Viyojit: Decoupling Battery and DRAM Capacities for Battery-Backed DRAM

Rajat Kateja
Carnegie Mellon University
rkateja@cmu.edu

Anirudh Badam
Microsoft
anirudh.badam@microsoft.com

Sriram Govindan
Microsoft
srgovin@microsoft.com

Bikash Sharma
Microsoft
bsharma@microsoft.com

Greg Ganger
Carnegie Mellon University
ganger@ece.cmu.edu

ABSTRACT

Non-Volatile Memories (NVMs) can significantly improve the performance of data-intensive applications. A popular form of NVM is Battery-backed DRAM, which is available and in use today with DRAMs latency and without the endurance problems of emerging NVM technologies. Modern servers can be provisioned with up-to 4 TB of DRAM, and provisioning battery backup to write out such large memories is hard because of the large battery sizes and the added hardware and cooling costs. We present Viyojit, a system that exploits the skew in write working sets of applications to provision substantially smaller batteries while still ensuring durability for the entire DRAM capacity. Viyojit achieves this by bounding the number of dirty pages in DRAM based on the provisioned battery capacity and proactively writing out infrequently written pages to an SSD. Even for write-heavy workloads with less skew than we observe in analysis of real data center traces, Viyojit reduces the required battery capacity to 11% of the original size, with a performance overhead of 7-25%. Thus, Viyojit frees battery-backed DRAM from stunted growth of battery capacities and enables servers with terabytes of battery-backed DRAM.

CCS CONCEPTS

• **Hardware** → *External storage*; *Batteries*; *Enterprise level and data centers power issues*;

KEYWORDS

NVM, Battery-backed DRAM

ACM Reference format:
Rajat Kateja, Anirudh Badam, Sriram Govindan, Bikash Sharma, and Greg Ganger. 2017. Viyojit: Decoupling Battery and DRAM Capacities for Battery-Backed DRAM. In *Proceedings of ISCA '17, Toronto, ON, Canada, June 24-28, 2017,* 14 pages.
https://doi.org/10.1145/3079856.3080236

1 INTRODUCTION

Availability of non-volatile memory (NVM) enables significant performance improvements in data-intensive systems. For example, its use as a cache in storage, file and database servers greatly reduces request response times as well as storage device traffic, which in turn reduces I/O system cost and wear (for Flash SSDs). As such, some NVM is used in most data centers and its use is growing.

Although truly NVM technologies, for e.g., 3D XPoint [8], are expected soon, the prevailing NVM approach is battery-backed DRAM (also referred to as Non-Volatile DRAM or NV-DRAM in short). [1] A server's DRAM can be made effectively non-volatile by provisioning sufficient battery capacity to ensure that the server can operate, after a power loss event, long enough to write out all of the DRAM contents to a non-volatile storage device (e.g., a Flash SSD). This approach has been used for storage systems, for many years, and is becoming common for general-purpose servers as more applications demand NVM performance [32].

Unfortunately, battery-backed DRAM creates major problems for data centers operators. While straightforward in theory, ever-increasing memory sizes make deploying battery power expensive in several ways. In advanced data center designs, the batteries are co-located with the servers, and their size and maintenance needs alone create issues. More importantly, batteries must be enclosed and cooled so as to address the thermal and safety hazards that they create, which becomes a major problem as their size grows (as well as a long-term sustainability problem). Moreover, their size grows rapidly, because battery energy density grows much more slowly than DRAM densities—for example, it has only increased three-folds in the last 25 years, during which DRAM density has grown by more than 50,000× (Fig. 1).

This paper describes Viyojit, which is a system to provide battery-backed DRAM at a small fraction of the battery requirements. The key insight is that skewed write patterns allow most pages to be kept in a clean state without excessive overhead. The NV-DRAM write working set is generally much smaller than the total NV-DRAM capacity. So, it is possible to decouple the amount of provisioned battery capacity from server DRAM capacity, addressing the size problems discussed above. Moreover, under skewed write patterns, the fraction of pages that are frequently updated naturally shrinks as the memory size grows, since more of the added capacity generally

[1]Even after such NVM technologies become available at commodity prices, some amount of battery-backed DRAM will be an important tool for hiding their higher expected latencies and addressing wear concerns associated with some of these technologies.

hold even less-frequently-written pages (Fig. 5). But, of course, it is not safe to simply assume that there will not be too many dirty pages—the system must ensure that there is no more data that needs to be written out than provisioned battery capacity can accommodate.

Viyojit provides an mmap-like API for explicit management of battery-backed DRAM. It adds support to track and explicitly bound the total amount of dirty data that must be written out upon power failure, safely presenting the illusion of full-sized battery-backed DRAM at a fraction of the battery needs. Whenever a new region is allocated in battery-backed DRAM, Viyojit write-protects the initialized pages. Whenever a page is written, a fault handler adds it to a list of dirty pages and, if necessary, writes out a page to keep the number of dirty pages below the upper bound. To avoid that case, of course, Viyojit writes out infrequently written pages in the background in order to maintain slack for absorbing write bursts. Write popularity is determined by periodically checking and clearing the page table dirty bits for known-to-be-dirty pages.

Experiments confirm both our insights that write skew should be expected and that Viyojit is able to efficiently bound battery requirements. Analysis of several real data access traces from Microsoft data-center workloads confirm the presence of both skewed write patterns and the expected large fraction of read-only data in NV-DRAM, beyond the DRAM capacity that does not need non-volatility (e.g., cached files and executable images). In almost all cases, battery would be needed for less than 15% of NV-DRAM allocated capacity, with proper management. Experiments with a version of the Redis [14] key-value store modified to use NVM and linked to Viyojit show that Viyojit performs well. Under various Yahoo! Cloud Serving Benchmark (YCSB) workloads, we find that efficient bounding of dirty NV-DRAM data is effective, even assuming much less skewed NV-DRAM write patterns than observed in the trace analyses. We vary the amount of provisioned battery assumed, and observe that end-to-end performance is reduced only by 7-25% when using a battery as little as 11% of the full-battery-backup case.

This paper makes three primary contributions. First, it introduces a solution to the battery scaling problem for battery-backed DRAM, based on explicit bounding of dirty NV-DRAM data and thereby enabling data center operators to provision much less battery capacity. Second, it describes Viyojit, a real implementation of this solution. Third, it presents analyses of real workload traces and results from real system experiments demonstrating the efficacy of Viyojit's approach to bounding battery requirements for data centers providing NVM via battery-backed DRAM today and in the future.

The rest of this paper is organized as follows. Section 2 presents the background on battery-backed DRAM and motivates the need for Viyojit. Section 3 presents an analysis of Microsoft data-center traces to quantify write skew. Sections 4 and 5 describe the design and implementation of Viyojit. Section 6 presents evaluation results. Section 7 discusses related work. Section 8 discusses additional benefits from using Viyojit and Section 9 concludes the paper.

2 BACKGROUND & MOTIVATION

Applications historically obtain persistence via SSDs or HDDs. These technologies require 10s of microseconds to a few milliseconds to ensure durability for data. Furthermore, these technologies can be addressed only at block granularities of 0.5KB (or multiples of it) at a time. This imposes strict lower bounds on the amount of data moved between memory and the durable medium, which necessitates the usage of software interfaces in the critical path that further increases the latency. Non-volatile main memories aim to solve both these problems by using materials that are not only fundamentally faster than NAND and magnetic drives but also are efficiently addressable at finer granularities, such as cache-lines. Therefore, a few CPU instructions – typically a store instruction followed by a CPU flush instruction – are enough for providing durability. Use of non-volatile memory can drastically reduce the latency of file systems [17, 27, 34, 46, 64], data bases [21, 36, 38], transactional memory systems [24, 26, 30, 58, 59], and their replicated versions [33, 41, 65].

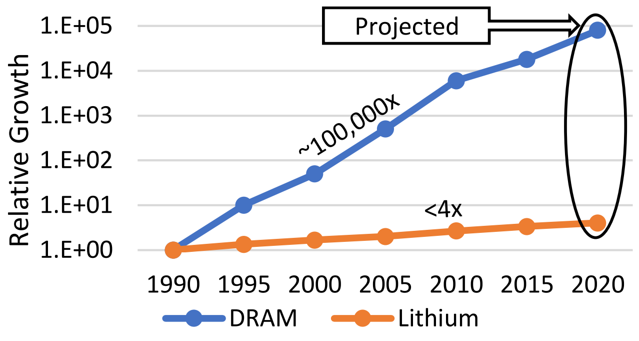


Figure 1: DRAM growth is out-pacing Lithium's. Battery density (joules per unit volume) has grown only by 3.3x in the last 25 years. In the same period, DRAM grew by more than four orders of magnitude (GB/RU). Today, it is already not practical to backup all the DRAM in large-memory systems that are fast becoming commodity.

2.1 Why Battery-Backed DRAM?

Unlike NVM technologies that rely on material science breakthroughs [8, 44], battery-backed DRAM (NV-DRAM) is a practical form of NVM available and in use today. For example, NV-DRAM DIMMs [1, 5, 11, 16, 20] typically contain some DRAM chips, a NAND chip, an ASIC or an FPGA controller, and an alternative power source such as a battery or a super-capacitor. Unfortunately, due to the non-commodity nature of such parts, the cost ($/GB) is quite high even when compared to DRAM. Apart from NV-DRAM DIMMs, high-end file servers, disk arrays, and database servers have used large, integrated batteries to allow their DRAM to be used as non-volatile write-back cache for disks.

Use of such large battery racks or rooms in data centers has become fairly common to survive transient power blips and allow controlled shutdowns of critical services. More recently, system designers have begun integrating commodity batteries into individual servers [10] and using custom firmware on commodity server boards to realize NV-DRAM with DRAM and SSDs [32]. The cost of such server-integrated NV-DRAM is low, because it is a small value-added feature on top of parts that a typical data center normally provisions.

Breakthrough NVM technologies such as 3D-XPoint [8] promise to scale beyond DRAM, consume less energy than DRAM, and provide two orders magnitude faster speeds compared to NAND. However, the solid state nature of such devices naturally implies wear issues and an asymmetry between reads and writes. Therefore, NV-DRAM offers an attractive trade-off of smaller capacity and higher performance without wear issues, even when the new technologies become available at commodity costs. Unfortunately, however, the scaling bottlenecks in batteries hinder system designers from backing up large amounts of DRAM.

2.2 Battery Scaling Challenges

Current NV-DRAM designs expect DRAM and battery to scale proportionately. But, in reality, battery scaling poses lot of challenges. First, battery real-estate is a key limiting factor for NV-DRAM in current data centers. Big data centers (including Microsoft [10] Google [4] and Facebook [6]) have already moved from centralized battery rooms (Lead-acid) to distributed server-level batteries (typically Li-ion) for several reasons, including lower cost, higher energy density, improved availability, modularity and serviceability.

Li-ion batteries, despite benefiting from massive economies of scale in the consumer electronics and electric vehicle markets, still struggle to keep pace with DRAM density scaling. For instance, Li-ion batteries have only tripled in density (joules per unit volume) in the last twenty-five years during which DRAM density has increased by four orders of magnitude. Figure 1 shows the disparity in battery and DRAM scaling over the last 25 years. The DRAM trend line tracks tracks the memory capacity of a typical high-end 1RU server while the battery trend line tracks the capacity (in joules per unit volume) of a typical phone-sized battery at that time. [2]. Indeed, the battery scaling issue is known to be a fundamental limitation associated with the battery chemistry [22], making future scaling prospects look bleak.

On the other hand, DRAM has enjoyed significant scaling due to engineering advancements in cooling, 3D stacking, and trading performance and reliability for higher capacity. As a concrete example, we quantify this challenge using a prototype one rack-unit server in our lab, which holds 4 TB DRAM ($32\times$128GB DDR4 LRDIMMs [15]). Assuming 4 GBPS SSD write bandwidth and a modest 300W server power, one would require a battery with ˜300KJ of energy for backing up the entire 4 TB DRAM, which is about 10x the volume of a typical smart phone battery (2000mAh).

Moreover, battery usage needs and reliability characteristics cause available battery capacity to vary over time due to variations in external power fluctuations, aging, ambient temperature and humidity variation, depth of discharge, etc. As a consequence, battery capacity can also vary dynamically. Further, additional battery cells are typically provisioned for redundancy and for other battery use-cases, such as peak-shaving [37, 42, 66], brief power blips and brownouts [62], and for addressing power/energy mismatch with newer power supplies like fuel cells [47] and renewable energy. Therefore, ensuring sufficient battery for safe NV-DRAM support requires over-provisioning for the peak need and worst-case conditions, adding another multiplier to the battery size.

Note that this is still a conservative estimate since data center batteries typically use ˜30% less dense battery material [22] to support higher power levels compared to consumer electronics. Increasing battery energy density typically has thermal implications, especially within servers that house high wattage components like CPUs. Datacenters already spend lot of energy and cost on removing heat from servers, which will be exacerbated with additional battery cells, which heat up during charges/discharges [35]. Further, it poses reliability challenges for both the battery cells themselves and for other server components.

Additionally, high-power density batteries such as the ones used in the data center are usually not discharged below 50% to make them last for 3–4 years. We assume as depth of discharge of 50% on batteries to ensure a four year lifetime [37, 43] which leaves effective capacity to be halved.

Realistically, a volume of over 25x the size of a smart-phone battery (i.e. the real-estate equivalent of 10–15 smart phones) is needed for each data center server to account for these factors. Conversations with data center engineers indicate that making that much space is an unrealistic expectation. Even more so when the DRAM growth further out-paces Lithium growth in the next several years.

Lastly, batteries are not cheap. Using our estimates, each server's battery may cost over 250$ while accounting for lithium, packaging, safety and charging circuitry, and maintenance overheads resulting in several million dollars increase in capital expenditure per data center. Battery disposal and carbon footprint costs are additional.

Although NV-DRAM offers significant performance advantages, its success will depend on adapting to battery scaling challenges. Therefore, we ask the following question: *Can we design an NV-DRAM solution where battery capacity need not scale with DRAM capacity?*

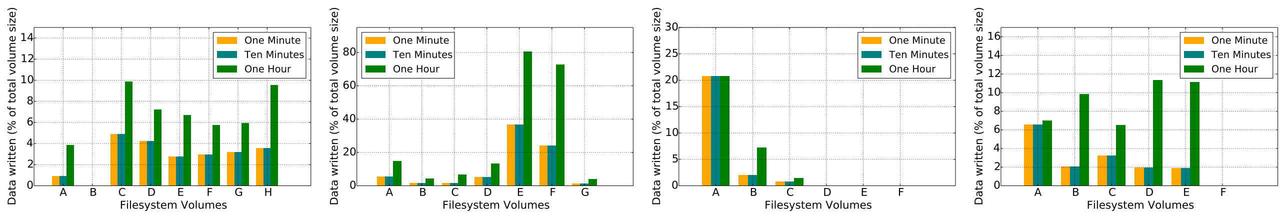
3 EXPLOITING ACCESS SKEW TO REDUCE BATTERY CAPACITY

Reducing the battery demand of NV-DRAM requires reducing the duration of time that operation must continue, after a power failure, in order to flush NV-DRAM contents. An important observation in this regard is that only "dirty" pages—pages that are not already present on the backing device—need to be flushed from DRAM to the backing device upon a power outage.

A key workload characteristic for enabling reduced battery demand is skew in the write access pattern. Most data access patterns are skewed, such that some portions of the data are used more than others, which is a key building block of caches and efficiency for the virtual memory system. We expect (and find) that this expected behavior exists when focusing only on update accesses as well: some pages are updated frequently, but most are not. If pages that are not updated frequently and recently are proactively written out to the backing store, by a background process, only frequently updated pages will need to be written out in the event of external power failure. Hence, for workloads with skewed access patterns, battery capacity corresponding to only a fraction of the total NV-DRAM capacity would suffice. The inherent write skew in real applications and the benefits thereof are key insights behind Viyojit.

To quantitatively support our claim of skewed write patterns, we present an analysis of real workload traces from a production datacenter at Microsoft. We procured and analyzed file system level traces for four sizable applications running in the data center:

[2]Collected from various sources including but not limited to [18], [69], [52], [49] and newegg.com.



(a) Azure blob storage (b) Cosmos (c) Page rank (d) Search index serving

Figure 2: Data written to as a fraction of the total volume size for different interval lengths. For a majority of the scenarios, the fraction of data written is less than 15%.

- **Azure blob storage [3]**: Online data blob store similar to Amazon S3 storage [2].
- **Cosmos [25]**: Map-Reduce [31] like framework for serving massive data-parallel workloads.
- **Page rank**: Algorithm for building a search index.
- **Search index serving**: System used for answering search queries from users.

For each application, except for Cosmos, the trace spans a continuous duration of 24 hours. For Cosmos, the trace corresponds to a non-contiguous duration of 3.5 hours. For each application, the trace was collected on a single machine serving that application. Each machine, however, had multiple file system volumes.

The file system volumes were originally hosted on hard disks and SSDs in production. We consider the scenario in which all volumes on a machine are instead hosted on NV-DRAM. As discussed in Section 2, servers with 4 TB/RU of DRAM are available today, and larger capacity severs are expected soon. Hence, it is realistic to assume that a handful of file system volumes amounting to a few hundred gigabytes can be kept entirely in NV-DRAM.

The workload traces are file system level traces that contain information about reads and writes in files and do not provide any information about the actual pages touched in the NV-DRAM, which is determined by the underlying file system. In order to be conservative, we begin by assuming an adversarial scenario in which all application writes always go to a unique page in the NV-DRAM file system. Log-structured file systems [55] are examples of file systems that would result in such behavior.

With the above assumptions, the first metric we look at is the (worst case) amount of data written in a given duration of time as a fraction of the total file system volume size. We slice the entire trace duration into multiple intervals of (say) ten minutes each and determine the amount of data written during each interval in the trace by treating each NV-DRAM page write as a write to an unique NV-DRAM page. Among all the intervals within a trace, we further consider the worst interval, i.e. the interval with the maximum amount of data written across all intervals. Fig. 2 presents the results for all four workloads, along with the results for interval durations of one minute and one hour.

For a majority of the workloads, the fraction of data written is always less than 15%. This suggests that even in the worst case in which each write to a file is considered as a write to a unique NV-DRAM page, only around 15% of the total NV-DRAM file system would be written to within an hour in most cases. This data confirms our expectation of write skew in real applications.

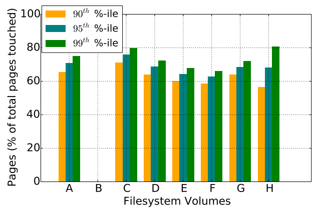
The analysis suggests that decoupling the battery and NV-DRAM capacity is not just viable, but rather an appealing option. State-of-the-art NV-DRAM systems would require a battery capacity corresponding to the entire NV-DRAM file system volume. However, our analysis shows that battery capacity corresponding to merely 15% of the total NV-DRAM file system volume capacity would be more than sufficient for a majority of the applications.

So far, the analysis shows that writes, as a whole, are a small fraction of total volume size in general. Next, we further quantify the skew within the writes. To that end, we look at the distribution of writes across the pages in a file system volume. For this analysis, we are not concerned with the actual pages written to in the NV-DRAM. Rather, we focus on the logical pages in the file system volumes.

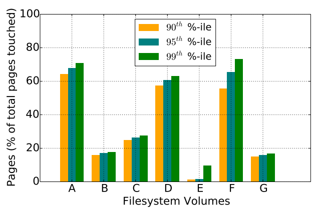
To identify the write skew within a volume, we do the following analysis. We first count the number of writes to logical pages in a given volume. Next, we count the number of pages that contribute to (say) 90% of the total writes on that particular volume during the entire trace duration. Finally, we divide this number of pages by the total number of pages touched (read or written) during the entire trace. Note that the total number of pages touched is not equal to, and always less than, the total volume size. We repeat the analysis for 95% and 99% of the total writes and present the results in Fig. 3.

The analysis brings out some interesting patterns. First, for most of the volumes across different applications, when the volumes have a low fraction of writes to begin with (e.g., volume A for Azure blob storage), the fraction of pages required to account for 90% or 99% of the total writes is quite high. This means that while the total number of writes are small, the writes happen to mostly unique pages. Second, for certain volumes, such as volumes B and C in from the Cosmos application, while the total number of writes are small to begin with, those writes are even further skewed since the fraction of pages required to account for 99% of the total writes is just ≈30%. Third, for volumes such as volume F in Cosmos, while the total fraction of writes is close to 70%, the writes are highly skewed with roughly 10% of the pages accounting for 99% of the writes. Finally, there are also volumes, such as volume E in Cosmos where the total fraction of writes is high and the writes are to mostly unique pages.

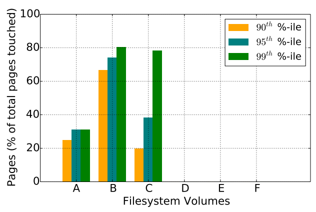
Based on the above classification, reducing battery capacity is most appealing for the second category of workloads with low fraction of writes that are even further skewed. Similarly, the first and third category of workloads would also benefit from reducing battery capacity because of the write skew. For the fourth category of workloads, however, reducing battery capacity might not be worthwhile



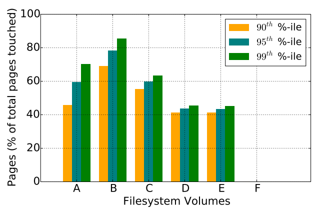
(a) Azure blob storage



(b) Cosmos

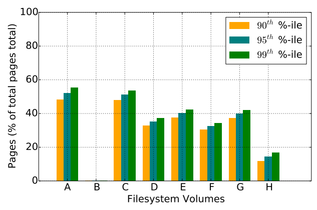


(c) Page rank

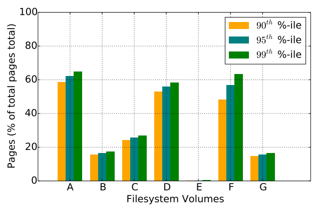


(d) Search index serving

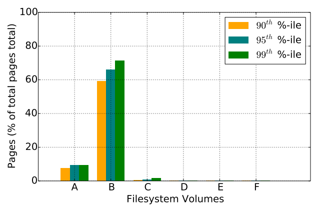
Figure 3: Number of pages required as a percentage of the total pages touched to account for different percentiles of the total writes.



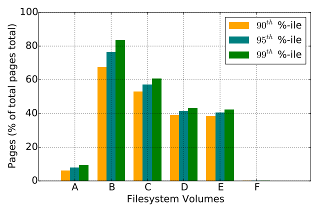
(a) Azure blob storage



(b) Cosmos



(c) Page rank



(d) Search index serving

Figure 4: Number of pages required as a percentage of the total pages total to account for different percentiles of the total writes.

since such workloads are likely to require a battery capacity corresponding to a relatively high fraction of the total DRAM capacity, thereby reducing the benefits of such a scheme. Fig. 4 presents the same results but represents the pages as percentages of the total pages in the volume as opposed to the touched pages. As expected, the percentages are lower than the case when the number of pages required are expressed as a percentage of the touched pages. The trends, however, are similar to the previous results, which leads to a similar classification and interpretation. To sum, the analysis suggests that there is a significant write skew under certain workloads. This presents an opportunity to reduce the amount of battery capacity required and address the battery scaling challenges, contingent on the workload.

While the actual skew in different workloads vary, access skew is typically described by rules of thumb like the 80/20 rule and the Zipf distribution. It is worth noting that for workloads that confirm to such skewed access patterns, the fraction of pages used at a given percentile of writes, say 90%, decreases proportionately as the total number of pages increase. We show this pattern using synthetically generated Zipf distribution in Fig. 5. This fact offers hope that the fraction of NV-DRAM that would be frequently updated will decrease as the NV-DRAM size increases, thereby making battery and DRAM capacity decoupling even more appealing.

Viyojit is designed to exploit write access skew, as is seen in the above analyses, to reduce battery requirements for NV-DRAM. It enforces an upper bound on the number of dirty pages in NV-DRAM, thereby allowing a cap on the battery requirement, but suffers little penalty because most pages are updated infrequently. By decoupling the battery and DRAM capacities, Viyojit would enable NV-DRAM capacity to scaled with DRAM as opposed to being bounded by battery scaling.

4 DESIGN CONSIDERATIONS

This section presents design goals for Viyojit, which implements a NV-DRAM solution to decouple the battery capacity from offered

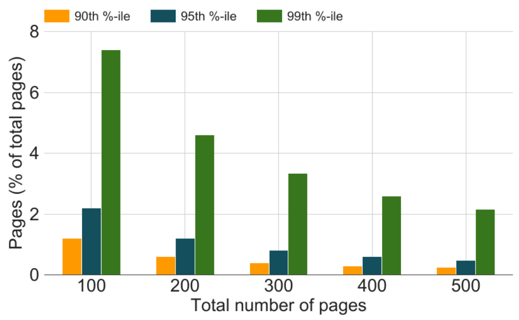


Figure 5: Number of pages as a fraction of the total pages required to account for different percentiles of writes under Zipf write distribution. The fraction of the pages required decreases as the total number of pages increase.

DRAM capacity so as to address the battery scalability concerns discussed in section 2.

As described earlier, state-of-the-art NV-DRAM solutions require battery capacity to be tightly coupled to the NV-DRAM[3] capacity since battery is provisioned to handle the worst case scenario where the entire DRAM has to be backed up on a power failure. The goal of *Viyojit* is to achieve de-coupling of battery and DRAM capacity by restricting battery usage to a pre-defined backup energy budget (say in Joules converted to a certain number of DRAM pages), while allowing NV-DRAM capacity to be scaled almost independently.

Viyojit's first and foremost design goal is *durability*. Even if the provisioned or allocated battery budget is insufficient to back up the entire NV-DRAM, all data in NV-DRAM should survive arbitrary power cycle events. While *durability* is a strict guarantee from Viyojit, other desirable but not necessary design goals include *performability* and *portability*. *Performability* ensures that the overall

[3]We refer to the DRAM portion intended to be non-volatile as NV-DRAM. We expect systems with both volatile (working memory) and non-volatile DRAM (persistent storage) regions, although the ideal capacity split between the two is orthogonal and beyond the scope of this paper.

system performance is not adversely affected in comparison to baseline NV-DRAM performance with full battery capacity. *Portability* ensures that Viyojit works with un-modified applications and does not require significant changes in existing systems. We now discuss these design goals and how Viyojit achieves the same in detail.

4.1 Durability

Battery energy is utilized in the event of a power outage to write out the data from the NV-DRAM onto persistent storage. Given a reasonable power model of system components such as the CPU, DRAM and persistent medium and a conservative bandwidth estimate between DRAM and persistent medium, the amount of battery backup energy required is directly proportional to the amount of data which needs to be written out on a power failure. Viyojit uses this relationship to compute a *dirty budget* – the maximum amount of data which can remain dirty in NV-DRAM (inconsistent with the persistent medium) at any point of time, for a given battery budget. Viyojit ensures *durability* by restricting the amount of dirty NV-DRAM data within the *dirty budget* at all times.

A seemingly plausible approach to enforce the *dirty budget* is to periodically count the number of dirty pages in NV-DRAM and perform flushes to persistent medium when the dirty count approaches the dirty budget. This, however, may not guarantee *durability* since the number of dirty pages in NV-DRAM could exceed the dirty budget in between two successive dirty count checks.

Viyojit is able to strictly enforce the *dirty budget* by tracking and having a synchronous view of the state of all DRAM data, whether dirty or not, at any given point of time. Viyojit tracks the dirty data in NV-DRAM at a page granularity. Since Viyojit is only concerned with the state of the page, whether dirty or clean, it is sufficient for Viyojit to have information about only the first write to a NV-DRAM page since the same page written repeatedly will only result in a single write flush to the persistent storage. To enforce *durability* using the dirty budget, Viyojit keeps a running count of the number of dirty pages in NV-DRAM along with their addresses. This counter is incremented whenever a page is dirtied (i.e. written for the first time) and decremented when a page is copied to persistent storage. Viyojit then ensures that this counter stays below the dirty budget at all points in time, as described in Section 5.

4.2 Performability

For Viyojit to be practically viable, it is not sufficient to just ensure *durability*. It is also important for Viyojit to not drastically affect the application performance because of the overhead of managing and restricting dirty data. There are two key aspects on which the *performability* of Viyojit depends: how much data needs to be frequently flushed and when is it flushed. If a large fraction of the total NV-DRAM capacity is frequently dirtied and needs to be flushed, then the overall system throughput would be determined by the bandwidth to the persistent storage, which is orders of magnitude worse than the DRAM bandwidth. Secondly, even with a small fraction of dirty data, a delayed flush of a dirty page can cause a NV-DRAM write could potentially block on a write to the persistent storage leading to a high performance overhead. Particularly, if the number of dirty pages in the system is equal to the dirty budget, then every write which dirties a new page would block on a write to the persistent storage, severely impacting the latency as well as the overall system throughput.

With regards to the fraction of data frequently updated, we make the observation, supported by our analysis of Microsoft data-center application traces as presented in Section 3, that most application writes are skewed. That is, only a small portion of the entire data set is dirtied over and over again with a long tail of infrequent writes. Given this skew, battery capacity corresponding to only a small fraction of the total NV-DRAM capacity is sufficient to ensure that Viyojit's throughput is not bottlenecked on the bandwidth to the persistent medium.

With regards to avoiding NV-DRAM writes blocking on writes to the persistent storage, Viyojit proactively copies dirty pages to the persistent medium and avoids reaching a state where the number of dirty pages in NV-DRAM is exactly equal to the dirty budget. Such proactive copying is analogous to the virtual memory system, with the physical memory size being analogous to the dirty budget, and swapping out a page from physical memory proactively due to memory pressure being analogous to copying out a page and removing it from the dirty page set. However, unlike virtual memory, Viyojit keeps a copy of the page in NV-DRAM even after it is written to persistent storage which provides DRAM latency reads to all the pages in NV-DRAM.

4.3 Portability

Widespread adoption of Viyojit depends on its ability to easily applicable to existing systems and applications. NV-DRAM from Viyojit must appear to the rest of the system exactly like any other NVM. Further, any server with a reasonable number of SSD program erase cycles must be able to benefit from Viyojit's battery reduction without overwhelming the SSD with write traffic generated due to dirty budgeting.

For *portability*, we implemented Viyojit as a shared library, that applications can link against and use the exposed mmap-like API. Since the API is similar to mmap, with a mmap and munmap function, NV-DRAM with under-provisioned battery appears to the rest of the system just as if it were NV-DRAM with full battery provisioning. Further, our design reserves the provisioned battery for the most frequently written pages of NV-DRAM while the less frequently written pages are copied to the SSD. By its very nature, this design minimize the SSD write traffic.

5 IMPLEMENTATION

We implement Viyojit using software management of x86_64 page tables. Viyojit exposes an API similar to the mmap API for allocating and accessing NV-DRAM. There are three key components in our implementation of Viyojit:

- Identifying and enforcing the dirty budget by tracking dirty pages.
- Choosing the target pages to be copied by maintaining the least recently updated list.
- Proactively copying pages based on the dirty page pressure.

We now describe each of these components in the context of our software system. Towards the end of this section, we discuss an alternative hardware-assisted implementation which requires modifications to the Memory Management Unit (MMU) along with the OS virtual memory manager. Throughout this section, we assume that a server has been provisioned with some fixed battery capacity, potentially determined using an analysis of the expected workloads similar to the one with Section 3, that is a fraction of the total NV-DRAM capacity.

5.1 Identifying and enforcing the dirty budget

Given a provisioned battery capacity, the dirty budget corresponding to the same is computed as follows. Using the peak power usage of different system components (CPU, DRAM, SSD, etc), we determine the amount of time the provisioned battery can support the entire system. Multiplying this time with a conservative estimate of the SSD write bandwidth gives the dirty budget. Note that the need for a conservative estimate of the SSD write bandwidth and the peak power usage of system components is not unique to Viyojit. Even traditional NV-DRAM systems need to estimate the above in order to provision battery corresponding to the entire NV-DRAM capacity. Viyojit on the other hand identifies the dirty budget for a given battery capacity using these estimates.

Having identified the dirty budget, Viyojit enforces the same by tracking and limiting the dirty pages in NV-DRAM. Figure 6 presents a flowchart of our software implementation. During startup, before exposing the NV-DRAM as non-volatile to any application, we use the MMU to write protect all the NV-DRAM pages (step 1). Upon a write to a NV-DRAM page (step 2), the MMU raises an write-protection violation interrupt which is handled by our system (step 3). The interrupt handler determines the address of the faulting page which is about to be written. We increment the NV-DRAM dirty pages count and update the list of all the dirty page addresses (step 4). We check the count of dirty pages in NV-DRAM and compare it to the dirty budget (step 5).

If the dirty count is lower than the budget, we immediately unprotect the page to allow all subsequent writes to this page go through in an unhindered manner. If the dirty count is equal to the budget, a target page is chosen based on a least recently updated policy implementation described below. We write protect the target page again (step 6) and write it to secondary storage (step 7). Following that, we remove the target page from the dirty page list. Write protecting the page before writing it to secondary storage is important because of the following reason. Pages present in the dirty page list are not write protected so as to allow future writes to a once-written page to proceed without requiring an interrupt. Under this setting, if a concurrent thread were to update the page content in NV-DRAM while the page was being written to secondary storage, marking the page clean upon completion of the secondary storage write would lead to an incorrect state and eventual data loss. Hence, we write protect the page in NV-DRAM before writing it to secondary storage.

Note that steps 6 and 7 described above are optional and occur only when the dirty budget is reached. In both the cases, either after performing steps 6 and 7, or directly from step 5, the system reaches step 8 where the write protection from the faulting page is removed, the dirty page count is incremented and the faulting page address is added to the dirty page list. After removing that write protection

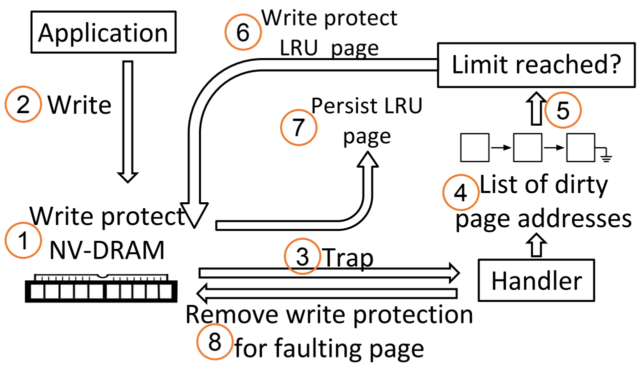


Figure 6: Flow chart describing Viyojit's implementation for tracking dirty pages and enforcing the dirty budget.

from the faulting page, the handler returns. The MMU then retries the original write which now completes successfully.

We implemented a kernel module to manipulate the page table entries and set or reset the write protection bits. Note that whenever the write protection of a page is changed, the TLB entry corresponding to the page needs to be removed. This is required to ensure that the next access to the page is a TLB miss and the updated page table entry corresponding to the page is read from memory. The TLB misses are an additional overhead over and above the overhead of the traps caused due to write protection. We quantify the effects of all these overheads later in this section 6.

5.2 Choosing target pages

As discussed in section 4.3, it is important to write out the pages which are not written frequently. We describe Viyojit's target page selection policy and rationale by making an analogy to the well understood virtual memory subsystem.

In order to identify targets for pages to copy out, Viyojit uses a least recently updated policy which is similar to the Least Recently Used (LRU) policy used by the virtual memory subsystem to choose targets for swapping out pages. While the virtual memory subsystem is interested in swapping out the pages which are least useful in terms of future reads as well as writes, we are interested in copying out pages which are least useful for future writes, since Viyojit always has a (potentially write protected, but never read protected) copy of all the pages in NV-DRAM. Hence, instead of accounting for read as well as write page references, and employing a least recently *used* policy, Viyojit accounts only for write page references and uses a least recently *updated* policy.

We use the x86_64 dirty bit to track page updates. We use an epoch based mechanism wherein we check and clear the dirty bit of all the pages in NV-DRAM on every epoch boundary by doing a page table walk. If the dirty bit for a page is found to be set while doing a page table walk at a given epoch boundary, it can be inferred that the page was updated during the last epoch. Using these periodic walks, Viyojit stores a history of the last 64 epochs for all the pages. Based on these histories, Viyojit sorts the pages according to update times and chooses the least recently updated pages as targets. It is important to note here that resetting dirty bits is not safe in general because that is the only information available to the virtual memory manager to infer the state of any page. However, since Viyojit keeps the list of dirty page addresses separately and does not rely on the virtual memory manager for ensuring durability, it is safe to reset dirty bits.

One interesting aspect of maintaining the least recently updated list is that reading up-to-date dirty bit information from the page table requires all the TLB entries to be flushed. If the TLB entries were not flushed, Viyojit may read stale dirty bit values, which may result in flushing frequently updated pages (as opposed to least updated ones). Although TLB flushes and subsequent TLB misses are additional sources of overhead [4], we empirically found that the benefit of reading updated dirty bits, and being able to identify better target pages, far outweighs the cost of TLB misses.

5.3 Proactive copying

The next implementation detail concerns with the number of pages to keep in dirty state. As discussed in Section 4.2, for *performability*, the system needs to start copying pages to secondary storage proactively in order to avoid NV-DRAM writes blocking on writes to secondary storage. We are then faced with the following question: what should be the dirty pages threshold for triggering flushes to persistent medium? If the threshold is very close to the dirty budget, a burst of new dirty pages would cause high write latencies. On the other hand, if the threshold is too low, Viyojit would unnecessarily copy data to secondary storage which may result in undesirable IO bandwidth contention Moreover, if the secondary storage device suffers from write wear (such as SSDs), aggressive copying would adversely affect the lifetime of the secondary storage device, which goes against the goal of *portability*, as discussed in section 4.3.

We use an online algorithm to tune our threshold of dirty pages at runtime. As mentioned above, Viyojit uses an epoch based approach to identify target pages. Viyojit also counts the number number of new dirty pages in each epoch. This does not lead to any additional overhead because Viyojit already performs a page table walk to check and reset the dirty bits. Given the count of new dirty pages in each epoch, Viyojit use an exponentially decaying average to predict the number of new dirty pages expected in the next epoch, which we refer to as the *dirty page pressure*. Viyojit assigns a weight of 0.75 to the number of dirty pages in the current epoch and a weight of $(1-0.75=)0.25$, to the previously predicted value to estimate the dirty page pressure. Finally, Viyojit sets the threshold as the dirty budget minus the dirty page pressure. By doing so, Viyojit tries to ensure that the system would be able to absorb all these new dirty pages without reaching the dirty budget.

5.4 Alternative Implementation: Offloading to the MMU

We implemented the system as described above in Linux as a shared library with 1,500 lines of code. The implementation inherently incurs the trap overhead for the first write to a page, which although minimized by the optimizations discussed, cannot be avoided altogether. This leaves the desire for offloading some components to MMU in the future. More specifically, we are exploring how to design a MMU that can enforce dirty limits with little overhead.

[4]Flushing all the TLB entries takes ˜3.5 ms on our development machine with Intel Nehalem Class Core i7 processor [7] and 16 GB DRAM. Setting and clearing the write protection bits also take roughly 3ms each.

Existing MMUs already set the dirty bit whenever a page is written. In order to count the number of dirty pages, the MMU can additionally check the dirty bit before setting it and increment a counter if the dirty bit was not already set. Next, the MMU could raise an interrupt whenever the number of dirty pages reach the OS specified dirty budget or threshold described above, so as to facilitate proactive copying. The OS could then handle such interrupts in a manner similar to Viyojit.

In order to safely implement a target selection policy such as the least recently updated policy, which relies on resetting the dirty bit in page table entries, the MMU could further support a shadow dirty bit. The shadow dirty bit could be set whenever the dirty bit is set. The OS could then read and reset this shadow bit to track recency, similar to Viyojit.

6 EVALUATION

We now present the experiments and results of our evaluation of Viyojit. The goal of our evaluation is to show that a significant reduction of the battery capacity is possible while still providing a comparable overall system performance and to show that proactively flushing to an SSD is an acceptable trade-off with respect to wear. Combined, these two make decoupling battery and NV-DRAM capacities a viable option with minimal resource overheads.

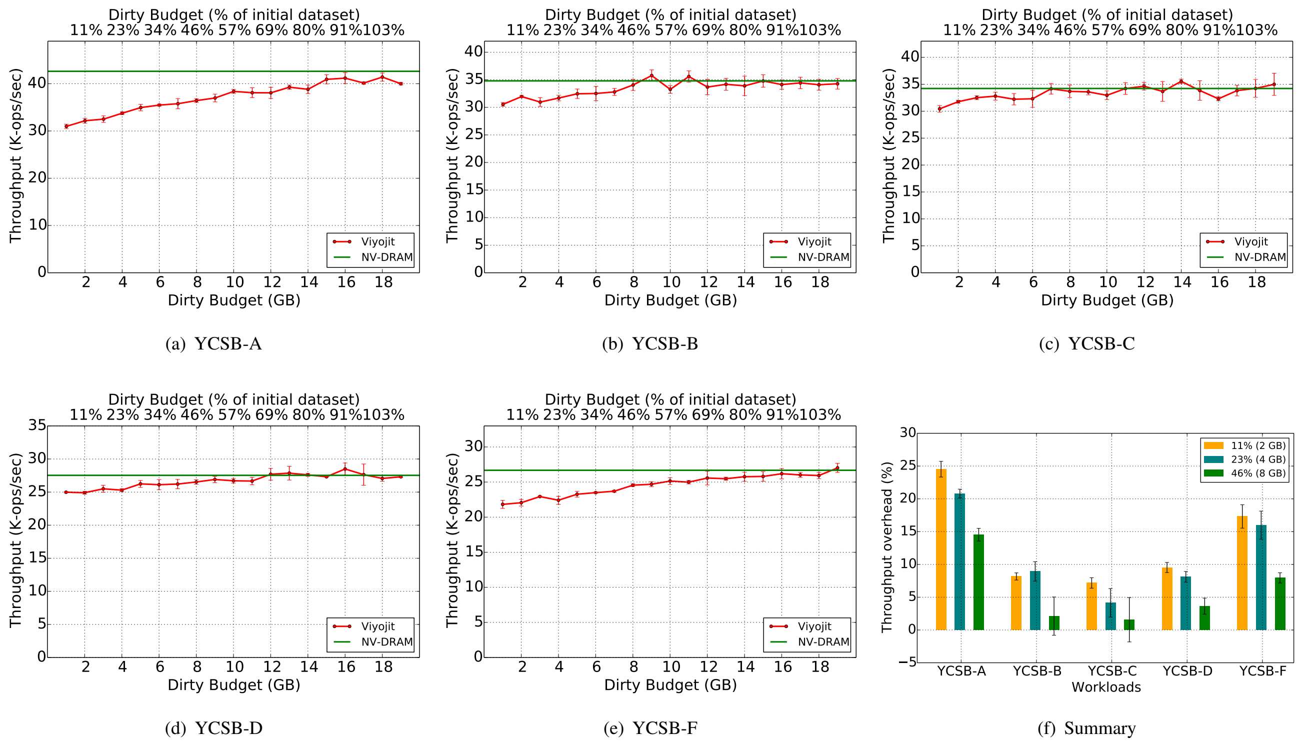
6.1 Experimental Setup

We performed out experiments on Microsoft Azure [9] Virtual Machines (VMs). We allocated VMs with 140 GB DRAM, 20 cores and 280 GB SSD. We ran our experiments on Ubuntu 16.04 with 4.4.0-45-generic Linux kernel. The SSD supported a maximum throughput of 625 K-IOPS. From the 140 GB DRAM, we emulated a region of 60 GB to be NV-DRAM.

In all our evaluations, we use the dirty budget, as described in section 4, as a proxy for the battery capacity. As discussed earlier, in a system where different components have fixed power usage and the main memory and secondary storage have constant read and write bandwidths, battery capacity is directly proportional to the dirty budget. Both these requirements are reasonable to assume from any given system in steady state. We begin by evaluating the impact of dirty limiting on the throughput and latency of cloud applications built for NV-DRAM.

We evaluate Viyojit with NV-DRAM being used as a persistent heap by an in-memory key-value store, Redis [14] modified to persistently store all of its data (and metadata) in NV-DRAM. Redis, being volatile in nature by default, is typically used as a front end cache by databases for serving data quickly from memory. However, after a power cycle, either planned or unplanned, Redis loses all of its data and has to start as a cold cache. The non-volatility of NV-DRAM can help Redis start as a warm cache which would improve the performance of the back-end database.

We modified Redis to store it's key-value pairs along with the associated metadata in a non-volatile heap using Intel's Persistent Memory Library (PMEM) [13]. We then emulate a region of DRAM as NV-DRAM by using Linux persistent memory emulation [12]. For the baseline, we consider the case where the battery capacity provisioned is sufficient to back up the entire NV-DRAM, i.e. 60



(a) YCSB-A (b) YCSB-B (c) YCSB-C

(d) YCSB-D (e) YCSB-F (f) Summary

Figure 7: Throughput for different YCSB workloads. The overhead of Viyojit for battery capacity corresponding to 11% of the NV-DRAM capacity varies from 7% for to 25%. The values on the top y-axis represents the dirty budget as a percentage of the initial heap size.

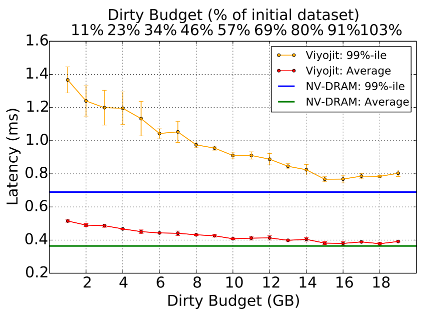
GB. For Viyojit, we vary the battery capacity by varying the dirty budget.

We use the following representative cloud benchmarks from the Yahoo! Cloud Serving Benchmark (YCSB) [28] suite as the workloads.

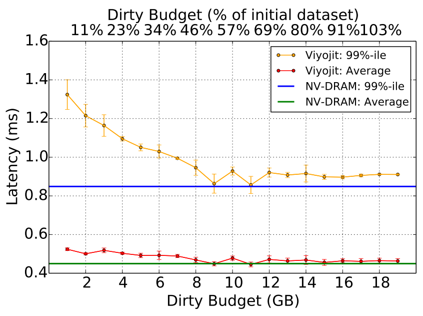
- YCSB-A: update heavy workload with 50% reads and 50% updates. This usage pattern represents interactive applications that create new content rapidly.
- YCSB-B: read mostly workload with 95% reads and 5% updates. This usage pattern represents document serving applications where documents are accessed frequently but edited rarely.
- YCSB-C: read only workload with 100% reads. This usage pattern represents image serving front-end servers that act as a cache for serving images to the wide area. Note that while the application is read-only, internally, Redis still performs several store instructions as part of the internal logic for metadata operations.
- YCSB-D: read latest workload 95% reads and 5% inserts with newly inserted records read with higher probability. This usage pattern represents social media posts where there are a few updates that are read by a vast number of users.
- YCSB-F: read-modify-write workload with 50% reads and 50% read-modify-writes. This usage pattern represents record stores in user record data bases that are read and modified.

We could not run YCSB-E because it requires cross key transactions which we do not support for now. We wish to add this to our NV-DRAM based Redis in the future. For each of the workloads, we initially created a database for which a 17.5 GB heap was allocated in NV-DRAM. In the results and discussion that follows, we describe the dirty budget as a fraction of this initial heap size. For e.g. we treat 2 GB dirty budget as 11% battery capacity. This is because a baseline system would guarantee durability of the entire Redis heap with a battery capacity corresponding to only 17.5 GB. The actual NV-DRAM capacity of 60 GB is arbitrary and hence is not used for computing the fraction of battery capacity. Note that for YCSB-D which consists of insert operations in addition to read operations, the total Redis heap and hence the required battery capacity for baseline, is more actually than the initial heap size of 17.5 GB. However, even for YCSB-D, we use the initial heap size of 17.5 GB for computing the battery fraction percentages so as to maintain uniformity across our results and discussions.

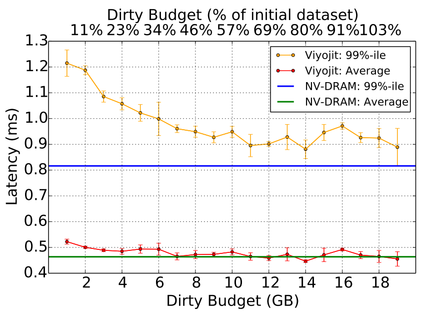
We configured all the benchmarks to perform 10 million operations. Unless mentioned otherwise, each data point is averaged over three runs and the error bars represent the root mean square error. We present the results with our system configured have no more than 16 outstanding IO requests at any point of time and an epoch duration of 1 ms. We experimented with other values for both of these parameters and the results were similar, hence we do not present them here.



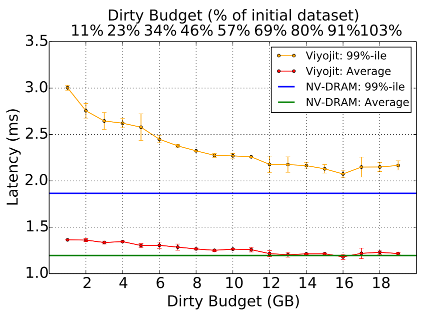
(a) YCSB-A update latency



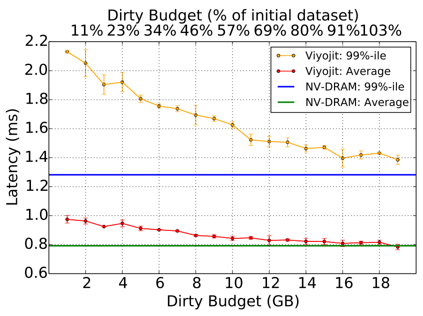
(b) YCSB-B update latency



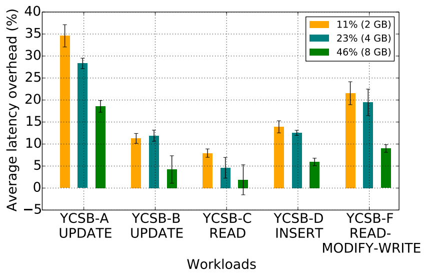
(c) YCSB-C read latency



(d) YCSB-D insert latency



(e) YCSB-F read-modify-write latency



(f) Summary

Figure 8: Average and 99th %-ile latencies for different operations in YCSB workloads. While the tail latencies are always higher than the baseline, the average latencies are close to the baseline for large enough battery capacity fractions. The high tail latencies are due to the write traps required to track dirty pages.

6.2 Results

We vary the dirty budget from 1GB to 19GB and measure the throughput of the workloads. Fig. 7 presents the results for the different workloads as well as a summary of all the workloads for select battery capacities.[5]

For read heavy workloads, i.e. YCSB-B and YCSB-C, with even as little as 11% battery which corresponds to a 2 GB dirty budget, the loss in throughput is only 8% and 7% respectively. For YCSB-C, with approximately 45% battery capacity, which corresponds to a dirty budget of 8 GB, the throughput is roughly the same as that of the baseline. On the other hand, for workloads with significant write operations, such as YCSB-A, YCSB-D and YCSB-F, the overhead of restricting battery sizes are more severe. With 11% battery, corresponding to 2 GB dirty budget, the throughput decreases by 25%, 10% and 17% compared to the baseline for YCSB-A, YCSB-D and YCSB-F respectively.

Next, we look at the average and 99th percentile latency for the different workloads. Since different workloads perform different operations, we focus on the latency of different operations for each workload. For YCSB-A and YCSB-B, we look at the update latency, whereas for YCSB-C, YCSB-D and YCSB-F, we look at the read, insert and read-modify-write latency respectively. Note that write, insert and read-modify-write operations are more prone to overheads arising from Viyojit because of the write traps caused by the writes performed by these operations. Hence, for workloads YCSB-A, YCSB-B, YCSB-D and YCSB-F, which consist of read operations as well, we are presenting results for the more conservative operations so as to highlight the overheads due to Viyojit. Since YCSB-C is a read-only workload, we present the read latency results for YCSB-C. The results are shown in Fig. 8.

Across all the battery capacities, the 99th percentile latency with Viyojit is always slightly higher than the baseline, even when the battery capacity is higher than the Redis heap size. This is because Viyojit is affected only by the total NV-DRAM capacity and not by the heap sizes of individual applications. Hence, the write protection is always needed for correctness and that affects the tail latency by generating write faults. The fact that the experimental workloads happen to be the only applications using the NV-DRAM, which makes it safe to turn of write protection once the dirty budget exceeds the heap size, is an artifact of our experimental setting and we do not modify Viyojit to treat this as a special case.

The average latencies, on the other hand, are close to the baseline latencies for large dirty budgets. Once the dirty budget is large enough, for example 12 GB for YCSB-A and 8 GB for YCSB-C, majority of the pages being written to are already maintained in the dirty state and hence the average latency is close to the baseline latency.

6.3 Discussion

The overheads observed arise because of three factors: traps on first write to write protected pages, NV-DRAM writes being throttled

[5] The NV-DRAM throughput presented is also an average of three runs and has some variance (around 2% for YCSB-B and less than 1% for all others), but this variance is not presented in the graphs for clarity.

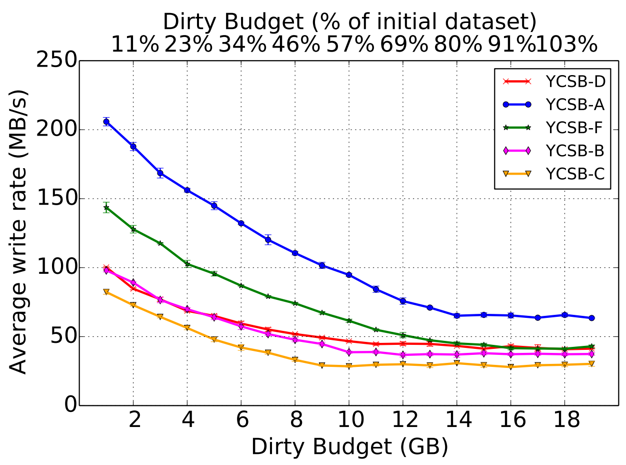


Figure 9: Average write rate during the duration of the experiments. Even the highest write rate of 200 MB/s corresponding to the write heavy workload YCSB-A with ˜11% battery capacity can be easily sustained by modern SSDs.

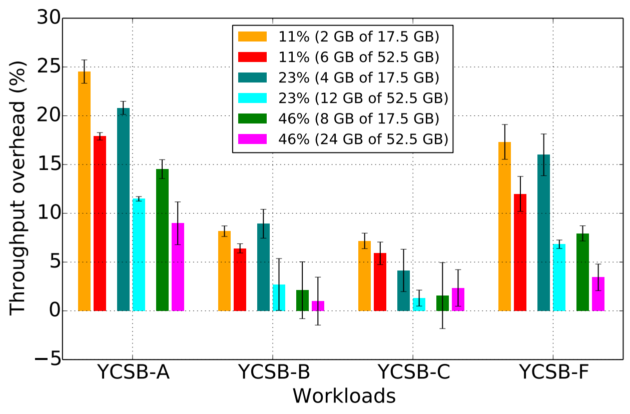


Figure 10: Performance overheads for different battery capacity fractions for different YCSB workloads across two initial heap sizes: 17.5 GB and 52.5 GB. The overheads decrease with an increase in the heap size which confirms our hypothesis that with increasing dataset sizes, the write skew becomes more severe.

by writes to the SSD and maintaining the least recently updated list which requires TLB flushes as discussed in section 5.2. The first two sources affect write heavy workloads more severely than read heavy workloads which leads to larger drop in throughput of write heavy workloads. It should be noted here that even the read only workload YCSB-C is impacted by these overheads because the workload is read-only from the application perspective, but consists of metadata updates performed by Redis.

We found the overhead imposed by TLB flushes in order to maintain the least recently updated list well worth the benefits. We performed experiments in which we turned off the TLB flushing which lead to reading stale dirty bit information from the page tables. The impact of having an imprecise least recently updated list caused the throughput to drop by more than half in cases with low battery provisioning such as with 2 or 3 GB dirty budget.

While the performance impact from reducing the battery capacity is highly workload dependent, it is acceptable across the board. Write heavy workloads require a battery capacity which corresponds to more than 50% of their total NV-DRAM heap allocations in order to achieve performance within 10% of the baseline. Read heavy workloads can achieve the same performance with approximately 11% battery capacity. Under-provisioning the battery capacity, however, has a consistent effect on the tail latency. This is implementation dependent and we believe that a hardware implementation as described in section 5.4 could eradicate such tail latency overheads.

The results make evident an interesting trade-off between battery capacity and performance. For example, write heavy workloads can either opt for a significant reduction in battery capacity of up-to 10x at the cost of 20-30% performance overhead or they can opt for a lower reduction in battery capacity of 2x with a lower performance overhead.

We envision Viyojit being used by cloud providers in order to reduce their battery capacity requirements and we make a case for such cloud providers to treat battery as a first class resource, much like DRAM itself. In such a setting, tenants can buy battery capacity based on their expected workload and required performance. Further, cloud providers can employ techniques similar to memory ballooning [60], to reallocate battery/dirty-budget among co-located tenants to benefit from inherent statistical multiplexing effects.

In order to quantify the effect on secondary storage bandwidth due to Viyojit, we measured the amount of data copied out of NV-DRAM during the duration of the workload and computed the average write rate by dividing it by the workload duration. We present the results for the same in Fig. 9. Note that the write rate computed also consists of the writing out the entire heap at the end of the experiment. These writes, at the end of the experiments, would be required by the baseline system as well. Further, YCSB benchmarks correspond to sustained high bandwidth applications whereas real applications are often bursty. Nonetheless, for a reasonable experimental setting of ours with 10 million operations, even the highest write rate of roughly 200 MB/s can be easily sustained by current SSDs.

To sum, the results show that decoupling the battery and NV-DRAM capacity is indeed a viable option. Doing so allows under-provisioning the battery capacity which would enable the amount of battery-backed DRAM per server to scale at the rate of DRAM capacities per server rather than being bounded by the rate of Lithium scaling.

6.4 Experiments with larger heap size

For the results presented in the above sections, the allocated heap was of a relatively small size of 17.5 GB. However, NV-DRAM is typically present at a much larger scale in real systems and we expect the write skew to be more apparent as the total NV-DRAM capacity grows, as discussed in Section 3.

To quantitatively support our argument, we present results from experiments with an initial heap size of 52.5 GB, i.e. 3x the original data-set size, for the same benchmarks. In Fig. 10, we compare the throughput overhead for YCSB-A, YCSB-B, YCSB-C and YCSB-F between the two data-set sizes. We could not perform experiments with larger heap size for YCSB-D because the heap for YCSB-D at the end of the benchmark, which performs insert operations thereby increasing the heap size further from 52.5 GB, exceeds the total NV-DRAM capacity of 60 GB. We present the results for battery capacities corresponding to 11%, 23% and 46%.[6]

The results clearly show that the overhead decreases with an increase in the heap size, thereby confirming our intuition of increasing skew with increased heap sizes. This makes Viyojit even more appealing for high capacity NV-DRAM systems.

7 RELATED WORK

Battery-backed DRAM: Batteries have been used as a backup power source to ensure durability of data in DRAM for a long time [29, 48, 54, 61]. More recently, the massive economies of scale in battery sales for consumer electronics has significantly reduced the price of battery backup [10, 32]. Based on this trend and the low cost of NAND, several vendors have designed Non-Volatile DIMMs that ensure durability of DRAM data when connected to a battery [1, 5, 11, 16, 20].

Unfortunately, however, these parts are specially designed for enterprise class servers and do not enjoy the massive economies of scale typically expected of data center parts. Therefore, system designers have proposed using commodity data center DRAM DIMMs, SSDs and batteries to implement battery-backed DRAM [32]. However, such systems are faced with the uphill challenge of bridging the gap between DRAM and Lithium capacities inside servers. Viyojit bridges this gap by decoupling these two entities by limiting the dirty data in DRAM.

Limiting dirty data: Prior works have proposed limiting dirty data in processor caches to improve cache performance and simplify the implementation of other cache optimizations [56, 57]. Our work is closest in spirit to these works, but focussed on limiting dirty data in main memory as opposed to processor caches. To the best of our knowledge, ours is the first work which proposes a mechanism for limiting dirty data in main memory.

Viyojit can also perform dirty tracking and limiting at a finer byte-level granularity using Mondrian Memory Protection [63], using the same dirty budgeting mechanism as proposed in this paper. This would not only enable better utilization of provisioned battery capacity but also reduce the write traffic to secondary storage. The write bandwidth to secondary storage could be further reduced by using compression and de-duplication [50, 68].

Memory management techniques: Existing memory managers [19] keep an approximate count of the dirty data in memory, but do not have any synchronous visibility in to the pages being dirtied. Mechanisms proposed in this paper are required to accurately track the dirty data in memory. Identifying target pages to swap out in face of memory pressure is a challenge common to all memory managers and also to Viyojit. Multiple target selection policies have been proposed in literature [23, 39, 40, 45, 51, 53, 67] for the same. Viyojit use a target selection policy similar to the LRU, which is one of the most widely used and policies, to copy pages to secondary storage.

[6] Note that the dirty budget for the same battery capacity fraction would be different between the two data-set sizes. For example, the dirty budget corresponding to 11% battery capacity for the 17.5 GB case is 2 GB, while the corresponding dirty budget for 52.5 GB case is 5 GB.

8 ADDITIONAL BENEFITS WITH VIYOJIT

This section outlines two additional benefits of decoupling battery capacity from DRAM capacity: reduction of reboot times and ability to deal with battery capacity changes.

Increased availability: As the amount of DRAM increases in a system with a given PCIe bandwidth, the time needed for backing up the entire DRAM to an SSD on a power outage increases. For instance, writing out 4 TB of DRAM onto an SSD at 4 GBPS flush rate requires close to 17 minutes. Reloading the DRAM contents takes a similar amount of time. This means that, in the worst case (i.e., if the entire DRAM needs to be written out), a reboot would require 17 minutes to shutdown followed by 17 minutes to start up again. The start up time can be optimized by fetching pages from SSD to DRAM on demand while sequentially reading data in the background after the OS boots. Unfortunately, shutdown has no such respite. Bounding the number of dirty pages in DRAM in turn bounds the flush time during shutdown, allowing Viyojit to reduce shutdown times, which in turn leads to higher availability.

Handling battery cell failures: Batteries, much like SSDs, wear out over time and lose capacity [22]. The capacity can also fluctuate based on the surrounding environment, such as the ambient temperature. Over-provisioning can help alleviate problems related to wear out and fluctuation, but it leads to increased costs. The infrastructure to limit the dirty data by the battery capacity enables tuning of the dirty budget at runtime according to changes in battery capacity. Such a design allows a server to deal with uncertainty better than by simply stopping operation when battery availability drops by more than over-provisioning.

9 CONCLUSION

Viyojit significantly reduces the battery needed to back up DRAM being used as NVM, by decoupling battery required from DRAM capacity. It exploits the write skew inherent in real-world applications to efficiently bound the number of dirty NVM pages based on the battery capacity provisioned. Viyojit tracks all dirty pages, identifies the least recently updated pages, and proactively copies some of them to the backing SSD to free up battery capacity for more frequently updated pages. Analyses of file system traces of Microsoft services show that most real applications update less than 15% of the total dataset over long durations of time. Experiments with a cloud key-value store using Viyojit show that all of the allocated heap can be treated as non-volatile, while using a small fraction of the battery capacity normally required, with little performance overhead. For example, with a battery capacity of only ~11%, the key-value store throughput reduces by 7% for read heavy workloads and 25% for write heavy workloads. By decoupling the battery and DRAM capacities, Viyojit enables servers to scale to terabytes of NV-DRAM.

Acknowledgments: We thank the anonymous reviewers for their feedback and suggestions. We thank the members and companies of the PDL Consortium (Broadcom, Citadel, Dell EMC, Google, HP Labs, Hitachi, Intel, Microsoft Research, MongoDB, NetApp, Oracle, Samsung, Seagate Technology, Tintri, Two Sigma, Toshiba, Uber, Veritas and Western Digital) for their interest, insights, feedback, and support.

REFERENCES

[1] *AGIGARAM Non-Volatile System*. http://www.agigatech.com/agigaram.php.

[2] *Amazon S3: Amazon Simple Storage Servive*. https://aws.amazon.com/s3/.

[3] *Azure Blob Storage*. https://azure.microsoft.com/en-us/services/storage/blobs/.

[4] *Google's One Battery Per Server Design*. https://gigaom.com/2009/04/01/a-key-to-googles-data-center-efficiency-one-backup-battery-per-server/.

[5] *HPE Persistent Memory*. https://www.hpe.com/us/en/servers/persistent-memory.html.

[6] *Inside the Open Compute Project Server at Facebook*. https://www.facebook.com/notes/facebook-engineering/inside-the-open-compute-project-server/10150144796738920/.

[7] *Intel Nehalem Class Core i7 Processor*. http://ark.intel.com/products/37150/Intel-Core-i7-950-Processor-8M-Cache-3_06-GHz-4_80-GTs-Intel-QPI.

[8] *Intel Optane/Micron 3d-XPoint Memory*. http://www.intel.com/content/www/us/en/architecture-and-technology/non-volatile-memory.html.

[9] *Microsoft Azure*. http://azure.microsoft.com.

[10] *Microsoft Reinvents Datacenter Power Backup with Lithium-ion Batteries*. https://blogs.technet.microsoft.com/hybridcloud/2015/03/10/microsoft-reinvents-datacenter-power-backup-with-new-open-compute-project-specification/.

[11] *NETLIST's NVvault NVDIMM-N*. http://www.netlist.com/products/vault-memory-storage/nvvault-ddr4-nvdimm/default.aspx.

[12] *Persistent Memory Emulation*. http://pmem.io/2016/02/22/pm-emulation.html.

[13] *PMEM: Intel persistent memory library*. http://pmem.io.

[14] *Redis: in-memory key value store*. http://redis.io/.

[15] *Samsung 128 GB DIMM*. https://news.samsung.com/global/samsung-starts-mass-producing-industrys-first-128-gigabyte-ddr4-modules-for-enterprise-servers.

[16] *SMART's NVDIMM*. http://www.smartm.com/products/dram/NVDIMM_products.asp.

[17] *System Support for NVMs in Linux*. http://nvdimm.wiki.kernel.org.

[18] *Tesla Future of Transportation Session Presentation*. https://www.slideshare.net/vvfvilar/jb-16-x9-tesla-vail-global-energy.

[19] *Understanding the Linux Virtual Memory Manager*. https://www.kernel.org/doc/gorman/pdf/understand.pdf.

[20] *Viking Non-Volatile Memory*. http://www.vikingtechnology.com/nvdimm-technology.

[21] Joy Arulraj, Andrew Pavlo, and Subramanya R. Dulloor. 2015. Let's Talk About Storage & Recovery Methods for Non-Volatile Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, New York, NY, USA, 707–722. https://doi.org/10.1145/2723372.2749441

[22] Anirudh Badam, Ranveer Chandra, Jon Dutra, Anthony Ferrese, Steve Hodges, Pan Hu, Julia Meinershagen, Thomas Moscibroda, Bodhi Priyantha, and Evangelia Skiani. 2015. Software Defined Batteries. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 215–229. https://doi.org/10.1145/2815400.2815429

[23] Sorav Bansal and Dharmendra S Modha. 2004. CAR: Clock with Adaptive Replacement. In *Proceedings of the FAST '04 Conference on File and Storage Technologies, March 31 - April 2, 2004, Grand Hyatt Hotel, San Francisco, California, USA*, Vol. 4. 187–200.

[24] Bill Bridge. 2015. NVM support for C applications. (2015). Available at http://www.snia.org/sites/default/files/BillBridgeNVMSummit2015Slides.pdf.

[25] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. 2008. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. *Proc. VLDB Endow.* 1, 2 (Aug. 2008), 1265–1276. https://doi.org/10.14778/1454159.1454166

[26] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 105–118. https://doi.org/10.1145/1950365.1950380

[27] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O Through Byte-addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 133–146. https://doi.org/10.1145/1629575.1629589

[28] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*. ACM, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[29] G. Copeland, T. Keller, R. Krishnamurthy, and M. Smith. 1989. The Case for Safe RAM. In *Proceedings of the 15th International Conference on Very Large Data Bases (VLDB '89)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 327–335. http://dl.acm.org/citation.cfm?id=88830.88887

[30] Intel Corporation. 2015. Persistent Memory Programming. (2015). http://pmem.io/nvml/

[31] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. https://doi.org/10.1145/1327452.1327492

[32] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15')*. ACM, New York, NY, USA, 54–70. https://doi.org/10.1145/2815400.2815425

[33] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15')*. ACM, New York, NY, USA, 54–70. https://doi.org/10.1145/2815400.2815425

[34] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*. ACM, New York, NY, USA, Article 15, 15 pages. https://doi.org/10.1145/2592798.2592814

[35] Ozan Erdinc, Bulent Vural, and Mehmet Uzunoglu. 2009. A dynamic lithium-ion battery model considering the effects of temperature and capacity fading. In *Clean Electrical Power, 2009 International Conference on*. 383–386. https://doi.org/10.1109/ICCEP.2009.5212025

[36] Ru Fang, Hui-I Hsiao, Bin He, C. Mohan, and Yun Wang. 2011. High Performance Database Logging Using Storage Class Memory. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering (ICDE '11)*. IEEE Computer Society, Washington, DC, USA, 1221–1231. https://doi.org/10.1109/ICDE.2011.5767918

[37] Sriram Govindan, Anand Sivasubramaniam, and Bhuvan Urgaonkar. 2011. Benefits and Limitations of Tapping into Stored Energy for Datacenters. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA '11)*. ACM, New York, NY, USA, 341–352. https://doi.org/10.1145/2000064.2000105

[38] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. 2014. NVRAM-aware Logging in Transaction Systems. *Proc. VLDB Endow.* 8, 4 (Dec. 2014), 389–400. https://doi.org/10.14778/2735496.2735502

[39] Song Jiang and Xiaodong Zhang. 2002. LIRS: An Efficient Low Inter-reference Recency Set Replacement Policy to Improve Buffer Cache Performance. In *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '02)*. ACM, New York, NY, USA, 31–42. https://doi.org/10.1145/511334.511340

[40] Theodore Johnson and Dennis Shasha. 1994. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB '94)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 439–450. http://dl.acm.org/citation.cfm?id=645920.672996

[41] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.* 185–201. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/kalia

[42] Vasileios Kontorinis, Liuyi Eric Zhang, Baris Aksanli, Jack Sampson, Houman Homayoun, Eddie Pettis, Dean M Tullsen, and Tajana Simunic Rosing. 2012. Managing distributed ups energy for effective power capping in data centers. In *Computer Architecture (ISCA), 2012 39th Annual International Symposium on*. IEEE, 488–499.

[43] Vasileios Kontorinis, Liuyi Eric Zhang, Baris Aksanli, Jack Sampson, Houman Homayoun, Eddie Pettis, Dean M. Tullsen, and Tajana Simunic Rosing. 2012. Managing Distributed Ups Energy for Effective Power Capping in Data Centers. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA '12)*. IEEE Computer Society, Washington, DC, USA, 488–499. http://dl.acm.org/citation.cfm?id=2337159.2337216

[44] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting Phase Change Memory As a Scalable Dram Alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*. ACM, New York, NY, USA, 2–13. https://doi.org/10.1145/1555754.1555758

[45] Donghee Lee, Jongmoo Choi, John-Hun Kim, Sam H. Noh, Sang L. Min, Yoonkun Cho, and Chong S. Kim. 2001. LRFU: A Spectrum of Policies That Subsumes the Least Recently Used and Least Frequently Used Policies. *IEEE Trans. Comput.* 50, 12 (Dec. 2001), 1352–1361. https://doi.org/10.1109/TC.2001.970573

[46] Eunji Lee, Hyokyung Bahn, and Sam H. Noh. 2013. Unioning of the buffer cache and journaling layers with non-volatile memory. In *Proceedings of the 11th USENIX conference on File and Storage Technologies, FAST 2013, San Jose, CA, USA, February 12-15, 2013*. 73–80. https://www.usenix.org/conference/fast13/technical-sessions/presentation/lee

[47] Yang Li, Di Wang, Saugata Ghose, Jie Liu, Sriram Govindan, Sean James, Eric Peterson, John Siegler, Rachata Ausavarungnirun, and Onur Mutlu. 2016. SizeCap:

Efficiently handling power surges in fuel cell powered data centers. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 444–456. https://doi.org/10.1109/HPCA.2016.7446085

[48] David E. Lowell and Peter M. Chen. 1997. Free Transactions with Rio Vista. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP '97)*. ACM, New York, NY, USA, 92–101. https://doi.org/10.1145/268998.266665

[49] Chris A. Mack. 2011. Fifty Years of Moore's Law. *IEEE Transactions on Semiconductor Manufacturing* 24, 2 (May 2011), 202–207. https://doi.org/10.1109/TSM.2010.2096437

[50] Thanos Makatos, Yannis Klonatos, Manolis Marazakis, Michail D. Flouris, and Angelos Bilas. 2010. Using Transparent Compression to Improve SSD-based I/O Caches. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys '10)*. ACM, New York, NY, USA, 1–14. https://doi.org/10.1145/1755913.1755915

[51] Nimrod Megiddo and Dharmendra S. Modha. 2003. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Proceedings of the FAST '03 Conference on File and Storage Technologies, March 31 - April 2, 2003, Cathedral Hill Hotel, San Francisco, California, USA*. http://www.usenix.org/events/fast03/tech/megiddo.html

[52] Ethan Mollick. 2006. Establishing Moore's Law. *IEEE Annals of the History of Computing* 28, 3 (July 2006), 62–75. https://doi.org/10.1109/MAHC.2006.45

[53] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. 1993. The LRU-K Page Replacement Algorithm for Database Disk Buffering. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD '93)*. ACM, New York, NY, USA, 297–306. https://doi.org/10.1145/170035.170081

[54] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. 2011. Fast Crash Recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. ACM, New York, NY, USA, 29–41. https://doi.org/10.1145/2043556.2043560

[55] Mendel Rosenblum and John K. Ousterhout. 1992. The Design and Implementation of a Log-structured File System. *ACM Trans. Comput. Syst.* 10, 1 (Feb. 1992), 26–52. https://doi.org/10.1145/146941.146943

[56] Vivek Seshadri, Abhishek Bhowmick, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. 2014. The Dirty-block Index. *SIGARCH Comput. Archit. News* 42, 3 (June 2014), 157–168. https://doi.org/10.1145/2678373.2665697

[57] Jaewoong Sim, Gabriel H. Loh, Hyesoon Kim, Mike O'Connor, and Mithuna Thottethodi. 2012. A Mostly-Clean DRAM Cache for Effective Hit Speculation and Self-Balancing Dispatch. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. 247–257. https://doi.org/10.1109/MICRO.2012.31

[58] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. 2011. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In *9th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, February 15-17, 2011*. 61–75. http://www.usenix.org/events/fast11/tech/techAbstracts.html#Venkataraman

[59] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 91–104. https://doi.org/10.1145/1950365.1950379

[60] Carl A. Waldspurger. 2002. Memory Resource Management in VMware ESX Server. *SIGOPS Oper. Syst. Rev.* 36, SI (Dec. 2002), 181–194. https://doi.org/10.1145/844128.844146

[61] An-I Wang, Peter L. Reiher, Gerald J. Popek, and Geoffrey H. Kuenning. 2002. Conquest: Better Performance Through a Disk/Persistent-RAM Hybrid File System. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference, June 10-15, 2002, Monterey, California, USA*. 15–28. http://www.usenix.org/publications/library/proceedings/usenix02/wang.html

[62] Di Wang, Sriram Govindan, Anand Sivasubramaniam, Aman Kansal, Jie Liu, and Badriddine Khessib. 2014. Underprovisioning Backup Power Infrastructure for Datacenters. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, New York, NY, USA, 177–192. https://doi.org/10.1145/2541940.2541966

[63] Emmett Witchel, Josh Cates, and Krste Asanović. 2002. Mondrian Memory Protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*. ACM, New York, NY, USA, 304–316. https://doi.org/10.1145/605397.605429

[64] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *14th USENIX Conference on File and Storage Technologies, FAST 2016, Santa Clara, CA, USA, February 22-25, 2016*. 323–338. https://www.usenix.org/conference/fast16/technical-sessions/presentation/xu

[65] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. 2015. Mojim: A Reliable and Highly-Available Non-Volatile Memory System. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, New York, NY, USA, 3–18. https://doi.org/10.1145/2694344.2694370

[66] Wenli Zheng, Kai Ma, and Xiaorui Wang. 2014. Exploiting thermal energy storage to reduce data center capital and operating expenses. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 132–141. https://doi.org/10.1109/HPCA.2014.6835924

[67] Yuanyuan Zhou, James Philbin, and Kai Li. 2001. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference, June 25-30, 2001, Boston, Massachusetts, USA*. 91–104. http://www.usenix.org/publications/library/proceedings/usenix01/zhou.html

[68] Benjamin Zhu, Kai Li, and Hugo Patterson. 2008. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In *6th USENIX Conference on File and Storage Technologies, FAST 2008, February 26-29, 2008, San Jose, CA, USA*. 269–282. http://www.usenix.org/events/fast08/tech/zhu.html

[69] Chen-Xi Zu and Hong Li. 2011. Thermodynamic analysis on energy densities of batteries. *Energy and Environmental Science* 4 (2011), 2614–2624. Issue 8. https://doi.org/10.1039/C0EE00777C