UBC: An Efficient Unified I/O and Memory Caching Subsystem for NetBSD

Chuck Silvers
The NetBSD Project
chuq@chuq.com, http://www.netbsd.org/

Abstract

This paper introduces UBC ("Unified Buffer Cache"), a design for unifying the filesystem and virtual memory caches of file data, thereby providing increased system performance. In this paper we discuss both the traditional BSD caching interfaces and new UBC interfaces, concentrating on the design decisions that were made as the design progressed. We also discuss the designs used by other operating systems to solve the same problems that UBC solves, with emphasis on the practical implications of the differences between these designs. This project is still in progress, and once completed will be part of a future release of NetBSD.

1 Introduction

Modern operating systems allow filesystem data to be accessed using two mechanisms: memory mapping, and I/O system calls such as `read()` and `write()`. In traditional UNIX-like operating systems, memory mapping requests are handled by the virtual memory subsystem while I/O calls are handled by the I/O subsystem. Traditionally these two subsystems were developed separately and were not tightly integrated. For example, in the NetBSD operating system[1], the VM subsystem ("UVM"[2]) and I/O subsystem each have their own data caching mechanisms that operate semi-independently of each other. This lack of integration leads to inefficient overall system performance and a lack of flexibility. To achieve good performance it is important for the virtual memory and I/O subsystems to be highly integrated. This integration is the function of UBC.

2 Background

In order to understand the improvements made in UBC, it is important to first understand how things work without UBC. First, some terms:

- "buffer cache":

  A pool of memory allocated during system startup which is dedicated to caching filesystem data and is managed by special-purpose routines.

  The memory is organized into "buffers," which are variable-sized chunks of file data that are mapped to kernel virtual addresses as long as they retain their identity.

- "page cache":

  The portion of available system memory which is used for cached file data and is managed by the VM system.

  The amount of memory used by the page cache can vary from nearly 0% to nearly 100% of the physical memory that isn't locked into some other use.

- "vnode":

  The kernel abstraction which represents a file.

  Most manipulation of vnodes and their associated data are performed via "VOPs" (short for "vnode operations").

The major interfaces for accessing file data are:

- `read()` and `write()`:

  The `read()` system call reads data from disk into the kernel's cache if necessary, then copies data from the kernel's cached copy to the application's address space. The `write()` system call moves data the opposite direction, copying from the application's address space into the kernel's cache and eventually writing the data from the cache to disk. These interfaces can be implemented using either the buffer cache or the page cache to store the data in the kernel.

- `mmap()`:

  The `mmap()` system call gives the application direct memory-mapped access to the kernel's page cache data. File data is read into the page cache lazily as processes attempt to access the mappings created with `mmap()` and generate page faults.

In NetBSD without UBC, `read()` and `write()` are implemented using the buffer cache. The `read()` system call reads file data into a buffer cache buffer and then copies it to the application. The `mmap()` system call, however, has to use the page cache to store its data since the buffer cache memory is not managed by the VM system and thus not cannot be mapped into an application address space. Therefore the file data in the buffer cache is copied into page cache pages, which are then used to satisfy page faults on the application mappings. To write modified data in page cache pages back to disk, the new version is copied back to the buffer cache and from there is written to disk. Figure 1 shows the flow of data between the disk and the application with a traditional buffer cache.

This double-caching of data is a major source of inefficiency. Having two copies of file data means that twice as much memory is used, which reduces the amount of memory available for applications. Copying the data back and forth between the buffer cache and the page cache wastes CPU cycles, clobbers CPU caches and is generally bad for performance. Having two copies of the data also allows the possibility that the two copies will become inconsistent, which can lead to application problems which are difficult to debug.

The use of the buffer cache for large amounts of data is generally bad, since the static sizing of the buffer cache means that the buffer cache is often either too small (resulting in excessive cache misses), or too large (resulting in too little memory left for other uses).

The buffer cache also has the limitation that cached data must always be mapped into kernel virtual space, which puts an additional artificial limit on the amount of data which can be cached since modern hardware can easily have more RAM than kernel virtual address space.

To solve these problems, many operating systems have changed their usage of the page cache and the buffer cache. Each system has its own variation, so we will describe UBC first and then some other popular operating systems.

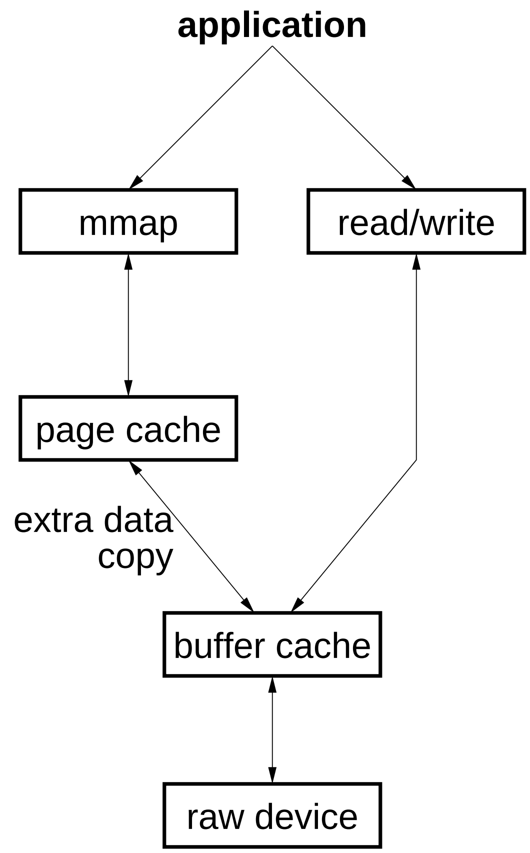

Figure 1: NetBSD before UBC.

3 So what is UBC anyway?

UBC is a new subsystem which solves the problems with the two-cache model. In the UBC model, we store file data in the page cache for both `read()`/`write()` and `mmap()` accesses. File data is read directly into the page cache without going through the buffer cache by creating two new VOPs which return page cache pages with the desired data, calling into the device driver to read the data from disk if necessary. Since page cache pages aren't always mapped, we created a new mechanism for providing temporary mappings of page cache pages, which is used by `read()` and `write()` while copying the file data to the application's address space. Figure 2 shows the changed data flow with UBC.

UBC introduces these new interfaces:

- `VOP_GETPAGES()`, `VOP_PUTPAGES()`

  These new VOPs are provided by the filesystems to allow the VM system to request ranges of pages

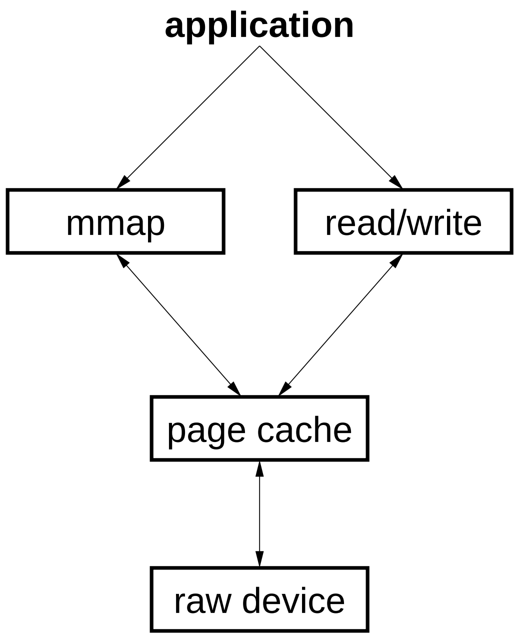

Figure 2: NetBSD with UBC.

to be read into memory from disk or written from memory back to disk. VOP_GETPAGES() must allocate pages from the VM system for data which is not already cached and then initiate device I/O operations to read all the disk blocks which contain the data for those pages. VOP_PUTPAGES() must initiate device I/Os to write dirty pages back to disk.

- `ubc_alloc()`, `ubc_release()`

  These functions allocate and free temporary mappings of page cache file data. These are the page cache equivalents of the buffer cache functions `getblk()` and `brelse()`[3]. These temporary mappings are not wired, but they are cached to speed repeated access to the same file. The selection of which virtual addresses to use for these temporary mappings is important on hardware which has a virtually-addressed CPU data cache, so the addresses are carefully chosen to be correctly aligned with the preferred addresses for user file mappings, so that both kinds of mappings can be present at the same time without creating coherency problems in the CPU cache. It is still possible for applications to create unaligned file mappings, but if the application lets the operating system choose the mapping address then all mappings will always be aligned.

- `ubc_pager`

  This is a UVM pager which handles page faults on the mappings created by `ubc_alloc()`. (A UVM pager is an abstraction which embodies knowledge of page-fault resolution and other VM data management. See the UVM paper[2] for more information on pagers.) Since its only purpose is to handle those page faults, the only action performed by `ubc_pager` is to call the new `VOP_GETPAGES()` operation to get pages as needed to resolve the faults.

In addition to these new interfaces, several changes were made to the existing UVM design to fix problems which were glossed over in the original design.

Previously in UVM, `vnodes` and `uvm_objects` were not interchangeable, and in fact several fields were duplicated and maintained separately in each. These duplicate fields were combined. At this time there's still a bit of extra initialization the first time a `struct vnode` is used as a `struct uvm_object`, but that will be removed eventually.

Previously UVM only supported 32-bit offsets into `uvm_objects`, which meant that data could only be stored in the page cache for the first 4 GB of a file. This wasn't much of a problem before since the number of programs which wanted to access file offsets past 4 GB via `mmap()` was small, but now that `read()` and `write()` also use the page cache interfaces to access data, we had to support 64-bit `uvm_object` offsets in order to continue to allow any access to file offsets past 4 GB.

4 What do other operating systems do?

The problems addressed by UBC have been around for a long time, ever since memory-mapped access to files was first introduced in the late 1980's. Most UNIX-like operating systems have addressed this issue one way or another, but there are considerable differences in how they go about it.

The first operating system to address these problems was SunOS[4, 5], and UBC is largely modeled after this design. The main differences in the design of the SunOS cache and UBC result from the differences between the SunOS VM system and UVM. Since UVM's pager abstraction and SunOS's segment-driver abstraction are similar, this didn't change the design much at all.

When work on UBC first began over two years ago, the other design that we examined was that of FreeBSD[6], which had also already dealt with this problem. The model in FreeBSD was to keep the same buffer cache interfaces to access file data, but to use page cache pages as the memory for a buffer's data rather than memory from a separate pool. The result is that the same physical page is accessed to retrieve a given range of the file regardless of whether the access is made via the buffer cache interface or the page cache interface. This had the advantage that filesystems did not need to be changed in order to take benefit from the changes. However, the glue to do the translation between the interfaces was just as complicated as the glue in the SunOS design and failed to address certain deadlock problems (such as an application calling `write()` with a buffer which was a memory mapping of the same file being written to), so we chose the SunOS approach over this one.

The approach taken by Linux[7] (as of kernel version 2.3.44, the latest version at the time this paper was written) is actually fairly similar to the SunOS design also. File data is stored only in the page cache. Temporary mappings of page cache pages to support `read()` and `write()` usually aren't needed since Linux usually maps all of physical memory into the kernel's virtual address space all the time. One interesting twist that Linux adds is that the device block numbers where a page is stored on disk are cached with the page in the form of a list of `buffer_head` structures. When a modified page is to be written back to disk, the I/O requests can be sent to the device driver right away, without needing to read any indirect blocks to determine where the page's data should be written.

The last of the operating systems we examined, HP-UX, takes a completely different stance on the issue of how to cache filesystem data. HP-UX continues to store file data in both the buffer cache and the page cache, though it does avoid the extra of copying of data that is present in pre-UBC NetBSD by reading data from disk directly into the page cache. The reasoning behind this is apparently that most files are only accessed by either `read()`/`write()` or `mmap()`, but not both, so as long as both mechanisms perform well individually, there's no need to redesign HP-UX just to fix the coherency issue. There is some attempt made to avoid incoherency between the two caches, but locking constraints prevent this from being completely effective.

There are other operating systems which have implemented a unified cache (eg. Compaq's Tru64 UNIX and IBM's AIX), but we were unable to find information on the design of these operating systems for comparison.

5 Performance

Since UBC is unfortunately not yet finished, a detailed performance analysis would be premature. However, we have made some simple comparisons just to see where we stand. The hardware used for this test was a 333MHz Pentium II with 64MB of RAM and a 12GB IDE disk. The operations performed were a series of "dd" commands designed to expose the behaviour of sequential reads and writes. We create a 1GB file (which is much larger than the physical memory available for caching), then overwrite this file to see the speed at which the data modifications caused by the `write()` are flushed to disk without the overhead of allocating blocks to the file. Then we read back the entire file to get an idea of how fast the filesystem can get data from the disk. Finally, we read the first 50MB of the file (which should fit entirely in physical memory) several times to determine the speed of access to cached data. See Table 1 for the results of these tests.

The great disparity in the results of the first four tests on the three non-UBC operating systems is due to differences in performance of their IDE disk drivers. All of the operating systems tested except NetBSD with UBC do sequential buffered reads from a large file at the same speed as reads from the raw device, so all we can really say from this is that the other caching designs don't add any noticable overhead. For reads, the UBC system is not yet running at device speed, so there's still room for improvement. Further analysis is required to determine the cause of the slowdown.

UBC obviously needs much improvement in the area of write performance. This is partly due to UVM not being very efficient about flushing modified pages when memory is low and partly because the filesystem code currently doesn't trigger any asynchronous writes to disk during a big sequence of writes, so the writes to disk are all started by the inefficient UVM code. We've been concentrating on read performance so far, so this poor write performance is not surprising.

The interesting part of this test series is the set of tests where we read the same 50MB file five times. This clearly shows the benefit of the increased memory available for caching in the UBC system over NetBSD without UBC. In NetBSD 1.4.2, all five reads occured at the speed of the device, whereas in all the other systems the reads were completed at memory speed after several runs. We have no explanation for why FreeBSD and Linux didn't complete the second 50MB read at memory speed, or why Linux didn't complete even the third read

Experiment			Run Time (seconds)			
Input	**Output**	**Size**	**NetBSD 1.4.2**	**NetBSD with UBC**	**FreeBSD 3.4**	**Linux 2.2.12-20smp**
raw device	/dev/null	1GB	72.8	72.7	279.3	254.6
/dev/zero	new file	1GB	83.8	193.0	194.3	163.9
/dev/zero	overwrite file	1GB	79.4	186.6	192.2	167.3
non-resident file	/dev/null	1GB	72.7	86.7	279.3	254.5
non-resident file	/dev/null	50MB	3.6	4.3	13.7	12.8
resident file	/dev/null	50MB	3.6	0.8	4.1	11.5
repeat above	/dev/null	50MB	3.6	0.8	0.7	4.5
repeat above	/dev/null	50MB	3.6	0.8	0.7	0.8
repeat above	/dev/null	50MB	3.6	0.8	0.7	0.8

Table 1: UBC performance comparison.

at memory speed.

6 Conclusion

In this paper we introduced UBC, a improved design for filesystem and virtual memory caching in NetBSD. This design includes many improvements over the previous design used in NetBSD by:

- Eliminating double caching of file data in the kernel (and the possibility of cache incoherency that goes with it) when the same file is accessed via different interfaces.
- Allowing more flexibility in how physical memory is used, which can greatly improve performance for applications whose data fits in physical memory.

7 Availability

This work will be part of a future release of NetBSD once it is completed. Until then, the source code is available in the "`chs-ubc2`" branch in the NetBSD CVS tree, accessible via anonymous CVS. See `http://www.netbsd.org/Sites/net.html` for details.

This being a work-in-progress, there is naturally much more work to do! Planned work includes:

- Integration of UBC into the NetBSD development source tree and performance improvement. The pagedaemon needs to be enhanced to deal with the much larger amount of page-cache data which will be dirty.
- Elimination of the data copying in `read()` and `write()` via UVM page loanout when possible. This could be done without UBC too, but with UBC it will be zero-copy instead of one-copy (from buffer cache to page cache).
- Elimination of the need to map pages to do I/O to them by adding a page list to `struct buf` and adding glue in `bus_dma` to map pages temporarily for hardware that actually needs that.
- Adding support for "XIP" (eXecute In Place). This will allow zero-copy access to filesystem images stored in flash roms or other memory-mapped storage devices.
- Adding support for cache coherency in layered filesystems. (The current UBC design does not address caching in layered filesystems.)

Acknowledgments

We would like to thank everyone who helped review drafts of this paper. Special thanks to Chuck Cranor!

References

[1] The NetBSD Project. The NetBSD Operating System. See `http://www.netbsd.org/` for more information.

[2] C. Cranor and G. Parulkar. The UVM Virtual Memory System. In *Proceedings of the 1999 USENIX Technical Conference*, June 1999.

[3] Marice J. Bach. The Design of the UNIX Operating System. Prentice Hall, February 1987.

[4] J. Moran, R. Gingell and W. Shannon. Virtual Memory Architecture in SunOS. In *Proceedings of USENIX Summer Conference*, pages 81-94. USENIX, June 1987.

[5] J. Moran. SunOS Virtual Memory Implementation. In *Proceedings of the Spring 1988 European UNIX Users Group Conference*, April 1988.

[6] The FreeBSD Project. The FreeBSD Operating System. See `http://www.freebsd.org/` for more information.

[7] L. Torvalds, et al. The Linux Operating System. See `http://www.linux.org/` for more information.