The Case For Flash-Aware Multi-Level Caching

Ioannis Koltsidas
School of Informatics
University of Edinburgh
i.koltsidas@sms.ed.ac.uk

Stratis D. Viglas
School of Informatics
University of Edinburgh
sviglas@inf.ed.ac.uk

ABSTRACT

The random read efficiency of flash memory, combined with its growing density and dropping price, make it well-suited for use as a read cache. In this paper we explore how a system can use flash memory as a cache layer between the main memory buffer pool and the magnetic disk. We study the problem of deciding which data pages to cache on flash and propose alternatives that serve different purposes. We provide an analytical cost model to decide the optimal caching scheme for any workload, taking into account the physical properties of the flash disk used. We discuss implementation issues such as the effect of the flash disk block size on performance. Our experimental evaluation shows that questions on systems with flash-resident caches cannot be given universal answers that hold across all flash disks and workloads. Rather, our cost model should be applied in each individual case to provide an optimal setup with confidence.

1. INTRODUCTION

With growing capacities, improved I/O performance, and constantly dropping prices, flash disks are now a solid storage alternative not only in personal computing, but also in the server market. In some cases, flash disks have completely replaced magnetic ones in the enterprise [12]; in other cases flash disks have been used along with magnetic disks to boost performance of a database [11]. The motivating impetus of our work is the observation that flash disks exhibit low latency and high random read efficiency. By comparing the price and performance characteristics of high-end flash disks to those of DRAM and magnetic disks, it follows that a flash disk is ideal to serve as a cache layer between main memory and magnetic disk. This implies a 3-tier memory hierarchy. In this paper we study various aspects of a system that uses a flash disk as a cache and provide analytical tools that enable the designer to decide with high confidence the optimal setup for such a system.

The (non-)mechanics of flash disks. Flash, or solid state, disks are an array of flash memory chips packaged with a controller in a single enclosure that provides a common interface (*e.g.*, SATA). Applications targeting flash disks should account for the I/O characteristics of flash memory; prominently, no mechanical moving parts and, thus, no mechanical latency. Hence, access latency is irrespective of the access pattern. At the same time, latency is orders of magnitude less than that of magnetic disks. The electrical properties of flash memory make reading the value of a bit faster than changing it. To make matters worse for writes, to update an already written sector one needs to first erase it. Erase operations can only be carried out in blocks, with each block (or, *erase unit*) typically consisting of 256 sectors. With each erase operation being two orders of magnitude more expensive than a read or a write, updating a sector is quite costly. The erase-before-write limitation of flash disks means that on-disk caches can do little to help. Therefore writes – especially random ones – perform very poorly. From a performance viewpoint, the random read efficiency of flash is its most advantageous feature, while its inherent shortcoming at random writes poses the greatest bottleneck.

Flash disks as caches. When designing a system with a 3-tier memory hierarchy like the one discussed here, one of the crucial decisions is determining the sizes of the main memory and the flash disk caches. As of November 2008, the cost of DRAM is about \$16/GB; the cost of flash disks varies from about \$1.6/GB for the low-performance ones [16], to about \$8/GB for the high-performance consumer flash disks [8], and to about \$30/GB for enterprise-level solutions [6]. Performance of flash disks in this price range varies by two orders of magnitude for reads and four orders of magnitude for random writes (refer to Table 2 of Section 5 for details). Considering the price/performance trade-off for the two types of cache, and given a specific budget, the question of what main memory and flash disk capacities minimize the price/performance ratio is not a straightforward one to answer. Should a flash disk be used as a cache at all? Or is it better to invest resources in DRAM only memory? Should one buy a small but very fast solid state disk or a larger (but slower) one? Such questions are crucial for the performance of the system, but cannot be given universally optimal answers. For instance, if one can buy enough DRAM to fit the working set of all the workloads that are likely to run on the system, then obviously one should go only with DRAM. Barring that, if the workloads are write-intensive it makes sense to invest in a high-performance flash disk to use as a cache, instead of a cheap one. In the general case it is neither easy nor safe to make decisions based on intuition; as both price and performance characteristics of flash disks are constantly

.

changing, decisions should constantly be re-evaluated.

At the next level, the designer is to decide which data will be cached on the flash disk. In contrast to buffering in main memory, pages do not need to be brought into the flash cache before they are processed. That is, a page may go directly from the magnetic disk to the memory and may well never be written to flash. Thus, deciding how data should flow from one level of the memory hierarchy to the others is not straightforward. A set of rules dictates the flow of data pages from one level to the others: we term this a *page flow scheme*. Relevant issues include how the workload of a page affects the decision about caching the page or not. For instance, in the ZFS filesystem [17] dirty pages are never cached on flash. From an implementation perspective questions arise about the directory of pages cached on the flash disk and the optimal page size to use on flash. We show why these questions are crucial and provide the tools to address them. Our proposals and results are independent of the page replacement algorithm used by either cache.

Contributions and organization. In what follows we present how a flash disk can efficiently act as a page cache between the main memory and the magnetic disk. Our main contributions are:

- We study the problem of deciding which data should be placed in the flash cache of a system. We identify and propose three invariants for the sets of pages cached either in main memory or on flash.
- For each invariant, the flow of pages between levels of the memory hierachy is different. We present the page flow scheme of each invariant and a I/O-based cost model for the scheme.
- We discuss several implementation issues that arise when using a flash disk as a cache: (*a*) the page directory for the cache, (*b*) the size of flash pages, and (*c*) caching only pages that satisfy specific predicates; we show the correlation between each alternative and the properties of the flash disk.
- We have implemented our proposals and conducted an extensive experimental study. Our results show that most questions regarding flash-resident caches cannot be given universally optimal answers; rather, our cost model should be used to answer such questions for each individual case with confidence.

The rest of this paper is organized as follows. Related work is presented in Section 2. We show the invariants of flash-resident data caching, and the page flow schemes implementing them in Section 3. In Section 4 we enumerate implementation alternatives for designing a flash cache. We undertake an extensive experimental study in Section 5, and discuss how our proposals can be used in a real system in Section 6. We conclude and present our future research directions in Section 7.

2. RELATED WORK

From a performance viewpoint, the main concern about flash memory is its random-write inefficiency due to its erase-before-write limitation. In [4] the authors study different write patterns on flash disks. Their results show that (*a*) latency greatly affects performance, (*b*) I/O in larger blocks can substantially improve random writes, (*c*) I/O blocks should be aligned to flash pages, and (*d*) if random writes exhibit some degree of spatial locality they can be performed almost as efficiently as sequential ones. The authors also point out that the behavior of flash disk drives is more complex than that of "bare" flash chips. This is due to the on-disk DRAM and controllers of flash disks; the main goal of such components is to improve random writes. To that end, they employ parallelism when accessing flash chips along with elaborate *Flash Translation Layer* (FTL) algorithms (for details see [1, 3, 5, 10]).

To improve write efficiency for flash-based databases, [12] proposes an *in-page logging* (IPL) scheme: changes to a data page are not written directly to disk, but to log records associated with the page. Changes are logged on a per-page basis, while each data page and its log records are on the same physical block of the disk, *i.e.*, in the same erase unit. When there is no room for new log records, log records and data pages are merged into a new erase unit. Thus, updates only write already erased sectors; simulation results show that IPL substantially improves performance. In [15] the authors argue that when writing to flash one should avoid in-place updates and sub-block deletions, while random writes should be replaced with semi-random ones; according to the latter pattern, blocks can be selected in any order to be written to, but sectors that belong to the same block are written sequentially from the start of the block. Experimental results confirm that these principles can boost random write performance. In [11] the authors study systems equipped with both a flash and a magnetic disk and propose placing data pages with read-intensive workloads on flash and pages with update-intensive workloads on the magnetic disk, to avoid the high-cost of random flash writes.

Research has also been conducted on buffer management over flash disks, *i.e.*, when the flash disk is used for persistent storage and not as an extended memory buffer. In [9] the authors propose BPLRU, a scheme for the on-disk buffer cache of flash disks. This scheme treats the buffer as a write cache and groups RAM buffers in blocks that are equal in size to the flash erase-unit; page replacement is performed with erase-unit granularity (using LRU). If not all sectors of a dirty victim page are present in-memory, the absent ones are read from disk so that the whole block can be written to a new flash location without the need for an in-place update. Additionally, a block that was written sequentially is moved to the tail of the LRU list and becomes the next victim. Experimental evaluation shows this technique to be very promising. Similarly, the authors of [18] propose that the buffer cache choose for replacement a clean page over a dirty one and therefore trade the number of writes with the number of reads. This concept is generalized in [11] for a system in which the same buffer pool holds pages from both the flash and the magnetic disks. In that case, not only the dirtyness of the page, but also the read/write costs of the storage medium and the access history of the page are considered when choosing a page to replace.

In [14] the authors describe an automated tool that can decide what storage hardware configuration is optimal for a specific workload, using multiple metrics such as performance and energy efficiency. The proposed tool works only in an offline fashion and for specific workloads and does not explore in any way the relationship of the data cached on flash with the data cached in RAM. Our proposals go beyond the offline choice of the optimal hardware, to the online decision of which data should be cached on flash, with respect to the ones cached in RAM. Another relevant piece

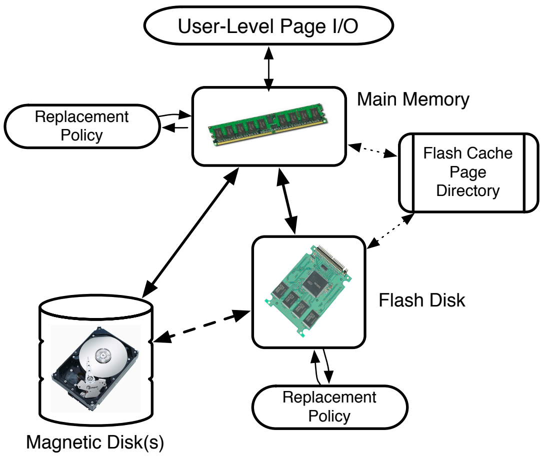


Figure 1: An overview of our system

of work – which, however, is not geared towards database workloads – is outlined in [13] and implemented in the ZFS filesystem [17]. The flash disk is used as a cache for the magnetic disk with the goal of improving the performance of random read workloads. In that setup there is no eviction from main memory to the flash disk; rather, the flash cache stores main memory pages before they are evicted. Filling the flash cache with pages is performed asynchronously, thereby avoiding write latencies on main memory evictions. By employing large sequential writes to predictively push data to the flash cache, the system also avoids paying the cost of random writes and increases the flash write bandwidth. Also, the flash cache never stores dirty pages and therefore no write-back to disk is required for flash pages. Our work is an analytical study of the behavior of flash caches; the techniques engineered in ZFS are thus complementary to our work and can provide a suitable implementation basis.

3. PAGE FLOW SCHEMES

We describe different page flow schemes to be used in a system employing a flash disk as a page cache between the main memory buffer pool and the hard disk. As our page flow schemes are independent of the page replacement policy, we do not consider the latter neither for the main memory buffer pool, nor for the flash-resident page cache on flash.

3.1 Problem Statement

Consider a database, or any other data processing system, with three hardware components for data storage and staging: (a) RAM memory (*e.g.*, DRAM chips), (b) one or more flash disks, and (c) persistent storage, *i.e.*, a single hard disk, an array of hard disks, or any other collection of storage media. Data processing requires demand paging: pages are brought into main memory before processing, and this happens only on page referencing. The high-level architecture of such a system is shown in Figure 1. We refer to main memory as *RAM* and to the main memory buffer pool as *RAM cache*. We use *FLASH* to refer to the system's flash disk(s) used as a page cache (the on-flash cache is called *FLASH cache*); *HDD* refers to the underlying long-term storage.

When designing a page cache, the principal decision is which pages will be cached; how long pages are cached for is determined by the replacement policy, which we do not consider. For a system of only a RAM cache and a hard disk, the former decision is not hard to make: *all* referenced pages will be written to the RAM cache, as this is required to use them. For a system with a FLASH cache in addition to the RAM one, however, there is no such requirement. As our goal is to reduce I/O operations to and from the HDD, the most reasonable thing to do under on-demand paging is to store on the FLASH cache the "hot" portion of the dataset that cannot fit in RAM. Let $P_{RAM}(t)$ be the set of pages stored in the RAM cache at some point in time t, and $P_{FLASH}(t)$ be the set of pages on the FLASH cache (for all practical cases, $|P_{RAM}(t)| < |P_{FLASH}(t)|$). We have identified the following three potential invariants:

1. $\forall t\ P_{RAM}(t) \bigcap P_{FLASH}(t) = P_{RAM}(t)$
   Whenever a page is in RAM, it is also cached on FLASH. This is analogous to the case of *inclusive* cache memory hierarchies of processors.
2. $\forall t\ P_{RAM}(t) \bigcap P_{FLASH}(t) = \emptyset$
   No page is stored on *both* RAM and FLASH at any time. A page brought from FLASH to RAM is removed from FLASH (and vice versa). Specifically, a RAM victim is stored in the frame of the page hit on FLASH, *i.e.*, a RAM page is *swapped* with a FLASH page.
3. $\forall t\ P_{RAM}(t) \bigcap P_{FLASH}(t) \subseteq P_{RAM}(t)$
   A page on RAM may or may not be cached on FLASH, depending on criteria either set by the user or decided based on the current workload.

Enforcing any one of the above invariants results in a different flow of pages across the levels of the memory hierarchy; thus, we define three different *page flow schemes*. Each scheme incurs a different I/O cost for a given workload. We detail all three schemes and model their I/O costs, without assuming any specific replacement policy for either the RAM or the FLASH cache. Page replacement is orthogonal to deciding which pages should be cached and where, which is the problem we study here. Therefore, our schemes may be used with any replacement policy.

3.2 The inclusive scheme

The inclusive scheme enforces the first invariant where the set of pages cached in RAM is always a subset of the pages cached on FLASH. The algorithm for fetching a page under this scheme is given in Figure 2. On a page reference, we look the page up in the directory for the main memory cache. If the page is found it is served in-memory. Else, we need to bring it in main memory and evict a page if memory is full. Given the invariant, the page must have been cached on flash, so it is written back only if it is dirty. We look the page up in the FLASH cache directory and, if the page is found, we read it from FLASH and put it into the RAM cache; else the page is read from HDD, written to the FLASH cache and then to the RAM cache. If the FLASH cache is full, a page needs to be evicted from FLASH; if dirty, it will be written to HDD. Since $|P_{RAM}(t)| < |P_{FLASH}(t)|$, v_f will not exist in RAM if both caches use the same page replacement algorithm; if this is not true the FLASH replacement policy needs to ensure that it never evicts a page currently in RAM.

Let h_r, m_r, h_f and m_f respectively be the total number of RAM hits, RAM misses, FLASH hits and FLASH misses incurred by the whole workload. Also, let F_R, F_W, D_R, D_W

```
Algorithm inclusive fetchPage (Page $pg$)
1.  if ($pg$ in RAM buffer pool)
2.    return $pg$
3.  else if ($pg$ in FLASH cache)
4.      Evict a victim page $v_r$ from RAM
5.      Write $v_r$ to FLASH, iff it is dirty
6.      Read $pg$ from FLASH
7.      return $pg$
8.    else
9.      Evict a victim page $v_r$ from RAM
10.     Write $v_r$ to FLASH, iff it is dirty
11.     Evict a victim page $v_f$ from FLASH
12.     Write $v_f$ to HDD, iff it is dirty
13.     Read $pg$ from HDD
14.     Write $pg$ to FLASH
15.     return $pg$
```

Figure 2: The inclusive page flow scheme

be the average cost of a flash read, a flash write, an HDD read and an HDD write, respectively. These include the cost of writing the page to RAM or reading the page from RAM. Furthermore, consider the probability that a page in RAM is dirty before its eviction and let this probability be p_d. Let R_{RAM} be the cost of running the replacement algorithm for the RAM cache and R_{FLASH} be the corresponding cost for the FLASH cache. For the remainder of this paper, we assume constant time replacement algorithms, *i.e.*, R_{RAM} and R_{FLASH} are negligible; still, we include them in the cost formulas for completeness.

For each RAM hit, there is no I/O cost. For each RAM miss either a FLASH hit or a FLASH miss occurs (*i.e.*, $m_r = h_f + m_f$). For each FLASH hit, a page is evicted from RAM with cost $R_{RAM} + p_d F_W$ and a page is read from FLASH with cost F_R. For each FLASH miss, a RAM page is evicted with cost $R_{RAM} + p_d F_W$; also, a flash page is evicted with cost $R_{FLASH} + p_d D_W$ and the referenced page needs to be read from disk and written to FLASH, with cost $D_R + F_W$. The cost C_1 of inclusive is:

$$\begin{aligned} C_1 &= h_f(F_R + R_{RAM} + p_d F_W) \\ &+ m_f(R_{RAM} + p_d F_W + R_{FLASH} + p_d D_W + D_R + F_W) \\ \Rightarrow C_1 &= h_f F_R + m_r(R_{RAM} + p_d F_W)) \\ &+ m_f(R_{FLASH} + p_d D_W + D_R + F_W) \end{aligned}$$

We have not taken into account the operations on the RAM and FLASH page directories. If the page directory is stored in-memory for both caches, the cost of a lookup or an update is $O(1)$ – at least for computationally cheap replacement policies like LRU. As we will discuss later, however, the page directory of the FLASH cache may require a substantial portion of the main memory. In systems with limited main memory it may be more efficient to store the FLASH directory on FLASH itself. On this ground, we also need to take into account the costs of lookups and updates to the FLASH cache directory. Let L be the cost of a lookup on the FLASH directory and U be the cost of a directory update. An update is either the insertion or deletion of a page, or a bookkeeping update (*e.g.*, moving the page to the MRU position, if LRU is used). On a RAM miss, inclusive pays the cost of a FLASH directory lookup; on RAM eviction, it pays the update cost if the victim page is dirty, *i.e.*, $p_d U$ (no lookup is required as the RAM victim will also be on FLASH). For each FLASH hit the bookkeeping of the directory is updated, while for each FLASH miss two updates are required: one for the victim page and one for the new page that is fetched. The cost of maintaining the directory C_1^d for the whole workload is:

$$\begin{aligned} C_1^d &= m_r(L + p_d U) + h_f U + m_f(U + U) \\ &= m_r(L + U + p_d U) + m_f U \end{aligned}$$

Hence, the total cost for inclusive is $C_1' = C_1 + C_1^d$.

```
Algorithm exclusive fetchPage (Page $pg$)
1.  if ($pg$ in RAM buffer pool)
2.    return $pg$
3.  else if ($pg$ in FLASH cache)
4.      Read $pg$ from FLASH
5.      Pick a victim page $v_r$ from RAM
6.      Replace $pg$ with $v_r$ on FLASH
7.      return $pg$
8.    else
9.      Evict a victim page $v_f$ from FLASH
10.     Write $v_f$ to HDD, iff it is dirty
11.     Evict a victim page $v_r$ from RAM
12.     Write $v_r$ to FLASH
13.     Read $pg$ from HDD
14.     return $pg$
```

Figure 3: The exclusive page flow scheme

3.3 The exclusive scheme

The exclusive page flow scheme enforces Invariant 2: the set of pages cached in-memory and the set of pages cached on FLASH are disjoint. The exclusive algorithm for fetching a page is given in Figure 3. The case for a RAM hit is the same as for inclusive. When a RAM miss occurs, we look the page up in the FLASH cache directory; if found, the page is read from FLASH. If the RAM cache is full, a page will be evicted from RAM. The victim is selected by the replacement policy and written to FLASH (whether it is dirty or not), while the referenced page is deleted from FLASH and inserted into RAM. Effectively, we swap the on-flash referenced page with the RAM victim. For a FLASH miss, the RAM victim is written to FLASH and the referenced page is read from the HDD into main memory. If the FLASH cache is full we also need to evict a page from FLASH.

For each RAM hit, no I/O cost is paid. Each RAM miss results in either a FLASH hit or a FLASH miss. For each FLASH hit the cost of evicting from RAM is $R_{RAM} + F_W$. The referenced page is read from FLASH with cost F_R and the victim page is written to FLASH with cost F_W. For each FLASH miss, FLASH eviction costs $R_{FLASH} + p_d D_W$ on top of the $R_{RAM} + F_W$ cost of evicting from the RAM cache. Moreover, reading the referenced page from HDD adds a cost of D_R. Thus, the cost of exclusive is:

$$\begin{aligned} C_2 &= h_f(R_{RAM} + F_R + F_W) \\ &+ m_f(R_{RAM} + F_W + R_{FLASH} + p_d D_W + D_R) \end{aligned}$$

Let us consider the FLASH cache directory maintenance cost for exclusive. For each RAM miss, L cost units are paid for a FLASH lookup. For a FLASH hit we pay U cost units: the hit page is replaced by the RAM victim and the directory bookkeeping is updated. In the event of a FLASH miss the cost is equal to $U+U$: a page is evicted from FLASH to HDD and another is evicted from RAM and written to FLASH. Hence, the directory maintenance cost is equal to:

$$\begin{aligned} C_2^d &= m_r L + h_f U + m_f(U + U) \\ &= m_r(L + U) + m_f U \end{aligned}$$

The total cost for exclusive is equal to $C_2' = C_2 + C_2^d$.

```
Algorithm lazy fetchPage (Page $pg$)
1.  if ($pg$ in RAM buffer pool)
2.    return $pg$
3.  else if ($pg$ in FLASH cache)
4.      Read $pg$ from FLASH
5.      Evict a victim page $v_r$ from RAM
6.      if $v_r$ in FLASH cache
7.          Write $v_r$ back to FLASH, iff it is dirty
8.      else
9.          Evict a victim page $v_f$ from FLASH
10.         Write $v_f$ to HDD, iff it is dirty
11.         Write $v_r$ back to FLASH
12.     return $pg$
13.   else
14.     Evict a victim page $v_r$ from RAM
15.     if $v_r$ in FLASH cache
16.         Write $v_r$ back to FLASH, iff it is dirty
17.     else
18.         Evict a victim page $v_f$ from FLASH
19.         Write $v_f$ to HDD, iff it is dirty
20.         Write $v_r$ back to FLASH
21.     Read $pg$ from HDD into RAM
22.     return $pg$
```

Figure 4: The lazy page flow scheme

3.4 The lazy scheme

The lazy page flow scheme enforces Invariant 3 by caching an arbitrary set of referenced pages in the FLASH cache. Generally, the system decides if a page will be cached on FLASH when it evicts the page from main memory, *i.e.*, after the system has an indication for the workload of a page by applying user-specified criteria. In the simple case, which we will focus on for the moment, no such criterion is used; rather, a RAM victim is always written to FLASH and stays there until evicted by the FLASH cache replacement policy. The lazy algorithm for fetching a page is given in Figure 4. A page is served in-memory if found in RAM. Else, we look it up in the FLASH directory. If a FLASH hit occurs, the page is read from FLASH (and the directory's bookkeeping is updated). If the RAM cache is full, a victim is picked by the RAM replacement policy and then evicted. We must then find out if the victim is also on FLASH. If so, it is written back only if it has been dirtied in RAM. Otherwise, a page is evicted from FLASH (and written back to HDD) to make room for the RAM victim to be written to FLASH. For a FLASH miss, a page is evicted from RAM and written to the FLASH cache in the same fashion as for a FLASH hit. The referenced page is read from HDD and brought directly into main memory. Note that on Line 9 one can apply any predicate based on the workload history for that page to decide whether the page should be cached on FLASH or not. We discuss such alternatives later on.

We now consider the cost of the lazy scheme. A main memory victim may or may not exist in the FLASH cache. We assume that the probability for a RAM victim to be on FLASH is q. The cost associated with a RAM victim C_3^V (*i.e.*, the cost of Lines 5-11, 14-20) is equal to $R_{RAM}+p_dF_W$ if the page is on FLASH and $R_{RAM}+R_{FLASH}+p_dD_W+F_W$ otherwise. Hence:

$$C_3^V = R_{RAM} + qp_dF_W + (1-q)(R_{FLASH} + p_dD_W + F_W)$$

For a FLASH hit the cost is $F_R + C_3^V$ and for a a FLASH miss the cost is $C_3^V + D_R$. The cost of this scheme is therefore:

$$\begin{aligned} C_3 &= h_f(C_3^V + F_R) + m_f(C_3^V + D_R) \\ &= (h_f + m_f)C_3^V + h_fF_R + m_fD_R \end{aligned}$$

As noted in Section 3.2, both h_f and m_f represent the total hits and misses for pages referenced in the workload. Thus, lookups in the FLASH index for the RAM victim are not accounted for by h_f and m_f. However, the probability of the RAM victim being on FLASH is expected to be equal to the probability of any referenced page being on FLASH: it does not depend on whether the looked-up page was in RAM at the time of the lookup. Therefore, $q = \frac{h_f}{h_f+m_f}$ and $1-q = \frac{m_f}{h_f+m_f}$, which gives that:

$$\begin{aligned} C_3^V &= R_{RAM} + \frac{h_f}{h_f+m_f}p_dF_W \\ &+ \frac{m_f}{h_f+m_f}(R_{FLASH} + p_dD_W + F_W) \end{aligned}$$

Then:

$$\begin{aligned} C_3 &= (h_f+m_f)R_{RAM} + h_fp_dF_W + h_fF_R \\ &+ m_f(R_{FLASH} + p_dD_W + F_W) + m_fD_R \\ \Rightarrow C_3 &= h_f(R_{RAM} + p_dF_W + F_R) \\ &+ m_f(R_{RAM} + R_{FLASH} + p_dD_W + D_R + F_W) \end{aligned}$$

As for the maintenance cost of the FLASH page directory, lazy pays L cost units for each RAM miss to look the page up in the FLASH directory. For a FLASH hit, U cost units are paid to update the entry for the hit page, in addition to the cost $C_3^{V,d}$ of updating the directory entry for the RAM victim. If a FLASH miss occurs, only $C_3^{V,d}$ cost units are paid for directory maintenance. For $C_3^{V,d}$, we have that:

$$C_3^{V,d} = L + qp_dU + (1-q)(U+U)$$

Hence, we have that:

$$\begin{aligned} C_3^d &= m_rL + h_f(U + C_3^{V,d}) + m_fC_3^{V,d} \\ \Rightarrow C_3^d &= m_r(L + C_3^{V,d}) + h_fU \\ \Rightarrow C_3^d &= m_r(2L + qp_dU + 2(1-q)U) + h_fU \\ \Rightarrow C_3^d &= 2(h_f+m_f)L + h_fp_dU + 2m_fU) + h_fU \\ \Rightarrow C_3^d &= h_f(2L + (p_d+1)U) + 2m_f(U+L) \end{aligned}$$

And, for lazy: $C_3' = C_3 + C_3^d$.

Various criteria can be applied to decide whether a RAM victim page should be cached on flash. For instance, if the flash disk used is poor in random writes, one can avoid some random writes by caching only clean pages. We experiment with this in Section 5.6. Alternatively, the access history for each FLASH page can be maintained, *e.g.*, by keeping track of the number of hits the page has seen, or the number of times it has been dirtied. The system could then maintain a set of the f hottest pages, where f is the number of pages that fit in FLASH. Then, only these f pages will be cached on flash, thus implementing a frequency based replacement policy. Similarly, one may decide to cache the f pages that have the most read-intensive workload as in [11]. Another alternative is to only cache pages that have been accessed at least twice while they were cached in RAM; that way, one can avoid polluting the flash cache when a file scan occurs. The mentioned options can even be combined; however, we will not study them further here, as they assume or define some aspects of the replacement policy of the cache.

3.5 Comparison

We compare the page flow schemes on the basis of their I/O costs. We assume (for now) that the FLASH cache directory is stored in main memory and we therefore do not consider directory maintenance costs. Given the cost formulas for C_1, C_2, C_3, it is tempting to factor out the common term $h_f F_R + m_f(R_{FLASH} + F_W + D_R) + m_r R_{RAM}$. However, this would make the invalid assumption that, for a fixed workload, h_f, m_f remain the same for all three page flow schemes. We will now show why this is not true.

Assume that any given workload is executed on the same system three times, once with each page flow scheme presented. In all cases the RAM and FLASH replacement policies are the same. What is more, assume a stack replacement algortithm (not a FIFO one), *i.e.*, one that does not exhibit Belady's anomaly [2] and therefore the hit ratio for the policy grows with cache size (that is, with the number of available page frames); this holds for virtualy all modern page replacement algorithms. Let r, f be the maximum capacity, in pages, of the RAM and FLASH caches, respectively. We define the *effective capacity* of a cache at level i to be the number of pages cached at level i that are guaranteed *not* to be cached at any level higher than i at the same time. The effective size of the RAM cache is $e_r = r$. For the FLASH cache, its effective size e_f is equal to the number of pages cached on FLASH that are unique on FLASH (*i.e.*, none of them are cached in RAM at the same time). Consequently, the effective size of the FLASH cache is different for each page flow scheme. For inclusive the effective size of the FLASH cache is $e_f^1 = f - r$, while for exclusive it is $e_f^2 = f$. For lazy, the subset of FLASH pages also cached in RAM varies with the workload; however the following always holds:

$$f - r \leq e_f^3 \leq f$$

Observe how the FLASH cache hit ratio depends on the effective size of the cache, not on its capacity. Consider, *e.g.*, inclusive: when it looks a page up on FLASH, it is only likely to find the requested page in $f-r$ pages, for if the requested page was any of the r pages cached in RAM, no lookup on FLASH would be needed. Therefore, the hit ratio is a function of the page replacement policy, the effective size of the cache, and the workload (the reference pattern). For a given replacement policy Y and a workload W, let the hit ratio be $H = H(Y, W, e_f)$. We have that:

$$H(Y, W, e_f^1) \leq H(Y, W, e_f^3) \leq H(Y, W, e_f^2)$$

Taking into account that $h_f = H \cdot |W|$, we have that:

$$h_f^1 \leq h_f^3 \leq h_f^2$$

for the three algorithms and since $m_r = h_f + m_f$:

$$m_f^1 \geq m_f^3 \geq m_f^2$$

The effective size of the RAM cache is the same for all different schemes; the same applies for the RAM hit ratio.

One can only model the hit ratio for a page replacement policy if the characteristics of the workload are priorly known. Our evaluation shows that for a given policy the hit ratio varies widely across different workloads. This suggests that in a real deployment, where the characteristics of the page reference pattern are not known *a priori*, one cannot statically determine the optimal page flow scheme. In our system we continuously monitor the hit ratio with respect to the effective size of the FLASH cache and accordingly adapt the flow of pages in the memory hierarchy. Specifically, we keep track of FLASH hits and misses and the rate at which pages are dirtied (p_d). Based on the normalized read and write costs for the flash and the hard disks, which are known (or measured) in advance, we periodically evaluate the cost formula for each page flow alternative and adopt the one that minimizes the total cost. In Section 6, we discuss some workload characteristics based on which one can decide the optimal scheme statically and with some confidence.

4. IMPLEMENTATION ISSUES

One important decision is the location of the FLASH cache page directory when the RAM cache is much smaller than the FLASH cache. For r pages cached in RAM, $b \cdot r$ is the size in bytes of the RAM directory, for b bytes per directory entry. Similarly, $b \cdot f$ bytes are needed for the FLASH directory. Let B be the size of a page in bytes, S_r be the size of RAM, and S_f be the size of the FLASH disk; we have that $f = \frac{S_f}{B}$. If the FLASH directory is stored in-memory, $S_r - bf$ bytes are left in main memory for caching. Hence, $m = \frac{S_r - bf}{B+b}$ pages are cached in RAM.

If the entire main memory is used for the RAM cache it fits $\frac{S_r}{B+b}$ pages. Given a replacement policy and a workload the following holds:

$$H\left(Y, W, \frac{S_r}{B+b}\right) \leq H\left(Y, W, \frac{S_r - bf}{B+b}\right)$$

Our experiments show the difference between these two hit ratios being significant, more so as the discrepancy between the RAM and FLASH sizes grows. Thus, it may be desirable to reduce the portion of main memory used for the FLASH page directory. To that end there are two alternatives: (a) use a larger page size for the FLASH cache, or (b) store the FLASH directory (or a part of it) on FLASH instead of RAM.

4.1 Using larger pages for flash

Let B_r and B_f be the RAM and FLASH page sizes respectively; b_r bytes are required for a RAM directory entry and b_f bytes for a FLASH directory one. Each directory entry holds, at the very least and in both cases, the HDD offset of the page (also serving as its identifier), a pointer to the page in the cache (a main memory pointer for a RAM page, or a disk offset for the FLASH cache) and a dirty bit. The replacement policy requires extra bytes for bookkeeping (*e.g.*, a pointer to the next page in an LRU queue) and for the in-memory hash table required for lookups. For the RAM directory some more bookkeeping is required for pinning/unpinning, concurrency control, *etc.*; we will not further ellaborate on b_r as it is not our focus.

If $B_f > B_r$, each FLASH page has $\frac{B_f}{B_r}$ RAM pages; we use the term *block* to refer to such FLASH pages. All I/O between the flash disk and the HDD is in blocks of B_f bytes, while data movement from and to the RAM cache is in pages of B_r bytes. The RAM cache and all in-memory structures use the HDD *offset* of a page as its universal identifier. Thus, the RAM directory uses $\frac{\mathit{offset}}{B_r}$ as the page identifier, while the FLASH directory uses $\frac{\mathit{offset}}{B_f}$ as the identifier for a block stored at *offset* on HDD. Therefore $\log_2 \frac{\mathit{offset}}{B_f}$ bits are required to identify a page in the FLASH directory. By knowing the RAM directory identifier of a page, one can use B_f and B_r to obtain the identifier of the FLASH block where the RAM

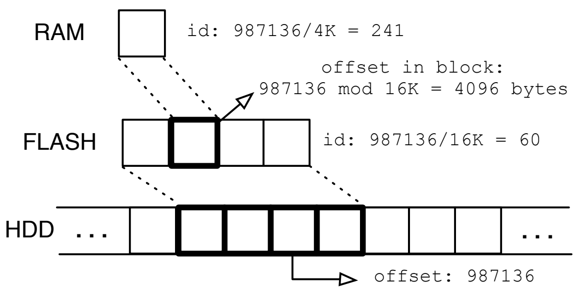


Figure 5: Using larger FLASH pages

page belongs. Let p_r be a RAM page of FLASH block p_f. For each reference to p_r, we look it up in the FLASH directory. If p_f is there, then p_r is located at offset $(p_r B_r \mod B_f)$ within p_f, the offset of which is given by $(p_r B_r \div B_f)$. Otherwise, p_f is read from HDD into FLASH and p_r is computed the same way. FLASH evictions take place with B_f granularity. The case for $B_r = 4$kB and $B_f = 16$kB is shown in Figure 5.

If p_r is evicted from RAM to FLASH but p_f is not cached on FLASH at that time (a case that arises under exclusive and lazy), writing page p_r of block p_f to FLASH is not straightforward. One solution is to first fetch p_f from HDD into FLASH and overwrite its p_r page incurring one additional HDD read. We refer to this technique as overwriting. Note that when fetching the whole block from the HDD, some pages of the block may already be cached in RAM. Therefore, the invariant $\forall t P_{RAM}(t) \bigcap P_{FLASH}(t) = \emptyset$ of the exclusive scheme is not strictly enforced with this technique. Under inclusive this never arises: any page cached in RAM will have its host FLASH block cached on FLASH.

An alternative solution is to assign a block to p_f on FLASH, invalidate all its pages but p_r, and overwrite p_r. If block p_f is later read from HDD, only the invalid pages will be overwritten on FLASH; if it is written to HDD, only the valid pages will be written. We term this technique invalidating. Except for a slight implementation complexity the main drawback of this solution is that a large number of pages in a flash block may become invalid, and, thus, waste space. This is especially true if the reference pattern exhibits poor spatial locality. A solution is for invalid pages not to be stored on FLASH blocks, but only *marked* as invalid in the FLASH directory. The following example shows the impact of each technique on the size of the FLASH directory.

Example. Let $B_r = 4$kB, $B_f = 64$kB and assume a 64-bit offset for HDD and FLASH, and 32-bit main memory addressing; $\log_2 \frac{2^{64}}{64\text{K}} = 48$ bits are required for the block identifier. For a 128GB FLASH cache, there are 2M flash blocks. Thus, a FLASH block requires $\log_2(2\text{M}) = 21$ bits for flash addressing (*i.e.*, the flash disk offset where the page is stored). Also needed are: 1 dirtyness bit for each block; 32 bits for the main-memory pointer in the LRU queue; and 32 bits for the main memory hash index (a pointer to the directory entry) for simple LRU. In total a FLASH directory entry requires at least $48+21+1+32+32 = 134$ bits. Using overwriting the FLASH directory occupies $\frac{2\text{M}\cdot 134}{8\cdot 1\text{M}} = 33.5$MB. Using invalidating, one additionally needs $\frac{64k}{4k} = 16$ bits per FLASH block as validity bits (one per block page) These sum up to 150 bits per directory entry, or 37.5MB. If larger FLASH blocks were not employed at all, *i.e.*, 4kB pages were used for FLASH, the identifier of a page would require 52 bits and there would be 32M block addresses on flash, the representation of each of which would require 25 bits. Thus, $52 + 1 + 25 + 32 + 32 = 142$ bits would be required for each of the 32M pages, or 568MB in total. □

Flash Page Size	overwriting (MBs)	invalidating (MBs)
4K	568	N/A
8K	280	284
16K	138	142
32K	68	72
64K	33.5	37.5
128K	16.5	20.5

Table 1: FLASH directory size

Table 1 shows the directory size for FLASH blocks of various sizes, for a 128GB flash disk. Using larger pages on FLASH saves a lot of main memory, which can be used for caching in RAM to increase the RAM hit ratio. Larger flash pages, however, reduce the paging (and thus caching) granularity, so the flash hit ratio is expected to drop, especially for workloads with poor spatial locality. We further explore this trade-off in Section 5.5. Note that, as studied in [4], writing on flash using a large block size (*e.g.*, 32 or 64KB) increases bandwidth and random write efficiency. Therefore, large flash blocks are not only a way to shrink the FLASH directory and increase RAM hits, but also a way to speed up random writes to flash.

4.2 Storing the page directory on flash

If the amount of main memory is small or the FLASH directory occupies too much memory, it may be preferable that the whole FLASH directory is stored on the FLASH disk (at least for low latency flash disks). Another reason is that, given the non-volatility of flash, if the FLASH directory is persistent, its contents can be preserved between crashes, thereby eliminating warm-up time. In this setup, the directory itself is stored on FLASH and some of its pages are buffered in main memory. Let f' be the number of pages cached on FLASH; hence the size of the directory is $b_f f'$. Then, $f' = \frac{S_f}{B_f + b_f}$ and the size of the directory is $\frac{b_f S_f}{B_f + b_f}$. Assuming that n bytes of the directory are buffered in main memory, with $b_r \leq n \leq b_f f'$, the size of main memory available for caching pages is $S_r - n$ and the number of pages cached is $r' = \frac{S_r - n}{B_r + b_r}$. It is clear that $f' < f$ and $r' > r$, where f, and r are for when the FLASH directory is kept in main memory, as in the previous sections. Therefore, $e'_f = e_f - b_f f' < e_f$ for all three page flow schemes if an external FLASH directory is used. Consequently, $H(Y, W, e'_f) \leq H(Y, W, e_f)$, *i.e.*, the hit ratio for the FLASH cache will drop. The size of RAM available for caching is now greater by $b_f f' - n$; thus, the RAM hit ratio will rise. Given that $S_r \ll S_f$, then, for small values of n, the increase of the RAM hit ratio will be greater than the drop of the FLASH hit ratio. Additionally, as n shrinks the cost of a directory lookup/update grows, as fewer pages of the directory are buffered. The hit ratio for directory pages is given by $H(Y, W_d, n)$, where W_d is the reference pattern for directory operations generated by workload W. If l directory page accesses are required for a lookup, u directory page accesses are required for an update, and the probability of a directory page being dirtied in RAM is p_d^d, we have:

$$L = l((1 - H(Y, W_d, n))F_R + p_d^d F_W)$$
$$U = u((1 - H(Y, W_d, n))F_R + p_d^d F_W)$$

neglecting the cost of directory page lookups/updates served in-memory. Substituting these formulas to the corresponding ones of Sections 3.2, 3.3, and 3.4, one can calculate the expected cost. Of course, the directory page miss ratio $1 - H(Y, W_d, n)$ and probability of a directory page being dirtied p_d^d will have to be monitored or otherwise estimated.

4.3 How much flash? How much RAM?

For the case that no FLASH cache is used, assume h_r RAM hits and m_r RAM misses occur for a workload. Then, the total cost C_0 for this case is $C_0 = m_r(D_R + p_d D_W)$. Using a simulator, one can simulate the cache behavior of a system with varying RAM and FLASH cache sizes (or, even with no FLASH cache). Specifically, by running the simulator for various cache sizes and for the type of workload the system will process, values can be collected for h_r, m_r, h_f, m_f, and p_d. By using the values in the cost formulas, along with the read/write costs of specific flash and magnetic disks, one can determine which combination of cache sizes and hardware devices is the most I/O-efficient for workloads of the given type. Moreover, the price-to-I/O-cost ratio for each case gives the most cost-efficient solution. Alternatively, the 5-minute rule of [7] can be used to determine the optimal memory and flash disk capacities required, assuming prior workload knowledge. Our cost formulas determine the type of flash disk that gives the best price/performance ratio for a type of workload. Naturally, the decision for the size of the main memory and the flash disk is an offline one and optimized for specific workloads. However, the optimal page flow scheme can be decided on-line, on a per-workload basis, by periodically evaluating the cost formulas. Our proposals are also applicable in database systems that employ per-file/relation buffer management: by monitoring the workload for each file and calculating the cost of each different scheme, our model may lead to different relations being buffered using different schemes.

5. EXPERIMENTAL STUDY

Setup. We implemented our algorithms to evaluate their performance under various workloads. Our system consists of a main memory buffer pool for caching in RAM, a page cache on a flash disk and a magnetic disk for persistent storage. Each page is identified by its disk offset on persistent storage. The system was implemented in C++ and we used an Intel Pentium 4 box clocking at 2.26GHz with 1.5GB of physical memory for our experiments. The Operating System was Debian GNU/Linux with the 2.6.26 kernel. The system had two magnetic disks and one flash disk. Our system and the OS ran from one of the magnetic disks and the other magnetic disk (referred to as the HDD hereafter) was used to store the dataset. The HDD was a 300GB Maxtor 6L300R0 with 16MB of cache. The flash disk was a Samsung MCAQE32G5APP, an MLC NAND flash disk with a capacity of 32GB. Both disks were using the IDE interface. To eliminate OS caching we used both storage media as raw devices: the OS did not cache data pages, pages were never double buffered and our system had absolute control of physical I/O. Read and write costs were estimated as in [11].

Flash Disks. The flash disk we used has a poor write performance and is unsuitable as a cache. Therefore, we considered other flash disks, more suitable for caching, by using their read and write costs in the equations of Section 3. For the I/O costs of these disks we used published benchmarks and documents about their efficiency in IOPS ([6, 8, 16]). We present the read/write costs of all flash disks considered in Table 2. Observe that random read performance varies up to two orders of magnitude among disks, while random write performance varies as much as four orders of magnitude.

Disk Model	4kB Read IOPS	4kB Write IOPS	\$/GB
Samsung	2500	21	1.6
Intel X25-M	12000	592	8.1
Intel X25-E	35000	3300	20
Fusion ioDrive	102000	101000	30

Table 2: Flash disks considered

Workloads. We used three different workloads. The first, referred to as IRP, follows an independent reference pattern where all pages in the dataset have the same probability of reference, *i.e.*, a random reference pattern. We varied the probability of a page being read or written to and created workloads with varying dirtyness ratios. For the second workload, referred to as TPC-C, we ran the TPC-C benchmark on the PostgreSQL database and collected a trace of all page references, which we then translated into HDD offsets. We did the same for the TPC-H benchmark to obtain the third workload. The results of this section are the execution of these traces by our system after varying its parameters. In all cases, the main memory page size was set to 4kB. For all experiments we used LRU as the page replacement policy (for both the RAM and the FLASH caches).

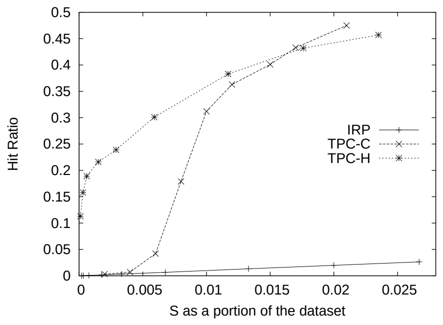
5.1 Impact of Cache Size on Hit Ratio

We measured the effect of the size of a page cache on its hit ratio, *i.e.*, how $H(Y, W, S)$ varies with S, the effective size of the cache, under LRU. We ran the three workloads for different page cache sizes; we report the hit ratio in Figure 6(a). The x-axis is S shown as a percentage of the size of the whole dataset. In all cases the hit ratio grows with the size of the cache, as discussed in Section 3.5. The growth rate varies widely with the workload: it is linear for IRP and non-linear for TPC-C and TPC-H. This is expected since both TPC-C and TPC-H have a working set, albeit of different size, while IRP does not. But the most important observation is that, apart from H growing with S, one cannot make assumptions that hold for all workloads.

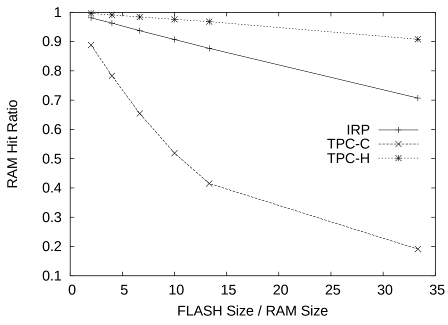
5.2 Impact of Flash Cache Size on RAM Hit Ratio

In our system, the directory for the FLASH cache is stored in main memory. As discussed in Section 4, as the size of the flash cache grows, the available main memory for the RAM cache shrinks and therefore the RAM hit ratio is expected to drop. We measured this effect by growing the size of the FLASH cache (and thus the FLASH directory) while keeping the size of the RAM cache fixed, and measuring the RAM hit ratio H. In addition, we ran the same workloads with no FLASH cache (and thus all main memory available for the RAM cache) and measured the RAM hit ratio H'. In Figure 6(b) we report $\frac{H}{H'}$ for different sizes of FLASH cache. As evident, there is a drop in the hit ratio for all workloads. This drop is linear for IRP as it has no working set, and for TPC-H as its working set fits in RAM in all cases. For TPC-C the working set fits in main memory for small FLASH sizes, but for larger ones it does not; thus, the hit ratio drops very quickly, giving a curve that is the inverse of the curve of Figure 6(a) for TPC-C. In all cases, the main memory occupied by the FLASH index has a big impact on the RAM hit ratio.

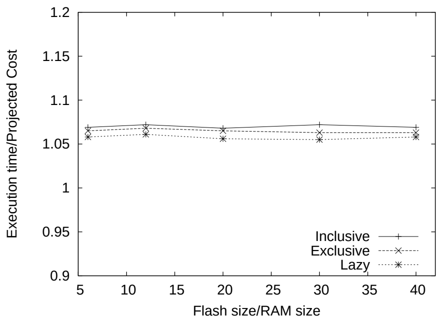
5.3 Validation of the Cost Formulas



(a) $H(Y, W, S)$ as a function of S.



(b) $\frac{H}{H'}$ for different sizes of FLASH cache.



(c) Validation of the cost formulas.

Figure 6: Impact of size on hit ratio, impact of FLASH size on RAM hit ratio and validation of cost formulas

We then went on to empirically verify the validity of the cost formulas of Section 3. We executed a synthetic IRP workload using the Samsung disk and measured the running time for each page flow scheme. We also used the cost formulas of Section 3 with the I/O cost metrics for the particular flash disk and HDD to calculate the total cost of each scheme. We plot the ratio of the execution time for each physical run and the cost projected by the formulas *for that scheme* in Figure 6(c). Observe that the ratio of the real cost over the projected one remains constant for all FLASH cache sizes. Also, this ratio remains the same across page flow schemes, indicating the consistency of the cost formulas. The ratio being 6% to 8% higher than 1 is due to our cost formulas not taking into account the warm-up time for the caches. Our formulas assume that each RAM miss results in a RAM eviction (and thus either a FLASH hit or a FLASH miss), which does not hold until after the RAM cache becomes full. The same applies for the warm-up time of the FLASH cache. Although one can adapt the formulas to account for this cost, as one can approximate after how many references each cache becomes full, we chose not to do so in the interest of simplicity; moreover, this cost is negligible for workloads of interest. Another caveat arises when using our formulas for very small datasets, in which the on-disk caches of the FLASH disk and the HDD affect the disk read/write costs. For all real-world workloads, however, our formulas were quite accurate in their cost estimation. As argued in [4], not all flash writes incur the same cost, just as is the case for magnetic disk writes. In this work we are interested in the cost of all writes throughout the workload, *i.e.*, in the average cost of random writes for blocks of specific size. This cost can be accurately approximated by performing random writes throughout the whole medium and taking the average time (as suggested in [11]).

5.4 Comparison of Page Flow Schemes

We compare the three page flow schemes across the different workloads.

5.4.1 Flash Hit Ratio

We first measured the FLASH hit ratio for each scheme and workload. We ran the experiments for different RAM and FLASH sizes obtaining similar results; due to space limitations we only report in Figure 7 the results for the FLASH cache being 6 times the size of the RAM cache. All hit ratios for each workload are normalized by the hit ratio of inclusive. As explained in Section 3.5, exclusive has the highest hit ratio for all workloads, while inclusive has the lowest. The hit ratio for lazy varies between the two. However, as we will see in the sequence, the highest hit ratio for exclusive does not always result in a lower I/O cost.

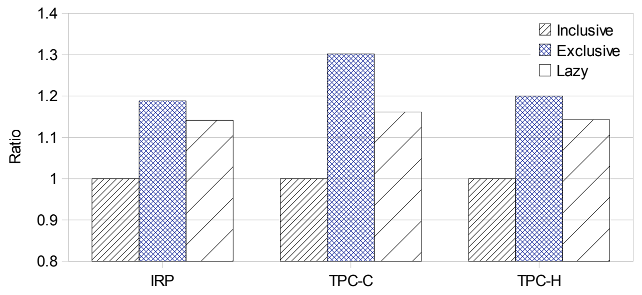


Figure 7: Flash hit ratios per scheme.

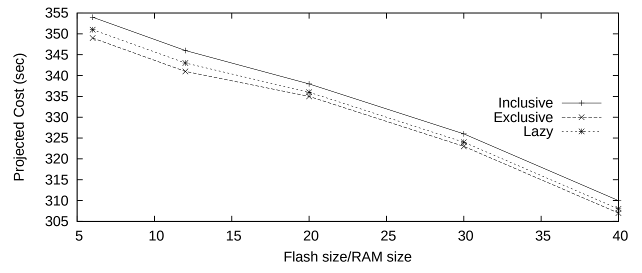


Figure 8: Total cost for TPC-H with Fusion ioDrive.

5.4.2 Total I/O cost

We ran TPC-H and TPC-C for a varying FLASH cache size and a fixed RAM size. We plotted the total I/O cost as calculated using the formulas of Section 3 for different flash disks. In the first experiment, we ran TPC-H with the FLASH cache size varying from 5 to 40 times the size of the RAM cache. The projected I/O cost of the FusionIO ioDrive is shown in Figure 8. In this case, the lazy scheme performs better than the other two for all FLASH cache sizes; the following experiments will reveal that this is not always the case. Observe also that increasing the size of the FLASH cache can benefit performance by a significant factor.

We then ran TPC-C for the same FLASH cache sizes as before and calculated the total cost based on the cost metrics of the Samsung flash disk; the results are shown in Figure 9(a). The exclusive algorithm is totally unsuitable in this case due to the disproportionally high write cost of the Samsung disk (as for each RAM eviction exclusive pays the cost of a flash write). As for inclusive and lazy, observe that while their cost is similar for large FLASH sizes, there is a big performance gap for small FLASH sizes (or, big RAM sizes).

We repeated the cost calculations for TPC-C, but used the I/O costs of the Intel X25-E flash disk; the results are shown in Figure 9(b). When the FLASH cache is less than

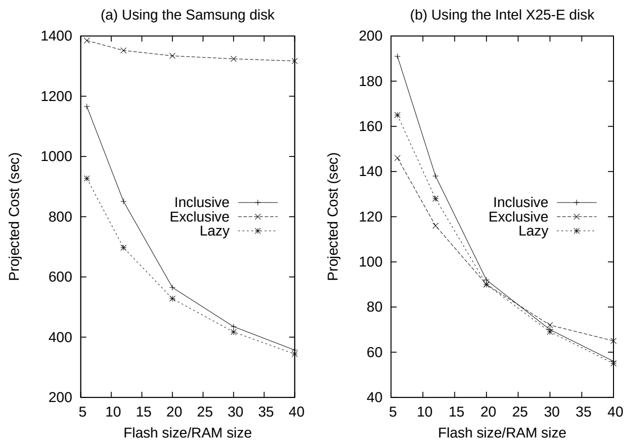


Figure 9: Cost for TPC-C under for each scheme.

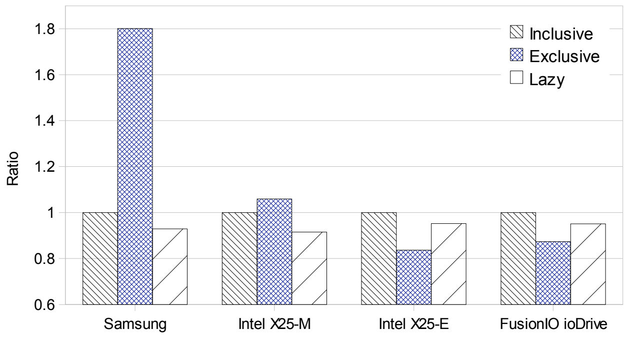


Figure 10: Total cost for TPC-C per flash disk.

15 times the size of the RAM cache, exclusive is the most efficient page flow scheme with a total I/O cost that is up to 30% lower than the I/O cost of inclusive and 14% lower than the I/O cost of lazy. On the other hand, for a FLASH cache size more than 35 times that of the RAM cache, lazy is the optimal scheme with an I/O cost that is 16% lower than the I/O cost of exclusive. Therefore, even for the same FLASH disk and workload the optimal scheme changes with the ratio of the FLASH cache size over the RAM cache size.

Next, we kept fixed sizes for the FLASH and RAM caches and ran the TPC-C workload using each proposed scheme and calculated the total I/O cost for all four disks; the results are shown in Figure 10. The optimal algorithm is different for each disk. Note that lazy is optimal for the two MLC disks (Samsung and Intel X25-M), while exclusive is optimal for higher-performance SLC devices (Intel X25-E and FusionIO). The graph confirms our hypothesis that no scheme is universally optimal across all workloads and flash disks.

5.4.3 Directory Operations

We have so far assumed that the page directory for the FLASH cache is maintained in main memory. Therefore, in our cost calculations we have not included directory maintenance costs. In Figure 11 we show the total number of directory operations, *i.e.*, lookups and updates, for each page flow scheme for the TPC-C workload and for different sizes of FLASH cache. The inclusive scheme is the most efficient, as it requires fewer directory operations than both lazy or exclusive. If the FLASH directory was kept on the flash disk, directory maintenance operations would affect the total I/O cost and should be considered in the cost formulas. For all schemes, as the FLASH cache size grows, so does the FLASH hit ratio and, thus, the number of directory operations required for each scheme drops: a FLASH hit requires fewer operations than a FLASH miss. Due to lack of space we do not present more results on the cost of directory operations. The general conclusion is that, similarly to the I/O cost, the cost of directory maintenance varies widely across different workloads, schemes, and flash disks.

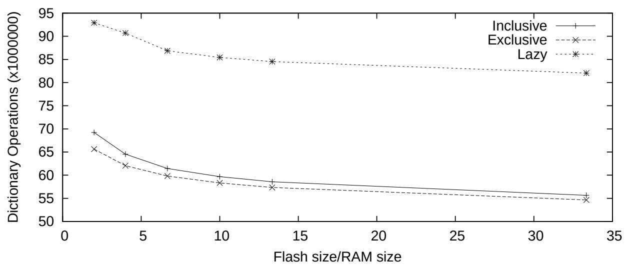


Figure 11: Total directory operations for TPC-C.

5.5 Impact of Flash Block Size

We then investigated how the size of the FLASH block affects performance. In all cases the RAM page size was set to 4kB. We first used a flash block size varying from 4kB to 128kB. For each block size we ran the TPC-H workload and measured the hit ratio for FLASH and the total number of HDD reads, using the overwriting technique of Section 4.1: if a page is evicted from RAM and its corresponding block is not on FLASH, then the whole block is brought from HDD to FLASH. We ran TPC-H using the inclusive and lazy schemes. In Figure 12 we show for each scheme the FLASH hit ratio (top graph) and the number of HDD reads (bottom graph). Under inclusive, before a page is brought into RAM its flash block is written to the FLASH cache. Subsequent accesses to the pages of that block will be served from FLASH. Thus, the hit ratio for inclusive increases as the block size grows and overwriting acts as a prefetching mechanism, greatly affected by locality of reference. Under lazy, a block is written to flash when any RAM page that belongs to that block is evicted from RAM for the first time. Even for workloads with a high degree of locality, pages of the same flash block will have most likely been read into RAM before one of them is evicted to FLASH. Therefore, locality does not affect lazy as much (at least for small block sizes). As the block size grows, so does the granularity at which the replacement policy tracks the reference pattern through access recency (or frequency). Therefore, the hit ratio drops (for inclusive this effect is cancelled by the effect of prefetching). As the bottom graph shows lazy performs about twice as many HDD reads as inclusive. This is not only due to its lower hit ratio: when a RAM victim is written to FLASH, the block it belongs to needs to be read from HDD if it is not cached on FLASH. Conversely, for inclusive, the first invariant guarantees that the block the page belongs to is on FLASH.

Next, we experimented with both overwriting and invalidating to gauge their performance under both TPC-C and TPC-H as we varied the flash block size from 4kB to 128kB; we used the lazy scheme in all cases. Let $h_{\text{overwriting}}$ and $h_{\text{invalidating}}$ be the hit ratios for overwriting and invalidating, respectively. In the top graph of Figure 13 we report the ratio $\lambda = \frac{h_{\text{overwriting}}}{h_{\text{invalidating}}}$ for the two workloads; in the bottom

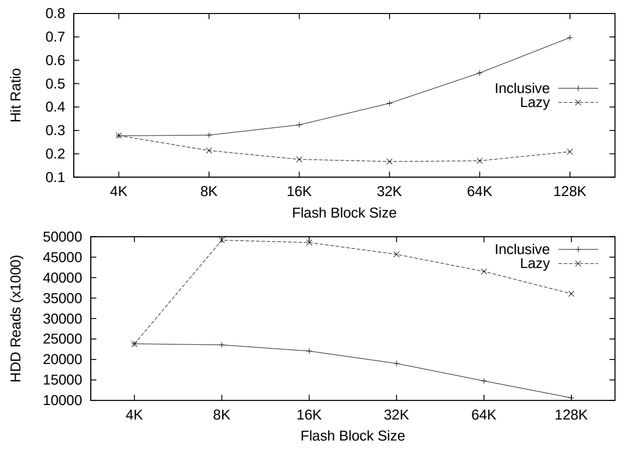


Figure 12: Impact of block size under overwriting.

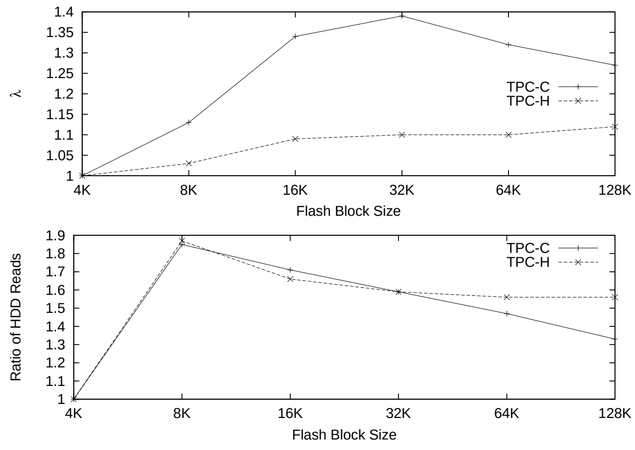


Figure 13: Comparison of overwriting and invalidating.

graph we show the corresponding ratio of HDD reads. For both workloads, overwriting had a higher hit ratio than invalidating, due to the prefetching effect described earlier. This effect was more evident in TPC-H as it exhibits a higher degree of locality than TPC-C. As explained earlier, under the lazy scheme the hit ratio for both overwriting and invalidating was less than the hit ratio for more fine grained replacement (*i.e.*, for 4kB blocks). For HDD operations, overwriting resulted in more HDD reads than invalidating for both workloads (ranging from twice to 1.3 times as many HDD reads). For all workloads with locality of reference, overwriting is expected to give a higher hit ratio than invalidating at the cost of extra HDD read operations. The optimal choice depends on the read efficiency of the flash disk and the HDD.

5.6 Caching Only Clean Pages

We evaluated the effect of caching dirty pages in the FLASH cache. Recall from Section 3 that, for the lazy scheme, one may apply any criterion to decide if a RAM victim page will be cached on FLASH or not. Dirty pages cached on FLASH are more likely to cause updates on the flash disk. Thus, if the flash disk is not efficient at random writes, it makes sense to restrict FLASH caching to clean pages only. Other criteria may be used as well, *e.g.*, the frequency of writes on a page, but we do not explore these further due to lack of space. We used IRP workloads with different dirtyness ratios, *i.e.*, the probability of a page being dirtied on each next reference. Each workload was executed using the lazy scheme twice: once caching all RAM victims on FLASH and once caching only the clean ones. We used the Samsung flash disk (which is inefficient at random writes) and measured the hit ratio for the FLASH cache and the total execution time for varying dirtyness ratios. Hit ratios are shown in the left graph of Figure 14 and execution times are shown in the right graph. The hit ratio drops when only clean pages are cached, as some of the hot dirty pages are evicted to HDD. For small dirtyness ratios, the drop in the hit ratio is gradual, as the hottest of the dirty pages fit in RAM. For a dirtyness ratio greater than 0.7, the hit ratio drops substantially. On the other hand, the execution time is much less when caching only the clean pages, due to the write inefficiency of the flash disk we used. Observe that for dirtyness ratios between 0.1 and 0.7 the running time remains the same when caching only clean pages, *i.e.*, the increased miss ratio is counterbalanced by the time saved by avoiding flash writes. For greater dirtyness ratios the hit ratio drop results in about a 10% increase in execution time.

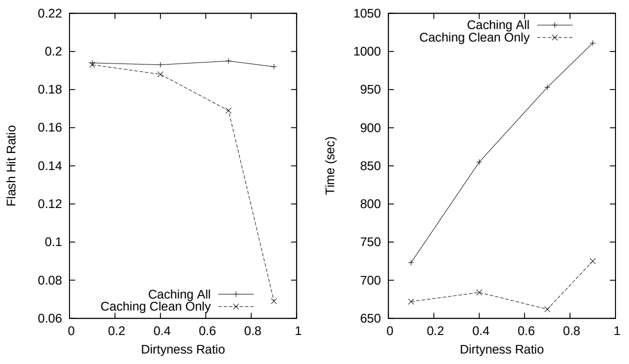


Figure 14: Effect of caching only clean pages.

6. DISCUSSION

Choosing the optimal scheme. Our experimental evaluation shows that the I/O cost of a workload depends heavily on: (*a*) the workload itself, (*b*) the page flow scheme, and (*c*) the I/O costs of the flash disk used. Therefore, one cannot decide with confidence the optimal scheme *a priori* without evaluating our cost formulas. Given the workload and the flash disk, we can hypothesize about the optimal scheme. For instance, exclusive performs one flash write for each RAM miss, whether the victim page is dirty or not; inclusive and lazy do so only for dirty pages. Thus, for write-intensive workloads, exclusive is likely to be more expensive, more so if the flash disk is not write-efficient. In such a case, a large number of flash writes can be avoided if only clean pages are cached on FLASH, as shown in Section 5. If the RAM cache is not a small percentage of the FLASH cache, then exclusive is likely the best option: no page will be cached on both caches, saving space on FLASH. Using similar arguments it is sometimes possible to make the optimal choice if the characteristics of the workload are well-known.

The experimental results also verify our hypothesis that hit ratios alone cannot fully describe the system's I/O efficiency. For instance, Figures 7 and 10 show that although exclusive has the highest hit ratio for TPC-C, it is not the optimal scheme across all flash disks *w.r.t.* to the total I/O cost. This holds for all workloads we have tested. Moreover, as Figure 9(b) shows, even for a specific workload and flash disk, the optimal scheme changes for different FLASH or RAM cache sizes. Note also that inclusive appears to always be less I/O-efficient than lazy, if the same page size is used for RAM and FLASH. However, this changes radically for different page sizes, or directory maintenance costs (*i.e.*, when the FLASH directory is stored on the flash disk).

Flash writes. We have so far assumed that pages are written to the flash disk in a random fashion. In [14], the flash disk is split in a write cache and a read cache, with the write cache employing a log-structured filesystem to speed up random writes. Such techniques are complementary to ours, especially for flash disks with poor random write performance; for high-end ones random writes are as efficient as sequential ones, *i.e.*, they are converted to sequential ones by the disk controller. In [13], the authors use large asynchronous sequential writes instead of synchronous random ones. Flash writes can be asynchronous in our case too: pages to be cached on flash are only marked as such in main memory and moved to flash asynchronously. Such writes can also be performed sequentially in large chunks, as in ZFS; then, one would need to adjust the write costs. This is analogous to using a larger flash block, as described in Section 4.1. In our case, large flash blocks are fetched from disk when their first page is accessed (*e.g.*, for inclusive), effectively prefetching all other pages of the block. ZFS only writes to flash pages from main memory (evictees) and thus this effect is absent. As shown in Section 5.5, prefetching can greatly enhance performance – at least for database workloads. For high-performance flash disks with high random write throughput and low latency, *e.g.*, the FusionIO ioDrive, large sequential writes will not make much difference: sequential and random writes have almost the same throughput. Additionally, such devices have comparable read/write latencies. Thus, flash writes can be synchronous at RAM eviction time, without bogging the system down. Our work focuses on deciding the size and the contents of the RAM and FLASH caches, while [13] mainly focuses on implementation efficiency. Therefore, we believe that our approach is complementary to the ZFS approach. That said, we feel that a system should primarily decide on the contents of the flash cache, and secondarily on implementation principles.

Remarks. Techniques that speed up random writes on flash disks, *e.g.*, IPL [12], are complementary to our work and important for caching efficiency. An interesting question is what on-flash data structure or filesystem serves caching needs best, especially if the page directory and the replacement algorithm are external memory ones. The answer depends on how pages are written to flash and how they are replaced upon eviction; we do not study this further here. As for metadata persistency, standard techniques (*e.g.*, write-ahead logging) can be employed. Given the non-volatility of flash memory, one can do the same for the flash page directory. That way, after a system failure, the flash cache will warm up instantly, speeding up system recovery.

7. CONCLUSIONS AND FUTURE WORK

Low read latencies, combined with constantly growing capacities and dropping cost, make flash disks ideal for caching data between the main memory and the hard disk. We studied the salient aspects of a system that uses a flash disk as a cache. We presented: (a) three invariants for the set of pages cached on flash, (b) algorithmic schemes implementing the invariants, and (c) an I/O-based cost model for the performance of the algorithms. We studied alternative implementation decisions, such as the optimal page size for a flash cache and how the size of the page directory impacts performance. We implemented our proposals and conducted an extensive experimental study of flash-resident caches. Our results show that there is no universally optimal design for a flash cache, across all workloads and flash disks. Therefore, to make informed design decisions, analytical tools such as the ones we provide are necessary.

Our analysis and results are independent of the replacement policy used in the flash cache. In the future, we plan to investigate the correlation between the replacement policy of the flash cache and system performance. Finally, we plan to study how we can improve the efficiency of an extended cache when it is aware of the database operation (*e.g.*, sorting, join evaluation, *etc.*) currently being executed.

8. REFERENCES

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for ssd performance. In *ATC'08: USENIX 2008 Annual Technical*, pages 57–70. USENIX Association, 2008.

[2] L. A. Belady, R. A. Nelson, and G. S. Shedler. An anomaly in space-time characteristics of certain programs running in a paging machine. *Commun. ACM*, 12(6):349–353, 1969.

[3] A. Birrell *et al.* A design for high-performance flash disks. *SIGOPS Oper. Syst. Rev.*, 41(2), 2007.

[4] L. Bouganim, B. P. Jonsson, and P. Bonnet. uFLIP: Understanding flash io patterns. *CIDR*, 2009.

[5] T.-S. Chung *et al.* System software for flash memory: A survey. In *EUC*, 2006.

[6] FusionIO - the power of 1000 hard drives in the palm of your hand. TGDaily. `http://tgdaily.com`.

[7] G. Graefe. The five-minute rule twenty years later, and how flash memory chenges the rules. *DAMON*, 2007.

[8] Intel X25-M SSD: Intel Delivers One of the World's Fastest Drives. AnandTech. `http://anandtech.com`.

[9] H. Kim and S. Ahn. BPLRU: a buffer management scheme for improving random writes in flash storage. In *FAST 2008, Usenix Association*, 2008.

[10] J. Kim *et al.* A space-efficient flash translation layer for compactflash systems. *Transactions on Consumer Electronics.*, 2002.

[11] I. Koltsidas and S. D. Viglas. Flashing up the storage layer. *Proc. VLDB Endow.*, 1(1):514–525, 2008.

[12] S.-W. Lee and B. Moon. Design of flash-based DBMS: An in-page logging approach. In *SIGMOD*, 2007.

[13] A. Leventhal. Flash storage memory. *Commun. ACM*, 51(7):47–51, 2008.

[14] D. Narayanan *et al.* Migrating enterprise storage to ssds: analysis of tradeoffs. *Microsoft Technical Report MSR-TR-2008-169*, 2008.

[15] S. Nath and P. B. Gibbons. Online maintenance of very large random samples on flash storage. *Proc. VLDB Endow.*, 1(1):970–983, 2008.

[16] OCZ Core Series 64GB SATA II 2.5" Solid State Drive. TigerDirect.com. `http://tigerdirect.com`.

[17] Sun Microsystems. The Solaris ZFS filesystem.

[18] S. yeong Park *et al.* CFLRU: a replacement algorithm for flash memory. In *CASES '06*. ACM, 2006.