs is focused on creating an efficient solution to specific problems – packaging Tcl scripts and various types of payload.

Tcl applications can contain large number of small files – around 1-5 kB. Mk4vfs and ZIP based solutions compress each file individually. Both of these filesystems use zlib for compression. Cookfs uses the same algorithm by default, but also supports bzip2.

Cookfs compresses large files individually, but groups and packages small files together. This can greatly reduce size for files such as `pkgIndex.tcl` files that usually have very similar contents. Cookfs also groups such files by their name so all `pkgIndex.tcl` files are usually compressed in the same chunk.

Cookfs can also be used to package additional payload that is not related to Tcl – such as installers that might store very big files, but can also contain small files. Of course all other solutions also allow you to do this, but cookfs stores this kind of data more efficiently due to file grouping. One example might be packaging the Joomla[6] application on Linux along with all of its prerequisites – Apache, PHP and MySQL. Such payload consists of 9000 files, 4000 of which are smaller than 1 kB and another 3000 files which are smaller than 5kB. Compressing such files individually is inefficient and therefore cookfs provides much better results at compressing such data, which is shown later in the document.

A very important benefit compared to other solutions, especially mk4vfs is that cookfs offers better memory management. Cookfs does not consume more than specified amount of memory when reading files from an archive. It is also possible to copy files directly to archive and avoid storing them in memory. This makes reading from and writing to an archive memory efficient even for very large files – such as hundreds of megabytes.

Cookfs also offers ability to choose between lower memory consumption and faster read access – configuring page cache can speed up reading process at the cost of more memory depending on which is more important for a particular application.

---

[5] http://www.activestate.com/tcl-dev-kit

[6] http://www.joomla.org/

How and where to get it?

Both cookfs and cookit projects are hosted on SourceForge as cookit project and is available at http://sourceforge.net/projects/cookit/.

Cookit binaries for Microsoft Windows, Linux and Mac OS X can be downloaded from SourceForge files section. They are located in cookit folder: http://sourceforge.net/projects/cookit/files/cookit/. Cookfs source code can be found in cookfs folder: http://sourceforge.net/projects/cookit/files/cookfs/. Latest source code can also be found in cookit Subversion repository under `cookit` and `cookfs` folders respectively.

Cookfs can either be used from a standard Tcl distribution – such as ActiveTcl – or using cookit standalone binaries. In the latter case, cookfs is already loaded in the main interpreter and nothing needs to be done.

For any other case, cookfs needs to be built from source code and installed. The build procedure follows well known standards:

- we need to run configure script first, providing `--prefix` and `--with-tcl` flags that provide path to installation and where Tcl libraries can be located; currently required Tcl version is 8.6 and configure script will mention this if older version is provided; in addition to this, tclvfs package is also required – typically it is installed with all ActiveTcl distributions
- next step is to run `make` to compile the package
- finally `make install` command will install cookfs to specified Tcl distribution

Using cookfs

Now that cookfs is available, using it is very similar to other virtual filesystems. In order to load it all that is needed is to run:

```
package require vfs::cookfs
```

Mounting an archive is as easy as doing `vfs::cookfs::Mount`, specifying path to archive and Tcl's local mount point. Cookfs archives often end with `.cfs` suffix. Both locations can be the same. For example:

```
vfs::cookfs::Mount /tmp/archive.cfs /tmp/archive.cfs
```

Next we can perform operations on */tmp/archive.cfs* and all subdirectories and those will be performed on cookfs archive. The command returns `fsid` (filesystem identifier), which can be used by other cookfs commands.

In order to unmount it we need to call `vfs::unmount` providing local mount path. For example:

```
vfs::unmount /tmp/archive.cfs
```

All typical operations such as file copying work without any issues. For example we can open a file for writing, store something in it, close it and read it afterwards. Unmounting the archive and mounting it again using same or different Tcl process will also allow us to read the file. In generic, cookfs archives can be read from multiple processes or threads, but only one process or thread can write to a cookfs archive at a time.

Please note that due to nature of cookfs it is essential to unmount it before exiting – otherwise index information will not be stored and archive will not be readable.

Additional options can be passed while mounting an archive. These options have to be specified before archive name and local mount point. The syntax is as follows:

```
vfs::cookfs::Mount
    ?-readonly? ?-compression none|zlib|bzip2?
    ?-pagesize bytes? ?-pagecachesize count?
    ?-smallfilesize bytes? ?-smallfilebuffer bytes?
    archive local
```

- flag `-readonly` tels cookfs to mount an archive as read only and is needed if a file cannot be written to – such as in standalone binaries. For example:

  ```
  vfs::cookfs::Mount -readonly \
      /tmp/archive.cfs /tmp/archive.cfs
  ```

- type of compression to use can be specified using `-pagesize`. Currently three values are accepted – `none`, `zlib` and `bzip2`. First does not perform any compression, second uses zlib library and last uses bz2 library. The default is `zlib`, which is very fast and offers good compression ratio. It is also used by ZIP and Mk4 based virtual filesystems.

- option `-pagesize` specifies maximum size of a page. It defaults to 256kB and should not be changed by default. Details on pages are described in next sections. This only affects files that will be written to archive as files already stored in the archive will not be modified in any way.

- number of pages that will be cached can be specified using `-pagecachesize` option. It defaults to 8, which means maximum of 2MB memory usage. In order to speed up reading from archive it can be increased. For example:

  ```
  vfs::cookfs::Mount -pagecachesize 64 \
      /tmp/archive.cfs /tmp/archive.cfs
  ```

- options `-smallfilesize` and `-smallfilebuffer` specify maximum size for small files as well as buffer for queuing small files' writes to disk. This mechanism is described in more details in next sections.

Cookfs provides feature to store all changes in a filesystem in memory. Command `cookfs::writetomemory` enables writing changes to memory. It accepts `fsid` parameter, which is returned by `vfs::cookfs::Mount` command. For example:

```
cookfs::writetomemory $fsid
```

This option can also be enabled by specifying `-writetomemory` flag when mounting an archive – for example:

```
vfs::cookfs::Mount -writetomemory \
    /tmp/archive.cfs /tmp/archive.cfs
```

When this option is enabled – during or after mount – any changes to this archive are not saved to disk. They are discarded whenever archive is unmounted or process exits. This can be used for running legacy scripts that require writing data to its relative location or as temporary file storage.

It is also possible to use a separate file for saving changes to an archive. Command `cookfs::aside` can be used for this and accepts `fsid` followed by path to file to store changes. For example:

```
cookfs::aside $fsid ~/.myapplication/updates
```

Changes in aside file persist across mounts of a filesystem and therefore can be used to store updates to standalone binaries or archives stored on read-only media such as CD or DVD disk. Aside file does not have to be specified at mounting time – it is possible to specify aside file after archive is mounted and changes in aside file will be visible immediately.

In current cookfs version only creating aside archive for read-only archives works properly. In order to create an aside file for read-write archives it is recommended to do:

```
vfs::unmount /path/to/archive.cfs
set fsid [vfs::cookfs::Mount -readonly \
    /path/to/archive.cfs /path/to/archive.cfs]
cookfs::aside $fsid /path/to/aside-changes.cfs
```

Cookfs provides a function for copying files directly to archive – this is mainly meant for performance issues. Command `cookfs::writeFiles` takes `fsid` returned by `vfs::cookfs::Mount` followed by list consisting of 4 elements for each file – path relative to archive, source type, data and file size. It bypasses Tcl I/O mechanisms for writing to an archive and uses cookfs functionality directly for faster writes. Currently only two types of source – `file` and `data`. For the first one, data specifies name of the file to copy from, for second one data is binary data that should be used as file contents.

For example in order to copy `/etc/passwd` as `passwd` and add a text file `textfile` to archive, we can do:

```
cookfs::writeFiles $fsid \
```

```
    passwd file /etc/passwd {} \
    textfile data {Hello world}
```

This will create those two files within cookfs archive and this operation will bypass writing those files to VFS layer.

Cookfs is used for storing Tcl libraries and scripts for standalone applications inside cookit. It is a proof of concept of using cookfs filesystem to create standalone binaries, similar to Tclkit. Cookit is described in more details in next sections.

Cookfs archives can be built using TABS (Tcl Automated Build System)[7]. It is a tool for building Tcl standalone applications and/or archives based on set of targets and predefined jobs. The wrap job by default uses mk4vfs archives, but providing -driver cookfs flag allows building cookfs based archives. For example:

```
target sample.cookfs -body {
    runjob wrap \
        -output sample.cookfs \
        -driver cookfs \
        -copy [list sample.vfs .]
}
```

This defines a target called `sample.cookfs` that creates cookfs archive by copying contents of `sample.vfs` directory to it. We can also use it to build cookit-based binaries by doing:

```
target sample.exe -body {
    runjob wrap \
        -binary cookit.exe \
        -output sample.exe \
        -driver cookfs \
        -copy [list sample.vfs .]
}
```

Tutorial on using TABS can be found at http://www.endorser.org/blog/tcl/tools/21-tabs.

Cookfs also introduces an idea of packages. These are regular cookfs archives that store libraries or packages in its `lib/` subdirectory. Currently tools such as TABS can be used to automatically add such a package by doing:

```
target sample.exe -body {
    runjob wrap \
        -packages pkg/sqlite3-win32.cfspkg \
        -binary cookit.exe \
        -output sample.exe \
        -driver cookfs \
        -copy [list sample.vfs .]
```

[7] http://sourceforge.net/projects/tabs/

```
}
```

With current solutions such as Mk4 VFS adding one or more packages requires mounting and copying it at VFS level. With cookfs design it is possible to copy entire archive by appending compressed data without decompressing and recompressing files. The only action that needs to be done is to add appropriate items to updated archive's index. When cookfs will provide APIs to merge archives packages will allow much easier creation of applications and will speed up process of building large binaries by reusing already compressed building blocks.

Overview and design

Cookfs consists of three basic elements – pages, file index and VFS layer.

- pages mechanism stores actual file contents and provides storage for filesystem index. It is the lowest layer of the filesystem. It allows storing multiple pages (chunks of information) along with compressing them. It provides API for adding a page, getting a page and storing index information by other parts of cookfs.
- File index provides means for importing and exporting file structure. It also provides API for getting, setting file information and listing file structure. Indexes also keep reference of pages that are used for storing a specified file.
- VFS layer provides way to access cookfs archive contents as regular files in Tcl and from Tcl's C API. It leverages pages and file index for storing file contents and its metadata.

Cookfs and file grouping

Ability to perform smart file grouping is one of most important features of cookfs. Whenever a file is to be added to a cookfs archive, its size is checked if it is smaller than small file limit. Threshold for files that should be grouped can be specified by `-smallfilesize` flag when mounting a cookfs archive and defaults to 32kB, which will be used for all examples in this section. Also, page size will be assumed to be 256kB.

Files that are larger than specified limit use up 1 or more pages to store only specified file. For example when writing a 100kB file, its contents will be stored in a newly created page and this page will only store contents of this file. Uncompressed size of this page will also be 100kB, however compressed page size will vary on compression ratio. As soon as file is written, information on this file is added to cookfs index – specifying page index and size of data in this page.

For a 1200kB file, its contents will be split in a total of 5 pages – storing 4 pages of 256kB and last page will store remaining 176kB. This information is added to index – specifying 5 pages, their order and sizes of each page.

For each small file that gets added to an archive, it is first added to a queue of small files – it is not immediately stored in cookfs archive. Files waiting on small files queue are written to disk when cookfs archive is about to be closed or queue has reached its size limit. By default queue size limit is 4MB, which means that after total size of all files in the queue exceeds 4MB, they are written.

The algorithm for grouping small files in pages in pseudo code below:

```
# initialize empty list of files for next page be added
set filelist [list]

foreach file [lsort ... $pendingFileQueue] {
  set newFilelist [linsert $filelist end $file]
  if {[pagesize $newFilelist] > $pagesize} {
    # add current list of files to page
    addNewPage $filelist

    # update indexes
    updateIndex $filelist

    # initialize empty list of files for next page
    set filelist [list $file]
  }  else  {
    # new file list does not exceed size, keep it
    set filelist $newFilelist
  }
}

If {[llength $filelist] > 0} {
    # add current list of files to page
    addNewPage $filelist

    # update indexes
    updateIndex $filelist
}
```

First files are sorted by their file name and then added to archive – called `pendingFileQueue`. List of files (`filelist`) to be added to next page is also created as empty list. List of all pending files (`pendingFileQueue`) is then iterated on.

If size of file currently iterated over (`file`) combined with `filelist` is larger than page size, files in current `filelist` are added as new page – cookfs indexes are updated and `filelist` is set to only file currently iterated over.

After iterating over `pendingFileQueue` is complete, if there are any files in `filelist`, they are added as new page.

Whenever a page is created – during iteration or afterwards – cookfs index gets updated to reflect that particular file's contents is now stored in specific page at specific offset and having specific size.

Cookfs pages

Cookfs archive stores contents of all files in pages. Each page is a chunk of binary data and cookfs consists of any number of pages followed by cookfs metadata – information on size and checksum of each page and index binary data. Limit for a single cookfs page size is 2GB (2^{31}). Limits of page size are specified when mounting a cookfs archive and are enforced in VFS layer. When specifying page size to an existing archive, existing pages will not be affected by newly set up limits.

The concept of pages is very generic – it allows the following operations to be performed on it:

- `add` *`binaryData`* – add specified data as new page and return its index
- `get` *`pageIdx`* – get data of specified page
- `length` – returns number of pages
- `index` – get cookfs index
- `index` *`newData`* – set cookfs index
- `delete` – save all changes and delete specified page object

Commands above are actual Tcl methods that can be invoked on pages. While these are not all the operations available, these are the ones needed for cookfs VFS layer to work.

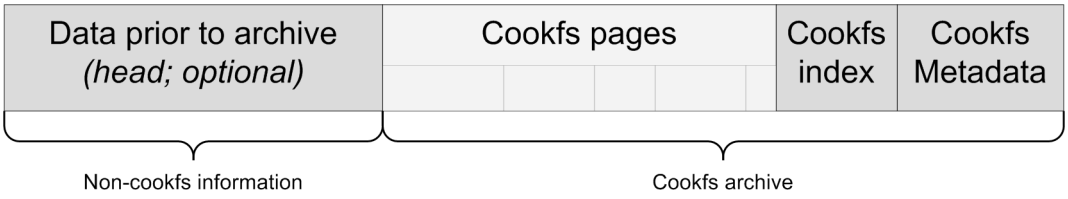
Page indexes are integer numbers starting from 0 up to `[length]-1`. Invoking `add` returns new page index, which can then be used to `get` pages. Pages can only be added – they cannot be deleted or changed. At first glance it may strike you as major drawback, but in fact real-life experience clearly indicates that the user far more often adds files to the archive than removes them. The assumption is that if cookfs archive has multiple updates it is better to create it from the beginning, which is also explained in next section.

Index information is a binary data that is stored in cookfs metadata. It is handled by separate part, which is able to import binary index data and export it when writing to a cookfs archive. Additional information on cookfs indexes can be found in next section of this document.

All pages are stored as binary data and are not modified in any way by cookfs paging mechanism. Pages can be compressed, though – in this case `add` and `get` operate on uncompressed data and handle compression internally when reading from disk or writing to

disk. It's worth mentioning that in all cases every page is compressed independently and all get operations are performed on a complete page.

Cookfs stores pages content followed by index and pages metadata:

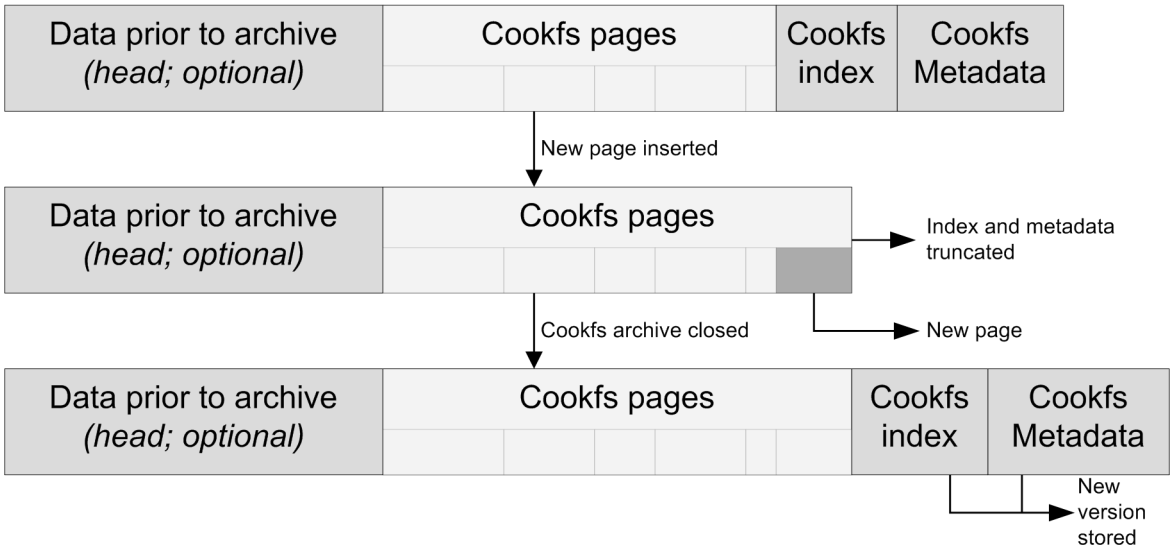


Cookfs pages are one or more pages, stored one after another. Pages do not contain any metadata about themselves and/or their relationship to any files in the VFS.

Pages are followed by metadata, consisting of index and information on sizes and checksums of all pages. Metadata contains enough information to locate all pages, index by just knowing the index of end of cookfs archive.

Cookfs archive can also be preceded by any data – such as platform-specific binary, which is used to create standalone applications. Archives are located at the end of a file or can be found by specifying location of end of archive, in case the archive is followed by additional information, such as digital signature or another type of VFS.

When an update is performed on existing archive, file is truncated to remove metadata and new pages are appended straight after last page. When the cookfs archive is closed, new metadata is appended at the end of the file.



Cookfs caches pages that are read up to specified amount of pages. This is especially useful for small files, in which cases a page can be accessed very often. This allows tuning between smaller memory usage and better performance. Further sections show how increasing cache reduces overall time for reading files.

While concept of pages is generic, cookfs uses them only for storing files. Large files can be stored in multiple pages – for example 10MB file can be stored in 40 chunks of 256kB each. Smaller files might be stored entirely in a page. Files under 32kB are usually grouped and stored in a single page. Information on which pages are used by which files is stored in the index and VFS layer is responsible for splitting files into pages when adding or updating files.

Cookfs is also fully using 64bit offsets, which allows it to store more than 4GB of data. The only limitation is that cookfs can handle up to 2^{31} pages, which allows storing up to 512TB with page size of 256kB.

Cookfs pages mechanism is currently implemented in C, although a limited implementation in Tcl also exists.

Index and directory hierarchy

Cookfs index stores information about directory structure of cookfs archive. It maintains information about archive's directory structure by storing a tree of files and directories in the archive.

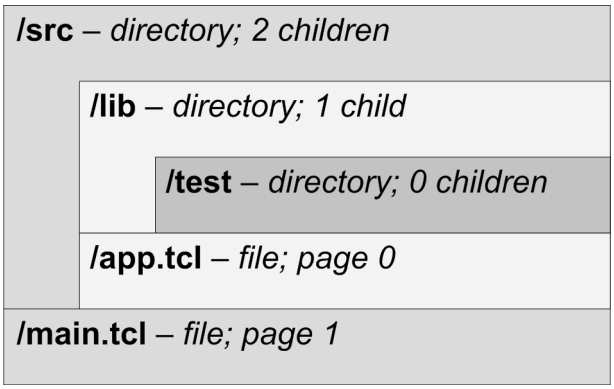
Similar to pages, index part provides set of Tcl methods that can be invoked on indexes. Below is a subset of these:

- `get` *`path`* – gets metadata about specified entry as Tcl list; for files it returns modification time, size and list of integers – their meaning is described in next section; for directories it returns only modification time
- `set` *`path mtime ?filedata?`* – adds file or directory to index; if `filedata` is not specified, entry is created as directory; otherwise item is created as a file `filedata` is a list of integers, same as the one later returned by `get` method
- `unset path` – removes an entry from index; only files or empty directories can be deleted
- `list` *`path`* – lists all items in specified path
- `export` – exports index as binary data
- `delete` – delete an index object

These are essential commands required for cookfs operations. Index can be created as empty one or by providing data that has previously been saved using `export` method. Paths are specified in the form of `dir/subdir/filename` – for example `lib/tcl8.5` specifies `tcl8.5` entry, located as subdirectory of `lib` directory.

Indexing mechanism automatically detects inconsistencies and throws an error in such cases. This makes it impossible to create file `lib/somefile` and later create another file `lib/somefile/invalidname`.

Indexes are internally stored using tree of hash tables, which represent files and directories hierarchy. Indexes are exported as binary data for storing on file system, where each directory stores its children recursively. For example for a simple directory structure the index would be stored as follows:



Information about directory `/src` would also store information on `/src/lib` recursively and `/src/app.tcl`. This allows storing the index efficiently. Storing information on all files in a single chunk also aids compressing index, especially when archive contains large number of files.

Cookfs index stores information about all directories and files in the archive. For each directory in an archive, index contains information about modification time and its child items.

For files, index stores information about file size, modification time and list of all pages the file is stored. Information on pages contains index of the page, starting offset in that page and number of bytes used for storing the file. This allows handling all types of files – ones that use multiple pages for their contents, files that use up exactly one page and files that are grouped in a single page.

Index handling is currently implemented in pure C, although a limited implementation in Tcl also exists.

Cookfs and VFS layer

VFS layer is at the heart of cookfs package. It leverages pages and index mechanisms to create an archive based virtual filesystem for Tcl.

Main features are reading and writing files from and to cookfs archive. Reading files is done using special type of channels and they do not read entire file contents in the beginning. As file is read, required part of the file is retrieved from cookfs pages. Pages are also cached,

which often causes reading multiple files to reuse cached pages and avoid I/O operations on cookfs archive.

As pages store parts of a file independently, seeking operations are very efficient – when 100MB file is to be read from 80th megabyte, previous 80MB do not need to be read. When compressing entire files, seek operation needs to re-read file from the beginning in order to be able to uncompress from 80th megabyte.

This can be useful for things like seeking in audio files or extracting parts of large file.

Writing files is done using memchan[8] package. Whenever a file is written, its contents are stored in memory. When the file is closed whatever was written to it is saved to cookfs archives, creating new pages if needed. This mechanism is used for all operations that can write to a file - writing to a new file and appending to existing files.

Cookfs also allows adding one or more files to archive directly. This way files are read directly from disk and step of reading entirely into memory is avoided. Writing files directly instead of writing to memchan greatly speeds up copying files.

Whenever new contents are to be added to an archive, a check of file size is made. Large files are added to cookfs archive immediately. Small files are not added to archive immediately, but are queued. They are added when the size of all files is reached or archive is unmounted. By default the size limit for small files is 32kB and limit after which they are added to archive is 4MB.

Small files are also grouped by their file names. This allows files such as `pkgIndex.tcl` to be stored next to each other, usually storing them one after another and in the same chunk. This helps compressing the files as files with similar names usually have similar content.

All files are added to archive index after their contents is added to cookfs pages. Index stores each file's modification time and information on pages storing its contents. This is stored as one or more group of 3 integers - page index, offset from page start and number of bytes. For files that use up entire page offset is 0 and size is the size of a page. For small files this will be the page index, offset of the data and file size.

Cookfs VFS layer supports all filesystem related operations that tclvfs package supports. This includes getting and setting file information, listing files and/or directories in specified path and deleting files/directories. These operations simply reuse index functionalities. Creating directories in a virtual filesystem simply adds them to index.

Even though cookfs allows deleting contents of an archive, this simply removes information from the index, without removing any pages that store particular file's content. This is due to cookfs design. Recreating a cookfs archive from scratch is the best way to remove pages no

[8] http://memchan.sourceforge.net/

longer used. This will also copy all files to new archive. In such case small files will be grouped and compressed again, which will also increase effectiveness of compression.

Currently the VFS layer is written in Tcl and leverages tclvfs[9] package. Operations that require reading or modifying information in cookfs index leverage those functionalities using Tcl API that index provides. Writing a file is done as memchan channel that is then added to cookfs after writable channel is closed. Reading files functionality is implemented in Tcl using `chan create`[10] to create script level channels.

Cookit – standalone applications using cookfs

Cookit is a project closely related to cookfs. It is an effort to create Tclkit-like standalone Tcl solution leveraging cookfs as VFS for storing Tcl files and application to run. Currently cookit and cookfs only supports Tcl version 8.6 due to Tcl 8.6 specific features used such as zlib compression.

Cookit binaries contain statically linked Tcl library and have a virtual filesystem appended to them to store additional files – both platform-specific libraries and Tcl scripts. Upon initialization cookit mounts cookfs archive, reads scripts required to initialize Tcl properly. Next if `main.tcl` file is present in the archive, it is loaded and cookit works as a standalone application. If the file is not there, cookit starts interactive shell.

Cookit comes with Tcl compiled statically and only ships libraries that are part of Tcl itself – such as tdbc and TclOO. Microsoft Windows binaries come in two flavors – cookit.exe, a console application (similar to tclsh or tclkitsh) and cookit-ui.exe, version including Tk, similar to wish or tclkit binaries. Windows version also ships dde and registry packages in addition to the default ones.

It is possible to add additional libraries such as Tk or SQLite to a cookit binary by adding them to archive simply by mounting it and copying libraries. Cookfs and cookit plans also include creation and easy adding of ready to use packages – such that SQLite can be easily built and packaged into a cookfs package that can then be used by tools such as TABS as ready to use building blocks.

Cookit binaries do not support reading from or writing to mk4vfs. However, it is possible to simply copy mk4vfs package to cookfs virtual filesystem. As cookfs's main goal is to use cookfs based archives, there are currently no plans to include mk4vfs support by default to cookit binaries.

---

[9] http://sourceforge.net/projects/tclvfs/

[10] http://www.tcl.tk/man/tcl8.5/TclCmd/chan.htm#M22

Building cookit

Cookit build process requires a small set of scripts and source code. Cookit is built automatically by retrieving source code for all required packages and then compiling them – this includes Tcl, cookfs and tclvfs source code. Source code is retrieved from each project's source code repository, such as SourceForge's CVS/Subversion repositories. It currently supports using cvs[11], subversion[12] and GIT[13] source code management solutions. Support for fossil[14] is planned in near future. Paths to each project's source code are embedded in the build scripts.

Afterwards all parts of cookit are configured and compiled – tcl, tk (optionally), tclvfs and cookfs. Finally small piece of code that initializes tclvfs and cookfs is compiled and binary is built.

Next Tcl-only implementation of cookfs is used to build a temporary cookfs binary. It is not compressed in any way and is only used to create final cookfs binary build – which is compressed and created using C-based version of cookfs. This way cookit binaries can be built from any Tcl version – and build system currently supports versions down to 8.3, which can be sometimes found on older versions of operating systems.

From user perspective building cookit is straightforward. Build process below is shown using a Linux machine. Windows machine will perform similar operations. After downloading and unpacking cookit build system from SourceForge downloads page[15] we have the cookit build system ready to go. All commands are done using `build.tcl` or `build.sh` scripts – the latter one is simply a wrapper for `build.tcl` script. Running `build.tcl -help` will print out help on all available build commands and options.

First we need to retrieve source code using `build.tcl retrievesource` command:

```
[ 1/ 3] Retrieve source code for cookfs
    [21:08:37] Retrieving
    [21:08:46] Success

[ 2/ 3] Retrieve source code for tcl
    [21:08:46] Retrieving
    [21:12:32] Success

[ 3/ 3] Retrieve source code for vfs
```

[11] http://www.cvshome.org/

[12] http://subversion.tigris.org/

[13] http://git-scm.com/

[14] http://www.fossil-scm.org/

[15] https://sourceforge.net/projects/cookit/files/cookit/

```
    [21:12:32] Retrieving
    [21:14:17] Success

[21:14:17] Command completed successfully
```

Currently commands to `svn` and `cvs` have to be installed in order to retrieve source code. Future versions of cookit might offer fallback to http based downloads.

All we have to do next is run `build.tcl build-cookit` and a binary will be built. We can also pass `-upx best` option to compress the final cookit binary using UPX[16].

```
$ tclsh build.tcl -upx best build-cookit

[ 1/ 6] Build tcl 8.6b1.2
    [21:22:57] Initializing
    [21:22:57] Configuring
    [21:24:39] Compiling
    [21:28:52] Installing
    [21:29:05] Finalizing

[ 2/ 6] Build cookfs 1.0
    [21:29:05] Initializing
    [21:29:05] Configuring
    [21:29:40] Compiling
    [21:29:43] Installing
    [21:29:43] Finalizing

[ 3/ 6] Build vfs 1.4.1
    [21:29:43] Initializing
    [21:29:43] Configuring
    [21:30:20] Compiling
    [21:30:21] Installing
    [21:30:23] Finalizing

[ 4/ 6] Build cookit 1.0
    [21:30:23] Initializing
    [21:30:23] Configuring
    [21:30:23] Compiling
    [21:30:24] Installing
    [21:30:24] Finalizing

[ 5/ 6] Link binary
    [21:30:24] Initializing
    [21:30:24] Linking
    [21:30:48] Stripping
```

[16] http://upx.sourceforge.net/

```
    [21:30:48] Compressing

[ 6/ 6] Add CookFS layer
    [21:30:57] Initializing
    [21:30:57] Adding temporary VFS
    [21:30:58] Re-compressing VFS
    [21:30:58] Finalizing


[21:30:58] Command completed successfully
```

We can now find our correctly built cookit in directory `_output/<platform>/cookit`. Since our build took place on Linux x86 machine, cookit is located in `_output/linux-x86/` directory.

Cookit build system currently offers stable mechanism to build cookit binaries. Currently only these commands work properly. In the future cookit build system will also offer stable mechanisms to build additional packages – such as tls with embedded openssl, sqlite, tcludp and tclx.

Cookit initialization code

Cookit initialization is similar to other solutions – `cookit.c` file provides `CooKit_AppInit` function, which initializes statically linked packages and passes Tcl code that initializes and mounts cookfs archive.

Tcl code that initializes the archive is very small and reads actual code to initialize required libraries from cookfs archive. The code simply creates cookfs pags object and gets first page, which stores entire code to initialize cookfs archive – such page is called bootstrap in cookit project.

Entire Tcl code embedded in C is as follows:

```
static char preInitCmd[] =
"namespace eval ::cookit {}\n"
"load {} Cookfs\n"
"load {} vfs\n"
"proc getCookfsBootstrap {} {\n"
"set ::cookit::cookitpages [cookfs::pages -readonly [info
nameofexecutable]]\n"
"uplevel #0 [$::cookit::cookitpages get 0]\n"
"}\n"
"getCookfsBootstrap ; rename getCookfsBootstrap {}\n"
"";
```

During cookit build process, page 0 is specially created that it combines multiple libraries – Tcl scripts from tclvfs library, Tcl scripts from cookfs itself, creates and registers cookfs

mount. Initialization code also checks for existence of `main.tcl` and reads it if the file exists. Bootstrap code only includes essential packages – files such as `main.tcl` are stored as regular files in cookfs archive.

Comparison – compressing sample filesystems

Cookfs was designed to create smaller files by performing grouping of files. Some samples below show what are the differences and also demonstrate how zlib based compression differ from bzip2 both in size factors and time taken to compress and decompress contents of the archive.

Packing is taking a directory structure from a real filesystem and putting it into an archive – this includes time to prepare archive, copy actual contents and finalize the archive. Unpacking an archive means mounting already created archive and copying its contents back to real filesystem. Both steps were done independently of each other not to reuse any information across writes and reads.

For the purpose of comparison, the following VFS technologies have been used:

- Mk4 VFS – most commonly used VFS solution
- ZIP VFS – commonly used format for archiving; zipper package used for compressing archives, `vfs::zip` package used for decompressing
- Cookfs with zlib compression – standard cookfs compression, same as in Mk4 and ZIP; archive compression was done using both writing using VFS layer and direct writing
- Cookfs with bzip2 compression – more efficient compression that takes significantly more time to compress and/or decompress; archive compression was done using both writing using VFS layer and direct writing

The following sample directory structures have been used for measuring size and timing differences:

- TclHttpd 3.5.1 – this is a distribution of TclHttpd package[17] version 3.5.1; it is a relatively small package but commonly put in standalone Tcl/Tk applications; uncompressed size: 460kB
- Tcllib 1.11.1 –Tcllib[18] is a library of commonly used functions in Tcl/Tk – a package that is very commonly embedded in Tcl/Tk applications; only Tcl scripts were used for testing, without compiled helper libraries; uncompressed size: 12MB

[17] http://sourceforge.net/projects/tclhttpd/

[18] http://sourceforge.net/projects/tcllib/

- ActiveTcl 8.4 for Linux – typical installation of ActiveTcl 8.4.19.0[19]; uncompressed size: 71MB
- Joomla with LAMP – good example of packaging non-Tcl content into an archive; contains entire BitNami Joomla stack[20] for Linux (along Apache, MySQL and PHP); uncompressed size: 224MB
- Redmine with RubyStack for Linux – largest sample archive; contains entire BitNami Redmine stack[21] for Linux (along with Ruby and all other artifacts) installed on a filesystem; uncompressed size: 535MB

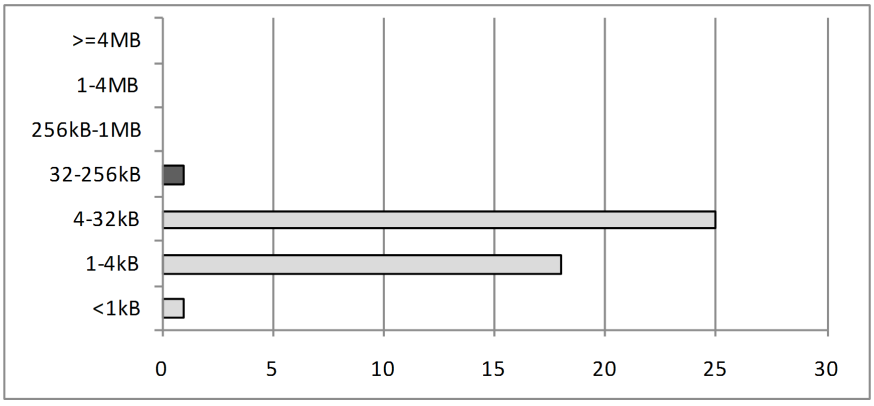
Tests are performed on two platforms – Microsoft Windows and Ubuntu Linux. Both machines have similar configuration – virtual machines using single core, 2GB of memory and 20GB disk drive. Tests were performed when both host and guest operating system were idle. All tests were repeated 5 times and average time was taken as resulting time.

Tclhttpd 3.5.1

First sample is a relatively small package, a minimal subset of Tclhttpd. It shows the difference that cookfs can make even for smaller packages – saving around 20% while still using zlib compression.

Below is analysis of how many files sample file structure included distributed by file sizes. It shows how many files are there in each of size range. Files up to 32kB were treated as small files and are shown in different color on the bar chart.

Size range	Count
>=4MB	0
1-4MB	0
256kB-1MB	0
32-256kB	1
4-32kB	25
1-4kB	18
<1kB	1
Total:	45



Bar chart below shows differences in archive file size for all types of compression:

[19] http://www.activestate.com/activetcl/downloads

[20] http://bitnami.org/stack/joomla

[21] http://bitnami.org/stack/redmine

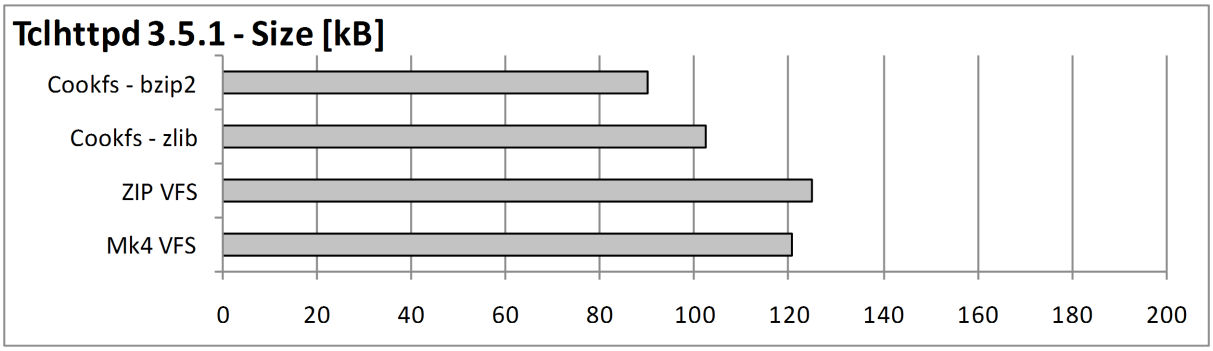


Table below shows sizes and time for each of compression types:

VFS technology	Size [kB]	Linux – Time [s]		MS Windows – Time [s]	
		Pack	Unpack	Pack	Unpack
Mk4 VFS	121	0.2	0.1	0.2	0.1
ZIP VFS	125	0.1	0.1	0.2	0.1
Cookfs – zlib (vfs write)	102	0.1	0.1	0.1	0.1
Cookfs - zlib 16MB (vfs write)		0.1	0.1	0.1	0.1
Cookfs – zlib (direct write)		0.1	0.1	0.1	0.1
Cookfs – zlib 16MB (direct write)		0.1	0.1	0.1	0.1
Cookfs - bzip2 (vfs write)	90	0.2	0.1	0.2	0.1
Cookfs - bzip2 (direct write)		0.1	0.1	0.1	0.1

Time needed to pack and unpack contents of Tclhttpd is small enough that differences are probably due to timing inefficiencies and other factors. However, benefits in size are significant – we can save over 20% of total Tclhttpd size by simply compressing it more efficiently. We can save additional 10% by switching to more optimal compression algorithm.

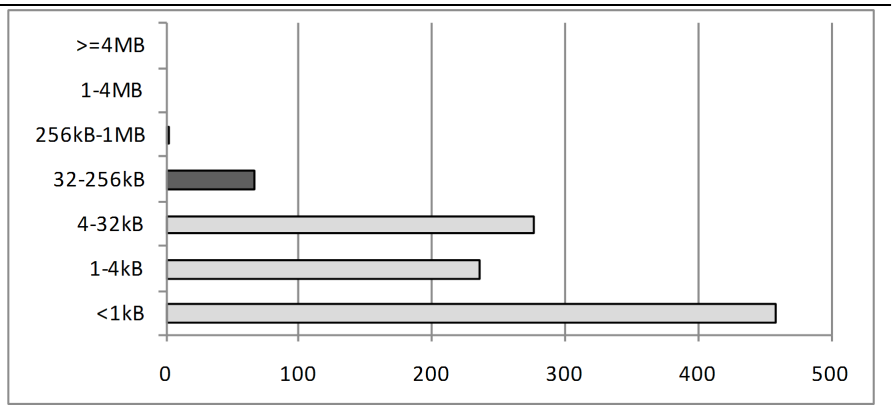
Tcllib 1.11.1

Tcllib is a set of Tcl libraries that is often embedded in standalone Tcl/Tk applications. This test is meant to illustrate how cookfs could compress scripts very similar to typical Tcl/Tk application.

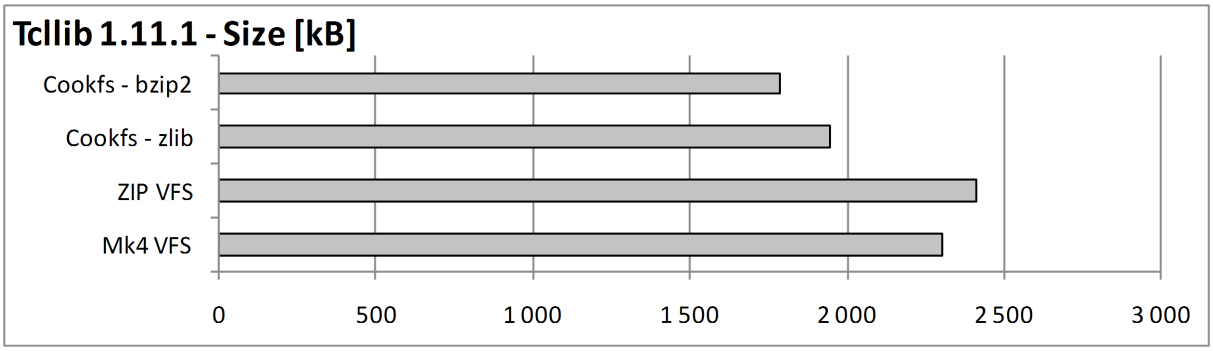
Below is analysis of how many files sample file structure included distributed by file sizes. It shows how many files are there in each of size range. Files up to 32kB were treated as small files and are shown in different color on the bar chart.

Size range	Count
>=4MB	0

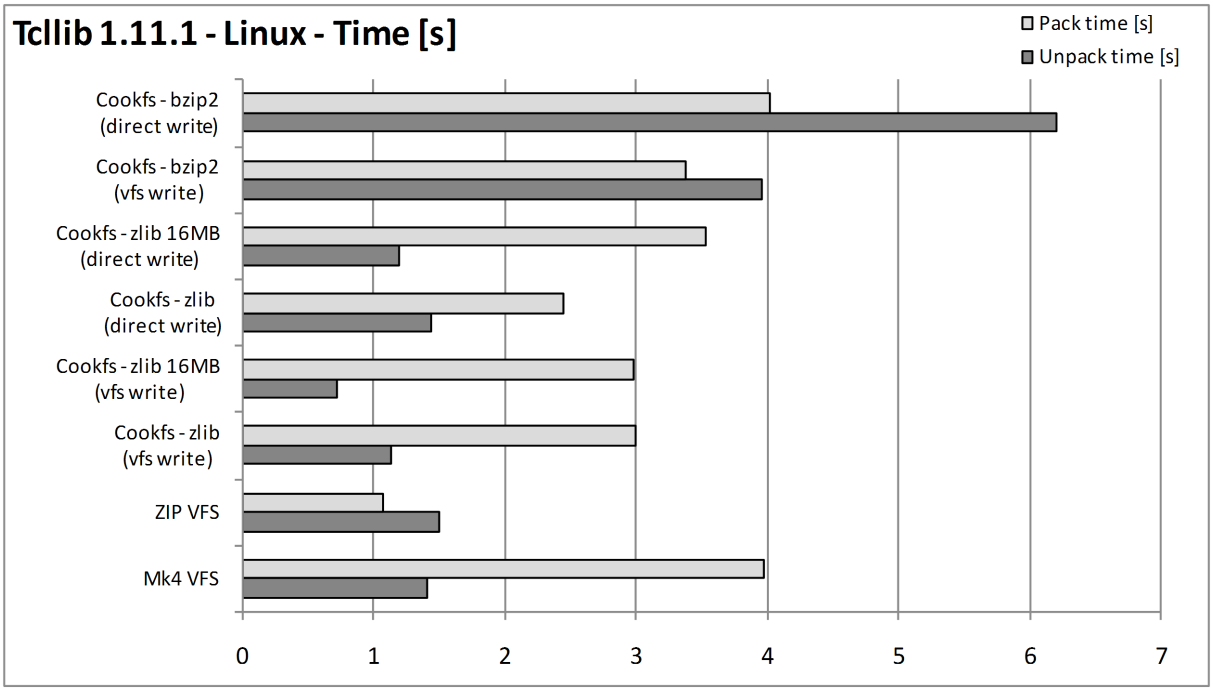
1-4MB	0
256kB-1MB	1
32-256kB	66
4-32kB	277
1-4kB	236
<1kB	458
Total:	1038



Bar chart below shows differences in archive file size for all types of compression:



Bar chart below shows differences in time taken to pack and unpack entire contents of the archive on Linux platform:



Bar chart below shows differences in time taken to pack and unpack entire contents of the archive on Microsoft Windows:

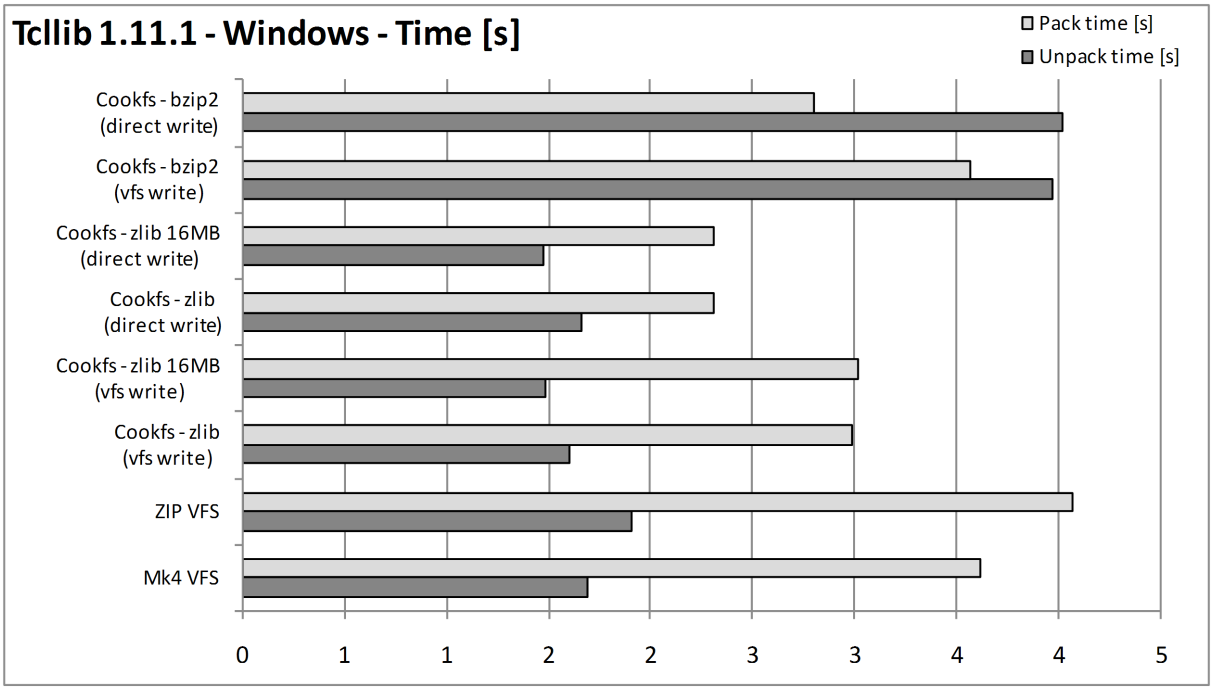


Table below shows sizes and time for each of compression types:

VFS technology	Size [kB]	Linux – Time [s]		MS Windows – Time [s]	
		Pack	Unpack	Pack	Unpack
Mk4 VFS	2 300	4.0	1.4	3.6	1.7
ZIP VFS	2 410	1.1	1.5	4.1	1.9
Cookfs – zlib (vfs write)	1 948	3.0	1.1	3.0	1.6
Cookfs - zlib 16MB (vfs write)		3.0	0.7	3.0	1.5
Cookfs – zlib (direct write)		2.4	1.4	2.3	1.7
Cookfs – zlib 16MB (direct write)		3.5	1.2	2.3	1.5
Cookfs - bzip2 (vfs write)	1 787	3.4	4.0	3.6	4.0
Cookfs - bzip2 (direct write)		4.0	6.2	2.8	4.0

This example shows that cookfs with zlib compression provides faster results in both compression and uncompression than Mk4 VFS. Using direct writes instead of using VFS layer provides additional benefit for packing and using larger cache provides in faster unpacking of the archive. Similar to Tclhttpd, simply using cookfs with same compression provides 15% gain in archive size. Using bzip2 compression provides additional 10% gain at the cost of higher compression and uncompression times.

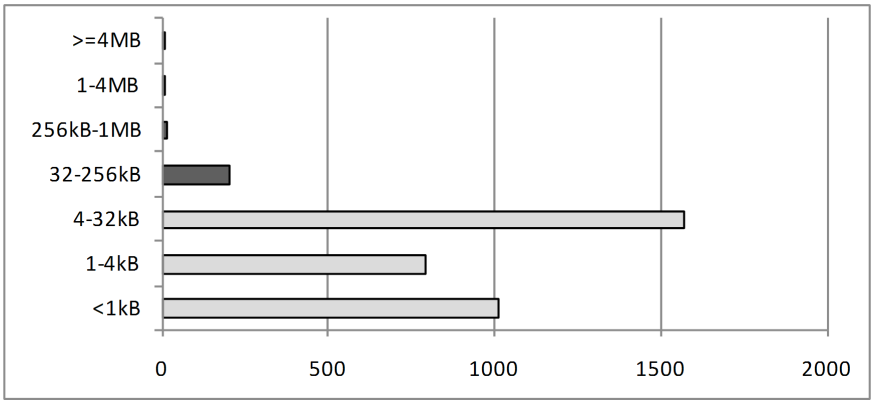
Difference in time taken to compress and uncompress all of zlib based solutions is similar, although cookfs offers a minor performance increase over Mk4 VFS and ZIP VFS. Using bzip2 clearly takes much longer to complete.

ActiveTcl 8.4 for Linux

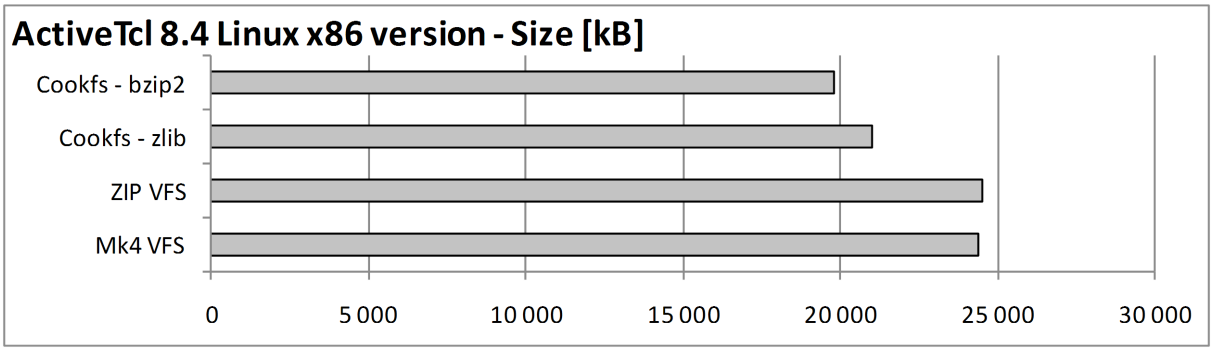
This part of comparison is meant to show an archive which contents is mixed – it consists of both large number of Tcl scripts and binary files – multiple binary files as well as all libraries that ActiveTcl provides. The reason version 8.4 was chosen is that it is the last version of ActiveTcl that ships all commonly used libraries by default.

Below is analysis of how many files sample file structure included distributed by file sizes. It shows how many files are there in each of size range. Files up to 32kB were treated as small files and are shown in different color on the bar chart.

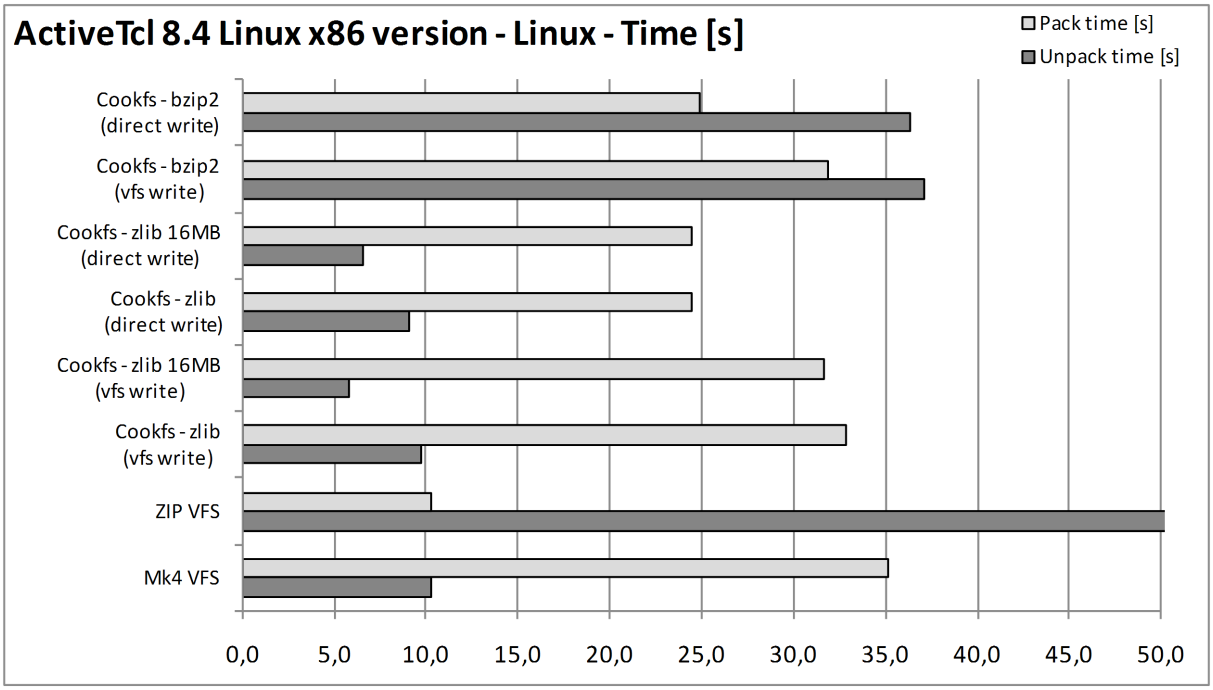
Size range	Count
>=4MB	1
1-4MB	10
256kB-1MB	18
32-256kB	205
4-32kB	1573
1-4kB	791
<1kB	1013
Total:	3611



Bar chart below shows differences in archive file size for all types of compression:



Bar chart below shows differences in time taken to pack and unpack entire contents of the archive on Linux platform:



Bar chart below shows differences in time taken to pack and unpack entire contents of the archive on Microsoft Windows:

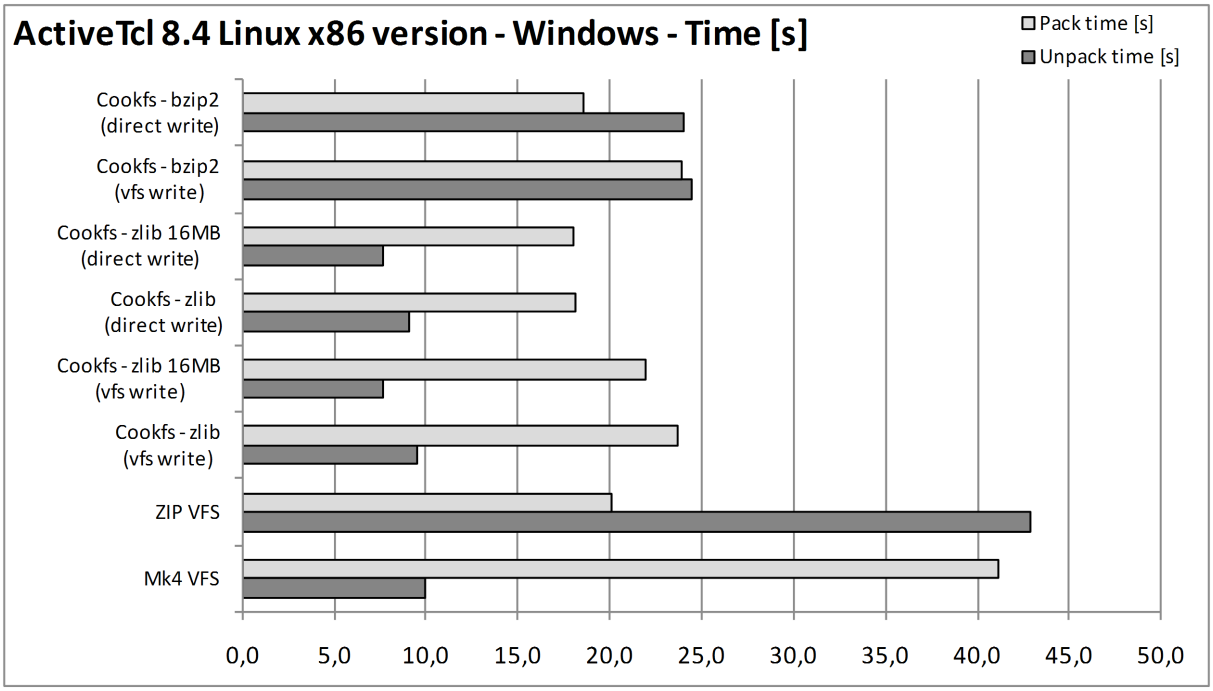


Table below shows sizes and time for each of compression types:

VFS technology	Size [kB]	Linux – Time [s]		MS Windows – Time [s]	
		Pack	Unpack	Pack	Unpack
Mk4 VFS	24 408	35.2	10.3	41.2	10.0
ZIP VFS	24 535	10.3	72.1	20.1	42.9

Cookfs – zlib (vfs write)	21 010	32.9	9.7	23.7	9.6
Cookfs - zlib 16MB (vfs write)		31.6	5.8	22.0	7.7
Cookfs – zlib (direct write)		24.5	9.1	18.1	9.1
Cookfs – zlib 16MB (direct write)		24.4	6.5	18.1	7.7
Cookfs - bzip2 (vfs write)	19 815	31.8	37.1	23.9	24.5
Cookfs - bzip2 (direct write)		24.9	36.3	18.6	24.0

Similar to previous examples cookfs clearly offers better compression ratio with zlib compression – around 15% smaller archive was created. Using bzip2 compression offers additional gain, but only over 5% at a cost of much higher compression and decompression times.

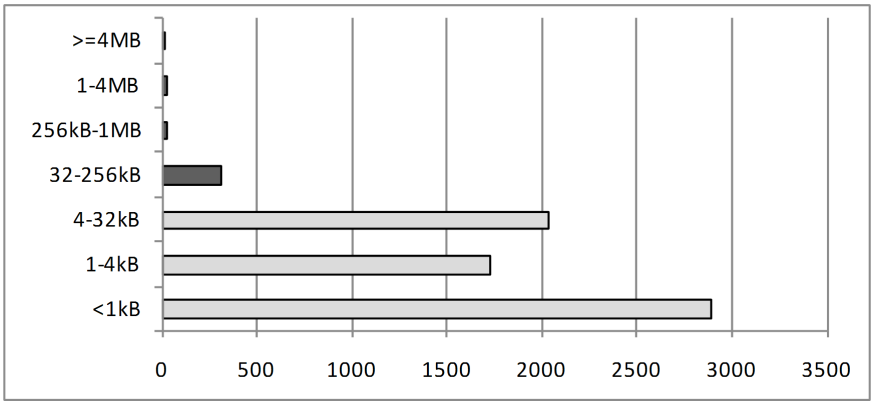
Times taken to compress the archives are similar for Mk4 VFS, ZIP VFS and cookit. The only difference is if cookit is used with direct writes. Unpacking from an archive works much faster with cookit, especially if using 16MB of memory for cookfs cache.

Joomla with LAMP

Next sample contens is a Joomla application along with all Linux x86 binaries needed in order to get it up and running – Apache, MySQL and PHP. The content was extracted from a fresh installation of Joomla using BitNami Joomla Stack[22]. This is first example that does not involve packaging files typical to Tcl.

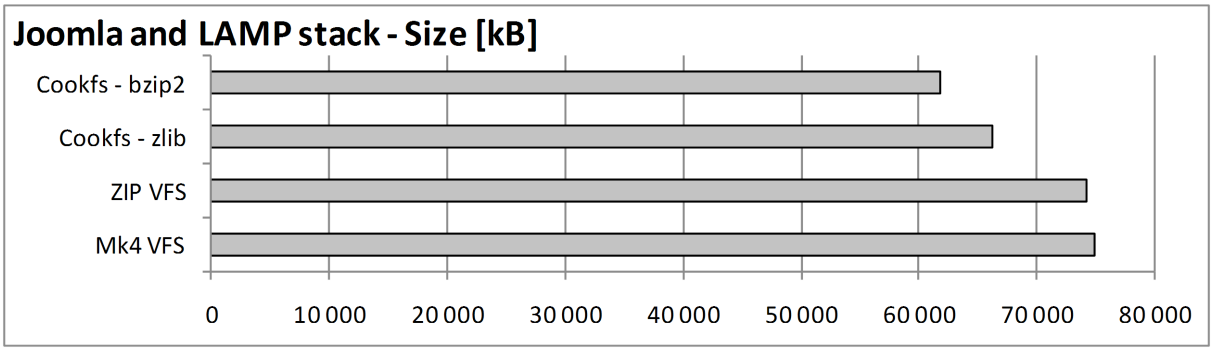
Below is analysis of how many files sample file structure included distributed by file sizes. It shows how many files are there in each of size range. Files up to 32kB were treated as small files and are shown in different color on the bar chart.

Size range	Count
>=4MB	14
1-4MB	25
256kB-1MB	23
32-256kB	318
4-32kB	2038
1-4kB	1729
<1kB	2892
Total:	7039

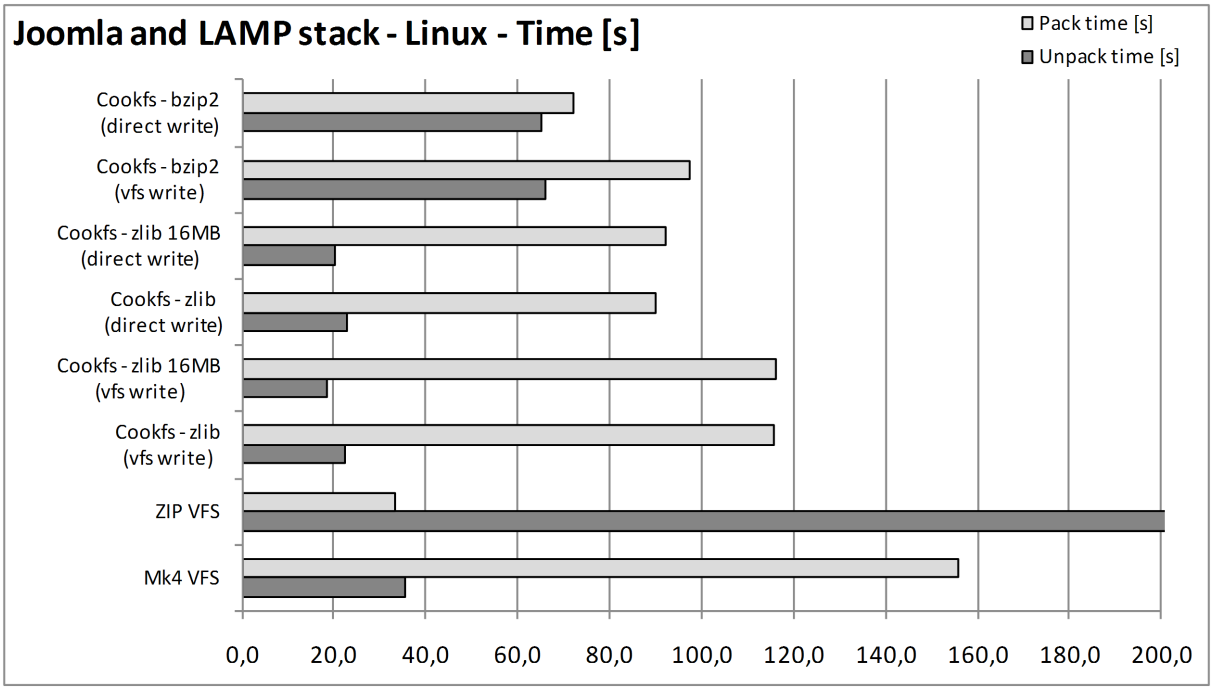


[22] http://bitnami.org/stack/joomla

Bar chart below shows differences in archive file size for all types of compression:



Bar chart below shows differences in time taken to pack and unpack entire contents of the archive on Linux platform:



Bar chart below shows differences in time taken to pack and unpack entire contents of the archive on Microsoft Windows:

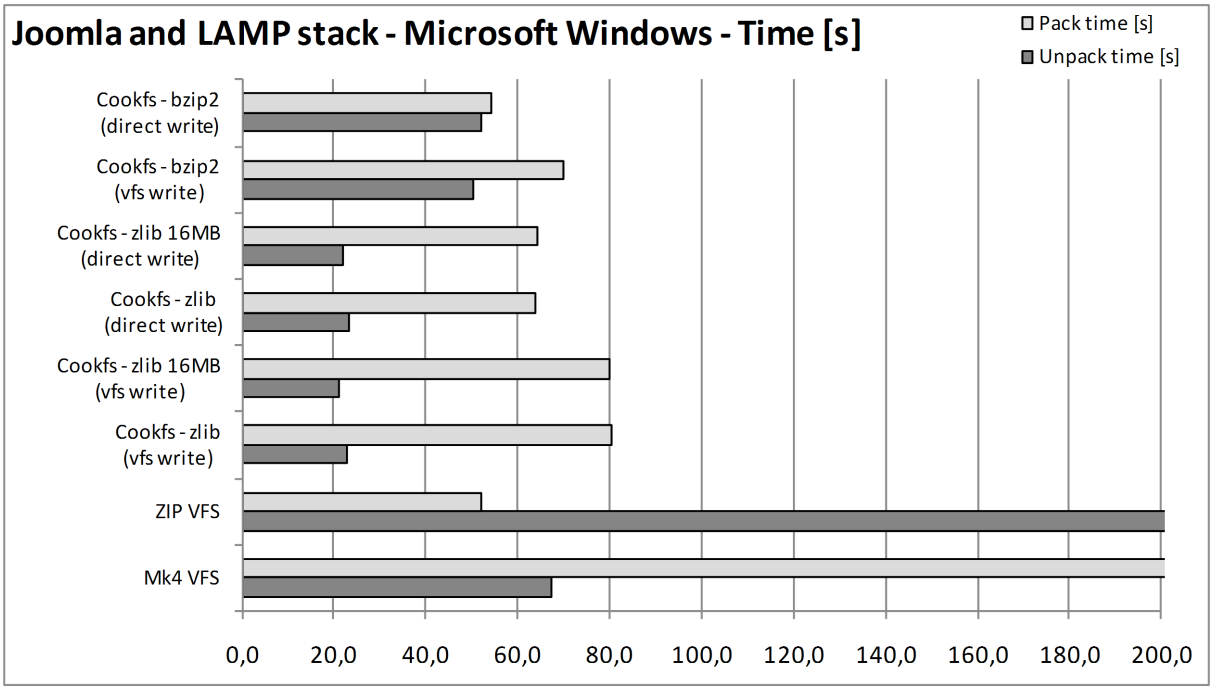


Table below shows sizes and time for each of compression types:

VFS technology	Size [kB]	Linux – Time [s]		MS Windows – Time [s]	
		Pack	Unpack	Pack	Unpack
Mk4 VFS	74 951	155.9	35.4	223.9	67.1
ZIP VFS	74 266	33.5	1263.3	52.1	713.4
Cookfs – zlib (vfs write)	66 260	115.9	22.4	80.4	22.9
Cookfs - zlib 16MB (vfs write)		115.9	18.4	79.9	21.0
Cookfs – zlib (direct write)		90.0	23.0	64.0	23.1
Cookfs – zlib 16MB (direct write)		92.0	20.2	64.4	21.9
Cookfs - bzip2 (vfs write)	61 800	97.6	65.9	69.8	50.4
Cookfs - bzip2 (direct write)		72.2	65.1	54.2	52.1

This example is interesting as it is not strictly related to Tcl – application itself is PHP and majority of size is taken by binaries and libraries needed for it to run.

Cookfs managed to get much better compression ratio – 10% smaller archive than Mk4 and ZIP VFS. Bzip gained another 7% increase, but at a very high cost, especially when decompressing archive's contents.

Mk4 VFS does take a lot of time to compress all the files – it probably is related to the fact that it creates each file individually and stores its index information individually while cookfs and ZIP VFS store this when finalizing archive.

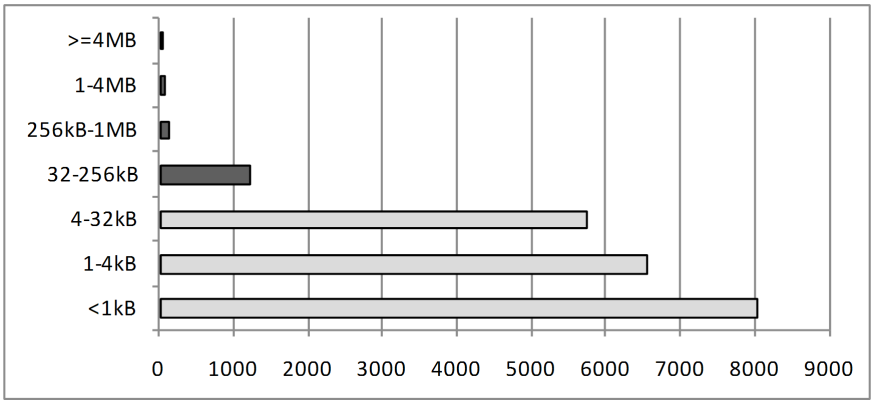
The fact that ZIP VFS is created much faster than any other technologies is intriguing – it seems to outrun each other technology, which might have something to do with how cookfs and Mk4 VFS handle larger number of files. Unpack time for ZIP VFS is also a bit out of order as it is around 50-100 times worse than other technologies, which is probably related to how `vfs::zip` package is written and that it might not be handling very large number of files efficiently as well. However, multiple runs of the test show conducted at different times showed the proportions to other technologies do not change.

Redmine and RubyStack for Linux

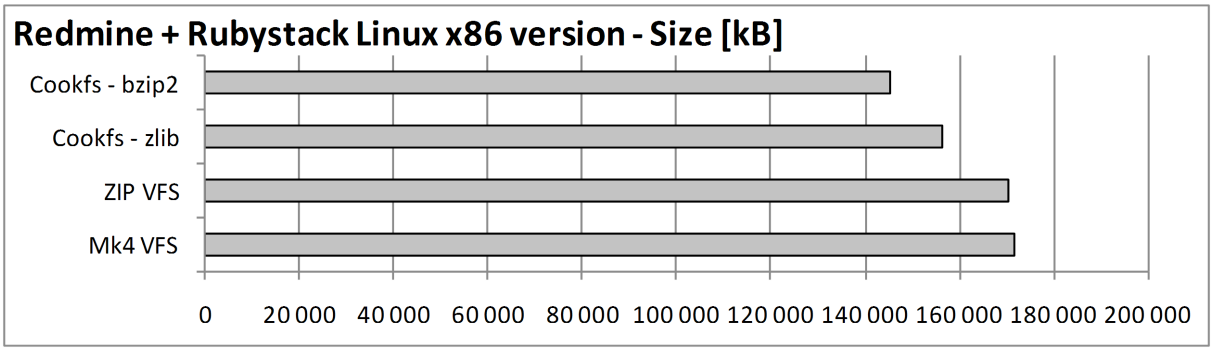
This example consists of all applications that are needed to run Redmine application – Apache, MySQL, Perl, Ruby and additional helper applications. Uncompressed size is 535MB, which is a great example for compressing content for installer applications.

Below is analysis of how many files sample file structure included distributed by file sizes. It shows how many files are there in each of size range. Files up to 32kB were treated as small files and are shown in different color on the bar chart.

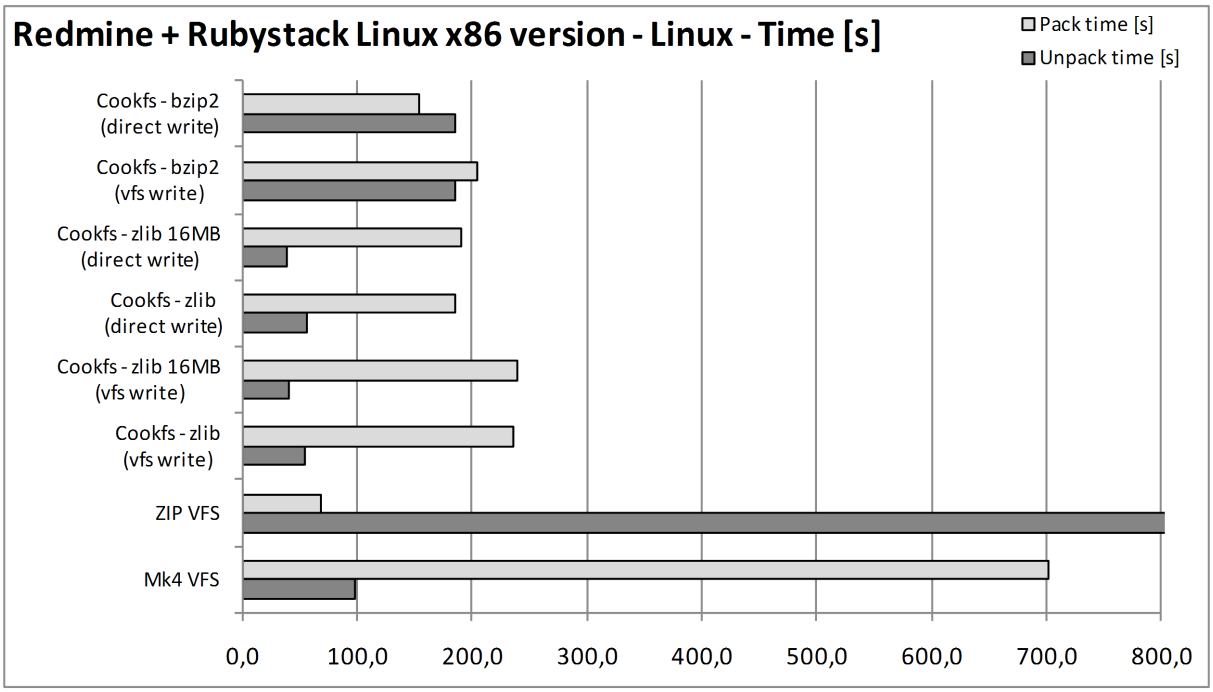
Size range	Count
>=4MB	17
1-4MB	59
256kB-1MB	138
32-256kB	1229
4-32kB	5753
1-4kB	6563
<1kB	8053
Total:	21812



Bar chart below shows differences in archive file size for all types of compression:



Bar chart below shows differences in time taken to pack and unpack entire contents of the archive on Linux platform:



Bar chart below shows differences in time taken to pack and unpack entire contents of the archive on Microsoft Windows:

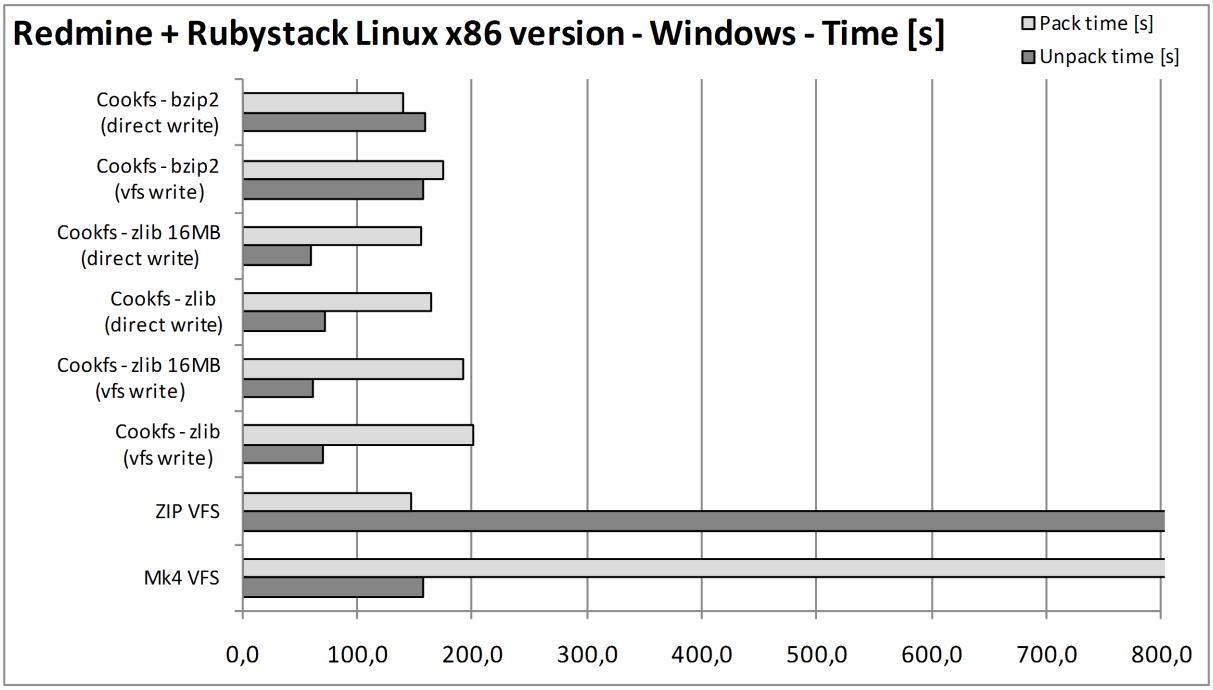


Table below shows sizes and time for each of compression types:

VFS technology	Size [kB]	Linux – Time [s]		MS Windows – Time [s]	
		Pack	Unpack	Pack	Unpack
Mk4 VFS	171 503	701.8	98.4	3650.4	158.3
ZIP VFS	170 306	69.5	1883.9	147.4	1215.6
Cookfs – zlib (vfs write)	156 134	236.4	55.3	201.3	70.5
Cookfs - zlib 16MB (vfs write)		239.1	39.9	192.6	61.1
Cookfs – zlib (direct write)		186.2	57.2	163.7	71.4
Cookfs – zlib 16MB (direct write)		189.9	39.7	155.0	60.7
Cookfs - bzip2 (vfs write)	145 368	203.9	185.3	175.8	158.3
Cookfs - bzip2 (direct write)		153.5	184.9	139.6	159.6

Similar to smaller Joomla example, cookfs managed to compress entire contents most efficiently – although in this case it was only 10% less than Mk4 VFS or ZIP VFS. Using bzip2 compression introduced only a small improvement of 7%. However, cookfs' compression and decompression times for zlib were much better than other ones shown in this example.

What is interesting is that cookfs is able to uncompress the data around 30% faster by using 16MB instead of 2MB for cache. This is especially interesting that archive is over 150MB, so only 10% of its entire contents is cached.

Same as in previous example, Mk4 VFS takes a lot of time to compress archive contents – even an hour on Microsoft Windows. Its unpack time is only a bit bigger than cookfs, unless cookfs.

Similar to Joomla's example ZIP VFS takes much smaller amount of time to compress the data. However its unpack time is also much lower – around 10-30 times slower than competitive solutions in this example.

Current status and future plans

Cookfs project is a reasonably stable and mature solution, therefore it is versioned as 1.0. While it does not currently offer all features that are planned, it can safely be used as VFS for standalone application as cookit project shows.

One of limitations is that cookfs and cookit only support Tcl 8.6 for now. Plan is to allow integration with older Tcl versions, starting with 8.5 and perhaps even supporting 8.4 versions.

Since pages and index mechanisms are complete and written in C, most changes will happen within VFS layer. Those changes will also likely not cause backward or forward incompatibilities. Newer cookfs versions will be able to read archives created using previous versions as well as older cookfs releases will be able to read archives created by more recent versions.

Cookfs currently still has some drawbacks or limitations – one of its bigger flaws is having code partially in Tcl and C, which then causes it to require tclvfs - both C level and Tcl level elements of tclvfs. Currently Tcl scripts handle reading of archive contents. This code will be replaced by creating a channel driver in C. This will speed up reading operations.

Another issue is that for writable channels, they can be implemented in C and without the need to fully buffer file contents before storing it. Files can be written to memory until enough content is written to this channel to create a new cookfs page. This will reduce time and memory usage when writing to VFS using Tcl I/O. It will reduce the difference between writing to VFS and directly using cookfs command.

Using tclvfs for providing VFS layer for Tcl can be replaced by fully implementing VFS at C level, similar to trofs[23] project. Cookfs would then offer a much better alternative to other VFS solutions and its performance would also increase. However, due to large effort to implement it, this is currently a low priority feature.

---

[23] http://math.nist.gov/~DPorter/tcltk/trofs/

Current VFS layer implementation also does not work properly with cookit and multi-threaded applications. Often deadlocks occur when creating a child thread that needs to access VFS during its initialization. This is related to using tclvfs and will no longer be an issue when cookfs provides its own C level VFS driver.

A temporary limitation of cookfs is that if an archive with any pending changes is not unmounted before process exits, data is lost and archive will be unreadable. This can be worked around by adding Tcl exit and/or thread exit handlers to properly unmount archives.

Cookfs will also include additional tools and commands for building applications and ease of archive management. A very useful command would be one that can compact an archive – recreating an archive to prevent wasting space in an archive that had multiple changes, especially file deletions.

In addition to this, cookfs will provide commands for merging one or more cookfs archives without recompressing their contents – by simply adding pages and merging indexes.

Additional feature worth considering as additional tool is being able to unpack large amounts of files more efficiently. Currently files are unpacked in order they are found. A smarter mechanism can order the reads by page indexes to speed it up even more.

Cookit project is currently stable and mature – binaries are available for most common platforms – Microsoft Windows 32bit, Linux 32bit, Mac OS X 32 bit, Solaris x86 32 bit. Cookit build process allows building on most platforms that are supported by configure scripts for each of the components so extending set of platforms should be at most adding platform-specific changes or hacks to allow these platforms to be built.