Hibachi: A Cooperative Hybrid Cache with NVRAM and DRAM for Storage Arrays

Ziqi Fan, Fenggang Wu, Dongchul Park[§], Jim Diehl, Doug Voigt[†], and David H.C. Du
University of Minnesota–Twin Cities, [§]Intel Corporation, [†]Hewlett Packard Enterprise
Email: fanxx234@umn.edu, {fenggang, park, jdiehl}@cs.umn.edu, doug.voigt@hpe.com, du@umn.edu

***Abstract*—Adopting newer non-volatile memory (NVRAM) technologies as write buffers in slower storage arrays is a promising approach to improve write performance. However, due to DRAM's merits, including shorter access latency and lower cost, it is still desirable to incorporate DRAM as a read cache along with NVRAM. Although numerous cache policies have been proposed, most are either targeted at main memory buffer caches, or manage NVRAM as write buffers and separately manage DRAM as read caches. To the best of our knowledge, cooperative hybrid volatile and non-volatile memory buffer cache policies specifically designed for storage systems using newer NVRAM technologies have not been well studied.**

This paper, based on our elaborate study of storage server block I/O traces, proposes a novel cooperative HybrId NVRAM and DRAM Buffer cACHe polIcy for storage arrays, named *Hibachi*. *Hibachi* treats read cache hits and write cache hits differently to maximize cache hit rates and judiciously adjusts the clean and the dirty cache sizes to capture workloads' tendencies. In addition, it converts random writes to sequential writes for high disk write throughput and further exploits storage server I/O workload characteristics to improve read performance. We evaluate *Hibachi* on real disk arrays as well as our simulator. The results show that *Hibachi* outperforms existing work in both read and write performance.

I. Introduction

Due to the rapidly evolving non-volatile memory (NVRAM) technologies such as 3D XPoint [1], NVDIMM [2], and STT-MRAM [3], hybrid memory systems that utilize both NVRAM and DRAM technologies have become promising alternatives to DRAM-only memory systems [4].

Storage systems can also benefit from these new hybrid memory systems. As an example, Figure 1 shows the storage systems can be a storage server, storage controller, or any part of a data storage system that contains a hybrid memory cache. Storage systems typically rely on DRAM as a read cache due to its short access latency. To hide lengthy I/O times, reduce write traffic to slower disk storage, and avoid data loss, storage system write buffers can use NVRAM.

Buffer cache policies have been studied for decades. They mostly examine main memory buffer cache strategies, e.g., LRU [5], ARC [6], and CLOCK-DWF [7]. Multilevel buffer cache studies focus on read performance, e.g., MQ [8] and Karma [9], or separately managed NVRAM as write buffers and DRAM as read caches, e.g., NetApp ONTAP caching software [10]. However, cooperative hybrid buffer cache policies that combine newer NVRAM technologies with DRAM targeted specifically for storage systems have not been well studied.

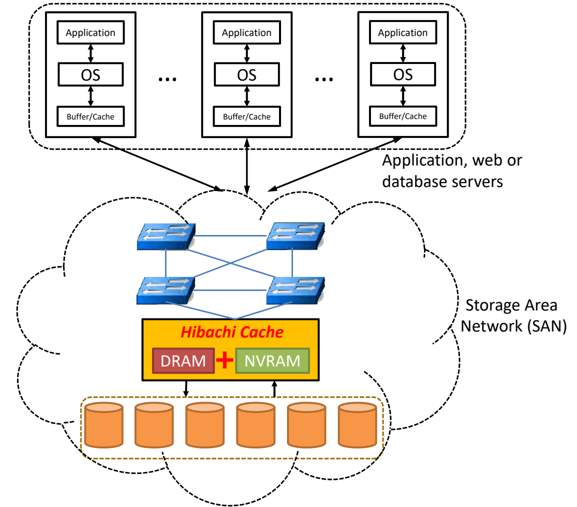
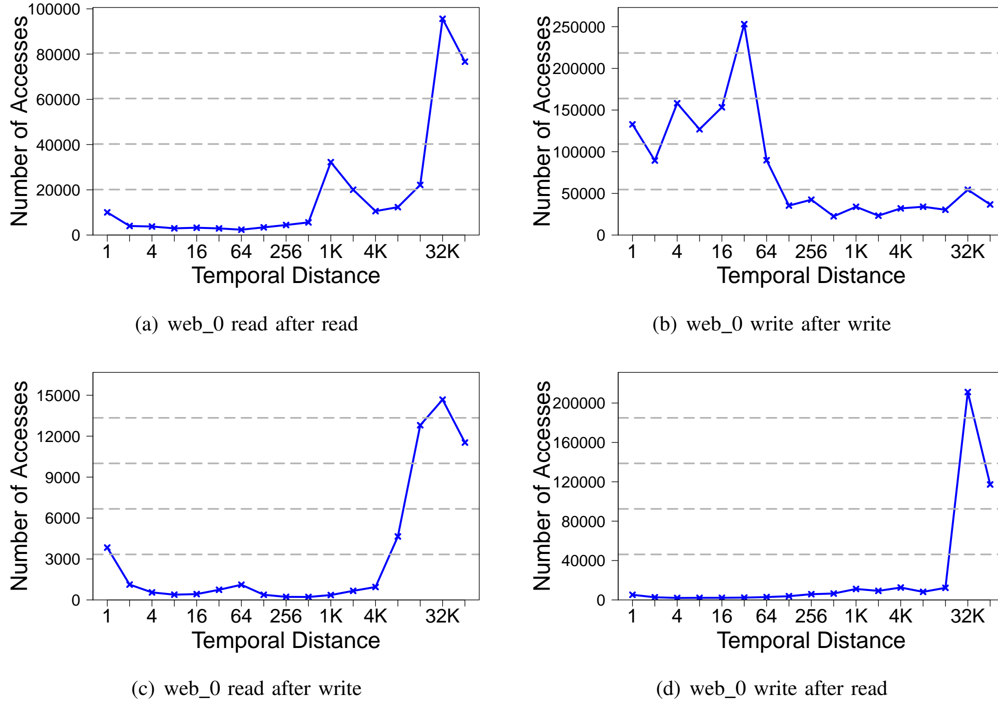


Fig. 1. **Overall System Architecture**

To gain hybrid buffer cache design insights, we make an elaborate study of storage system I/O workloads. These storage system level I/O workloads are very different from server-side I/O workloads due to server-side buffer/cache effects [11]. We evaluate and analyze the impact of different NVRAM sizes, access latencies, and cache design choices on storage performance. Based on these key observations, we propose a novel cooperative HybrId NVRAM and DRAM Buffer cACHe polIcy for storage disk arrays, named *Hibachi*. *Hibachi* transcends conventional buffer cache policies by 1) distinguishing read cache hits from write cache hits to improve both read and write hit rates; 2) learning workload tendencies to adjust the page caching priorities dynamically to shorten page access latencies; 3) regrouping cached dirty pages to transform random writes to sequential writes to maximize I/O throughput; and 4) using accurate and low-overhead page reuse prediction metrics customized for storage system workloads.

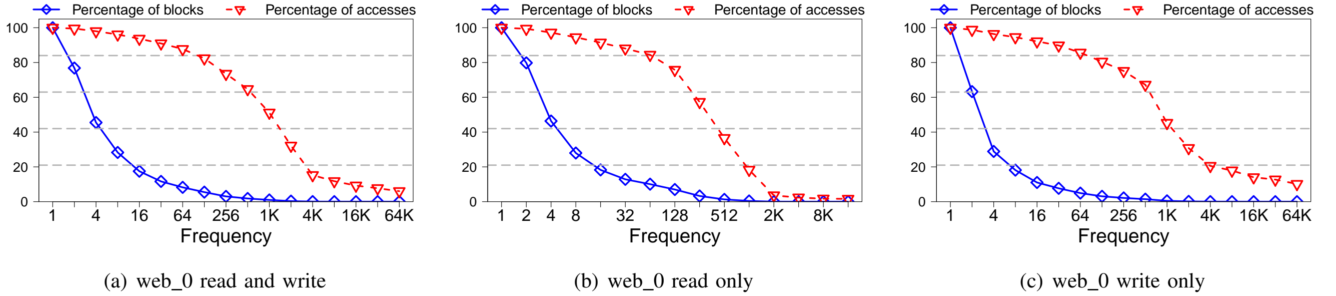
We evaluate *Hibachi* with real block I/O traces [12], [13] on both simulators and disk arrays. Compared to traditional buffer cache policies, *Hibachi* substantially improves both read and write performance under various storage server I/O workloads: up to a 4× read hit rate improvement, up to an 8.4% write hit rate improvement, and up to a 10× write throughput improvement. We believe our work shows the potential of designing better storage system hybrid buffer cache policies



(a) web_0 read after read (b) web_0 write after write

(c) web_0 read after write (d) web_0 write after read

Fig. 2. Temporal distance histograms of a storage server I/O workload. Four figures represent temporal distance in terms of a read request after the same read request, write after write, read after write, and write after read.



(a) web_0 read and write (b) web_0 read only (c) web_0 write only

Fig. 3. Access and block distributions for various frequencies. Three figures show frequency in terms of combined read and write requests, read requests only, and write requests only. For a given frequency, the blocks curve shows the percentage of the total number of blocks that are accessed at least that many times, and the accesses curve shows the percentage of the total number of accesses that are to blocks accessed at least that many times.

and motivates future work in this area.

The rest of the paper is organized as follows. Section II provides our observations on storage system workload studies. Section III discusses the impact of NVRAM on cache performance and cache design choices. Section IV gives a detailed description of our proposed cooperative hybrid buffer cache policy followed by an evaluation in Section V. In Section VI, we present some related work about NVRAM and caching policies. Finally, Section VII concludes our work.

II. Storage System Workload Properties

Storage system workloads are very different from server-side workloads. Zhou et al. [8] claimed temporal locality (recency) is notably poor in storage level (second level) workloads since the server-side buffer caches filter out the majority of recent same data requests. At the storage system level, frequency, which is the total number of times the same data is accessed over a longer period, can more accurately predict a page's reuse probability.

This paper simultaneously considers both the read cache and write buffer by expanding prior workload characterization work to examine both read access patterns and write access patterns instead of only focusing on read patterns. The temporal distance is measured as the number of unique page or block requests between one request and the same request in the future. We examine temporal distances in terms of a read request after the same read request, a write after write, a read after write, and a write after read of the same data. Frequency, a different measurement, shows whether a majority of accesses are concentrated within a small portion of blocks (which are ideal pages to buffer) in terms of combined read and write requests, read requests only, and write requests only. The

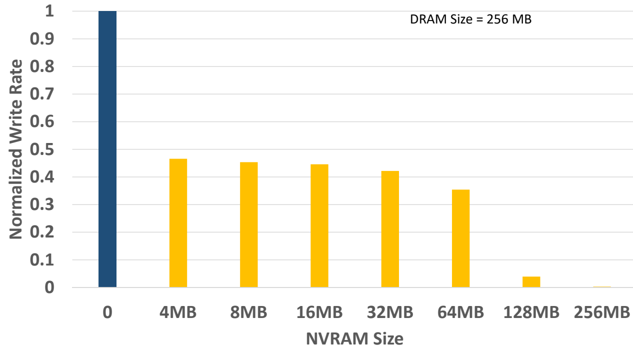


Fig. 4. **NVRAM impact on storage write traffic.**

detailed temporal distance and frequency calculation method can be found in Zhou's work [8].

Though we analyze many traces, we choose one representative, web_0, from the MSR Traces to demonstrate our findings (other traces show similar patterns). web_0 is a one-week block I/O trace captured from a Microsoft enterprise web server [12], [13]. Figure 2 presents our workload analysis results for temporal distance. Figure 3 presents our workload analysis results for frequency. The majority of read after read accesses (Figure 2(a)) have a large distance, which means poor temporal locality or recency. However, in contrast to previous work that used different traces, we found that a large number of write after write temporal distances (Figure 2(b)) are short, which shows strong recency (likely due to the server-side forced synchronization). For read after write (Figure 2(c)) and write after read (Figure 2(d)), recency properties are no better than read after read. Note that the total number of read after write requests is very small compared to the other types of requests.

For frequency of mixed read and write (Figure 3(a)), read only (Figure 3(b)), and write only (Figure 3(c)), the wide areas between the two percentage curves indicate that most accesses are concentrated within a few blocks. This implies both read and write requests show good frequency and can be optimized with an appropriate caching policy.

III. Insight and Discussion

Because of their non-volatility, NVRAM write buffers can minimize write traffic by reducing and delaying dirty page synchronization. To demonstrate their effectiveness of write traffic reduction compared to a DRAM-only buffer cache, we conduct some simple experiments. As a baseline, we consider a fixed-size, DRAM-only buffer cache that periodically flushes dirty pages to persistent storage (disk) similar to Linux's "pdflush" threads functionality. Next, we add different amounts of NVRAM to work with the DRAM. For simplicity and a fair comparison, both NVRAM and DRAM adopt the LRU replacement policy. NVRAM caches dirty pages which are evicted only when the NVRAM's capacity is reached. Figure 4 shows one example of our experiments with the MSR rsrch_0 trace. Compared to the DRAM-only buffer cache, adding NVRAM that is as little as 1.56% of the amount of DRAM can reduce write traffic by 55.3%. This huge impact is mostly caused by strong write request recency in the storage server I/O workload.

Currently, the access latency of most NVRAM technologies is several times longer than DRAM's. When NVRAM is used to buffer dirty pages, a read hit on a dirty page needs to access NVRAM. This is slower than reading the page from DRAM. This leads to a question about the trade-off between page migration and page replication policies. If we choose a page replication mechanism, a read hit in NVRAM could trigger a page copy from NVRAM to DRAM. We could gain read performance if the page is later accessed multiple times. But if not, one page of cache space in DRAM is wasted. On the other hand, if we choose page migration (which means a single page can never exist in both NVRAM and DRAM), no space is wasted, but there is a risk of longer average read access latency if there are updates between reads that bring the page back to NVRAM. We execute another experiment to compare page migration vs. page replication performance. For storage server I/O workloads, we found that page replication causes a lower cache hit ratio and does not improve read performance. As Section V-B1 will show, this result is because read hits occur mostly in DRAM and rarely in NVRAM. Therefore, we choose page migration for our storage system buffer cache design.

IV. *Hibachi*: A Hybrid Buffer Cache

Due to the different performance characteristics of NVRAM and DRAM, *Hibachi* utilizes DRAM as a read cache for clean pages and NVRAM mostly as a write buffer for dirty pages. However, our four main design factors (described in Section IV-B through IV-E) significantly improve *Hibachi*.

A. Architecture

Figure 5 presents our proposed *Hibachi* architecture. *Hibachi* largely consists of two real caches and two ghost caches. The clean cache manages all the clean pages in DRAM and NVRAM (possibly), and the dirty cache keeps track of all the dirty pages in NVRAM. The ratio between the clean cache and the dirty cache capacity (i.e., black arrow in the figure) dynamically adjusts according to the current workload's tendency assisted by the two ghost caches. For example, some NVRAM space can be borrowed as an extension of the clean cache to cache hot clean pages. Unlike the real caches, the clean ghost cache and the dirty ghost cache only maintain metadata (i.e., page numbers) of recently evicted pages, not real data. A recency-based policy such as LRU (Least Recently Used) manages the dirty cache and a frequency-based cache policy such as LFU-Aging (Least Frequently Used with Aging) [14] manages the clean cache. The following section (Section IV-B) describes the rationale for adopting these two different policies. Each data page in both real caches maintains a counter. Clean page counters record their access frequencies and dirty page counters are used for migration purposes. To further improve write performance, *Hibachi* converts random

disk writes to sequential disk writes to exploit HDD's superior sequential access performance. To efficiently maintain and identify consecutive dirty pages, the dirty cache maintains two hashmaps and a sequential list (i.e., data structures in the dashed line rectangle).

B. Right Prediction

To achieve a high cache hit ratio, the ultimate question is how to predict whether a page will be reused in the near future. Our workload characterization on recency and frequency in Section II sheds some light on this question. Note that recency and frequency are the most effective and accurate prediction metrics [6]. For storage system level workloads, the temporal distance of a read request after the same read request is considerably long. Thus, recency is not so helpful to predict clean page reuse. On the contrary, the temporal distances of write after write requests are relatively short. Thus, recency can be useful to predict dirty page reuse. The frequency metrics for both read and write requests are fairly good since most accesses concentrate on a small portion of pages.

Based on the above analysis, *Hibachi* uses a frequency-based cache policy, i.e., LFU-Aging [14] (Least Frequently Used with Aging), to manage the clean cache, and a recency-based cache policy, i.e., LRU (Least Recently Used), to manage the dirty cache. The reasons we choose LFU-Aging and LRU are twofold: they are widely adopted, and they cause low overhead. Clearly, other fancier cache policies can be designed or applied to each side with the potential of increasing cache hit ratio, but high algorithm overhead can offset the overall performance gain.

LFU-Aging [14] is an improved version of the traditional LFU algorithm. LFU is prone to cache pollution problems due to some items only being frequently accessed for a short period of time. Without properly enforced aging, these items waste cache space and receive no hits. To resolve this problem, LFU-Aging reduces all frequency counts by half when the average of all the frequency counters in the cache exceeds a given *average frequency threshold*. In this paper, we set the *average frequency threshold* to 5.5 as used in [15].

C. Right Reaction

Typically, if a page gets a hit, its priority increases to delay eviction. For recency, the page is moved to the MRU (Most Recently Used) position. For frequency, the page's frequency counter is increased. However, for hybrid memory, considering the limited NVRAM space and DRAM's shorter access latency, *Hibachi* distinguishes between read hits and write hits in order to fully utilize NVRAM to improve write hit rate (i.e., minimize write traffic) and to fully utilize DRAM to shorten read access latency.

Based on our observations, only write hits on dirty pages save storage write traffic. If a page is written once or rarely, but frequently read, keeping it in NVRAM wastes precious NVRAM space. Also, since NVRAM's read latency is several times longer than DRAM's, we should quickly migrate the page from NVRAM to DRAM. Therefore, we need to detect

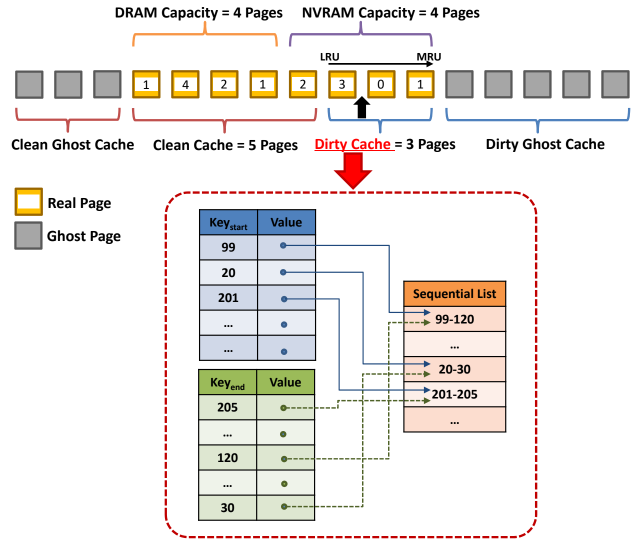


Fig. 5. ***Hibachi* Architecture and Algorithm.**

these kinds of pages and treat them differently. Our measurement in Section II shows that read after write temporal locality is poor and read and write frequency is good. Keeping this in mind, our design includes a frequency counter for each dirty page. A dirty page's counter increases only when a read hit happens. On the other hand, when a write hit happens on the dirty page, its counter will not be increased. Instead we move the page to the most recently used position, since the dirty cache is managed by the LRU policy.

When a dirty page is selected for eviction from NVRAM, we first compare its frequency with the frequency of the least frequent clean page in DRAM. If the dirty page's frequency is greater than the clean page's frequency, the clean page is evicted instead of the dirty page, and the dirty page will be migrated to DRAM. Otherwise, the dirty page is evicted from NVRAM. Before migrating a dirty page from NVRAM to DRAM, or evicting a dirty page, the page must be first synchronized to storage. Note that the frequency counters of dirty pages are also aged by the LFU-Aging algorithm.

D. Right Adjustment

For higher cache hit rates, *Hibachi* can adjust the dirty cache size and the clean cache size according to workloads' tendency (e.g., read intensive versus write intensive). For example, if the current workload is read intensive, *Hibachi* can detect it and borrow NVRAM to cache hot clean pages. To decrease storage writes, we prioritize the dirty cache over the clean cache during cache size adjustment. Two ghost caches, one for each real cache, are maintained to assist the adaptively resizing process. A ghost cache only stores the page number (metadata) of recently evicted pages, not the actual data. A similar adjustment mechanism can be found in our previous work [16]. However, the adjustment mechanism in that work is designed for an NVRAM-only buffer cache, so we modify and extend it to fit the hybrid memory architecture as shown below.

We use $\hat{D}$ to denote the desired size for the dirty cache, and $\hat{C}$ to denote the desired size for the clean cache. We use S to denote the sum of maximum real pages that can be stored in both NVRAM and DRAM.

If a page hits the clean ghost cache, it means we should not have evicted this clean page. To remedy this, we will enlarge $\hat{C}$. Every time there is a ghost hit, $\hat{C}$ increases by one page and $\hat{D}$ decreases by one page. Note that neither $\hat{C}$ nor $\hat{D}$ can be larger than S.

Similarly, a page hitting the dirty ghost cache implies we should not have evicted this dirty page. To remedy this, we will enlarge $\hat{D}$. To save write traffic and keep dirty pages in the cache longer, we enlarge $\hat{D}$ much faster. If the clean ghost cache size is smaller than the dirty ghost cache, $\hat{D}$ increases by two. If the clean ghost cache size is greater than or equal to the dirty ghost cache size, $\hat{D}$ increases by two times the quotient of the clean ghost cache size and the dirty ghost cache size. Thus, the smaller the dirty ghost cache size, the larger the increment.

E. Right Transformation

Buffer cache policies usually only evict a single page at a time when the cache needs to reclaim space. Moreover, if the victim is a dirty page, it should be flushed before its eviction. For *Hibachi*, if the victim is in the clean cache, it deals with it similarly to the majority of buffer cache policies. However, *Hibachi* will do quite intelligent and efficient judgment and operations when the victim is in the dirty cache since it is designed for disk arrays. As spinning devices, HDD's sequential access speed (on the order of 100MB/s) is orders of magnitude faster than its random access speed (roughly 1MB/s) [17]. The slow random access speed is always a bottleneck constraining HDD I/O performance. This section presents how *Hibachi* transforms random disk writes to sequential disk writes to exploit the fast sequential write speed [18].

When evicting from the dirty cache, *Hibachi* first tries to synchronize the longest set of consecutive dirty pages (with the help of the *sequential list* described later), evict the least frequent page, and migrate the rest of the pages to the LRU end of the clean cache. However, to execute this series of operations, we have to ensure the length of the longest set of consecutive dirty pages is over a given *threshold* (e.g., 10). If its length is below the *threshold*, *Hibachi* evicts the LRU page from the dirty cache. In either case, it inserts the evicted page's page number into the MRU position of the dirty ghost cache. Since a dirty page is evicted, it must update the *sequential list* accordingly.

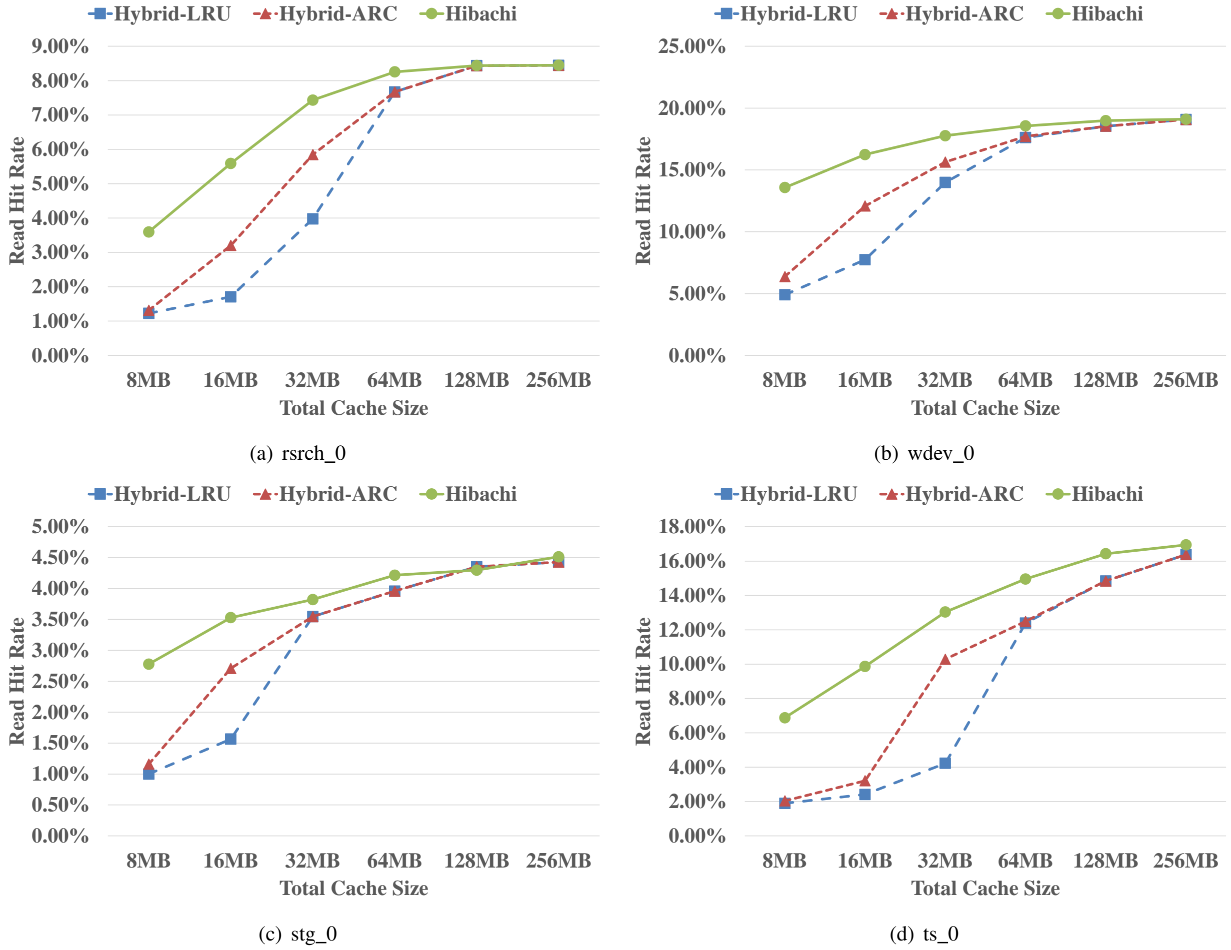
The *sequential list* is designed to accelerate identifying the longest set of consecutive dirty pages in the dirty cache. If several pages have consecutive page numbers, they constitute a *sequential page list* (*sequential list* for short). All sequential lists are stored in a priority queue ordered by the length (page count) of the sequential list. It has a double-HashMap structure that efficiently keeps track of the consecutiveness of the cached dirty pages. *Hibachi* stores two pieces of HashMap information of both the start and end page number into the sequential list. Every time a new dirty page is added, *Hibachi* checks whether it can merge into any existing sequential lists by looking up two Hash Tables. If the new dirty page can merge into any existing sequential lists, Hibachi merges two sequential lists into one larger list. Then, the two HashMaps and the sequential list information are updated accordingly. For example, assuming a sequential list with page numbers 3, 4, 5, and 6 (represented by 3-6) already exists, *Hibachi* stores 3→(3-6) in the *start HashMap* and 6→(3-6) in the *end HashMap*. When a new dirty page number 7 enters the dirty cache, Hibachi consults two HashMaps to see whether there are any sequential lists starting with 8 or ending with 6. In this example, since Hibachi finds sequential list (3-6) ends with 6, it merges the new page number 7 with the existing sequential list (3-6) into a larger sequential list (3-7). Hibachi updates all HashMaps and the sequential list entry accordingly.

Note that the *sequential list* only introduces negligible space overhead. A sequential list update only occurs when a dirty page is inserted to the cache or a dirty page is evicted. For a dirty page synchronization or consecutive dirty pages synchronization, the corresponding sequential list entry containing the page(s) is deleted. For a dirty page insertion, if the dirty page has no consecutive neighbors in the dirty cache, a new sequential list entry is created and inserted into the HashMaps and the priority queue.

The aforementioned *threshold* mechanism is very important to *Hibachi*. Intuitively, continuously flushing the longest dirty pages to disk arrays could "fully utilize" HDDs' fast sequential write performance. However, based on our analysis and evaluation, we found this approach can lead to a low cache hit ratio and poor storage performance because always flushing the longest set of consecutive dirty pages may cause an inevitable situation where there are no more consecutive dirty pages. We call the beginning of this situation "bad time." After the "bad time," if a newly inserted dirty page can merge with an existing dirty page, they become the longest set of consecutive dirty pages. Per the policy, they will be the eviction candidate next time. However, this newly inserted dirty page may be hot data and should not be evicted so early. Thus, by avoiding this problem, the *threshold* mechanism provides a simple and effective way to exploit sequential write opportunities without losing temporal locality.

F. Put Them All Together: Overall Workflow

Now, we integrate all these four approaches together and describe an overall workflow. A cache hit triggers one of the following three cases: 1) if it is a read hit on a clean page or a dirty page, we increase its frequency counter; 2) if it is a write hit on a clean page, we migrate the page to NVRAM and insert it to the MRU position; or 3) if it is a write hit on a dirty page, we keep it in NVRAM and move it from the current position to the MRU position. When a page hits a ghost cache, we enlarge the desired size for its corresponding cache. Note that the dirty cache can only grow up to the capacity of the NVRAM, while the clean cache can grow up to the sum of the capacity of the DRAM and NVRAM.



(a) rsrch_0

(b) wdev_0

(c) stg_0

(d) ts_0

Fig. 6. **Average read hit rates**

For cache misses, when both the DRAM and NVRAM are not full, missed pages are fetched from storage and inserted into DRAM for read misses and into NVRAM for write misses. When the cache space is full, if the clean cache size is larger than its desired size, *Hibachi* evicts the least frequent clean page. Otherwise, *Hibachi* evicts a victim from the dirty cache side. Before eviction, the length of the longest consecutive dirty pages is checked. If the length is above the threshold, *Hibachi* evicts a dirty page with the least frequency among these consecutive dirty pages and migrates the remaining pages to the clean cache. However, if the length is equal to or below the threshold, *Hibachi* favors temporal locality more. Thus, *Hibachi* checks the LRU page of the dirty cache. If its frequency is greater than the least frequently used page in the clean cache, it evicts the clean page and moves the dirty page to the clean cache. Otherwise, the dirty LRU page is evicted. As stated above, a dirty page must be synchronized to storage before getting migrated from NVRAM to DRAM or evicted from NVRAM.

V. Performance Evaluation

A. Evaluation Setup

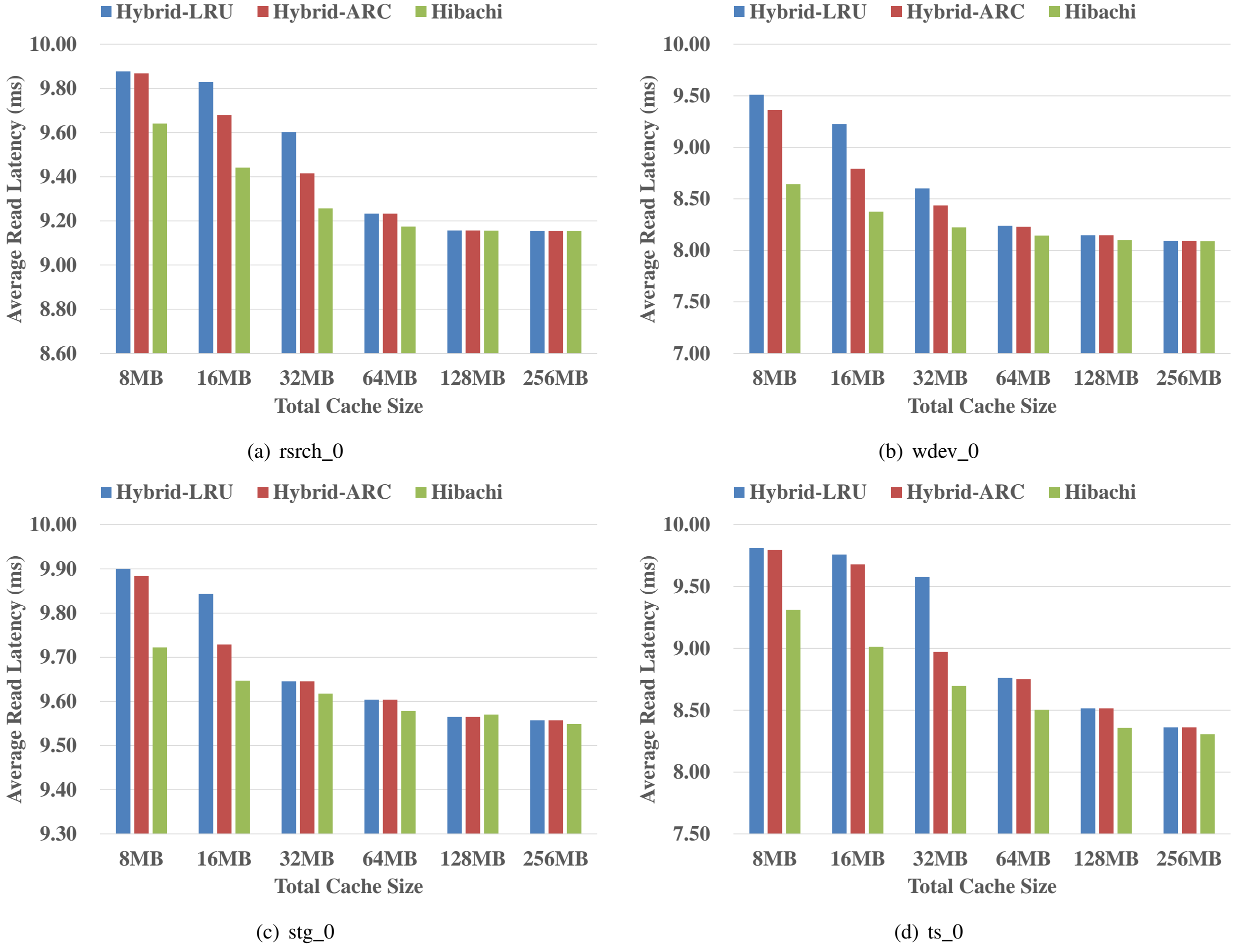
To evaluate our proposed buffer cache policy, we compare *Hibachi* with two popular cache policies: LRU (Least Recently Used) and ARC (Adaptive Replacement Cache) [6]. We modified both policies to fit into hybrid memory systems as follows;

- *Hybrid-LRU*: DRAM is a clean cache for clean pages, and NVRAM is a write buffer for dirty pages. Both caches use the LRU policy.
- *Hybrid-ARC*: An ARC-like algorithm to dynamically split NVRAM to cache both clean pages and dirty pages, while DRAM is a clean cache for clean pages.

Hibachi and these two policies are implemented on *Sim-ideal*, a public, open-source, multi-level caching simula-

TABLE I
Trace Summaries.

Trace Name	Total Requests	Unique Pages	R/W Ratio
rsrch_0	3,044,209	87,601	1:10.10
wdev_0	2,368,194	128,870	1:3.73
stg_0	5,644,682	1,507,247	1:2.28
ts_0	3,779,371	222,267	1:3.79



(a) rsrch_0

(b) wdev_0

(c) stg_0

(d) ts_0

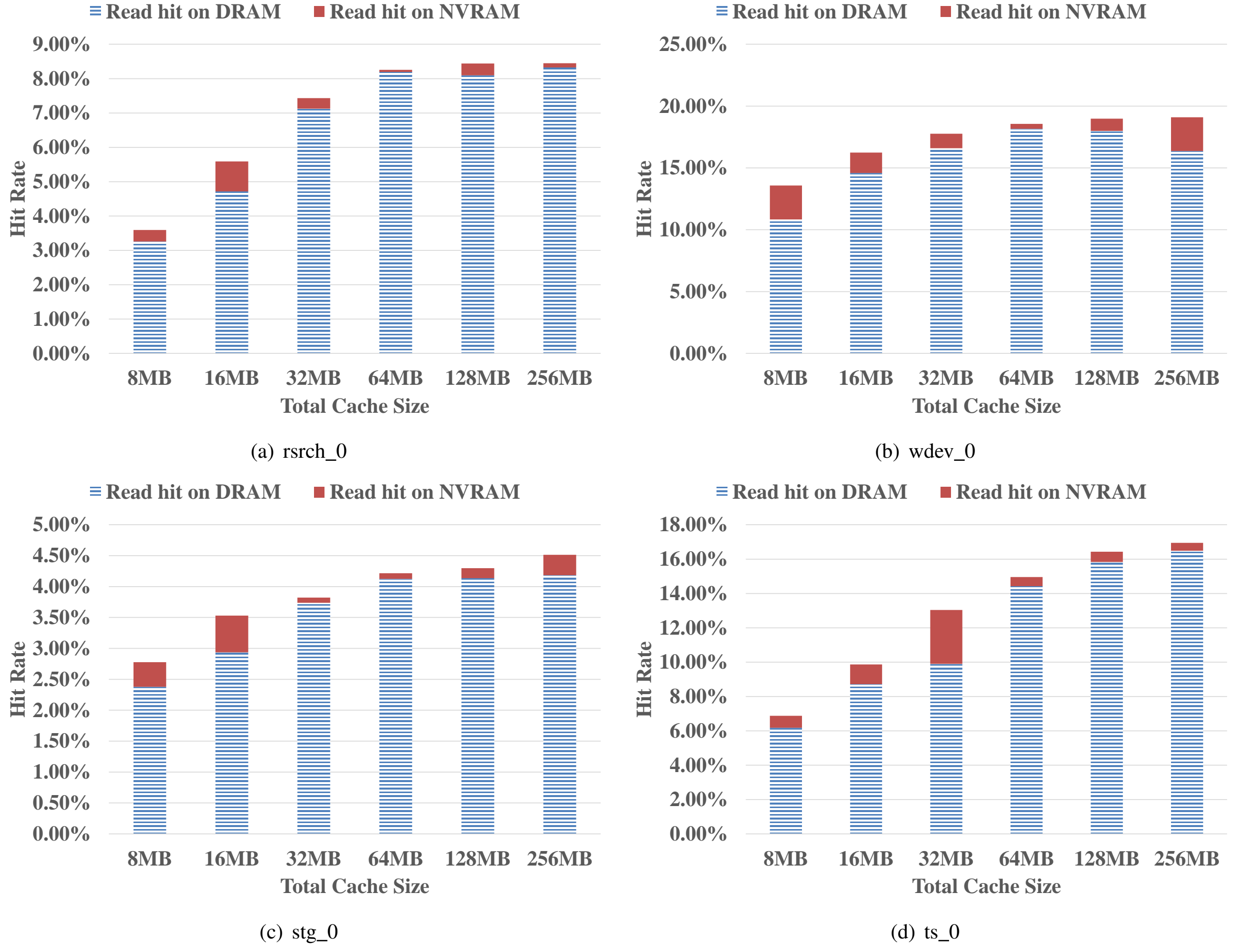
Fig. 7. **Average read latency. Note that the y-axis does not start from zero.**

tor [19]. For evaluations, we configure a 4KB cache block size and employ the popular MSR Cambridge enterprise server traces [12], [13]. As in Table I, four MSR traces (rsrch_0, wdev_0, stg_0 and ts_0) are adopted because the experimental results of the rest of the MSR traces show similar patterns. *Cache size* refers to the total size of both DRAM (half of the total) and NVRAM (the other half). The cache sizes vary from 8MB (2,048 of 4KB blocks) to 256MB (65,536 of 4KB blocks). Note that the MSR traces have relatively small working set sizes, such that our cache sizes can cover a small to large percentage of the workload working set sizes (i.e., unique pages for each trace in Table I).

For performance metrics, both read and write hit rates are employed to evaluate caching policy performance. In addition to these hit rates, we evaluate cache latency with various DRAM and NVRAM latency configurations to consider diverse NVRAM and DRAM performance disparities. Lastly, real disk array write throughput is also evaluated. To observe *Hibachi*'s performance impact on disk arrays, whenever a dirty page(s) needs to synchronize with storage, *Sim-ideal* logs the I/O requests to a file. We make these I/O requests compatible to the Fio [20] tool, which will later replay these requests on a disk array for *Hibachi*, *Hybrid-LRU*, and *Hybrid-ARC* evaluation. Fio is a flexible tool that can both produce synthetic I/O traces and replay I/O traces. To avoid host interference, we set Fio with "direct=1" and "ioengine=sync," which bypasses the host page cache. For the disk array, we use *mdadm* [21] – a Linux Software RAID array management tool to create a RAID 5 with six Seagate SAS disk drives (ST6000NM0034-MS2A, SAS 12Gbps, 6TB, 7200rpm).

B. Evaluation Results

1) Read Performance: Figure 6 presents average read hit rates of *Hibachi* under different cache sizes with different workloads. Compared to *Hybrid-LRU*, *Hibachi* significantly improves the read hit ratio by an average of 3× (8MB total cache size), 2.9× (16MB), 1.8× (32MB), and 1.1× (64MB), respectively. It improves up to approximately 4× (2.41% in Hybrid-LRU vs. 9.87% in *Hibachi* with a 16MB cache size, ts_0 workload). *Hibachi* also outperforms *Hybrid-ARC* by an average of 2.7× (8MB), 1.9× (16MB), 1.2× (32MB), 1.1× (64MB), respectively. As Figure 6 illustrates, the performance improvement generally increases with smaller cache sizes. This improvement results from *Hibachi*'s *Right Prediction*, *Right Reaction*, and *Right Adjustment*. Considering these storage server I/O workloads are typically write-intensive



(a) rsrch_0

(b) wdev_0

(c) stg_0

(d) ts_0

Fig. 8. ***Hibachi*'s NVRAM and DRAM contribution to total read hit rates**

and the total read hit rate percent is quite low, these are substantial improvements.

Figure 7 shows the average system read latency for each cache policy. Note that the y-axis does not start from zero. The following formula can calculate the average system read latency:

$$ASRL = Lat_{RAM} * HitRate + Lat_{HDD} * (1 - HitRate)$$

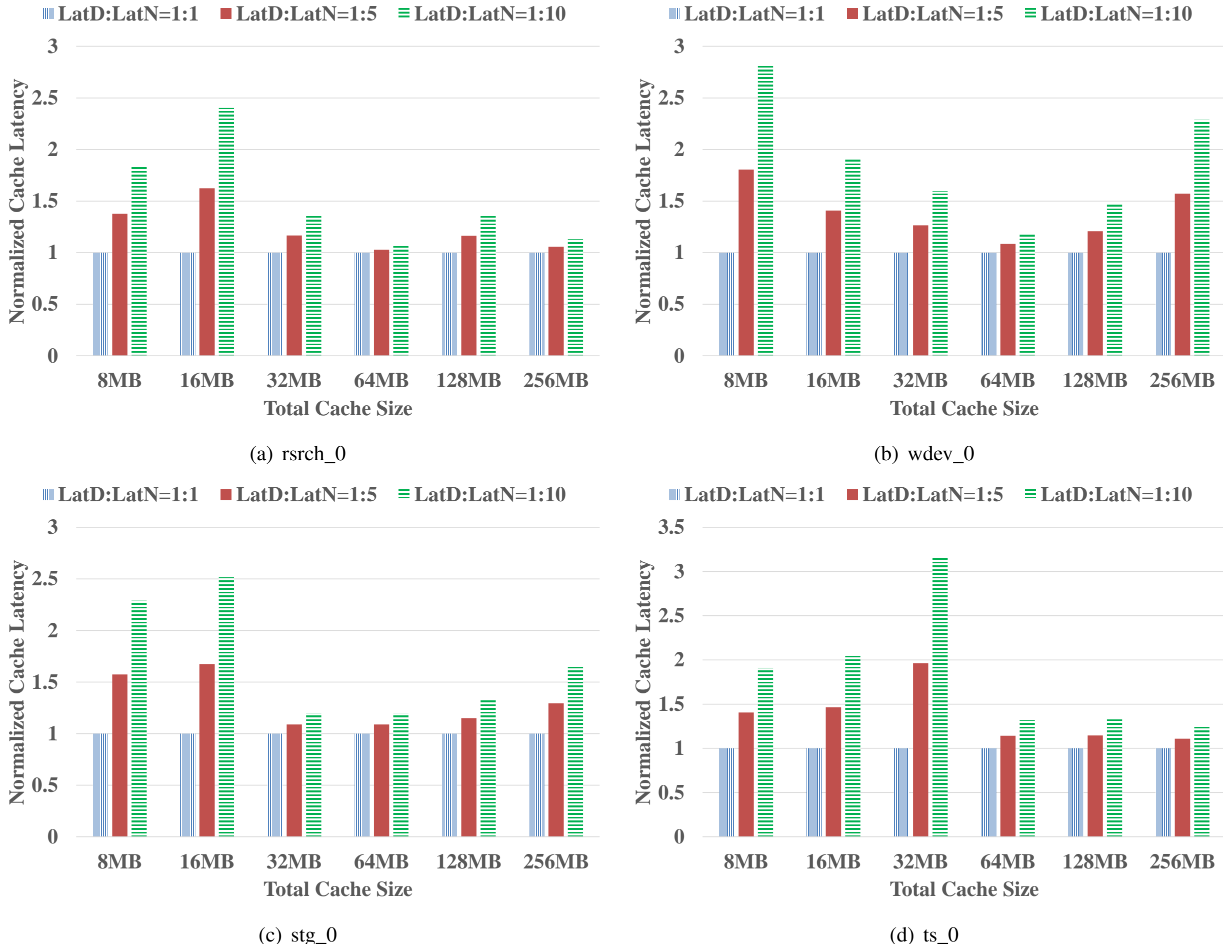
$ASRL$ is the average system read latency, Lat_{RAM} is average RAM read latency, Lat_{HDD} is average HDD read latency, $HitRate$ is read hit rate at RAM. We assume the average access read latency for RAM (both NVRAM and DRAM) are 50 ns, and the average access read latency for hard disk drive is 10 ms. From the figures we can find out when the cache size is relatively small, *Hibachi* can help to reduce decent amount of average system read latency compared with *Hybrid-LRU* and *Hybrid-ARC*.

Figure 8 is a stacked column chart displaying both NVRAM and DRAM contributions to *Hibachi* total read hit rates. Thus, Figure 8 decomposes *Hibachi*'s total read hit rates into NVRAM hit rates and DRAM hit rates for better understanding. As expected, a dominant portion of read hits occurs in DRAM because DRAM is configured as a read cache (i.e., the clean cache). Even though NVRAM is primarily configured as a write buffer (i.e., the dirty cache), *Hibachi* dynamically adjusts the dirty cache size and the clean cache size according to workload characteristics. Thus, for read-intensive periods in each workload, *Hibachi* dynamically borrows NVRAM space to cache hot clean pages to increase cache hit ratios. Consequently, NVRAM read hit rate contributions are clearly visible in Figure 8, which verifies *Hibachi*'s dynamic adjustment feature (i.e., *Right Adjustment* described in Section IV-D).

In general, average read cache latency is more important than average write cache latency because read operations are more response-time critical than write operations. Different NVRAM technologies have very different read latencies. Compared to DRAM, their read access latency can range from similar (e.g., NVDIMM) to 10× longer (e.g., PCM). To explore the impact of NVRAM's access latency on read performance, the following formula can calculate the average read cache latency:

$$ARCL = Lat_N * Rate_N + Lat_D * Rate_D$$

$ARCL$ is the average read cache latency, Lat_N is average NVRAM read Latency, Lat_D is average DRAM read Latency, $Rate_N$ is Read hit rate at NVRAM, and $Rate_D$ is Read hit



(a) rsrch_0

(b) wdev_0

(c) stg_0

(d) ts_0

Fig. 9. **Normalized read cache latency for *Hibachi*.** *Lat_D stands for average DRAM access latency. Lat_N is average NVRAM access latency.*

rate at DRAM. We assume average DRAM access latency is 50 ns. Our experiments vary the average NVRAM access latency to 1, 5, and 10× longer than DRAM's.

As a *Hibachi* example, $Rate_N$ and $Rate_D$ are presented in Figure 8 and the corresponding normalized $ARCL$ is plotted in Figure 9. As in Figure 9, even for the case that assumes NVRAM read latency is 10× longer than DRAM, the largest $ARCL$ performance disparity is just 3.16× the case where we assume NVRAM read latency is the same as DRAM. However, in most cases, there is not such performance degradation. A meaningful takeaway is that even though some NVRAM's read latency is far from DRAM's, an intelligent hybrid scheme (such as *Hibachi*'s *Right Reaction* to identify hot read pages and migrate them to DRAM quickly) can minimize overall performance degradation.

2) Write Performance: Figure 10 presents average write hit rates of all three cache policies under different cache sizes with different workloads. Compared to *Hybrid-LRU*, *Hibachi* improves write hit ratios by an average of 2.2% (8MB total cache size), 2.5% (16MB), 4.7% (32MB), 4.8% (64MB), 8.4% (128MB), and 3.4% (256MB), respectively. Similarly, compared to *Hybrid-ARC*, performance improves by an average of 2.7%, 3.4%, 5.7%, 5.0%, 8.4%, and 3.4%, respectively. A higher hit rate results in write response time improvement and reduced write traffic to storage arrays.

Figure 11 plots average write throughput on the real disk array. *Hibachi* improves average write throughput across all cases and up to more than 10× Hybrid-LRU and Hybrid-ARC when the cache size is large. This is because a large NVRAM cache space enables *Hibachi* to accumulate more consecutive dirty pages with the help of *Hibachi*'s *Right transformation*. On the other hand, both Hybrid-LRU and Hybrid-ARC do not have a strong correlation between the cache size and write throughput. This implies that write traffic in the larger write buffer cache is not necessarily more friendly to disk storage arrays if the cache scheme is not equipped with an intelligent write buffer management mechanism like *Hibachi*.

VI. Related Work

DRAM-based cache policies (i.e., page replacement policies) have been studied for decades. Belady's cache policy is well-known for an offline optimal algorithm [22]. A primary goal of traditional inline cache policies is to improve a cache hit rate towards the optimal rate. Most cache policies utilize recency and frequency to predict a page's future access. The most representative recency-based cache policy is the Least

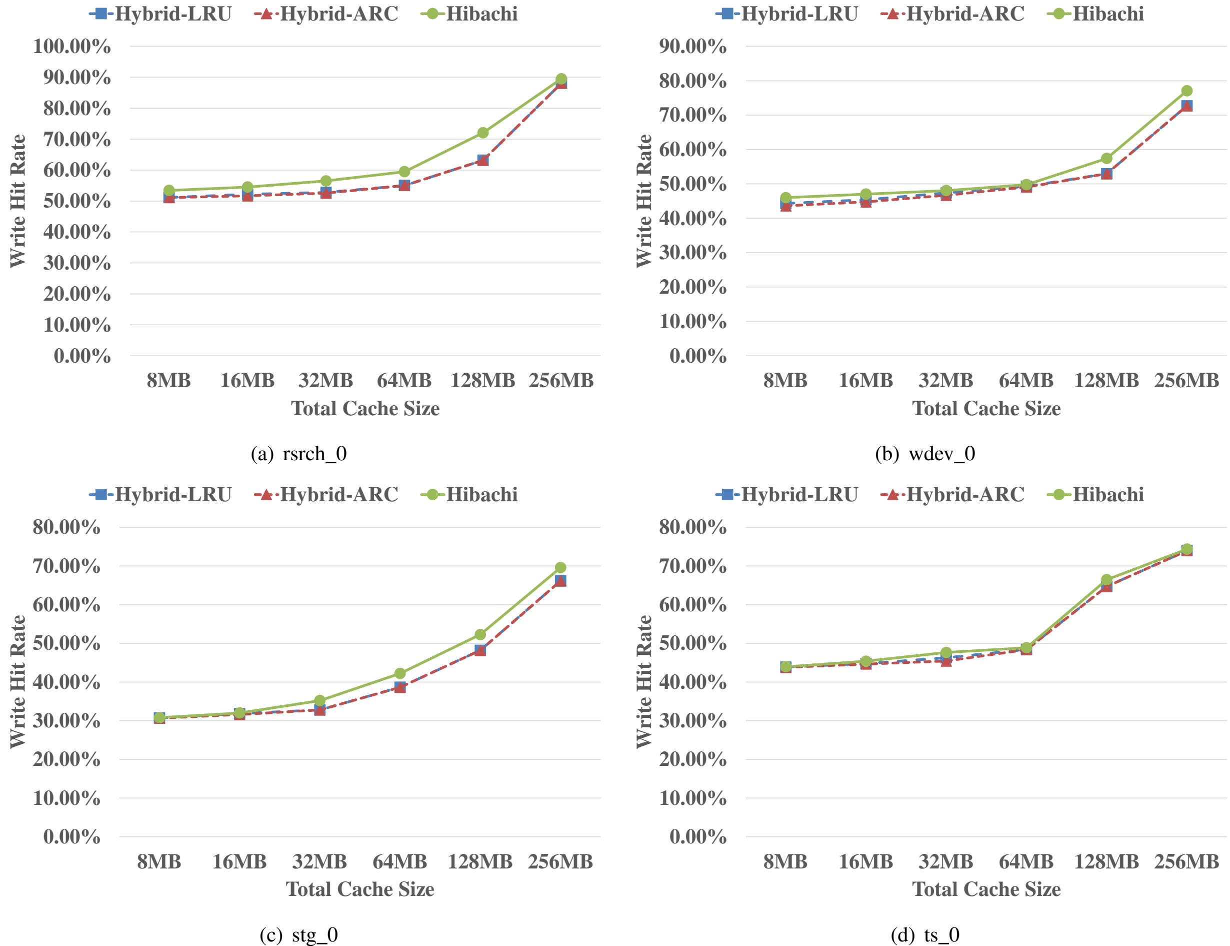


Fig. 10. **Total write hit rate comparison**

Recently Used (LRU) algorithm. However, this overlooks frequency factor. Many LRU variants have been proposed for different use cases [23], [24]. Similarly, the Least Frequently Used (LFU) algorithm only considers frequency, and many variants have been developed [8], [14]. To take advantage of both frequency and recency, some cache policies (e.g., *ARC*) are designed by adapting to workload characteristics [6], [16].

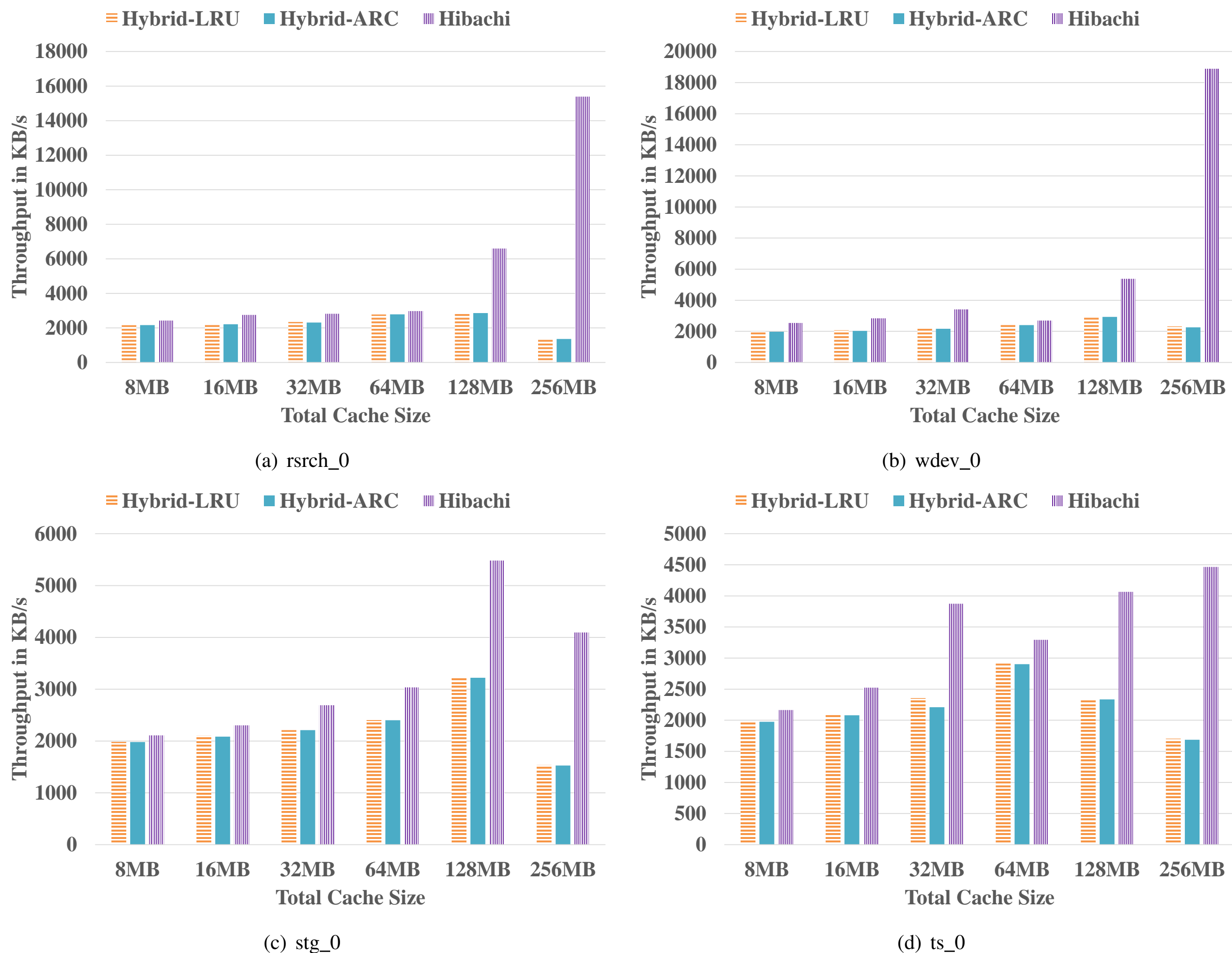
With the advancement of new NVRAM technologies, replacing DRAM with NVRAM, or using DRAM and NVRAM hybrid systems, is recently drawing more attention. New NVRAM technologies include STT-RAM [3], [25], MRAM [26], NVDIMM [27], PCM [28], [29], RRAM [30], and 3D XPoint [1], [31]. Among them, STT-RAM, MRAM, and NVDIMM have potential to replace DRAM, while the other are still several times slower than DRAM.

NVRAM-based cache policies are generally co-designed with storage devices such as HDDs or SSDs. Hierarchical ARC (*H-ARC*) employs NVRAM as main memory and SSDs as storage to extend SSD lifespan. *H-ARC* is designed to minimize storage write traffic by dynamically splitting the cache space into four subspaces: dirty/clean caches and frequency/recency caches [16]. *I/O-Cache* adopts NVRAM as main memory and HDDs as storage to maximize I/O throughput [18]. *I/O-Cache* intelligently regroups many small random write requests into fewer and longer sequential write requests to exploit HDDs' fast sequential write performance.

DRAM and NVRAM hybrid memory systems are another approach to integrate NVRAM into computer systems. PDRAM [32] is a heterogeneous architecture for main memory composed of DRAM and NVRAM memories with a unified address. A page manager is utilized to allocate and migrate pages across DRAM and NVRAM. Qureshi *et al.* [33] discuss a design of NVRAM-based primary main memory with DRAM as a faster cache. Our proposed *Hibachi* policy belongs to this category. However, *Hibachi* is designed for storage arrays instead of main memory. Moreover, it manages NVRAM and DRAM cooperatively instead of individually.

VII. Conclusion

Based on our in-depth study of storage server I/O workloads and our insightful analysis of hybrid memory properties, this paper proposes *Hibachi* – a novel cooperative hybrid cache exploiting the synergy of NVRAM and DRAM for storage arrays. *Hibachi* utilizes DRAM as a read cache for clean pages and NVRAM mostly as a write buffer for dirty pages. In addition, it judiciously integrates the proposed four main design



(a) rsrch_0

(b) wdev_0

(c) stg_0

(d) ts_0

Fig. 11. **Average write throughput with disk arrays**

factors: *Right Prediction, Right Reaction, Right Adjustment, and Right Transformation.* Consequently, *Hibachi* outperforms popular conventional cache policies in terms of both read performance and write performance. This work shows the potential of designing better cache policies to improve storage system performance and motivates us to put more effort into this area of research.

Acknowledgment

The work was partially supported by the following NSF awards: 1053533, 1439622, 1217569, 1305237 and 1421913. This work was done before the author (Dongchul Park) joined Intel.

References

[1] Micron, "3D XPoint Technology," tech. rep., Micron Technology, 2015.

[2] mandetech, "Micron brings nvdimms to enterprise."

[3] EETimes Asia, "STT-MRAM to lead 4.6B dollars non volatile memory market in 2021," 2016.

[4] J. Xu and S. Swanson, "Nova: A log-structured file system for hybrid volatile/non-volatile main memories," in *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, FAST'16, (Berkeley, CA, USA), pp. 323–338, USENIX Association, 2016.

[5] A. Dan and D. Towsley, "An approximate analysis of the lru and fifo buffer replacement schemes," in *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '90, (New York, NY, USA), pp. 143–152, ACM, 1990.

[6] N. Megiddo and D. S. Modha, "ARC: a Self-tuning, Low Overhead Replacement Cache," in *FAST*, pp. 115–130, 2003.

[7] S. Lee, H. Bahn, and S. H. Noh, "Clock-dwf: A write-history-aware page replacement algorithm for hybrid pcm and dram memory architectures," *IEEE Trans. Comput.*, vol. 63, pp. 2187–2200, Sept. 2014.

[8] Y. Zhou, Z. Chen, and K. Li, "Second-level buffer cache management," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, pp. 505–519, June 2004.

[9] G. Yadgar, M. Factor, and A. Schuster, "Karma: Know-it-all replacement for a multilevel cache," in *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, FAST '07, (Berkeley, CA, USA), pp. 25–25, USENIX Association, 2007.

[10] M. Woods, "Optimizing storage performance and cost with intelligent caching," tech. rep., NetApp, August 2010.

[11] Z. Yang, J. Wang, D. Evans, and N. Mi, "AutoReplica: Automatic Data Replica Manager in Distributed Caching and Data Processing Systems," in *1st International workshop on Communication, Computing, and Networking in Cyber Physical Systems (CCNCPS)*, IEEE, 2016.

[12] D. Narayanan, A. Donnelly, and A. I. T. Rowstron, "Write Offloading: Practical Power Management for Enterprise Storage," in *FAST*, pp. 253–267, 2008.

[13] Storage Networking Industry Association (SNIA). http://www.snia.org/.

[14] M. Arlitt, L. Cherkasova, J. Dilley, R. Friedrich, and T. Jin, "Evaluating content management techniques for web proxy caches," *SIGMETRICS Perform. Eval. Rev.*, vol. 27, pp. 3–11, Mar. 2000.

[15] S. Romano and H. ElAarag, “A quantitative study of recency and frequency based web cache replacement strategies,” in *Proceedings of the 11th Communications and Networking Simulation Symposium*, CNS ’08, (New York, NY, USA), pp. 70–78, ACM, 2008.

[16] Z. Fan, D. H. Du, and D. Voigt, “H-ARC: A non-volatile memory based cache policy for solid state drives,” in *MSST*, pp. 1–11, 2014.

[17] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.91 ed., May 2015.

[18] Z. Fan, A. Haghdoost, D. H. Du, and D. Voigt, “I/O-Cache: A Non-volatile Memory Based Buffer Cache Policy to Improve Storage Performance,” in *MASCOTS*, pp. 102–111, 2015.

[19] A. Haghdoost, “sim-ideal: Ideal multi-level cache simulator,” 2013.

[20] fio. http://freecode.com/projects/fio.

[21] “mdadm - manage md devices aka linux software raid. http://neil.brown.name/blog/mdadm.”

[22] L. A. Belady, “A Study of Replacement Algorithms for Virtual Storage Computers,” *IBM Systems Journal*, vol. 5, no. 2, pp. 78–101, 1966.

[23] S. Park, D. Jung, J. Kang, J. Kim, and J. Lee, “CFLRU: a Replacement Algorithm for Flash Memory,” in *CASES*, pp. 234–241, 2006.

[24] H. Jung, H. Shim, S. Park, S. Kang, and J. Cha, “Lru-wsr: integration of lru and writes sequence reordering for flash memory,” *IEEE Transactions on Consumer Electronics*, vol. 54, no. 3, pp. 1215–1223, 2008.

[25] T. Kawahara, K. Ito, R. Takemura, and H. Ohno, “Spin-transfer torque RAM technology: review and prospect,” *Microelectronics Reliability*, vol. 52, no. 4, pp. 613–627, 2012.

[26] W. J. Gallagher and S. S. P. Parkin, “Development of the Magnetic Tunnel Junction MRAM at IBM: From First Junctions to a 16-Mb MRAM Demonstrator Chip,” *IBM Journal of Research and Development*, vol. 50, no. 1, pp. 5–23, 2006.

[27] Viking, “NVDIMM-Fastest Tier in Your Storage Strategy,” tech. rep., Viking Technology, 2014.

[28] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, and C. H. Lam, “Phase-change random access memory: A scalable technology,” *IBM Journal of Research and Development*, vol. 52, no. 4, pp. 465–479, 2008.

[29] S. Raoux, F. Xiong, M. Wuttig, and E. Pop, “Phase change materials and phase change memory,” *MRS Bulletin*, vol. 39, no. 08, pp. 703–710, 2014.

[30] R. Waser, R. Dittmann, G. Staikov, and K. Szot, “Redox-based resistive switching memories–nanoionic mechanisms, prospects, and challenges,” *Advanced materials*, vol. 21, no. 25-26, pp. 2632–2663, 2009.

[31] Intel, “3D XPoint Unveiled: The Next Breakthrough in Memory Technology,” tech. rep., Intel Corporation, 2015.

[32] G. Dhiman, R. Ayoub, and T. Rosing, “Pdram: A hybrid pram and dram main memory system,” in *ACM/IEEE Design Automation Conference*, pp. 664–669, July 2009.

[33] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, “Scalable High Performance Main Memory System Using Phase-change Memory Technology,” in *ISCA*, pp. 24–33, 2009.