Flash Caching on the Storage Client

DAVID A. HOLLAND, ELAINE ANGELINO, GIDEON WALD, AND MARGO I. SELTZER

David A. Holland is a researcher in software systems at Harvard. His core research interest is figuring out how to write better software, which covers a broad range of projects and applications. He wrote the OS/161 instructional operating system. dholland@eecs.harvard.edu

Elaine Angelino is a Ph.D. student in Computer Science at Harvard SEAS. Her advisor is Professor Margo Seltzer. elaine@eecs.harvard.edu

Gideon Wald is an entrepreneur living and working in San Francisco. He was a product manager at Google for three years on Search and Chrome before leaving to co-found a nascent company in the enterprise software space. gideon.wald@gmail.com

Margo Seltzer is the Herchel Smith Professor of Computer Science at Harvard's School of Engineering and Applied Sciences, an architect at Oracle Corporation, and the current USENIX board President. Her research and commercial activities revolve around all sorts of systems: operating systems, database systems, file systems, learning systems, etc. margo@eecs.harvard.edu

Most use of flash memory for caching so far has been on the storage server side. Using a trace-driven simulator we examined the use of flash as a large client-side cache. We found that the benefit of such a cache derives chiefly from its size, not the persistence of flash; but persistent caches offer additional benefits. We also found that the cache can be write-through without harming performance, and that for some workloads it allows freeing up system RAM that would otherwise be needed for caching.

In recent years, flash memory has gained attention not only as a medium for storage but also as a component of storage system caches. Most such uses have been on the server side: flash deployed in direct combination with disks. Our study [1] examined the use of flash on the client side of a network, such as on the compute nodes in a cluster. This arrangement reduces access latency and network load at the cost of requiring a flash device on each node. For shared storage, it can also introduce cache consistency problems. We ran simulations to examine the range of possible designs of this type and their various costs and benefits.

In our system model (Figure 1), an application performs I/O into a RAM cache (the ordinary operating system disk cache), which connects in turn to a flash cache. These components access a file server across a network. Many scenarios, ranging from Web application servers to render-farm nodes, share this basic structure.

We treat the flash cache as a SATA-attached solid-state drive. PCI flash devices that behave like SATA-attached drives should give similar results. We also modeled the file server as a "smart" enterprise-grade filer with lots of fancy prefetching and caching logic. The flash cache will help plain disk arrays more as they are slower.

Design Space

We examined the tradeoffs that arise when designing a client-side flash cache. We asked four key questions: whether the flash cache can/should be write-through or write-back, the degree of integration with the operating system required, the cost/benefit of cache persistence, and the need for cache consistency management.

The motivating question for this study was whether the flash cache can be write-through. With a write-through cache, managing crash recovery and maintaining cache consistency is easier; however, write-back caches generally perform better. We wanted to know the magnitude of this effect.

Another question was whether the flash cache must be integrated with the operating system and the operating system's disk cache. An implementation that operates as an independent layer will be much easier to build and deploy; however, an integrated implementation might potentially perform much better, so we wanted to know what the tradeoffs would be.

The third question was whether the flash cache needs to survive crashes. A persistent cache must store recoverable cache metadata in the flash, as opposed to just using RAM; this creates additional overhead. On the other hand, as (re)filling a 64 GB cache to full effectiveness can take hours or even days, not making the cache persistent can lead to substantial periods

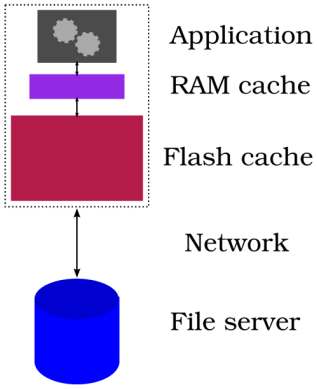

Figure 1: System model

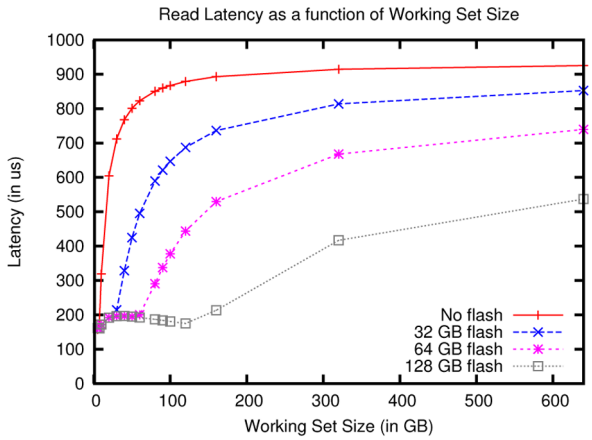

Figure 2: Application read latency as a function of working set size

of reduced performance. We wanted to find out how much the performance would be reduced, and for roughly how long.

Finally, we wanted to know what the consequences would be for cache consistency management. We were chiefly concerned with serving private disk images, but shared storage volumes are also important.

As the design space produced from these questions is enormous, we chose to do a simulator-based study that would allow us to explore these tradeoffs relatively inexpensively.

Our simulator reads a trace of I/O events, where each event is a read or write access to a particular region of a file, done by a specific thread on one of perhaps many hosts. The traces we used for our study were statistically generated using a tool we wrote for the purpose. We did use some real traces to validate the simulator against an existing implementation (NetApp's Mercury); this allowed us to be reasonably confident that the simulator was producing plausible results.

Results

Our first result was not the answer to a design question but a rather more basic issue: whether a client-side flash cache is a win. It is; a client-side flash cache provides a fairly substantial benefit, both for medium-sized workloads that fit into the flash but do not fit into RAM and for large workloads that do not fit into even a large flash device.

Figure 2 shows the average latency seen by the application for read operations (per 4096-byte block) for a range of workload sizes and four different flash sizes. This is with an 8 GB RAM cache; the workloads are 30% writes and 70% reads. At the bottom left where the workload fits into the cache, a large flash cache offers in-cache performance for much larger workloads than possible without it; on the right where the workload is 5x to 10x the flash size there is still a substantial benefit.

In this environment the file server's prefetching performance is critical. The application's read latency is dominated by reads that have to go all the way to disk. (This takes milliseconds and everything else is measured in microseconds.) If—by inserting a large cache in front of the filer—we hamper the filer's ability to prefetch, we can easily lose most or all of the flash cache's performance gain. We believe that adjusting the filer's internal tuning can avoid this effect; however, deploying client-side flash caches in front of an old filer that does not know how to cope may not provide the benefit that one might expect.

The flip side of this issue is that the ability of a plain disk array to prefetch is negligible under all circumstances compared to a filer. So when the backend is a plain disk array, the flash cache offers a much greater benefit.

Our first design question above was whether the flash cache could be write-through or whether this hampers performance. Also, the RAM cache needs to write data back to the flash cache; policies that work well for disks might not be appropriate in this environment. To investigate this we implemented four simple cache write-back policies:

- Synchronous write-through: block the app until the write to the next layer is complete.
- Asynchronous write-through: start writing to the next layer immediately, but do not block on it.
- Periodic: every so often a background thread writes out modified blocks.
- None: let the cache fill and write updates back only when evicting old blocks.

Trying four different time periods for the periodic policy gives seven settings each for the RAM and flash caches, making forty-nine cases in total.

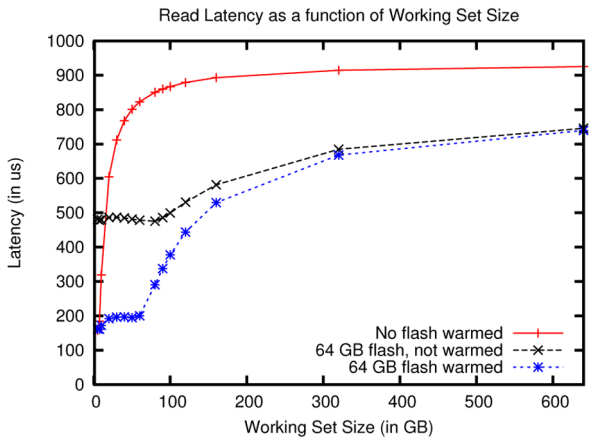

Figure 3: Effect of persistence on application read latency

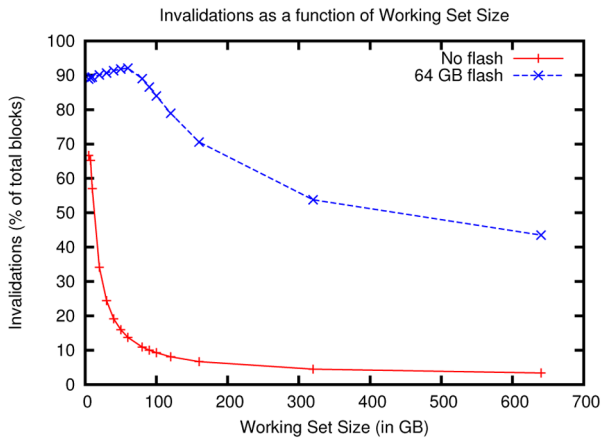

Figure 4: Invalidations required as a function of working set size

The obviously silly policies, such as synchronously writing from the RAM cache while the application waits, perform badly. ("None" turns into "synchronous" once the cache fills and also performs badly.) Otherwise, we found (somewhat to our surprise) that all other policies perform identically. The flash is large enough that as long as changes get written in some reasonable way, there is plenty of room for new incoming data.

Consequently, we did not try anything more complicated. The conclusion is that the flash cache can be write-through without hurting performance. This makes dealing with cache consistency for shared volumes much easier.

The second design question we addressed was whether the flash cache needs to be integrated with the operating system buffer cache. We compared the "naive architecture," in which the flash appears as an independent layer underneath the RAM cache with no integration whatsoever, and the "unified architecture," where the flash and RAM are fully integrated into a single cache framework. We found that the unified cache performed better for reads and worse for writes.

The chief difference between these models is that in the naive architecture the contents of the RAM cache become duplicated in the flash. The unified cache can avoid this and as a result becomes effectively larger. By tinkering with timings and settings, we ascertained that the improved read performance of the unified cache was exactly due to this effect. Given the price of flash compared to the assorted costs of implementing and deploying a unified cache, buying more flash is much cheaper.

Meanwhile, the worse write performance arose from an implementation issue: writes go to the next available block. With 8 GB of RAM and 64 GB of flash, 8/9 of the blocks are flash; the average write latency seen by the application was 8/9 of the flash write latency. A smarter implementation could hide this latency.

Persistence

Much of the benefit of using flash for caching comes simply from its size and speed; however, because flash is persistent, an obvious question is whether the flash cache should be persistent as well. As discussed earlier, this has both benefits and costs.

To approximate the performance overhead, we doubled the simulated time for writing to the flash: one write for the data and another write for metadata. This is pessimistic: in practice one can get away with much less metadata write traffic. There was no visible effect whatsoever on the application: given a reasonable policy for writing from the RAM cache to the flash cache, these writes happen in the context of the kernel's background processes and are fully hidden from the application.

To investigate the benefit, we ran the same workloads on warmed and unwarmed caches. Normally we use the first half of each generated I/O trace to warm up the cache and collect timing data on the second half. For the unwarmed case, which is equivalent to crashing right before starting the workload, we skipped the first half instead.

The results are shown in Figure 3. This graph requires some explanation. It shows application read latency for three cases: no flash cache, an unwarmed flash cache, and a warmed flash cache. In our study we pegged the total run size of our traces to the working set size; each trace pushes through a volume of twice the working set size during the measurement phase. Therefore, for the smallest workloads (left side of the graph) the trace finishes long before the flash cache fills, and the behavior shown on the graph is the performance seen during the warming phase. Moving to the right, the traces become far larger than a 64 GB flash, and the average behavior over the whole trace converges to the behavior with a warm cache.

What this shows is that the performance with a cold cache is considerably worse than with a warm cache, but the cache

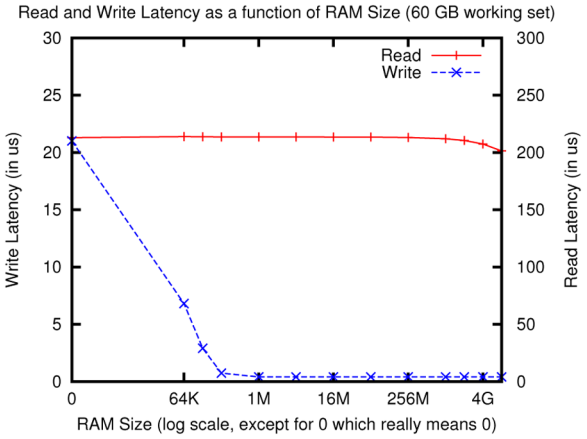

Figure 5: Application read and write latencies with small RAM sizes and 60 GB working set

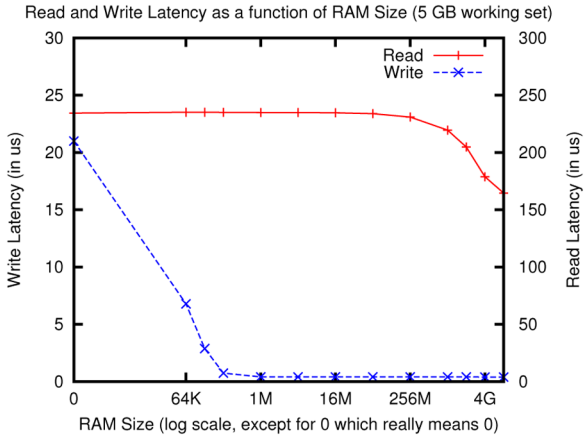

Figure 6: Application read and write latencies with small RAM sizes and 5 GB working set

warms rapidly enough that having it is still better for all but the very shortest and smallest workloads. In simulator time, the smallest workload in this graph completed in less than ten minutes; the largest took about a day. The cross-over point between the no-flash and cold-flash lines corresponds to roughly 20–25 minutes. How simulator time corresponds to real time in real-life workloads is not so clear. Twenty minutes of simulator time might correspond to several hours of real time, depending on the intensity and concurrency of the workload.

The conclusion, however, is that while making the cache persistent offers significant and noticeable performance gains, unless you plan to be crashing regularly it isn't necessary to realize much of the cache's performance gain.

Cache Consistency

As mentioned above we were primarily looking at serving private disk images; however, shared data is also important and cache consistency is a significant issue when handling it. This is a complex problem with complex solutions; we did not implement any particular cache consistency protocol in our simulator. Instead we used a simple scheme where the simulator took advantage of its own global knowledge to automatically invalidate stale blocks wherever they appeared. The results we have, therefore, do not take into account the network traffic generated by a cache consistency protocol; but they do take into account the overhead caused by needing to re-fetch blocks that have become obsolete.

Figure 4 shows the percentage of writes that incurred an invalidation over a range of working set sizes. This is for two hosts sharing the same working set (a fairly adverse situation); as elsewhere, this is with an 8 GB RAM cache and 30% of the I/Os are writes.

For workloads that fit into the flash cache, upwards of 90% of write operations cause an invalidation. This is much higher than without the flash, even for the smallest workloads that fit into RAM. And for larger workloads, the invalidation rate drops off much more slowly.

This effect is potentially enough to affect the performance or scalability of existing cache consistency protocols. An additional problem arises for persistent caches of shared data: a host that is offline and rebooting cannot participate in an online cache consistency protocol and would need to be able to catch up afterwards.

Our study and our materials do not really examine consistency issues in detail; further work, including a detailed implementation of one or more specific protocols, is probably indicated. But we can tentatively conclude that with shared data, particularly broadly shared data and particularly for write-heavy workloads, consistency management overhead may erase most or all of the benefit of the client-side cache.

No RAM Cache

We came across an additional unexpected phenomenon: in at least some cases, it appears that cutting back the amount of RAM used for caching to (almost) zero makes sense. Figure 5 shows the read and write latency seen by the application as the RAM size is reduced (moving right to left) from the default 8 GB down to 64 KB and then all the way to zero. For all points the flash size is 64 GB; the RAM-to-flash writeback policy has been changed to asynchronous write-through.

Notice that the write latency remains the same all the way down to 256 KB of RAM . . . and the read latency is effectively unchanged. The read latency is slightly worse compared to the largest RAM sizes, but this effect is negligible (around 2%).

Upon reflection one might expect this result, because the working set is much larger than the RAM size and the hit rate in the RAM cache is miserably low. (With 8 GB of RAM the hit rate is about 14%; the flash hit rate is over 85%.) The effect appears to a surprising extent even in small workloads. Figure 6 shows the same thing, but for a 5 GB workload. The far right point is for 8 GB of RAM, in which the working set fits completely. The penalty here is about 25–30%. This is substantial, but it is not necessarily fatal. There are almost certainly workloads where a 30% reduction in read performance is worth being able to repurpose 8 GB of RAM; for example, there are many applications where an extra 8 GB will more than offset this penalty.

This tradeoff is made possible by the flash cache; without the flash, the cost of shrinking the RAM cache is not merely 25–30%; reads become some *five times* slower.

One of the less obvious reasons for this effect is that in our workloads, like most real workloads, some accessed data is outside the working set. These I/Os tend to miss in normal-sized caches; the flash is large enough to help with them.

We should also stress that this is something of a preliminary result, in that we are not yet sure how well it will translate to real-life workloads in real-life situations. But it certainly bears consideration.

Conclusions

The results of our simulations show that even the simplest form of client-side flash caching provides significant benefits to applications. We also identified a number of points that simplify the space of designs worth pursuing. First, it is perfectly fine from a performance standpoint for the flash cache to be write-through, or to use any other reasonable write-back policy. Second, there is no need to integrate the flash cache tightly with the operating system; the benefit of doing so is purely that the cache becomes slightly larger, but it is much cheaper to buy more flash. Third, much of the benefit of the flash cache can be gained without making it persistent; however, persistence offers additional benefits, incurs little or no overhead in practice, and is probably worthwhile. Fourth, cache consistency becomes a serious issue with caches of this size if multiple hosts are actively modifying overlapping working sets. Even with a write-through cache, such workloads cause substantially more invalidation traffic than we see with traditional RAM-based caches. Traditional cache consistency protocols may also not be able to cope with a persistent cache being offline during a reboot.

Acknowledgments

This work was supported by NetApp. Additionally, James Lentini, Keith Smith, and Chris Small, all of NetApp, were tremendously helpful in providing us with the means and expertise to validate our simulator.

References

[1] D. A. Holland et al., "Flash Caching on the Storage Client," Proceedings of the 2013 USENIX Annual Technical Conference (San Jose, CA, 2013).

NOVEMBER 3–8, 2013 • WASHINGTON, D.C.

27th Large Installation System Administration Conference

Keynote Address: "Modern Infrastructure: The Convergence of Network, Compute, and Data" by Jason Hoffman, *Founder, Joyent*

Join us for **6 days of practical training** on topics including:

SRE Classroom: Non-Abstract Large System Design for Sysadmins by John Looney, *Google*

Root Cause Analysis by Stuart Kendrick, *Fred Hutchinson Cancer Research Center*

PowerShell Fundamentals by Steven Murawski, *Stack Exchange*

Introduction to Chef by Nathen Harvey, *Opscode*

The **3-day Technical Program** includes:

Plenaries by Hilary Mason, *bitly,* and Todd Underwood, *Google*

Invited Talks by industry leaders such as Ariel Tseitlin, *Netflix;* Jeff Darcy, *Red Hat;* Theo Schlossnagle, *Circonus;* Matt Provost, *Weta Digital;* and Jennifer Davis, *Yahoo!*

Paper presentations, workshops, vendor exhibition, posters, Guru Is In sessions, BoFs, and more!

New for 2013: The LISA Lab Hack Space!

Register by October 15 and save. Additional discounts are available!

www.usenix.org/lisa2013

Sponsored by

in cooperation with LOPSA