Ex-Tmem: Extending Transcendent Memory with Non-volatile Memory for Virtual Machines

Vimalraj Venkatesan
National University of Singapore
Email: matvv@nus.edu.sg

Qingsong Wei
Data Storage Institute, A*STAR
Email: WEI_Qingsong@dsi.a-star.edu.sg

Y. C. Tay
National University of Singapore
Email: dcstayyc@nus.edu.sg

***Abstract*—Virtualization and multicore technology now make it possible to consolidate heterogeneous workloads on one physical machine. Such consolidation helps reduce the amount of idle resources. In particular, transcendent memory is a recent idea to gather idle memory into a pool that is shared by virtual machines (VMs). However, the size of transcendent memory is unstable and frequently fluctuates with changing workloads. Contention among VMs over transcendent memory can cause increased cache misses.**

In this paper, we propose a mechanism to extend transcendent memory (called Ex-Tmem) by using emerging non-volatile memory. Ex-Tmem stores clean pages in a two-level buffering hierarchy with locality-aware data placement and replacement. In addition, Ex-Tmem enables memory-to-memory swapping by using non-volatile memory and eliminates expensive I/O caused by swapping. Extensive experiments on implemented prototype indicate that Ex-Tmem improves performance by up to 50% and reduces disk I/O by up to 37%, compared to existing Tmem.

Keywords : Virtual Machines (VMs), Transcendent Memory, Non-volatile Memory.

I. INTRODUCTION

For the past few decades, DRAM has been the building block of main memory of computer systems due to its structural simplicity and relatively low cost. However, recent research has manifested that it is technically difficult to manufacture DRAM with large capacity [12] [14]. As a consequence, DRAM is becoming insufficient for modern systems with an increasing need of large main memory. Memory capacity per core drops 30% every 2 years [12] [14].

To address this issue, several non-volatile memory (NVM) technologies to replace DRAM have been under active development, such as Phase-Change Memory (PCM) [10], Spin Torque Transfer Magnetic RAM (STT-MRAM) [9] and Resistive RAM (RRAM) [5]. These new memory technologies possess the potential to provide a much higher capacity than DRAM while offering comparable read/write performance. More importantly, being non-volatile, they can retain data even if they are not powered. Though currently unfit for complete replacement due to cost, NVM has become a viable component that may be added to current systems to improve performance.

Virtualization is a widely used technology that now supports a multibillion-dollar industry. With this technology, an enterprise can consolidate heterogeneous workloads by giving each a dedicated virtual machine (VM) and running the VMs in parallel on one physical machine. Due to fixed memory allocation for each VM, there are needy VMs demanding more memory and VMs with idle memory unused.

Transcendent memory, or Tmem, is a new approach to optimize RAM utilization in a virtual environment where underutilized RAM from each guest VM and RAM unassigned to any guest (fallow memory), are collected into a central pool at hypervisor (or VMM), that is shared by VMs. It can be viewed as a new level in the memory hierarchy for VMs, between main memory and disks.

A VM can request Tmem to cache its evicted pages from page-cache to save disk reads. However, Tmem is an unstable resource, whose size varies frequently with changing memory demand and working set size at VMs. Contention among VMs over Tmem also causes increased cache misses. This approach only works well for guests with non-concurrent memory pressure. The cached clean pages in the Tmem are discarded when hypervisor shrinks Tmem due to memory pressure. If these pages are requested in future, they incur expensive disk reads.

In this paper, we propose a mechanism to extend transcendent memory (called Ex-Tmem) with NVM so that more pages can be cached. Ex-Tmem puts clean pages in a two-level cache hierarchy considering access locality and different features of DRAM and NVM. With Ex-Tmem, memory-to-memory swapping is enabled and expensive swap I/O is totally removed. We have implemented our Ex-Tmem in Xen hypervisor. Evaluation indicates that Ex-Tmem is efficient in improving performance and reducing disk I/O. We believe, to the best of our knowledge, that our work is the first to propose such a mechanism.

The rest of this paper is organized as follows. Section II provides background and motivation. We present design details of Ex-Tmem in Section III. Section IV gives implementation details. Evaluation and system performance results are presented in Section V. Section VI surveys related work. We summarize this paper with the conclusion and possible future work in Section VII.

II. BACKGROUND AND MOTIVATION

A. Non-volatile Memory

Increasing multi-core concurrency increases the demand on the main memory to retain larger working set, so as to maintain the performance growth. However, DRAM is already hitting density, power and cost limits, i.e. as much as 40% of total system energy is consumed by DRAM [14].

As a DRAM replacement, NVM technologies have been actively developed, which are fast, byte-addressable and power

Technology	Read latency (ns)	Write Latency (ns)	Read Voltage(V)	Write Voltage (V)	Endurance (writes/cell)
SLC Flash	25,000	200,000-500,000	2	15	10^5
PCM	48	150	< 3	< 3	10^8
STT-MRAM	32	40	0.7	+1	10^{15}
RRAM	10-50	10-50	< 3	< 3	10^8
DRAM	15	15	1.8	2.5	10^{18}

TABLE I: A Summary of NVM Characteristics.

efficient. PCM, STT-MRAM and RRAM are representative NVM candidates for DRAM replacement. The main attributes and technical parameters of these NVM technologies are summarized in Table I.

PCM [10] uses distinct phase change materials (crystalline or amorphous) with different resistance to store values of 1 or 0. Because PCM changes the state of underlying material with thermal change, its write performance is slower than DRAM and cells are limited to as few as 10^8 writes. PCM writes consume more power than reads by about one order of magnitude. Since PCM has higher density, but lower performance and endurance than DRAM, it is a good candidate to build hybrid main memory with DRAM. PCM chip with 2Gbits capacity is commercially available.

STT-MRAM [9] has advantages of lower power consumption over DRAM, more write cycles over PCM and better scalability over conventional MRAM which uses magnetic fields to flip the active elements. Its access latencies are 32 ns for read and 40 ns for write, at the same level as that of DRAM [14]. A commercial 64Mbits STT-MRAM chip with DDR3 interface was announced in 2013. However, cell size is the biggest barrier, which limits the density of STT-MRAM.

RRAM [5] is another NVM technology with performance that is comparable to DRAM but better write endurance than Flash. However, this technology is still in its development stages and commercial chip is not available yet.

NVM technology makes it possible to attach it directly to the memory bus. NVM rapidly is becoming a promising technology for the next-generation memory and increasing the attention of system researchers. However, NVM with DIMM interface is not commercially available.

As alternative, Non-Volatile DIMM (NVDIMM) [4] offers a practical NVM as memory for system integrators. NVDIMM combines DRAM and NAND Flash with DIMM interface. It can be plugged into DIMM slots of a standard server and it operates at DRAM speed. It is a persistent device which retains data in the event of power failure or system crash. A NAND Flash of the same capacity as DRAM is placed and does not take any reads or writes during normal operation. The Host will signal the NVDIMM to save or restore the DRAM contents to/from the NAND flash when power is down or up. NVDIMM is commercially available [4]. The Ex-Tmem proposed here is implemented and evaluated using NVDIMM as Tmem extension.

B. Transcendent Memory

In a typical virtualized system, the hypervisor contains fallow memory which is not allocated to VMs and VMs have idle unused memory. Transcendent memory (Tmem) is the pool of memory obtained by collecting the host and guest idle memory. This memory is managed by the hypervisor (or VMM) and is available for all VMs. One can view Tmem as a memory level residing between RAM and disk in the memory hierarchy.

Tmem was first proposed and implemented by Magenheimer [11] and it is now part of major operating systems (e.g. Xen 4.0, Linux 2.6.39 and Oracle VM Server 2.2). VM can utilize Tmem by requesting the hypervisor to create a Tmem pool. Once hypervisor successfully creates a pool to the VM, the VM uses *pool_id* to perform operations on the pool. There are two main usage models of Tmem:

1) Cleancache: This is a cache for clean pages. When the Guest kernel evicts a page, it first attempts to use Cleancache to store the evicted page. The Cleancache is not directly addressable by the guest kernel. The hypervisor can discard its contents anytime (e.g. under memory pressure), so it is *ephemeral* [1].

2) Frontswap: Frontswap provides an interface to store swap pages that a VM would otherwise have swapped to disk [3]. The kernel will always attempt to store the swap page to Frontswap before swapping to disk. A successful Frontswap store avoids the disk write and a later disk read to swap disk.

Cleancache and Frontswap can each be private or shared. What a VM puts into a private pool cannot be used by other VMs, whereas pages in shared pools can be used by any VM. There are four variations (private/shared x cleancache/frontswap). Shared pools works only for cluster file system. This paper focuses on private pools.

The two primary operations performed on a Tmem pool are *tmem_put_page* (or *put*) and *tmem_get_page* (or *get*). In general, a *put* to an ephemeral Cleancache pool will rarely fail but a *get* from a Cleancache pool will often fail. For a persistent Frontswap pool, a *put* may frequently fail, but once successfully *put,* a *get* will always succeed.

A VM evicts a clean page from page buffer and puts it to Tmem Cleancache. Whenever the VM requests this page, it checks Tmem and gets the page back from Tmem. This get can fail, since Tmem can flush (i.e. delete) pages from an ephemeral pool at any time, e.g. when Tmem shrinks due to memory pressure. If a get succeeds, the page is copied to VM's page buffer and deleted from Cleancache, so Cleancache is exclusive cache. If the VM cannot find the page in the Tmem, the page fault generates a disk read as usual. However, the read bypasses Cleancache and goes to VM's page buffer directly.

A *put* to Tmem or *get* from Tmem requires a page copy within RAM and incurs only instruction overhead. We can thus

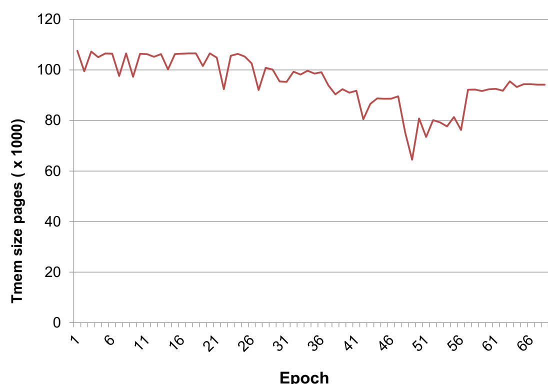


Fig. 1: Dynamically changing size of Tmem

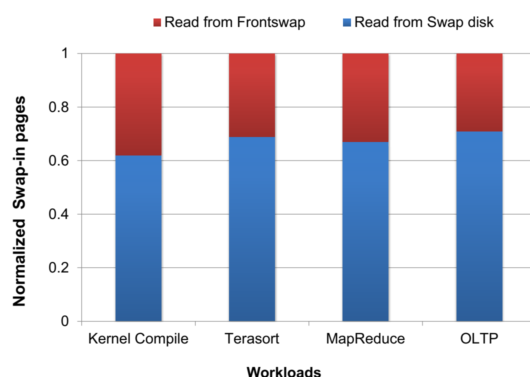


Fig. 2: Swapping of workloads

view Tmem as another level in the memory hierarchy which is a little slower than RAM, but much faster than disk.

Recall that the goal of Tmem is to make better utilization of the host's unallocated memory and the VM's idle memory. **Ballooning** is the underlying mechanism that gathers idle memory at VMs. Ballooning utilizes a balloon driver in the VMs which 'inflate' to get memory from the kernel and return it to the hypervisor. Similarly, when the hypervisor decides to return memory claimed from the VMs, it deflates the VM's balloon, the balloon driver unpins the page-frames and returns it to the VM.

C. Motivation

The size of Tmem is unstable because it may be reclaimed by the hypervisor when it sees memory pressure coming from the VMs. The hypervisor discards the Cleancache page data and return the pages to requesting VMs'. If these discarded pages are accessed later, the hypervisor will suffer a miss at Tmem and have to read the page from disk. To understand the dynamic behavior of Tmem, we measure the Tmem size at runtime when 3 VMs are running different workloads, as shown in Figure 1. We can see that Tmem is a dynamic resource fluctuating in size. Tmem shrinking results in discarding cached pages, which affects performance of the system.

VM guest may swap memory pages to swap disk due to memory pressure. Existing implementation of Tmem puts swap pages into Frontswap to save disk I/O. But a successful *put* cannot be guaranteed due to limited size of Tmem. In this case, swap pages have to be written to swap disk. Later, when the swap pages are accessed, they are read from Frontswap or swap disk. We run 4 workloads in 4 VMs to collect the number of swap pages read from Frontswap and swap disk. Figure 2 shows the normalized value of pages read from Frontswap and swap disk. We can see that a large portion of swap read is from swap disk. For example compiling Linux kernel reads 62% of swapping page from swap disk, while only reads 38% pages from Frontswap of Tmem. Swap-out and swap-in incur lots of disk write and read, which is detrimental to system performance.

Based on the above observation, we are motivated to extend Tmem with emerging NVM so that more pages can be cached to improve system performance and save expensive disk I/O. We also found Tmem is a good interface to integrate NVM into virtual machine environment.

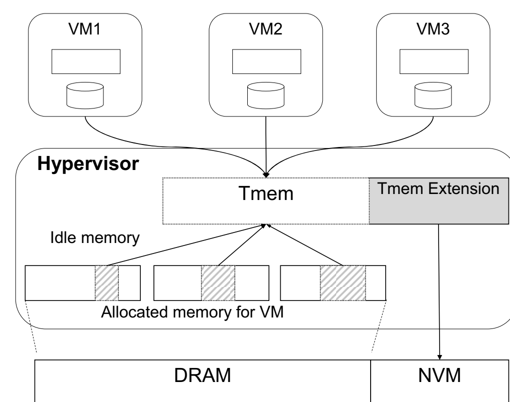


Fig. 3: Ex-tmem with NVM Extension

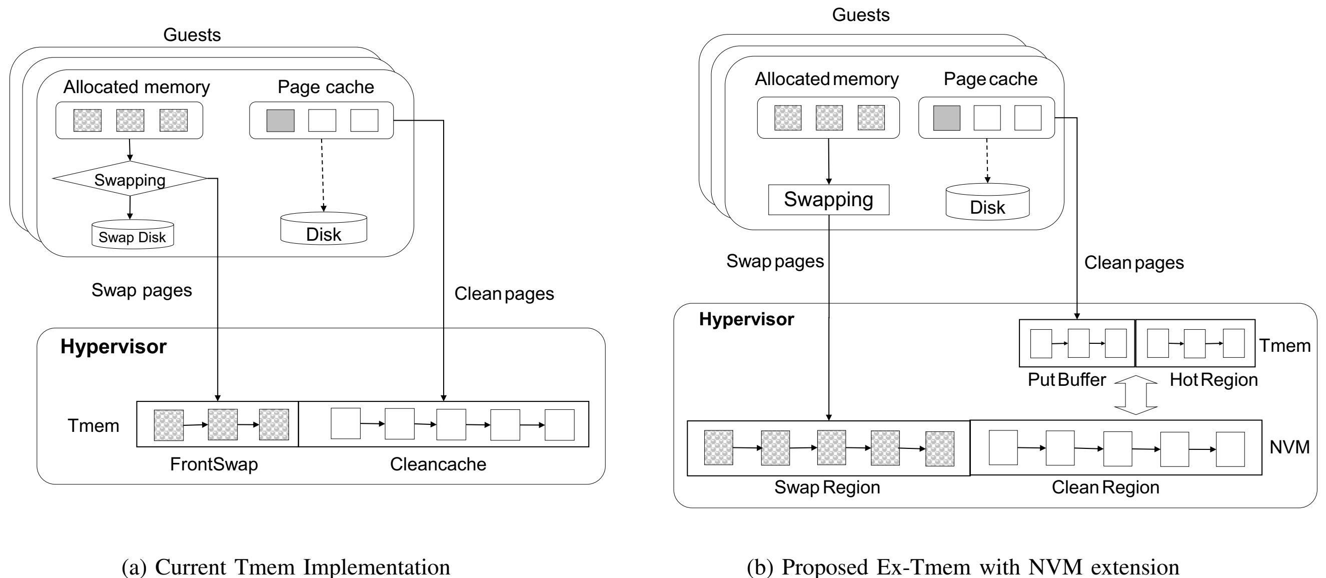
III. Design of Ex-Tmem

In this section, we now present the design of our Ex-Tmem for virtual machine.

A. Design Rationale

The main idea of Ex-Tmem is to use NVM as an extension of the volatile Tmem so that more pages can be cached to save disk I/O. NVM is physically placed on the memory bus alongside DRAM. NVM appears as special memory space in host and it is managed exclusively by the hypervisor. Hypervisor allocates memory from DRAM for virtual machines as usual. The NVM space is not virtualized and reserved only for Tmem extension and not available to the kernel to access directly (Figure 3).

Ex-Tmem stores swap pages and clean pages in the following ways: NVM is partitioned into two regions, **Swap Region**



(a) Current Tmem Implementation

(b) Proposed Ex-Tmem with NVM extension

Fig. 4: Ex-Tmem Architecture

and **Clean Region** as shown in Figure 4(b). Swap Region stores swap pages, while Clean Region and Tmem both cache clean pages. By moving Frontswap from Tmem (DRAM) to NVM, the entire Tmem is now available to Cleancache. There is thus no contention between Frontswap and Cleancache in Tmem.

Swapping memory page to disk is costly. Our Ex-Tmem design eliminates swap I/O entirely by replacing swap disk with NVM through Frontswap interface. Removing swap disk is feasible because NVM density is higher than DRAM. Ex-Tmem leverages on memory-to-memory swapping to write swap data to Swap Region removing the expensive swap I/O to disk. More on memory-to-memory swapping is discussed in Section III-B.

Ex-Tmem exploits a two-level buffering hierarchy to store clean pages. A locality-aware data placement and replacement policy is proposed to identify hot pages and cold pages so that they are stored in different levels, which will be discussed in Section III-C.

B. Memory-to-Memory Swapping

In a VM, sometimes the memory requirements exceed available allocated memory, thus causing page swapping. The current Tmem scheme *puts* the swap pages to Tmem Frontswap. If the *put* is successful, disk I/O is avoided. Otherwise, the swap pages are written to swap disk, as shown in Figure 4(a).

When swapping in, the system searches the Tmem Frontswap first. If found, the swapped pages are read from Tmem to save disk reads. Otherwise, they are read from swap disk. Our earlier experiement in section II-C shows that a larger amount of swap pages are read from swap disk than from Frontswap.

To improve swap performance and reduce swap I/O, we remove the swap disk and always *put* swap pages into NVM. When a VM decides to swap out pages, Ex-Tmem puts them in the Swap Region without writing to swap disk. The swap pages are written to NVM through an operation that we call **memory-to-memory swapping**. Our memory-to-memory swapping is a fast page copy mechanism that swaps out pages from VM's memory to NVM. This eliminates swap I/O and improves the system performance.

A straight-forward approach to use NVM for swapping is to make NVM a ramdisk and allocate a portion of it as swap disk to each VM. However, using ramdisk as swap disk is inefficient because it incurs high overhead of I/O stack. Our memory-to-memory swapping is more efficient than ramdisk approach because of low latency of memory copy.

There is no boundary separating the Swap Region and Clean Region. Ex-Tmem gives higher priority to swap pages to be stored in Swap Region. Writing swap pages into Swap Region will always be successful. When a swap page cannot find free slots in NVM, the clean page at the tail of Clean Region is discarded to make space in NVM. When swap pages are read from the Swap Region, they are invalidated and the space is reclaimed.

C. Two-level caching of Clean Pages

As mentioned in Section III-A, Tmem and Clean Region form a two-level cache space for clean pages. We refer to the DRAM back-end of Transcendent Memory as **Tmem** and the NVM back-end of Transcendent Memory as **Clean Region**. Tmem is small and dynamic, while Clean Region is from NVM, which is large and stable. In general, NVM is slower than DRAM, and write-limited, i.e. PCM and RRAM.

We construct DRAM-based Tmem as first-level cache, while NVM-based Clean Region as second-level cache for

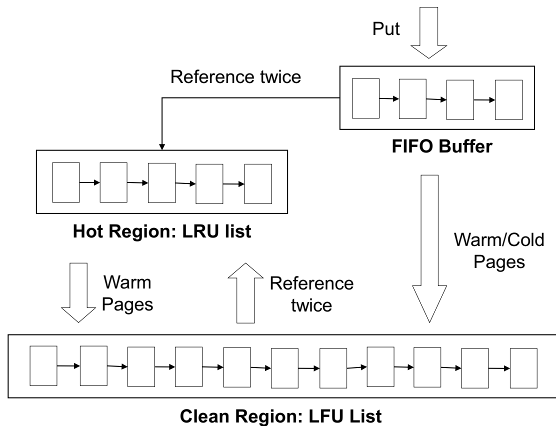


Fig. 5: Two-level buffering of clean pages

clean pages. We place clean pages in the two levels according to the workload access pattern and memory features. We design algorithm to achieve the following goals:

1) Improve the overall hit rate to reduce access latency and disk I/O.
2) Maximize the hit rate of the first-level buffer, which is faster than the second-level buffer.
3) Reduce the number of writes to second-level which is lifetime-limited.

Based on the access pattern for clean pages, they can be classified as hot pages, warm pages and cold pages. We always keep hot and warm pages in either the first-level or the second-level cache, but evict cold pages from the buffer space.

First-level cache: One straight-forward method is to use existing cache algorithms (i.e. LRU, LFU, 2Q, ARC) to manage the first-level cache. For example, LRU is able to remove cold pages from the first-level buffer to make space for the most recently accessed page. However LRU may displace a warmer page to make space for a cold one if the most recently accessed page is cold. Furthermore, a cold page can reside in the first-level cache for a considerable amount of time, polluting the space. We improve on this by partitioning the first-level cache (Tmem) into Hot Region and Put Buffer, thereby quickly identifying and removing cold pages from the first-level cache.

The Put Buffer is organized as a FIFO list, while the Hot Region is organized as an LRU list (See Figure 5). The Put Buffer is a unique portal to receive clean pages in FIFO order, functioning as a filter to identify hot pages and cold pages. If a page residing in the Put Buffer is referenced twice before it is evicted, it is probably a hot page. The page is then moved to the Hot Region and moved to the head of the LRU list. If a page is accessed once or not accessed during its Put Buffer residency, it is probably a warm or cold page, and it will be evicted from the Put Buffer to the second-level cache – the Clean Region. When the Put Buffer is full, the oldest residing page in the FIFO list is evicted to the Clean Region. When the Hot Region is full, the LRU end of the region is evicted to the Clean Region.

Second-level cache: The second-level cache, i.e. Clean Region, uses LFU (i.e. the least-frequently-used replacement policy) to organize cold pages evicted from the first-level cache. When a page from the Put Buffer or the Hot Region is replaced to Clean Region, we use its reference frequency to place the page in the LFU list. Pages with same reference numbers are sorted by their recency of access. When the Clean Region is full, the page at the tail of the LFU list (which has the page with the lowest reference frequency) is discarded. A page accessed twice in the Clean Region is moved up to the Hot Region of first-level cache.

Therefore, the Hot Region stores pages with high short-term access rates. The Clean Region on NVM detects pages that have high long-term access rates. Hence, we could say that the first-level cache captures recency, while the second-level cache captures frequency. In this way, we efficiently make use of the two-level cache to maximize the hit-rate and eliminate disk I/O as much possible. The procedures of *put* and *get* operation are described in Algorithm 1 and Algorithm 2.

Flush-on-reclaim: When host or VM requests back the memory donated to Tmem, the memory is reclaimed from the Tmem, causing Tmem shrink in size. A Flush-on-reclaim policy is used when Tmem shrinks, the pages close to the tail of the LRU list in the Hot Region are flushed to the Clean Region that is part of NVM. These pages are then reclaimed back to VM or host.

IV. IMPLEMENTATION

We have implemented Ex-Tmem on a server platform provided by vendor Viking. This platform is called ADK (ArxCis Development Kit) which is standard server with NVDIMM. In ADK platform, both DRAM and NVDIMM coexist. NVDIMM is supercapacitor-powered Non-Volatile DIMM with DDR3 interface. NVDIMM combines the speed and endurance of DRAM, together with data retention properties of NAND flash. It performs at DRAM speeds and can sustain itself from host power failure or a system crash. This solution can be viewed as the first commercially viable alternative for NVM. With modified BIOS, NVDIMM is isolated and reserved as a special memory zone to Linux kernel, which is used as extension of Tmem in this paper. Hypervisor allocates memory from DRAM for each VM as usual, but the NVM space is not visible to VMs. NVM space is allocated at page granularity. It provides dedicated *NVM_Read()* and *NVM_Write()* to read and write pages from/to NVM space.

We have implemented our scheme in Xen-4.1.3 hypervisor. In addition to the LRU page list in default Tmem, we add two more page lists - a FIFO list to manage the buffer and a LFU page list to manage the NVM. The page to location mapping is handled efficiently using radix-tree. *do_tmem_put*, *do_tmem_get*, *tmem_evict* are the three main methods that perform write, read and evict operations on Tmem respectively.

Our modification in *do_tmem_put* writes the page always to buffer list. Once the hypervisor decides to perform read/write operation on NVM, it calls the corresponding NVM read/write

Algorithm 1: Put (page X)

```
Begin
Initialize:
Buffer queue B, Hot Region queue T and Clean Region queue P
C_i = count of number of references of page i
C_pi = count of number of references of page i in P
if there are free page slots in B then
    add X to B;
else
    remove page Y from tail of B;
    add X to B;
    if there are free page slots in P then
        insert Y in P ordered by C_y;
    else
        remove least frequently used page Z from P
        where C_z = min(C_pi);
        insert Y in P ordered by C_y;
    end
end
```

Algorithm 2: Get (page X)

```
Begin
Initialize:
C_bx = count of number of references of page X in B
C_px = count of number of references of page X in P
C_i = count of number of references of page i
if X is in T then
    move X to head of T;
end
else if X is in B then
    if C_bx >= 2 then
        move X to head of T;
    else
        C_bx = C_bx + 1
    end
end
else if X is in P then
    if C_px >= 2 then
        if there are free slots in T then
            move X to head of T;
        else
            remove page Y from tail of T;
            insert Y in P ordered by C_y;
            add X to head of T;
        end
    else
        C_px = C_px + 1
    end
end
```

API to immediately to complete the operation. This essentially makes it a write-through cache. *do_tmem_get* finds the page in these three lists. *tmem_evict* discards the page from the Cleancache LFU list unless its a shared page.

Concurrency is achieved by using a good locking strategy. There are locks at each datastructures that ensure to avoid deadlock or livelock issues. There is a per-pool read-write lock (*pool_rwlock*), per-object spinlock (*obj_spinlock*), a single global spinlock (*eph_lists_spinlock*). These locks must be acquired to perform any modifications on the data structures.

V. Evaluation

Our experiments are performed with Ex-Tmem on the ADK server platform with Intel 2.5GHZ CPU, 8GB DRAM and a 1TB SATA hard disk plus 16GB NVDIMM. Hypervisor is Xen 4.1.3 with 10 VMs deployed for experiements. Each VM is allocated with 2 vCPUs, 500MB RAM, and 100GB virtual disk. Ubuntu 12.04 is operating system in guest VM.

In our configuration, NVDIMM is twice the size of DRAM. For each VM with 500MB allocated memory, 1GB NVDIMM is enough to store swapping pages. So 16GB NVDIMM is enough to store whole swapping pages for each VM and their clean pages.

We evaluate our system against a) default Xen Transcendent Memory [2] in Linux Kernel with no NVM and NVM extended Transcendent Memory with existing schemes b) traditional LRU [7][13], c) CLOCK-pro [8] which is an improved CLOCK replacement policy and d) Multi-queue replacement algorithm (MQ) [18].

For default Tmem, we consider only the DRAM backend (without any compression or deduplication enabled in Transcendent Memory) for our experiments and evaluation. The other configurations include an additional NVM backend. The DRAM and NVM make a hierarchal Transcendent Memory which are managed by LRU, CLOCK-pro and MQ respectively. In all these 3 policies DRAM holds higher level in cache hierarchy than NVM. The hot pages are given priority to be placed in DRAM and the coldest pages (defined by the policy) in NVM are discarded. In LRU, DRAM and NVM make a page list with the most recently accessed end at DRAM and the least frequently accessed end at NVM. In CLOCK-pro, the active-list is given priority to be placed in DRAM and others placed in NVM. In MQ, the higher level queues with pages that have been accessed frequently in the past are given priority to be placed in DRAM, while others are moved to NVM. Additionally, these configurations store all swap pages into NVM without swapping to disk.

A. Workloads

We use a set of real traces to study the efficiency of our proposed Ex-Tmem scheme on a wide spectrum of workloads. Considering the rapidly growing demand for big-data operations on servers, we evaluate our system with real-time big-data workloads generated by Apache Hadoop v2.3.0 on a single-node Hadoop cluster with HDFS setup on a VM. The workloads are:
MapReduce is a workload that transfers the large input dataset through the shuffle and performs the three steps of generating random data, performing the sort, and finally validating the results.
Terasort is a MapReduce sort that samples the data generated by Teragen. Terasort partitioner uses a two level trie that indexes into a list of sample keys which are written to HDFS.
TestDFSIO benchmarks the underlying HDFS by performing reads and writes.
We also use five workloads generated from **Filebench** which is a storage benchmark that produces a realistic view of real-time production systems. It creates file sets prior to actually running

a workload by pre-creating thousands of files in hundreds of directories with files of varying sizes at specified ranges on which to actually test. The 5 workloads are:
Web server, which generates sequences of open, read and close on multiple files with multiple threads.
File server generates sequences of file operations – create, delete, append, read and write.
Video server generates two different sets of videos, one which is actively served and other inactive. Multiple threads serve the active videos and a thread performs the task of replacing the inactive videos.
Web proxy generates create-write-close, open-read-close, and delete operations of multiple files in directory.
OLTP generates operations using the Oracle 9i database, which performs random reads and writes, and writes to a log file.

B. Performance

Running time: This is the measurement of the running time of the application in the VM. Figure 6(a) shows data comparing the running time of default Xen Transcendent Memory, NVM extended Transcendent memory with LRU, CLOCK-pro, MQ and our design. Results indicate extending Transcendent Memory with NVM reduces running time of the workloads by twice in many cases. This owes to the cost of NVM. Comparing our scheme against LRU, we observe up to 20% improvement for Hadoop workloads and 30% for Filebench workloads. Our scheme also performs better than CLOCK-pro and MQ by 5–20%. The running time depends on the number of hits at first-level cache and the disk I/O caused by cache misses. Our algorithm obtains large number of cache hits from first-level than other schemes, which apparently reduces the running time of the applications.

Hit rate: Fetching data from disk requires at least a factor of 1000 more time than fetching data from a RAM buffer. For this reason, improving overall hit rate can significantly improve the response time of data-intensive applications. NVM extension increases the hit ratio by 25-50% in Figure 6(b), compared to default Tmem, reflecting the increase in capacity to cache more pages. Our scheme effectively uses the Tmem by increasing the number of hits at the cache layers. We find a 4-12% increase in over-all hit rate with our policy compared to CLOCK-pro and MQ. We have a two-level cache layers that capture both the recency and frequency, which reduces the misses. The results indicate that our algorithm eliminates the cold pages from the cache faster than other policies. For example, a one-time access page in LRU scheme has to travel across the entire Tmem list before getting evicted to NVM. But in our scheme, pages which are referenced once are filtered quickly from the FIFO buffer. Our scheme thus prevents the cold pages from occupying page slots in the top-level cache. Our approach of cache design, reduces the disk I/O by 10–37%.

Hits on Tmem: In a layered cache it is important to have large number of hits at top level. This comparison is on the number of hits at Tmem which is the top level cache. Figure 6(c) shows the comparison of the hits at Tmem which illustrates that our scheme achieves about 25% more hits at Tmem than other schemes.

Writes to NVM: In two-level cache hierarchy, our scheme achieves higher number of hits at Tmem than at NVM. This results in fewer replacements from Tmem to NVM which reduces the writes to NVM. In Figure 6(d), our scheme achieves fewer writes to NVM compared to other traditional schemes. It saves 40% of writes at VM running Web-proxy and 28% at VM running Web Server compared to LRU. Our scheme provides better lifetime to write-limited NVM like PCM.

VI. Related Work

Transcendent memory was conceived and first implemented by Magenheimer [11]. Among Tmem's four possible combinations (private/shared x ephemeral/persistent), private Cleancache was the first to appear in releases of Xen and Linux. We extend Magenheimer's Transcendent Memory to NVM and propose a algorithm for managing the two-level cache.

Li et.al. [6] and Wong et.al. [17] have shown that eviction-based placement is more suitable for buffer cache, since it can provide a better hit rate. An eviction-based placement puts a block into the cache when it is evicted from the upper level cache (the page-cache in the guest OS in our case).

Chen et.al. [6] have showed that an eviction-based lower-level cache can provide higher hit ratio than access-based one. They also presented a tracking table based approach that can detect evictions at reuse time, without modifying client software. In a virtual machine environment, however, the guest OS does not usually use a dedicated buffer cache, making it difficult to precisely detect guest evictions.

LRU [7], [13] is a traditional replacement policy and widely used in cache management. CLOCK-pro[8] and MQ [18] are advanced replacement policies that perform better than LRU and other LRU enhancements like LFU, 2Q, LRU-K, EELRU, LRFU. Our results have shown Ex-Tmem outperforming all these algorithms.

Many cache replacement algorithms have been studied e.g., LRU-k [15], in the presence of concurrent workloads. LRU-k prevents useful buffer pages from being evicted due to sequential scans running concurrently. Ren et.al., [16] propose a 'Least Popularly Used' (LPU) algorithm that tracks disk blocks by recency of access. The Least Popularly Used queue is scanned after every 50,000 I/O requests to select reference blocks based on their popularities. This method incurs large overhead and makes hypervisor (KVM) complex and would suffer scalability issues.

VII. Conclusion

In this paper, we propose a mechanism to extend transcendent memory (called Ex-Tmem) with NVM so that more pages can be cached, instead of discarding them. Ex-Tmem puts clean pages in a two-level memory hierarchy considering access locality and different features of DRAM and NVM. With Ex-Tmem, memory-to-memory swapping is enabled and expensive swap I/O is totally removed. We have implemented our Ex-Tmem in the Xen hypervisor platform. Experimental results have shown that Ex-Tmem is efficient in improving performance and reducing disk I/O. In this virtualization system, the VMs share the resources DRAM and NVM. It is

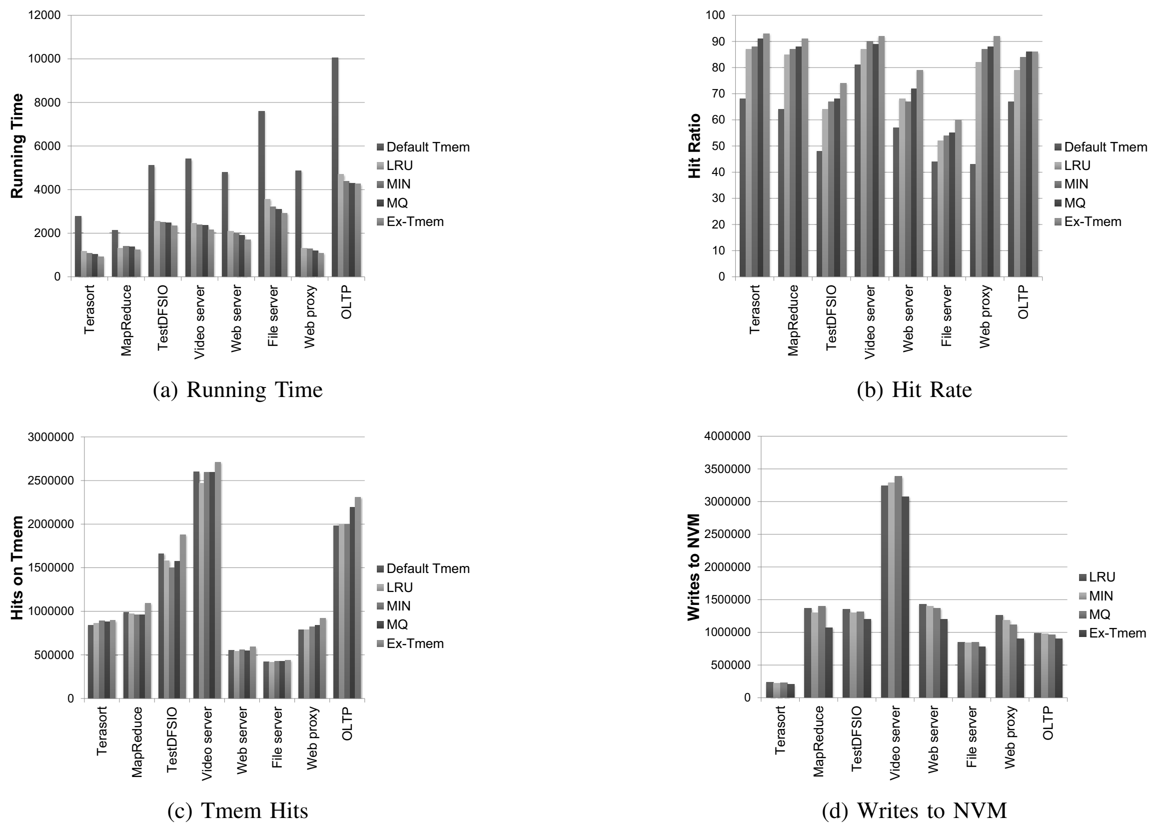


Fig. 6: Performance Evaluation

necessary to ensure performance isolation where one VM does not adversely affect another VM's performance. Our future work is to partition Tmem among VMs and provide a fair allocation of DRAM and NVM.

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their feedback and suggestions for improvement. This work was supported by Agency for Science, Technology and Research (A*STAR), Singapore under Grant No. 112-172-0010.

REFERENCES

[1] Cleancache. https://www.kernel.org/doc/Documentation/vm/cleancache.txt.

[2] Cleancache and frontswap. https://oss.oracle.com/projects/tmem/.

[3] Frontswap. https://www.kernel.org/doc/Documentation/vm/frontswap.txt.

[4] Nvdimm technology. http://www.vikingtechnology.com/nvdimm-technology.

[5] Rram technology overview. http://www.crossbar-inc.com/technology/resistive-ram-overview.html.

[6] Zhifeng Chen, Yuanyuan Zhou, and Kai Li. Eviction based cache placement for storage caches. In *USENIX Annual Technical Conference. Usenix*, 2003.

[7] Peter J. Denning. The working set model for program behavior. *Commun. ACM*, 11(5):323–333, May 1968.

[8] Song Jiang. Clock-pro: An effective improvement of the clock replacement. In *In Proceedings of USENIX Annual Technical Conference*, 2005.

[9] Takayuki Kawahara. Scalable spin-transfer torque ram technology for normally-off computing. In *IEEE Design and Test of Computers*, pages 52–63, 2011.

[10] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable dram alternative. In *ISCA*, pages 2–13, 2009.

[11] Dan Magenheimer. Transcendent memory on xen, 2009.

[12] Jack A. Mandelman, Robert H. Dennard, Gary B. Bronner, John K. DeBrosse, Rama Divakaruni, Yujun Li, and Carl J. Raden. Challenges and future directions for the scaling of dynamic random-access memory (dram). *IBM Journal of Research and Development*, 46(2-3):187–222, 2002.

[13] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[14] W. Mueller, G. Aichmayr, W. Bergner, E. Erben, T. Hecht, C. Kapteyn, A. Kersch, S. Kudelka, F. Lau, and J. Luetzen. Challenges for the dram cell scaling to 40nm. In *IEEE International Electron Devices Meeting*. IEEE, 2005.

[15] Elizabeth J. O'Neil, Patrick E. O'Neil, Gerhard Weikum, and ETH Zurich. The lru-k page replacement algorithm for database disk buffering. pages 297–306, 1993.

[16] Jin Ren and Qing Yang. A new buffer cache design exploiting both temporal and content localities. In *ICDCS*, pages 273–282. IEEE Computer Society, 2010.

[17] Theodore M. Wong and John Wilkes. My cache or yours? making storage more exclusive. In *In Proceedings of the 2002 USENIX Annual Technical Conference*, pages 161–175, 2002.

[18] Yuanyuan Zhou, James F. Philbin, and Kai Li. The multi-queue replacement algorithm for second level buffer caches. In *In Proceedings of the 2001 USENIX Annual Technical Conference*, pages 91–104, 2001.