Flash Caching on the Storage Client

David A. Holland, Elaine Angelino, Gideon Wald, Margo I. Seltzer
Harvard University

Abstract

Flash memory has recently become popular as a caching medium. Most uses to date are on the storage server side. We investigate a different structure: flash as a cache on the client side of a networked storage environment. We use trace-driven simulation to explore the design space. We consider a wide range of configurations and policies to determine the potential client-side caches might offer and how best to arrange them.

Our results show that the flash cache writeback policy does not significantly affect performance. Write-through is sufficient; this greatly simplifies cache consistency handling. We also find that the chief benefit of the flash cache is its size, not its persistence. Cache persistence offers additional performance benefits at system restart at essentially no runtime cost. Finally, for some workloads a large flash cache allows using miniscule amounts of RAM for file caching (e.g., 256 KB) leaving more memory available for application use.

1 Introduction

Recently flash memory has become popular not only as a storage medium but also as a caching layer in high-end storage systems. The typical scenario has been to combine flash with disks, either locally or on a file server. We look at the opposite case: flash combined with the operating system's buffer cache, on the client side of a networked storage system.

We consider compute servers running storage-intensive workloads that are themselves clients in a networked storage environment. There are many examples of such servers: application servers in three-tier web applications, compute servers in data centers, render farms used in animation, and compute nodes in scientific computation clusters all fit this model. Our analysis explores a range of design issues arising from this configuration:

- Must the flash cache be managed together with the file system RAM cache or can it act as an independent layer below it?
- Should the RAM cache be a proper subset of the flash cache or should the two caches be treated as a single unified cache to avoid duplication?
- How large must the flash cache be relative to RAM?
- What writeback policies should be used from RAM to flash and from flash to the file server?
- Should a flash cache be persistent and recoverable?
- How critical is consistency across multiple caches?

This design space is already enormous, so we put aside other relevant but secondary considerations, such as cache replacement policy (we use LRU) and wear leveling algorithms. We assume our flash device comes equipped with a flash translation layer that handles wear leveling, erase cycles, and other considerations that arise if one uses raw flash chips directly.

We explore this design space via trace-driven simulation, which allows us to examine the behavior of an extensive range of configurations and cache sizes. We validated our simulator and traces against actual workloads, but use stochastically generated workloads for our analysis, because we could not find real-world traces with workloads large enough to stress the flash.

Our results show that all simple writeback policies, short of synchronously writing from RAM all the way through to the file server, produce comparable results. This means that flash caches can be write-through, which simplifies cache consistency handling. We also find the primary benefit of flash caching comes from its density. A volatile cache medium available for a reasonable price in similar sizes would also be attractive.

In the next section, we briefly discuss the various ways flash is being used to boost storage performance. We then outline the flash cache design space in Section 3. We describe our traces in Section 4 and our simulator in Section 5. We discuss how we validated our tools and models in Section 6 and then present the results of our simulation study in Section 7. Our conclusions are in Section 8.

2 Related Work

Flash is widely used in high end storage servers [2, 3] and more recently in hybrid drives that package flash and spinning media inside a single device [18, 20]. The NetApp FlashCache[17] is a device that transparently sits in front of a storage server, using the persistent cache to reduce latency. FlashTier [19] is a disk controller with an on-board persistent flash cache. It explores the possibilities of using a custom flash translation layer optimized for caching rather than storage. All of these solutions place flash on the storage side of a network (or local SATA), combining flash with disk drives. Our work examines flash on the client side, combining flash with the operating system buffer cache.

NetApp's Project Mercury [6] is a client-side flash cache that avoids explicit integration with the operating system. It is a block-level cache that can be deployed in various ways: a hypervisor filter driver, an OS filter driver, an application cache, or a proxy cache for network storage protocols. Mercury is one point in the design space this study explores. In Mercury, RAM stores a proper subset of the data stored in the flash cache, the writeback policy from RAM is the operating system's, and the writeback policy from flash is write-through.

Microsoft's ReadyBoost [15] is a software solution in recent Windows releases that uses a standard flash device as an extension to memory for random read caching. Windows gradually fills the flash cache with data and then services random reads from that cache, when doing so improves performance.

Recently, Koller et al. [11] experimented with a range of more sophisticated writeback policies for a flash cache. They found (as we did) that synchronous write-through all the way to disk is slow. Their work is otherwise complementary to ours as it explores write-back policies more sophisticated than those we considered. (They found, for example, that their policies can increase write throughput by improving the batching of back-end write requests; our simulator does not model this effect.) One key difference is that they were working in an environment where applications wait until writes propagate all the way to disk. We concentrate on a more conventional environment where writes return to the application once the data is written into the operating system's buffer cache. As we will see, this hides the write latency of the underlying storage tiers except under heavy write traffic. We also assume a high-performance filer with sophisticated read-ahead, nonvolatile cache, and large server memory at the back end, rather than a simple disk array.

3 Flash Cache Design Space

We model an application server environment consisting of one or more compute servers ("hosts") and a file server ("filer") connected by private network segments. Each host runs one or more applications, involving one or more threads of execution. Each host has cache space that is partially RAM and partially flash. As previously mentioned this environment reflects a number of real-life situations. We consider storage-intensive workloads.

We now address the design issues from Section 1.

3.1 Flash-RAM Integration

We begin by asking whether flash cache support should be integrated into the operating system's buffer manager or if it performs acceptably as an independent entity, as in Mercury. The former case requires substantial kernel modifications. The latter case allows deploying the flash cache in (or as) a self-contained device driver.

The need for integration depends on the level of coordination required between the RAM and flash caches. If accessing the flash via ordinary block reads and writes performs adequately, the flash cache can be independent. On the other hand, if special policies are required, or extra metadata must be provided to the flash cache, then kernel support is required.

3.2 Placement

Our second design question is whether the RAM cache can be a subset of the flash cache. This is effectively a choice of block placement policy. The straightforward approach is to structure the flash cache as an additional independent tier of cache below the RAM cache. The flash cache services the RAM cache and the file server services the flash. Newly referenced blocks are first placed in flash, then into RAM; the RAM cache is always a subset of the flash cache. This policy wastes some of the capacity of the flash, but is relatively simple.

Alternatively, one could use two separate layers of cache, but choose some more elaborate policy; for example, one might place blocks initially into RAM and

then migrate less recently (or less frequently) used blocks down to flash. Another option is to treat the two stores as a single unified cache and come up with some policy for initial placement and perhaps also internal migration.

The basic question is whether the simple approach is good enough. We would also like to estimate how much better (if at all) an alternate placement scheme performs.

3.3 Cache Architecture

We handle integration and placement as a single choice of cache architecture. Because the number of possible fill and migration policies is near infinite, we chose three simple alternatives to implement and test. Other options are certainly possible and may be a worthwhile subject of future research. These are the three architectures:

- **Naive.** The flash cache is treated as an independent cache layer beneath the RAM cache; the RAM cache is always a subset of the flash cache, requiring no integrated management.
- **Lookaside.** Based on Mercury [6], writes go directly from RAM to the file server instead of being routed through the flash. The flash is updated *after* the file server and never contains dirty data. Applications see persistence guarantees identical to a system without flash. The RAM cache is a subset of the flash cache, requiring no integrated management.
- **Unified.** RAM and flash are managed together using a single LRU chain. Data blocks are placed into the least recently used buffer, whether RAM or flash, and are never migrated. No attempt is made to prefer RAM to flash. Here the RAM cache is not a subset of the flash, so integrated management is needed.

3.4 Relative Size

What size does the flash cache need to be relative to the RAM cache to be effective? We use 8 GB as the baseline RAM size and examine flash sizes ranging from 8 GB to 128 GB (1x to 16x RAM). We use 64 GB as the baseline flash size based on the old rule of thumb that each successive layer of cache should be roughly an order of magnitude larger. (Note that the RAM size actually reflects the amount of RAM available for file system caching. For many real-life workloads this is substantially smaller than the total amount of RAM in the machine.)

3.5 Flash Writeback Policy

We next consider the question of when dirty blocks move from flash to the file server. We chose four policies:

- *write-through* - data is immediately written to the server, blocking the requester until completion.
- *asynchronous write-through* - data is immediately written to the server without blocking the requester.
- *periodic* - dirty data remains in the cache until a syncer thread flushes the data back to the server.
- *none* - dirty data remains in the cache until evicted for capacity reasons.

We run the periodic case with syncer periods of 1, 5, 15, and 30 seconds, resulting in seven different policies.

3.6 RAM Writeback Policy

We now consider RAM writeback policies. Since (at least for the *naive* architecture) these writebacks go to the flash cache, it does not necessarily follow that the standard behavior of file system RAM caches is correct.

We tested the same seven writeback policies that we used for flash writeback, yielding 49 different policy configurations for each of the three architectures.

We did not try other more elaborate policies (such as trickle-flushing, writing back asynchronously after a delay, etc.) for either flash or RAM, because we found that nearly all the policy combinations perform identically.

3.7 Cache Persistence

Volatile RAM caches are emptied by system restart and are typically left to refill naturally. However, a cache kept in persistent memory can potentially be recovered after a crash, to avoid the performance degradation that occurs when refilling the cache [12]. The Rio File Cache research prototype demonstrated the potential of such approaches as early as 1996 [7]. Today, the NetApp Mercury cache exploits persistence to avoid performance degradation after reboot [6], and high end file servers typically use battery-backed memory similarly to accomplish such warm restarts [1, 2]. With flash caches, cached data can survive a restart, but the system must take precautions to ensure that the data is valid.

Our results show that the price/performance of flash makes it attractive simply as a larger cache. However, taking advantage of its persistence can provide additional benefit. There are three chief obstacles: First, cache consistency needs to be maintained; this is discussed in the next section. Second, the cache indexing structures must themselves be kept in the flash and kept up to date and consistent with the data blocks in the flash. This creates additional flash traffic and additional overhead. A naive implementation adds an additional flash write latency every time the flash cache is updated; a clever implementa-

tion can batch those writes. Third, if the crash was caused by corruption in the flash itself, a simple reboot may not be sufficient to restore the system to a running state.

In the *lookaside* architecture blocks in the flash are never dirty, so the system cannot crash with dirty blocks that *must* be recovered and written back to the file server.

3.8 Cache Consistency

Normally one writes updated blocks in the RAM cache back to the file server quickly, because RAM is volatile. This motivation disappears with a persistent cache. If the flash cache is recoverable, as discussed in the previous section, cache writebacks can be delayed. Some writes will then die in the cache, reducing network contention.

However, for shared data, it also complicates cache consistency handling. Data not written back to the file server right away must still be reported back to the server so other hosts do not read stale versions. And, of course, unmodified data retained in the cache must also be tracked in case some other host updates it.

Cache consistency is not a new problem [9, 16, 21] and does not need a new solution; however, two new issues arise. The size of flash caches may affect the scalability of consistency protocols; detailed modeling of this effect is beyond the scope of our work. Furthermore, a recoverable cache is unavailable during a reboot; it cannot flush dirty data or participate in cache consistency protocols until afterwards. As reboots typically take at least minutes, this may induce unacceptable delays.

We concentrate primarily on non-shared data, e.g., disk images provided to clients over a SAN. We touch briefly on cache consistency only to quantify the magnitude of the problem. The simulator invalidates stale copies of blocks instantly (using global knowledge) when a new version is first written into a cache. This exposes the overhead caused when these blocks must be fetched again later. However, we only count invalidations; we do not model the overhead of cache consistency traffic, nor do we adopt any particular real-world cache consistency model. This information gives designers a basic overview of the circumstances that arise with the much larger caches that flash allows.

4 Traces

For our trace-driven simulation, we use block-level traces containing read and write operations. Each operation identifies a file and a range of blocks within that file. Each operation also carries a thread ID and host ID.

During development and validation, we used traces from the SNIA repository and the Mercury traces, but for our analysis we use synthetic traces. Adequately large real traces are, by and large, not available; when working with a 128GB flash device, we need a trace that churns through enough data to fill it and then work with it for long enough to access plenty of data that both is and is not in the original fill. The largest trace for which we present results moves roughly 2.5 TB of data, all told; we were unable to locate any real traces this large.

We wrote a trace generator to produce large traces with characteristics similar to real traces. The trace generator starts from a list of files and file sizes from the Impressions file system generator [4]. It samples this file server model to produce working sets, then samples these to produce I/O requests. A portion of the I/O requests are sampled instead from the whole file server. The distribution of I/Os among hosts and threads is uniform; the distribution of I/Os among files (and selection of files for working sets) is weighted by popularity, where small integer popularities are generated from a Zipfian distribution. The distribution of I/O sizes (and selection of file subregions for working sets) is Poisson, modified by clamping to the filesize. The distribution of I/O starting points (and file subregion starting points) is uniform.

All traces used in the results presented are based on the same 1.4 TB file server model we generated with Impressions. (This is larger than any of the cache sizes we use.) They use 4K blocks and have 80% of the I/Os coming from the working set. They also use eight threads per host. They grind through a total volume of data that is, in all cases, four times the working set size, half of it being devoted to a warmup period for which statistics are not collected. This ensures the cache fills thoroughly. (We checked the results of changing the working set percentage and the number of threads; these did not affect the conclusions about our key questions.)

The two traces we use as a baseline use one host, one working set, working set sizes of 60 and 80 GB (for use with a 64 GB flash), and 30% writes. For many of the experiments we vary one or more of these parameters.

5 Simulator

As discussed earlier, we model an environment where some number of computation servers ("hosts") share a single networked file server. We wrote a trace-driven simulator for this environment.

The simulator issues I/O requests from the trace as quickly as possible given that each application thread can have only one I/O in progress. I/O requests may stall at

various points in the system; all executions are fully interleaved. We do not try to produce realistic application-level I/O schedules; not only is scheduling I/O traces a known hard problem [10, 14, 22], but flash substantially changes the timing. Timestamps taken from environments without flash would have dubious value.

We model the caches in detail; each is a single LRU chain of blocks. We treat the flash itself as a block device; that is, we write blocks to it and read them back. We assume a flash translation layer but do not model it directly. We use average per-block access times derived from testing real flash devices. (See Sections 6.1 and 6.2.)

The network is modeled less exactly: each segment can carry one packet at a time, and each I/O request uses one packet in each direction. Each packet is assumed to incur a fixed latency (for headers, block information, and so forth) plus a small amount of additional time per bit of block data transferred.

We do not attempt to model the caches or prefetching behavior of the filer directly. Many man-years of effort have gone into providing high-end file servers with clever and aggressive caching logic, and modeling this is irrelevant to the main goals of this work. Instead we use a simple model: a "fast" latency for cache hits, a "slow" latency for misses, and a prefetch success rate that determines what fraction of reads are fast. (*Which* reads are fast is random. Writes are buffered and always fast.)

We do not model application overhead, user-kernel transitions, hypercall delays, processing latency in the nework stack, etc. Most of these are invariant under caching or can be incorporated elsewhere.

6 Validation

We validate two parts of our system that could produce fallacious results if not done properly. First, we validate our simulator against data using NetApp's Mercury flash cache. Second, we validate that average read/write latencies for our device reasonably approximate actual flash latencies.

6.1 Simulator Validation

We validated our simulator against NetApp's Mercury [6], a hardware implementation of a client-side flash cache. Working with the Mercury group, we took four days of traces from a NetApp Windows laptop and played them back both on their hardware and on our simulator. These traces were collected below the file system, i.e., under the buffer cache, so we played them back directly through a 32GB flash cache. (In our simulator, that means we set the RAM cache size to zero.)

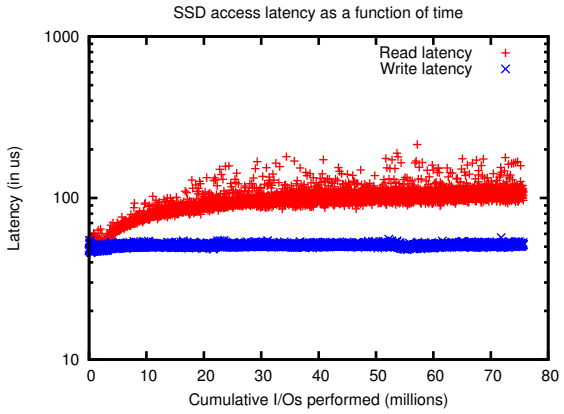

Figure 1: Flash device read (top) and write (bottom) latency; 60GB working set workload on a 58GB device. Each point is the average of 10,000 block I/Os.

We debugged the simulator and adjusted our timing models as necessary until the I/O throughput and latencies seen above and below the flash cache, as well as accessing the flash device, plus the cache hit rates, all or nearly all matched within 10%. Many of the statistics matched more closely. A perfect alignment is not possible, because (besides the inherent limitations of simulators) Mercury is not structured identically to the simulator. The simulator also does not account for an additional application-level or other systemic overhead of roughly 10% seen in the end-to-end run times.

These measurements gave us confidence that the simulator accurately models the system behavior and that its results are meaningful.

6.2 Flash Modeling Validation

We worried that average write latencies might not adequately model the behavior of a real device in the presence of flash erase cycles. We bought two low-end consumer grade SSDs and evaluated their latency behavior.

We modified the simulator to log I/Os to the flash as it ran and captured the results for a variety of workloads. Then we replayed these I/Os to the SSDs and recorded the actual read and write latencies. We also tried fully random reads and writes with a read/write mix similar to that found in the simulator logs.

We found three things of possible interest. First, while both devices exhibited high variance in their access latency, this variance is short-term; across a group of 10,000 to 100,000 block accesses (much less than the length of our traces) the variance is high, but from group

to group the average behavior is quite reasonable. Second, and perhaps of more interest, both devices maintained a single average write latency from beginning to end across essentially all the workloads. This included workloads with up to 90% (application) writes. Only the read latency fluctuated significantly over time as the device filled. We observed a weak relationship between higher write volumes and worse read performance; whether this is due to erase cycles or caching or some other internal phenomenon is anyone's guess.

Third, the read performance replaying the simulator logs is much better than the read performance doing purely random I/Os. Caching workloads are not random.

Figure 1 shows a scatter plot of the read and write latencies against time for a typical workload run. Each point is the average of 10,000 block I/Os.

Our conclusion was that a single average access latency is fine for modeling writes, and viable, though not ideal, for reads. However, our experience with flash devices is that each model is different, exhibiting its own average latencies and behavioral quirks. Fortunately the system performance does not appear to be highly sensitive to flash performance; see Section 7.7.

7 Results

We chose a per-block RAM access time of 400 ns, corresponding to roughly 10 GB/sec memory bandwidth. An internal limitation of the simulator restricts it to integer multiples of 100 ns, so this speed roughly reflects the 10-12 GB/s expected (and observed on an Intel Core i7 [13]) bandwidth of DDR3 RAM.

We used the performance data from validating against Mercury to choose timing models for the flash and the combined network and file server accesses. We then picked latencies loosely corresponding to a gigabit network for the network and attributed the rest of the combined network and file server times to the file server. Table 1 summarizes the timing parameters.

In evaluating possible configurations, we use the latency experienced by the application as the governing metric. Although the simulator captures a variety of other metrics (including throughput and latencies at every level of the stack), we use those only to explain behavior rather than to evaluate policies.

Parameter	Value
RAM read	400 ns / 4K block
RAM write	400 ns / 4K block
Flash read	88 μs / 4K block
Flash write	21 μs / 4K block
Network base latency	8.2 μs / packet
Network data latency	1 ns / bit
File server fast read	92 μs / 4K block
File server slow read	7952 μs / 4K block
File server write	92 μs / 4K block
File server fast read rate	90%

Table 1: Timing Model Parameters

7.1 Architecture and Writeback Policy

We begin our analysis by evaluating our *naive*, *lookaside*, and *unified* architectures and how they are affected by the 49 combinations (seven each for RAM and flash) of writeback policies. Identifying the promising configurations from among the 147 possibilities allows for a more focused comparison in the rest of the evaluation.

We used the two baseline traces described in Section 4. We ran these traces on the corresponding baseline simulator configuration: 8 GB of RAM and 64 GB of flash.

Figure 2 shows the average read and write latency seen by the application across all 49 policies for the three different architectures. We show the 80 GB workload; the 60 GB graphs are nearly identical.

Cursory inspection of the figures reveals the first important result: excepting policies that result in synchronous writes to the filer (synchronous or none) the writeback policy does not matter. The "none" policy leads to synchronous evictions once the cache fills. When the RAM policy allows this effect in the flash cache to show through to the application, as seen in the front left and right corners of the write latency graph, multiple threads doing evictions contend for the network, convoy, and slow down to (less than) the speed of the file server.

While this result initially surprised us, it is entirely reasonable: flash caches are so large that any reasonable writeback policy maintains an ample supply of clean blocks to evict and replace; the latency exposed above the flash cache is never greater than the flash write latency.

For the application to observe greater latency, it would have to sustain a write bandwidth greater than the writeback bandwidth to the file server for sufficiently long to fill many gigabytes of flash with dirty blocks. While workloads exhibiting this behavior probably exist, we expect them to be rare. Furthermore, upon filling the flash, write latency will largely revert to that of the file server. This produces the same effect as having no flash cache.

Based on this exploration, we use one policy combination for most of the remaining analysis: a one-second periodic RAM writeback policy (as this most closely matches real system behavior) and asynchronous write-

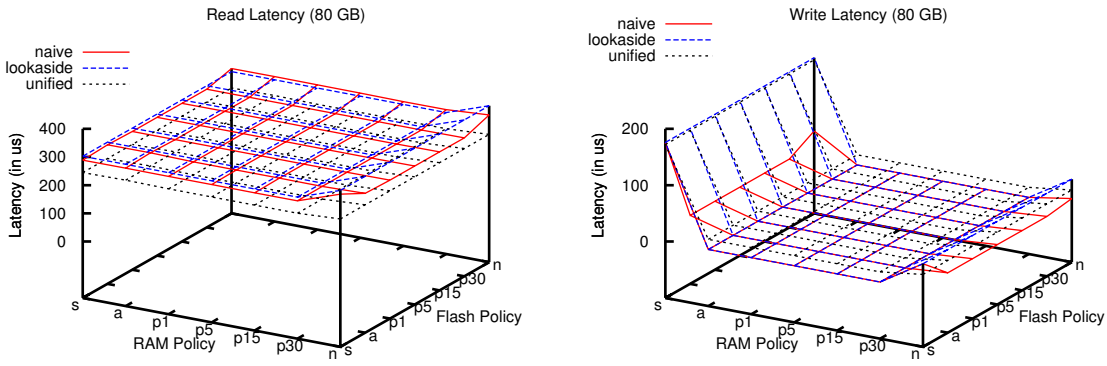

Figure 2: Application read and write latency on the 80 GB working set as a function of RAM and flash writeback policies.

through for the flash cache. Asynchronous write-through seems like the best overall choice for the flash, as it is equivalent to synchronous write-through for consistency and integrity purposes. Meanwhile it avoids exposing synchronous file server writes if the RAM cache becomes synchronous through dysfunction, e.g., thrashing.

Figure 2 also shows the *unified* architecture produces the lowest read latencies while the *naive* and *lookaside* architectures produce the lowest write latencies. The read latency results are unsurprising, because the effective capacity of the *unified* architecture is greater: it is the sum of the RAM and flash sizes (72 GB) instead of just the flash size (64 GB). When the working set fits in the flash (60 GB), the difference is tiny, only 3.5%. However, when the working set falls out of the flash (80 GB), we see that the larger effective cache size produces a significant benefit, improving read latency by as much as 20%. Figure 3 illustrates in more detail how the effective total cache size affects performance. For two of the cases in this graph we pretended that the flash has the same access latency as RAM. This allows distinguishing the structural effects from the latency properties of the cache materials. Although it is difficult to see in the graph, the performance of the RAM-only *unified* architecture with 8 and 56 GB caches is identical to that of the RAM-only *naive* architecture with 8 and 64 GB caches. The difference between that line and the one above it reflects the effect the slower flash has on read latency.

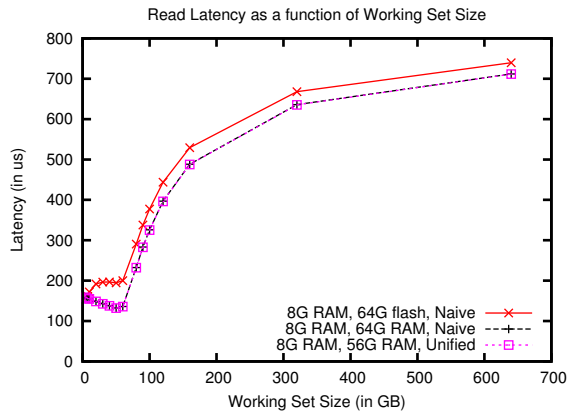

Figure 3: Application read latencies comparing effective cache sizes. See discussion in text.

Returning to the policy comparison in Figure 2, on the write side, the *naive* and *lookaside* architectures perform at RAM speed, because all writes go directly to RAM (except for very high write rates). The *unified* architecture also exposes flash latency by nature; since only $1/9$ of the data is placed in RAM and the rest in flash, on average we see $8/9$ of the 21 μs flash latency.

Stepping back, these results suggest that for read performance, bigger is better and that for write performance, the key is to avoid exposing applications to the flash timing. If we assume a given cost budget, an attractive strategy is to use only enough RAM to act as an effective write buffer and then buy as much flash as the budget allows. We explore this option in Section 7.5. Unless otherwise specified, we use the *naive* architecture in the remaining analyses, as it hides the flash write latency and offers the simplest implementation alternative.

7.2 Flash vs. No Flash

Having settled on policies, we now investigate the advantage the flash cache offers. To this end we ran a range of working set sizes, ranging from 5 GB to 640 GB, on three sizes of flash cache (32 GB, 64 GB, and 128 GB)

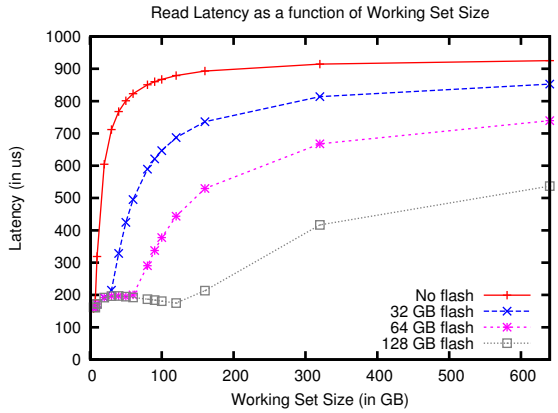

Figure 4: Read latencies as a function of working set size across a variety of flash sizes. As expected, when the working set fits in the flash, read latency improves dramatically over a RAM-only system.

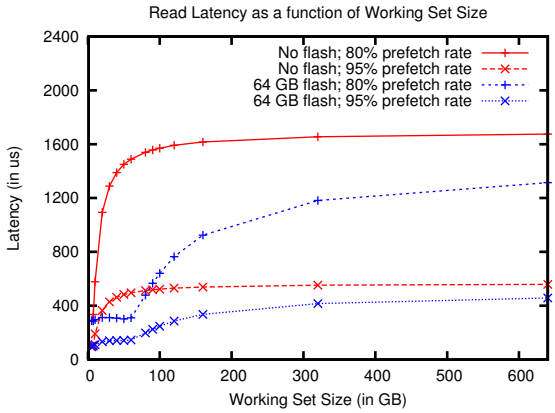

Figure 5: Application-level read latency for different workload sizes and two filer prefetch rates. Comparing the lines of similar shape demonstrates the dramatic effect that filer prefetching has on the resulting latency.

as well as with no flash cache. The RAM cache size is 8 GB. The working set sizes range from smaller than RAM to substantially larger than the largest flash cache.

Figure 4 shows that even when the working set far exceeds the flash size, the flash improves performance significantly, because the difference between flash performance and filer performance is substantial. In all configurations, the RAM hit rate is only 3.4%, but the flash hit rate varies from 0 (with no flash) to 47% in the 128 GB configuration. Although the filer fast read time (92 μs) is quite close to that of flash (88 μs), the two orders of magnitude difference between fast and slow filer read times is significant, even with the 90% fast filer read rate. As we shall see in the next section, the filer's ability to read ahead is critical in any configuration. The write latency figures from this experiment are not interesting: all writes see the RAM write latency of 0.4 μs.

7.3 Filer Read-Ahead

An effect observed in Mercury [6] suggests that a large cache reduces the file server's ability to prefetch data. We cannot yet quantify this effect, but we can bound it. In Figure 5 we show the spread between an 80% prefetch rate, which we believe to be a reasonable lower bound, and a 95% prefetch rate, which serves as a plausible upper bound. The graph shows the spread for the 64 GB flash, as well as for no flash, using the same range of working set sizes used in the previous section.

The application read latency is dominated by the cost of file server misses, which cost milliseconds. In an ideal world, installing the flash cache would not affect the file server's prefetch ability. Then the flash cache is beneficial for almost all workload sizes, as can be seen in the figure. In a pessimal world, the prefetch rate might drop substantially; in this case the cache is beneficial for a much narrower range of workloads: those that fit in flash but not in RAM. This can be seen in Figure 5 as the pocket between the lower (better) no-flash curve and the upper (worse) with-flash curve.

Avoiding the pessimal world is an engineering challenge and a critical issue for the adoption of flash caching. In the presence of a flash cache, the filer cache transitions from a second level cache to a third level cache; its prefetching and replacement policies must therefore adapt accordingly [5, 8, 23].

However, in environments where the back end is not a filer but a plain disk array [11], the prefetch rate will be negligible and a flash cache is a huge win.

7.4 Flash Cache Size

We next examined the converse case: given a fixed workload, what happens as we increase the flash cache size. As expected, the read latency decreases as a greater portion of the working set falls in the cache until the flash cache is large enough to capture the entire working set, at which point the read latency is that of flash. As there is nothing unexpected in these results, we have omitted the corresponding graphs.

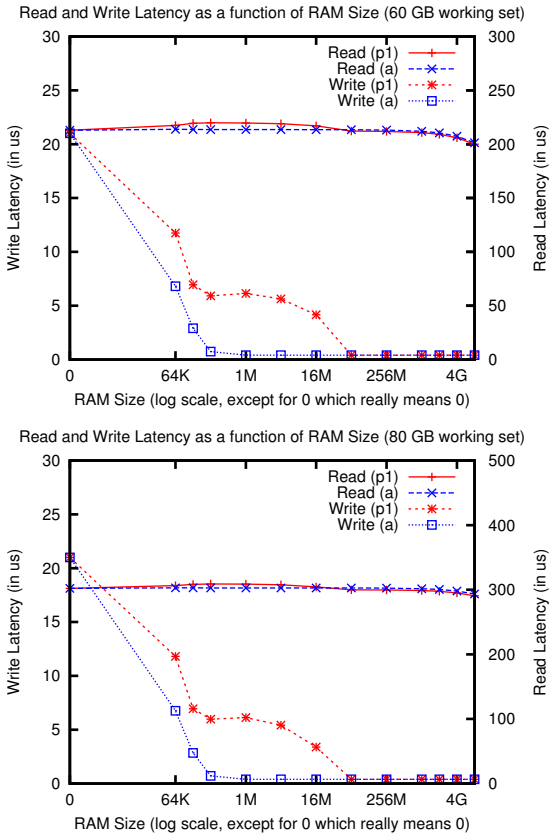

Figure 6: Application read and write latencies with small RAM cache sizes. The (a) and (p1) notations in both graphs refer to the RAM write-back policy: asynchronous write-through and 1-second periodic respectively. Surprisingly, a small (256 KB) cache achieves performance comparable to much larger ones.

7.5 No RAM Cache

One intriguing possibility suggested by the previous results is to dispense with the RAM cache entirely. We run the baseline workloads with a fixed 64 GB flash cache and RAM cache sizes ranging from zero to the baseline 8 GB. We run these with both the asynchronous write-through RAM policy (a) as well as the default 1-second periodic writeback (p1) we chose above.

Figure 6 shows the application read and write latencies for the 60 GB and 80 GB working sets, respectively. The X axis is the base 2 log of the RAM size or zero for none.

The no-RAM configuration does not work well, but it is surprising how well a relatively small (e.g., 64 MB) RAM cache performs. If we use the asynchronous write-through policy, a tiny 256 KB is sufficient as a write buffer. For the smallest caches the periodic syncer does not run often enough, so the RAM cache fills with dirty

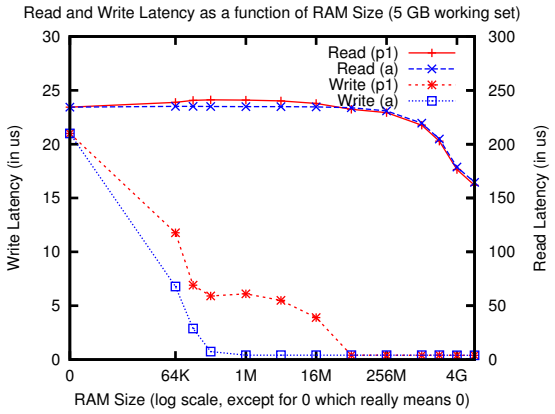

Figure 7: Application read and write latencies with small RAM cache sizes and a small workload.

blocks and performance drops.

The somewhat startling conclusion is that with a large, cheap flash cache, and a workload much larger than RAM, we can allocate minimal RAM (large enough to act as a speed-matching buffer) to file system caching, leaving the rest of memory available for application or operating system use!

This was tantalizing, so we tried the small RAM configuration on RAM-sized workloads. Figure 7 shows the latencies for a workload with a 5GB working set. As seen at the right, this configuration carries a 25-30% penalty, which is noticeable but far less than the factor of five or so seen without the flash cache. It may be an acceptable tradeoff in some circumstances.

7.6 Read-mostly vs. Write-mostly

The previous results all assumed a 30% write percentage. We next investigate the sensitivity of our results to the write percentage. We use our baseline working set sizes (60 GB and 80 GB) and cache sizes (8 GB RAM cache and 64 GB flash cache), while varying the percentage of writes in the trace from 0% to 100%. Figure 8 shows the application-level read and write latencies. As expected, read latency remains stable. The write latency is also unaffected except at very high write rates, where we start seeing synchronous writebacks from the RAM cache that expose the flash's write latency. As the proportion of writes increases, the trace runs faster, because writes are faster than reads. At very high write rates the 1-second RAM-to-flash syncer starts to fall behind. Several other effects come into play as well, such as network saturation, resulting in complex behavior that may be imperfectly modeled. The portion of the graphs above 90%

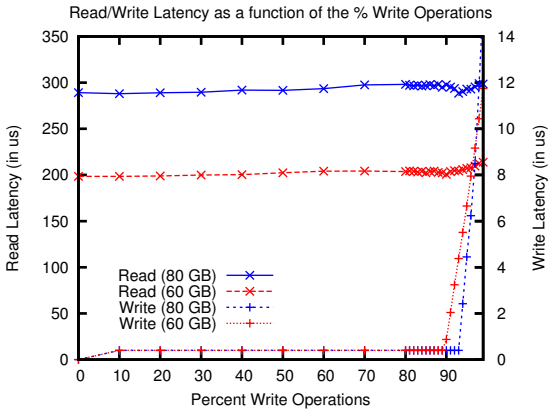

Figure 8: Application read and write latencies (in seconds) as a function of write percentage. As long as the write percentage remains below 90%, avoiding synchronous RAM evictions, performance is independent of the write rate.

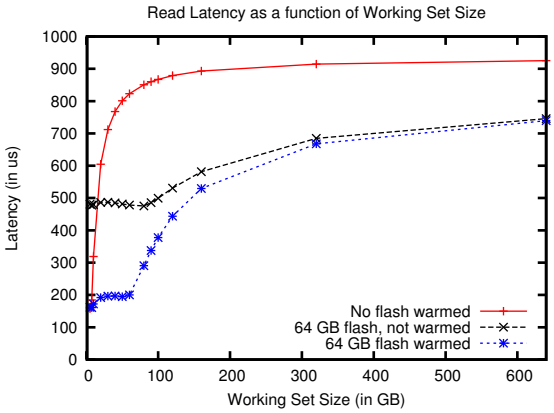

Figure 10: Effect of persistence. The not-warmed case is equivalent to having a non-persistent cache and crashing at the beginning of the simulator run. The no flash case is provided for comparison.

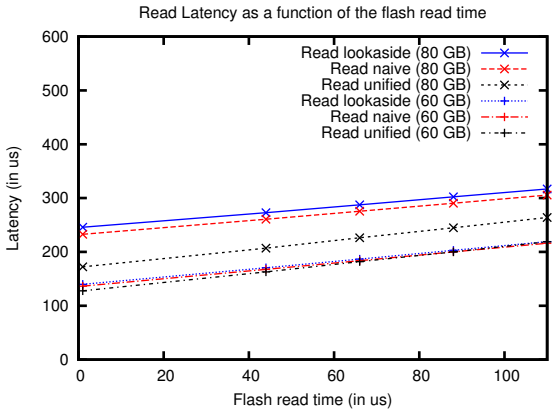

Figure 9: Application read latencies (in μs) for a range of flash read latencies (shown) and write latencies (proportional), in μs.

writes should be taken with a grain of salt.

The benefit of flash caching increases with write ratio because writes never incur a file server latency by missing in the cache: they always go straight to cache and are written back in the background.

7.7 Flash Timings

As flash devices vary a good deal in performance, we wanted to test a variety of flash timing configurations. Once again, the results were as expected: where the flash latencies appear directly, they scale with the flash speed; where they are hidden, changing the flash speed has no effect; and where they participate in the total latency, the overall latency scales linearly.

Figure 9 shows the application-level read latency for a range of flash timings for both standard traces and all three cache architectures. The leftmost point represents the potential performance of phase-change memory.

When the working set fits in flash, the architecture makes little difference, but when it falls out, we see the benefit of the larger effective sizes of the *unified* architecture. In all cases, however, application latency scales linearly with the flash latency, so improvements in flash timings are readily visible to the application.

7.8 Persistence

We approximated the cost making the flash persistent by doubling the flash write latency to model performing two flash writes per block, one of the data and one for the meta-data describing the block. (We did not attempt to simulate the recovery phase.) We investigated the benefit by skipping the warming phase of our traces; this is equivalent to having a non-persistent flash cache and crashing at the start of the simulator run.

The result is that the increased flash write latency associated with persistence is invisible to the application. This is consistent with our other results where the flash write latency is also invisible. However, the benefit of persistence, or rather the potential cost of not providing persistence, is substantial, as shown in Figure 10.

7.9 Cache Consistency

As discussed in Section 3.8, flash caches introduce two problems related to consistency: their larger size, and,

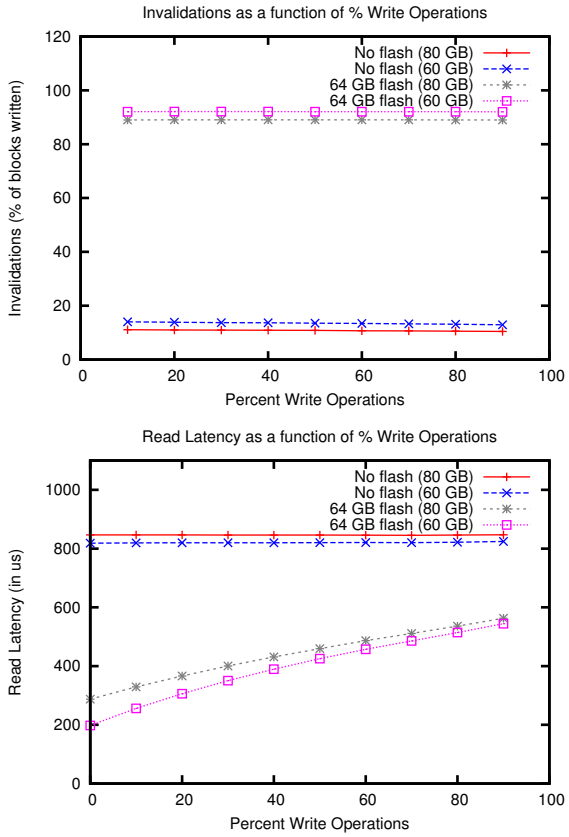

Figure 11: Invalidations required, and read latency, as a function of write percentage.

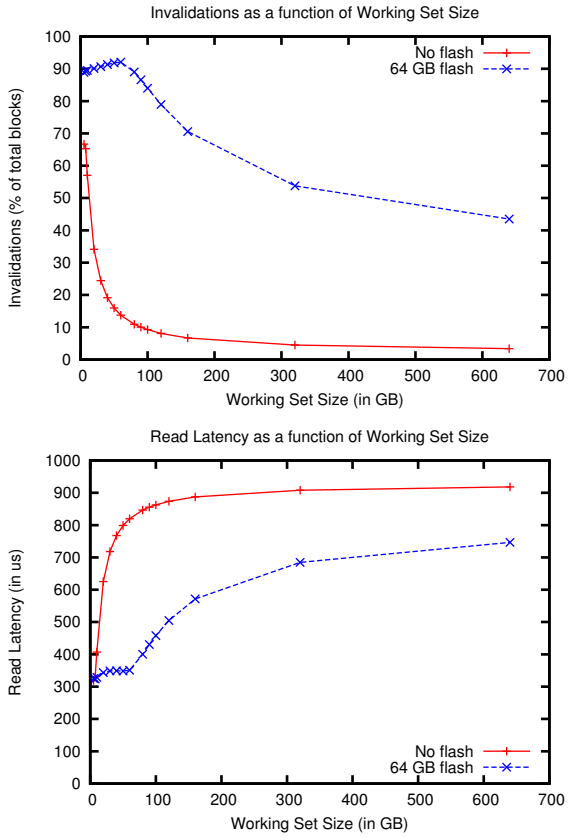

Figure 12: Invalidations required, and read latency, as a function of working set size.

for recoverable caches, that the cache is offline during reboots. These may affect cache consistency protocols.

We generated two additional families of traces, using two hosts, to investigate the effect of size on consistency control. As a worst-case scenario we make the two hosts share one working set. In the first family, we examine varying write percentages; in the second, we examine a range of working set sizes. Writing a new version of a block into a cache must invalidate all copies in other caches. We measure the fraction of (application-level) block writes that require invalidations.

Figure 11 shows the percentage of blocks written requiring invalidation and application read latency, as a function of the write percentage. The write latencies (for the 64 GB flash) are comparable to those in Figure 8.

Figure 12 shows, for the baseline setting of 30% writes, the percentage of invalidations and the application read latency as a function of the working set size. The write latency results are uniform and are not shown.

The primary finding is that for workloads that fit in flash, the percentage of writes requiring invalidation is high, even relative to workloads that fit in RAM with no flash. The invalidation rate drops off for out-of-cache workloads, but neither as quickly nor as significantly as with the smaller RAM cache. This has implications for read performance as well. Comparing the application read latency graphs (Figure 11 to Figure 8 and Figure 12 to Figure 4), we see that while the flash provides an advantage, read latency increases with the fraction of invalidations, because invalidated blocks must be reread from the filer. Although this is a worst case analysis (both servers share the entire working set), these results highlight critical areas in cache management design.

8 Conclusions

The results of our simulations show that even in its simplest implementation, a client-side flash cache provides significant benefits to applications. We now review our findings regarding the design questions from Section 1.

The flash cache does not need to be integrated with the

file system. While doing so increases the effective size of the cache, given the relative sizes (and prices) of RAM and flash this effect is fairly small and may not justify the implementation complexity.

The flash cache can be as large relative to RAM as desired. In fact, except for workloads that fit entirely into RAM, it makes sense to limit the RAM cache to the space needed to buffer writes, keeping the cache only in flash.

Any writeback policy that avoids synchronous writes and does not allow the cache to become full of dirty data produces good performance. Prompt writeback from flash exposes cache consistency events at no cost, and these cache consistency events are potentially important.

It is not necessary to make the cache persistent (that is, recoverable) to benefit from it. However, doing so offers significant additional benefit.

Cache consistency is a serious issue when multiple hosts actively modify a shared working set. Even with a write-through flash cache, such workloads cause substantially higher invalidation traffic than we see with traditional RAM-based caches. Also, traditional cache consistency protocols may not be able to cope with a recoverable cache being offline while recovering.

There is much follow-on work to be done. The most important area of further research is adapting file servers to these larger caches, ensuring that we can retain excellent read-ahead behavior when we do miss in the flash. In the presence of data shared among multiple hosts, each with its own flash cache, it is necessary to explore the details of maintaining cache consistency among the multiple caches. Finally, flash caching is a good candidate for a custom flash translation layer [19] – exploring approaches and algorithms as well as establishing satisfactory lifetime for this application remains as future work.

9 Acknowledgements

This work was supported by NetApp. In addition, 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] Smart Array technology: Advantages of battery-backed cache. http://h10032.www1.hp.com/ctg/Manual/c00257513.pdf, 2002.

[2] Oracle, Sun launch high-end OLTP server. PCWorld, Sep 2009.

[3] EMC outlines strategy to accelerate flash adoption. In *EMCWorld 2011* (May 2011), http://www.emc.com/about/news/press/2011/20110509-05.htm.

[4] AGRAWAL, N., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Generating realistic impressions for file-system benchmarking. *Trans. Storage 5* (December 2009), 16:1–16:30.

[5] BUTT, A. R., GNIADY, C., AND HU, Y. C. The performance impact of kernel prefetching on buffer cache replacement algorithms. In *Proc. SIGMETRICS 2005* (Banff, Alberta, Canada, 2005), ACM, pp. 157–168.

[6] BYAN, S., ET AL. Mercury: Host-side flash caching for the data center. In *28th IEEE Symposium on Mass Storage Systems and Technologies (MSST 2012)* (April 2012), pp. 1 –12.

[7] CHEN, P. M., NG, W. T., CHANDRA, S., AYCOCK, C., RAJAMANI, G., AND LOWELL, D. The Rio file cache: Surviving operating system crashes. In *Proc. ASPLOS* (October 1996).

[8] FORNEY, B. C., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Storage-aware caching: revisiting caching for heterogeneous storage systems. In *Proc. FAST* (Monterey, CA, 2002), USENIX Association, pp. 5–5.

[9] HOWARD, J. H., ET AL. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst. 6* (February 1988), 51–81.

[10] JOUKOV, N., WONG, T., AND ZADOK, E. Accurate and efficient replaying of file system traces. In *Proc. FAST* (San Francisco, CA, 2005), USENIX Association, pp. 25–25.

[11] KOLLER, R., ET AL. Write policies for host-side flash caches. In *Proc. FAST* (San Jose, CA, 2013), USENIX Assoc., pp. 45–58.

[12] KOURAI, K. CacheMind: Fast performance recovery using a virtual machine monitor. In *Dependable Systems and Networks Workshops (DSN-W)* (July 2010), pp. 86 –92.

[13] MCCALPIN, J. D. Stream: Sustainable memory bandwidth in high performance computers. Tech. rep., University of Virginia, Charlottesville, Virginia, 1991-2011. A continually updated technical report. http://www.cs.virginia.edu/stream/.

[14] MESNIER, M. P., ET AL. Trace: parallel trace replay with approximate causal events. In *Proc. FAST* (San Jose, CA, 2007), USENIX Association, p. 24.

[15] MICROSOFT. ReadyBoost. http://windows.microsoft.com/en-US/windows7/products/features/readyboost, 2009.

[16] NELSON, M. N., WELCH, B. B., AND OUSTERHOUT, J. K. Caching in the Sprite network file system. *ACM Trans. Comput. Syst. 6* (February 1988), 134–154.

[17] NETAPP. Flash Cache. http://www.netapp.com/us/products/storage-systems/flash-cache/.

[18] RAIDON. HyBrid RunneR iH2420-2S-S2 data sheet. http://www.raidon.com.tw/content.php?sno=0000462&p_id=113, 2010.

[19] SAXENA, M., SWIFT, M. M., AND ZHANG, Y. FlashTier: a lightweight, consistent and durable storage cache. In *Proc. EuroSys* (Bern, Switzerland, 2012), ACM, pp. 267–280.

[20] SEAGATE. Momentus XT product data sheet. http://www.seagate.com/docs/pdf/datasheet/disc/ds_momentus_xt_retail.pdf, 2009.

[21] SHEPLER, S., ET AL. NFS version 4 protocol. http://www.ietf.org/rfc/rfc3530.txt, April 2003.

[22] VIJAYAKUMAR, K., MUELLER, F., MA, X., AND ROTH, P. C. Scalable I/O tracing and analysis. In *Proc. Workshop on Petascale Data Storage* (Portland, Oregon, 2009), ACM, pp. 26–31.

[23] YADGAR, G., FACTOR, M., AND SCHUSTER, A. Karma: know-it-all replacement for a multilevel cache. In *Proc. FAST* (San Jose, CA, 2007), USENIX Association, pp. 25–25.