# How could a flash cache degrade database performance rather than improve it? Lessons to be learnt from multi-tiered storage.

*Hyojun Kim [#], Ioannis Koltsidas [*], Nikolas Ioannou [*], Sangeetha Seshadri [#]*
*Paul Muench [#], Clement L Dickey [#], Lawrence Chiu [#]*

[#] *IBM Research - Almaden,* [*] *IBM Research - Zurich*

## Abstract

Contrary to intuition, host-side flash caches can degrade performance rather than improve it. With flash write operations being expensive, cache hit-rates need to be relatively high to offset the overhead of writes. Otherwise, the end-to-end performance could be worse with flash cache.

We believe that some lessons learnt from multi-tiered storage systems can be applied to flash cache management. Multi-tiered storage systems migrate data based on long-term I/O monitoring, carefully ensuring that the background data migration does not adversely affect foreground I/O performance.

To test our hypothesis, we designed and implemented a new flash cache, named Scalable Cache Engine (SCE). In SCE, cache populations occur in the background in 1 MiB sized fragment units rather than the typical storage I/O size (4 KiB). By doing so, we warm-up the flash cache much faster while also benefiting from a prefetching effect that is very effective for improving cache hit-rates when the workload demonstrates strong spatial locality. Additionally, large, aligned writes to flash are much more efficient than small random ones and therefore reduce the cache population overhead. We show that our approach successfully tackles several issues of existing flash cache management approaches and works well for OLTP database workloads. For instance, the throughput under a TPC-E workload actually degraded by 79.1% with *flashcache*, a popular open-source solution, compared to the baseline performance. For the same conditions, SCE could achieve a 301.7% improved throughput.

## 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, it is not surprising that flash-based storage has been deployed at various levels of the enterprise storage architecture ranging from a storage tier in a multi-tiered environment (e.g., IBM Easy Tier [11], EMC FAST [6]) to a caching layer within the storage server (e.g., IBM XIV SSD cache [13]), and to a host-side cache (e.g., IBM Easy Tier Server [12], EMC XtreamSW Cache [7], NetApp Flash Accel [24], FusionIO ioTurbine [8]). More recently, several all-flash storage systems that completely eliminate HDDs (e.g., IBM FlashSystem 840 [10], Pure Storage [28]) have been introduced and have gained significant traction.

Due to its performance, capacity, and cost characteristics, flash memory fills the gap between DRAM and magnetic HDDs quite 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 nearly 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 on-line 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.

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 com-

pletely 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 [14]. 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 already been committed to the SAN such 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) use flash SSD instead of DRAM as the caching media, and 2) sit underneath DRAM caches in the storage hierarchy. With DRAM caches, population implies only a memory copy operation, and therefore, it is a good idea to populate on each cache miss; caches aim to cache the 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; some profits are expected with nearly zero investment. However, blindly populating all the data accessed by the user into the flash cache is not always prudent for flash-based caches. Cache population requires expensive flash write operations, which take long time and also consume device lifetime. It has already been pointed out that *populating upon every cache miss* may have a negative influence on the end-to-end performance [27, 20].

To offset the cache population overhead, high cache hit-rate are necessary. Unfortunately, there are multiple factors preventing flash-based caches from achieving good cache hit-rates. Flash cache typically sits under a substantial amount DRAM cache, which absorbs the hottest potion of the workload. Therefore, it 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, and the chance of cache hit becomes lower. To achieve high cache hit-rate for sparser and wider ranged data accesses, large capacity flash caches are required. Otherwise, *cache trashing* will happen and seriously degrade cache hit-rate. Lastly, the big capacity of flash cache brings another problem: slow cache warm-up time – it will be difficult to achieve a high cache hit-rate with a cold cache. Flash caches are expected to have orders-of-magnitude higher capacity than DRAM-based ones, in some instances up to several TiBs in size. At this scale, it may take several hours to fill up the cache. For instance, a 300 GiB-sized flash cache took longer than 10 hours to reach its maximum cache hit rate for an enterprise workload [4]. Even though cache warm-up is only a one-time event, it can still be a critical issue for some cases such as virtual machine migration and change of workload. Bonfire [31] was proposed as an auxiliary solution to accelerate cache warm-up for large storage caches.

We believe that flash-based caching techniques should learn lessons from multi-tiered storage systems such as IBM EasyTier [11] and EMC FAST [6]. Multi-tiered storage systems have multiple classes of storage devices such as high performance SSDs, enterprise-class HDDs with SAS interface, and nearline HDDs having SATA interfaces. To maximize performance and minimize cost, multi-tiered storage systems dynamically migrate data between storage tiers: hot data is placed on SSDs and cold data is placed on SATA HDDs. In these systems, the granularity of data movement is coarser compared to caching, and migration is typically performed in the background with a limited I/O bandwidth budget so as to not hurt foreground performance. Since the cost of moving larger chunks of data is high, the data migration decisions are made very cautiously based on long-term I/O accesses frequencies. Our intuition is that such mechanisms could also be applied to flash cache management.

To validate our hypothesis, we designed and implemented a new flash cache solution named Scalable Cache Engine (SCE) from scratch. We apply the lessons learned from the data migration process of multi-tiered storage systems to flash cache population. We propose a cache population scheme that has three major differences from existing approaches. First, we use bigger unit (e.g, 1 MiB) for cache population than the typical I/O access size (4 KiB). The typical approach to caching is keeping recently accessed data on the faster media, and therefore the cache population unit normally equals the I/O operation size. However, there are multiple reasons for choosing a larger cache management unit, and the reasons are connected to the other two differences of our population scheme. The second difference of our population scheme is to *not* populate on each cache-miss – in other words, we perform *selective cache population* like SieveStore does [27]. We monitor the I/O traffic and choose a population candidate based on access recency and frequency, and our coarse granularity is definitely useful for I/O monitoring. Finally, we propose to separate the cache population process from the foreground I/O activities so as to not influence I/O latency, similarly to how multi-tiered storage systems do background data migration. This separation allows us great flexibility; population related decisions can be dynamically made based on run-time metrics. For example, SCE performs cache

population more aggressively when the cache hit-rate is low and there is a lot of free space, and the population speed slows down as the cache hit-rate ramps up and the remaining cache space is reduced.

We have implemented SCE on Linux and carried out a performance study using th Sysbench OLTP benchmark [21] as well as the industry-standard TPC-E benchmark [5]. The results we obtained validate our expectations and confirm the superior performance of SCE compared to *flashcache* for both benchmarks. For a four-hour run of the Sysbench OLTP benchmark, SCE achieved 70.4% and 36.1% higher throughput than *flashcache* with 20 GiB (small) and 80 GiB (large enough) cache configurations, respectively. For an eight-hour run of the TPC-E benchmark with 40 GiB cache size, *flashcache* reduced the throughput by 79.1% compared to the baseline performance without flash cache. For the same conditions, SCE achieved a throughput improvement of 301.7%. The results demonstrate that our approach is an exceptionally good match for OLTP database workloads.

The rest of the paper is organized as follows: In Section 2, we present Sysbench OLTP benchmark results with *flashcache* that show how current flash cache management approaches fails to improve storage performance. We then present our approach to flash caching in Section 3. In Section 4 we revisit the experimental evaluation with Sysbench OLTP benchmark and further with the TPC-E benchmark. An overview of related work is given in Section 5. We present our conclusions and future work in Section 6.

## 2 Motivation

We tested *flashcache* version 3.1.1 with the Sysbench OLTP benchmark [21] to understand how a flash cache cause the performance of an OLTP database workload to degrade. Our experimental setup is a fairly typical one, where an enterprise database system is running within a virtual Linux 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 20 GiB / 40 GiB sized SSD partition as the caching device while the whole VM image size (including the Linux OS and the Sysbench OLTP database) was 160 GiB; the initial database size used by Sysbench OLTP benchmark was about 77 GiB – more technical details about our setup are given in Section 4. We repeated the same experiment for three configurations: 1) without a flash cache, 2) *flashcache* with a 20 GiB SSD, and 3) *flashcache* with a 40 GiB SSD. *flashcache* was configured in write-through mode, which is the standard choice for enterprise storage systems. For each run we measured the number of transactions per second (TPS), as well as I/O performance metrics and cache performance statistics.

In Figure 1 (a) we present the end-to-end Sysbench OLTP results for a 4-hour run of the benchmark. Surprisingly, the TPS numbers were reduced with flash caching: 22.6% and 9.5% fewer TPS were measured for the 20 GiB and 40 GiB cache configurations, respectively. For a better understanding, we present the I/O traffic statistics for the HDD and SSD in Figures 1 (b), (c), and (d). Without flash caching, about 14 MiB/s of read and 17 MiB/s of write traffic reached the HDD (Figure 1 (b)). With flash caching enabled, about 22 MiB/s of write traffic to the SSD was observed with both the 20 GiB and the 40 GiB configurations, while only about 2.5 MiB/s and 6.5 MiB/s of read traffic reached the SSD for the two configurations, respectively.

The results can be explained by the properties of the workload and the configuration. First, the workload incurs more writes than reads – flash caching with a *write-through policy* is more beneficial for read-intensive workloads. Second, the cache size is smaller than the size of the working set. In other words, the cache hit rate was too low in both cases; about 20% and 50% read-hit rates were achieved for the 20 GiB and 40 GiB cache configurations, respectively. The rate of SSD reads represents the rate of cache read-hits and the rate of SSD write represents the rate of cache population. Figures 1 (c) and (d) clearly show that there was too much cache population, i.e., too many flash writes.

We also experimented with the *write-around policy* of *flashcache*; it reduced the amount of write traffic substantially. The SSD write traffic became almost same with the HDD read traffic because HDD writes did not cause cache population. Nevertheless, there was no improvement in TPS and the amount of write traffic was still high.

On one hand, the observed performance degradation seems to be natural and unavoidable – *cache trashing* is a well-known problem and can be alleviated with bigger cache capacity. On the other hand, it still feels wrong; with a 40 GiB flash cache for a 80 GiB sized database, we only got a 50% cache read hit-rate, and about 10% worse overall performance. The 50% read hit-rate is also interesting if one takes into account that we configured the random function of Sysbench to be a *Pareto* distribution, i.e., follow the 80-20 rule. Why was only 50% read-hit rate achieved with 40 GiB sized flash cache? 20% of 80 GiB database is only 16 GiB, and thus a cache hit-rate higher than 80% was expected. These anomalies can be explained as follows: the flash cache sits underneath DRAM caches – for this measurement, the system memory size was 15 GiB. Since the hottest portion of the working-set was absorbed by the DRAM cache, the flash cache received almost uniformly distributed storage accesses. In such cases, cache population solely relying

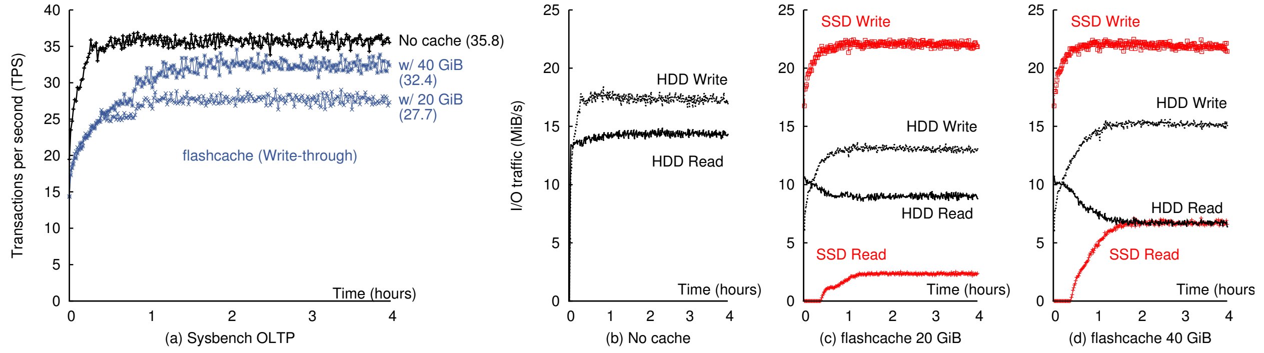


Figure 1: Sysbench OLTP results with *flashcache*: *(a) shows TPS for a 4-hour run for three configurations: without flash cache, with 20 GiB , and with 40 GiB cache capacity. (b), (c), and (d) show the amount of I/O traffic given to the SSD and HDD while running Sysbench OLTP – flashcache makes TPS number worse.*

on recency is clearly not enough. This indicates that we need more sophisticated population techniques for flash cache management.

## 3 Scalable Cache Engine

### 3.1 Selective coarse-grained population

Unconditional cache population on a cache-miss can cause performance issues a) for write intensive workloads and b) when the flash cache size is not big enough to hold the entire working set. Our solution, 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, that is, a fragment size of 1 MiB. 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 most recently accessed fragments. This also enables faster population since population occurs in larger units (1 MiB fragments), as opposed to finer 4 KiB pages.

### 3.2 Asynchronous background population

In-line cache population can slow down cache missed read operations. It can cause the overall I/O performance to degrade, especially when too many cache misses happen – for instance when the flash cache is empty or the workload changes. To deal with this, SCE employs *asynchronous background cache population* at a fragment granularity. As shown in Figure 2, we separate the cache population path from the foreground I/O data path. This

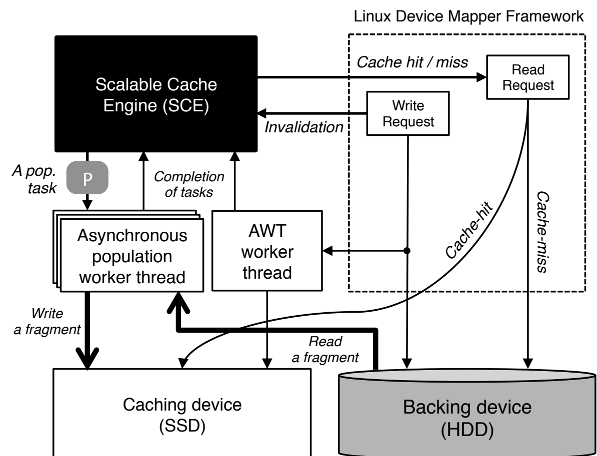


Figure 2: Asynchronous cache population in SCE: *SCE provides cache functions as a service; read requests are directed to the caching device or the backing device based on cache mapping information; write requests invalidate page validity bitmaps and are passed to backing device; multiple threads perform cache populations asynchronously in the background.*

approach borrows ideas from automated storage tiering mechanisms [6, 11]. 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 brought 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 degrada-

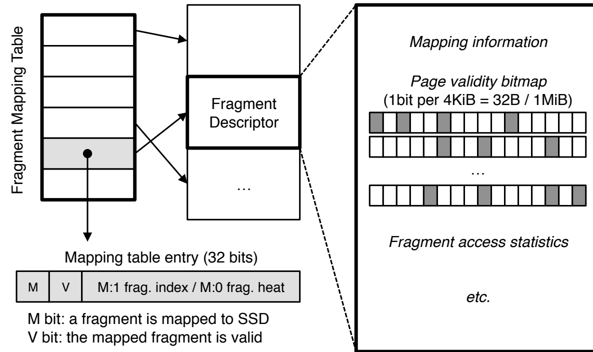


Figure 3: Coarse-grained cache management: *a direct mapping table is used to maintain fragment level mappings; each fragment descriptor has a page validity bitmap to track which parts of the fragment are valid.*

tion 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 a 4 KiB block may result in a 1 MiB fragment population. In practice, however, we have found that this acts as an effective prefetching mechanism that not only accelerates the 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 [23]. 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 virtual block size [29].

## 3.3 Coarse-grained cache management

Existing flash cache solutions normally 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.

Recall that 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 occurs at a 4 KiB block granularity. Obviously, coarse-grained cache management is more memory efficient than a fine-grained one. 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 the flash capacity grows.

A more complicated issue arises with respect to handling mismatches between I/O request size and 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. *flashcache* does not populate when an I/O request size does not match the cache mapping block size.

Figure 3 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 the 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.

# 4 Evaluation

For our evaluation, we used two OLTP workloads (Sysbench OLTP and TPC-E) with two database engines (MySQL for Sysbench and DB2 for TPC-E) and two different SSDs (1.8 TB and 825 GB PCI-e attached flash SSDs from different vendors) on two different back-end storage systems (a locally attached HDD and a remote SAN-attached storage volume).

## 4.1 Sysbench OLTP

We have already shown the performance impact of flash caching on Sysbench OLTP throughput in Section 2. To repeat our measurement easily, we created a Virtual Machine (VM) on KVM, and installed the benchmark within the VM. Flash caching was enabled in the KVM host running RHEL 6.5 on an IBM System x3650 M4 server. As a backing storage device, we created a 160 GiB sized partition on a RAID0 device using two 10 kRPM 500 GiB HDDs. For flash caching, we created a partition on the eMLC-based PCI-e enterprise-class SSD with a 1.8 TiB capacity.

### 4.1.1 Sysbench OLTP with 20 GiB

We first tested the case in which the flash cache capacity is much small compared to the working-set size, i.e., with 20 GiB flash cache size. Figure 4 (a) presents the measured throughput for three cases: 1) baseline, 2) *flashcache*, and 3) SCE. *flashcache* reduced the TPS throughput by 23% while SCE increased it by 32%. Figure 4 (b) shows the I/O traffic generated by *flashcache* to the HDD and to the SSD. When the flash cache was not large enough, *flashcache* generated roughly 9× more SSD write traffic than SSD read traffic; the SSD write traffic seems to be equal to the sum of the HDD write and HDD read traffic. Figure 4 (c) explains how SCE can achieve much higher throughput than *flashcache*; unlike *flashcache*, the SSD write traffic was well suppressed. This result is especially interesting, as the coarse-grained cache management of SCE could be expected to lose due to worse space efficiency.

### 4.1.2 Sysbench OLTP with 80 GiB

We validated that SCE could improve the performance of an OLTP database workload even with a flash cache capacity smaller than the active working-set size. What would happen when there is the cache capacity is large enough? Would SCE still be better than *flashcache*? To find out, we tested with 80 GiB of cache size. Figure 5 (a) shows the TPS throughput during the course of 4 hours. This time, *flashcache* could achieve a higher TPS throughput than the baseline; after the 4-hour run, the throughput had increased by 39%, and was still increasing. SCE achieved a higher throughput than *flashcache* in multiple ways: 36% higher TPS were measured, cache warm-up was dramatically faster, and the rate of SSD writes was much lower.

## 4.2 TPC-E

For the TPC-E benchmark we used the same method with a few differences: a 825 GiB sized PCI-e SSD and a SAN volume connected by Fibre Channel were used. A significant difference between a local HDD and a SAN volume is that SAN storage server has a substantial amount of RAM buffers – write response time becomes shorter. The TPC-E benchmark runs within a Kernel-based Virtual Machine (KVM) running Red Hat Enterprise Linux (RHEL) 6.5 and IBM DB2 Express-C v10.5. The TPC-E KVM instance has 8 CPUs and 15 GiB of RAM allocated to it – as suggested for an extra-large DB instance in Amazon cloud [1]. We tested *flashcache* tested in write-through and in write-back modes, and with a flash cache size of 40 GiB.

Figure 6 (a) and (b) show measured throughput and average response time. *flashcache* in a write-through configuration degraded the performance terribly: TPS throughput dropped by 5×, and response time increased by 14×. Unlike *flashcache*, SCE achieved 4× higher TPS and 2× shorter response time. We can see that the cache management of SCE is an exceptionally good match for the TPC-E OLTP database workload. SCE even achieved higher performance than *flashcache* in a write-back configuration – note that, SCE uses a mixture of write-around and write-through mode (no write-back). When a write request is given to a mapped fragment, SCE acts in write-through mode. Otherwise, it only writes to the backend storage (write-around mode). From Figure 6 (c), (d), and (e), we can see the amount of I/O traffic given to the SSD and to the HDD. SCE generated much higher SSD read traffic than SSD write traffic – which is desirable for a flash cache; get a large benefit with a small investment.

# 5 Related work

Traditionally, the focus of research around cache management has been on cache eviction policies. Various eviction policies have been proposed including LRU, Clock [3], Generalized Clock [30], 2Q [17], LIRS [16], ARC [22], CAR [2] and Clock-Pro [15]. 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 hit rate, as well as to gracefully adapt to changing workloads. More recently, flash-aware cache management schemes such as CFLRU [25], LRU-WSR [18], and SpatialClock [19] 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 the academic research community. Re-

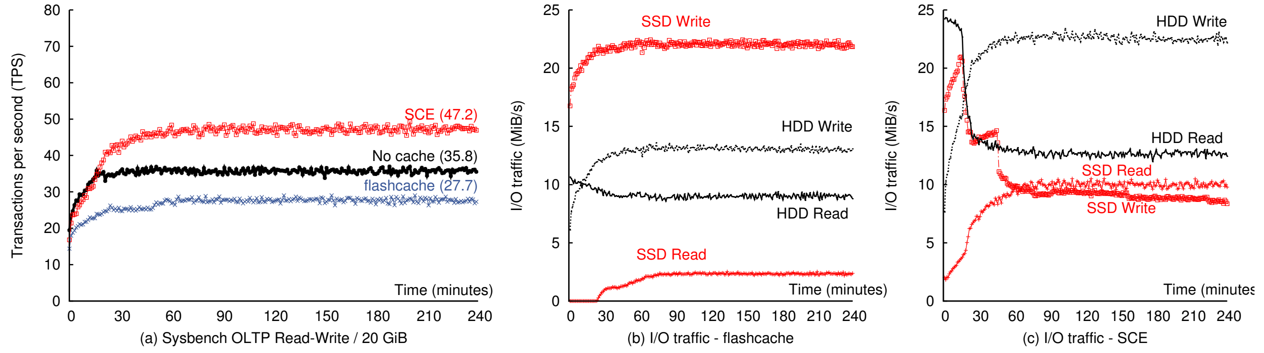


Figure 4: Sysbench OLTP read-write results with 20 GiB SSD, (a) TPS: *SCE increased TPS by 32% while flashcache decreased TPS by 23%.*, (b) I/O traffic for *flashcache*: *roughly 9× more SSD write traffic was observed than SSD read traffic – too much cache population happened.* (c) I/O traffic for SCE: *Compared to flashcache, much higher SSD read and much lower SSD write traffic were observed – background cache population increases HDD read traffic as much SSD write traffic.*

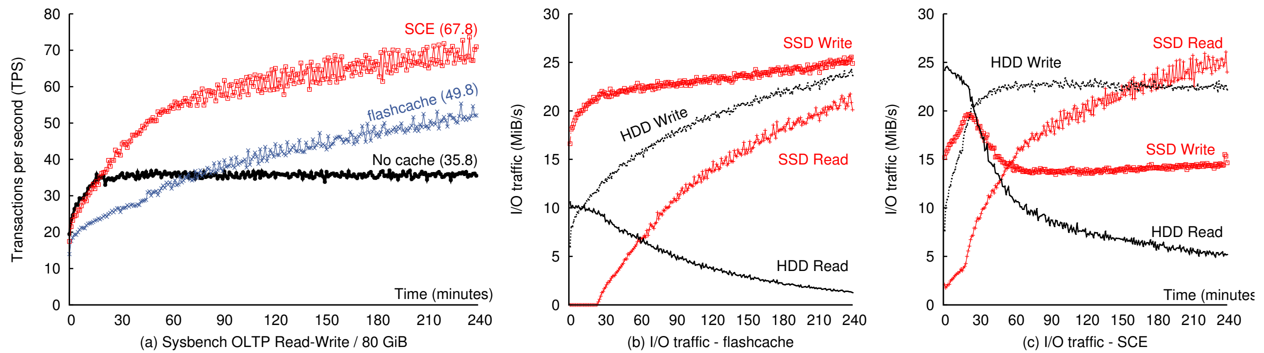


Figure 5: Sysbench OLTP read-write results with 80 GiB SSD, (a) TPS: *both SCE and flashcache increased TPS by 89% and 39%, respectively.*, (b) I/O traffic for *flashcache*: *SSD read traffic increased rapidly over time – still SSD write traffic was much higher than SSD read traffic* (c) I/O traffic for SCE: *after one hour after beginning, SSD read traffic became higher than SSD write traffic.*

searchers 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 [20]. 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 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 [26] for drives, dual controllers, multiple paths, etc.).

Bonfire [31] was proposed to address the 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 executes 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 [9] Holland *et al.* present a performance evalu-

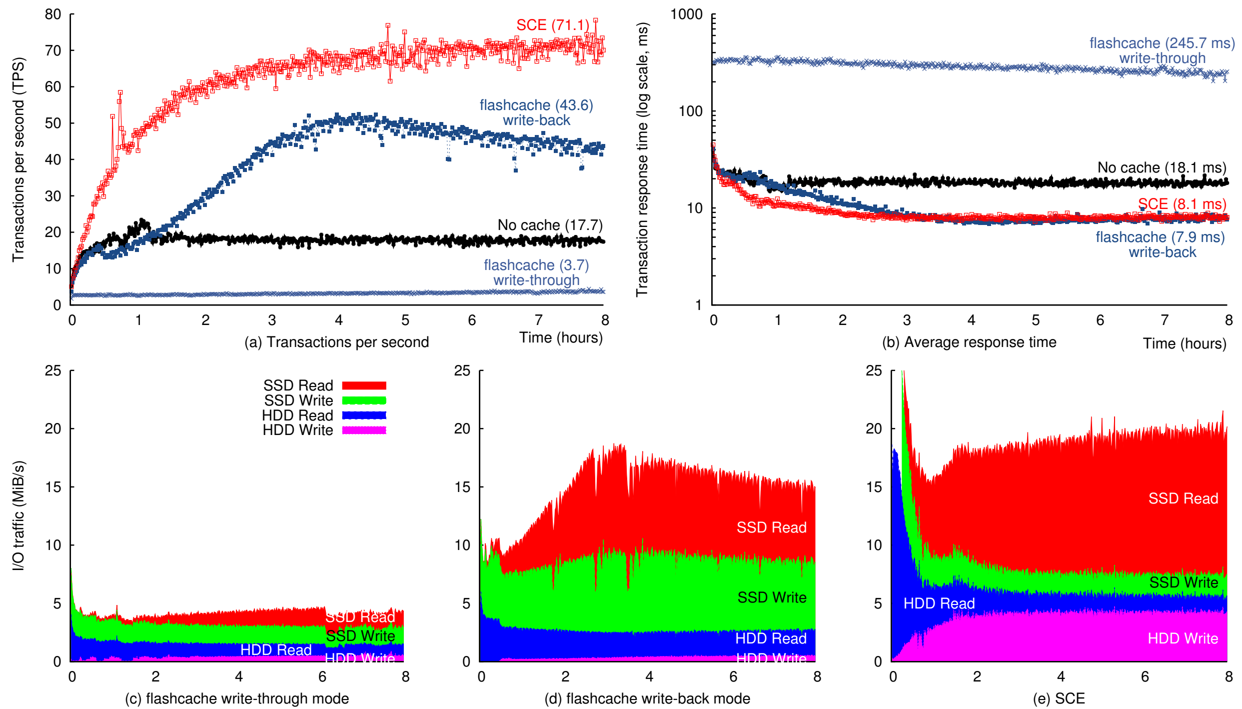


Figure 6: TPC-E results with 40 GiB SSD: (a) TPS: *SCE achieved 4× increased TPS than baseline but flashcache decreased TPS by 79% – SCE achieved even higher TPS than flashcache with write-back configuration*, (b) Average transaction response time: *SCE reduced the average response time by 55% while flashcache made it 14× longer than baseline – SCE achieved almost same response time with flashcache with write-back configuration.*, (c) I/O traffic for *flashcache* write-through mode, (d) I/O traffic for *flashcache* write-back mode, (e) I/O traffic for SCE.

ation 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 a 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 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. We show that flash-based caching is different than traditional DRAM-based caching: it can even make database performance worse. To alleviate this and maximize the benefit of flash caches, we design a flash cache from scratch. We borrow ideas from multi-tiered storage systems, and propose to use coarse grained, asynchronous, selective cache population. Our evaluation study shows that our approach works exceptionally well for OLTP database workloads. Even though our coarse-grained cache management sacrifices space efficiency, our selective asynchronous cache population could still improve the performance under a small flash cache capacity. With a big-enough flash cache, SCE demonstrated much faster cache warm-up time with less SSD write traffic, which translates to a longer lifetime for a flash device.

Even though extremely beneficial for OLTP database workloads, we are not claiming that the policy of SCE is always better than conventional fine-grained cache management. Our contribution is rather providing multiple knobs to control the behavior of flash cache population such as the fragment size and the number of cache population threads to improve the cache hit-rate and population speed. We were able to find an appropriate configuration and achieve a dramatic performance improvement for OLTP database workloads. In the future, we plan to introduce dynamic control of the configuration parameters to adapt to various types of storage workloads.

## References


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

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

[3] BENSOUSSAN, A., CLINGEN, C., AND DALEY, R. C. The multics virtual memory: Concepts and design. *Communications of the ACM 15* (1972), 308–318.

[4] BYAN, S., LENTINI, J., MADAN, A., PABON, L., CONDICT, M., KIMMEL, J., KLEIMAN, S., SMALL, C., AND STORER, M. Mercury: Host-side flash caching for the data center. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on* (2012), pp. 1–12.

[5] COUNCIL, T. P. P. 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. 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.

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

[9] HOLLAND, D. A., ANGELINO, E., WALD, G., AND SELTZER, M. I. Flash caching on the storage client. In *Proceedings of the 11th USENIX conference on USENIX annual technical conference* (2013), USENIXATC'13, USENIX Association.

[10] IBM. IBM FlashSystem 840. http://http://www-03.ibm.com/systems/storage/flash/840.

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

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

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

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

[15] JIANG, S., CHEN, F., AND ZHANG, X. Clock-pro: an effective improvement of the clock replacement. In *Proceedings of the annual conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2005), ATEC '05, USENIX Association, pp. 35–35.

[16] JIANG, S., AND ZHANG, X. 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).

[17] JOHNSON, T., AND SHASHA, D. 2q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 1994), VLDB '94, Morgan Kaufmann Publishers Inc., pp. 439–450.

[18] JUNG, H., SIM, H., SUNGMIN, P., KANG, S., AND CHA, J. LRU-WSR: Integration of LRU and Writes Sequence Reordering for Flash Memory. *IEEE Transactions on Consumer Electronics 54*, 3 (2008), 1215–1223.

[19] KIM, H., RYU, M., AND RAMACHANDRAN, U. 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* (New York, NY, USA, 2012), SIGMETRICS '12, ACM, pp. 235–246.

[20] KOLLER, R., MARMOL, L., RANGASWAMI, R., SUNDARARAMAN, S., TALAGALA, N., AND ZHAO, M. Write policies for host-side flash caches. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)* (San Jose, CA, 2013), USENIX, pp. 45–58.

[21] KOPYTOV, A. SysBench: a system performance benchmark 0.5. https://code.launchpad.net/~sysbench-developers/sysbench/0.5.

[22] MEGIDDO, N., AND MODHA, D. S. ARC: a self-tuning, low overhead replacement cache. In *FAST '03: Proc. of the 2nd USENIX conf. on File and Storage Technologies* (Berkeley, CA, USA, 2003), USENIX Association, pp. 115–130.

[23] MIN, C., KIM, K., CHO, H., LEE, S.-W., AND EOM, Y. I. Sfs: Random write considered harmful in solid state drives. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2012), FAST'12, USENIX Association, pp. 12–12.

[24] 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.

[25] PARK, S.-Y., JUNG, D., KANG, J.-U., KIM, J.-S., AND LEE, J. CFLRU: a replacement algorithm for flash memory. In *CASES '06: Proc. of the 2006 International conf. on Compilers, Architecture and Synthesis for Embedded Systems* (New York, NY, USA, 2006), ACM, pp. 234–241.

[26] PATTERSON, D. A., GIBSON, G., AND KATZ, R. H. A case for redundant arrays of inexpensive disks (raid). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 1988), SIGMOD '88, ACM, pp. 109–116.

[27] PRITCHETT, T., AND THOTTETHODI, M. Sievestore: A highly-selective, ensemble-level disk cache for cost-performance. In *Proceedings of the 37th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2010), ISCA '10, ACM, pp. 163–174.

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

[29] SAXENA, M., ZHANG, Y., SWIFT, M. M., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. 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).

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

[31] ZHANG, Y., SOUNDARARAJAN, G., STORER, M. W., BAIRAVASUNDARAM, L. N., SUBBIAH, S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Warming up storage-level caches with bonfire. In *Proceedings of the 11th Conference on File and Storage Technologies (FAST '13)* (San Jose, California, 02/2013 2013).