CacheFS – *A hybrid RAM, SSD and HDD file system*

Ioan Ovidiu Hupca, Ionuţ Bogdan Petre, Alexandru Cristian Stan

Abstract

The focus of this project is a hybrid file system, with the declared purpose of accelerating file access time and throughput as well as a possible improvement of power consumption through the combined usage of random-access memory, solid-state drives and hard disk drives. The idea is simple: use the fast memory (RAM and SSD) as a two-level cache. The most accessed files will be stored into RAM, while the less intense accessed files will reside in the SSD. When a file is not located into the cache memory, a cache miss event takes place and data is read from the hard drive.

Keywords: filesystem, hybrid, cache, solid-state drive

I. Introduction

The focus of this project is a hybrid file system, with the declared purpose of accelerating file access time and throughput as well as a possible improvement of power consumption through the combined usage of random-access memory, solid-state drives and hard disk drives. The idea is simple: use the fast memory (RAM and SSD) as a two-level cache. The most accessed files will be stored into RAM, while the less intense accessed files will reside in the SSD. When a file is not located into the cache memory, a cache miss event takes place and data is read from the hard drive.

By using the SSD as a level 2 cache, large files can be accessed very fast (as compared to the HDD). Large files cannot typically be held into a RAM disk, as its capacity can be easily filled. Current SSD models have capacities of 120 GB and reading speeds of up to 250 MB/s. This is well beyond what a high speed hard drive can offer (Western Digital Velociraptor topping at ~100 MB/s).

Performance tuning can be done for specific applications by creating file access profiles and then storing that data into the cache levels. Also, a library for allowing cache-aware applications is implemented.

Power usage improvement is achieved by obtaining a "stable file access profile": all the needed files are located in the RAM and SSD, therefore making it possible to shut down the hard drive in order to save power.

The inspiration for this project was the Conquest-2 project at UCLA which used only RAM and HDD, but employed a more sophisticated file access pattern profiling that what is planned.

II. Technology

A. Random-access Memory

Random-access memory (RAM) is a data storage which uses integrated circuits to store data that can be accessed in any order. Any piece of data in the memory can be accessed in constant time, regardless of its physical location and whether or not it is related to the previous piece of data.

Random-access memory is a volatile memory, where the information is lost after the power is switched off. Random-access memory has a much better access time and a much higher read rate compared to solid-state drives and hard disk drives. On the other hand, RAM is much more expensive than SSD and HDD, so it has a smaller capacity. At the date of writing this paper, the average price per gigabyte was 25$, for a DDR3 module with a peak transfer rate of 12 GB/s, higher than any other of the consumer available storage technologies.

B. Solid State Drives

A solid-state drive (SSD) is a data storage device that uses solid-state memory to store persistent data. An SSD emulates a hard disk drive interface, thus easily replacing it in most applications. With no moving parts, solid-state drives are less fragile than hard disks and are also silent, as there are no mechanical delays, they benefit from low access time and latency.

The current generation of SSDs uses non-volatile flash memory. Whilst they are much more expensive than a hard-disk at the same capacity, the solid-state drives win in speed. Most SSDs now have a read speed in excess of 200 MB/s, for an average price per gigabyte of 3.5 $. An SSD now offers a very good compromise between price and capacity, making it ideal as a secondary cache level.

C. Hard Disk Drives

A hard disk drive (HDD) is a non-volatile storage device that stores digitally encoded data on rapidly rotating platters with magnetic surfaces. Hard disk drives store information permanently, unlike RAM and like SSD.

Hard disk drives have capacities up to 2 terabytes, much larger than the maximum capacity of random-access memories and solid-state drives. On the other hand HDDs have a much higher access time and a data read rate much lower than either of the two. This is because RAMs and SSDs do not have a seek time. Also HDDs have higher power consumption. However, a classical hard disk drive has the best price per gigabyte of all three storage technologies: 0.18$.

III. CacheFS

A. CacheFS Architecture

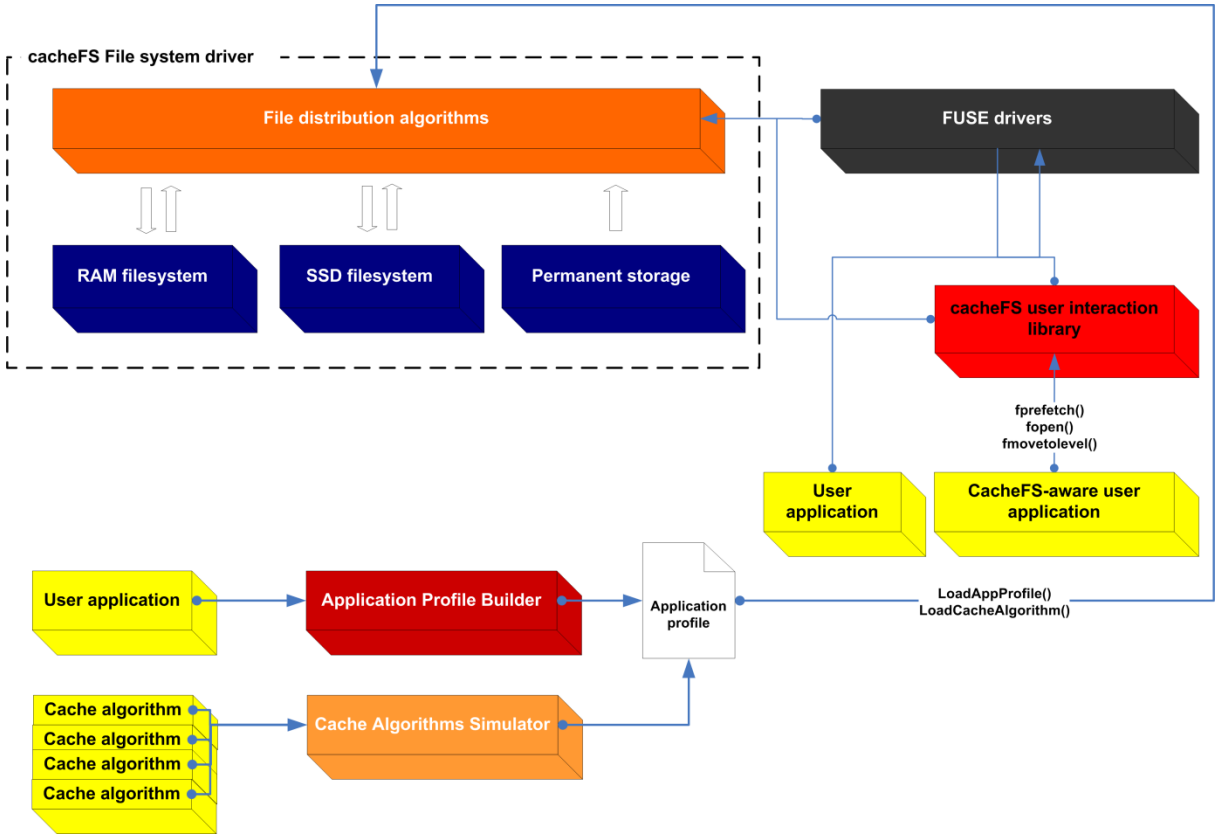

Figure 1 - CacheFS Architecture

The File System Driver is the core module of the project. Its task is to supply files from the three available data stores (by speed: RAM, SSD and HDD) and apply different caching and prefetching algorithms for moving data in-between the data stores. Currently, FUSE is used for implementing the driver, with plans to move to kernel-mode.

There are several caching algorithms implemented. Their performance is tested in a practical versus theoretical results comparison via the Cache Algorithms Simulator. The simulator runs the algorithms and produces multiple metrics. Those with the best results are implemented into the driver and their performance is compared.

Some caching algorithms can load an application profile in order to fine-tune themselves for that application. For example, by using such a profile, the most accessed files can be loaded directly to RAM whilst others can be copied to SSD or simply left on the hard disk. It is done without using the existing metadata and access counters (therefore less overhead). Also, the files are in the best location at the best possible time.

The profiles are built with the Application Profile Builder. By using function interception and library preloading, all the file access routines are hijacked by the application and used for creating a description of the file interaction patterns.

B. File System Driver

The File System Driver supplies files from the three available data stores (RAM, SSD and HDD) and applies different caching and prefetching algorithms for moving data in-between the data stores.

The current implementation, allowing for easier performance testing and debugging, uses FUSE and runs the file system routines in user space. Once its feasibility is proven, the routines can be ported to kernel space.

Several caching algorithms are implemented, permitting multiple levels of operation optimization.

The FUSE file system, when mounted, provides a virtual view of the cached files. Therefore, an application can benefit from the features of cacheFS without any knowledge of it. No modification or recompilation is required.

B1. Driver Architecture

The internal structures maintain details regarding the cache level (either SSD or RAM) as well as metadata that allows for further optimization.

The architecture is modular, allowing for plug and play modules and algorithms: there is an interface describing the caching operations and one for the caching algorithms. This allows for multiple, easily switchable, implementations of both.

Cache control interface

This interface is implemented by each cache implementation module**.**

- `initCache` - initializes the file system, allocated RAM and SSD space and internal structures
- `releaseCache` - releases the allocated space and cleanups the internal structures
- `cacheFile` - caches a file on the HDD. Can force a target level (RAM or SSD) or allow for it to be automatically selected by the current algorithm
- `getCacheLevel` - queries the cache level of a file
- `getFreeSpace` - queries for cache file system free space
- `cleanup` - forces a call to the cache garbage collector
- `getCacheState` - queries current cache file system state

Caching algorithms interface

This interface is implemented by each caching algorithm**.**

- `initCacheStructs` - initializes caching algorithm internal structures
- `releaseCacheStruct` - cleanups caching algorithm internal structures
- `cacheFile` - caches a file on the HDD. Can force a target level (RAM or SSD) or allow for it to be automatically selected
- `uncacheFile` - removes a file from the cache
- `moveFile` - moves a file from one cache level to the other
- `garbageCollect` - calls garbage collector (try to free some space on the file system by applying different rules)
- `getCacheLevel` - queries the cache level of a file
- `getCacheStatus` - queries current cache file system state

Cache implementation modules

These modules represent the implementation of the file system operations. They also contain wrappers for the routines of each algorithm.

Simple Cache

It represents the simplest possible implementation. It uses `/dev/shm` for RAM storage and another mounted partition as the SSD (ex: `/mnt/ssd`). It is mostly a wrapper around existing file system calls. Basically, when files a copied to the cache, they are copied to one of the two mount points. Different file systems can be used for the SSD partition, allowing a better fine-tuning of the file operations.

Raw Cache

Like Simple Cache, it uses `/dev/shm` and a mounted partition. But it allocates a large single file on each of them and manages the structures internally. It is useful for large files.

Caching algorithms

The caching algorithms control the way the RAM and SSD cache is filled, how a cache level is selected for a file and how the garbage collector works.

Simple Algorithm

As the name says, it is a simple algorithm. It just moves files to the level specified by a rule (file size for example). It does not make any optimizations during operation.

C. Application profile builder

In order for the caching and prefetching algoritms to work optimally for a given application, an application profile can be used.

The application profile will indicate what files are accessed, when they are accessed and in what sequence the files are usually loaded. By marking sequence start points, the driver can preload the files that are next in the sequence before the request for them actually arrives.

The application profile builder will be tasked with creating and analyzing the file accesses of a given application and create a task list for the driver, in the form of an application profile. It will contain lists of files to preload and the triggers for the preload. The caching algorithms usable can be specified.

This profile will be followed by the driver like a map. No analysis will be done.

D. Cache algorithms simulator

The algorithms used for moving the data files around the data stores are paramount to providing the speed improvement desired. Therefore, a variety of algorithms are implemented and they are tested on a variety of test cases.

We use a two-level cache (RAM and SSD) for improving access time to read-only files (the files are only accessed to be read). A possible use for this could be a web server.

There are more algorithms which are implemented and compared.

One of the proposed algorithms tries to copy an accessed file to RAM first (if it is enough space), than to SSD (also it is enough space). If neither RAM nor SSD has enough free space, the algorithm tries to free space on RAM or SSD. If the file which is accessed can fit into RAM, files are removed from RAM using FIFO algorithm until there is enough space for the new file. Otherwise, if the file which is accessed can fit into SSD, files are removed from SSD using FIFO algorithm until there is enough space for the new file. If the file is very large and cannot be copied to RAM or SSD, it is accessed from the hard disk.

Another algorithm splits the files according to their size: the files smaller than a certain size are copied only to RAM and the rest of files are copied only to SSD. Therefore, the performance will be better because small and large the files are not longer mixed together.

The algorithm used for removing files from RAM or SSD is FIFO (First-in, first-out). This algorithm keeps a queue of accesses to the files. When a file must be removed, the first file in the queue (which was introduced least recently) is chosen. This algorithm does not perform very well in practice, so the Second-chance algorithm, a modified form of FIFO, is implemented. This algorithm gives the files a second chance: a bit of access is used and if a file is first in the queue, but was used (has the access bit set) it is not removed, but its bit is cleared and the page is moved to the back of the queue and the new head of the queue is analyzed. Otherwise, the page is removed from the cache.

Because the file system implements file-level caching, sometimes is it necessary to remove more than one file from the cache in order to make space from a new file. Therefore the algorithms are adapted in order to do that: if a large file must be introduced in cache, there must be calculated how many files are to be removed.

The two-level cache is emulated in order to test the theoretical performance of the system, which will be compared to the practical performance.

These algorithms are implemented in Java, using Java Development Kit version 1.6.

IV. Results

A. File system driver and simulator

The benchmark for the file system driver has been done in a VMWare virtual machine, running Fedora 10 x86 with 1 GB allocated RAM. The host machine is an Intel Core 2 Duo, T7500 running at 2.2 GHz with 4 GB of RAM. Unfortunately, no SSD was available for testing so real speed tests were unable to be conducted. A hard-disk mount point was used instead.

Linux (like all modern operating systems) has a caching mechanism for disk read and writes. In order to obtain real speed values, this caching mechanism must be turned off. This was possible for the “control-test”, reading data from the hard-disk directly, by using the O_DIRECT flag for the open() function. However, FUSE does not support this flag and, therefore, direct, uncached I/O.

Timing an operation is difficult in a multitasking environment, as the kernel can stop a running process and run another one, without warning. Since there is no way of locking a portion of code or a process from being preempted, timing results are difficult to obtain accurately. It has been implemented by calling the *gettimeofday()* library function and subtracting the two values. However, preempting problems aside, there is also the problem of resolution: modern processors can be supply a 1 microsecond resolution for this timer, however the system call needed for getting the actual value implies 5-8 microseconds. Take into account the fact that a virtual machine was used, resulting into more overhead. It is then almost impossible to get accurate results when it comes to small individual files.

The metrics used for measuring the results were *access time* and *total read time*. Specifically, access time is the time needed for the open call to complete and the total read time is the time needed for reading the entire file.

HDD read	Access time	Read time
Small file (<100 KB)	5-20 us	40 us – 40 ms
Large file (~ 8 MB)	5-20 us	200 ms

Table 1 - HDD read test

With disk caching turned off, the HDD read test provided the results presented in Table 1. As expected, the time needed for reading the large file was higher than the one needed for a small file. However, the results for large identical files were very oscillating. VMWare involvement is suspected.

Simple Algorithm (cache to RAM)	Access time	Read time
Small file (<100 KB)	50-80 us	40 us-700 us
Large file (~ 8 MB)	200-300 us	100 ms

Table 2 – Simple Algorithm test results

Simple Split Algorithm	Access time	Read time
Small file (<100KB)	50-80 us	40 us-700 us
Large file (~ 8 MB)	350-400 us	100 ms

Table 3 – Simple Split algorithm test results

When running the tests on the cacheFS mounted files with the Simple Algorithm (with caching done just in the RAM, as there was sufficient space), a climb is seen for the Access time. This is because when a file access is detected (i.e. open()), cacheFS calls the algorithm and executes the copying, if needed. Therefore, access time when using cacheFS involves waiting for the file to be copied to the right location at first use. Subsequent uses will access the file directly.

When using the Simple Split Algorithm on a large file, it is copied to the SSD partition. Unfortunately, because there was no SSD available for testing, this actually involves reading the data from HDD and writing it to the same HDD. Obviously, this makes the speed data useless.

In order to picture what the speed increase can be, the simulator is used. By using standard read rates for SSD, RAM and HDD, it can give an educated guess on what the performance can be.

Simple Algorithm (cache to RAM)	Access time	Read time
Small file (<100 KB)	30-40 us	30 us-500 us
Large file (~ 8 MB)	150-200 us	50 ms

Table 4 – Simple Algorithm simulation results

Simple Split Algorithm	Access time	Read time
Small file (<100 KB)	30-40 us	30 us-500 us
Large file (~ 8 MB)	250-300 us	80 ms

Table 5 - Simple Split Algorithm simulation results

Overall, the times obtained with the simulator are better than the times obtained in real testing. This is because the simulator tests the cache in ideal conditions. The read time for large files using Simple Split Algorithm is better than the same time needed to read files from the hard disk because the files are read only once from HDD and then are accessed more quickly from the SSD or RAM.

B. Application profile builder

Application Profile module provides files access monitoring. This module was build to keep the list of all accessed files and the time when they were accessed. It also does analysis of the file accesses and some statistics like providing the number of accesses of each file.

The list with the sequence of files is used by the driver which reads the files list. Driver should preload the files provided by Application Profile Builder and this will decrease the access time as the files will be in cache memory before they are actually requested.

The list of the accessed file was built intercepting system calls of an application or script. The interception of system calls was done using *strace* command which tracks the system calls of a command and dumps the desired information to a file. Information retrieved by *strace* is dumped into a file and in our case we select only open system call:

strace -t -f -o dump.txt -e trace=open sh script.sh

The dump files are built for every application analyzed and they are parsed to obtain the files accesses. The list of all accessed files and the timestamps for these accesses are loaded into profile files. The format for the list of files is <file, timestamp> as you can see below:

/etc/ld.so.cache - 05:30:55
/lib/tls/i686/cmov/libc.so.6 - 05:30:55
/etc/ld.so.cache - 05:30:55
/proc/filesystems - 05:30:56
/usr/lib/locale/locale-archive - 05:30:56
/usr/share/locale/locale.alias - 05:30:56

Besides the list of the files accessed described above, also some statistics were done. A list with the number of accesses for every file form the sequence list is built with display format as: <file, accesses>. These statistics were done to optimize the caching algorithms utilized by CacheFS. Developing a caching algorithm that takes care of access statistics is included as a future work for this project.

V. Related Work

A similar approach for developing a hybrid file system was discussed at 2002 USENIX Annual Technical Conference and the title of the paper is Conquest: Better Performance Through A Disk/Persistent-RAM Hybrid File System.

Conquest is a research project which takes care of developing a file system that can simultaneously provide high file systems performance and energy efficiency. The key for reconciling these generally conflicting goals is based on the observation that RAM is becoming an inexpensive resource that can deliver all file system services with the single exception of large storage capacity. In essence, instead of storing most files on disks, Conquest stores them only in RAM.

Since most files can then be accessed directly from RAM, there is no need to keep the disk spinning. Only large files need to be kept on disk, and such files will be accessed infrequently. While RAM running in its normal mode consumes a great deal of power, RAM can be put into an extremely power-efficient mode and returned to its normal mode much more rapidly than disk can.

Our approach brings an additional data store, the solid-state drive which will be the second data store as priority after RAM. The arhitecture of our file system supposes two levels of data storage: RAM and a solid-state drive(SSD), SSD being used to store large files that cannot be kept into RAM. These files will now be stored to SSD instead of going to HDD. This approach

provides higher energy consumption than Conquest as the HDD can be turned off for long periods and also higher performance.

Higher performance is obtained through caching algorithms used for the two-level cache arhitecture to move data between the two data stores. The other optimization for the time access is the prefetching of files provided by Application Profile Builder.

VI. Future Work

Running the tests on a real machine, on an SSD, to judge the real performance brought by cacheFS, as well as the overhead involved by FUSE and the cacheFS code.

Finding a better method of timing the read operation.

Implementing better caching algorithms, tuned for specific applications.

An advanced application profile builder and analyzer, with the ability to detect file access patterns and an associated caching algorithm for taking advantage of it.

A more advanced simulator, taking into account more factors and capable of simulating advanced caching algorithms.

VII. Conclusions

After developing cacheFS, we can conclude that the use of a hybrid RAM-SSD caching filesystem is useful and can give a great speed increase for certain types of applications. Web and application servers that routinely process a large number of files can benefit from the speed and capacity provided by the SSD.

By storing all the files required for operation in the low-power-consuming RAM and SSD, the power-hungry disk arrays can be turned off, resulting in an increase in power saving. This is important as the energy costs of a data center are constantly climbing.

The implementation of cacheFS using FUSE was thought as a test bed for the caching algorithms. The idea was proven feasible. By using the right algorithms, the filesystem can be tuned for the maximum disk performance of an application. However, if FUSE will not support direct-I/O in the near future (bypassing the system cache) cacheFS will have to be redesigned to work in kernel space, for maximum performance and efficiency.

VIII. References

1. Conquest-2 project at UCLA - http://www.lasr.cs.ucla.edu/Conquest-2.html
2. Proceedings of the 2002 USENIX Annual Technical Conference. Conquest: Better Performance Through A Disk/Persistent-RAM Hybrid File System - http://www.usenix.org/events/usenix02/full_papers/wang/wang.pdf
3. FUSE – Filesystem in Userspace - http://fuse.sourceforge.net/
4. Linux Function Interception - http://uberhip.com/people/godber/interception/index.html
5. Caching Algorithms http://en.wikipedia.org/wiki/Cache_algorithms