Flash-Conscious Cache Population for Enterprise Database Workloads

Hyojun Kim
IBM Research - Almaden
hyojun@us.ibm.com

Ioannis Koltsidas
IBM Research - Zurich
iko@zurich.ibm.com

Nikolas Ioannou
IBM Research - Zurich
nio@zurich.ibm.com

Sangeetha Seshadri
IBM Research - Almaden
seshadrs@us.ibm.com

Paul Muench
IBM Research - Almaden
pmuench@us.ibm.com

Clement L Dickey
IBM Research - Almaden
dickeycl@us.ibm.com

Lawrence Chiu
IBM Research - Almaden
lchiu@us.ibm.com

ABSTRACT

Host-side flash caching has lately emerged as a suitable and effective means of accelerating enterprise workloads. However, cache management for flash-based caching is different from traditional DRAM-based caching. A flash cache sits underneath the DRAM cache. Its position in the hierarchy combined with the unique characteristics of flash, calls for a different cache management solution. Specifically, cache population, an aspect of cache management which is not attributed much importance in DRAM caches, becomes crucial in flash-based caches.

In this paper, we first present a performance evaluation of three popular open-source flash cache implementations: *flashcache*, *bcache*, and *EnhanceIO*. We evaluate them under an industry-standard database benchmark and identify their limitations. We demonstrate that several shortcomings are due to sub-optimal cache population. We propose a novel set of techniques for cache population, and present the design of the Scalable Cache Engine (SCE) – a new flash cache solution that incorporates our cache population techniques. We demonstrate that SCE remarkably outperforms the existing open-source solutions: 45% higher throughput, 55% lower latency, 12× faster cache warm-up than *flashcache*, and 95% less memory usage than *EnhanceIO*.

1. INTRODUCTION

Over the last decade solid-state storage technology has dramatically changed the architecture of enterprise storage systems. Advancements in flash-based solid state drive (SSD) technology have resulted in SSDs that outperform traditional hard disk drives (HDDs) along a number of dimensions: SSDs have higher storage density, lower power consumption, a smaller thermal footprint, and orders of magnitude lower latency and higher throughput. Thus, flash-based storage devices have been deployed at various levels of the enterprise storage hierarchy ranging from a storage tier in a multi-tiered environment (e.g., IBM Easy Tier [13], EMC FAST [6]) to a caching layer within the storage server (e.g., IBM XIV SSD cache [15]), and to a host-side cache (e.g., IBM Easy Tier Server [14], EMC XtreamSW Cache [8], NetApp Flash Accel [27], FusionIO ioTurbine [9]). Recently, several all-flash storage systems that completely eliminate HDDs (e.g., IBM FlashSystem 840 [12], Pure Storage [30]) have been introduced and gained significant traction.

Due to its performance, capacity, and cost characteristics, flash memory fills the gap between DRAM and magnetic HDDs nicely. Flash is roughly 20 times faster than HDDs, and about 100 times slower than DRAM. Also, flash is around 10 times more expensive than HDDs, but some 10 times less expensive than DRAM. This makes flash a good choice as a caching layer between DRAM and HDDs. In a typical environment, host servers utilize directly-attached (DAS) SSDs to cache data resident in a storage network (SAN) backend. By placing data close to the applications and eliminating network latencies, application performance is improved. For instance, in server virtualization or online transaction processing (OLTP) environments with SAN-attached storage back-ends, host-side flash caching can reduce latency, eliminate congestion at the SAN backend, and thereby improve overall system throughput.

The use of flash-based SSDs as a caching layer is particularly interesting in enterprise environments since it can provide targeted performance acceleration. For example, host-side flash caches can be utilized selectively at hosts running performance-critical applications. Moreover, as a storage layer cache, the presence of the cache is completely transparent to the application. By contrast, when using flash SSDs as persistent storage in the SAN or the host, appropriate volumes need to be allocated by the administrator and some tuning is required at the application side to ensure that the appropriate data end up on the SSDs and even to ensure that the SSD does not become a performance bottleneck. Ineffective tiering in such cases may result in lower-than-expected performance [16]. In order to continue

to benefit from the high-availability, resiliency and management functionality (such as snapshots and remote mirroring functions) provided by the storage backend, host-side flash caches typically operate in a *write-through* mode. That is, the host only caches unmodified data and data that have been already committed to the SAN so that a host failure will not impact data availability.

Host-side flash caches inherently differ from traditional DRAM caches in two ways: flash caches 1) sit underneath DRAM caches in the storage hierarchy, and 2) use flash SSD instead of DRAM as the caching media. The first difference influences the characteristics of workloads encountered by the flash cache. When there is a substantial amount of DRAM cache above the flash cache, the hottest potion of workload is absorbed by the DRAM cache, and the flash cache receives storage accesses for a sparser and wider range of data with lower access frequency – in other words, data locality at the flash cache is weakened. In such a multi-level cache configuration, the second level cache must have a relatively large capacity; otherwise, a performance improvement is unlikely.

The use of flash SSD instead of DRAM as the caching media directly impacts the cache management policies. More specifically, the asymmetric read-write characteristics and endurance issues of flash render cache population an expensive affair. With DRAM caches, population implies only a memory copy operation, and therefore, it is a good idea to populate on each cache miss. But, using these same cache population policies on flash has several shortcomings, which we describe next.

First, read operations that result in a cache-miss may be slowed-down by cache population, due to high flash write latencies. In traditional cache implementations, cache population occurs upon a cache miss: the missing blocks of data are read from the source and copied into the cache before being returned to the user. Typically, caches aim to cache Most Recently Used (MRU) blocks as they have higher likelihood of access in the near future, evicting the Least Recently Used (LRU) blocks to make space if there is none. Such mechanisms are perfectly suitable for DRAM-based caches, where the cache size is small and population is done by just a main memory write operation. However, blindly populating all the data accessed by the user into the flash cache is not always prudent for flash-based caches. It has already been pointed out that *populating upon every cache miss* may actually lower the end-to-end performance of a flash cache rather than improve it [22]. Unfortunately, the window of occurrence of this issue is not negligible. The typical expectation is that once the cache is warmed up, the cache miss rate becomes low, and that cache population occurs rarely. However, for flash caches, cache warm-up takes significant amount of time due to the relatively large capacity (typically hundreds of gigabytes).

Slow cache warm-up is the second issue of cache management strategies that populate data upon a cache miss. Flash caches have orders-of-magnitude higher capacity than DRAM-based ones, in some instances up to several TiBs of data. At this scale, it may take many hours to fill up the cache: a 300 GiB-sized flash cache took over 10 hours to reach its maximum hit rate for an enterprise workload [4].

The next issue is related to the handling of write I/O. As mentioned earlier, a write-through policy is a standard choice for enterprise storage. The backend SAN storage servers are designed to be highly reliable and available even in the event of hardware failures. On the other hand, with use of commodity parts and lack of redundancy, host-side SSDs are typically not as reliable. Therefore, most enterprise class flash cache solutions use write-through policy as a default option. However, write-through policy can result in even worse performance for a write intensive workload because every write incurs a cache population operation, which is expensive on flash [22]. Unlike the first issue – increased I/O latency from in-line cache populations – this issue remains even after flash cache is fully warmed up.

Another issue arises when the cache capacity is smaller than the working-set size – a possibility that cannot be ruled out. In such a case, unconditional cache population will result in thrashing, adversely impacting the cache hit rate and resulting in poor performance. In conclusion, a flash-conscious, selective cache population policy is necessary to eliminate unnecessary cache population.

We have conducted a performance evaluation study with three open-source solutions: *flashcache* [26], *bcache* [33], and *EnhanceIO* [34]. The evaluation is based on the TPC-E benchmark [5, 35], an industry-standard benchmark that represents a typical workload to which flash caching is applied. We created a typical database server configuration, and evaluated the three cache solutions with TPC-E. The results show that these existing solutions have significant shortcomings that prevent them from realizing the full potential of flash caching. Shortcomings include a long ramp up time due to slow population as well as a large main memory footprint. In some cases, the performance improvement over the baseline was marginal; this was due to inefficient cache management.

In this paper we present the Scalable Cache Engine (SCE), a novel host-side flash-based cache management framework that addresses the issues listed above and aims to alleviate the performance and scalability shortcomings of existing approaches. We focus on cache population, and propose three major changes to cache population policies. First, we propose to *not* populate on each cache-miss – in other words, we propose *selective cache population*. We monitor I/O traffic and choose a population candidate based on access recency and frequency. Second, we propose to separate the cache population path from the foreground I/O activities so as to not influence I/O latency. Instead, cache population occurs in the background and consumes a fixed, reserved I/O budget. Moreover, we propose *asynchronous write-through (AWT)* instead of synchronous write-through to avoid adding delay to write requests. Finally, SCE minimizes its memory footprint, an important consideration for a scalable solution.

The key contributions of our work can be summarized as follows:

- We present an experimental study with existing open-source flash cache implementations under real-world enterprise workloads. We analyze their performance and identify major performance bottlenecks.

- Based on our observations, we introduce SCE, a novel caching engine optimized for flash and geared towards enterprise storage workloads. We detail the key design and implementation decisions that allow SCE to achieve scalability and high performance.

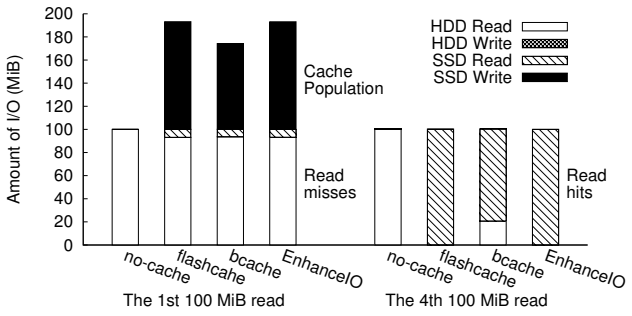

Figure 1: The amount of I/O for 100 MiB of random read tests: *bcache populated 20% less than the others, but there was no other significant difference in the amount of I/O.*

- We present results based on real-world workloads that demonstrate significant improvements over existing open-source solutions: 45% higher throughput, 55% shorter latency, 12× faster cache warm-up than *flashcache*, and 95% less memory usage than *EnhanceIO*.

The rest of the paper is organized as follows: In Section 2, we present our experimental study with three open-source flash cache implementations under the TPC-E workload and provide insights into their performance shortcomings. We then discuss our approach to flash caching in Section 3. In Section 4 we revisit the experimental evaluation with TPC-E, this time comparing SCE to the existing approaches. An overview of related work is given in Section 5. We present our conclusions and future work in Section 6.

2. CACHE EVALUATION WITH TPC-E

Flash caching in host servers is gaining significant attention in the market and in the research community. However, the performance impact of host-side flash caching has not yet been studied systematically. Currently available results are limited to micro-benchmarks with simple access patterns, rather than to realistic enterprise workloads.

We evaluated three popular open-source flash cache solutions with the TPC-E benchmark [5, 35] to understand the performance impact of flash caching. Since database acceleration is one of the most popular enterprise use cases for flash caching, we used TPC-E as the workload of choice, as it has been designed to be representative of modern Online Transaction Processing (OLTP) workloads. In particular, TPC-E emulates the OLTP workload of a brokerage firm where transactions are processed by a central database. The transactions are related to the firm's customer accounts and include trades, account inquiries, and market research, as well as interactions of the firm with financial markets to execute orders on behalf of the customers and to update relevant account information. The benchmark results in a highly random I/O workload with an 87% read and 13% write mix.

Our experimental setup is a fairly typical one, where an enterprise database system is running within a virtual Linux [1] server. The physical host is running Linux and uses a directly-attached SSD for caching, while network-attached storage volumes are used as the storage backend. We configured a 200 GiB SSD partition as the caching device while the whole VM image size (including the Linux OS and the TPC-E database) was 160 GiB; thus, there is enough cache space to hold the entire data set and no eviction needs to be exercised by the cache drivers. More technical details about our setup are given in Section 4. We repeated the same experiment for all three cache drivers, namely *flashcache*, *bcache*, and *EnhanceIO*, using the same database configuration and the same hardware. All drivers were configured in write-through mode, which is the typical choice for enterprise storage. We also ran the benchmark for the baseline system, i.e., without flash caching, and for a system that uses the SSD as persistent storage space with the database on SSD; we refer to the former as *no-cache* and to the latter as *all-flash*. For each run we measured the number of TPC-E transactions per second (TpsE), as well as I/O performance metrics and cache performance statistics.

[1]Linux is a registered trademark of Linus Torvalds in the United State or other countries, or both.

Table 1: TPC-E average TpsE and memory usage with three open-source flash cache solutions: *flashcache shows the highest average TpsE and largest memory usage.*

	(Last 1 hour average) TpsE	Read hit rate	Memory Usage
no-cache	11.2	N/A	N/A
all-flash	87.9	N/A	N/A
flashcache	56.9	98.66%	1,212 MiB
bcache	17.3	96.75%	15 MiB
EnhanceIO	17.5	96.52%	415 MiB

In Figure 2 we present the end-to-end TPC-E results for an 8-hour run of the benchmark. Table 1 shows the average TpsE and read hit rate during the last hour of the run, as well as the memory footprint of the caching drivers. Interestingly, *flashcache* shows a much higher performance improvement (5× against *no-cache*) than the others (1.5× by *bcache* and *EnhanceIO*). We also compared the memory footprints of the three solutions for the configuration with a 200 GiB cache device and an 160 GiB source device; *bcache* used only 15 MiB of main memory since it uses a B-tree structure (stored in a flash caching device) for its index management – only few B-tree nodes are loaded in main memory. On the other hand, *flashcache* and *EnhanceIO* maintained their metadata in main memory, resulting in memory footprints of 1,212 MiB and 415 MiB respectively, or 24 bytes and 8 bytes per 4 KiB cached block, respectively.

The average hit rate is also shown in Table 1: *flashcache* achieved a 2% higher read hit rate than other solutions, but showed a much higher TpsE throughput. A small hit rate difference can make a significant difference in end-to-end performance in some cases, but this is not the case here. Indeed, *flashcache* exhibited a 96.74% read hit rate about 4.6 hours into the run, which is comparable to the final hit rate of the other two solutions. Yet, even with comparable hit rates, *flashcache* performance was 45 TpsE, which is already much higher than what the other two drivers achieved eventually.

To better understand the behavior of the caching solutions under consideration, we ran a simple random read test. After initializing the flash cache, we performed 100 MiB worth

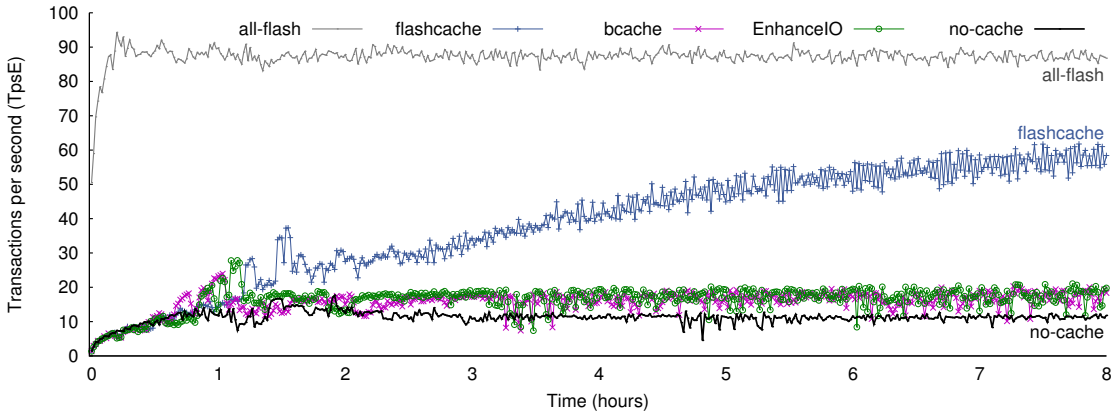

Figure 2: TPC-E results with three open-source flash cache solutions: *after 8 hours of runtime flashcache achieves about 5× performance improvement, whereas bcache and EnhanceIO reach about 1.5× performance improvement.*

of random 4 KiB reads to the cache-enabled volume. We repeated the same workload four times to make sure that the data was populated into the flash cache, capturing the amount of I/O traffic to both the backend device and the cache device. Figure 1 shows the amount of I/O collected for the first and fourth iteration: since the workload remained the same, one would expect that the cache hit rate would have reached 100% at the fourth iteration. However, *bcache* only populated 80% of the data. We observed no other significant difference. We carried out the same experiment with writes instead of reads but we did not observe any significant difference among the solutions. This result supports the claim that cache hit rate is not the main reason for the performance shortcomings of *bcache* and *EnhanceIO*.

The next interesting observation is related to cache warm-up times. As shown in Figure 2, it takes *flashcache* a significant amount of time before it reaches a substantial performance improvement over the baseline, an indication that the cache warm-up procedure is slow. Such a behavior is expected due to the 4 KiB cache block size: a 100 GiB flash cache will only be filled up after 26,214,400 4 KiB cache block misses. Cache warm-up times are particularly critical in use cases such as live Virtual Machine migration and system fail-over, and slow warm-up introduces performance variance depending on host machine state (e.g., depending on the latest reboot cycle). More importantly, none of the caching drivers get even close to realizing the full potential of the SSD: even *flashcache* only reaches about 67% of the *all-flash* performance after 8 hours of runtime, despite the fact that the dataset could easily fit into the cache in its entirety.

3. SCALABLE CACHE ENGINE

In this section, we first present three key changes to cache population policies, and describe our implementation of SCE.

3.1 Selective coarse-grained population

Unconditional cache population on every cache-miss can cause performance issues a) for write intensive workloads and b) when the cache size is too small to hold the entire working set. SCE chooses to populate *selectively* based on I/O traffic, which implies that we need to allocate some memory per observation unit. Clearly, 4 KiB is too small a granularity to maintain such information, and therefore we define a *fragment* as the cache management unit. A fragment consists of N logically contiguous blocks (of 4 KiB each); in our approach we use N = 256 for a 1 MiB fragment size. The user workload is continuously monitored and hot fragments are identified based on the foreground I/O traffic. SCE picks the hottest fragments for population from among the 100 most recently accessed fragments. This enables faster population since population occurs in larger units (1 MiB fragments) as opposed to 4 KiB pages.

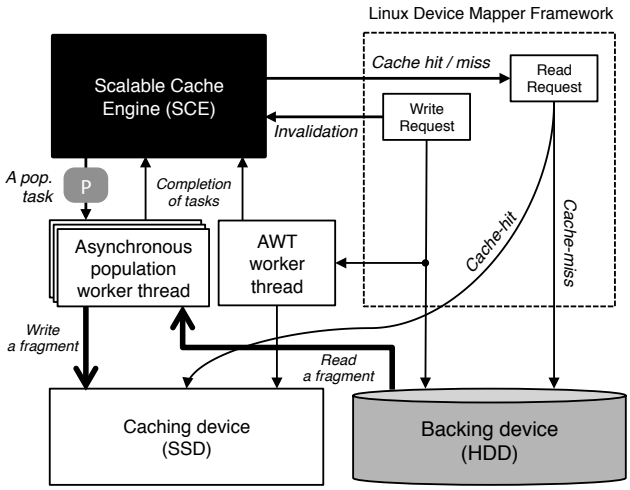

Figure 3: Asynchronous cache population in SCE: *SCE provides cache functions as a service; read requests are directed to the caching device or the backing device based on cache mapping information; write requests invalidate page validity bitmaps and are passed to backing device and the AWT worker thread; multiple threads perform cache populations asynchronously in the background.*

3.2 Asynchronous background population

In-line cache population, that is, population that occurs within the user data path, can slow down cache-missed read operations. It can degrade overall I/O performance, especially when too many cache misses happen – for instance

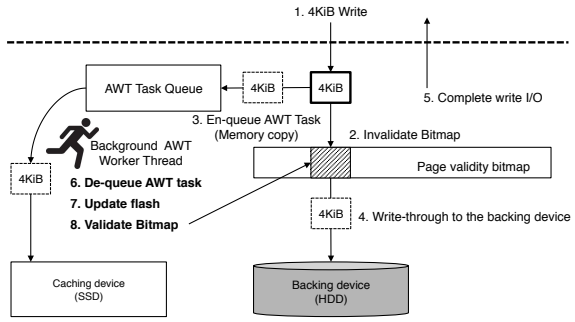

Figure 4: Asynchronous write-through (AWT): *to not influence write latencies, AWT updates cache device asynchronously, and page validity bitmap is used to prevent servicing read requests until the cache update operation is completed.*

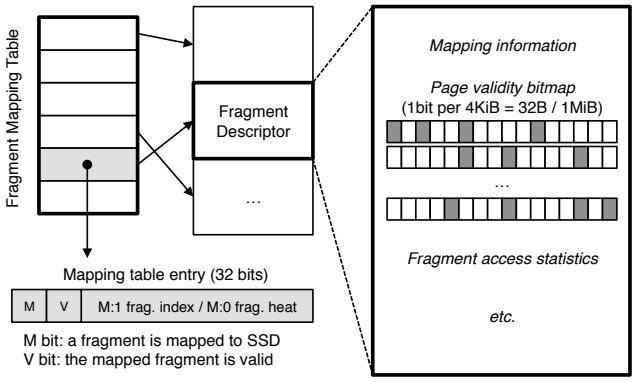

Figure 5: Coarse-grained cache management: *a direct mapping table is used to maintain fragment level mappings; each fragment descriptor has a page validity bitmap to track which parts of the fragment are valid.*

when the flash cache is empty or the workload changes. To avoid this, SCE employs *asynchronous background cache population* at a fragment granularity. As shown in Figure 3, we separate the cache population path from the foreground I/O data path. This approach borrows ideas from automated storage tiering mechanisms [6, 13]. The user workload is continuously being monitored as already explained in Section 3.1 and hot fragments are identified based on the foreground I/O traffic. Once identified, they are populated into the cache by multiple asynchronous population worker threads in the background.

This separation of concerns completely decouples the two data paths, effectively removing the flash write latency from the user requests. As a result, this approach gives the cache more control and flexibility about how much and when to populate. For instance, the cache can limit its population rate to avoid performance degradation for write intensive workloads or when it finds that the working set has already been populated, freeing up bandwidth from the device to serve read hits.

Since population is done at a fragment granularity, it is possible that a cache miss for 4 KiB block may result in a 1 MiB fragment population. In practice, however, we have found that this acts as an effective pre-fetching mechanism that not only accelerates cache warm-up but also improves the cache hit rate. Moreover, fragment-based population is desirable for flash based SSDs because it results in large writes to the SSD thereby improving the endurance of the flash device [25]. By changing the fragment size, the write pattern can even be customized to be optimal for a specific SSD based on the internal SSD geometry such as the virtual block size [31].

3.3 Asynchronous write-through

Write-through is another aspect of the cache implementation that needs to be adapted to the specific characteristics of flash because flash writes are relatively expensive in terms of latency and device wear. In a traditional implementation every user write will be submitted to both the source device and the caching device, and the user request will be completed only after the two writes (to the cache and source devices) have been completed. However, as researchers have pointed out, this can cause significant performance issues under write-mostly or write-intensive workloads [22, 23] due to the dependency of the user write latency on the write latency of the flash device. To eliminate this effect, SCE employs *asynchronous write-through (AWT)*, which removes this dependency by parallelizing the two writes and completing the flash write asynchronously.

In Figure 4 we illustrate how a user write operation is handled using asynchronous write-through. Once the write reaches the cache, SCE invalidates the appropriate pages using a *page validity bitmap* (Figure 4) to prevent servicing read requests to those pages while write-through is still ongoing. Then a copy of the user buffer is made and the request is forwarded to the source device. In parallel, the cloned buffer is passed to the AWT thread, which in turn submits a write request to the SSD using that buffer. Once the write to the source device returns, the user request is completed. When the SSD write returns upon completion, the appropriate pages are re-validated and the write-through completes. In order to address the issue of duplicate or concurrent write operations to the same fragment, SCE allocates an additional bitmap to keep track of duplicate writes. The pages are validated at once when the last write request completes. Additional care needs to be taken to handle cases where the write to the source or to the SSD fails. If the source write fails, then SCE invalidates the pages again after the SSD write completes. If the SSD write fails, then the previously invalidated pages are not re-validated.

3.4 Coarse-grained cache management

Flash caches tend to be much larger than DRAM-based ones. In the market, vendors are already offering flash cache solutions with more than 2 TB capacity [7]. With the density of flash continuously increasing, flash caches are bound to soon reach 10's of TBs in capacity. Existing flash cache solutions use a cache block size of 4 KiB, similar to file systems and operating system (OS) page caches. At this granularity, the scalability of the cache is limited by the size of the mapping metadata (since each cache block requires a corresponding metadata entry). For instance, *flashcache* uses roughly 24 bytes of main memory per 4 KiB block which implies that nearly 12 GiB of memory would be required to track metadata for a 2 TiB flash cache. On the other hand, at 8 bytes per block *EnhanceIO* would require only 4 GiB of main memory to track metadata for a 2 TiB cache.

As mentioned earlier, SCE uses a 1 MiB sized fragment as the cache allocation and management unit. Cache allocation, population, and eviction occur at a fragment granularity, while read hits, invalidates and write-through occur at a 4 KiB block granularity. Obviously, coarse-grained cache management is more memory efficient than a fine-grained approach. In our approach we use roughly 76 bytes per cached fragment (1 MiB) for cache metadata, achieving a 152 MiB main memory footprint for 2 TiB of flash cache.

Coarse-grained cache management requires several issues to be addressed. The first issue is cache space efficiency, since a whole fragment is cached although only a few blocks in the fragment may be hot. In contrast, with a fine-grained mapping, the cache can allocate only hot blocks, thereby utilizing the flash space more efficiently. However, spatial locality in the access pattern mitigates this effect for coarse-grained caches. In addition, this issue becomes less critical as flash capacities grow.

A more complicated issue arises with respect to handling mismatches between the I/O request size and the cache mapping size, especially in the case that the I/O request size is not a multiple of the fragment size. For instance, to populate upon a write miss, the cache would need to read the rest of the data (i.e., the blocks not covered by the write request but belonging to the same fragment) from the source device. Alternatively, the cache may allocate the fragment but only fill it partially, effectively ending up wasting some space on flash. Both *flashcache* and *EnhanceIO* choose not to populate when an I/O request size does not match the cache mapping block size. SCE does not face this problem, as it populates the cache asynchronously in the background.

Figure 5 shows how SCE manages cache metadata using a coarse-grained mapping. Because SCE maintains mappings at a fragment granularity, it is feasible to use a direct mapping instead of a hash table. The *fragment mapping table* contains mapping table entries for the entire address space of the source device (e.g., a logical volume in the SAN): each table entry maps a logical fragment in the source device to a *fragment descriptor* if the fragment is cached. The fragment descriptor describes the state of each cached fragment, including a *page validity bitmap* to keep track of which 4 KiB pages in the fragment are valid. The bitmap maintains a bit per 4 KiB page, resulting in a bitmap size of 32 bytes per 1 MiB fragment. Note that by using a direct mapping table a fragment-level lookup can be done with just one memory reference. For a fragment found in the cache, efficient bitmap operations on the page validity bitmap can be used to determine a hit or a miss, even when the request spans multiple pages.

With this design the size of the metadata depends on the size of the source volumes in addition to the cache size. However, for large cache sizes the footprint of the fragment descriptors dominates that of the mapping table. For instance, with 100 TiBs of backend storage and 16 TiB of cache, the size of the mapping table is 400 MiB, while the size of the fragment descriptor array is 1,152 MiB. Even for 1 PiB of storage and 256 TiB of flash cache, SCE only needs about 22 GiB of memory, which is still feasible.

3.5 Cache eviction

SCE employs a variation of the generalized CLOCK algorithm [32] as the cache eviction policy. SCE uses a reference counter instead of a reference bit per fragment. When a fragment is accessed, its reference counter is incremented, up to a ceiling value (set to 4). As the clock hand moves past the fragment, the counter is decremented; a fragment having a reference counter of 0 is eligible for eviction. This algorithm, which is similar to the buffer cache replacement algorithm of PostgreSQL [10], takes into account both the access frequency of a fragment and its access recency to decide fragment replacement.

3.6 Implementation

We now turn to the implementation of SCE. We have implemented SCE on Linux using the Device Mapper (DM) framework. SCE creates a DM device on top of each source volume, which intercepts user read requests and forwards them to the cache. If a request results in a cache hit, it is served from the cache; otherwise, a miss is returned and the request is forwarded to the backing source device. Write requests are intercepted in a similar manner and write-through is performed as described in Section 3.3.

For the asynchronous cache population, SCE uses a group of parallel threads, each of which is running its own instance of the cache population policy. As the hit rate grows and the free space in the cache shrinks, the number of threads is reduced to slow down the population rate. The intuition behind this is that when the hit rate becomes high, this means that most of the working set has been already cached. Thus the population can proceed at a slower rate, effectively freeing up SSD bandwidth to be used for serving cache hits. In particular, cache miss notifications are distributed to population threads and each thread decides whether the corresponding fragment should be populated or not. Subsequently, the population thread suspends its operation for some time by sleeping, effectively limiting its population rate. The operation of the population threads is, thus, regulated by three parameters: a) the target miss rate (*target_missrate*), b) the minimum percentage of free fragments in the cache (*min_free_pcnt*) and c) the sleep time for a population cycle (*period*). Algorithm 1 shows how these parameters are used by each one of the threads.

In our current implementation, SCE uses eight threads, all of which run until the cache read miss rate becomes less than 35% or the percentage of free space in the cache reaches 10% or less. After that point only four threads perform cache population until read cache miss-rate becomes less than 15%. When the cache miss-rate goes lower than 15%, only 1 thread remains running and sleeps for 500 milliseconds at each population cycle. Each population thread uses its own fragment-sized memory buffer for its on-going population; thus by default SCE uses 8 MiB (1 MiB $\times$ 8 threads) of memory for asynchronous cache population. Similarly, SCE pre-allocates memory space for asynchronous write-through to hold copies of in-flight user data. SCE uses another 8 MiB for that purpose; if that is not sufficient, e.g., due to a very high write rate, then SCE adopts a write-around policy.

4. EVALUATION

4.1 Evaluation method

We now turn to the experimental evaluation of SCE. Figure 6 illustrates the performance measurement setup that was used. The VM server is a 24-core x86 server with 64 GiB of RAM, running Linux kernel 3.11. An enterprise class

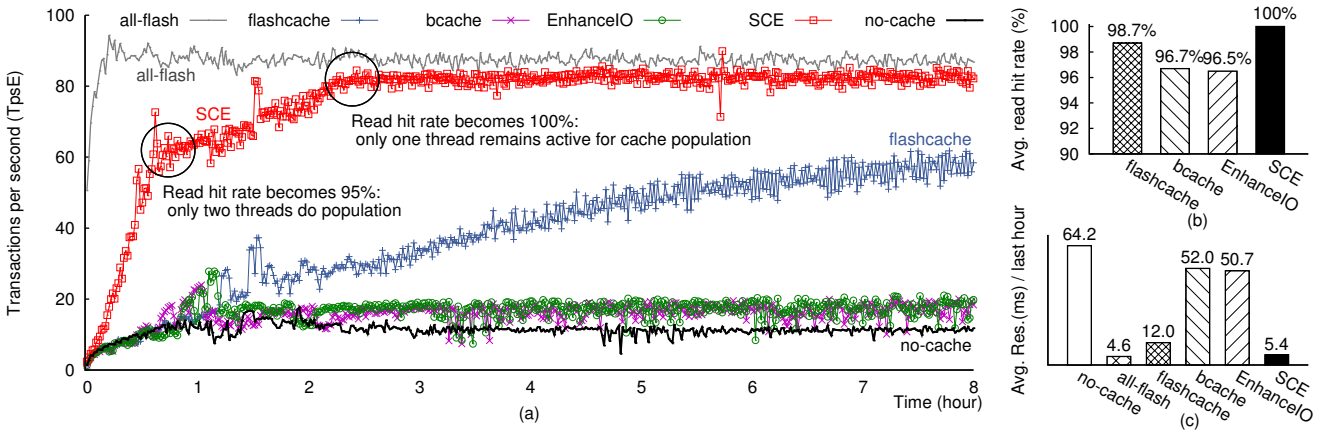

Figure 7: Eight hour TPC-E with a 200 GiB flash cache (a) TpsE: *SCE demonstrates superior performance to three open-source flash cache solutions – it is much faster and achieves much higher TpsE*, **(b) Last hour average read hit rate:** *SCE achieved 100% cache read hit rate while others achieved 96.5-98.7%*, **(c) Last hour average response time:** *SCE shows only 17% increased average response time of all-flash configuration while the next best(flashcache) yielded about 2.6× longer response time and bcache and EnhanceIO yielded much worse about 11× longer response times than all-flash.*

Algorithm 1 Asynchronous cache population thread

```
loop
    Sleep for period
    Read current cache status
    if cur_free_pcnt ≥ min_free_pcnt then
        if cur_missrate ≥ target_missrate then
            Promote the hottest fragment among the
            100 MRU fragments
        end if
    end if
end loop
```

eMLC PCIe SSD card with a capacity of 1.8 TiB is attached to the VM server. The iSCSI target comprises 3× 73 GB, 15 kRPM, 6 Gbps SAS hard disks in a RAID0 setup. The VM server and the iSCSI target are connected over Gigabit Ethernet. The TPC-E benchmark is installed and runs within a Kernel-based Virtual Machine (KVM) running Red Hat Enterprise Linux (RHEL) 6.4 and IBM DB2 Express-C v10.5. The TPC-E KVM instance has 8 CPUs and 15 GiB of RAM allocated to it – this is based on the suggested specification for an extra-large DB instance within Amazon cloud [1]. We used the default configurations for the flash cache drivers: write-through mode, 4 KiB block size (except for *bcache*, which uses 512 B by default), 512 associativity for *flashcache*, and 256 associativity for *EnhanceIO*. Note that *flashcache* will not cache from any I/O that has a size smaller than the configured block size (it marks any such I/O as "uncachable"). Setting the *flashcache* block size to 1MiB, for example, results in zero cached data for the evaluated workload. Thus, we were not able to emulate a coarse-grained cache management with *flashcache*. After each TPC-E run, the VM disk image file is rolled back to a clean state, the KVM host machine is rebooted, and PCIe SSD is raw formatted with a vendor provided utility program. Disk I/O statistics are sampled every minute.

4.2 Eight hour run with a 200 GiB flash cache

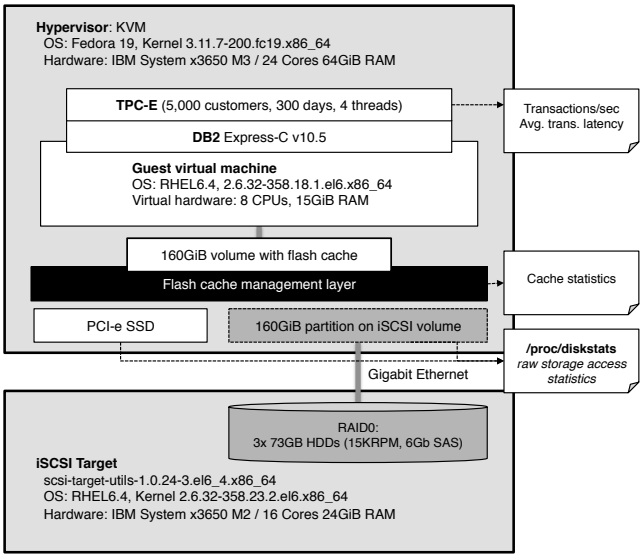

Figure 6: Measurement setting: *two servers, three 15 kRPM HDDs, one PCIe eMLC SSD are used to create a typical database server configuration; one server runs as an iSCSI target with 3× HDDs; the other server runs as a KVM host using the PCIe SSD as flash cache and a virtual machine runs TPC-E benchmark on DB2 database.*

We first evaluated the flash cache solutions running TPC-E in a setup where the flash cache capacity is larger than the workload. More specifically, we created a 200 GiB partition on the PCIe SSD with the VM disk image being 160 GiB. In such a scenario the flash cache should eventually migrate the entire workload into flash.

Figure 7(a) presents the bottom line TpsE performance, where one point was collected every minute during an eight-hour-long TPC-E benchmark run. SCE exhibits a significant performance boost compared to the other solutions, ranging from 1.5× compared to *flashcache*, to 4.8× compared to

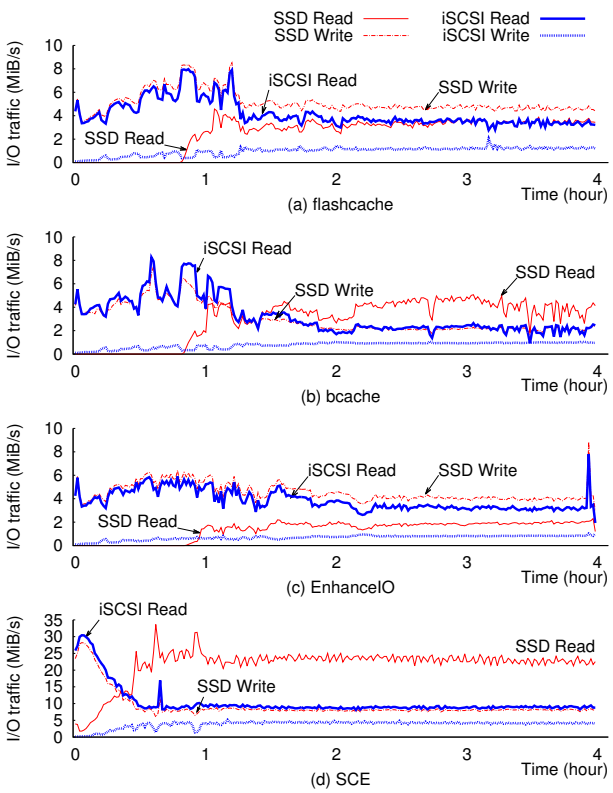

Figure 8: The amount I/O comparison for four hour run with 30 GiB flash cache: *the amount of SSD reads represents how many read requests are serviced by flash cache and the amount of SSD writes means the amount of cache population; with flashcache and EnhanceIO population rate is higher than cache service rate, which is not desirable as a cache, while with bcache and SCE, the cache service rates are higher than cache population rates – especially on SCE, cache service rate is about 3× higher than cache population rate, and it results in the highest TpsE result in Figure 9.*

bcache and *EnhanceIO*. More significantly, SCE results in a performance that is within 6% of the *all-flash* setup. Furthermore, SCE is faster to adapt to the workload: it only took 38 minutes for SCE to exceed the maximum performance of *flashcache*, and its total ramp up time was less than 2.5 hours compared to nearly 7 hours for *flashcache*. Figure 7(b) illustrates the read hit rate during the last hour after a 7 hour long cache warm-up. SCE achieves 100% of read hit rate while others show slightly lower rates between 96.5-98.7%. Figure 7(c) compares average response time during the last hour; SCE shows 2.2× shorter response time.

4.3 Four hour run with a 30 GiB flash cache

We next evaluated TPC-E performance of the flash caches in a more realistic scenario, where the cache is significantly smaller than the workload dataset: the flash cache comprised a 30 GiB partition of the PCIe SSD, with dataset size at the backend remaining at 160 GiB. Under such a scenario, and after the flash cache is filled for the first time, the cache replacement path become effective – unlike the first experiment where it was never exercised.

Figure 9 depicts the results. Notably, unlike the first experiment, *flashcache* did not perform significantly better than *bcache* and *EnhanceIO*; this can be explained by the reduced cache read hit rate of the *flashcache*, going from 99% in the 200 GiB flash cache setup to 50% in the limited flash cache size setup. To better understand this behavior, we analyzed the amount of I/O traffic during the experiment. Figure 8 compares the I/O traffic going to the SSD and iSCSI volume for each of the four flash cache solutions. Since the caches were operating in write-through mode all the SSD writes can be attributed to cache population traffic (as well as to persistent cache metadata management in the *bcache* case). Both *flashcache* and *EnhanceIO* seem to have been populating every cache-missed block: the amount of SSD write was higher than the amount of SSD reads. As of the tested version (3.1.1), *flashcache* performs LRU (or FIFO, based on the chosen policy) replacement without taking into account frequency of accesses – each read cache miss will result in a block population and thus a write to the SSD. We assume something similar for *EnhanceIO*. Based on the difference in read hit rate between *bcache* (83%) and that of *flashcache* (50%) and *EnhanceIO* (25%), we assume that *bcache* employs some kind of frequency filter on top of LRU to avoid constant cache replacement. Despite the higher hit rate, *bcache* did not offer a correspondingly higher TpsE performance. This is attributed to the persistent cache metadata management of *bcache*, which also translates to a relatively high average response time (Figure 9(c)).

SCE achieves significant improvements in cache hit rate, average latency, and bottom line performance. In terms of TpsE, SCE is 3.6×, 4.1×, and 4.9× higher than *flashcache*, *bcache*, and *EnhanceIO*, respectively (Figure 9(a)). We attribute the improvements to the asynchronous write-through, coarse-grained asynchronous population and to the replacement policy of SCE that takes recency and frequency into consideration. Meanwhile, the other approaches utilize synchronous write-through, fine-grained population and a replacement policy based solely on recency.

4.4 Write-through vs. write-back

In enterprise storage systems, direct-attached flash cache solutions typically augment the SAN in a non-disruptive manner: the backend storage's high reliability and availability are left intact. In such a scenario the flash cache operates in write-through mode. In a non-enterprise, share-nothing scenario, however, write-back mode could also be employed without affecting the reliability of the system. Write-back usually results in higher performance than write-through, since the write requests can be serviced locally in the flash cache and lazily committed to the primary storage at a later point in time.

We evaluated all three open source flash cache solutions under write-back mode running the same TPC-E experiment as in Section 4.3. Because SCE only supports asynchronous write-through and write-around modes, we show the result of SCE with asynchronous write-through instead.

Results depicted in Figure 10 verify that the write-back mode significantly improves the performance compared to write-through: 97%, 83%, and 136% improved TpsE values were measured on *flashcache*, *bcache*, and *EnhanceIO*, respectively. Despite this improvement, SCE still outperformed the other solutions by at least 1.8×. More importantly, SCE does so without disrupting the resilience and

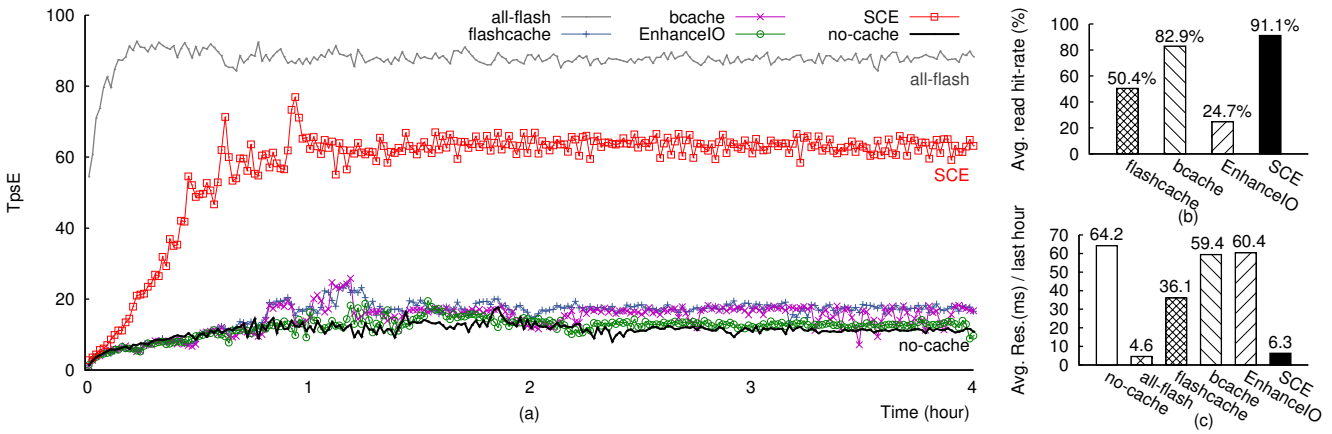

Figure 9: Four hour TPC-E run with a 30 GiB flash cache (a) TpsE: *when the flash cache size is smaller than the active working set size, the performance gains of the three open-source flash cache solutions are remarkably reduced – only about 55% by flashcache; SCE still demonstrated huge performance improvements – about 445%, which is about 3.6× higher TpsE than flashcache*, **(b) Last hour average read hit rate:** *SCE achieved 91.1% read hit rate in spite of limited cache size, and interestingly, the read hit rate of bcache is higher than that of flashcache, but its TpsE is similar or slightly lower than TpsE of flashcache.*, **(c) Last hour average response time:** *compared to the result with big enough flash cache (Figure 7) bcache and EnhanceIO show little increased response times and flashcache shows 3× longer response time; SCE achieved only 40% longer response time than the response time of all-flash configuration.*

high availability functions of the primary storage backend.

4.5 Impact of AWT and fragment size

As described in Section 3, two key features of SCE are AWT and coarse-grained cache management. We tried to quantify the effect of these features on the performance of SCE. To this end, we run TPC-E for four hours with a 30 GiB flash cache with four different SCE configurations: (a) SCE baseline: AWT enabled and a fragment size of 1 MiB, (b) AWT disabled and a fragment size of 1 MiB, (c) AWT enabled and a fragment size of 128 KiB, and (d) AWT enabled and fragment size of 8 MiB. With AWT disabled, SCE operates in write-around (or write-invalidate) mode. In this mode a write request is passed to the source device directly, and if a mapped fragment exists in the flash cache, the written pages are invalidated by using the page validity bitmap of the fragment.

Figure 11 depicts the performance for each of these four SCE configurations under TPC-E. First, AWT clearly is critical in sustaining high performance in the steady state under coarse-grain cache management and a population grain that is significantly larger than the size of the user I/O requests. By asynchronously re-populating invalidated pages in cached fragments when they are written to, SCE manages to keep fragments valid in their entirety. Second, the choice of fragment size makes a big difference in performance: small values result in significantly longer ramp-up time due to slower population (having 128 KiB fragments takes 8× longer to reach steady state than 1 MiB fragment); large values result in significantly reduced steady state performance due to a limited agility to adapt to the workload because of the increased population and eviction overheads.

4.6 SSD endurance

Unlike DRAM, flash has limited endurance in terms of the number of program erase cycles, and it is thus desirable to minimize the amount of writes to the flash caching SSD

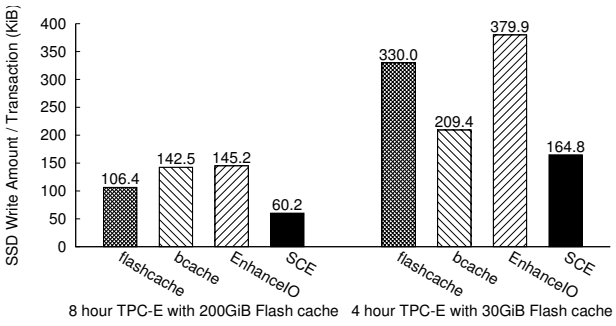

Figure 12: The amount of write to SSD per transaction: *SCE writes 43-50% less than the next best(flashcache) does.*

device in order to maximize its lifetime. We therefore compared the flash caches based on the amount of write I/O to the caching SSD for both the 200 GiB SSD cache and the 30 GiB SSD cache TPC-E runs. Figure 12 shows the total amount of writes to the SSD divided by total number of transactions (SSD writes in KiB/transaction). First, we observed that the 30 GiB flash cache configuration exhibits significantly higher SSD write traffic across all the flash cache solutions than the 200 GiB configuration. This can be attributed to the fact that with a 30 GiB configuration, the working set size is larger than the cache size. As a result both cache evictions and cache population are ongoing. Second, SCE exhibited a significantly lower amount of writes per transaction in both configurations: 43% less than *flashcache* in the 200 GiB configuration and 21% less than *bcache* in the 30 GiB configuration. It does so despite the fact that it populates at a 1 MiB fragment granularity as opposed to the other flash cache solutions that populate in 4 KiB blocks.

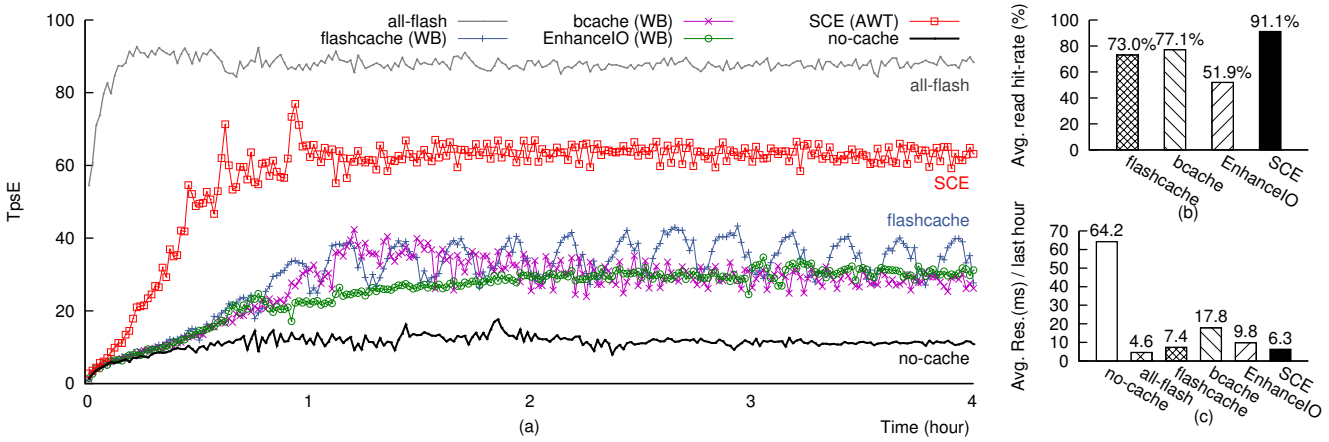

Figure 10: TPC-E results with write-back policy and 30 GiB flash cache (a) TpsE: *all three open-source flash cache solutions show much higher TpsE than with write-through policy*, **(b) Last hour average read hit rate:** *flashcache and EnhanceIO show higher hit rates while bcache shows lower hit rate than with write-through policy*, **(c) Last hour average response time:** *bcache achieved higher TpsE with much shorter response time but lower read hit rate than with write-through policy; this means that I/O latency can be more important than cache hit rate for database performance.*

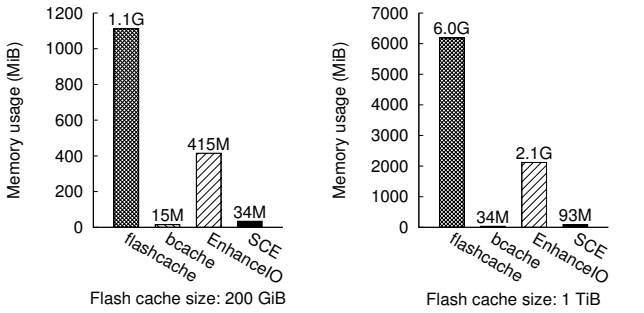

Figure 13: Memory foot-prints: *bcache used the smallest amount of memory because it keeps cache mapping information in a cache device as a B-tree; besides bcache, the memory foot-size of SCE is much smaller than flashcache and EnhanceIO.*

4.7 Memory usage

Flash cache solutions typically maintain their metadata in main memory; the space requirements of the metadata are typically proportional to the size of the cache. With SSD capacities already reaching several TBs, the memory footprint of a flash cache solution can be significant. We thus evaluated the memory efficiency of flash cache solutions by measuring their memory usage over the Linux */proc/meminfo* interface. Specifically, we first dropped all page, dentry and inode caches, and then read */proc/meminfo* before and after initializing the flash cache. We calculated the memory footprint of each flash cache solution based on the amount of free memory before and after initialization.

Figure 13 compares the memory foot-prints for two cache configurations: 200 GiB and 1 TiB of flash cache. *bcache* is the most memory efficient as it maintains its main metadata mapping information in the cache device as a B-tree. The non-volatility of *bcache* metadata, however, induces a heavy performance overhead as shown by the consistently high average response time it exhibited across our performance evaluation as presented in the previous paragraphs. SCE, on the other hand, achieves a small cache metadata footprint through coarse-grained cache management of 1 MiB fragments without sacrificing performance. This resulted in a 66× and 23× smaller memory usage for the 1 TiB cache size compared to *flashcache* and *EnhanceIO*, respectively.

5. RELATED WORK

Traditionally, the focus of research in the area of cache management has been on cache eviction policies. A number of cache eviction policies have been proposed including LRU, Clock [3], Generalized Clock [32], 2Q [19], LIRS [18], ARC [24], CAR [2] and Clock-Pro [17]. These policies have mostly been developed with RAM-based caches in mind and their main goal has been to optimally combine recency and frequency of accesses to maximize the cache hit rate, as well as to gracefully adapt to changing workloads. More recently, flash-aware cache management schemes such as CFLRU [28], LRU-WSR [20], and SpatialClock [21] have been proposed. These schemes have been designed for RAM-based caches on top of flash-based backing storage and their key focus continues to be on cache eviction. To the best of our knowledge, our work is the first attempt to throw the spotlight on cache population as opposed to cache eviction.

In the past few years, flash-based caching solutions have begun to attract the attention of both the industrial as well as academic research community. Researchers from NetApp proposed a flash caching solution called Mercury [4]. While interesting results were reported, the cache management scheme itself was not tailored to the characteristics of flash and therefore not of particular interest to flash-based caching.

Koller *et al.* published a study on write policies for host-side flash caches [22]. This work addressed the performance issues associated with write-through cache policies and proposed two new policies: *ordered write-back* and *journaled write-back*. While the policies address consistency issues arising from write-back caching, the proposed approaches

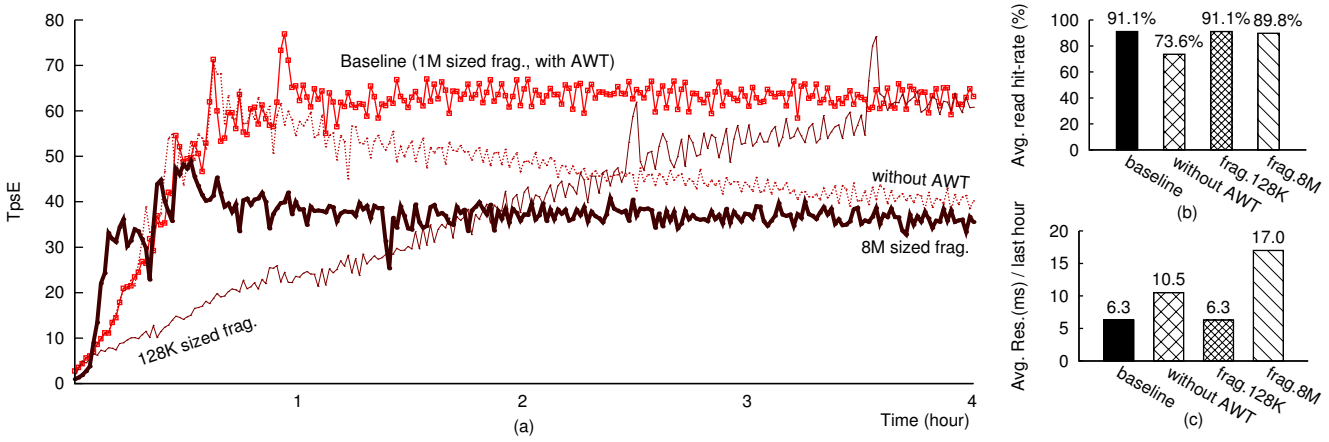

Figure 11: TPC-E with 30 GiB flash cache for different SCE configurations (a) TpsE: *AWT is critical in sustaining high performance; smaller sized (128 KiB) fragment configuration results in slower performance improvement while bigger sized (8 MiB) fragment configuration results in lower TpsE.* **(b) Last hour average read hit rate:** *without AWT, read hit rate is remarkably lower and it is the reason for the lower TpsE; 128 KiB fragment configuration achieved the same read hit rate with the baseline configuration; 8 MiB fragment configuration loses only little amount of cache hits but its TpsE value is even lower than without AWT configuration – this may be because of the the inefficiency of maintaining cache with a too coarse granularity.* **(c) Last hour average response time:** *without AWT configuration, response time increases because of lower cache hit rate while 8 MiB fragment configuration does because of too much cache population traffic – about 5.5× more amount of write traffic than baseline was given to the SSD.*

cannot achieve enterprise-class reliability and high availability which requires data to be available even in the event of SSD failures. Typically such reliability and high availability is achieved by employing redundancy at the storage backend (e.g., with RAID [29] for drives, dual controllers, multiple paths, etc.). We demonstrate through our experiments that SCE with asynchronous write-through can even outperform open-source solutions that use write-back policies as shown in Section 4.4.

Bonfire [36] was proposed to accelerate cache warm-up for large storage caches. However, the approach followed in that work is very different from our approach. Bonfire is external to the cache, i.e., it is a component that functions independently from the cache manager. Bonfire monitors storage workloads, records relevant metadata, and uses that information to execute warm-up explicitly. Our approach, on the other hand, is one that is integrated with cache management and does not rely on external components or prior knowledge of the workload. In SCE, the cache management module itself controls the rate at which the cache gets populated. Configuration parameters, such as the number of population threads, can be used to influence the population rate. That said, the two approaches are essentially orthogonal and, when used in combination, the benefits of both can be reaped.

In [11] Holland *et al.* present a performance evaluation of flash caches that utilizes a trace-driven simulation methodology and aims to understand the performance impact of various configuration parameters. However, some of our results seem to conflict with their conclusions. For example, our studies with the TPC-E benchmark indicate that caches with write-back policy can provide significantly better performance compared to caches with write-through policy. However, their study seems to indicate that the two policies do not impact performance significantly. This may be an artifact of the evaluation methodology or the workload used or may even be attributed to the write performance characteristics of the SSDs and the backend storage used in each case.

6. CONCLUSION

The focus of this paper has been on host-side flash-based caches, which have recently gained traction in enterprise environments to accelerate storage workloads. An experimental study of existing open-source flash cache implementations using the industry-standard benchmark TPC-E has revealed that there is still much to be desired in terms of performance and scalability. Prompted by our empirical observations, we architected and implemented SCE, a novel caching engine that employs coarse-grained caching in combination with background cache population and asynchronous write-through. Our experiments showed that SCE outperforms existing solutions, offering higher performance, shorter warm-up times and high scalability, thereby demonstrating both the potential and the necessity of this approach.

In the future, we plan to further optimize SCE to make it more appropriate for enterprise environments. To eliminate warm-up times after reboots we will exploit the non-volatility of flash to enable SCE to cache data persistently on the SSD. We also plan to add performance management capabilities, so that the caching driver can ensure that the SSD will never become a performance bottleneck. Another direction for future research is experimentation with different SSD devices to study how cache management algorithms should adapt to accommodate widely varying SSD performance characteristics.

7. REFERENCES

[1] Amazon. Amazon Relational Database Service (Amazon RDS) / DB Instance Classes. http://aws.amazon.com/rds/.

[2] S. Bansal and D. S. Modha. CAR: Clock with adaptive replacement. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, FAST '04, pages 187–200, Berkeley, CA, USA, 2004. USENIX Association.

[3] A. Bensoussan, C. Clingen, and R. C. Daley. The multics virtual memory: Concepts and design. *Communications of the ACM*, 15:308–318, 1972.

[4] S. Byan, J. Lentini, A. Madan, L. Pabon, M. Condict, J. Kimmel, S. Kleiman, C. Small, and M. Storer. Mercury: Host-side flash caching for the data center. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1–12, 2012.

[5] T. P. P. Council. A new On-Line Transaction Processing (OLTP) workload. http://www.tpc.org/tpce/.

[6] EMC. FAST: Fully Automated Storage Tiering. http://www.emc.com/storage/symmetrix-vmax/fast.htm.

[7] EMC. XtreamSF. https://store.emc.com/Solve-For/STORAGE-PRODUCTS/EMC-XtremSF/p/EMC-XtremSF.

[8] EMC. XtreamSW Cache: Intelligent caching software that leverages server-based flash technology and write-through caching for accelerated application performance with data protection. http://www.emc.com/storage/xtrem/xtremsw-cache.htm.

[9] Fusion-IO. ioTurbine: Turbo Boost Virtualization. http://www.fusionio.com/products/ioturbine.

[10] T. P. G. D. Group. PostgresSQL. http://www.postgresql.org/.

[11] D. A. Holland, E. Angelino, G. Wald, and M. I. Seltzer. Flash caching on the storage client. USENIXATC'13. USENIX Association, 2013.

[12] IBM. IBM FlashSystem 840. http://www.ibm.com/systems/storage/flash/840.

[13] IBM. IBM System Storage DS8000 Easy Tier. http://www.redbooks.ibm.com/abstracts/redp4667.html.

[14] IBM. IBM System Storage DS8000 Easy Tier Server. http://www.redbooks.ibm.com/abstracts/redp5013.html.

[15] IBM. IBM XIV Storage System. http://www.ibm.com/systems/storage/disk/xiv.

[16] IDC. Taking enterprise storage to another level: A look at flash adoption in the enterprise. http://www.idc.com/getdoc.jsp?containerId=236366.

[17] S. Jiang, F. Chen, and X. Zhang. Clock-pro: an effective improvement of the clock replacement. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 35–35, Berkeley, CA, USA, 2005. USENIX Association.

[18] S. Jiang and X. Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proc. of the ACM SIGMETRICS International conf. on Measurement and Modeling of Computer Systems*, 2002.

[19] T. Johnson and D. Shasha. 2q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 439–450, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[20] H. Jung, H. Sim, P. Sungmin, S. Kang, and J. Cha. LRU-WSR: Integration of LRU and Writes Sequence Reordering for Flash Memory. *IEEE Transactions on Consumer Electronics*, 54(3):1215–1223, 2008.

[21] H. Kim, M. Ryu, and U. Ramachandran. What is a good buffer cache replacement scheme for mobile flash storage? In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 235–246, New York, NY, USA, 2012. ACM.

[22] R. Koller, L. Marmol, S. Sundararaman, N. Talagala, and M. Zhao. Write policies for host-side flash caches. FAST'13. USENIX Association, 2013.

[23] L. S. P. Ltd. SSD caching: don't get too excited. http://www.raid6.com.au/posts/SSD_caching_problems/.

[24] N. Megiddo and D. S. Modha. ARC: a self-tuning, low overhead replacement cache. In *FAST '03: Proc. of the 2nd USENIX conf. on File and Storage Technologies*, pages 115–130, Berkeley, CA, USA, 2003. USENIX Association.

[25] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom. Sfs: Random write considered harmful in solid state drives. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, pages 12–12, Berkeley, CA, USA, 2012. USENIX Association.

[26] D. Mituzas. Flashcache at Facebook: From 2010 to 2013 and beyond. https://www.facebook.com/notes/facebook-engineering/flashcache-at-facebook-from-2010-to-2013-and-beyond/10151725297413920,h://github.com/facebook/flashcache/.

[27] NetApp. Flash Accel software improves application performance by extending NetApp Virtual Storage Tier to enterprise servers. http://www.netapp.com/us/products/storage-systems/flash-accel.

[28] S.-Y. Park, D. Jung, J.-U. Kang, J.-S. Kim, and J. Lee. CFLRU: a replacement algorithm for flash memory. In *CASES '06: Proc. of the 2006 International conf. on Compilers, Architecture and Synthesis for Embedded Systems*, pages 234–241, New York, NY, USA, 2006. ACM.

[29] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (raid). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, SIGMOD '88, pages 109–116, New York, NY, USA, 1988. ACM.

[30] PureStorage. FlashArray, Meet the new 3rd-generation FlashArray. http://www.purestorage.com/flash-array/.

[31] M. Saxena, Y. Zhang, M. M. Swift, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Getting real: Lessons in transitioning research simulations into hardware systems. In *Proceedings of the 11th Conference on File and Storage Technologies (FAST '13)*, San Jose, California, 02/2013 2013.

[32] A. J. Smith. Sequentiality and prefetching in database systems. *ACM Trans. Database Syst.*, 3(3):223–247, Sept. 1978.

[33] W. Stearns and K. Overstreet. Bcache: Caching beyond just RAM. https://lwn.net/Articles/394672/,http://bcache.evilpiepirate.org/.

[34] STEC. EnhanceIO SSD Caching Software. https://github.com/stec-inc/EnhanceIO.

[35] P. Tözün, I. Pandis, C. Kaynak, D. Jevdjic, and A. Ailamaki. From a to e: Analyzing tpc's oltp benchmarks: The obsolete, the ubiquitous, the unexplored. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, pages 17–28, New York, NY, USA, 2013. ACM.

[36] Y. Zhang, G. Soundararajan, M. W. Storer, L. N. Bairavasundaram, S. Subbiah, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Warming up storage-level caches with bonfire. FAST'13, San Jose, California, 02/2013 2013.