Paravirtualized Paging

Dan Magenheimer, Chris Mason, Dave McCracken, and Kurt Hackel
{first.last}@oracle.com

Oracle Corporation

Abstract

Conceptually, fast server-side page cache storage could dramatically reduce paging I/O. In this workshop extended abstract, we speculate how such a device might be used, then show how it can be implemented virtually in a hypervisor. We then introduce hcache (pronounced "aitch-cash"), our prototype implementation built on the Xen hypervisor and utilized by slightly modified Linux paravirtualized domains. We discuss the implementation and the current status of hcache, present some performance results, compare it to related work, and conclude with some speculation of other possible uses for hcache.

1. Introduction

Imagine a new very fast but somewhat quirky device that might someday become widely available on many systems; let's call it an "hcache" (pronounced "aitch-cash"). The device is essentially a very fast, page-granularity, fully-associative cache, which is so fast that DMA requests take no longer than a few times as long as a RAM-to-RAM copy. Thus an operating system can access the device synchronously, when a lock is held, and even when interrupts are disabled. The driver for this device might have the following very simple API, where a "handle" is a unique identifier determined by the operating system:

- hcache_put(page_frame, handle)
- hcache_get(empty_page_frame, handle)
- hcache_flush(handle)

The "put" function saves the data from the page and associates it with the specified handle. The "get" function finds a page in the hcache with the handle and fills the empty page frame with the data. The "flush" function disassociates the handle from any data so that subsequent "get" calls with that handle will fail.

Here's the quirky part: the size of the cache is unknown and cannot be determined. Sometimes a page "put" will be found by a "get" and sometimes not; it's impossible to tell a priori. However, like any cache, it's fast enough and large enough that using it is almost always a good thing.

How might an operating system use such a device? Since persistence is not guaranteed, dirty pages cannot be placed in the hcache, only clean pages. As a result, the hcache unfortunately can't be used as a general-purpose storage device. But it still has at least two interesting applications:

- Whenever the operating system is about to evict a clean page from its page cache, it can "put" the page to the hcache. And whenever the operating system is about to request that a disk driver DMA a page into a page frame, it would first try a "get" from the hcache to see if a prior "put" had saved the page in the hcache, thus saving the cost and latency of a disk access. Depending on access and eviction patterns, paging from disk may be greatly reduced.
- In a partitioned, containerized, or virtualized system running multiple operating systems, the hcache could be used as a quickly accessible copy of a read-only clustered filesystem. For example, if different partitions are running the same Linux operating system, the hcache might contain a copy of a commonly executed program such as the shell or compiler. After one partition promotes a page of the program from the disk to its buffer cache and "puts" it into the hcache, other partitions can "get" the copy from the hcache, thus similarly reducing paging from disk.

The reader is invited to suggest additional uses as there are certainly more.

2. A hypervisor-based cache ***in the hypervisor***

The physical device described in the previous section is only metaphorical, but represents a very realistic capability that can be implemented not using physical storage media, but instead with "spare" physical memory in the hypervisor of a virtualized system. This hypervisor-based cache -- or "hcache" -- can be accessed by a slightly-modified operating system using simple hypercalls and can be viewed by such operating systems as a second-chance page cache for evicted clean pages or by a cluster of operating systems as a shared server-side filesystem cache similar to, but much faster than, the cache RAM in a modern disk array.

In a virtualized system with multiple hcache-aware paravirtualized guests, available hcache memory should be divided equitably and dynamically. To each guest, hcache appears as a private page cache of unknown size but since no persistence guarantees are made, a mostly idle guest may be allocated a smaller portion of the hcache, or even none at all, while the allocation for a very active guest could be increased dynamically as needed. This is sort of a "fair share memory scheduler" for page cache space and could be controlled with internally derived policies, by administrator-supplied parameters, or by derivation from parameters provided for virtual machine CPU scheduling.

3. Hcache implementation

An hcache implementation has been prototyped with changes to a paravirtualized Linux guest and with code added to the Xen 3.3 hypervisor. To accommodate real operating system usage, the generic API has been extended in a number of ways: First, each domain can allocate multiple independent hcaches, and an explicit "initialize hcache" call has been added with a parameter indicating whether it is private or shared. (At the time of this writing, only the second-chance page cache mechanism has been implemented, so the shared-inclusive mechanism is not yet used.) Next, the handle has been divided into three components: a hcache identifier, a 64-bit object identifier and a 32-bit page identifier. These are roughly analogous to a "filesystem," a "file" and a page-granularity offset into a file. Finally, "flush object" call and a "flush hcache' call have been added to the API to simplify implementation of file-removal-like operations and filesystem "unmount".

Linux-side changes require the addition of "put" hypercalls at a single code location in the generic page cache removal code and a single "get" hypercall in the generic filesystem code. The difficult part is the correct placement of a handful of "flush" hypercalls to ensure that data consistency is maintained between the hcache and the operating system page cache. Fortunately, the potential race conditions are the same as for managing the page cache vs persistent storage, so are well understood. The object identifier is the Linux inode number and the index is the page offset into the inode. When Linux discards or truncates an inode, a flush-object hypercall is made and when a filesystem is unmounted, a flush-hcache call is made.

The Xen-side hcache code efficiently implements the basic get/put/flush/flushobject operations utilizing a hierarchy of dynamic data structures: A domain-private hcache is explicitly created when a filesystem is mounted or dynamically when the first hcache_put hypercall is performed with a page belonging to a filesystem. This hcache is implemented as a hashed-list of objects; each object is created as needed and serves as the root of a "radix tree" [1] of nodes for fast lookup of indices. The leaf nodes of the radix tree point to page descriptors, which in turn point to pageframes containing the actual data. The page descriptors are kept in two doubly-linked LRU lists: one private list for each domain, and one global list across all domains. Thus, unutilized pages can easily be recycled as needed to accommodate constantly changing "memory scheduling" needs. Finally, counters are kept for all data structures and pages are timestamped so that utilization can be easily determined and rebalanced as necessary.

When a guest performs an hcache_put hypercall, the Xen hcache code allocates an unused memory page and any necessary data structures and copies the page of data from the guest. If there is insufficient memory, one or more pages may be first evicted from either the global LRU list or private LRU list, depending on memory scheduler parameters and policy. If memory is still not available, the hcache_put simply fails -- since there is no guarantee of persistence, there is no requirement that a put is

successful; no indication of failure is even necessary, though one is provided in the hypercall return value.

For an hcache_get, the specified object identifier is hashed and the corresponding radix tree found. The radix tree is searched for the index and, if a match is found, the data is copied to the guest. In the case of a private-exclusive hcache_get, the page and associated data structures are then freed; for a shared-inclusive get, the page and data structures are left intact but the lists are updated to mark the page as recently used.

An hcache_flush is simply a private hcache_get with no copying. An hcache_flush_object walks the radix tree and flushes and frees all pages and data structures associated with that object. Finally, a function is provided to destroy and recyle an entire private hcache, so that memory can be proactively recovered when Xen destroys an entire domain; technically this is not necessary as all unused pages will eventually move to the end of the LRU queues and be evicted.

There are some interesting locking challenges, memory allocation issues, and hypercall sequence corner cases. For example, in a put-get-get sequence of the same handle, is it possible that the first get will fail but the second get will succeed? And what is the cause and proper response to a put when the handle already maps to existing data in the hcache? These are beyond the scope of this introductory abstract.

4. Hcache status

We have completed a prototype implementation of the second-chance cache functionality of hcache and the cluster/sharing functionality is currently under development. We have only as yet measured hcache with small workloads on a single domain; comprehensive testing will require multiple simultaneous virtual machines with real or simulated workloads. Still, preliminary results are promising.

We have heavily instrumented the hcache code in order to collect a large set of internal statistics for analysis; this has already pointed out some tuning opportunities we have fixed. For example, an unexpectedly high ratio of hcache_flush_object() calls led us to rewrite the linux-side interface to utilize inode numbers instead of the linux "address space" abstraction as an identifier for objects. This not only reduced hcache overhead, but also led to a cleaner linux-side implementation. Another example: Profiling hcache identified the Xen dynamic memory allocation ("xmalloc") code as a horrible bottleneck, driving worst case hcache call times into the millions of cycles. This led to the wholesale replacement of Xen xmalloc with a much faster TLSF-based [8] allocator, a change which has already been pushed upstream into xen-unstable.

5. Hcache performance

We have measured hcache on a simple but widely used "benchmark", compiling the Linux kernel. We test on two hardware platforms: a dual core 3GHz processor and 2GB physical memory; and a 2.9GHz quad core with hyperthreading and 4GB physical memory. The software foundation is an hcache-modified 64-bit Xen 3.3 hypervisor with Oracle Enterprise Linux 5.2 (OEL) as domain0. At boot, Xen absorbs about 42MB of memory and domain0 is restricted to 512MB via boot parameter. Our test domain is a 32-bit OEL guest configured to use a "tap:aio" virtual disk, with between 256MB and 2GB of memory and with either 2 vcpus or 4 vcpus.

For our workload, we use a "make -j 10" of linux-2.6.25.10 accelerated with the "ccache" [2] preprocessor. Our methodology is to measure five runs, discard the lowest and highest measurements and average the remaining three. We reset the environment before each compile with a "make clean" and a command to flush the page cache. We time the compile (only) rounded to the nearest second and bracket the compile with "iostat" to measure disk block reads and round this metric to the nearest thousand.

Table 1, at the end of the paper, shows our measurements. To briefly summarize, hcache on this workload reduces disk reads by nearly 95% and as a result increases throughput by between 22% and 50%, with best results when more CPU resources are available to the guest.

Some additional interesting data we gleaned from our instrumentation when hcache is enabled:

- hcache_get hit ratio is about 80%
- average cost for hcache get's and put's is about 2.5x the cost of an average page copy, which we measure at about 1.5usec on one platform and about half that on the other; maximum cost is about twice the maximum cost for a page copy
- about 80% of the 1.7M hcache calls do an hcache_flush, showing we may be over-paranoid on the linux side to guarantee data consistency, and so our implementation may still have room to improve
- hcache data structures are comfortably managing over 100K pages, belonging to 20K unique objects (inodes) in four hcaches (filesystems); note that this also reveals some insight into the working set size of the workload
- one hash table is seeing a maximum hash chain length of over 50 entries, showing yet another opportunity for improvement

Since the single guest, large memory environment, cold page cache, and the diskbound workload all favor hcache, some may argue that the benchmark is a bit contrived. To counter this concern, we provide a second set of test runs, where we disable compiler acceleration, remove the command to drop the page cache between compiles, and even "warm" the page cache with a pre-measurement compile. But we also reduce the guest memory size to simulate a poorly provisioned guest. As shown in Table 2, without hcache on a 128MB guest, some thrashing occurs and, as a result, disk reads climb dramatically and performance plummets. But with hcache enabled, performance is roughly the same as if the guest were properly provisioned with twice as much memory -- or greatly *over*provisioned with eight times as much.

Our intent is certainly not to claim that hcache will demonstrate such outstanding results on a much wider variety of environments and workloads, but rather merely to show that hcache has strong potential in some cases -- and more room to improve.

6. Related work

Lu and Shen [6] introduce the concept of a hypervisor-based page cache, which influenced the ideas behind hcache. However, cached pages in their implementation are stored not in the hypervisor but in the "service" domain (dom0), which requires costly interdomain transfer and coordination; this is because they do not constrain the cache to clean pages and must map and track physical device I/O performed in the service domain. The exclusiveness also obviates its use for sharing between multiple virtual machines.

Geiger [5] studiously avoids changes to the OS but uses a hypervisor to passively infer useful information about a guest's unified buffer cache usage, with goals of working set size estimation and improving hit rate in remote storage caches. Interestingly, Geiger's success is measured against "the ideal eviction detector" -- an OS modified exactly as needed for hcache.

Much of the excellent analysis in Wong and Wilkes [10] reapplies easily to hcache. Indeed, the DEMOTE operation is analogous to hcache_put, though the data is copied to a remote disk-array cache rather than a server-side hypervisor cache. In particular, we intend to try some of the same benchmarks and compare some of the resulting curves, and we are eager to attempt some of the adaptive cache insertion policies.

Finally, the transparent content-based page sharing described by Disco [3] and by Waldspurger [9] likely utilizes a hypervisor-cache-like mechanism to assist in memory overcommitment. We wonder whether the explicit white-box sharing we intend to employ with read-only clustered filesystems might prove superior on some consolidated workloads to the black-box copy-on-write mechanisms used in VMware ESX.

7. Conclusions and future work

We have introduced *hcache*, a hypervisor-based non-persistent page cache that allows underutilized memory to act as a "second-chance" page cache and, potentially, as a shared page cache for clustered filesystems. We have implemented a prototype of the second-chance cache functionality and have measured it against a single but non-trivial workload, demonstrating preliminary but surprising performance improvement potential in some environments and workloads.

We expect mixed benefit in other workloads: For example, large memory domains and applications with large sequential datastreams may not benefit much and will likely challenge the memory scheduler. Domains with limited memory, or high density consolidations using automatic ballooning techniques [7,9] may benefit much more. Indeed we are already considering combining self-ballooning with hcache to potentially better optimize memory utilization. Memory consumption may also be further reduced when hcache is leveraged to share read-only cluster filesystem data. And we speculate that Geiger's goals such as working set size estimation and remote storage cache hit rate improvement may be achieved more effectively with an hcache-based approach.

As hcache is applied to a wide variety of simultaneous workloads, we expect to focus on challenges in implementing policy code. For example, how do we balance hcache usage between multiple competing memory-hungry domains? Other less ambitious but potentially advantageous ideas for future work have been proposed: 1) Compress pages in the OS prior to cacheing [4], or optionally in the hypervisor. 2) Add an hcache_flush_range() hypercall to make file truncation more efficient. We are eager to hear additional ideas. 3) Use part of hcache memory to serve as a *ghost cache* [10] to assist in determining if increasing or decreasing cache size would be beneficial.

8. References

1. Bovet, D.P and Cesati, M. Understanding the Linux Kernel, Third Edition, O'Reilly & Associates, Inc. 2005
2. Brown, M. Improve collaborative build times with ccache, http://www.ibm.com/developerworks/linux/library/l-ccache.html
3. Bugnion, E., Devine, S., Rosenblum, M. Disco: Running commodity operating systems on scalable multiprocessors. In *Proc. 6th Usenix Symp. on Operating System Principles (SOSP'97),* pp 143-146, Saint-Malo France, October 1997.
4. Gupta, N. Compressed Caching for Linux http://code.google.com/p/compcache/
5. Jones, S.T., Arpaci-Dusseau, A.C., and Arpaci-Dusseau R.H. Geiger: Monitoring the buffer cache in a virtual machine environment. In *Proc 12th ASPLOS*, San Jose CA, October 2006.
6. Lu, P. and Shen, K. Virtual machine memory access tracing with hypervisor exclusive cache. In *Proc. 2007 Usenix Annual Technical Conference*, Santa Clara CA, June 2007.
7. Magenheimer, D., Memory Overcommit… without the commitment. Xen Summit 2008. http://wiki.xensource.com/xenwiki/Open_Topic_For_Discussion?action=AttachFile&do=get&target=Memory+Overcommit.pdf
8. Masamo, M., Ripoli I., et al. Implementation of a constant-time dynamic storage allocator. *Software Practice and Experience.* vol 38 issue 10, pp 995-1026. 2008.
9. Waldspurger, C.A.. Memory Resource Management in VMware ESX Server. In *Proc. 5th Usenix Symp. on Operating System Design and Implementation (OSDI'02),* pp 181-194, Boston MA, December 2002.
10. Wong, T.M. and Wilkes, J. My Cache or Yours? Making Storage More Exclusive. In *Proc 2002 Usenix Annual Technical Conference,* pp. 161-175, Monterey CA, June 2002.

physical cpus	*virtual cpus*	*guest memory (MB)*	*hcache enabled*	*time (s)*	*relative to hcache*	*disk reads (K)*	*relative to hcache*
2	2	256	***yes***	***51***	**--**	***6***	**--**
2	2	256	no	62	122%	98	1633%
2	2	1024	no	63	123%	98	1633%
4	2	256	***yes***	***45***	**--**	***6***	**--**
4	2	256	no	56	124%	98	1633%
4	2	1024	no	57	127%	97	1616%
4	4	256	***yes***	***26***	**--**	***6***	**--**
4	4	256	no	39	150%	98	1633%
4	4	1024	no	38	146%	98	1633%
4	4	2048	no	39	150%	98	1633%

Table 1. Linux compiles (using cold page cache and ccache) -- hcache is superior

physical cpus	*virtual cpus*	*guest memory (MB)*	*hcache enabled*	*time (s)*	*relative to hcache*	*disk reads (K)*	*relative to hcache*
4	4	128	***yes***	***53***	**--**	***323***	**--**
4	4	128	no	114	215%	537	166%
4	4	256	no	52	98%	18	6%
4	4	1024	no	52	98%	0	0%

Table 2. Linux compiles (with warm page cache and *not* using ccache) -- hcache compensates for underprovisioned memory